# DESIGNING ETL PIPELINES WITH


STRUCTURED STREAMING

DELTA LAKE

*How to architect things right*

Tathagata "*TD*" Das

@tathadas

Spark Summit Europe
16 October 2019

databricks

# About Me

Started Apache Spark Streaming project in AMPLab, UC Berkeley

Currently focused on Structured Streaming and Delta Lake

Staff Engineer on the StreamTeam @ databricks

*Team Motto: "We make all your streams come true"*

# APACHE Spark Structured Streaming

## Distributed stream processing built on SQL engine

- High throughput, second-scale latencies
- Fault-tolerant, exactly-once
- Great set of connectors

## Philosophy: Treat data streams like unbounded tables

- Users write batch-like queries on tables
- Spark will continuously execute the queries incrementally on streams

# Apache Spark Structured Streaming

## Example

Read JSON data from Kafka

Parse nested JSON

Store in structured Parquet table

Get end-to-end failure guarantees


```
spark.readStream.format("kafka")
      .option("kafka.boostrap.servers",...)
      .option("subscribe", "topic")
      .load()
      .selectExpr("cast (value as string) as json")
      .select(from_json("json", schema).as("data"))
      .writeStream
      .format("parquet")
      .option("path", "/parquetTable/"
      .trigger("1 minute")
      .option("checkpointLocation", "…")
      .start()
```

# Anatomy of a Streaming Query

```
spark.readStream.format("kafka")
      .option("kafka.boostrap.servers",...)
      .option("subscribe", "topic")
      .load()
      .selectExpr("cast (value as string) as json")
      .select(from_json("json", schema).as("data"))
      .writeStream
      .format("parquet")
      .option("path", "/parquetTable/"
      .trigger("1 minute")
      .option("checkpointLocation", "…")
      .start()
```

Specify where to read data from

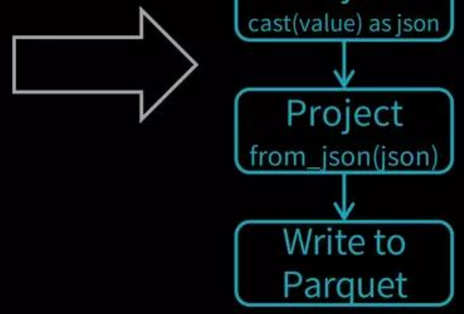
Specify data transformations

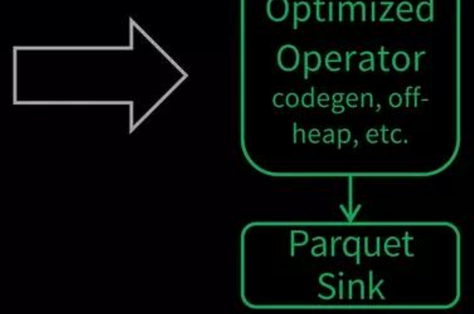
Specify where to write data to

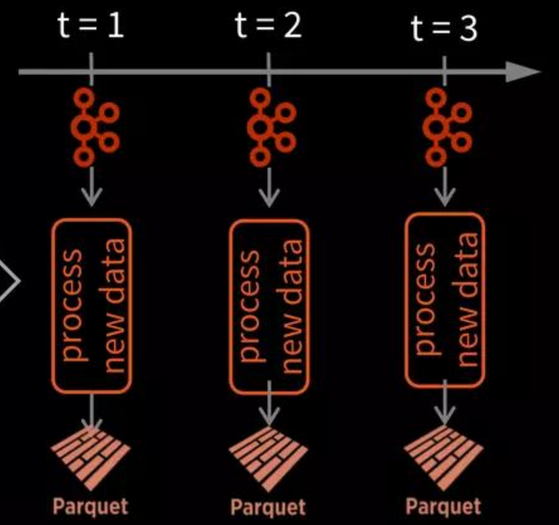
Specify how to process data

# Spark automatically streamifies!

```
spark.readStream.format("kafka")
      .option("kafka.boostrap.servers",...)
      .option("subscribe", "topic")
      .load()
      .selectExpr("cast (value as string) as json")
      .select(from_json("json", schema).as("data"))
      .writeStream
      .format("parquet")
      .option("path", "/parquetTable/"
      .trigger("1 minute")
      .option("checkpointLocation", "…")
      .start()
```




DataFrames, Datasets, SQL

Logical Plan

Optimized Plan

Series of Incremental Execution Plans

Spark SQL converts batch-like query to a series of incremental execution plans operating on new batches of data

# DELTA LAKE

Open-source storage layer that brings ACID transactions to Apache Spark™ and big data workloads.

- ACID transactions
- Schema Enforcement and Evolution
- Data versioning and Audit History
- Time travel to old versions
- Open formats
- Scalable Metadata Handling
- Great with batch + streaming
- Upserts and Deletes

https://delta.io/

# DELTA LAKE

Open-source storage layer that brings ACID transactions to Apache Spark™ and big data workloads.

### THE GOOD OF DATA WAREHOUSES

- Pristine Data
- Transactional Reliability
- Fast SQL Queries

### THE GOOD OF DATA LAKES

- Massive scale out
- Open Formats
- Mixed workloads

https://delta.io/


STRUCTURED STREAMING

DELTA LAKE

# How to build streaming data pipelines with them?


STRUCTURED STREAMING

DELTA LAKE

What are the *design patterns*
to correctly architect
streaming data pipelines?

# *Another streaming design pattern talk????*

**Most talks**

Focus on a pure streaming engine

Explain one way of achieving the end goal

**This talk**

Spark is more than a streaming engine

Spark has multiple ways of achieving the end goal with tunable perf, cost and quality

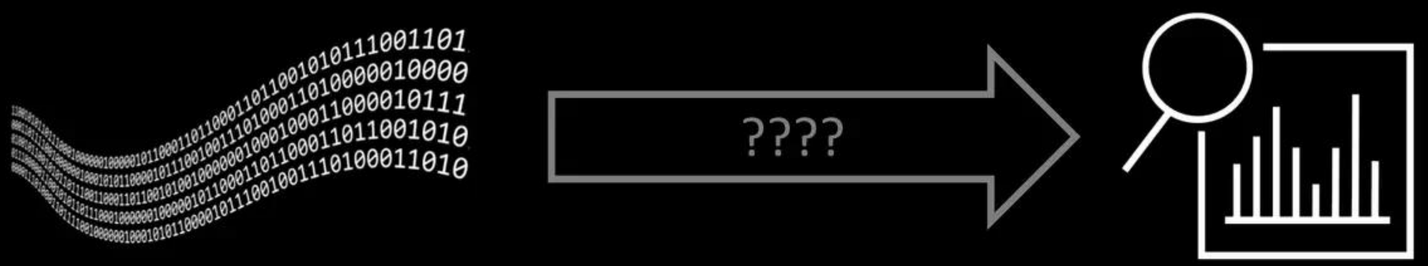
# This talk

How to think about design

Common design patterns

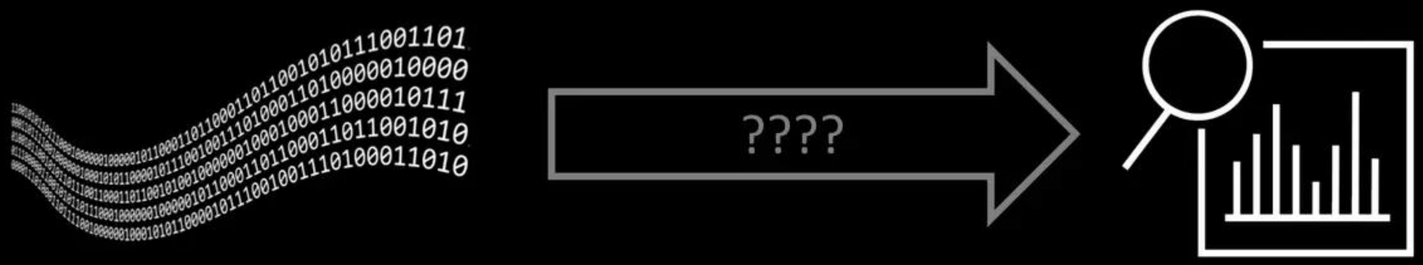
How we are making this easier

# Streaming Pipeline Design




Data streams

Insights




**What?**

**Why?**

**How?**

# What?

## What is your input?

What is your data?

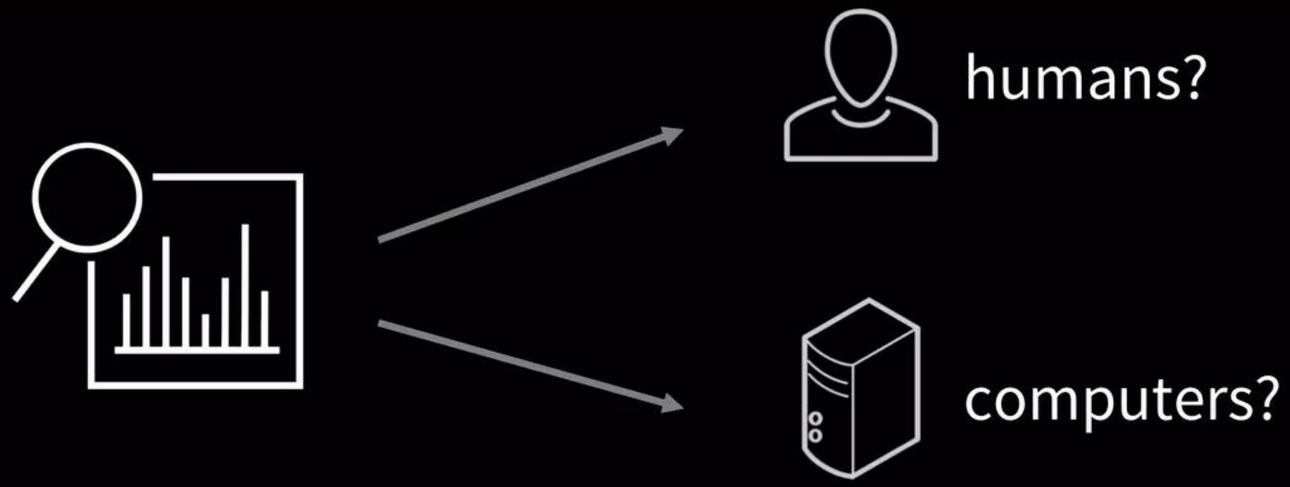
What format and system is your data in?

## What is your output?

What results do you need?

What throughput and latency do you need?

# Why?




## Why do you want this output in this way?

Who is going to take actions based on it?

When and how are they going to consume it?

# Why? Common mistakes!

#1

"I want my dashboard with counts to be updated every second"

No point of updating every second if humans are going to take actions in minutes or hours

#2

"I want to generate automatic alerts with up-to-the-last second counts"

*(but my input data is often delayed)*

No point taking fast actions on low quality data and results

# Why? Common mistakes!

#3

"I want to train machine learning models on the results"

*(but my results are in a key-value store)*

Key-value stores are not great for large, repeated data scans which machine learning workloads perform

# How?

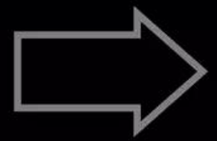
How to process the data?

????

How to store the results?


# Streaming Design Patterns

What?
Why?



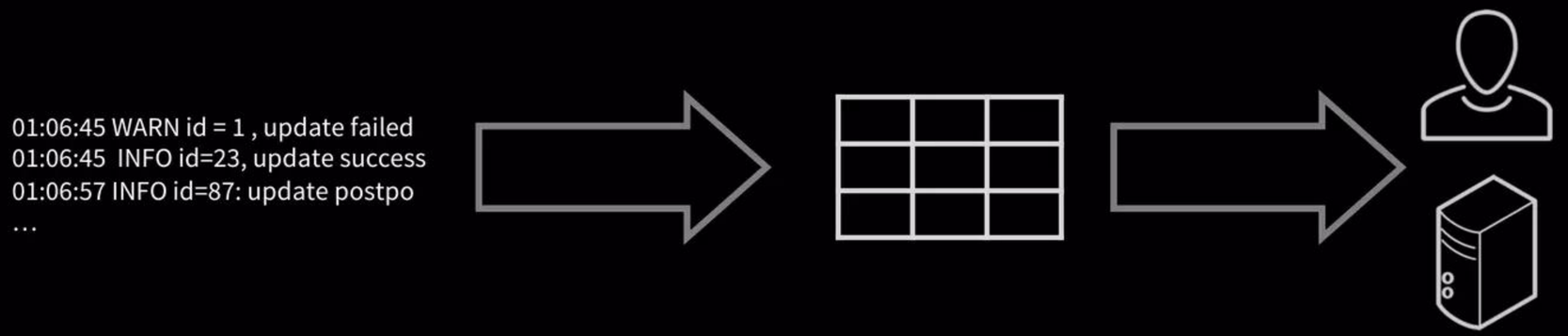
How?

# Pattern 1: ETL

**What?**

Input: unstructured input stream from files, Kafka, etc.

Output: structured tabular data




**Why?** Query latest structured data interactively or with periodic jobs

# P1: ETL

## What?

Convert unstructured input to structured tabular data

Latency: few minutes

## Why?

Query latest structured data interactively or with periodic jobs

## How?

Process: Use Structured Streaming query to transform unstructured, dirty data

Run 24/7 on a cluster with default trigger

Store: Save to structured scalable storage that supports data skipping, etc.

E.g.: Parquet, ORC, or even better, Delta Lake

# P1: ETL with Delta Lake

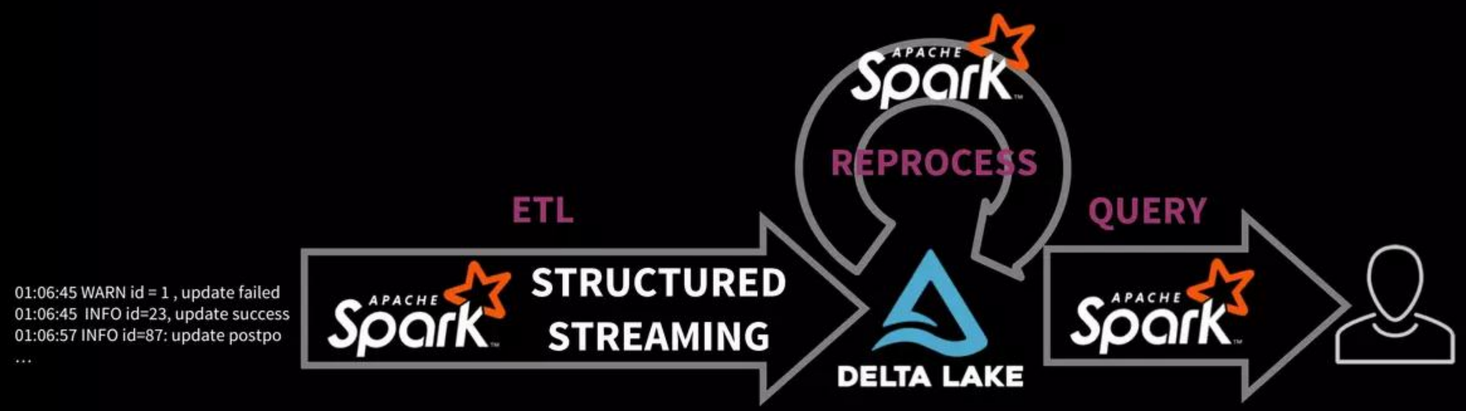
**How?**

Store: Save to DELTA LAKE

Read with snapshot guarantees while writes are in progress

Concurrently reprocess data with full ACID guarantees

- Coalesce small files into larger files
- Update table to fix mistakes in data
- Delete data for GDPR



# P1.1: Cheaper ETL

**What?**

Convert unstructured input to structured tabular data

Latency: ~~few minutes~~ hours

Not have clusters up 24/7

**Why?**

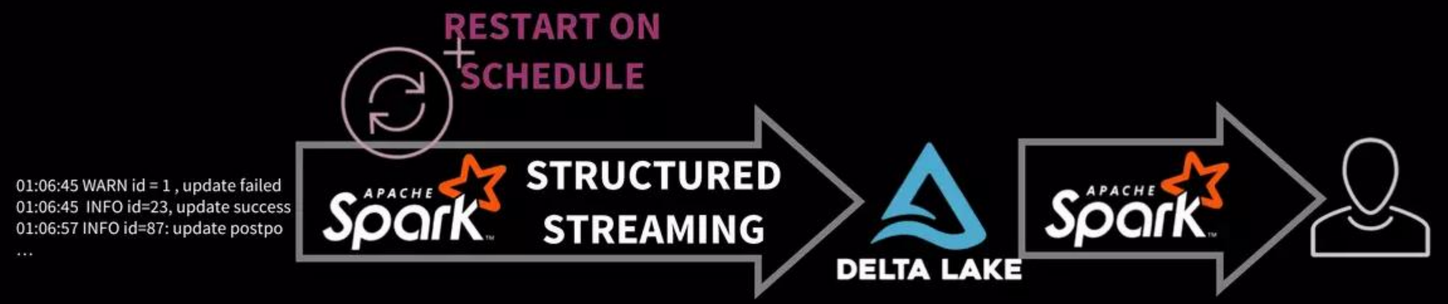
Query latest data ~~interactively or~~ with periodic jobs

Cheaper solution

**How?**

Process: Still use Structured Streaming query!

- Run streaming query with "trigger.once" for processing all available data since last batch
- Set up external schedule (every few hours?) to periodically start a cluster and run one batch



# P1.2: Query faster than ETL!

**What?**

Latency: ~~hours~~ seconds

**How?**

Query data in Kafka directly using Spark SQL

Can process up to the last records received by Kafka when the query was started

**Why?**

Query latest up-to-the last second data interactively

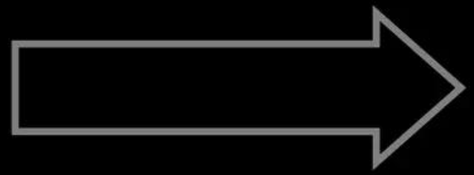
# Pattern 2: Key-value output

**What?**

Input: new data for each key

Output: updated values for each key

```
{ "key1": "value1" }
{ "key1": "value2" }
{ "key2": "value3" }
```



| KEY | LATEST VALUE |
|---|---|
| key1 | value2 |
| key2 | value3 |

Aggregations (sum, count, …)

Sessionizations

**Why?**

Lookup latest value for key (dashboards, websites, etc.)

OR

Summary tables for querying interactively or with periodic jobs

# P2.1: Key-value output for lookup

**What?**

Generate updated values for keys

Latency: seconds/minutes

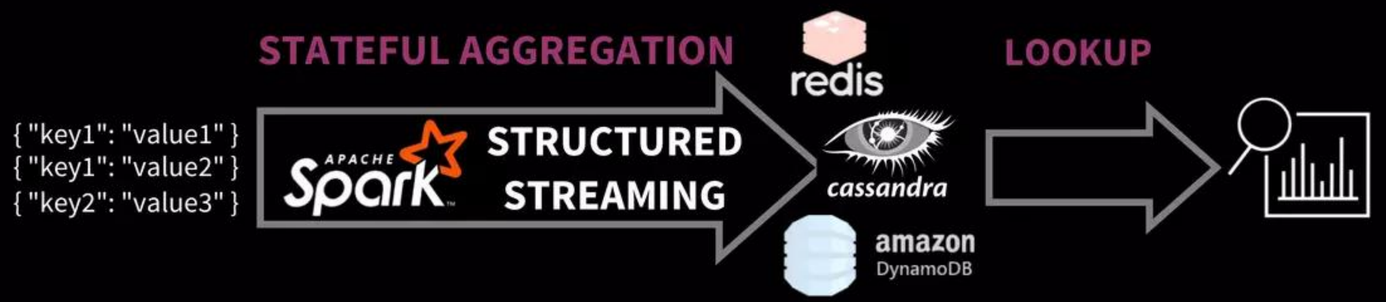
**How?**

Process: Use Structured Streaming with stateful operations for aggregation

Store: Save in key-values stores optimized for single key lookups

**Why?**

Lookup latest value for key



# P2.2: Key-value output for analytics

**What?**

Generate updated values for keys

Latency: ~~seconds/~~minutes

**How?**

Process: Use Structured Streaming with stateful operations for aggregation

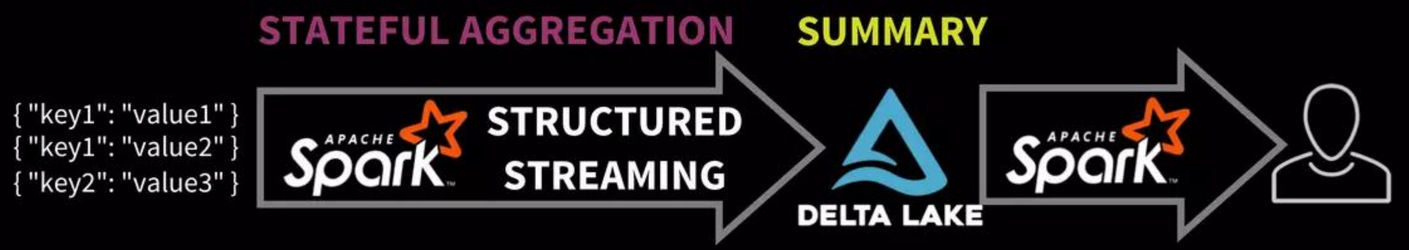
Store: Save in *Delta Lake*!

*Delta Lake* supports upserts using MERGE

**Why?**

~~Lookup latest value for key~~

Summary tables for analytics



# P2.2: Key-value output for analytics

**How?**

Stateful operations for aggregation

Delta Lake supports upserts using Merge SQL operation

- Scala/Java/Python APIs with same semantics as SQL Merge
- SQL Merge supported in Databricks Delta, not in OSS yet


```
streamingDataFrame.foreachBatch { batchOutputDF =>
    DeltaTable.forPath(spark, "/aggs/").as("t")
        .merge(
            batchOutputDF.as("s"),
           "t.key = s.key")
        .whenMatched().update(...)
        .whenNotMatched().insert(...)
        .execute()
}.start()
```

# P2.2: Key-value output for analytics

**How?**

Stateful operations for aggregation

Delta Lake supports upserts using Merge

Stateful aggregation requires setting watermark to drop very late data

Dropping some data leads some inaccuracies

# P2.3: Key-value aggregations for analytics

## What?

Generate aggregated values for keys

Latency: hours/days

Do not drop any late data

## Why?

Summary tables for analytics

Correct

## How?

Process: ETL to structured table (no stateful aggregation)

Store: Save in Delta Lake

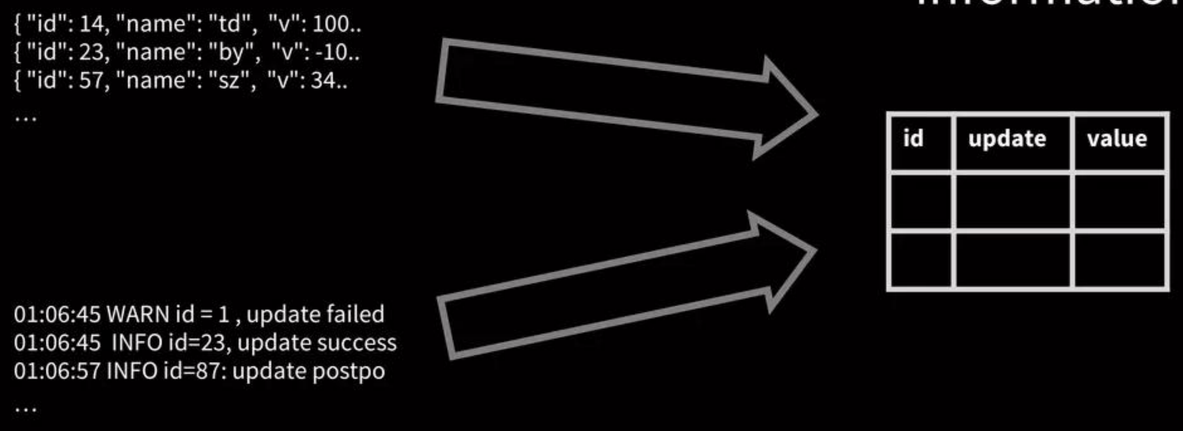
Post-process: Aggregate after all delayed data received

# Pattern 3: Joining multiple inputs

**What?**

Input: Multiple data streams based on common key

Output: Combined information



# P3.1: Joining fast and slow streams

## What + Why?

Input: One fast stream of facts and one slow stream of dimension changes

Output: Fast stream enriched by data from slow stream

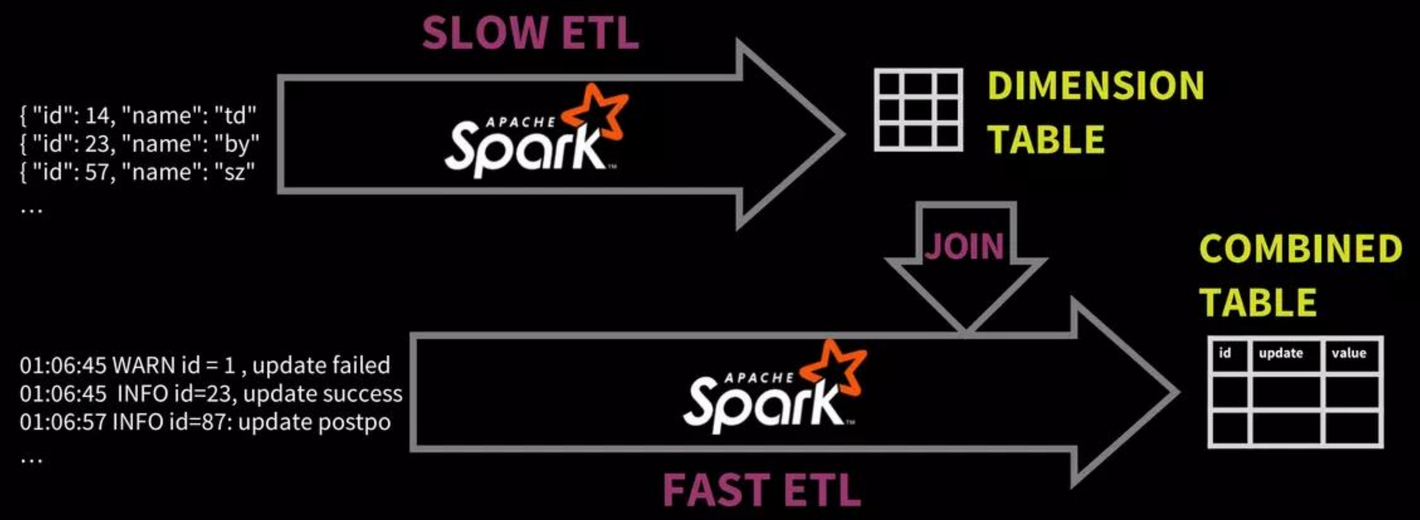
Example:

product sales info ("facts") enriched by more product info ("dimensions")

## How?

ETL slow stream to a dimension table

Join fast stream with snapshots of the dimension table

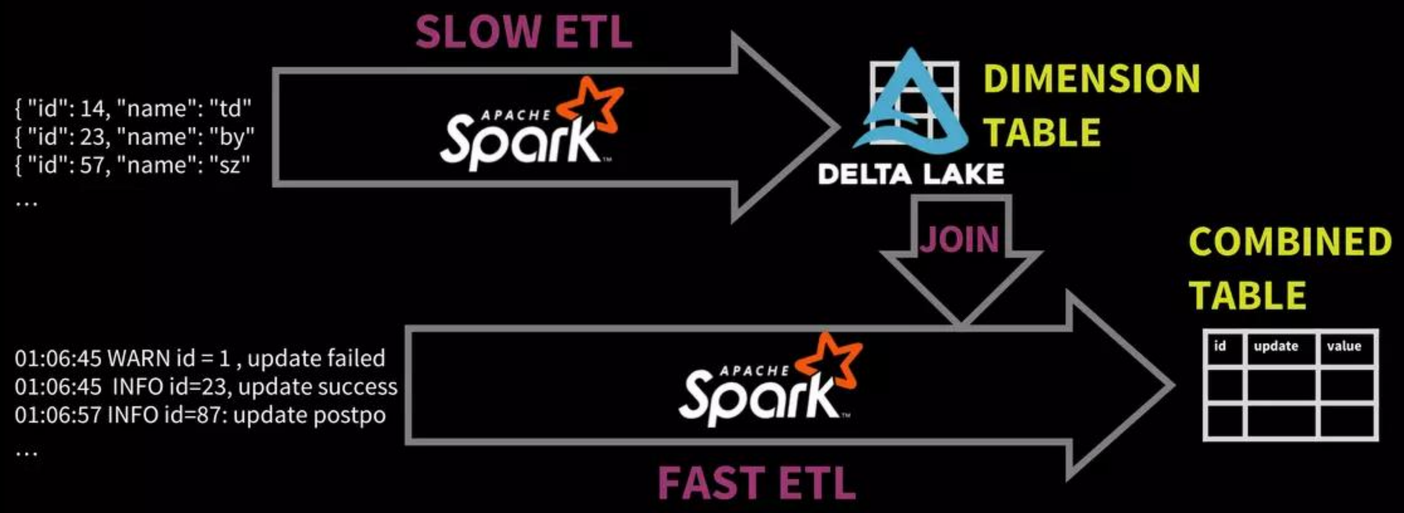


# P3.1: Joining fast and slow streams

## How? - Caveats

Structured Streaming does not reload dimension table snapshot

Changes by slow ETL wont be seen until restart




## Better Solution

Store dimension table in Delta Lake

Delta Lake's versioning allows changes to be detected and the snapshot automatically reloaded without restart**

*** available only in Databricks Delta Lake*

# P3.1: Joining fast and slow stream

## How? - Caveats

Delays in updates to dimension table can cause joining with stale dimension data

E.g. sale of a product received even before product table has any info on the product

## Better Solution

Treat it as a "joining fast and fast data"

# P3.2: Joining fast and fast data

## What + Why?

Input: Two fast streams where either stream maybe delayed

Output: Combined info even if one is delayed over another

Example:

ad impressions and ad clicks

## How?

Use stream-stream joins in Structured Streaming

- Data will be buffered as state
- Watermarks define how long to buffer before giving up on matching


*See my past deep dive talk for more info*

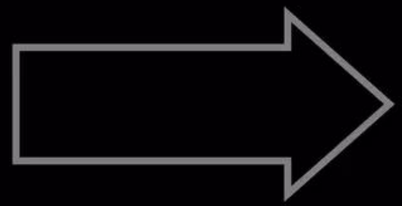
# Pattern 4: Change data capture

**What?**

Input: Change data based on a primary key

Output: Final table after changes

```
INSERT a, 1
INSERT b, 2
UPDATE a, 3
DELETE b
INSERT b, 4
```



| key | value |
|---|---|
| a | 3 |
| b | 4 |

**Why?**

End-to-end replication of transactional tables into analytical tables

# P4: Change data capture

## How?

Use foreachBatch and Merge

In each batch, apply changes to the Delta table using Merge

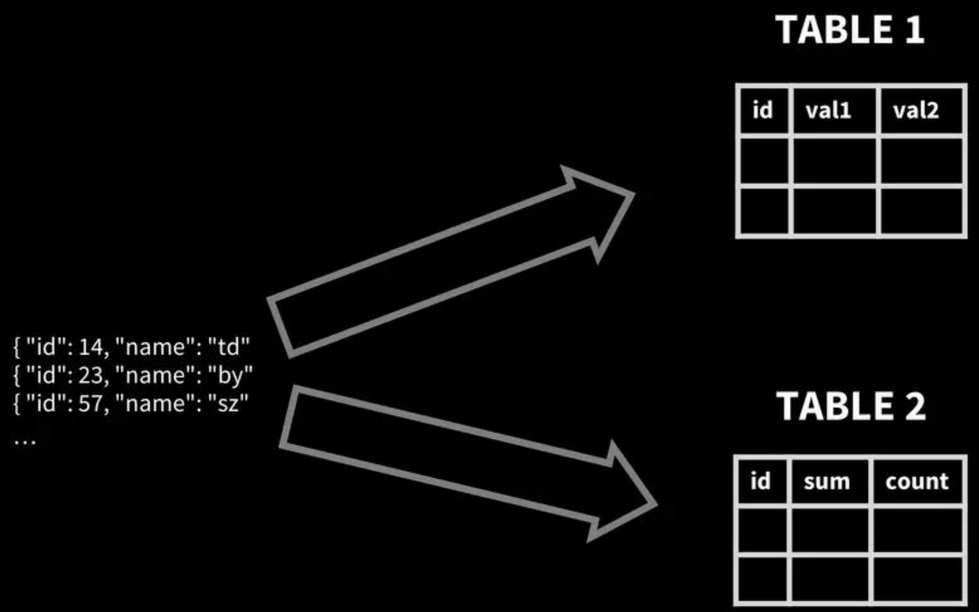
Delta Lake's Merge support extended syntax with includes multiple match clauses, clause conditions and deletes

```
streamingDataFrame.foreachBatch { batchOutputDF =>
    DeltaTable.forPath(spark, "/deltaTable/").as("t")
        .merge(batchOutputDF.as("updates"),
            "t.key = s.key")
        .whenMatched("delete = false").update(...)
        .whenMatched("delete = true").delete()
        .whenNotMatched().insert(...)
        .execute()
}.start()
```

```
INSERT a, 1
INSERT b, 2
UPDATE a, 3
DELETE b
INSERT b, 4
```


See merge examples in docs for more info

# Pattern 5: Writing to multiple outputs



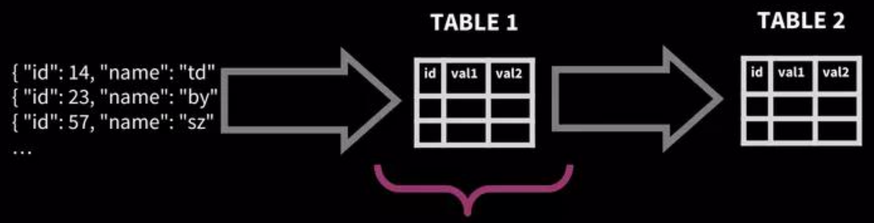

**What?**

| RAW LOGS | TABLE CHANGE LOG | SUMMARY FOR ANALYTICS |
|---|---|---|
| + | + | + |
| SUMMARIES | UPDATED TABLE | SUMMARY FOR LOOKUP |

# P5: Serial or Parallel?

## How?

### Serial

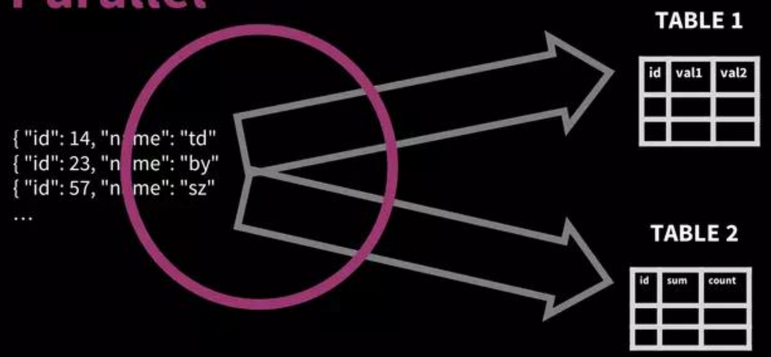



Writes table 1 and reads it again

Cheap or expensive depends on the size and format of table 1

Higher latency

### Parallel



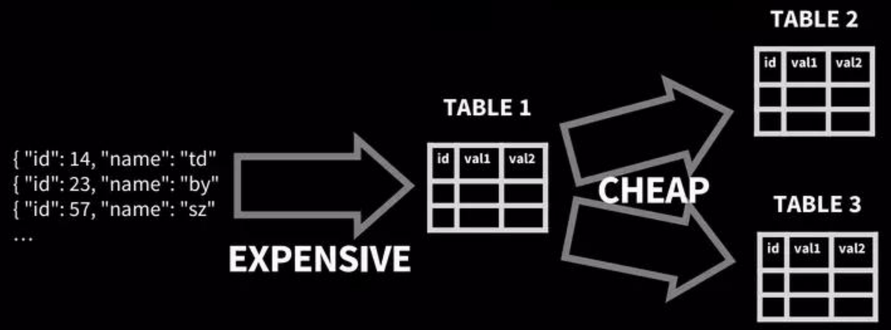

Reads input twice, may have to parse the data twice

Cheap or expensive depends on size of raw input + parsing cost

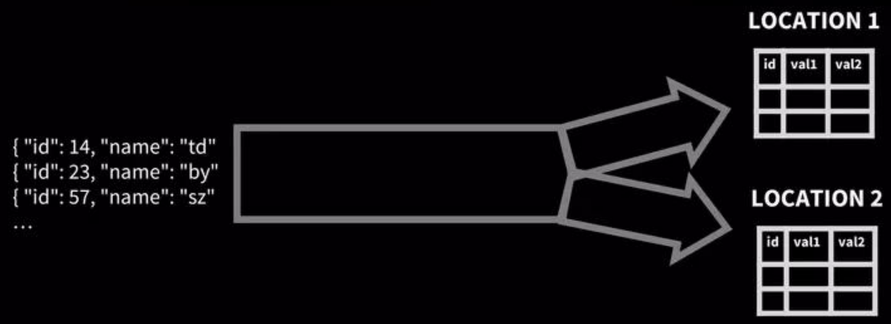
# P5: Combination!

## How?

### Combo 1: Multiple streaming queries




### Combo 2: Single query + foreachBatch




Do expensive parsing once, write to table 1

Do cheaper follow up processing from table 1

Good for

- ETL + multiple levels of summaries
- Change log + updated table

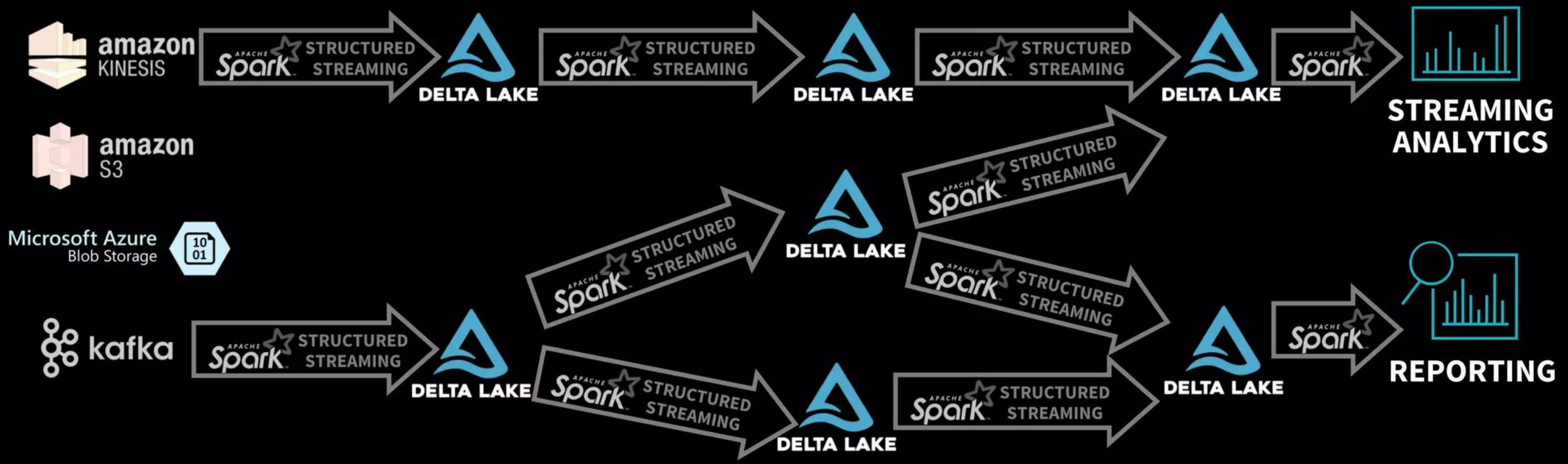
Still writing + reading table1

Compute once, write multiple times

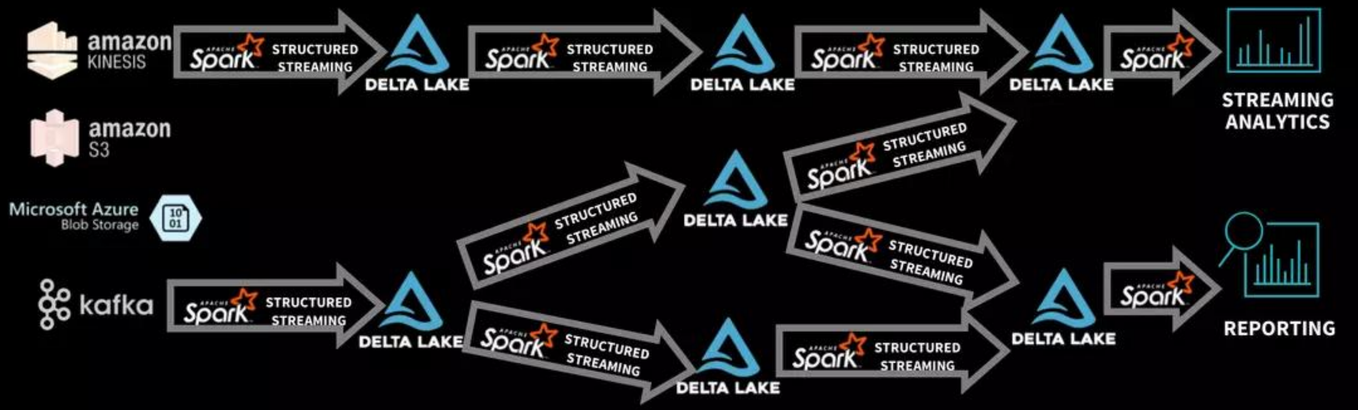
```
streamingDataFrame.foreachBatch { batchSummaryData =>
  // write summary to Delta Lake
  // write summary to key value store
}.start()
```

Cheaper, but looses exactly-once guarantee

# Production Pipelines @ databricks



# In Progress: Declarative Pipelines




Allow users to declaratively express the entire DAG of pipelines as a dataflow graph

# In Progress: Declarative Pipelines

Enforce metadata, storage, and quality declaratively

```
dataset("warehouse")
  .query(input("kafka").select(…).join(…))     // Query to materialize
  .location(…)                                 // Storage Location
  .schema(…)                                   // Optional strict schema checking
  .metastoreName(…)                            // Hive Metastore
  .description(…)                              // Human readable description
  .expect("validTimestamp",                    // Expectations on data quality
    "timestamp > 2012-01-01 AND …",
    "fail / alert / quarantine")
```

*Coming Soon

# In Progress: Declarative Pipelines

Enforce metadata, storage, and quality declaratively

```
dataset("warehouse")
  .query(input("kafka").withColumn(…).join(…)) // Query to materialize
  .location(…)                                 // Storage Location
  .schema(…)                                   // Optional strict schema checking
  .metastoreName(…)                            // Hive Metastore
  .description(…)                              // Human readable description
  .expect("validTimestamp",                    // Expectations on data quality
    "timestamp > 2012-01-01 AND …",
    "fail / alert / quarantine")
```

*Coming Soon

# In Progress: Declarative Pipelines

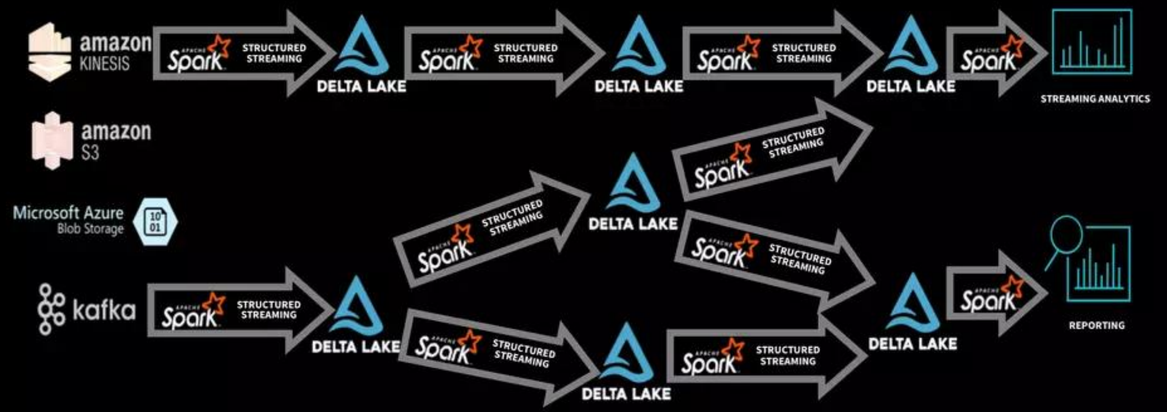
Enforce metadata, storage, and quality declaratively

```
dataset("warehouse")
  .query(input("kafka").withColumn(…).join(…)) // Query to materialize
  .location(…)                                 // Storage Location
  .schema(…)                                   // Optional strict schema checking
  .metastoreName(…)                            // Hive Metastore
  .description(…)                              // Human readable description
  .expect("validTimestamp",                    // Expectations on data quality
    "timestamp > 2012-01-01 AND …",
    "fail / alert / quarantine")
```

*Coming Soon

# In Progress: Declarative Pipelines

- Unit test part or whole DAG with sample input data
- Integration test DAG with production data
- Deploy/upgrade new DAG code
- Monitor whole DAG in one place
- Rollback code / data when needed




*Coming Soon

# Build your own Delta Lake at **https://delta.io**