Databricks Streaming and Lakeflow Spark Declarative Pipelines

Databricks Academy

Agenda

1. Introduction to Streaming	Lecture	Demo	Lab
Streaming Data Concepts	✓		
Introduction to Structured Streaming	✓		
Reading from a Streaming Query		✓	
Streaming from Delta Lake	✓		
Streaming Query			✓
2. Aggregations, Time Windows, Watermarks	**Lecture**	**Demo**	**Lab**
Aggregations, Time Windows, Watermarks	✓		
Event Time + Aggregations Over Time Windows	✓		
Stream Aggregations			✓
Windowed Aggregation with Watermark		✓	
3. Streaming Joins (Optional)	**Lecture**	**Demo**	**Lab**
Streaming Joins (Optional)	✓		
Stream/Stream Joins (Optional)		✓	
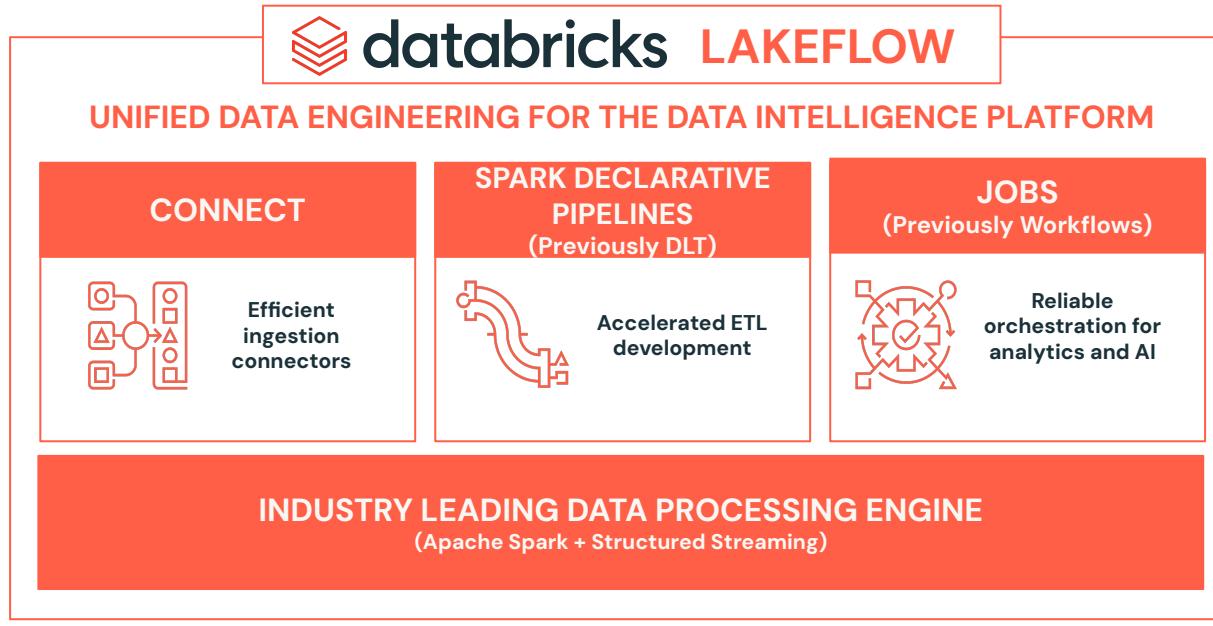
Agenda

4. Streaming ETL Patterns with Lakeflow Spark Declarative Pipelines	Lecture	Demo	Lab
Data Ingestion Patterns	✓		
Auto Load to Bronze		✓	
Stream from Multiplex Bronze		✓	
Data Quality Enforcement	✓	✓	
Streaming ETL			✓

Data Engineering in Databricks

databricks LAKEFLOW

UNIFIED DATA ENGINEERING FOR THE DATA INTELLIGENCE PLATFORM

CONNECT	SPARK DECLARATIVE PIPELINES (Previously DLT)	JOBS (Previously Workflows)
Efficient ingestion connectors	Accelerated ETL development	Reliable orchestration for analytics and AI

INDUSTRY LEADING DATA PROCESSING ENGINE
(Apache Spark + Structured Streaming)

UNIFIED GOVERNANCE	Unity Catalog
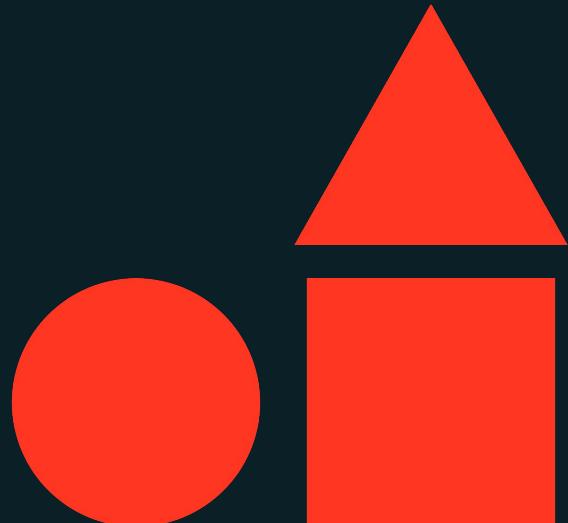
OPTIMIZED STORAGE	DELTA LAKE Parquet ICEBERG

Introduction to Streaming

Databricks Streaming and Lakeflow Spark Declarative Pipelines

databricks
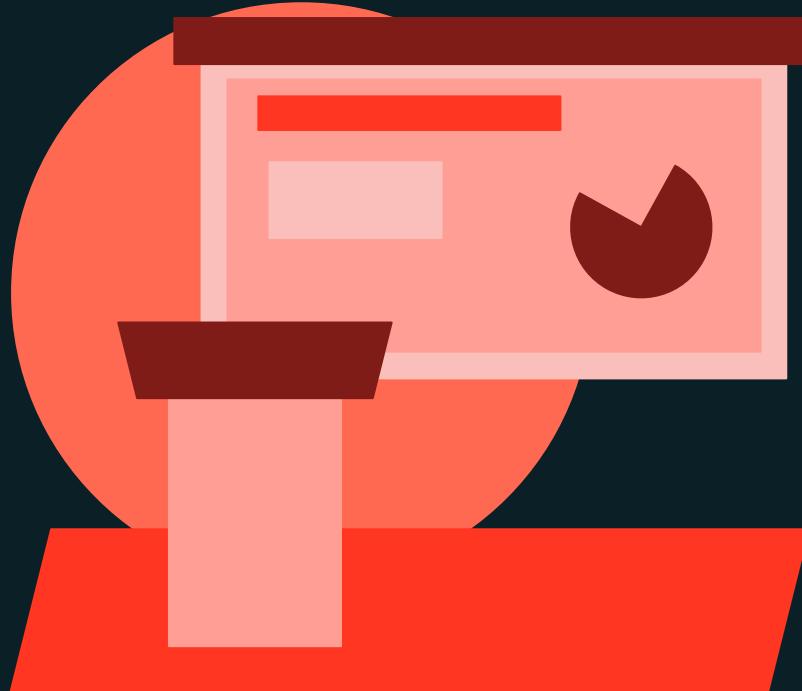
Introduction to Streaming

LECTURE

Streaming Data Concepts

Stream Processing

Why is stream processing getting popular?

Data Velocity & Volumes — Rising data velocity & volumes requires continuous, incremental processing – cannot process all data in one batch on a schedule

Real-time analytics — Businesses demand access to fresh data for actionable insights and faster, better business decisions

Operational applications — Critical applications need real-time data for effective, instantaneous response

💡 **The vast majority of the data in the world is streaming data!**

Stream Processing Use Cases

Stream processing is a key component of big data applications across all industries

Notifications
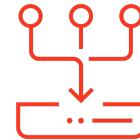
Real-time reporting

Incremental ETL

Update data to serve in real-time

Real-time decision making

Online ML

Advantages of Stream Processing

Why use streaming (vs. batch) ?

Incremental processing of new data is natural and intuitive

Better fault–tolerance upon restarts

Unlocks low latency SLAs for time sensitive workloads
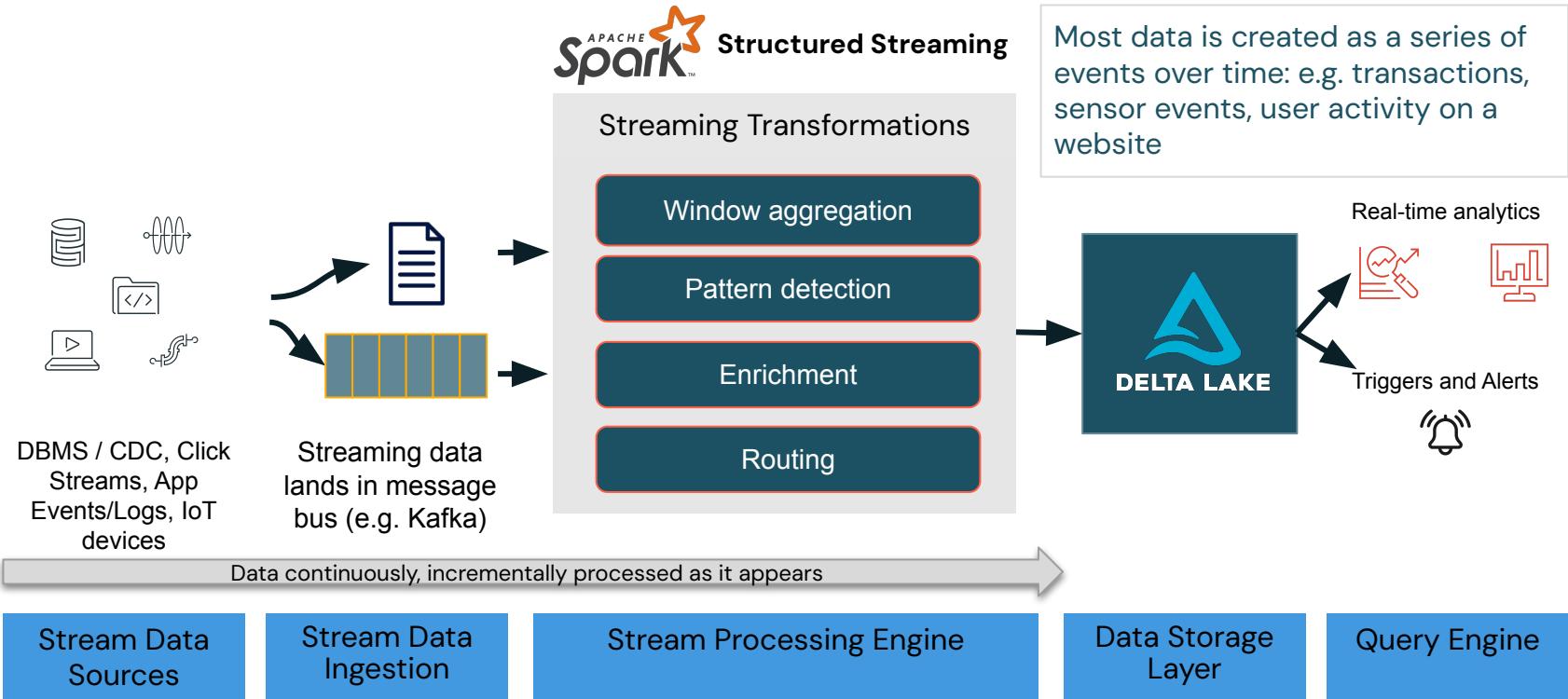
Higher compute utilization and scalability through continuous and incremental processing

Stream Processing Architecture

Modern data processing pipeline

Structured Streaming

Streaming Transformations

- Window aggregation
- Pattern detection
- Enrichment
- Routing

Most data is created as a series of events over time: e.g. transactions, sensor events, user activity on a website

DELTA LAKE

Real-time analytics

Triggers and Alerts

DBMS / CDC, Click Streams, App Events/Logs, IoT devices

Streaming data lands in message bus (e.g. Kafka)

Data continuously, incrementally processed as it appears

Stream Data Sources	Stream Data Ingestion	Stream Processing Engine	Data Storage Layer	Query Engine

Challenges of Stream Processing

Stream processing is not easy

- Processing each event exactly once despite machine failures
- Processing out-of-order data based on application timestamps (also called event time)
- Maintaining large amounts of state
- Handling load imbalance and stragglers
- Determining how to update output sinks as new events arrive
- Writing data transactionally to output systems

SOLUTION

APACHE
Spark™
Structured Streaming

databricks
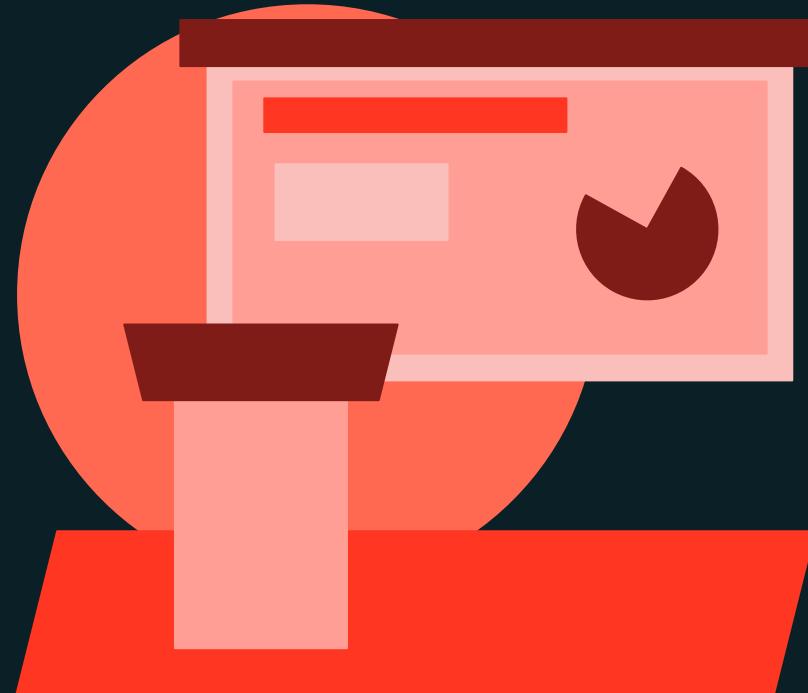
Introduction to Streaming

Introduction to Structured Streaming
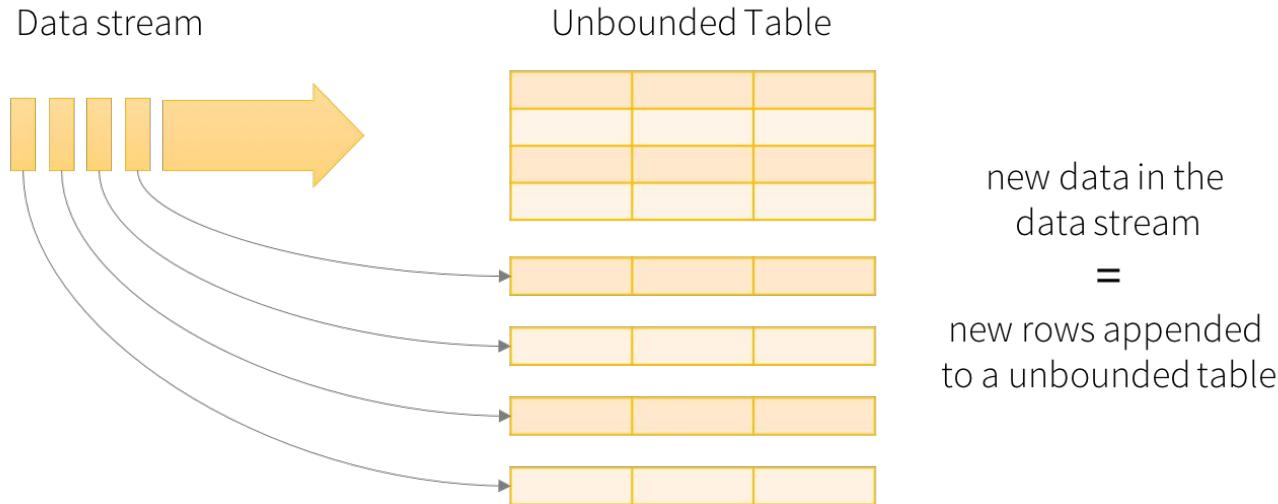
What is Structured Streaming

Apache Spark structured streaming basics

- A scalable, fault-tolerant **stream processing framework** built on Spark SQL engine.
- Uses **existing structured APIs** (DataFrames, SQL Engine) and provides similar API as batch processing API.
- Includes **stream specific features**; end-to-end, exactly-once processing, fault-tolerance etc.
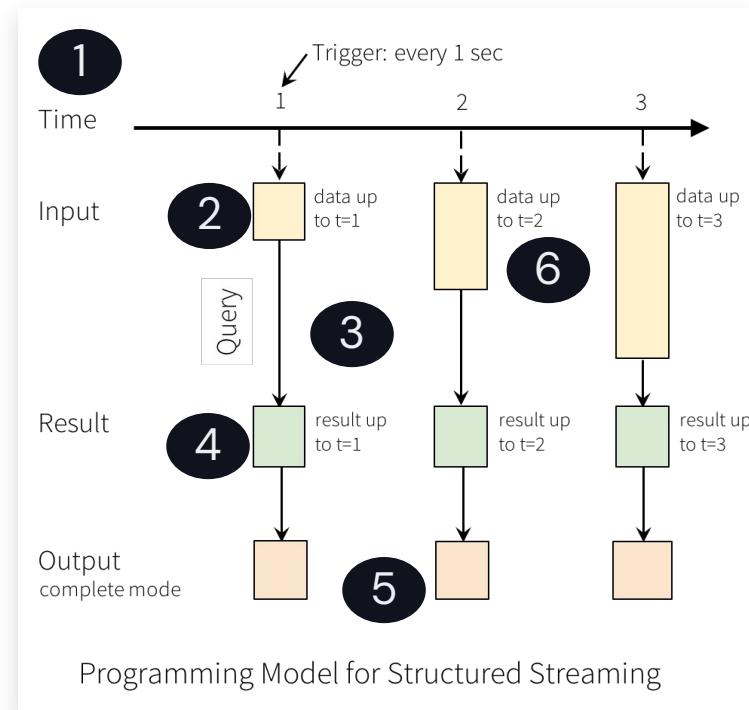
Streams as Unbounded Tables

Data stream

Unbounded Table

new data in the data stream

=

new rows appended to a unbounded table

How Structured Streaming Works

Execution mode
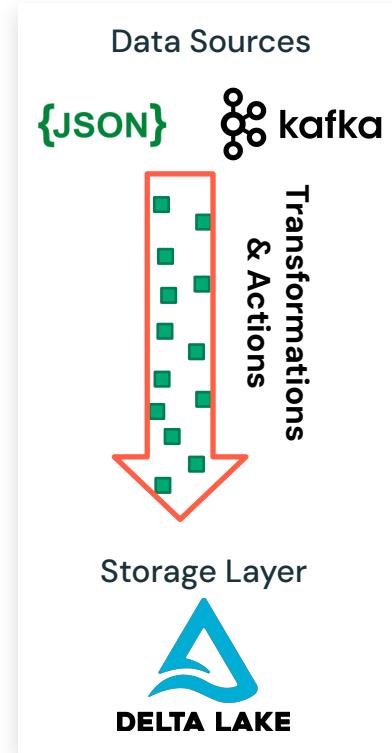
1. Timeline
2. An **input table** is defined by configuring a streaming read against **source**.
3. A **query** is defined against the input table.
4. This logical query on the input table generates the **results table**.
5. The **output** of a streaming pipeline will persist updates to the results table by writing to an external **sink**.
6. New rows are appended to the input table for each **trigger interval**.

Programming Model for Structured Streaming

Anatomy of a Streaming Query
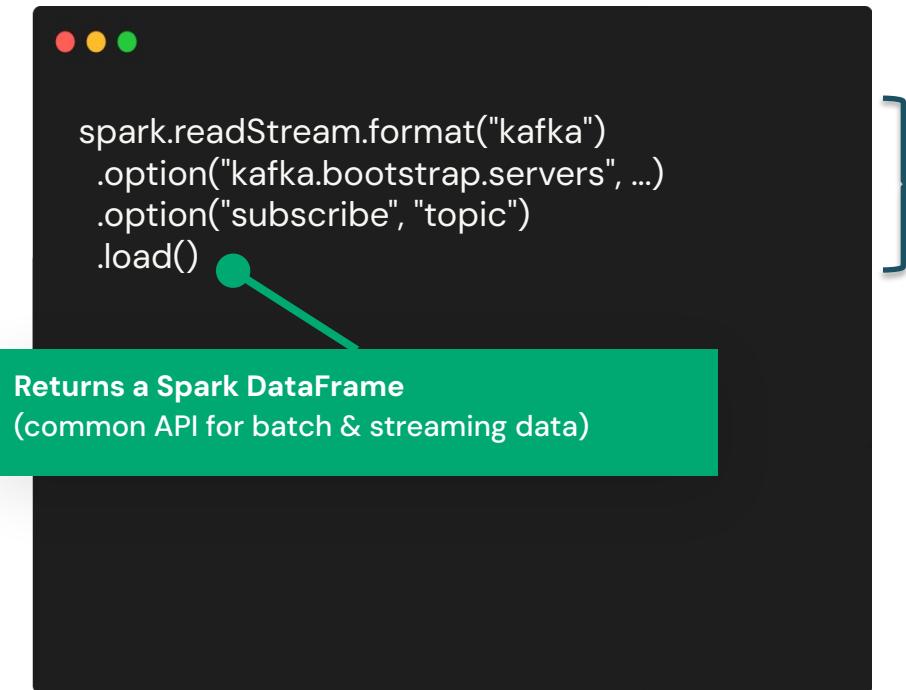
Structured streaming core concepts

- Example:
  - Read JSON data from Kafka
  - Parse nested JSON
  - Store in structured Delta Lake table

- Core concepts:
  - Input sources
  - Transformations
  - Sinks
  - Triggers

Data Sources

{JSON}   kafka

Transformations & Actions

Storage Layer

DELTA LAKE

Anatomy of a Streaming Query

Structured streaming core concepts

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", ...)
  .option("subscribe", "topic")
  .load()
```

Returns a Spark DataFrame
(common API for batch & streaming data)

Source:

- Specify where to read data from
- OS Spark supports Kafka and file sources
- Databricks runtimes include connector libraries supporting Delta, Event Hubs, and Kinesis

```
spark.readStream.format(<source>)
.option(<>,<>)...
.load()
```

Anatomy of a Streaming Query

Structured streaming core concepts

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", ...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
```

Transformations:

- 100s of built-in, optimized SQL functions like from_json

- In this example, cast bytes from Kafka records to a string, parse it as JSON, and generate nested columns

Anatomy of a Streaming Query

Structured streaming core concepts

```
spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", ...)
  .option("subscribe", "topic")
  .load()
  .selectExpr("cast (value as string) as json")
  .select(from_json("json", schema).as("data"))
  .writeStream
  .format("delta")
  .option("path", "/deltaTable/")
```

Sink: Write transformed output to external storage systems

Databricks runtimes include connector library supporting Delta

OS Spark supports:

- Files and Kafka for production
- Console and memory for development and debugging
- foreachBatch to execute arbitrary code with the output data

Anatomy of a Streaming Query

Structured streaming core concepts

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

- **Checkpoint location:** For tracking the progress of the query

- **Output Mode:** Defines how the data is written to the sink; Equivalent to "save" mode on static DataFrames

- **Trigger:** Defines how frequently the input table is checked for new data; Each time a trigger fires, Sparks check for new data and updates the results

DataFrame as a Unifying API

Batch

```
spark
 .read
  .table("source_name")
  .withColumn("...")
  .select(...)
 .write
[ .outputMode(...) ]
  .saveAsTable("sink_table")
```

Streaming

```
spark
 .readStream
[ .withWatermark(...) ]
  .table("source_name")
  .withColumn("...")
  .select(...)
 .writeStream
[ .trigger(...) ]
[ .queryName(...) ]
  .option("checkpointLocation", ...)
  .toTable("sink_table")
```

Anatomy of a Streaming Query

Structured streaming core concepts

Trigger Types:

Fixed interval micro batch	.trigger(processingTime = "2 minutes")	Micro-batch processing kicked off at the *user-specified interval*
Triggered One-time micro batch	.trigger(once=True)	DEPRECATED – Process all of the available data as a *single micro-batch* and then automatically stop the query
Triggered One-time micro batches	.trigger(availableNow=True)	Process all of the available data as *multiple micro-batches* and then automatically stop the query
Continuous Processing	.trigger(continuous= "2 seconds")	EXPERIMENTAL – Long-running tasks that *continuously* read, process, and write data as soon events are available, with checkpoints at the specified frequency
Default		Databricks: 500ms fixed interval OS Apache Spark: Process each microbatch as soon as the previous has been processed

Anatomy of a Streaming Query

Structured Streaming Core Concepts

Output Modes:

Complete	• The entire updated Result Table is written to the sink. • The individual sink implementation decides how to handle writing the entire table.
Append	Only the new rows appended to the Result Table since the last trigger are written to the sink.
Update	Only new rows and the rows in the Result Table that were updated since the last trigger will be outputted to the sink.

Note: The output modes supported depends on the type of transformations and sinks used by the streaming query. Refer to the the [Structured Streaming Programming Guide](#) for details.
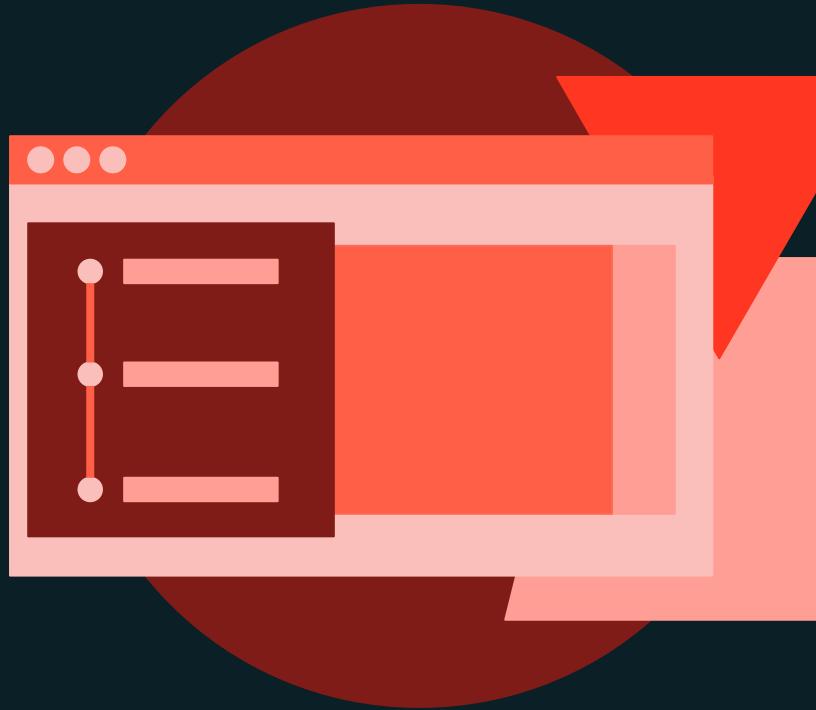
Introduction to Streaming

Reading from a Streaming Query

databricks
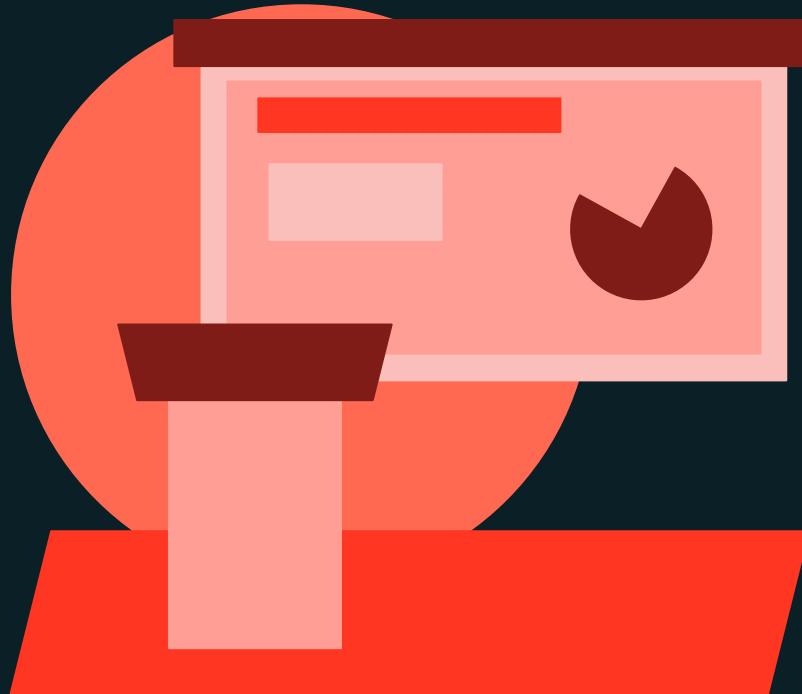
Introduction to Streaming

LECTURE

Streaming from Delta Lake

Streaming from Delta Lake

Using a Delta table as a streaming source

- Each committed version represents new data to stream. Delta Lake transactions logs identify the version's new data files

- Structured Streaming assumes append-only sources. Any non-append changes to a Delta table causes queries streaming from that table to throw exceptions.

  - Set `delta.appendOnly = true` to prevent non-append modifications to a table.

  - Use Delta Lake change data feed to propagate arbitrary change events to downstream consumers (discussed later in this course).

Streaming from Delta Lake

Using a Delta table as a streaming source

- You can <u>limit the input rate</u> for micro-batches by setting DataStreamReader options:
  - maxFilesPerTrigger: Maximum files read per micro-batch (default 1,000)
  - maxBytesPerTrigger: Soft limit to amount of data read per micro-batch (no default)
  - Note: Lakeflow Spark Declarative Pipelines auto-tune options for rate limiting, so you should avoid setting these options explicitly for your pipelines.

Streaming to Delta Lake

Using a Delta table as a streaming sink

- Each micro-batch written to the Delta table is committed as a new version.

- Delta Lake supports both append and complete output modes.
  - Append is most common.
  - Complete replaces the entire table with each micro-batch. It can be used for streaming queries that perform arbitrary aggregations on streaming data.

databricks

Introduction to Streaming

LAB EXERCISE

Streaming Query Lab

Aggregations, Time Windows, Watermarks

Databricks Streaming and Lakeflow Spark Declarative Pipelines
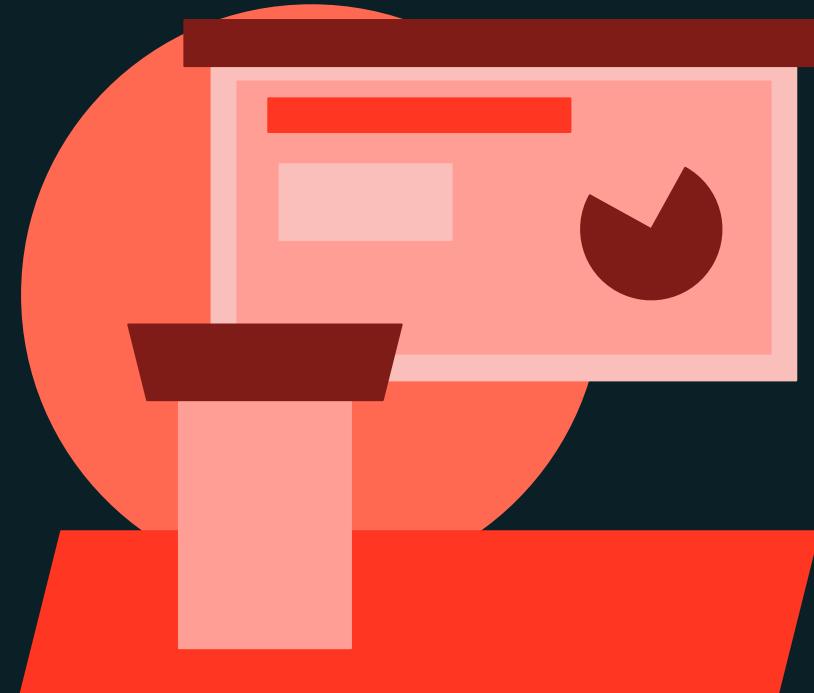
databricks

Introduction to Streaming

Aggregations, Time Windows, Watermarks

Types of Stream Processing

Stateless vs. Stateful processing

- **Stateless**
    - Typically trivial transformations. The way records are handled do not depend on previously seen records.
    - Example: Data Ingest (map-only), simple dimensional joins
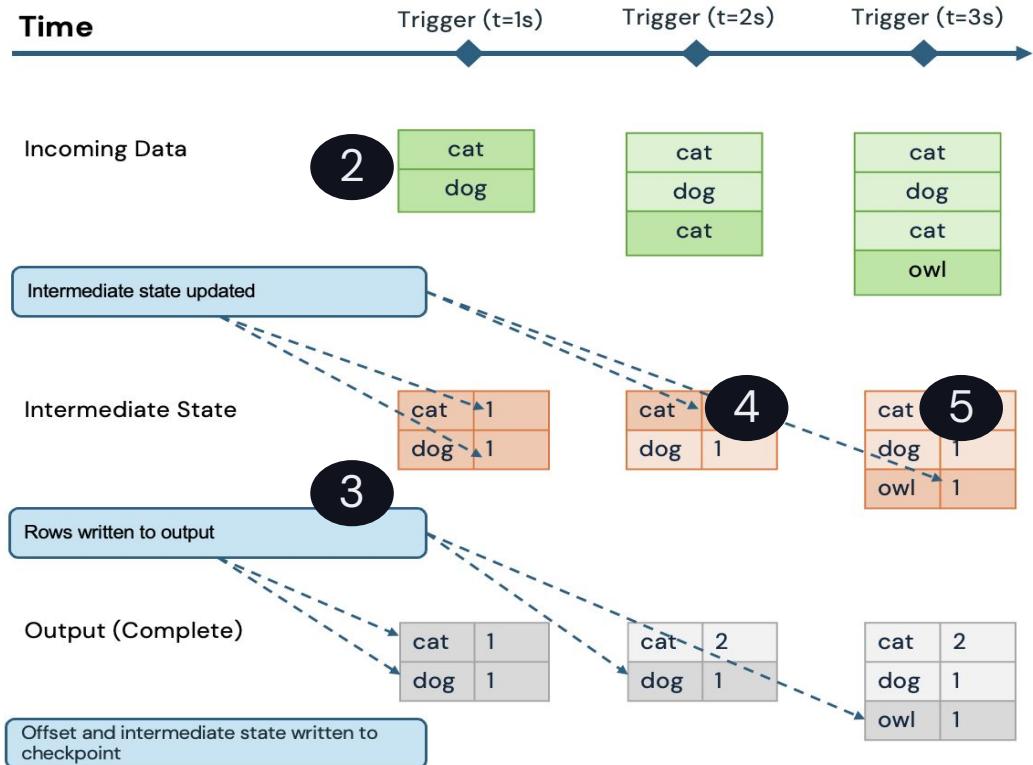
- **Stateful**
    - Previously seen records can influence new records
    - Example: Aggregations over time, Fraud/Anomaly Detection

Stateful Stream


```
spark
 .readStream
 .<source info>
 .groupBy(animal)
 .count()
 .writeStream
 .mode("complete")
 .<sink info>
 .trigger("1s")
 .start()
```

databricks
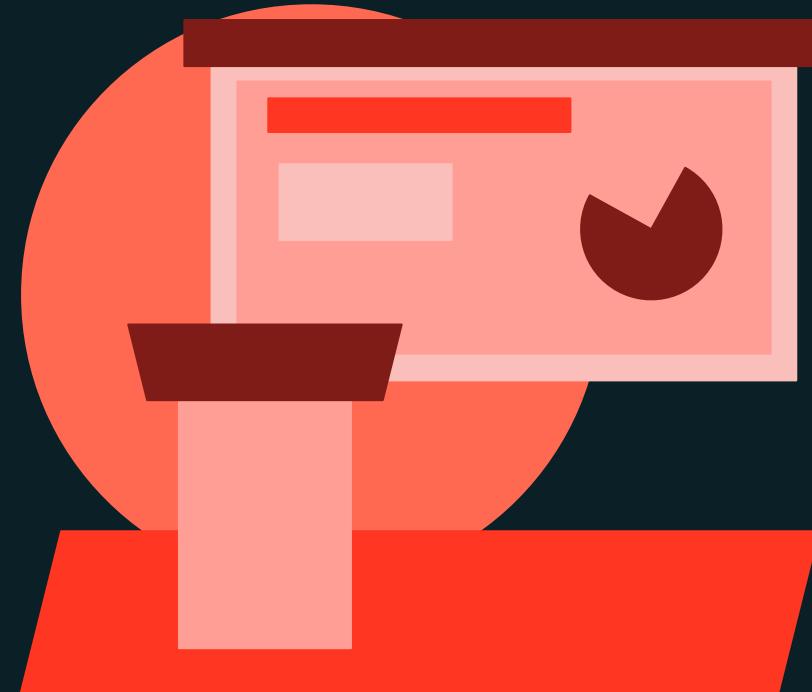
Introduction to Streaming

Event Time + Aggregations over Time Windows

Reasoning About Time

Event time vs. Processing time

Event time vs. processing time

- **Event Time:** time at which the event (record in the data) actually occurred.

- **Processing time:** time at which a record is actually processed.

- Important in every use case processing unbounded data in whatever order (otherwise no guarantee on correctness)

Time Based Windows

Tumbling window vs. Sliding window

Tumbling Window

- **No window overlap**

- Any given event gets aggregated into **only one** window group (e.g. 1:00–2:00 am, 2:00–3:00 am, 3:00-4:00 am, …)

Sliding Window

- **Windows overlap**

- Any given event gets aggregated into **multiple window** groups (e.g. 1:00–2:00 am, 1:30–2:30 am, 2:00–3:00 am, …)
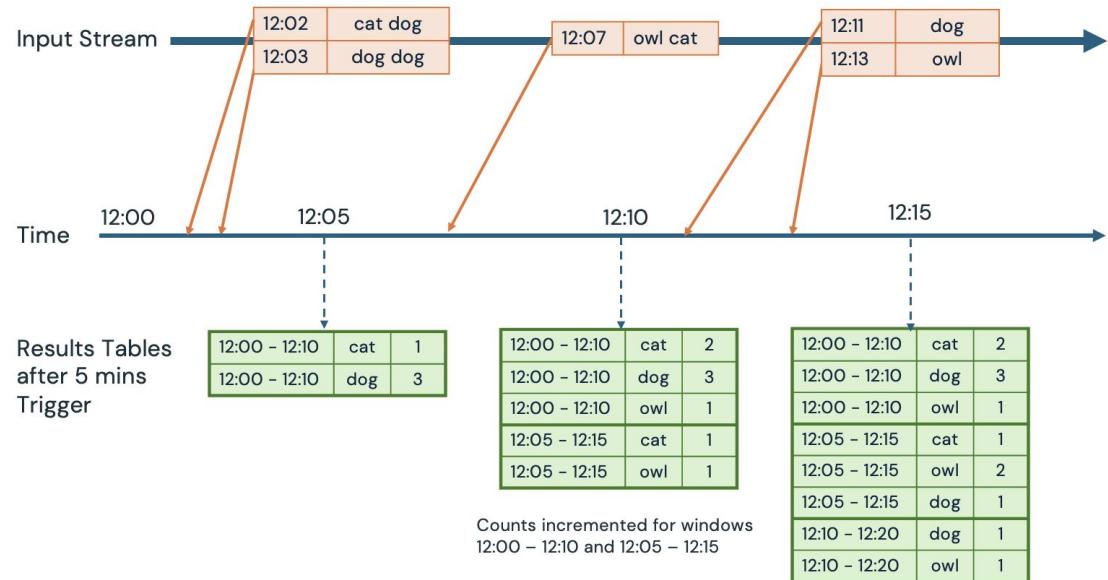
Time Based Windows

Sliding window example
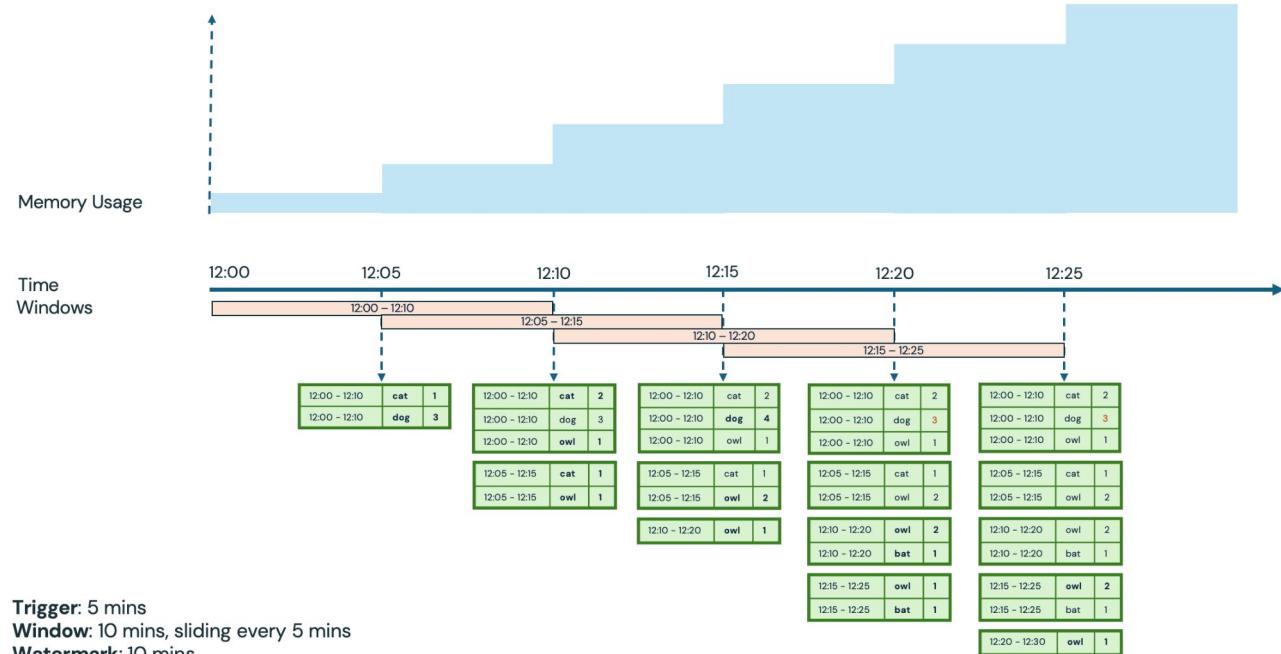

```
windowedDF =
  (
    eventsDF
    .groupBy(window("eventTime",
      "10 minutes",
      "5 minutes"))
    .count()
    .writeStream
    .trigger(processingTime="5 minutes")
  )
```

Input Stream

| 12:02 | cat dog |
| 12:03 | dog dog |

| 12:07 | owl cat |

| 12:11 | dog |
| 12:13 | owl |

Time

12:00   12:05   12:10   12:15

Results Tables after 5 mins Trigger

| 12:00 – 12:10 | cat | 1 |
| 12:00 – 12:10 | dog | 3 |

12:00 – 12:10	cat	2
12:00 – 12:10	dog	3
12:00 – 12:10	owl	1
12:05 – 12:15	cat	1
12:05 – 12:15	owl	1

Counts incremented for windows 12:00 – 12:10 and 12:05 – 12:15

| 12:00 – 12:10 | cat | 2 |
12:00 – 12:10	dog	3
12:00 – 12:10	owl	1
12:05 – 12:15	cat	1
12:05 – 12:15	owl	2
12:05 – 12:15	dog	1
12:10 – 12:20	dog	1
12:10 – 12:20	owl	1

Challenges: Memory Pressure

Long running query example

- Size of intermediate state will keep increase over time.
- Default behaviour is to maintain state in executor memory and DBFS.
- GC pauses will take longer and longer. Failure is inevitable.

Memory Usage

Time Windows

12:00   12:05   12:10   12:15   12:20   12:25

12:00 – 12:10
12:05 – 12:15
12:10 – 12:20
12:15 – 12:25

| 12:00 – 12:10 | cat | 1 |
| 12:00 – 12:10 | dog | 3 |

12:00 – 12:10	cat	2
12:00 – 12:10	dog	3
12:00 – 12:10	owl	1
12:05 – 12:15	cat	1
12:05 – 12:15	owl	1

12:00 – 12:10	cat	2
12:00 – 12:10	dog	4
12:00 – 12:10	owl	1
12:05 – 12:15	cat	1
12:05 – 12:15	owl	2
12:10 – 12:20	owl	1

12:00 – 12:10	cat	2
12:00 – 12:10	dog	3
12:00 – 12:10	owl	1
12:05 – 12:15	cat	1
12:05 – 12:15	owl	2
12:10 – 12:20	owl	2
12:10 – 12:20	bat	1
12:15 – 12:25	owl	1
12:15 – 12:25	bat	1

12:00 – 12:10	cat	2
12:00 – 12:10	dog	3
12:00 – 12:10	owl	1
12:05 – 12:15	cat	1
12:05 – 12:15	owl	2
12:10 – 12:20	owl	1
12:10 – 12:20	bat	1
12:15 – 12:25	owl	2
12:15 – 12:25	bat	1
12:20 – 12:30	owl	1

Trigger: 5 mins
Window: 10 mins, sliding every 5 mins
Watermark: 10 mins

Solution 1: RocksDB as external state store

One technique to reduce memory usage on the cluster is to save intermediate state off-heap.

```
spark
spark.conf.set(
  "spark.sql.streaming.stateStore.providerClass",
  "com.databricks.sql.streaming.state.RocksDBStateStoreProvider")
```

References:
[Databricks Blog on RocksDB](#)
[Structured Streaming Programming Guide](#)

Solution 2: Watermarking / Late Threshold

Capture "Event Time" on incoming data and set Watermarks.

Purpose:

1. Reclaim memory by purging old state

2. Set expectations on "how late is too late?"

   Event Time < (Max event time seen by the engine – late threshold)

```
Spark
.readStream
.withWatermark("event_timestamp", "10 minutes")
.<source info>
```
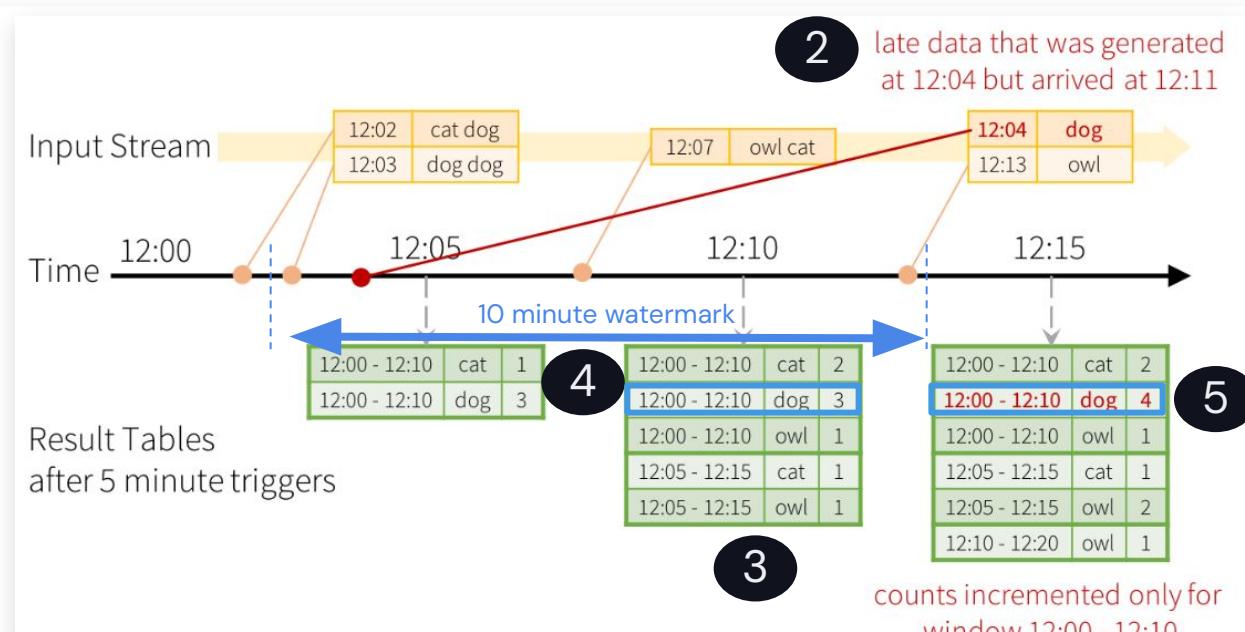
Handling Late Arrival Events with Watermark

Delayed data will be processed if within Watermark threshold – Guaranteed!

1 Configuration:
1. **Trigger**: 5 min
2. **Window**: 10 mins
3. **Sliding**: 5 min
4. **Watermark**: 10 mins
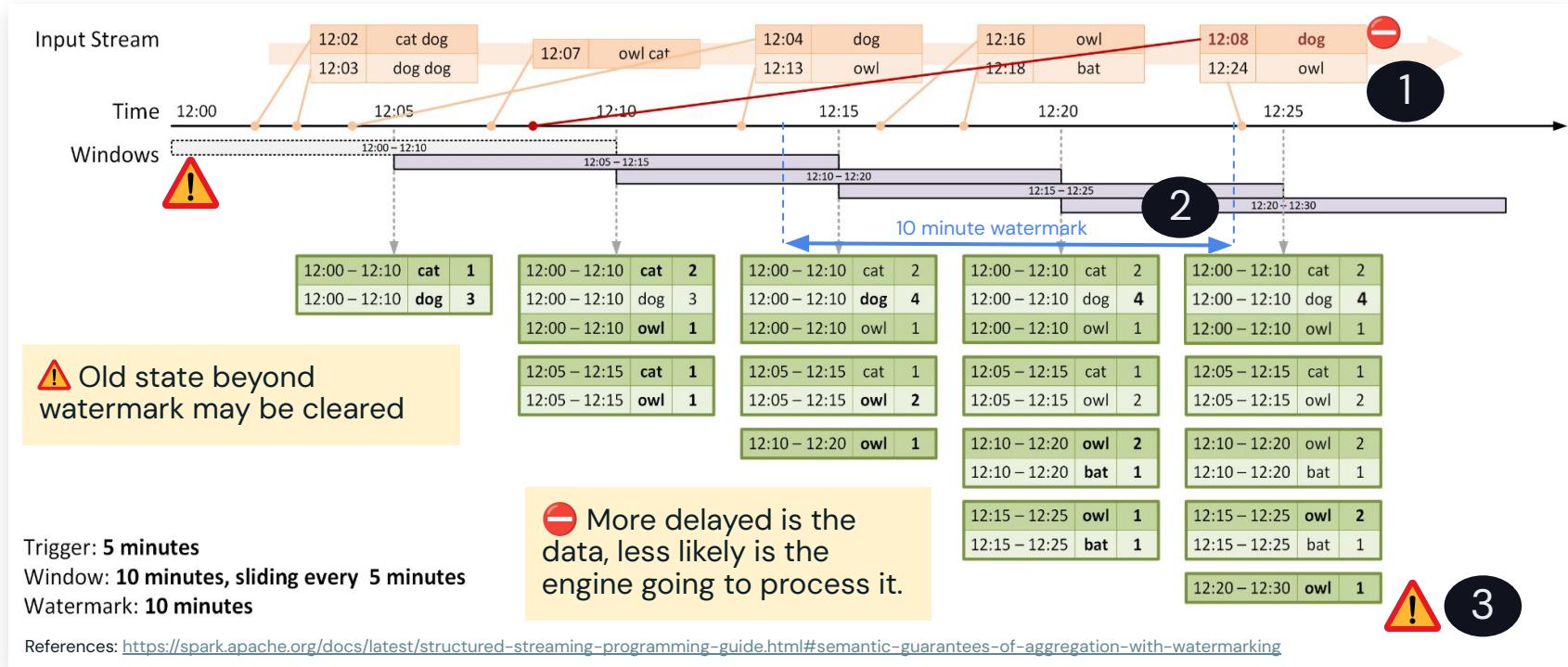
References: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#semantic-guarantees-of-aggregation-with-watermarking

Control size of state

Late data MAY be dropped

databricks
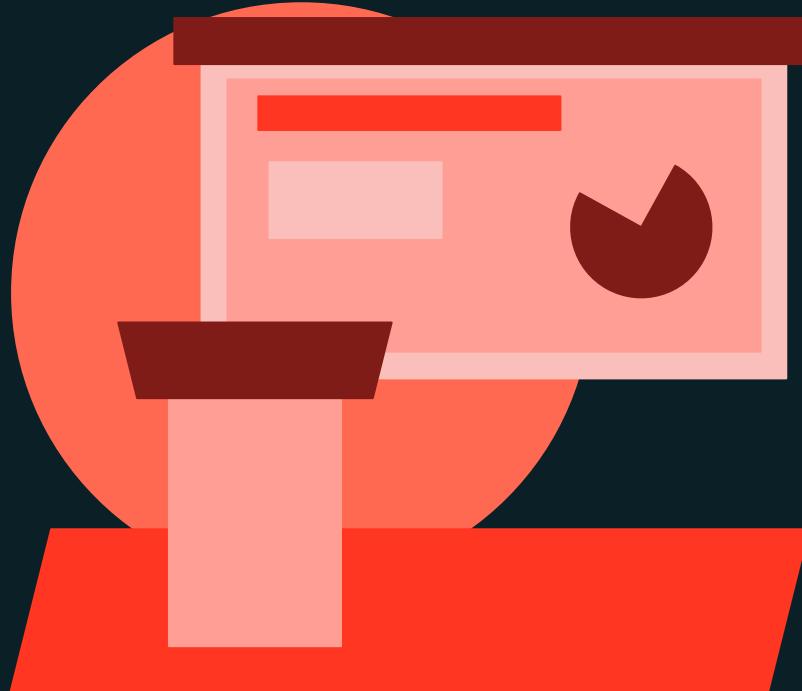
Introduction to Streaming

LECTURE

Trigger Types and Output Modes

Trigger Types

When to stream data in

Default	Process each micro-batch as soon as the previous one has been processed (no coding required)
Fixed Interval	Micro-batch processing kicked off at the user-specified interval: `.trigger(processingTime = '1 second'))`
One-time	Process all available data as a single micro-batch and then automatically stop the query: `.trigger(once = True)`
availableNow	Like **Trigger Once,** available data processed before query stops, but in multiple batches instead of one: `.trigger(availableNow = True)`
Continuous Processing	Long-running tasks that continuously read, process, and write data as soon events are available: `.trigger(continuous = '1 second')`

Output Modes

What to write

```
hello world I say hello
hello goodbye
```

APPEND

Add new records only
(*default*)

value
hello
say
I
say
hello

value
hello
goodbye

UPDATE

Update changed records in place

value	count
hello	2
say	1
world	1
I	1

value	count
hello	2
goodbye	1

COMPLETE

Rewrite full output

value	count
hello	2
say	1
world	1
I	1

value	count
hello	3
say	1
world	1
I	1
goodbye	1

Only rows updated since last trigger written. Different from **Complete** Mode since **Update** Mode outputs only changed rows since last trigger. If query doesn't contain aggregations, **Update** same as **Append** mode

databricks

Introduction to Streaming

LAB EXERCISE

Stream Aggregations Lab
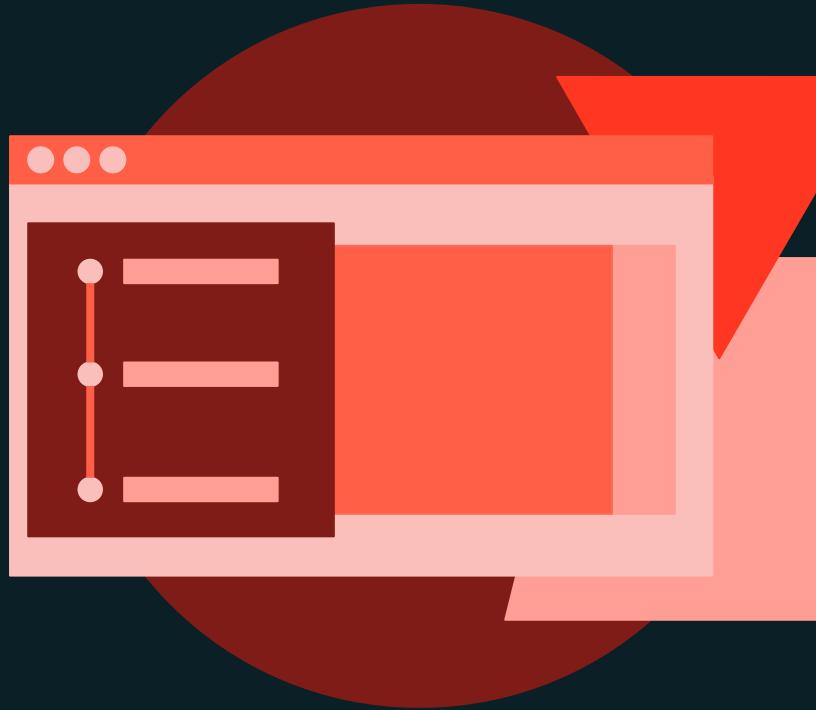
databricks

Introduction to Streaming

Windowed Aggregation with Watermark

Notebook: SDLT 1 – Incremental Processing with Spark Structured Streaming/SDLT 1.4 – Windowed Aggregation with Watermark
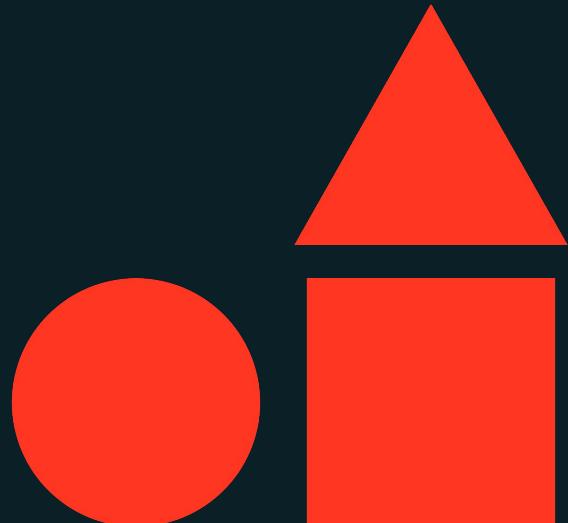
Streaming Joins (Optional)

Databricks Streaming and Lakeflow Spark Declarative Pipelines

databricks

Streaming Joins

Streaming Joins (Optional)

Join Operations

- Structured Streaming supports joining a streaming Dataset/DataFrame with a static Dataset/DataFrame as well as another streaming Dataset/DataFrame.
- The result of the streaming join is generated incrementally, similar to the results of streaming aggregations in the previous examples.
- In all the supported join types, the result of the join with a streaming Dataset/DataFrame will be the exactly the same as if it was with a static Dataset/DataFrame containing the same data in the stream.

Managing Memory Pressure

As the stream runs, the size of streaming state will keep growing indefinitely as all past input must be saved as any new input can match with any input from the past.

Avoid Unbounded State by:

1. Define Watermark delays on both inputs
2. Add **Event-Time Constraints** on join condition

   e.g. `JOIN ON leftTime BETWEEN rightTime AND rightTime + INTERVAL 1 HOUR`

Reference: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#stream-stream-joins

Streaming Joins Supported

Left Side ↓	Right Side →	
	Static	**Stream**
Static	✅All join types supported – Stateless (This is not on streaming join even though it can be present in a streaming query)	✅ Inner Join – Stateless ✅ Right Outer – Stateless ⛔ All Other joins
Stream	✅ Inner Join – Stateless ✅ Left Outer – Stateless ✅ Left Semi – Stateless ⛔All Other joins	✅ Inner Join – Stateful. Watermarks Optional ✅ Left Outer – Stateful. Watermark mandatory on right ✅ Left Semi – Stateful. Watermark mandatory on right ✅ Right Outer – Stateful. Watermark mandatory on left ✅ Full Outer – Stateful. Watermark mandatory on at least side (Specify watermark on both sides + event time constraints for full state cleanup)

Reference: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#support-matrix-for-joins-in-streaming-queries
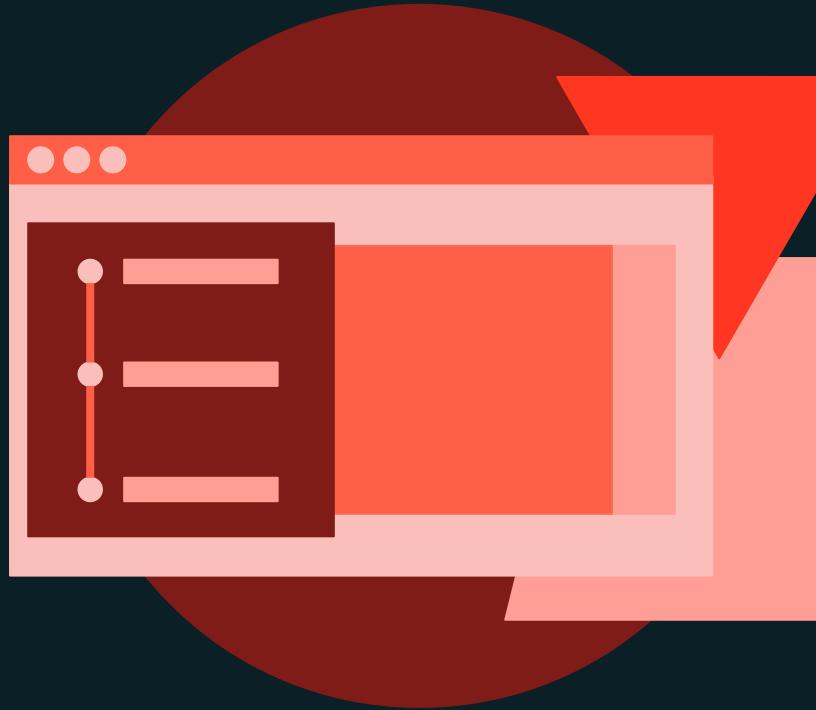
databricks

Streaming Joins

DEMONSTRATION

Stream/Stream Joins (Optional)

Notebook: SDLT 1 – Incremental Processing with Spark Structured Streaming/SDLT 1.5 – Optional – Stream–Stream Joins
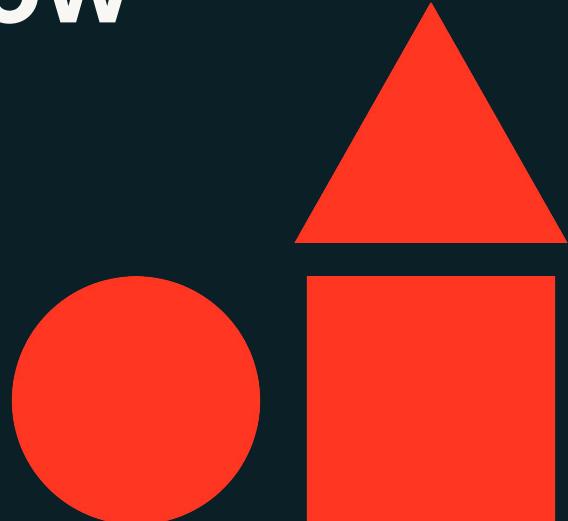
databricks
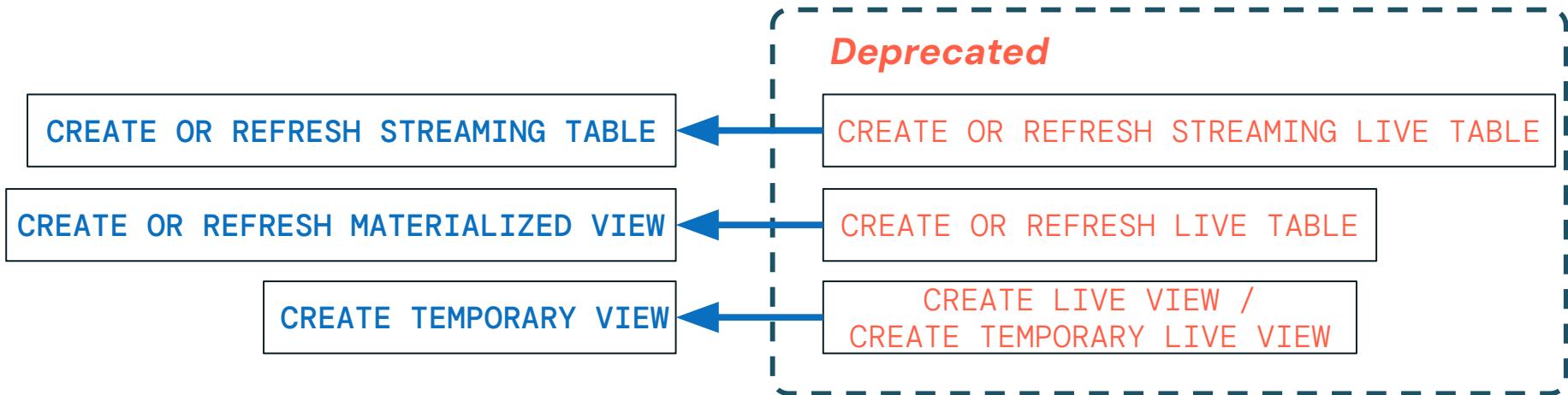
Streaming ETL Patterns with Lakeflow Spark Declarative Pipelines

Databricks Streaming and Lakeflow Spark Declarative Pipelines

Lakeflow Spark Declarative Pipelines Terminology

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

Deprecated

`CREATE OR REFRESH STREAMING TABLE` ← `CREATE OR REFRESH STREAMING LIVE TABLE`

`CREATE OR REFRESH MATERIALIZED VIEW` ← `CREATE OR REFRESH LIVE TABLE`

`CREATE TEMPORARY VIEW` ← `CREATE LIVE VIEW / CREATE TEMPORARY LIVE 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

databricks
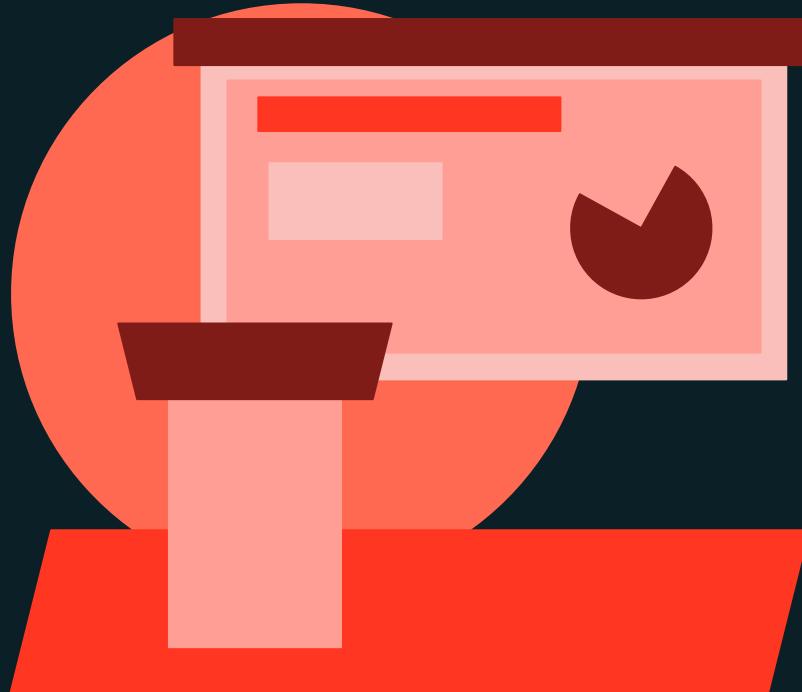
Streaming ETL Patterns with Lakeflow Spark
Declarative Pipelines

LECTURE

Data Ingestion Patterns

Why Do We Need These Patterns?
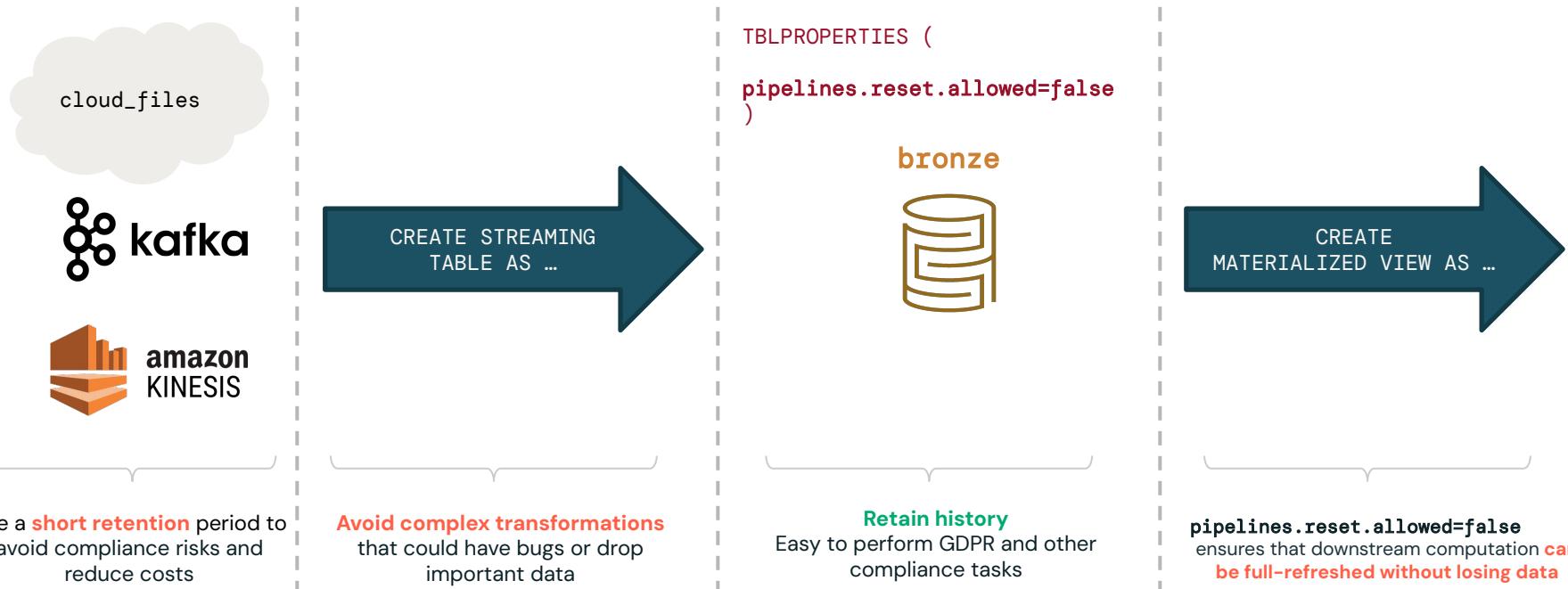
Limitations at Data Ingestion Stage

- 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
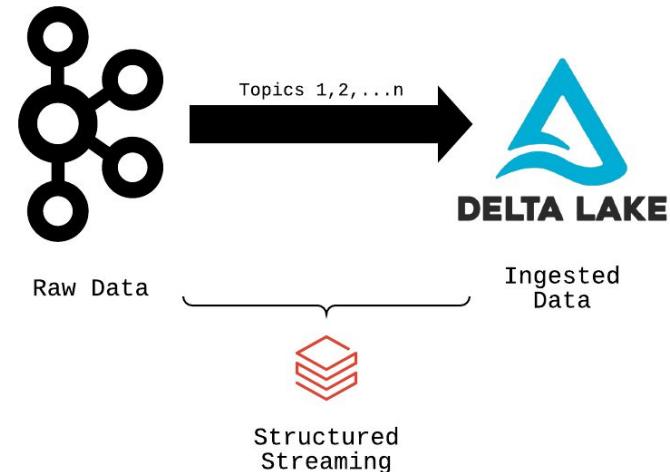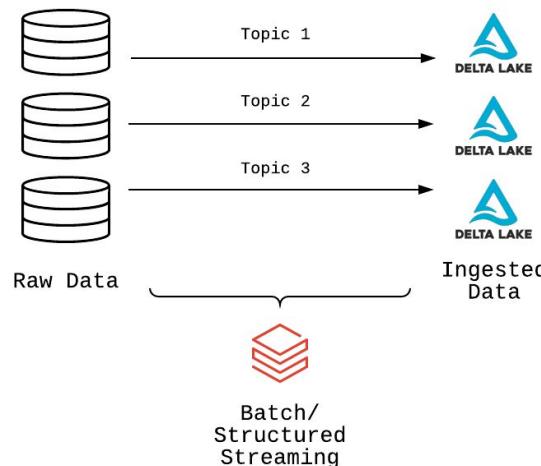
Delta provides cheap, elastic and governable storage for transient sources


```
cloud_files
```

kafka

amazon KINESIS

CREATE STREAMING TABLE AS …

```
TBLPROPERTIES (

pipelines.reset.allowed=false

)
```

bronze

CREATE MATERIALIZED VIEW AS …

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

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

Pattern 2: Multiplex Ingestion

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

- First, let's discuss Simplex vs. Multiplex patterns
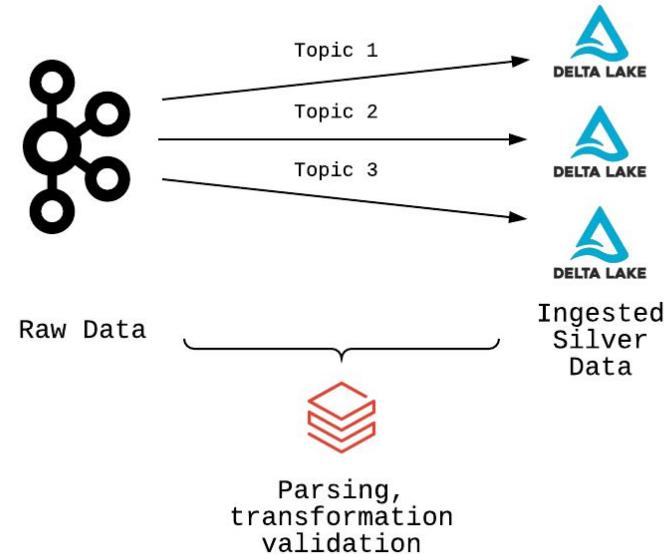
Pattern 2: Multiplex Ingestion

Anti–Pattern: Using Kafka as Bronze Table
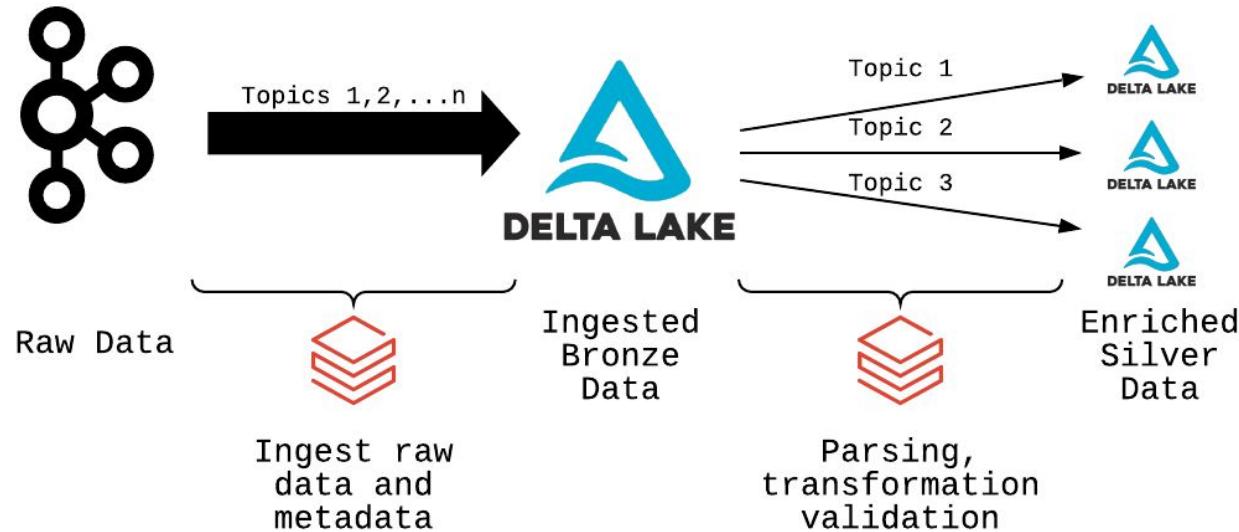
Don't use Kafka as Bronze Table:

- 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 2: Multiplex Ingestion Pattern

Multiplexing is used when a set of independent streams all share the same source
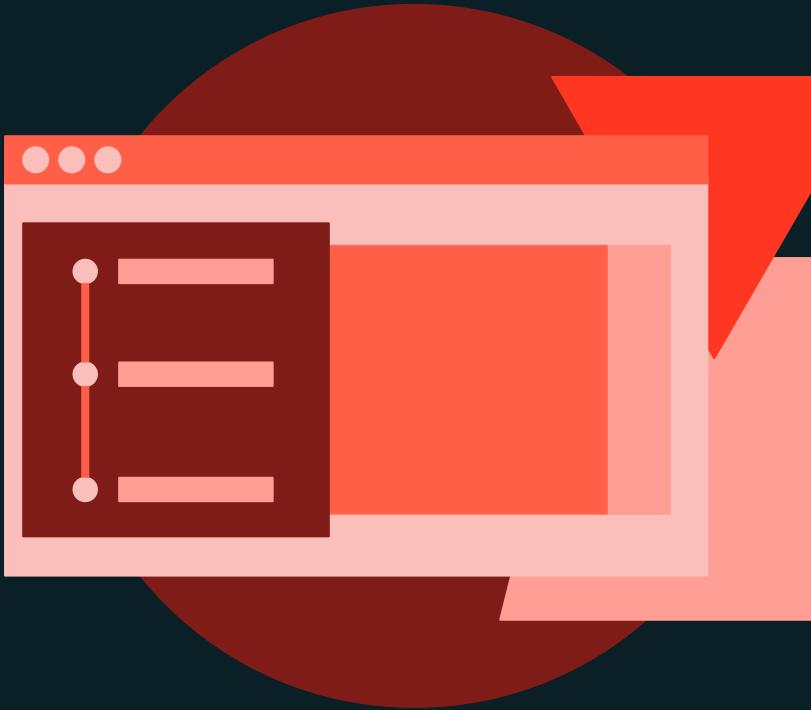
Streaming ETL Patterns with Lakeflow Spark Declarative Pipelines

DEMONSTRATION

Auto Load to Bronze

**Notebook**: SDLT 2 – Streaming ETL Patterns with Lakeflow Spark Declarative Pipelines/SDLT 2.1 – Auto Load to Bronze
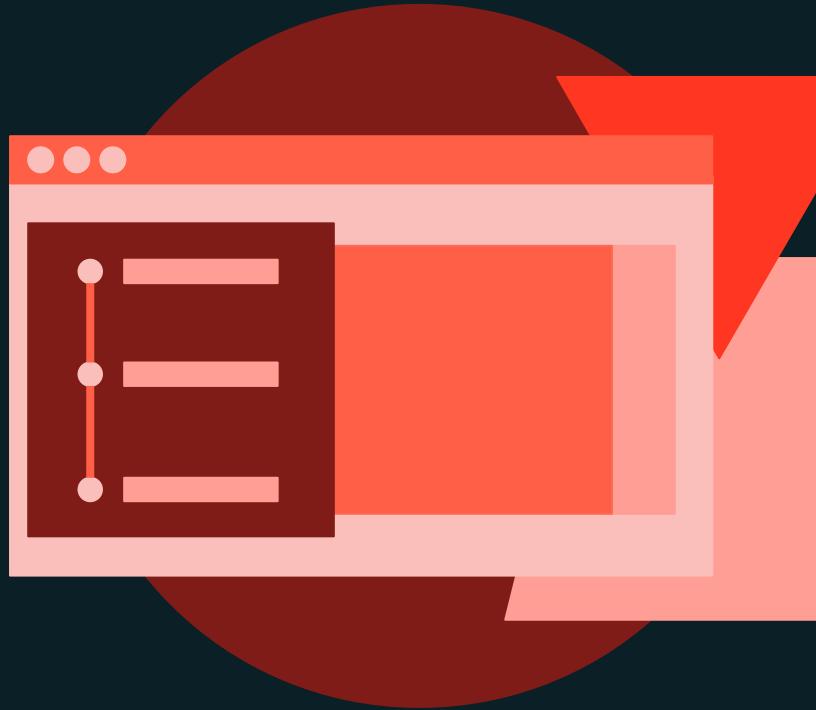
databricks

Streaming ETL Patterns with Lakeflow Spark
Declarative Pipelines
DEMONSTRATION

Stream from Multiplex Bronze
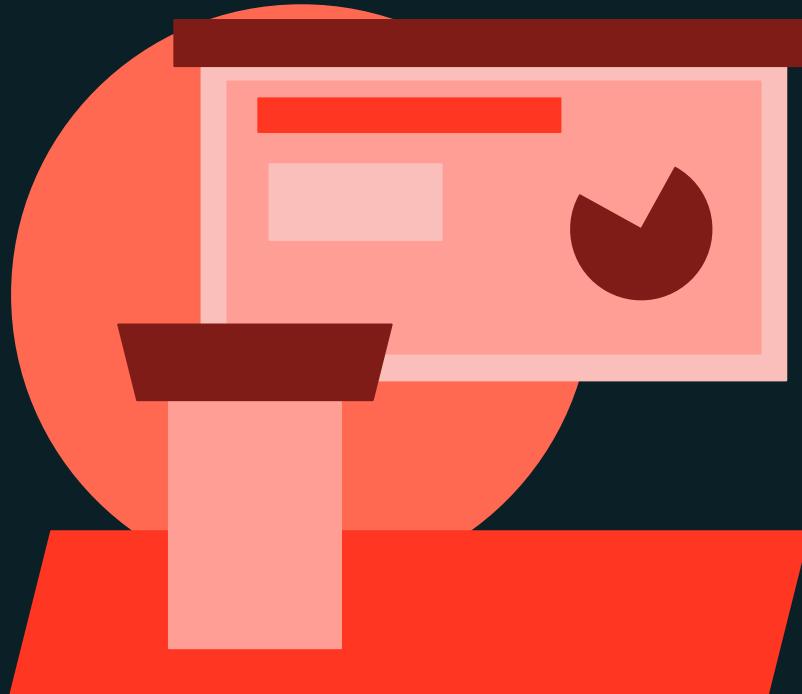
databricks

Streaming ETL Patterns with Lakeflow Spark
Declarative Pipelines

Data Quality Enforcement

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

Records meeting expectations (V=1)

Records meeting inverse expectations (V=0)

Limitations:

- Processes the data twice

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


Partition by V

Limitations:

- Doesn't use expectations; data quality metrics are not available in the event logs or the pipelines UI.
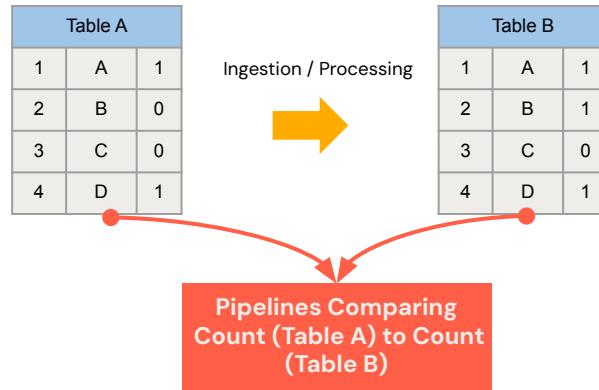
Pattern: Verify Data with Row Comparison

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 Lakeflow Spark Declarative Pipelines UI.

Table A		
1	A	1
2	B	0
3	C	0
4	D	1

Ingestion / Processing

Table B		
1	A	1
2	B	1
3	C	0
4	D	1

Pipelines Comparing Count (Table A) to Count (Table B)

1: Lakeflow Pipeline SQL

```
CREATE OR REFRESH MATERIALIZED VIEW count_verification(
  CONSTRAINT no_rows_dropped EXPECT (a_count == b_count)
)
AS
SELECT *
FROM
  (SELECT COUNT(*) AS a_count FROM tbla) a,
  (SELECT COUNT(*) AS b_count FROM tblb) b;
```

Pattern: Define Tables for Adv. Validation

Perform advanced data validation with Pipeline 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 a Lakeflow Declarative Pipeline using aggregate and join queries and use the results of those queries as part of your expectation checking.
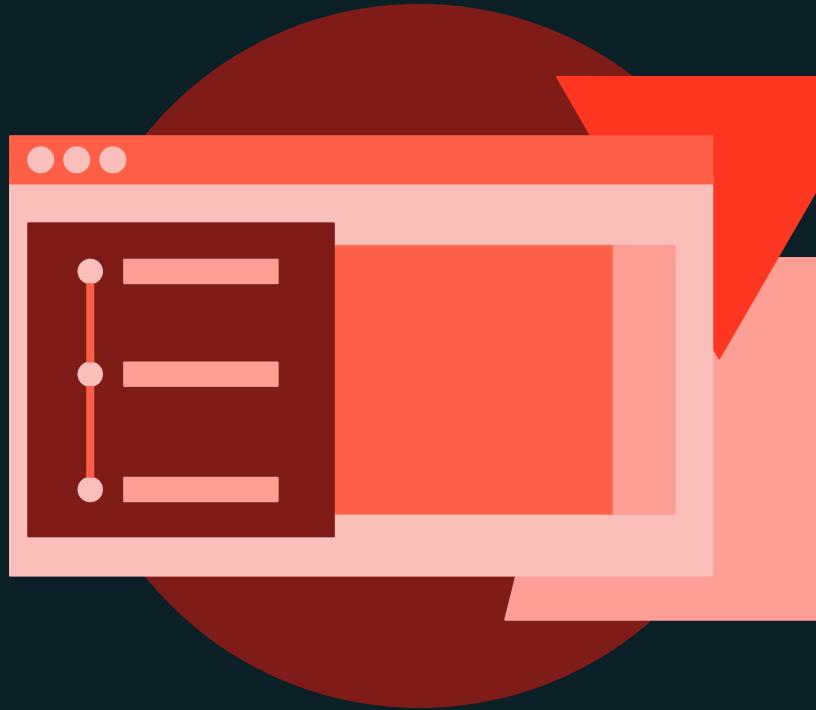
```sql
CREATE OR REFRESH MATERIALIZED VIEW compare_tests (
  CONSTRAINT no_missing_records EXPECT (r.key IS NOT NULL)
)
AS
SELECT *
FROM validation_copy v
LEFT OUTER JOIN report r
  ON v.key = r.key;
```

![databricks]

Streaming ETL Patterns with Lakeflow Spark Declarative Pipelines

DEMONSTRATION

Data Quality Enforcement

databricks

Streaming ETL Patterns with Lakeflow Spark
Declarative Pipelines

Streaming ETL Lab

Questions?