Spark Structured Streaming

CLOUD DATA DRIVEN – JULY 2024

About me

- ✓ Passionate about Data ☺
- ✓ Founder at Softentity, 9+ years of experience in the industry
- ✓ Certified Microsoft Data Engineer Associate
- ✓ Certified Microsoft Power BI Analyst Associate
- ✓ Databricks Certified Data Engineer Associate
- ✓ Databricks Certified Data Engineer Professional

Connect with me:
LinkedIn: https://www.linkedin.com/in/iulianatuhasu/
Email: tuhasu.Iuliana@gmail.com
Blog: iulianatuhasu.com

Batch Processing

Full Batch Processing:

- **Approach:** Reprocess the entire dataset periodically.
- **Use Case:** Suitable for smaller datasets or scenarios where data changes infrequently. E.g. Adhoc data imports

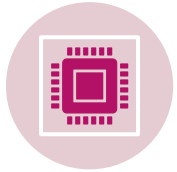
Incremental Batch Processing:

- **Approach:** Extract and process only data that has been modified or added since the last update.
- **Use Case:** Suitable for data warehouses, where changes are tracked by timestamps or IDs and recurrence is Daily, Weekly, Monthly.

The need

Immediate Data Insights: instant data updates

Reduced Latency: minimal delay between data arrival and processing



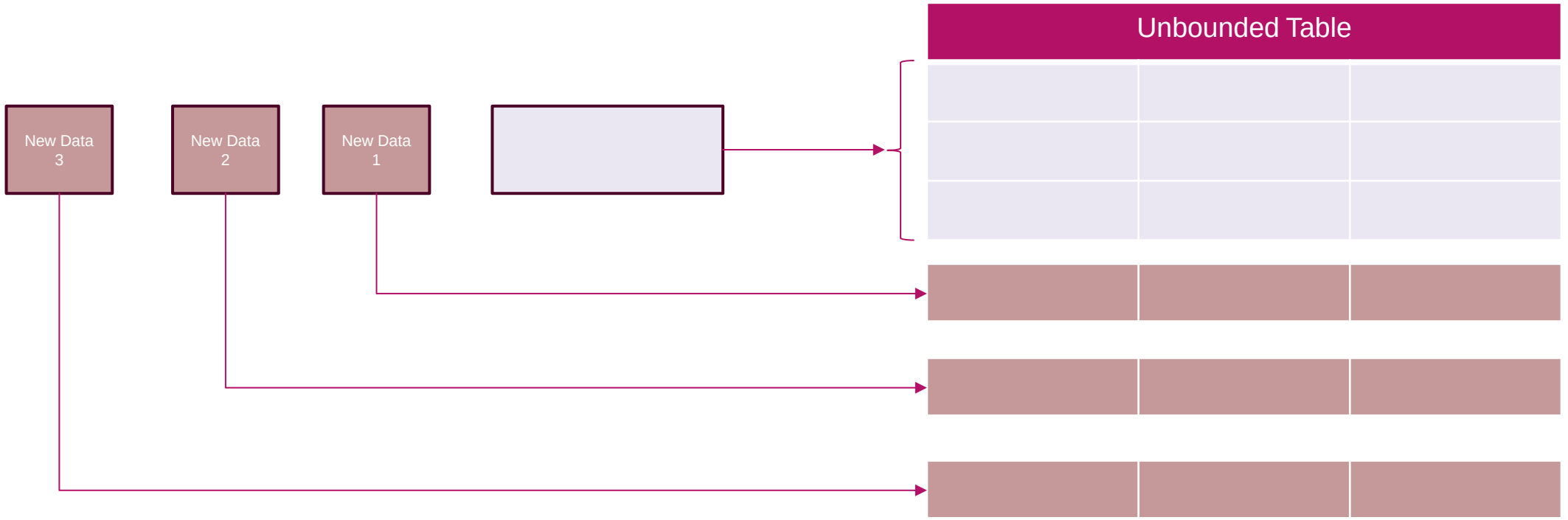
High Throughput: processing million of events

Spark Structured Streaming

- **Micro-Batch Processing engine**
  - Process data in small batches at short, regular intervals (e.g., every few seconds)
  - Latencies of ~100 ms
- **Continuous Processing**
  - Latencies of ~ 1ms
  - Experimental

Structured Streaming
=
Scalable, fault-tolerant engine on top of Spark SQL engine

Treat live data stream as an unbounded table

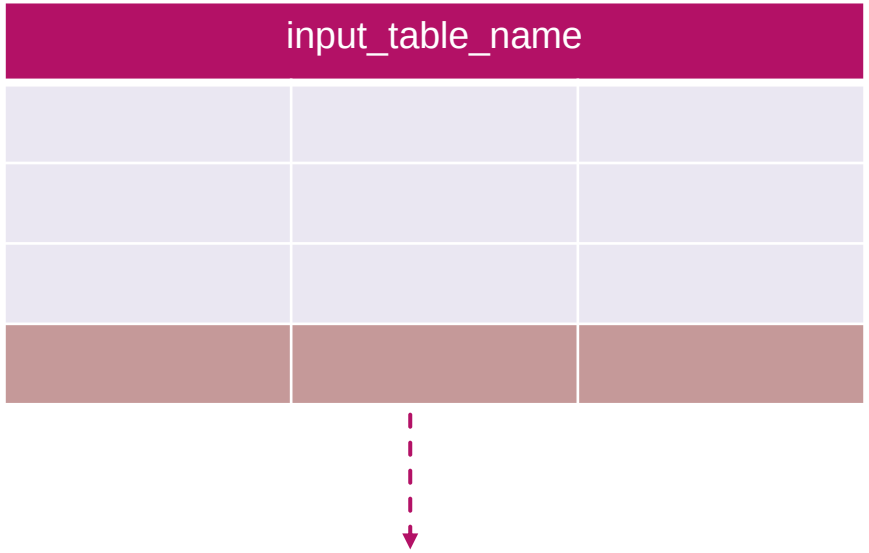


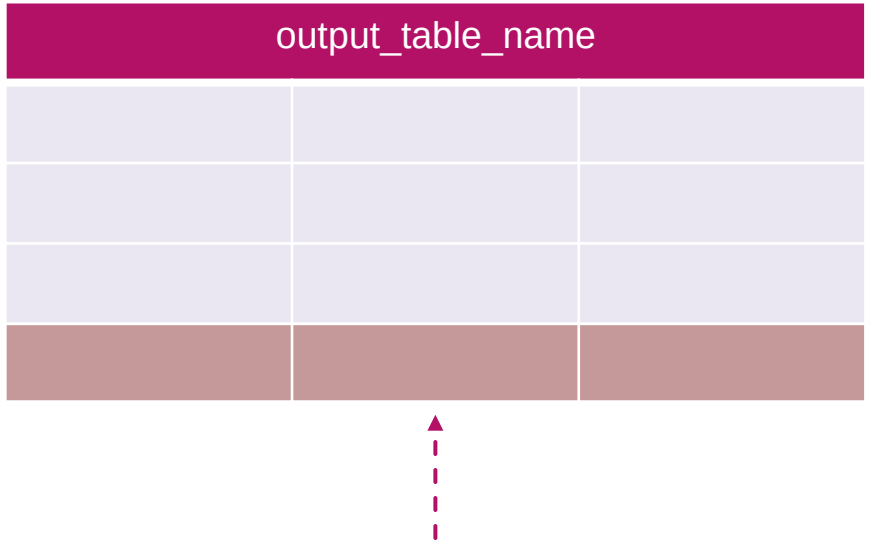
Data Stream

Continuous Growth Sources: Any data source that continually expands over time

- **Cloud Storage:** New files being added to cloud storage systems like AWS S3, Azure Blob Storage, or Google Cloud Storage
- **CDC Feeds:** Updates to a database tracked via Change Data Capture (CDC) mechanisms
- **Event Queues:** Events being queued in a publish/subscribe messaging system such as Apache Kafka or Azure Event Hubs

readStream & writeStream






```
streamingDF = spark.readStream
                .table(“input_table_name)
```

```
streamingDF.writeStream
        .trigger(processingTime=“3 minutes”
        .outputMode(“append”)
        .option(“checkpointLocation”,”/path”)
        .table(“output_table_name”)
```

Trigger intervals

Trigger Method Call	Behavior
Default: unspecified	Process data in micro-batch mode. Micro-batches are generated once the previous micro-batch has completed.
.trigger(processingTime=”2 minutes")	Process data in micro-batches at the user-specified intervals (e.g., every 2 minutes).
.trigger(once=True) - *deprecated*	Process all available data in a single batch, then stop.
.trigger(availableNow=True)	Process all available data in multiple micro-batches, then stop.

Output Modes

Append Mode (default)

- Only new rows from the streaming source will be written to sink
- .outputMode(“append”)

Complete Mode

- The output table is being overwritten
- Supported for aggregation queries
- .outputMode(“complete”)

Update Mode

- Upsert the output data
- .outputMode(“update”)

Checkpoints

Stores the Stream State

- Maintains intermediate data and variables
- Essential for accurate ongoing computations
- Example: During aggregations, the system needs to remember previous events or partial results to compute the current output

Tracks the Progress of the Stream Processing

- Records which data has already been processed and what remains to be done
- Ensures no data is skipped or reprocessed

Note: The same checkpoint cannot be used by two separate streams

Quick Demo

Incremental Data Ingestion From Files

COPY INTO

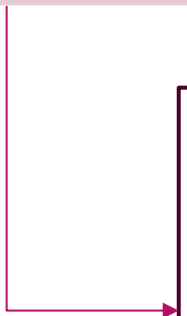
- **Incremental Loading**: Incrementally loads data from external storage into Delta Lake tables
- **Use Case**: Suitable for scenarios with smaller volumes of data
- **Simplicity**: Simple to use with SQL-like syntax
- **Trigger**: Requires manual trigger or scheduling

Auto Loader

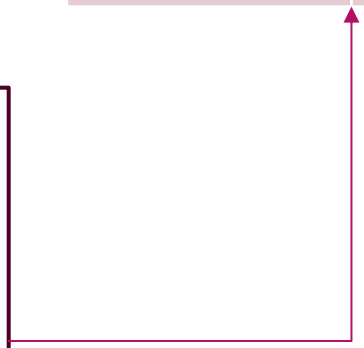
- **Automatic Detection**: Automatically detects and processes new files landing in a cloud storage location
- **Trigger**: Automatically triggered as new data arrives
- **Use Case**: Ideal for real-time or near-real-time data ingestion without manual intervention
- **Advanced Processing**: Leverages Structured Streaming for continuous data processing

Auto Loader

/demo/input
Data_2.csv
Data_3.csv
Data_3.csv



```
spark.readStream
        .format("cloudFiles")
                .option("cloudFiles.format", <source_format>)
                .load('/demo/input')
        .writeStream
        .option("checkpointLocation", <checkpoint_directory>)
        .table(<table_name>)
```



output_table_name	

Auto Loader

- **Incremental Data Processing**
  - Auto Loader efficiently processes new data files as they arrive in cloud storage (or optionally all files from the storage), utilizing cloudFiles as a Structured Streaming source
- **Scalability**
  - Capable of near real-time ingestion of millions of files per hour
- **Checkpointing**
  - Metadata of files is persisted, ensuring data is processed exactly once
  - In case of failure, it resumes from where it left off
- **Supported Storage**
  - AWS S3, ADLS Gen2, Google Cloud Storage, Azure Blob Storage, DBFS
- **File Formats**
  - Supports JSON, CSV, XML, PARQUET, AVRO, ORC, TEXT, and BINARYFILE formats

Quick Demo

Stateless Transformations

Definition: Transformations that do not depend on any state or data from previous events.

Examples

Map: Transforms each input element into exactly one output element.
- *Example*: map(lambda x: x * 2) transforms each element by multiplying by 2.

Filter: Filters elements based on a predicate.
- *Example*: filter(lambda x: x > 10) filters elements greater than 10.

FlatMap: Transforms each input element into zero or more output elements.
- *Example*: flatMap(lambda x: [x, x * 2, x * 3]) generates multiple outputs for each input.

Select: Projects a set of expressions and returns a new DataFrame.
- *Example*: select("name", "age") selects only the "name" and "age" columns.

Explode: Transforms each element in an array or map into multiple rows.
- *Example*: explode("array_column") turns each element in the array into a separate row.

Stateful Transformations

Definition

- Transformations that depend on state that is accumulated and updated over time
- Written between readStream & writeStream

Examples

- **Aggregation**: counts, sums, etc
- **Windowed Operations**: tumbling, sliding windows, etc
- **Joins**: Joins data streams based on keys over a time window.

```
# Read the streaming data
input_stream = spark.readStream \
    .format("delta") \
    .table("silver_stateful_table")

# Perform the aggregation
aggregated_data = input_stream \
    .groupBy("CustomerId") \
    .agg(_sum("AccumulatedAmountSpent").alias("TotalAmountSpent")) \
    .withColumn("TotalDiscountGiven", expr("TotalAmountSpent * 0.01"))

# Write the result
query = aggregated_data.writeStream \
    .outputMode("update") \
    .foreachBatch(merge_to_gold_table) \
    .option("checkpointLocation", "dbfs:/mnt/output/aggregated_data_checkpoint") \
    .start()
```

Input – batch1

Transacti...	CustomerId	TransactionDate	AccumulatedAmountSpent
T001	C001	2024-07-10T10:00:00.000+00:...	100
T002	C002	2024-07-10T11:00:00.000+00:...	200
T003	C003	2024-07-10T12:00:00.000+00:...	300

Output after batch 1

CustomerId	TotalAmountSpent	TotalDiscountGiven
C001	100	1
C002	200	2
C003	300	3

Input – batch2

TransactionId	CustomerId	TransactionDate	AccumulatedAmountSpent
T001	C001	2024-07-10T10:00:00.000+00:...	100
T002	C002	2024-07-10T11:00:00.000+00:...	200
T003	C003	2024-07-10T12:00:00.000+00:...	300
T004	C001	2024-07-10T13:00:00.000+00:...	200
T005	C002	2024-07-10T14:00:00.000+00:...	300
T006	C004	2024-07-10T15:00:00.000+00:...	400

Output after batch 2

CustomerId	TotalAmountSpent	TotalDiscountGiven
C001	300	3
C002	500	5
C003	300	3
C004	400	4

Stateless aggregation

```python
# Read the streaming data
input_stream = spark.readStream \
    .format("delta") \
    .table("silver_stateful_table")

# Perform the aggregation
aggregated_data = input_stream \
    .groupBy("CustomerId") \
    .agg(_sum("AccumulatedAmountSpent").alias("TotalAmountSpent")) \
    .withColumn("TotalDiscountGiven", expr("TotalAmountSpent * 0.01"))

# Write the result
query = aggregated_data.writeStream \
    .outputMode("update") \
    .foreachBatch(merge_to_gold_table) \
    .option("checkpointLocation", "dbfs:/mnt/output/aggregated_data_checkpoint") \
    .start()
```

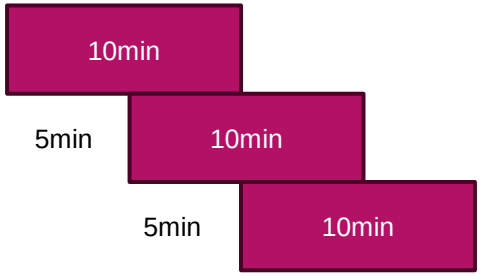
```python
def merge_to_gold_table(batch_df, batch_id):
    aggregated_data = batch_df \
        .groupBy("CustomerId") \
        .agg(_sum("AccumulatedAmountSpent").alias("TotalAmountSpent")) \
        .withColumn("TotalDiscountGiven", expr("TotalAmountSpent * 0.01"))

    .....
```

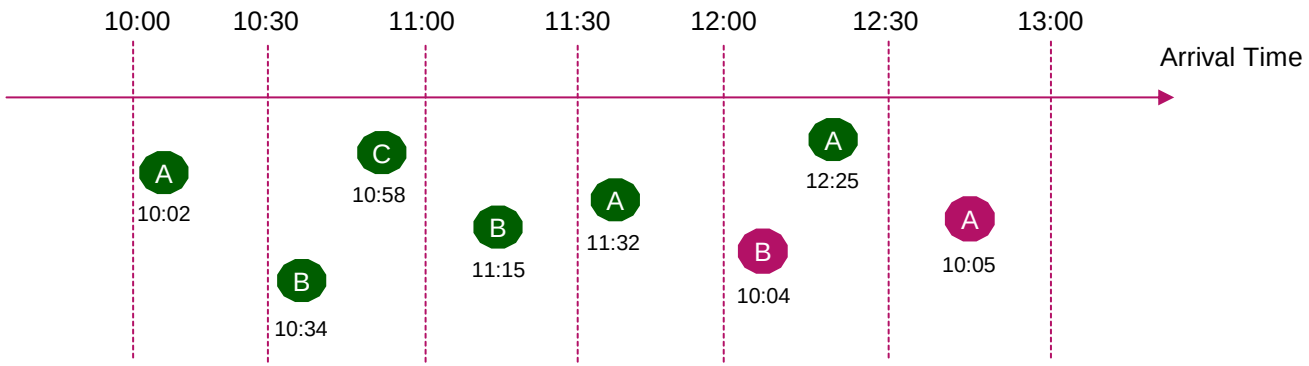
Types of Windows – examples

Tumbling Windows:
•Fixed-size, non-overlapping time intervals.
•Example: Counting events per 10-minute window.

Sliding Windows:
•Fixed-size, overlapping time intervals.
•Example: Counting events every 5 minutes over a 10-minute window.



Late arriving data



Letter	CountOfLetters
A	4
B	3
C	1

By Maintaining State -> updates counts in previous windows when late data arrives

Increasing State -> Risks of out-of-memory issues

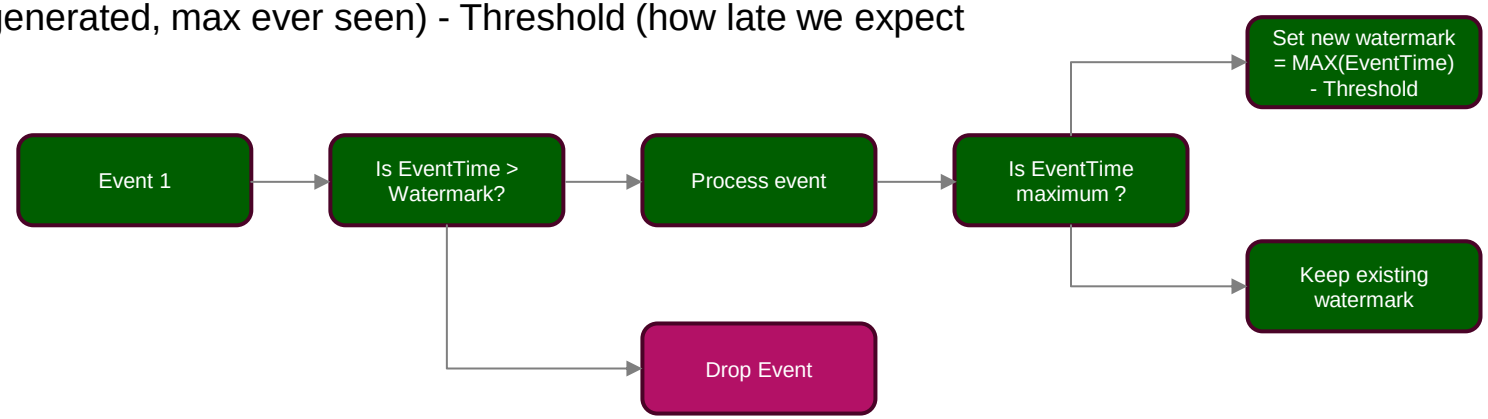
Solution -> Use Watermarking to drop unnecessary / old windows

Watermarking

- Technique to handle late data in stream processing
- Discard state for old windows
- Output Mode – either Append or Update

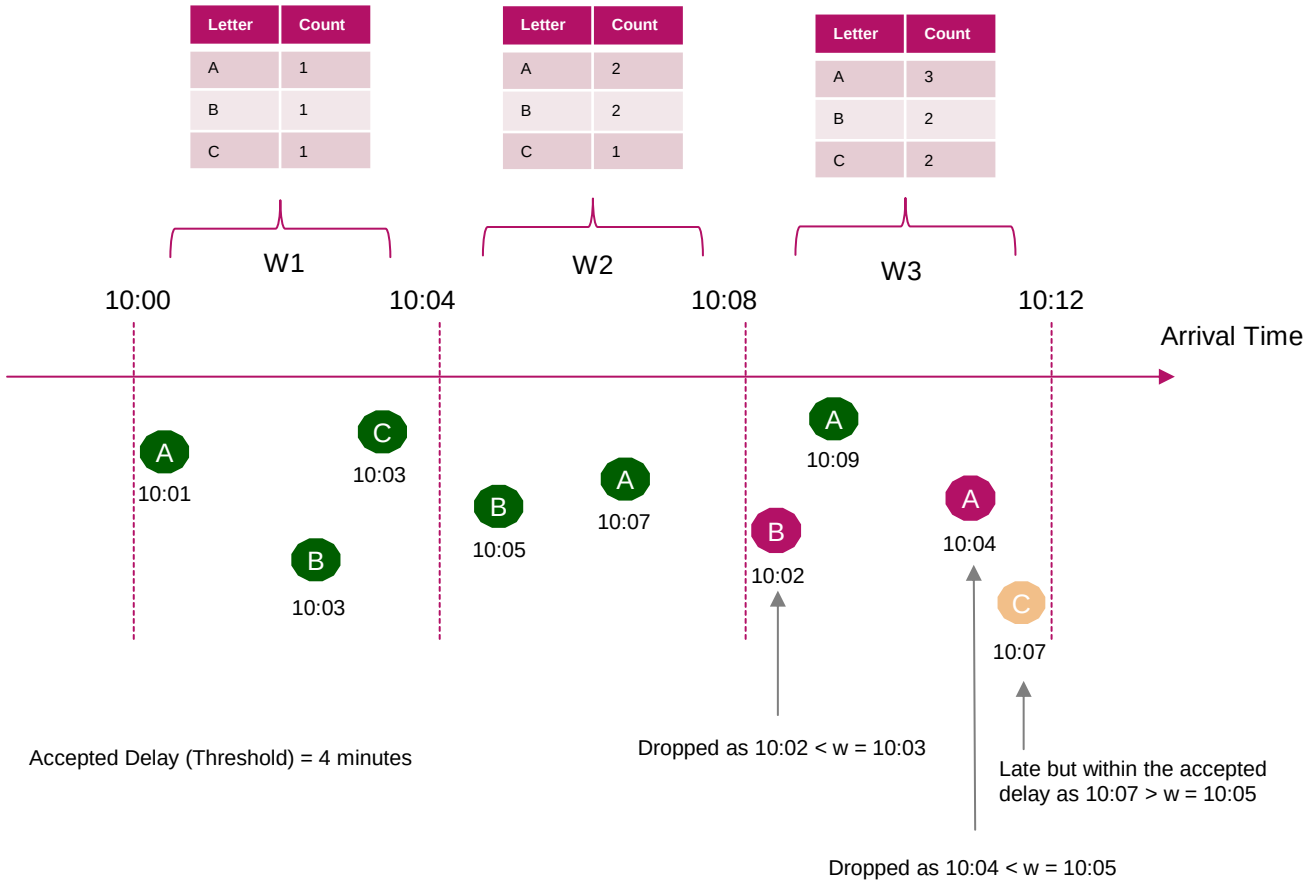
Watermark = EventTime (when the event has been generated, max ever seen) - Threshold (how late we expect the data to arrive)

```
watermarked_df = streaming_df \
    .withWatermark("event_time", "4 minutes") \
    .groupBy(
        window(col("event_time"), "4 minutes"),
        col("value")
    ) \
    .count().alias("count") \
    .select("window.start", "window.end", "value", "count")
```



Accepted Delay (Threshold) = 4 minutes

Watermarking



Quick Demo

Joins

Stream – Static joins

- Between a streaming table and a static dataframe
- Not stateful
- Inner, Left Outer, Left Semi (stream-static)
- Inner, Right Outer (static-stream)

Stream – Stream joins

- Between two streaming dataframes
- Stateful
- Inner (optional watermark, optional time constraints)
- Left Outer, Right Outer, Full Outer, Left Semi (mandatory watermark + time constraints)

Stream-Static Join

Transactions Table - Streaming

TransactionId	CustomerId	Transaction Date	TypeId
T001	1	2023-07-20	1
T002	2	2023-07-21	2
T003	3	2023-07-22	4

Transaction Type Table - Static

TypeId	TypeName
1	Card
2	Cash

Output

TransactionId	CustomerId	TransactionDate	TypeId	TypeName
T001	1	2023-07-20	1	Card
T002	2	2023-07-21	2	Cash

Stream-Static Join

Transactions Table - Streaming

TransactionId	CustomerId	Transaction Date	TypeId
T001	1	2023-07-20	1
T002	2	2023-07-21	2
T003	3	2023-07-22	4

Transaction Type Table - Static

TypeId	TypeName
1	Card
2	Cash
4	Bank Statement

Output

TransactionId	CustomerId	TransactionDate	TypeId	TypeName
T001	1	2023-07-20	1	Card
T002	2	2023-07-21	2	Cash

- Only matched records at the time the stream is being processed will be part of the output
- The streaming part of the join drives the join.

Stream-Stream Join

Transactions Table - Streaming

TransactionId	CustomerId	Transaction Date	TypeId
T001	1	2023-07-20	1
T002	2	2023-07-21	2
T003	3	2023-07-22	4

Transaction Type - Streaming

TypeId	TypeName
1	Card
2	Cash
4	Bank Statement

Output

TransactionId	CustomerId	TransactionDate	TypeId	TypeName
T001	1	2023-07-20	1	Card
T002	2	2023-07-21	2	Cash
T003	3	2023-07-22	4	Bank Statement

- Future input data can match past input data
- State can grow indefinitely => limit with watermark

Quick Demo

Streaming Deduplication – dropDuplicates()

dropDuplicates()

- **by using a Unique Identifier (e.g., Id + EventTime)**
- **Similar to distinct**

- **Without Watermark:**
  - A duplicate record can arrive at any time.
  - Past records are kept as state, causing state to grow over time.
- **With Watermark:**
  - Limits how late a duplicate record can arrive.
  - The watermark removes old state, managing the state size efficiently.

```
deduped_df = (bronze_transformed
                .withWatermark("TransactionDate", "30 seconds")
                .dropDuplicates(["TransactionId", "TransactionDate"]))
```

Streaming Deduplication – Complex logic

Custom logic within foreachBatch()

- Data already exists in the output (upsert new records from the input stream without creating duplicates)
- Deduplication based on sorting criteria (e.g., retaining the most recent record or the record with the highest priority)

2 parameters:

- microbatchDF -> dataframe with output data of the micro-batch
- batch -> unique id of the micro-batch

```
query = (deduped_df.writeStream
                   .foreachBatch(upsert_data)
                   .option("checkpointLocation", "dbfs:/mnt/data/checkpoints/silver_transactions")
                   .trigger(availableNow=True)
                   .start()
                   )
```

```
def upsert_data(microBatchDF, batch):
    microBatchDF.createOrReplaceTempView("transactions_microbatch")

    sql_query = """
      MERGE INTO silver_transactions a
      USING transactions_microbatch b
      ON a.TransactionId=b.TransactionId and a.TransactionDate=b.TransactionDate
      WHEN MATCHED THEN UPDATE SET *
      WHEN NOT MATCHED THEN INSERT *
    """

    microBatchDF.sparkSession.sql(sql_query)
```

Quick Demo

Q&A