databricks

Incremental Processing with Structured Streaming

Databricks Academy
2023

Agenda

Incremental Processing with Spark Structured Streaming and Delta Lake

Lesson Name	Lesson Name
Lecture: Streaming Data Concepts	Lecture – Aggregations, Time Windows, Watermarks
Lecture: Introduction to Structured Streaming	ADE 1.3L – Stream Aggregations Lab
ADE 1.1 – Follow Along Demo – Reading from a Streaming Query	Lecture: Delta Live Tables Review
ADE 1.2L – Streaming Query Lab	Lecture: Auto Loader

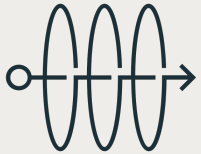
Streaming Data Concepts

What is streaming data?

Continuously generated and unbounded data

Typical data sources

DB change data feeds



Clickstreams

Machine & application logs

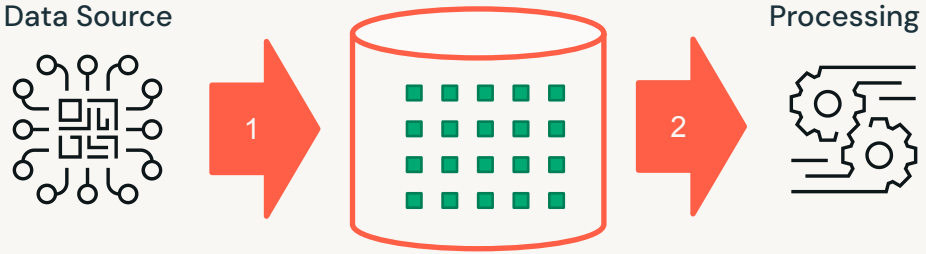
Application events

Mobile & IoT data

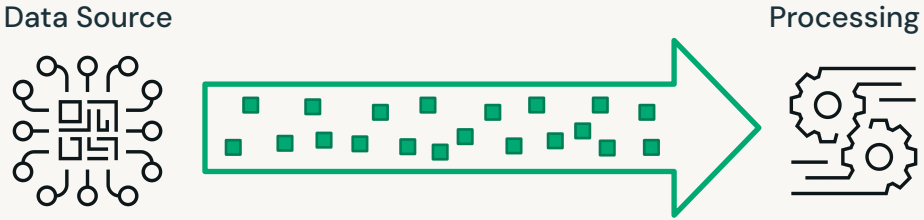
The vast majority of the data in the world is streaming data!

What is stream processing?

Traditional batch-oriented data processing is one-off and bounded.



Stream processing is continuous and unbounded



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

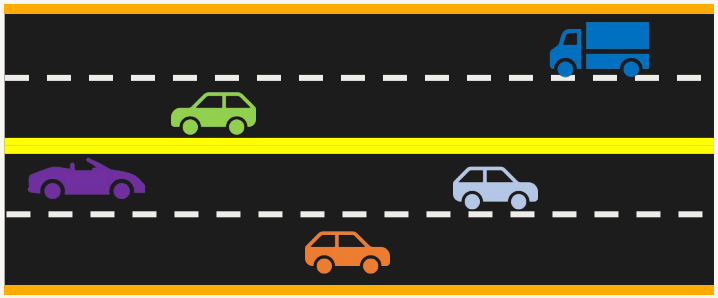
Bounded vs. Unbounded Dataset

Bounded Data

- Has a finite and unchanging structure at a the time of processing.
- The order is static.
- **Analogy:** Vehicles in a parking lot.

Unbounded Data

- Has an infinite and continuously changing structure at the time of processing.
- The order not always sequential.
- **Analogy:** Vehicles on a highway



Batch vs. Stream Processing

Batch Processing

- Generally refers to processing & analysis of ***bounded*** datasets (ie. size is well known, we can count the number of elements, etc.)
- Typical of applications where there are loose data latency requirements (ie. day old, week old, month old).
- This was traditional ETL from transactional systems into analytical systems.

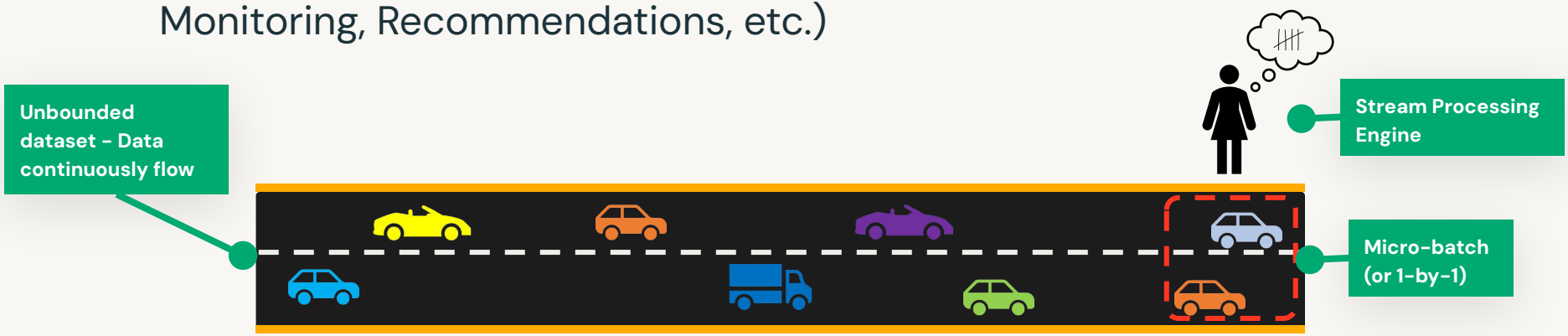
Batch Processing

Traditional data processing pipeline

Batch vs. **Stream** Processing

Stream Processing

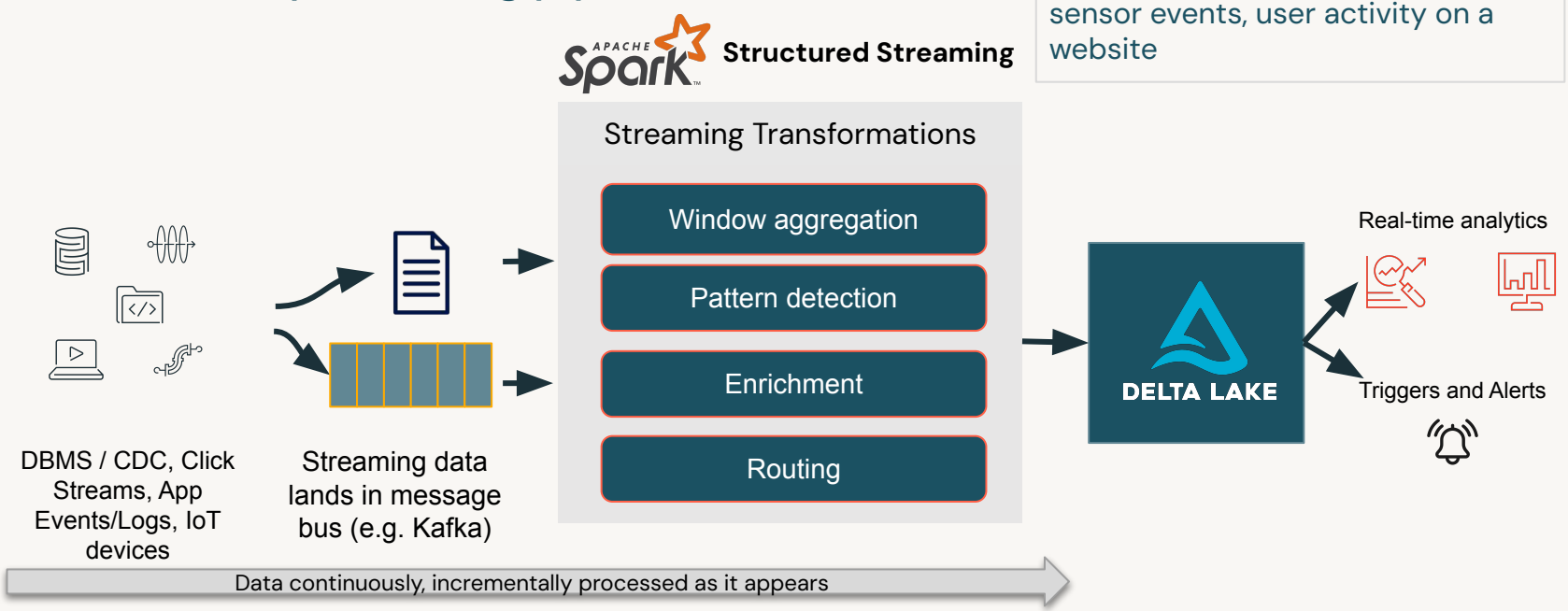
- Datasets are **continuous and unbounded** (data is constantly arriving, and must be processed as long as there is new data)
- Enables low-latency use cases (ie. real-time, or near real-time)
- Provides fast, actionable insights (ie. Quality-of-Service, Device Monitoring, Recommendations, etc.)



Stream Processing

Modern data processing pipeline

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



Stream Processing vs. Batch Processing

Similarities and differences between Stream and Batch Processing

Differences:

How to process in one run?	Batch processing engine	Stream processing engine
Bounded dataset	Big batch	Row by row / mini-batch
Unbounded dataset	NA (multiple runs)	Row by row / mini-batch
Query computation	Only once	Multiple

Similarities:

- Both have data transformation
- Output of streaming job is often queried in batch jobs
- Stream processing often include batch processing (micro-batch)

Advantages of Stream Processing

Why use streaming (vs. batch) ?

A more intuitive way of capturing and processing continuous and unbounded data

Lower latency for time sensitive applications and use cases

Better fault-tolerance through checkpointing

Automatic bookkeeping on new data

Higher compute utilization and scalability through continuous and incremental processing

Challenges of Stream Processing

Stream processing is not easy

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

Introduction to Structured Streaming

What is Structured Streaming

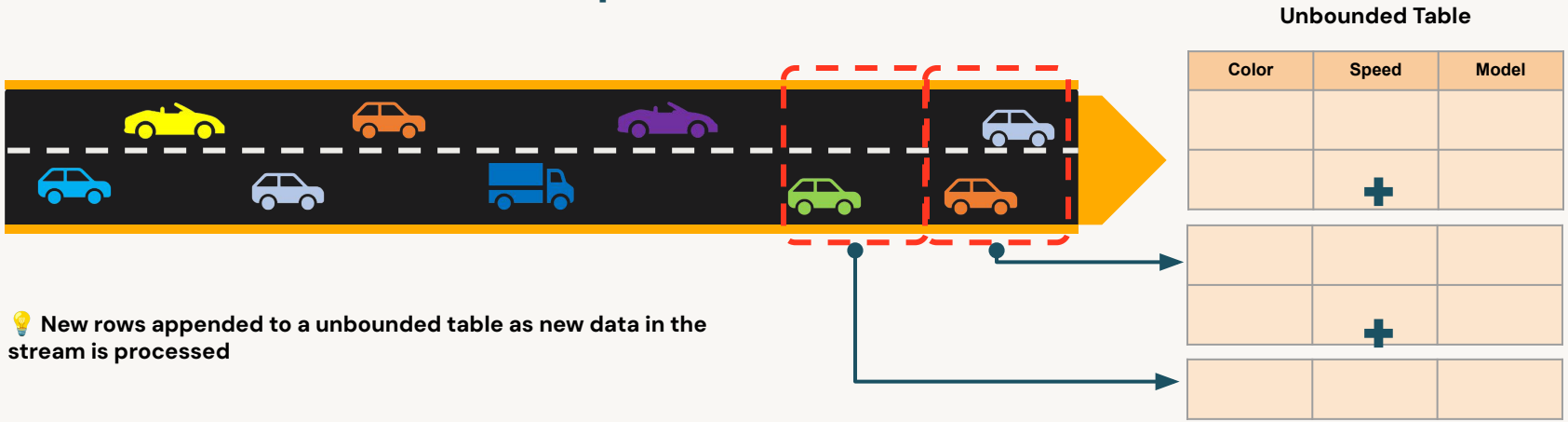
Apache Spark Structured Streaming Basics

- 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.

How Structured Streaming Works

Incremental Updates - Data stream as an unbounded table

- Streaming data is usually coming in very fast.
- The magic behind Spark Structured Streaming: **Processing infinite data as an incremental table updates**.

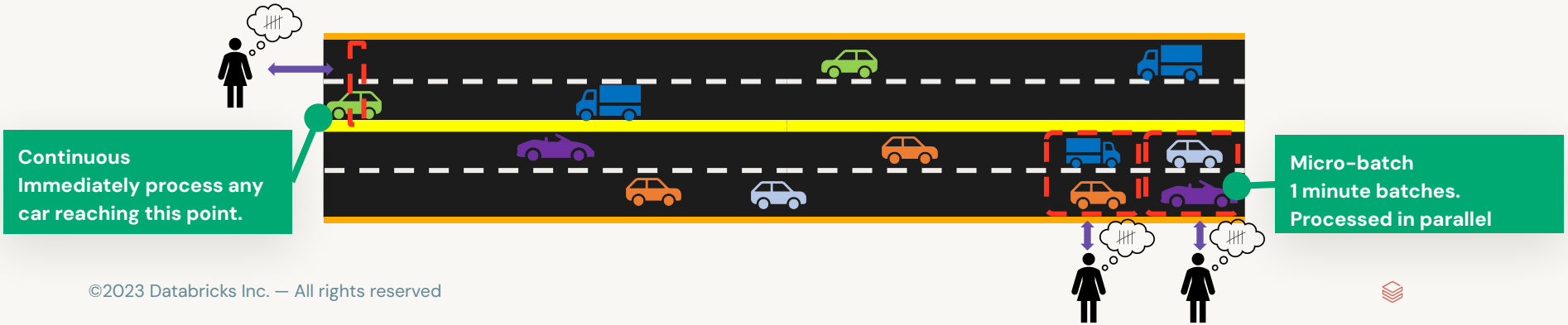


💡 **New rows appended to a unbounded table as new data in the stream is processed**

How Structured Streaming Works

Micro-Batch Processing

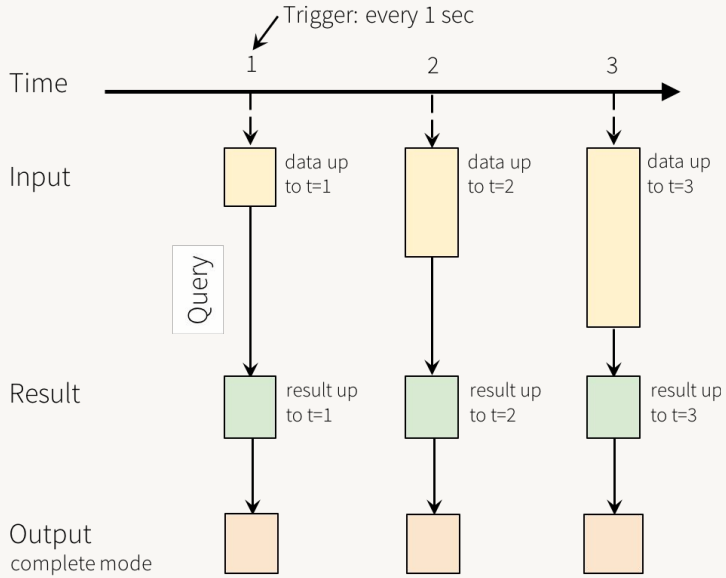
- **Micro-batch Execution:** Accumulate small batches of data and process each batch in parallel.
- **Continuous Execution (EXPERIMENTAL):** Continuously listen for new data and process them individually.



How Structured Streaming Works

Execution mode

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

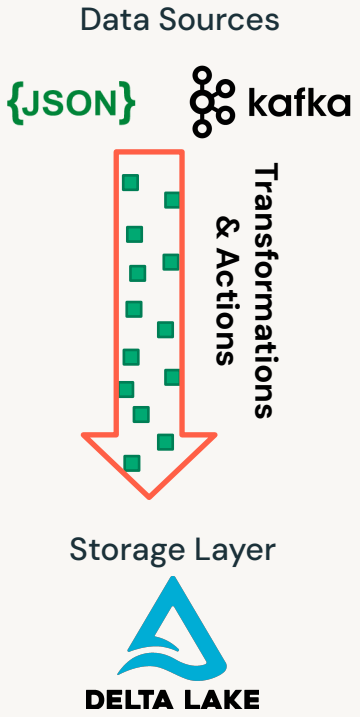


Programming Model for Structured Streaming

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
  - Sinks
  - Transformations & actions
  - Triggers



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

Anatomy of a Streaming Query

Structured Streaming **Core Concepts**

Trigger Types:

Fixed interval micro batches	`.trigger(processingTime = "2 minutes")`	Micro-batch processing kicked off at the *user-specified interval*
Triggered One-time micro batch	`.trigger(once=True)`	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")`	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**

```
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", "...")
  .outputMode("complete")
  .start()
```

Output Mode:

- Defines how the data is written to the sink
- Equivalent to “save” mode on static DataFrames

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.

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", "...")
  .outputMode("append")
  .start()
```

Raw data from Kafka available as structured data in seconds, ready for querying

Benefits of Structured Streaming

Unification

Unified API for Batch and Stream Processing

- Same API is used for batch and stream processing.
- Supports Python, SQL or Spark's other supported languages.
- Spark's built-in libraries can be called in a streaming context, including ML libraries.

Note: Most operations on a streaming DataFrame are identical to a static DataFrame. There are some exceptions to this, for example, sorting is not supported with streaming data.

Fault Tolerance

End-to-end fault tolerance

- Structured Streaming ensures end-to-end exactly-once fault-tolerance guarantees through **checkpointing**.
- In case of failures; the streaming engine attempts to restart and/or reprocess the data.
- This approach requires;
  - **Replayable streaming** source such as cloud-based object storage and pub/sub services.
  - **Idempotent sinks** - multiple writes of the same data (as identified by the offset) do not result in duplicates being written to the sink.

Handle Out-of-Order Data

Support for “event time” to aggregate out of order data

- Supports event-time-window-based aggregation queries
- Supports watermarking which allows users to the threshold of late data

Structured Streaming with Delta Lake

Delta Lake Benefits

An open format storage layer built for the lakehouse architecture

Extensibility

Streaming DataFrame API — Open file access — SQL API — Python API

Reliability

- ACID transactions
- Schema Enforcement & Evolution
- Rollbacks
- Time travel

Performance

- Advanced indexing with Z-order
- Caching
- Auto tuning of storage block sizes
- Data skipping

Governance

- Integrated with data catalog
- Audit history
- GDPR and CCPA compliance
- Data Pseudonymization

Flexibility

- Open format built on Parquet
- Streaming + Batch
- Unstructured data types
- Easy replication with Delta Clones

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

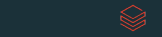
- You can limit the input rate for micro-batches by setting DataStreamReader options:
  - maxFilesPerTrigger: Maximum files read per micro-batch (default 1,000)
  - maxBytesPerTrigger: Soft limit to amount of data read per micro-batch (no default)
  - Note: Delta Live Tables pipelines auto-tune options 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

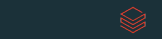
- Each micro-batch written to the Delta table is committed as a new version.
- Delta Lake supports both append and complete output modes.
  - Append is most common.
  - Complete replaces the entire table with each micro-batch. It can be used for streaming queries that perform arbitrary aggregations on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

Demo: Reading from a Streaming Query



Lab: Streaming Query

Aggregations, Time Windows, Watermarks

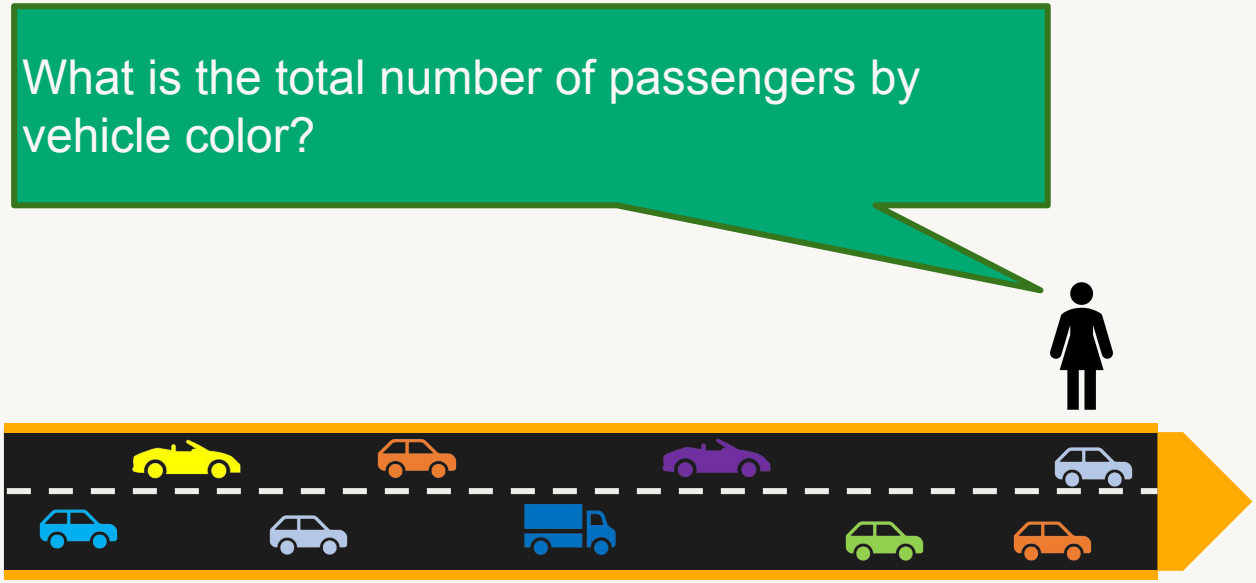


Types of Stream Processing

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

Stream Aggregations



Stream Aggregations

- Continuous applications often require near real-time decisions on real-time, aggregated statistics
  - Examples: Aggregating errors from IoT devices, behavior analysis on instant messages via hashtags
- In the case of streams, you generally don't want to run aggregations over the entire dataset. Why;
  - There conceptually is no end to the flow of data, data is continuous
  - The size of the dataset grows in perpetuity; will eventually run out of resources
- Solution: Instead of aggregating over the entire dataset, we can aggregate over data grouped by windows of time (say, every 5 minutes). This is referred to as windowing.

Time Based Windows

Tumbling window vs. Sliding window

Tumbling Window

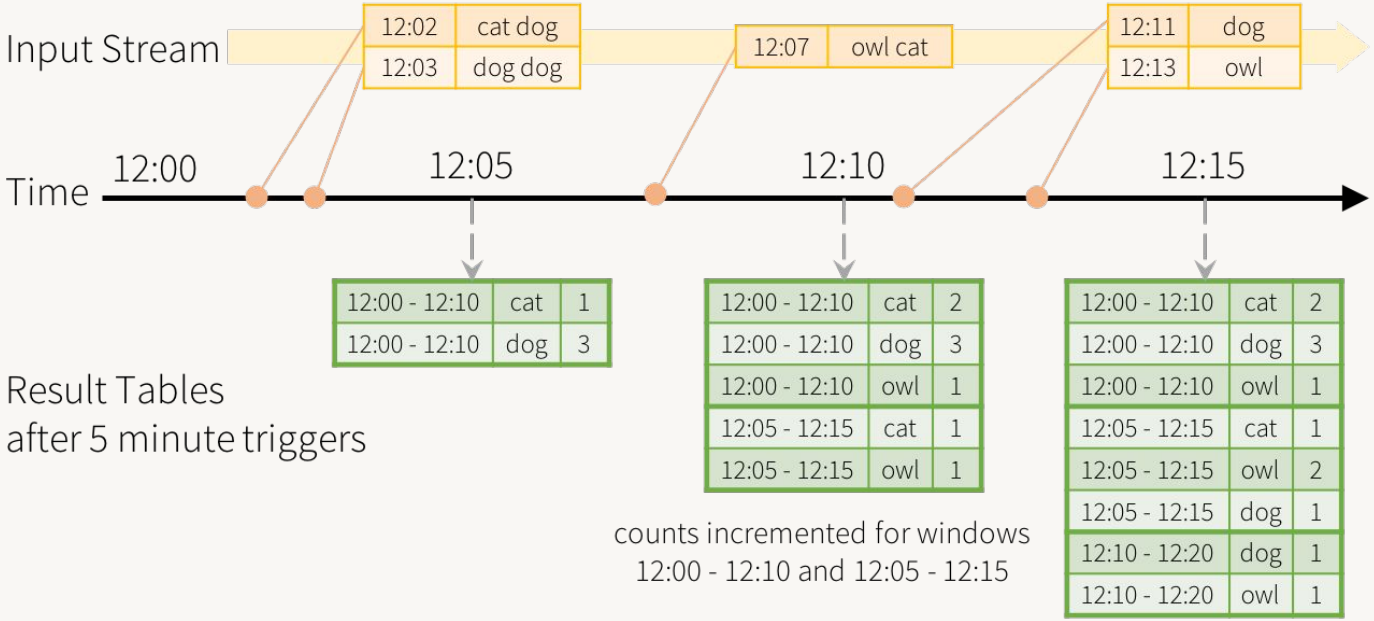
- **No window overlap**
- Any given event gets aggregated into **only one** window group (e.g. 1:00–2:00 am, 2:00–3:00 am, 3:00-4:00 am, ...)

Sliding Window

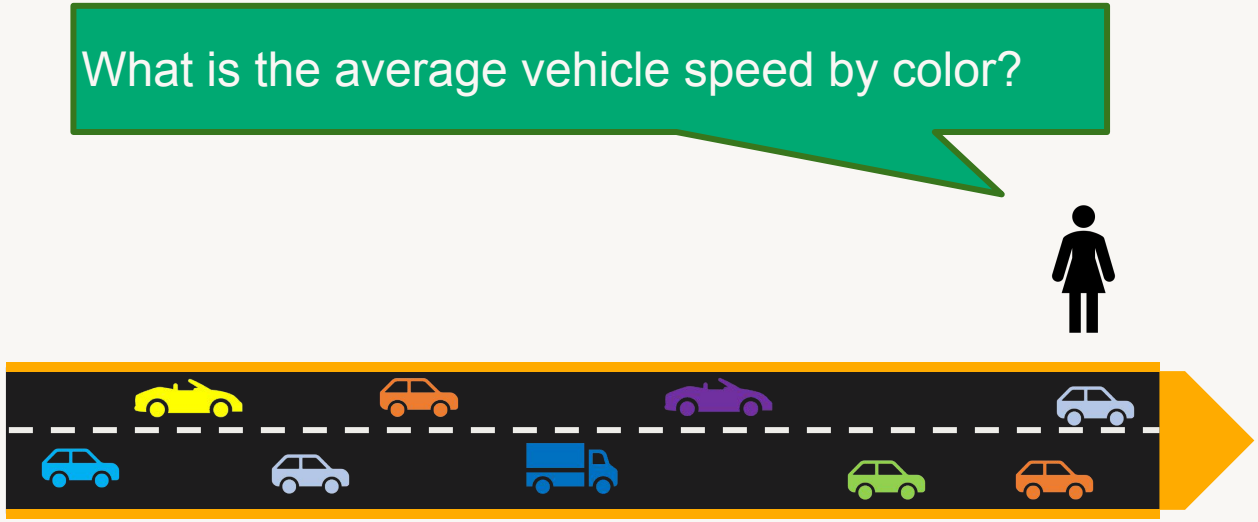
- **Windows overlap**
- Any given event gets aggregated into **multiple window** groups (e.g. 1:00-2:00 am, 1:30–2:30 am, 2:00–3:00 am, ...)

Time Based Windows

Sliding window example

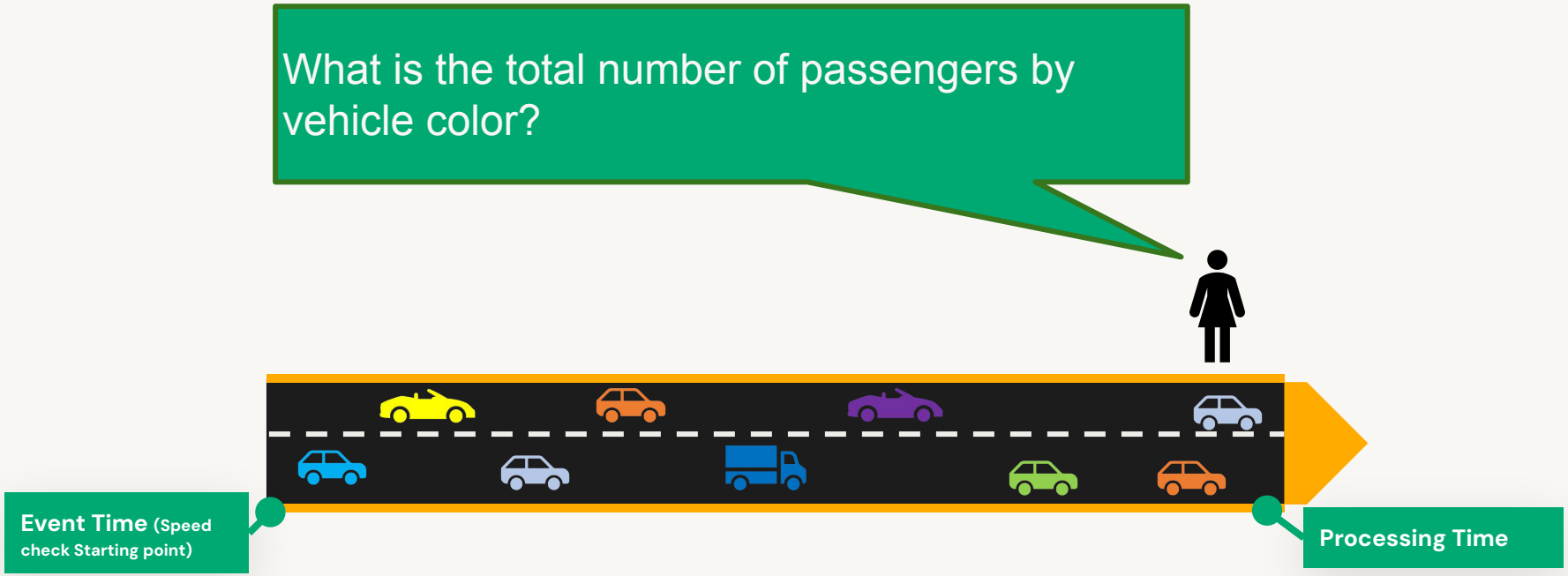


Reasoning About Time



Reasoning About Time

Event time vs. Processing time



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 Domain Skew

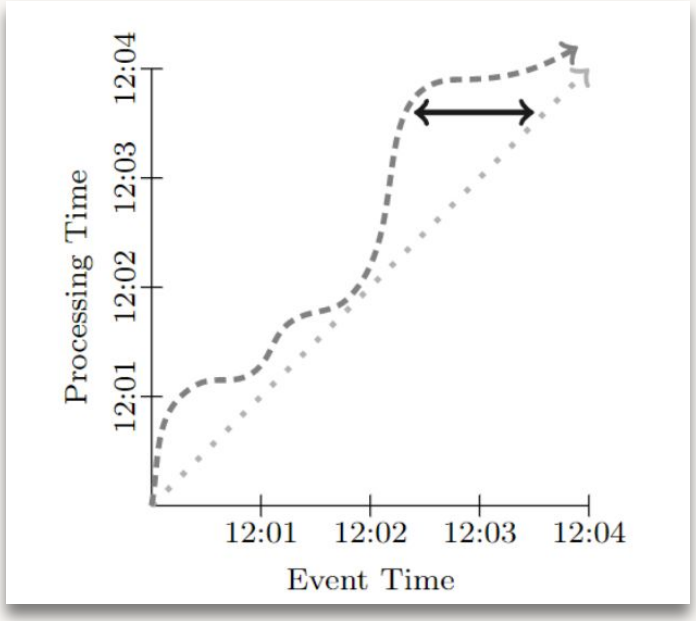
Time in batch vs. stream processing

When batch processing:

- **Processing time** per definition much later (e.g. an hour or day) than **event time**
- Data assumed to be complete (or settle for incompleteness)

When stream processing:

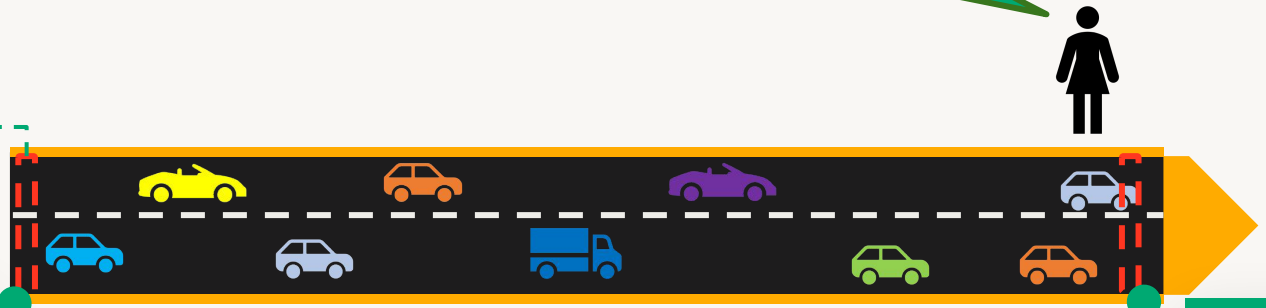
- **Processing time** >= **event time** but often close (e.g. seconds, minutes)
- Challenge when **processing time >>> event time** (late data): not able to conclude anything easily, how long to wait for the data to be complete?



Reasoning About Time

What is the total number of passengers by vehicle color?

Every vehicle's speed is recorded and sent to processing point



Event Time (Speed check Starting point)

Processing Time

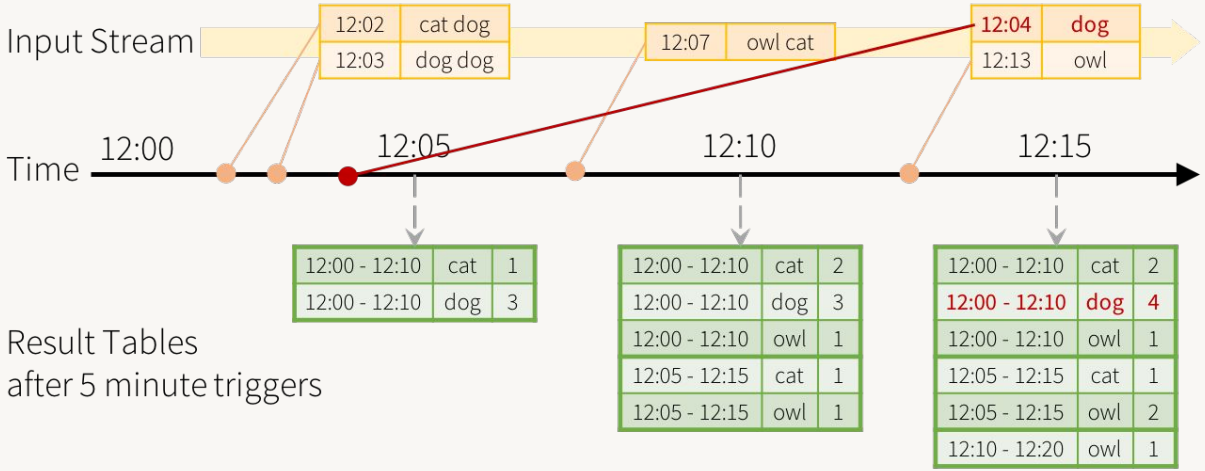
Let's say using 5 minute windowing, what if a vehicle is very slow? Do we need to wait for it?

Reasoning About Time

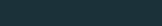
Handling Late Data and Watermarking

Watermark: Handle late data and limit how long to remember old data

- Analogy: Highway **minimum speed** limit

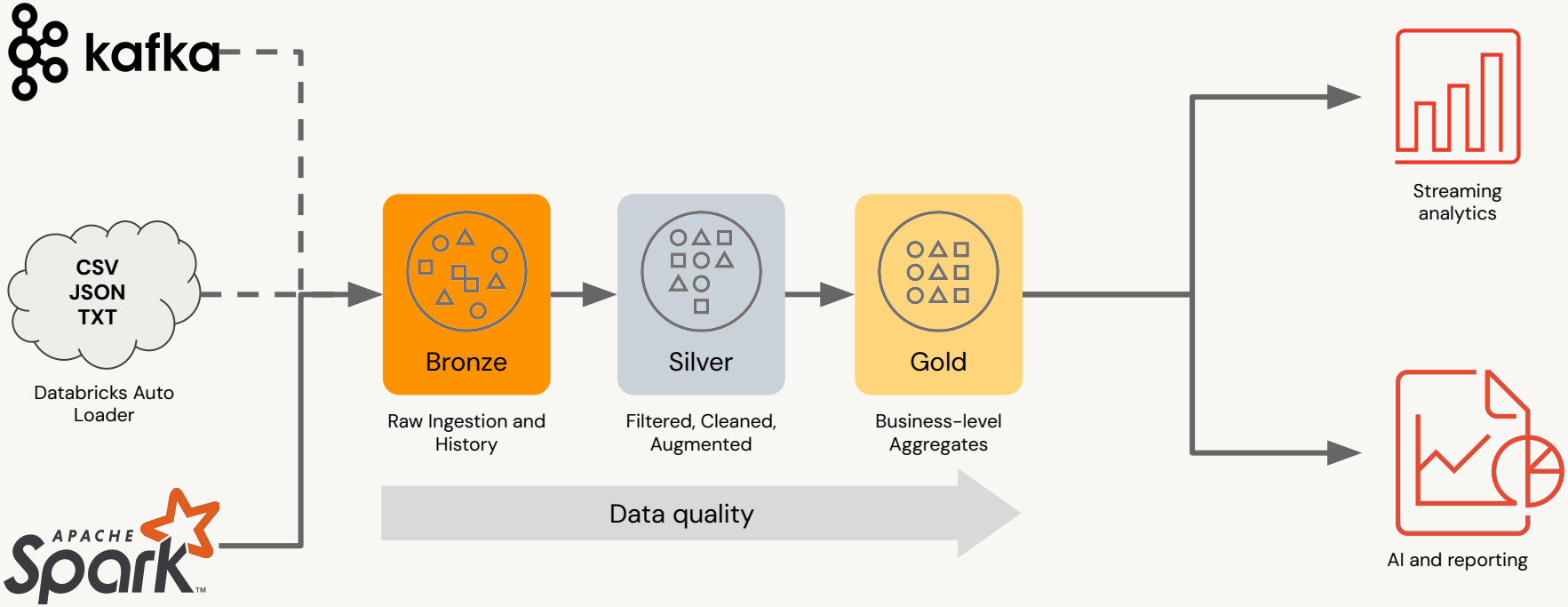


ADE 1.3L – Streaming Aggregation

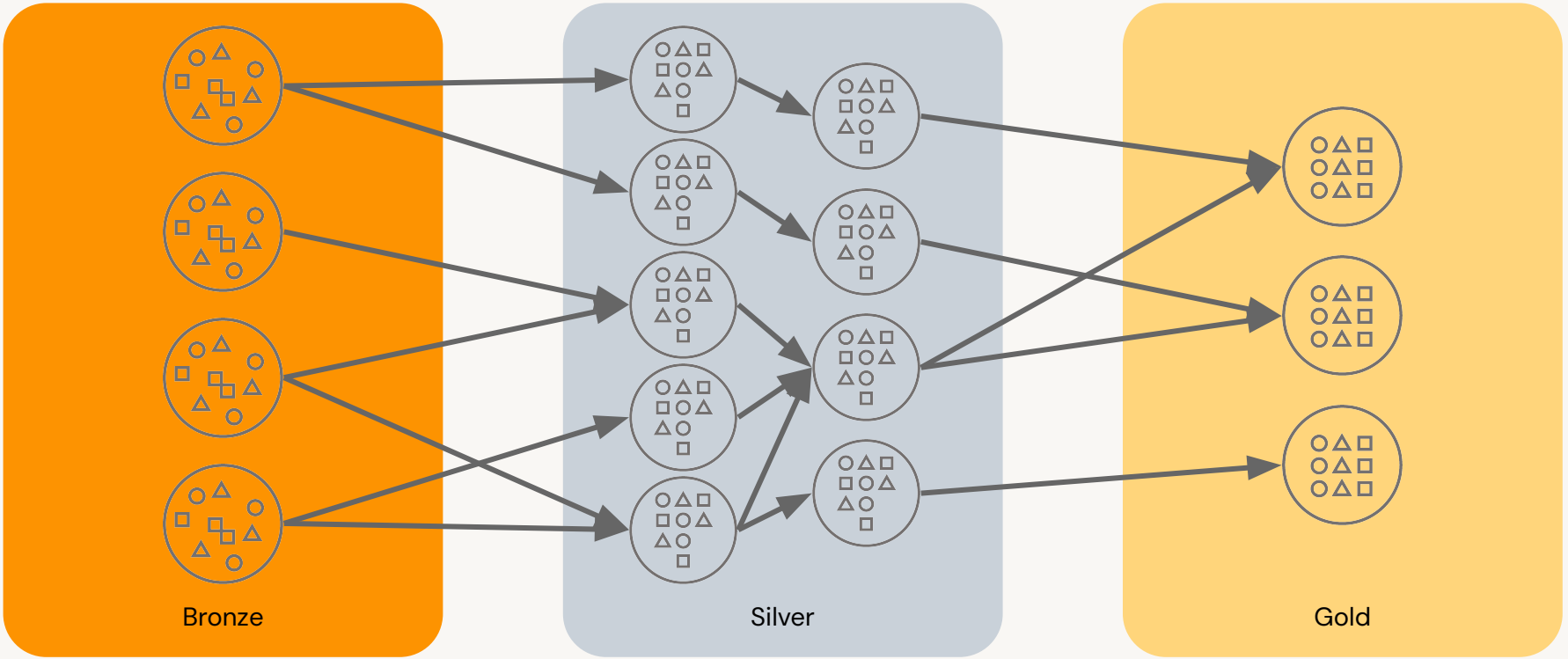


Delta Live Tables Review

Multi-Hop in Databricks



The Reality is Not so Simple



Large scale ETL is complex and brittle

Complex pipeline development

Hard to build and maintain table **dependencies**

Difficult to switch between **batch** and **stream** processing

Data quality and governance

Difficult to monitor and enforce **data quality**

Impossible to trace data **lineage**

Difficult pipeline operations

Poor **observability** at granular, data level

Error handling and **recovery** is laborious

Introducing Delta Live Tables

Make reliable ETL easy on Delta Lake

Operate with agility

Declarative tools to build batch and streaming data pipelines

Trust your data

DLT has built-in declarative quality controls

Declare quality expectations and actions to take

Scale with reliability

Easily scale infrastructure alongside your data

Delta Live Tables

Streaming data ingestion and transformation made simple

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


CREATE LIVE TABLE clean_data
AS SELECT …
FROM LIVE.raw_data
```

Accelerate ETL development
Declare SQL/Python and DLT automatically orchestrates the DAG, handles retries, changing data

Automatically manage your infrastructure
Automates complex tedious activities like **recovery**, **auto-scaling**, and **performance optimization**

Ensure high data quality
Deliver reliable data with built-in **quality controls**, **testing**, **monitoring** and **enforcement**

Unify batch and streaming
Get the simplicity of SQL with the freshness of streaming with one unified API

What is a Live Table?

Live Tables are materialized views for the data intelligence platform.

A live table is:

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

```
CREATE OR REFRESH 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.

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:

Auto Loader

Auto Loader Benefits

Highly Scalable

Discover billions of files efficiently

Cost Effective

Avoid costly LIST operations with the file notification mode

Highly Performant

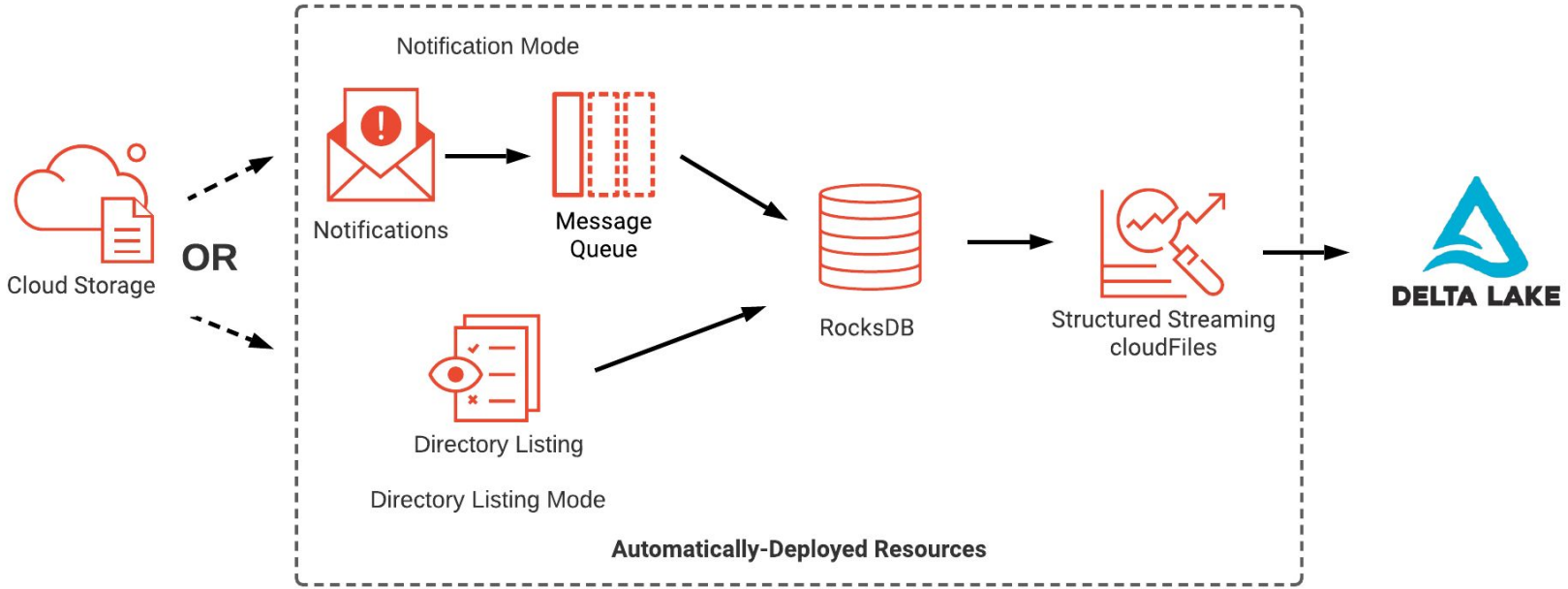
Optimized file discovery with the directory listing mode

Schema Inference & Evolution

Detect schema drifts and rescue data automatically

Auto Loader Under the Hood

```
.option("cloudFiles.useNotifications","true")
```

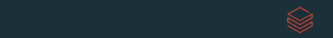


Auto Loader Best Practices

- Leverage **incremental listing** for directory listing mode
  - Files must be lexicographically ordered
  - Determination is automatic
    - `cloudFiles.useIncrementalListing = "auto"` is default
  - DBR 9.1+
  - Use **file notification** mode If incremental listing is not possible

- Consider processing delays while configuring lifecycle policies on object storage services

Knowledge Check



Which of the following could be used as sources in a stream processing pipeline?

Select two responses

A. change data capture (CDC) feed
B. Kafka
C. Delta Lake
D. IoT devices

Which of the following statements about propagating deletes with change data feed (CDF) are true?

Select two responses

A. Deletes cannot be processed at the same time as appends and updates.
B. Commit messages can be specified as part of the write options using the **`userMetadata`** option.
C. Deleting data will create new data files rather than deleting existing data files.
D. In order to propagate deletes to a table, a **`MERGE`** statement is required in SQL.

Which of the following are considerations to keep in mind when choosing between micro-batch and continuous execution mode?

Select two responses.

A. Desired latency
B. Total cost of operation (TCO)
C. Maximum throughput
D. Cloud object storage

Which of the following functions completes the following code snippet to return a Spark DataFrame in a structured streaming query?

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

  _____
```

Select one response.

A. .load()
B. .print()
C. .return()
D. .merge()

In stream processing, datasets are _____.

Select one response

A. continuous and bounded
B. continuous and unbounded
C. micro-batch and unbounded
D. micro-batch and bounded