THE DATA SCIENCE LAB
Introduction to Data Stream Processing

COM 490 – Spring 2025

Week 10

EPFL

Stream Processing Module

- Objectives
  - Review concepts of stream processing
  - Experiment with typical tools for
    - Data ingestion and processing
- Week 9
  - Concepts
  - Experiments
- Week 10
  - Advanced topics
    - Operations on streaming data (joins)
    - Time constraints
- Week 11
  - Analytics on data at rest and data in motion

Agenda for Today

- Lecture
- Presentation of Final Project
- Exercises

Reminder: Previous Lesson

- Why Stream Processing?
  - Relevance (vs batch)
- Application of Stream Processing
  - Computing, Real-time monitoring, Social Media, etc
- Constraints and Challenges
  - Nature of Input, Output Considerations
- Window Processing
  - And related Concepts
- Stream Processing Tools
- Introduction to Kafka and Spark Streaming
- Exercises

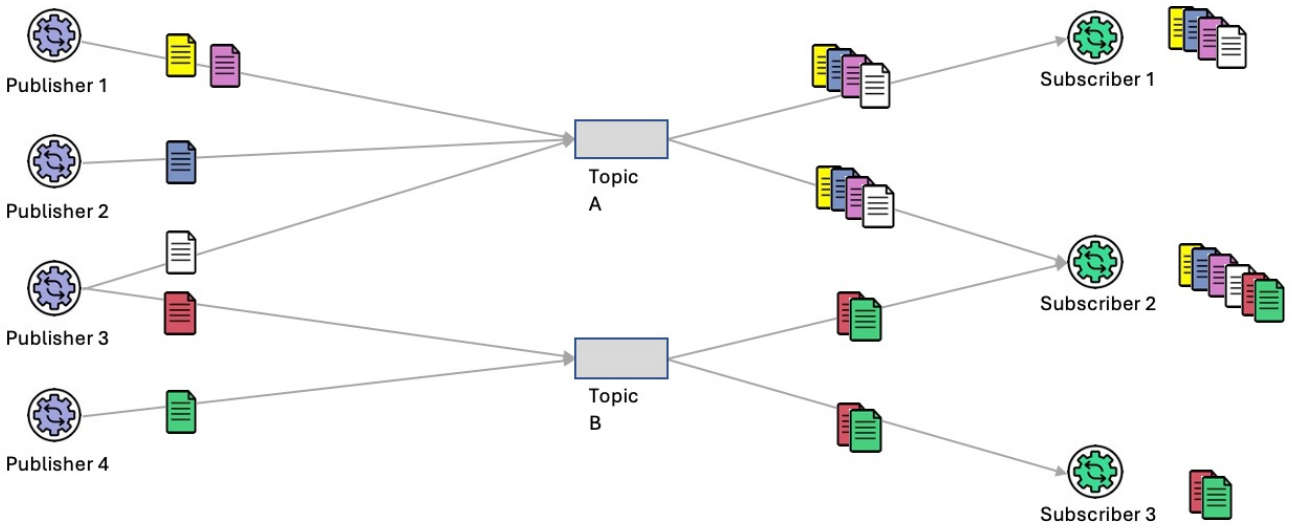
Kafka - Overview

Kafka is a **distributed event streaming platform** used to:

- Publish and subscribe to event streams
- Store streams of data reliably
- Process streams in real-time

Kafka Terminology

- Publisher => **Producer**
- Subscriber => **Consumer**



Kafka - Components

Component	Role
Producer	Sends data (events) to Kafka
Message	Array of bytes *(+optional key, otherwise round robin)*
Broker	Kafka server that stores data
Topic	Named category where records are published
Partition	Subdivision of a topic for parallelism
Offset	Unique ID of each record within a partition
Consumer	Reads data from Kafka
Consumer Group	A group of consumers that share a workload
ZooKeeper (legacy)	Coordinates brokers (being replaced by **KRaft**)

Kafka – Broker and Cluster

- A Kafka **broker** is a server that stores topics and serves data to clients
- A Kafka **cluster** = multiple brokers
- Each broker manages **some partitions**
- Kafka **replicates partitions** across brokers for **fault tolerance**
  - A partition has a **leader broker**, others are **followers**

ZooKeeper or KRaft (New)

ZooKeeper (*typically)* used for:

- Broker registration
- Leader election
- Metadata management

Now being replaced by **KRaft mode** (Kafka Raft)

- Kafka becomes self-managing
- More scalable (auto-scaling)
- No external dependency needed (less complexity and more secure)

Kafka – Topics and Partitions

Topic = named stream of data (e.g. sensor-data, logs)

Each topic has one or more **partitions**

- A partition is an **ordered log of records**
- Enables **parallelism** across consumers and brokers
- Ensures **scalability**
- Improves **performance**

Kafka – Messages

- A **Kafka message** (also called a **record**) is the **smallest unit** of data stored and transmitted in Kafka.
  Each message is written to **one partition** within a **topic**

Field	Description
Key (optional)	Used to determine **which partition** the message goes to. Preserves order per key.
Value (required)	The **actual data payload** (e.g. JSON, Avro, plain text)
Offset (auto)	A unique, **sequential ID** within the partition — used for tracking
Timestamp	Time when the message was created or received (event time vs processing time)
Headers (optional)	Extra metadata (key-value pairs)

Kafka – Offsets: Tracking Position

- Every record in a partition has a unique **offset**
- Consumers **use offsets to track what they've read**
- Offsets can be:
  - **Committed manually or automatically**
  - **Reset** to latest, earliest, or a specific value
- Offsets are how Kafka achieves **exactly-once** or **at-least-once** delivery semantics

Kafka – Producers

A **producer** sends data to a Kafka topic

Can:

- Send records with or without keys
- Control which partition data goes to (via key or partitioner)
- Choose **acks** settings for durability

Setting	Behavior
acks=0	Fire-and-forget
acks=1	Wait for broker to ack
acks=all	Wait for all replicas to ack (safest)

Kafka – Consumer Groups

A **consumer group** is a named group of consumers that share a subscription

Kafka **distributes partitions** across group members

- Each partition → assigned to **only one consumer**
- Enables **horizontal scalability** and **fault tolerance**

Spark (Structured) Streaming

Apache Spark Structured Streaming is a **high-level streaming API** built on **Spark SQL and DataFrames**.

=> You write queries as if working with static data

Key Characteristics

Key Characteristics

- Unified with **Spark SQL**: write streaming jobs with SQL/DataFrames
- Handles **event time**, **watermarking**, **stateful aggregations**
- Supports **exactly-once** semantics (with proper sinks)
- Runs on the **same Spark engine** as batch jobs

Core Concepts

Concept	Description
Source	Where the data comes from (e.g., Kafka, files, sockets)
Sink	Where the data goes (e.g., console, Kafka, files, custom logic)
Streaming DataFrame	A logical table that continuously updates with new data
Output Mode	Controls what data is written: append, update, complete
Watermark	Mechanism for handling **late data**
Windowing	Group events based on time intervals

Basic Example (Console Output)

```
df = spark.readStream.schema(schema).json("input_dir")

df_transformed = df.withColumn("temp_C", col("temp_F") - 32 * 5/9)

query = df_transformed.writeStream \
    .format("console") \
    .outputMode("append") \
    .start()
```

This code reads streaming data from JSON files, transforms it, and writes results to the console.

Other Supported Operations

- Streaming Joins
  - Across Streams or with Static Data
- Stateful Operations
  - Require Spark to **remember past data** across batches.
    - Typical use cases
      - Session tracking
      - Deduplication
      - Running totals
      - Windowed aggregations with updates

Streaming Join Example

- Problem
  - We want to **join live temperature readings** from weather stations with a **static table of station metadata** to enrich the stream with human-readable info (e.g., location name, altitude).
  - Stream 1 — Temperature Data (from Kafka)
    - `{"station_id": "S01", "timestamp": "2025-04-29T14:00:00", "temperature": 21.5 }`
  - Static Table — Station Info (from CSV or DB)
    - `station_id, location_name, altitude`
    - `S01,"Zürich, CH",410`
    - `S02,"Geneva, CH",375`

Streaming Join Example

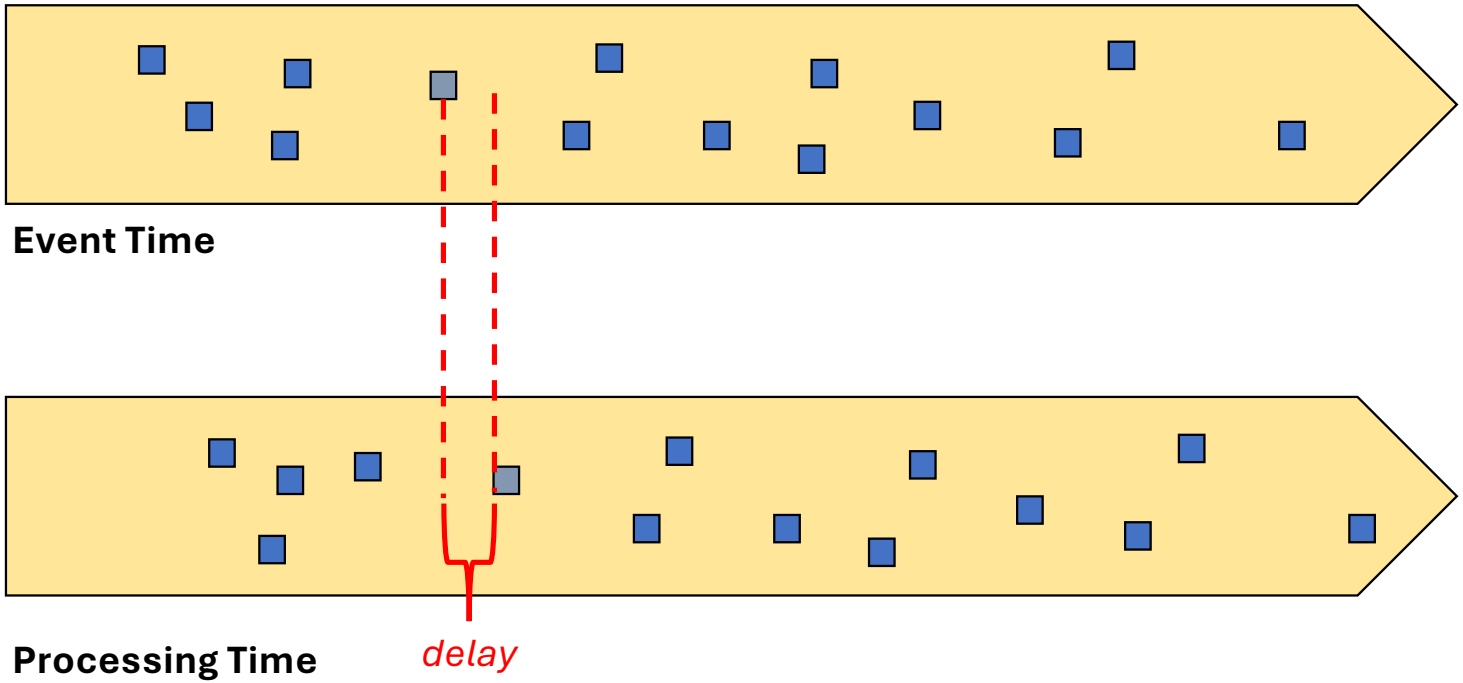
- Spark Code — Stream–Static Join

```
from pyspark.sql.functions import from_json, col,
to_timestamp
from pyspark.sql.types import StructType, StringType,
TimestampType, DoubleType

# Define schema for temperature data
temp_schema = StructType() \
    .add("station_id", StringType()) \
    .add("timestamp", TimestampType()) \
    .add("temperature", DoubleType())

# 1. Read stream from Kafka
temperature_stream = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers",
"localhost:9092") \
    .option("subscribe", "temperature-stream") \
    .load() \
    .selectExpr("CAST(value AS STRING) AS json") \
    .select(from_json(col("json"),
temp_schema).alias("data")) \
    .select("data.*") \
    .withWatermark("timestamp", "10 minutes")

# 2. Load static station info (e.g., from CSV)
station_info = spark.read \
    .option("header", True) \
    .csv("station_metadata.csv")

# 3. Perform the join (stream + static)
enriched_stream = temperature_stream.join(
    station_info,
    on="station_id",
    how="left"
)

# 4. Write to console (or sink)
query = enriched_stream.writeStream \
    .format("console") \
    .outputMode("append") \
    .option("truncate", False) \
    .start()
```

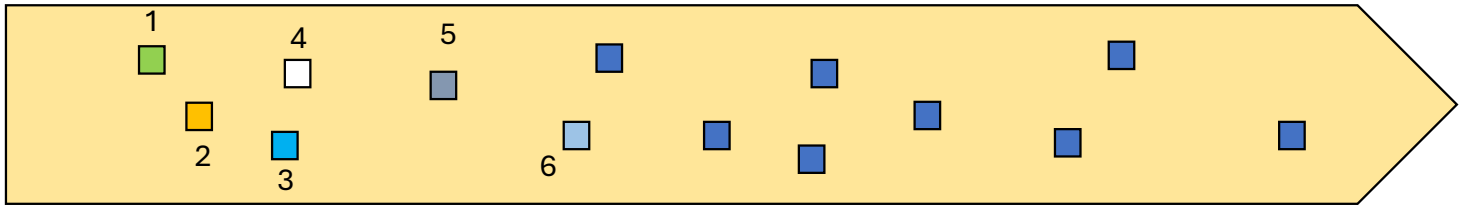
Core Concepts (cont)

Concept	Description
Source	Where the data comes from (e.g., Kafka, files, sockets)
Sink	Where the data goes (e.g., console, Kafka, files, custom logic)
Streaming DataFrame	A logical table that continuously updates with new data
Output Mode	Controls what data is written: append, update, complete
Watermark	Mechanism for handling **late data**
Windowing	Group events based on time intervals

Event Time vs Processing Time

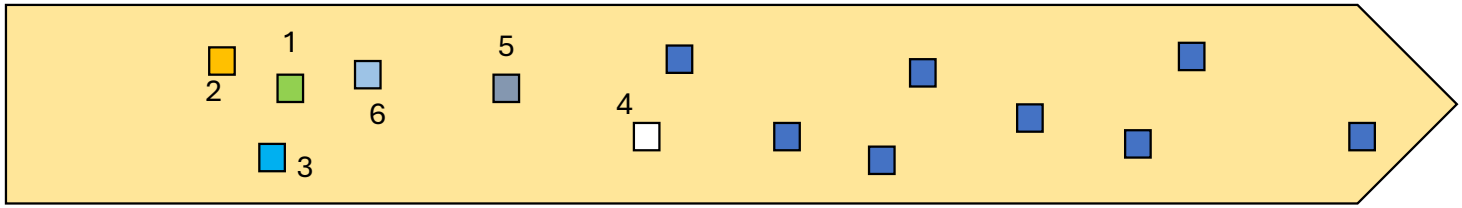


Event Time vs Processing Time



Event Time

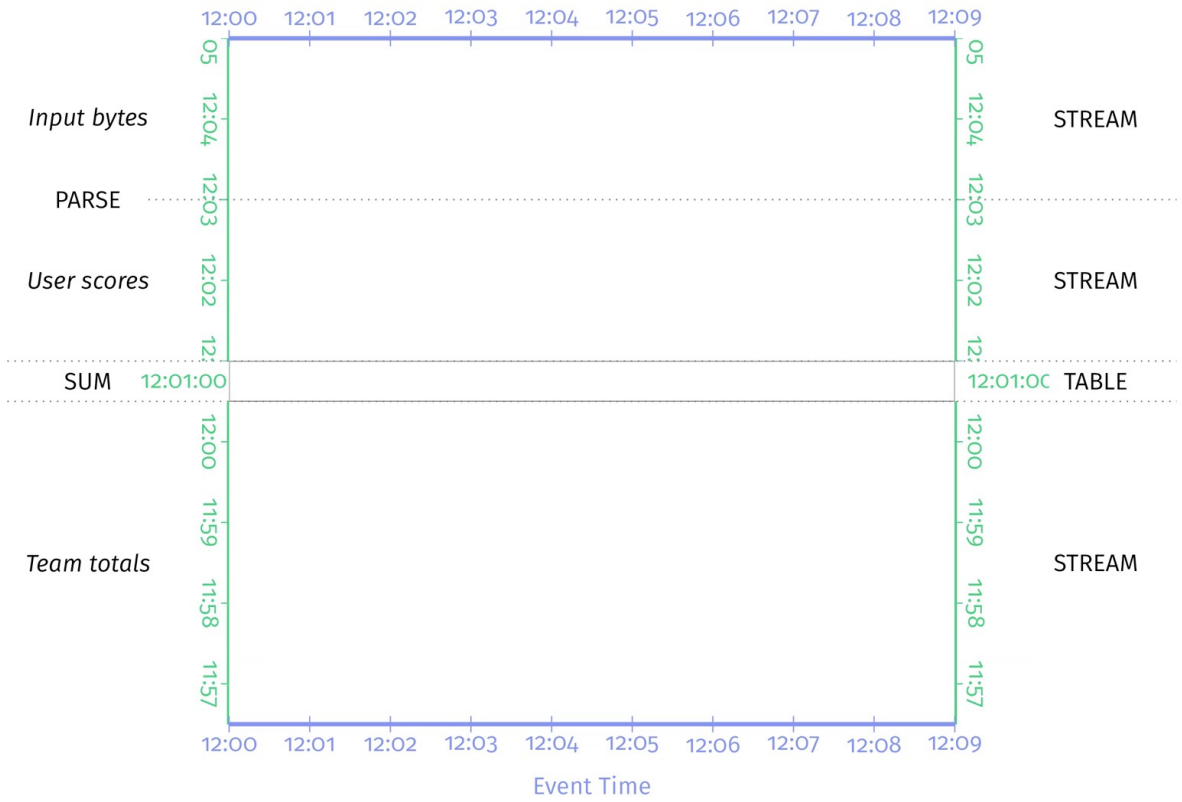
1, 2, 3, 4, 5, 6, ...



Processing Time

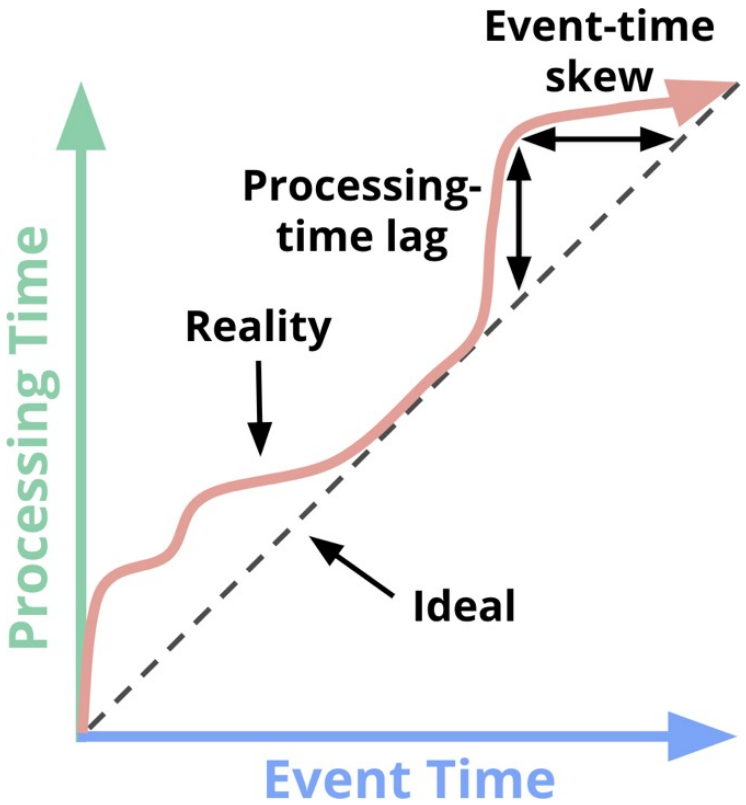
2, 3, 1, 6, 5 , 4, ...

Event Time vs Processing Time



Credits: Tyler Akidau (et al.), Streaming Systems, O'Reilly Media, 2018.

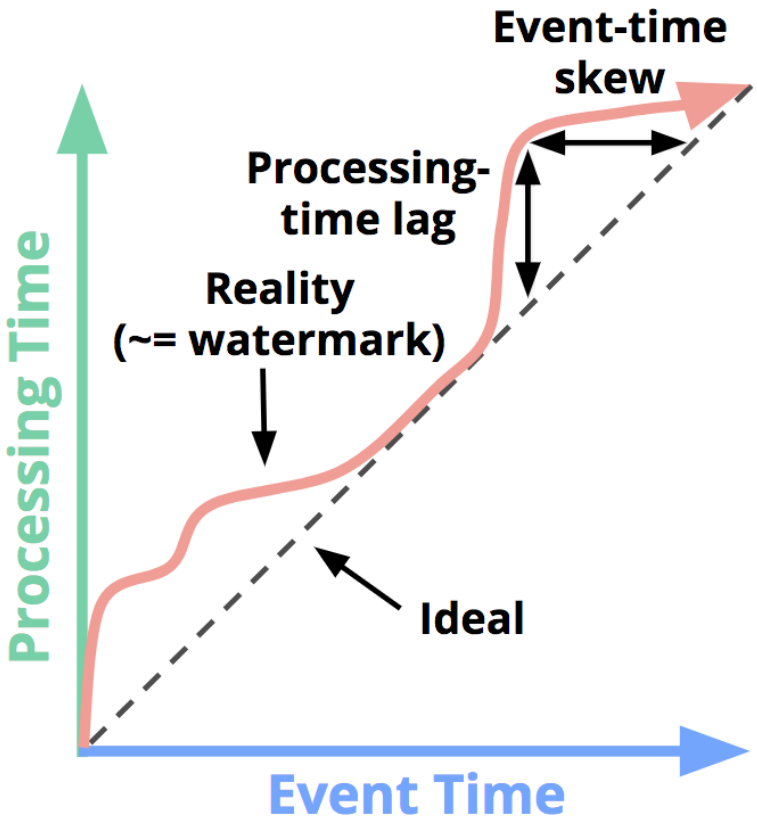
Event Time vs Processing Time



Concept	Meaning
Event Time	The time **embedded in the data** (e.g. when a temperature reading was captured by a sensor)
Processing Time	The time **system receives and processes** the data (i.e. system clock time)

Credits: Tyler Akidau (et al.), Streaming Systems, O'Reilly Media, 2018.

What is a Watermark?



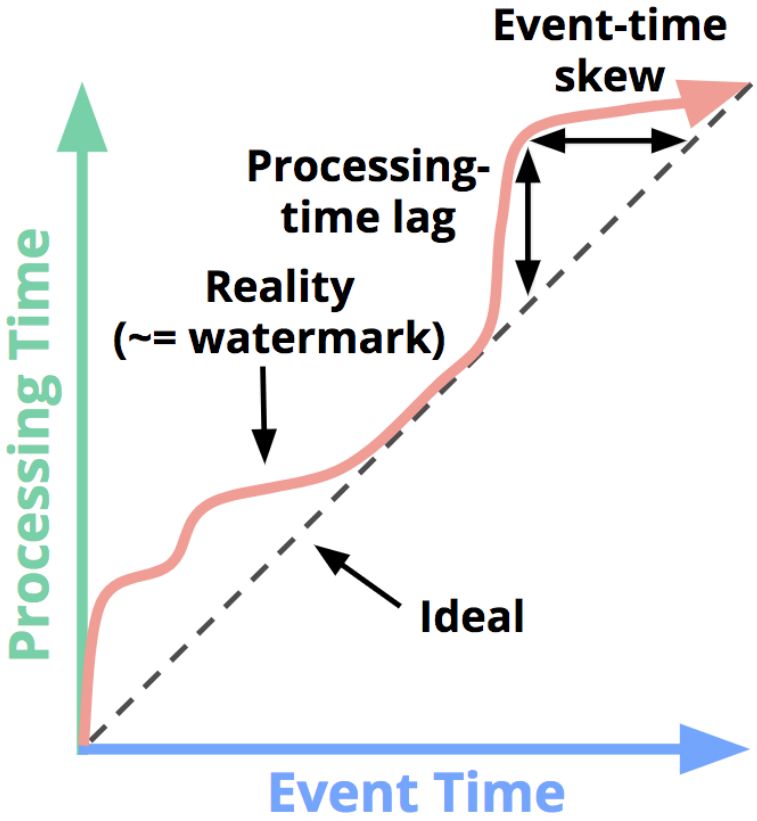
Problem:
How to deal with **late-arriving data** while still **finalizing results**?

Solution: Watermark
• Tells System *(e.g. Spark Streaming)* :
"I'm okay with accepting events **up to N minutes late**, based on their **event time**."

Behavior:
• System **continues to update** aggregations until watermark passes.
• Once watermark passes a window:
  - System **evicts** the window from memory.
  - Any event for that window arriving after → **ignored**.

Credits: Tyler Akidau (et al.), Streaming Systems, O'Reilly Media, 2018.

Watermark & Finalization



Important Distinction:

Stage	What happens?
Before watermark	System may **recompute** results as new (late) data arrives
After watermark	System **finalizes** the result and stops updates for that time window

Credits: Tyler Akidau (et al.), Streaming Systems, O'Reilly Media, 2018.

Watermarking in Spark Streaming

Average Temperature Every 5 Minutes (Tumbling Windows)

Problem Statement:

- A Kafka topic contains JSON data from weather stations:
  - Fields: `station_id, timestamp, temperature`
- Goal: Calculate **average temperature every 5 minutes** (non-overlapping windows)
- e.g. [14:00–14:05], [14:05–14:10], …

Assumptions:

- Events may arrive **up to 10 minutes late**
- Use **event time** for aggregation

Watermarking in Spark Streaming

Spark Pseudo-code: Tumbling Windows

```
readStream from Kafka → parse JSON

withWatermark("timestamp", "10 minutes")

groupBy:
    window(col("timestamp"), "5 minutes"),   # fixed windows
    col("station_id")

agg:
    avg("temperature")

writeStream to console or sink
```

[14:00–14:05], [14:05–14:10], ...

Watermarking in Spark Streaming

Spark Pseudo-code: ~~*Tumbling*~~ Sliding Windows

```
readStream from Kafka → parse JSON

withWatermark("timestamp", "10 minutes")

groupBy:
    window(col("timestamp"), "5 minutes", "1 minute"),   # sliding windows
    col("station_id")

agg:
    avg("temperature")

writeStream to console or sink
```

[14:00–14:05], [14:01–14:06], [14:02–14:07], ...

Useful references

[1] https://spark.apache.org/docs/latest/streaming-programming-guide.html

[2] https://kafka.apache.org/documentation/

[3] https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/

[4] http://www.streamingbook.net/

[5] Streaming 101 and 102

https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101
https://www.oreilly.com/radar/the-world-beyond-batch-streaming-102/