SPARK STRUCTURED STREAMING

2024

The Hoang

LEARNING OBJECTIVES

Understanding Structured Streaming and Kafka Basics

Advanced Topics and Optimization

- Stateful processing and event-time aggregation
- Watermarking and late event handling

Spark and Kafka Integration Overview

Real-World Applications

Data processing – What is Stream Processing?

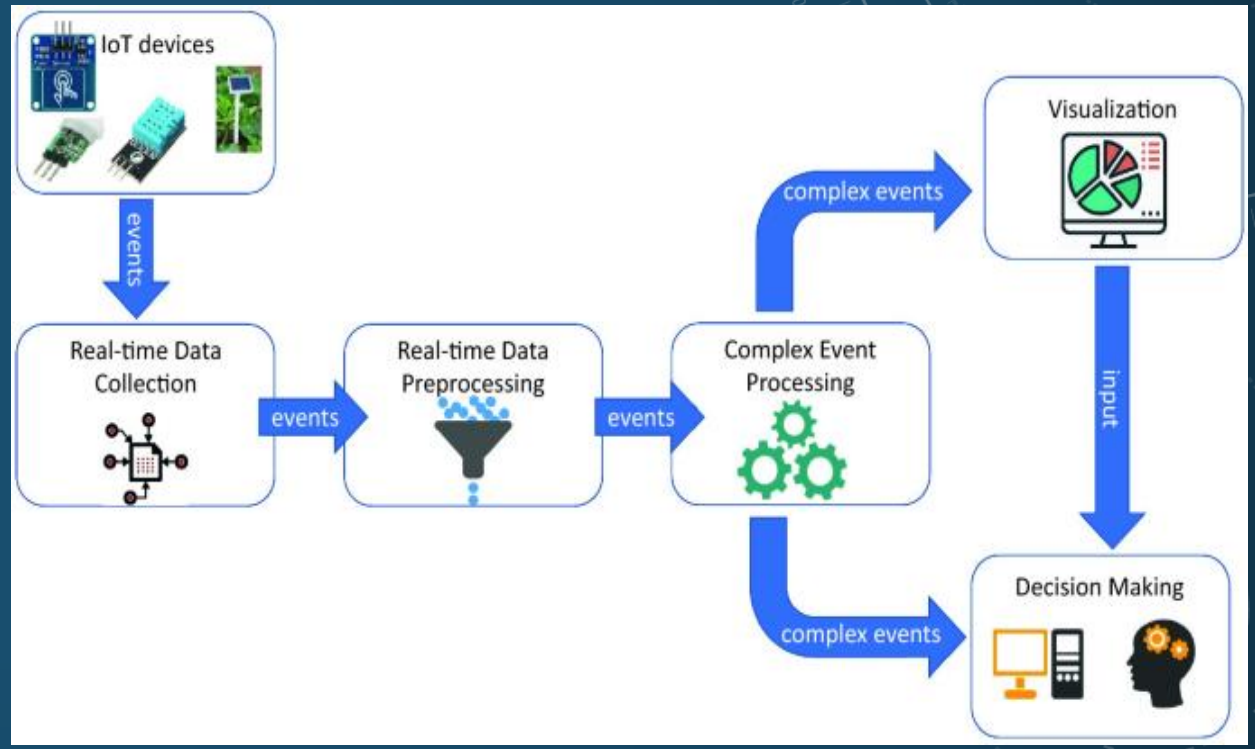
RECAP

Definition

Stream processing involves processing data in real-time as it arrives, enabling low-latency processing.

Characteristics

- Continuous input and processing of data.
- Unbounded, they have no predefined end.
- Suitable for real-time analytics and monitoring.
- Lower latency but can be more complex to implement.



Example

Using Apache Kafka and Apache Flink for real-time fraud detection in financial transactions.

Data processing – Stream processing example

RECAP

Scenario

Real-time monitoring of credit card transactions to detect fraud.

Steps

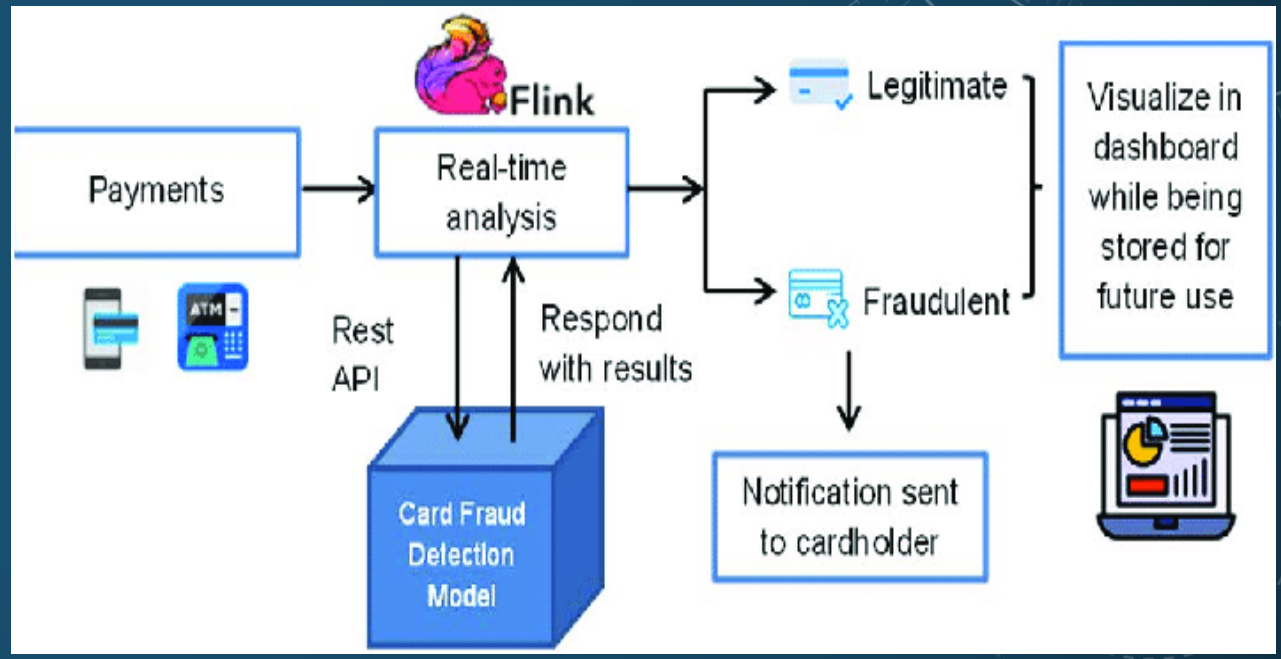
- Data Ingestion: Transactions are ingested from Kafka in real-time.
- Stream Processing: Apache Flink processes each transaction to identify anomalies.
- Anomaly Detection: Suspicious transactions are flagged immediately.
- Output: Real-time alerts are sent to the fraud detection team.

Advantages

Immediate detection and response to fraud.

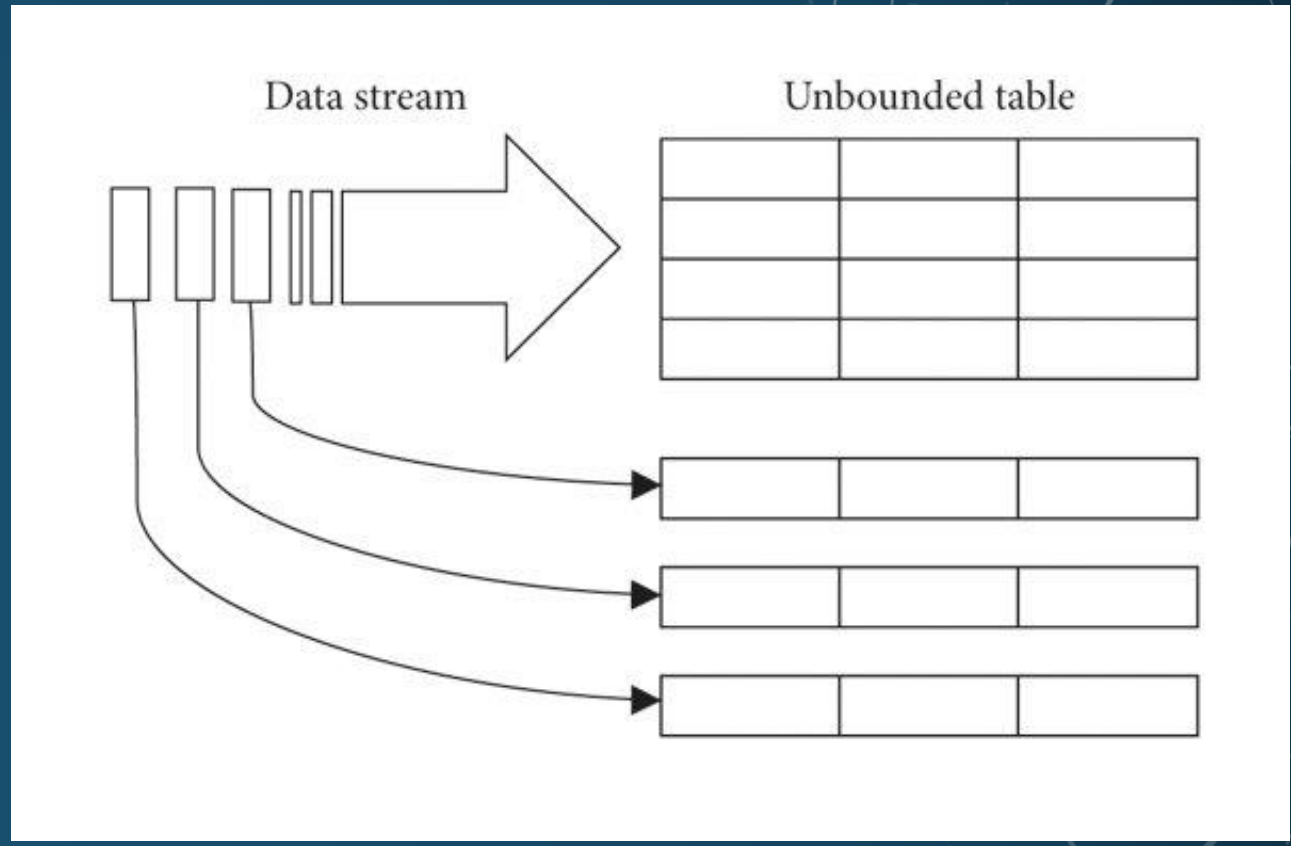
Disadvantages

More complex to manage and scale.



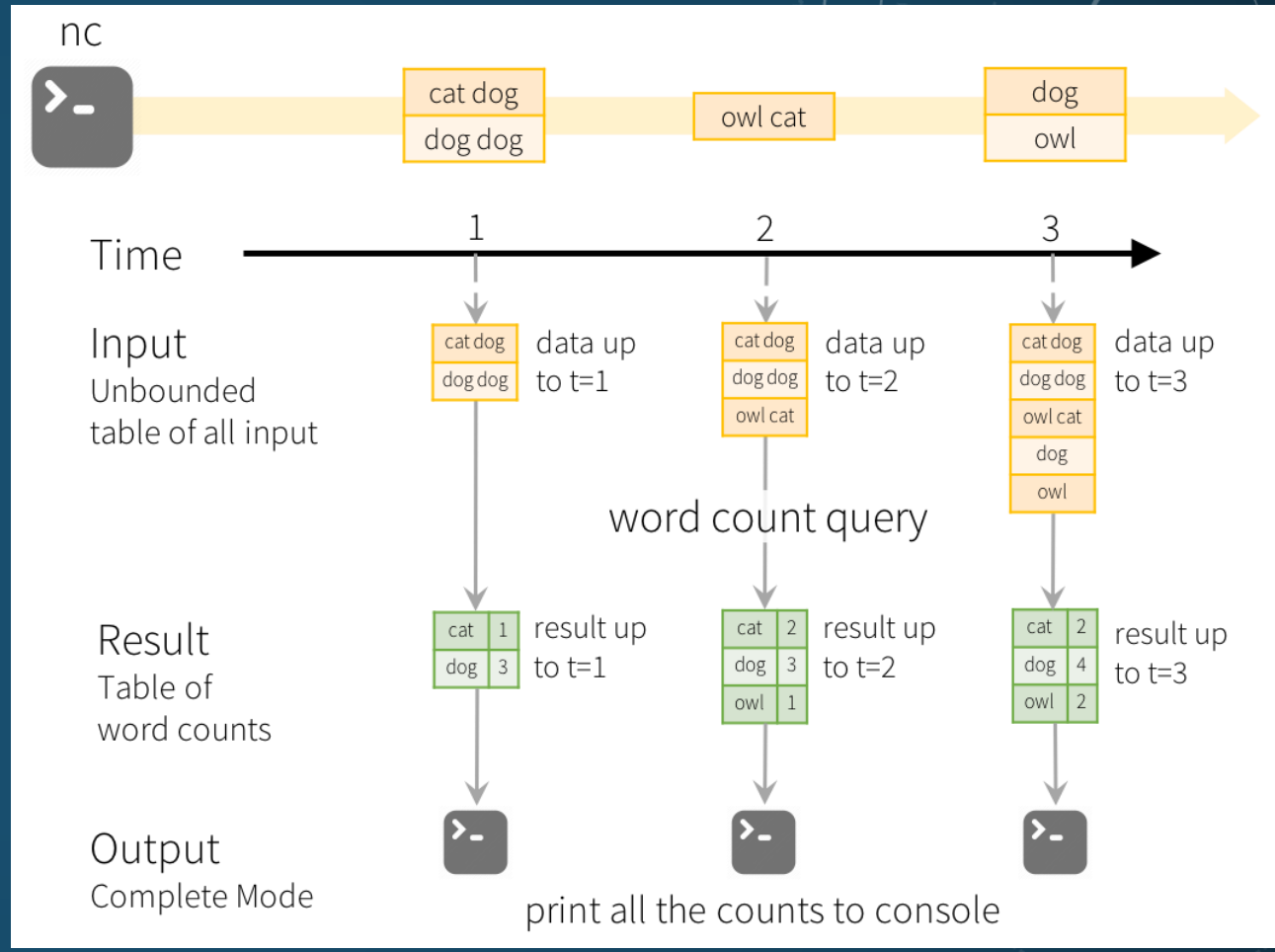
SPARK STRUCTURED STREAMING - INTRODUCTION

- Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.
- It allows you to express streaming computations in the same way you would express batch computations on static data.
- The Spark SQL engine manages the execution of these computations incrementally and continuously, updating the results as new data arrives.
- This approach provides end-to-end exactly-once fault-tolerance guarantees through mechanisms like checkpointing and Write-Ahead Logs.
- Structured Streaming treats a live data stream as a continuously appended table, enabling users to perform operations like aggregations and event-time windows seamlessly



STREAMING DATAFRAMES / DATASETS IN SPARK

- In Structured Streaming, a streaming DataFrame or Dataset represents a continuously updating table. Each row corresponds to a new data item arriving in the stream.
- Operations can be performed on these streaming DataFrames, as filtering, aggregating, and joining with other DataFrames.
- The operations are defined in a way that they can be executed incrementally, allowing Spark to process new data as it arrives without needing to reprocess the entire dataset.



Data processing – Batch vs. Stream Processing

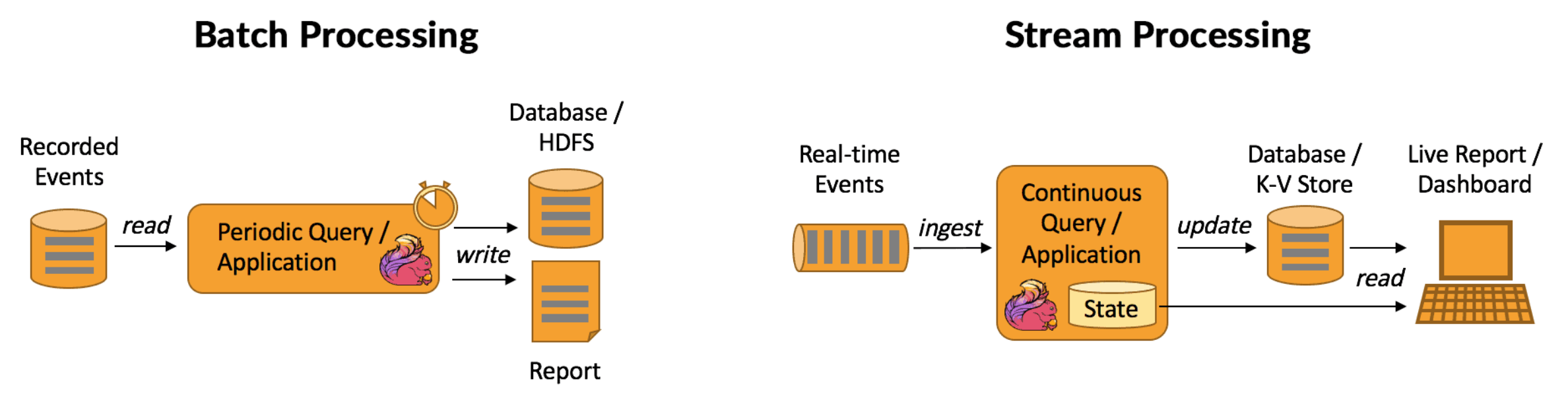
RECAP

Comparison

Feature	Batch Processing	Stream Processing
Data Ingestion	Collected over time	Continuous
Processing Frequency	Scheduled intervals	Real-time
Latency	High	Low
Use Cases	Historical data analysis	Real-time monitoring
Complexity	Simpler	More complex

When to Use

- Batch Processing: When processing large volumes of historical data.
- Stream Processing: When immediate data processing and response are required.

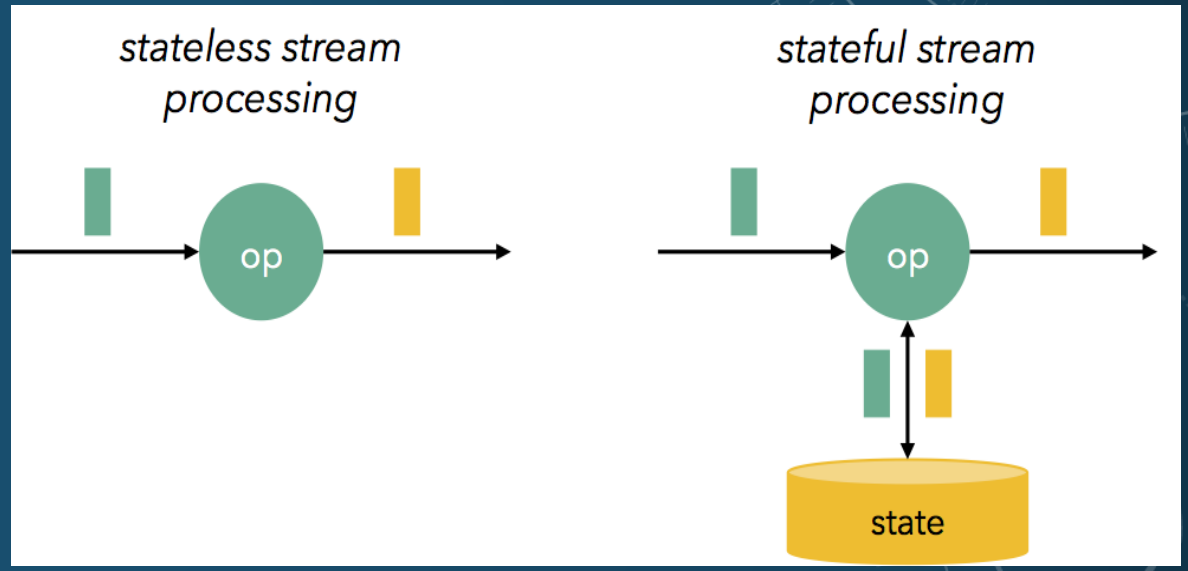


Data processing – Deep Dive into Stream Processing

RECAP

Stateless Processing

Stateless streaming processing is a type of data processing where the system does not maintain any state or memory of past events. Each incoming data item is processed independently, without relying on information from previous items.



Common use cases for stateless streaming processing:

- Ingesting, cleaning, and transforming data.
- Filtering out unwanted data based on specific criteria.
- Applying transformations to data, such as converting data formats or calculating derived values.
- Calculating basic aggregations like sums, averages, or counts over a window of data.

Data processing – Deep Dive into Stream Processing

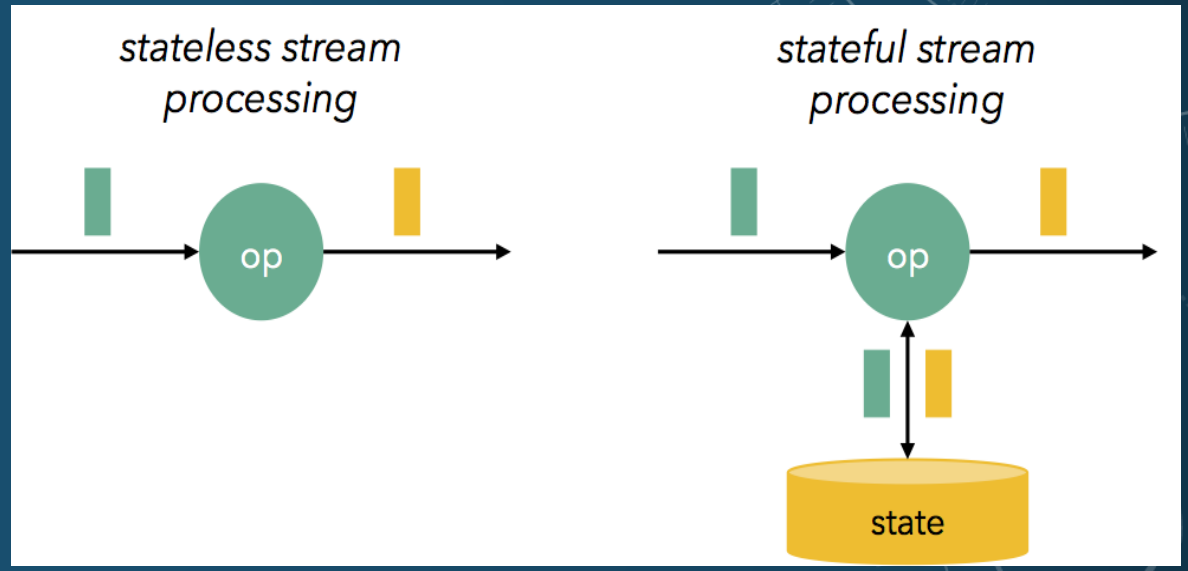
RECAP

Stateful Processing

Keeping track of state across multiple events.

Key concepts

- State: information about past events or calculations
- Stateful operators: maintain and update state. E.g.: *aggregate, reduce,* and *window*
- State store: storage mechanism for state. It can be in-memory, persistent storage (like HDFS or S3), or a combination of both.



Data processing – Deep Dive into Stream Processing

RECAP

Examples

Stateless:

Filtering financial transactions: Filtering out fraudulent transactions based on real-time rules.

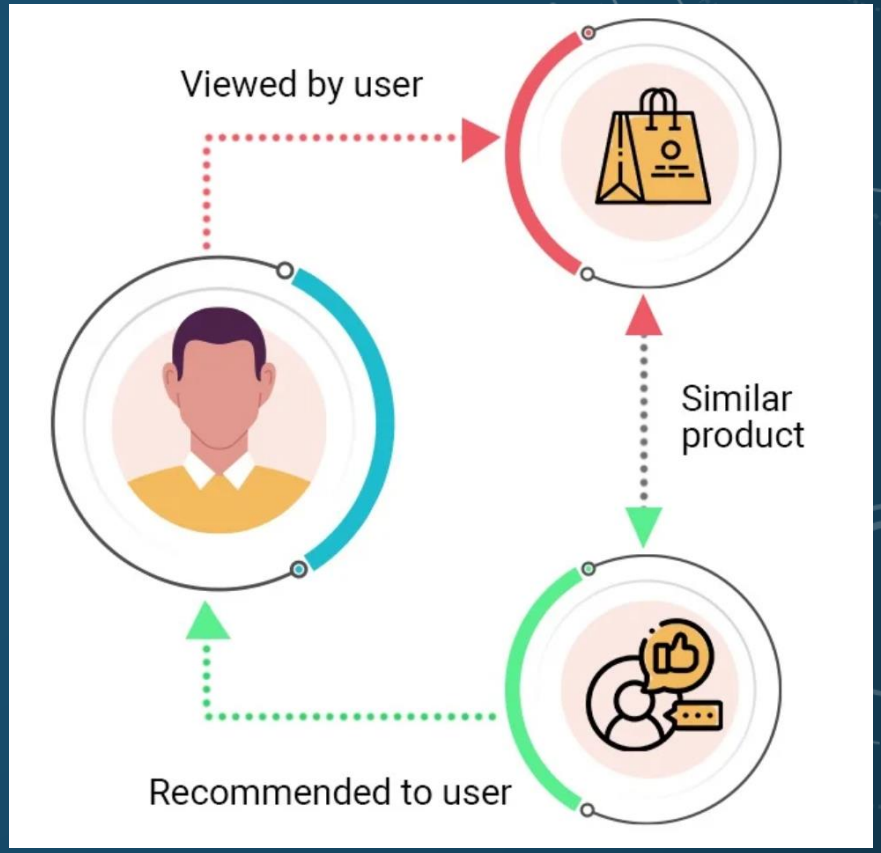
Analyzing IoT sensor data: Detecting anomalies in sensor data to identify equipment failures.

Stateful:

Recommender systems: Providing personalized recommendations based on user behavior and historical data.

Fraud detection: Detecting fraudulent activities by tracking user behavior over time and identifying suspicious patterns.

Inventory management: Tracking inventory levels in real-time and triggering replenishment orders when stock levels fall below a threshold.



Data processing – Deep Dive into Stream Processing

RECAP

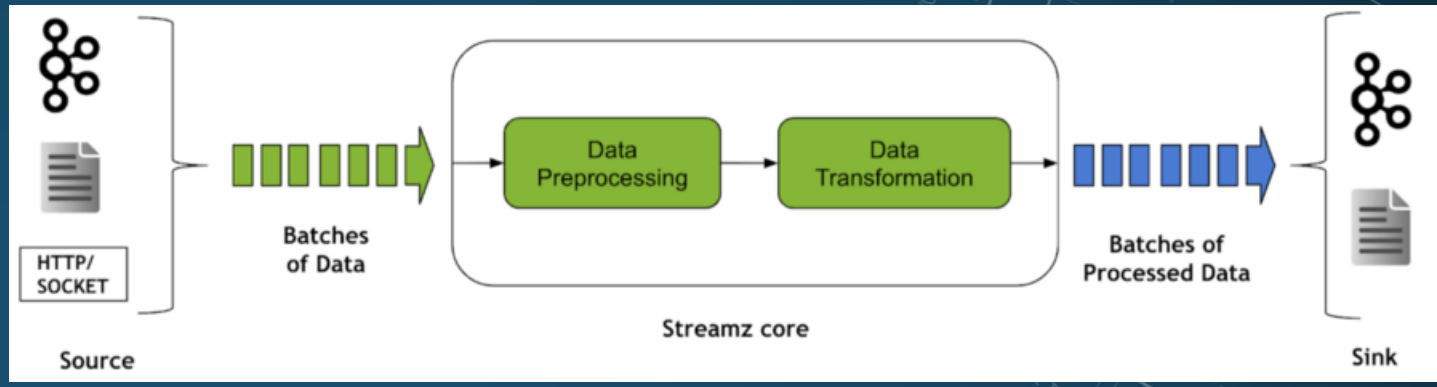
Types of time windows

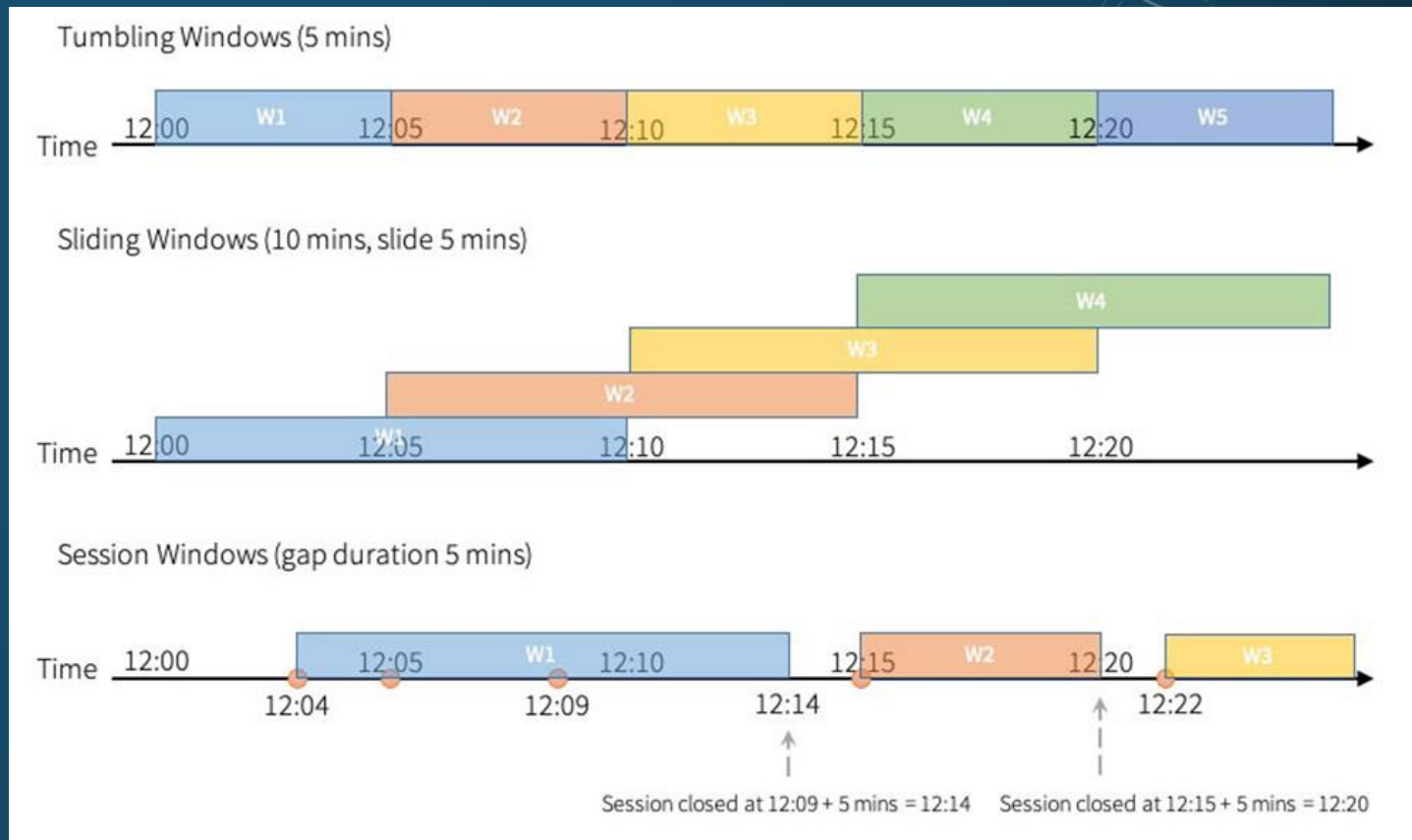
Windows

Time-based grouping of data streams

Types

- **Tumbling windows**: fixed-sized, non-overlapping, continuous time intervals; An input can only be bound to a single window.
- **Sliding windows**: fixed-sized, can overlap duration of slide is smaller than the duration of window. In this case an input can be bound to the multiple windows
- **Session windows**: dynamic size of the window length, depending on the inputs; starts with an input, and expands itself if following input has been received within gap duration. A session window closes when there's no input received within gap duration after receiving the latest input.





Data processing – Deep Dive into Stream Processing

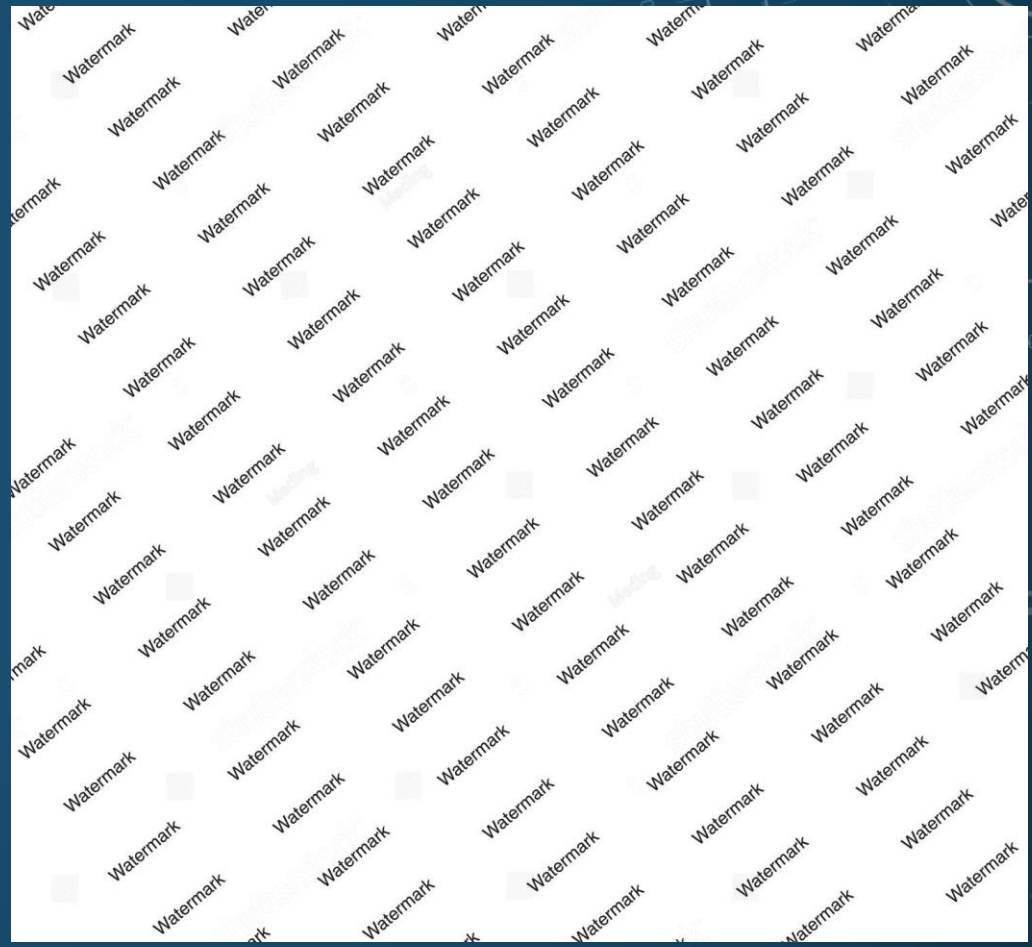
RECAP

Handle late events with watermark

Watermarks: a design or pattern embedded in paper during its manufacture.

However, in streaming processing, watermark is mechanism to handle late-arriving data.

Fyi: *The term "watermark" is a metaphor drawn from paper and printing. The watermark is a timestamp that's embedded in the stream of data*.



Data processing – Deep Dive into Stream Processing

RECAP

Handle late events with watermark

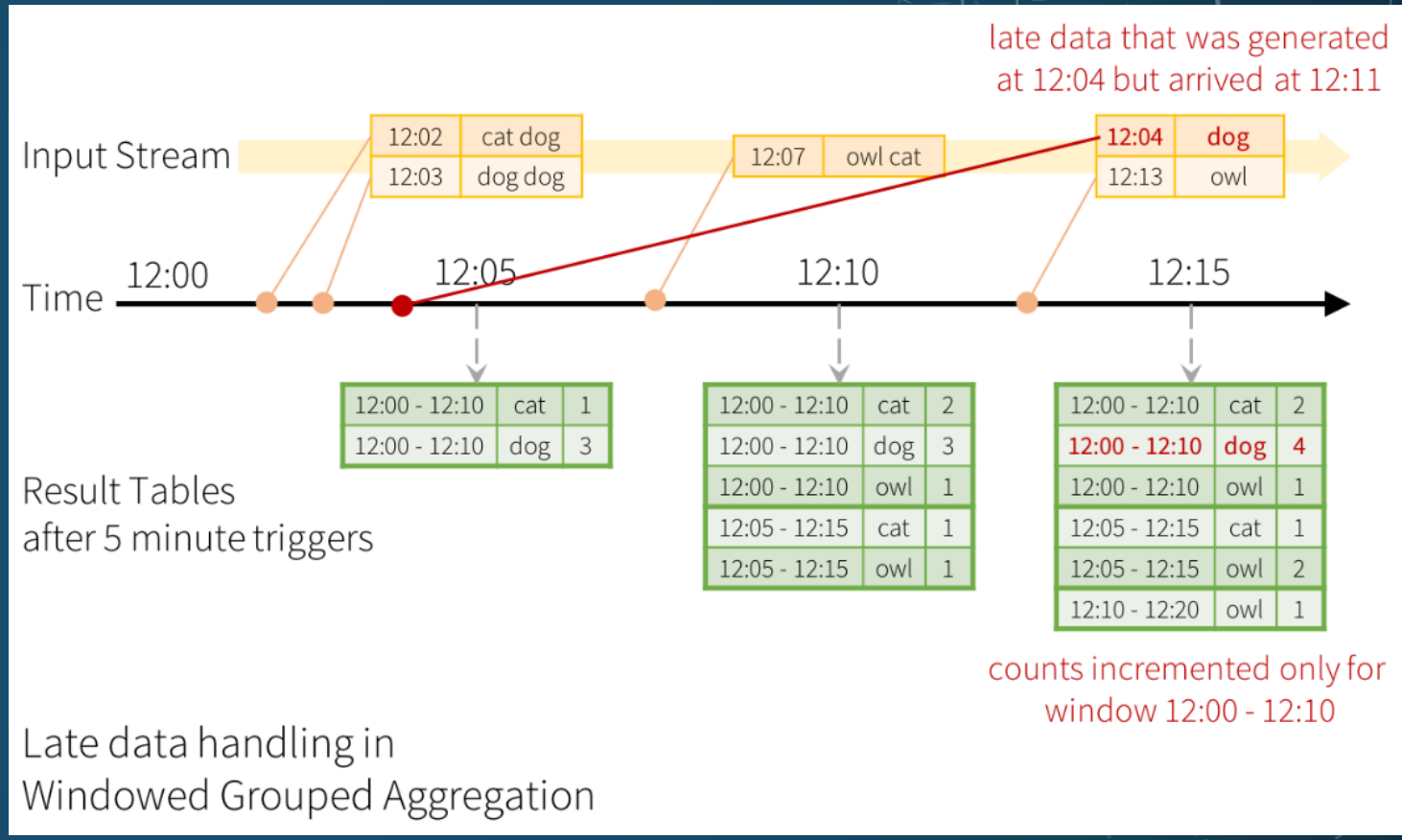
Watermarks: Mechanism to handle late-arriving data.

Types of timestamps:

- Event time: event occurred in the real world
- Arrived time / ingestion time
- Processing time
- Watermark timestamp: indicates the latest event time that has been processed by the system.

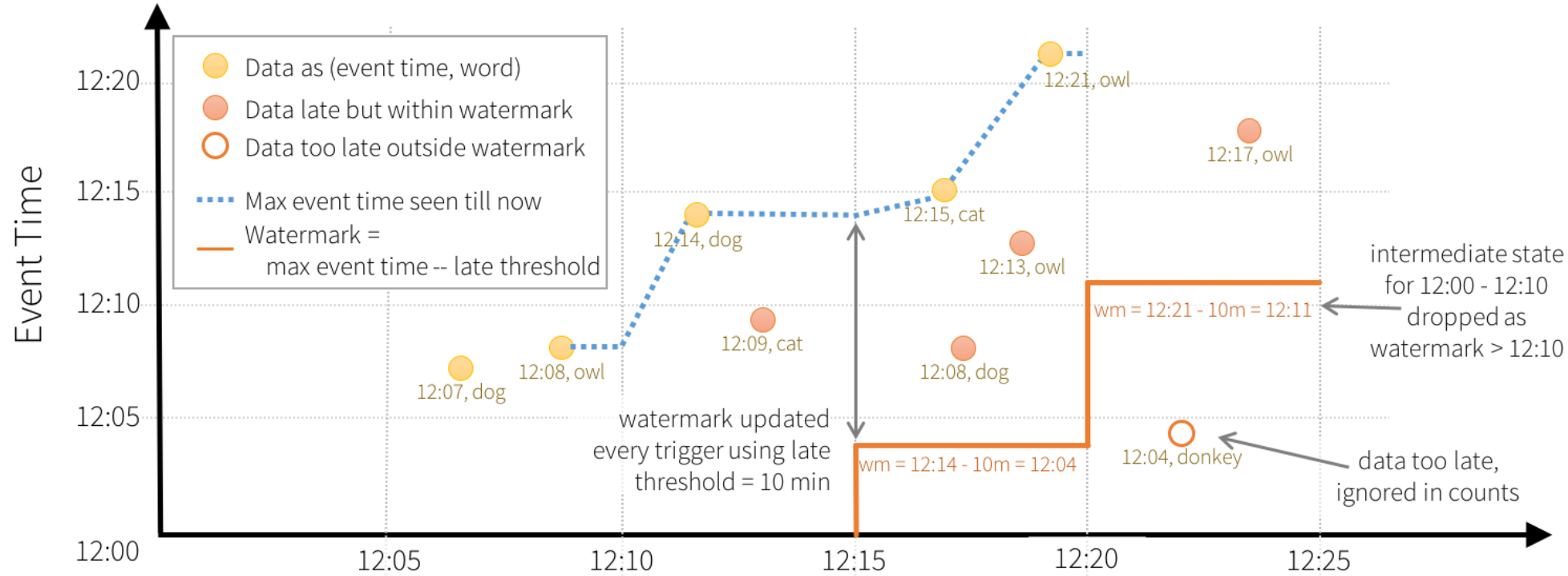
Late events handling:

- Buffering: store late events for later processing
- Processing with a delay: wait for a period, then process
- Dropping



Data processing – Deep Dive into Stream Processing

RECAP



Result Tables after each trigger

12:00 - 12:10	owl	1
12:00 - 12:10	dog	1
12:05 - 12:15	owl	1
12:05 - 12:15	dog	1

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

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

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

table *not* updated with too late data (12:04, donkey)

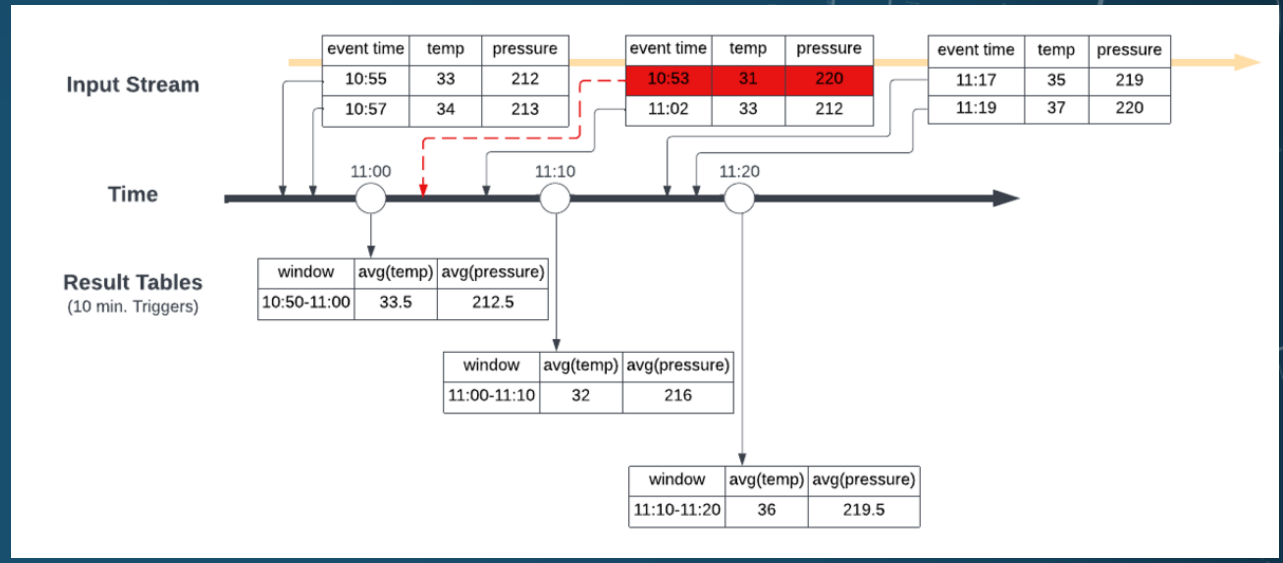
table updated with late data (12:17, owl)

purple rows are updated rows that are written to the sink as output

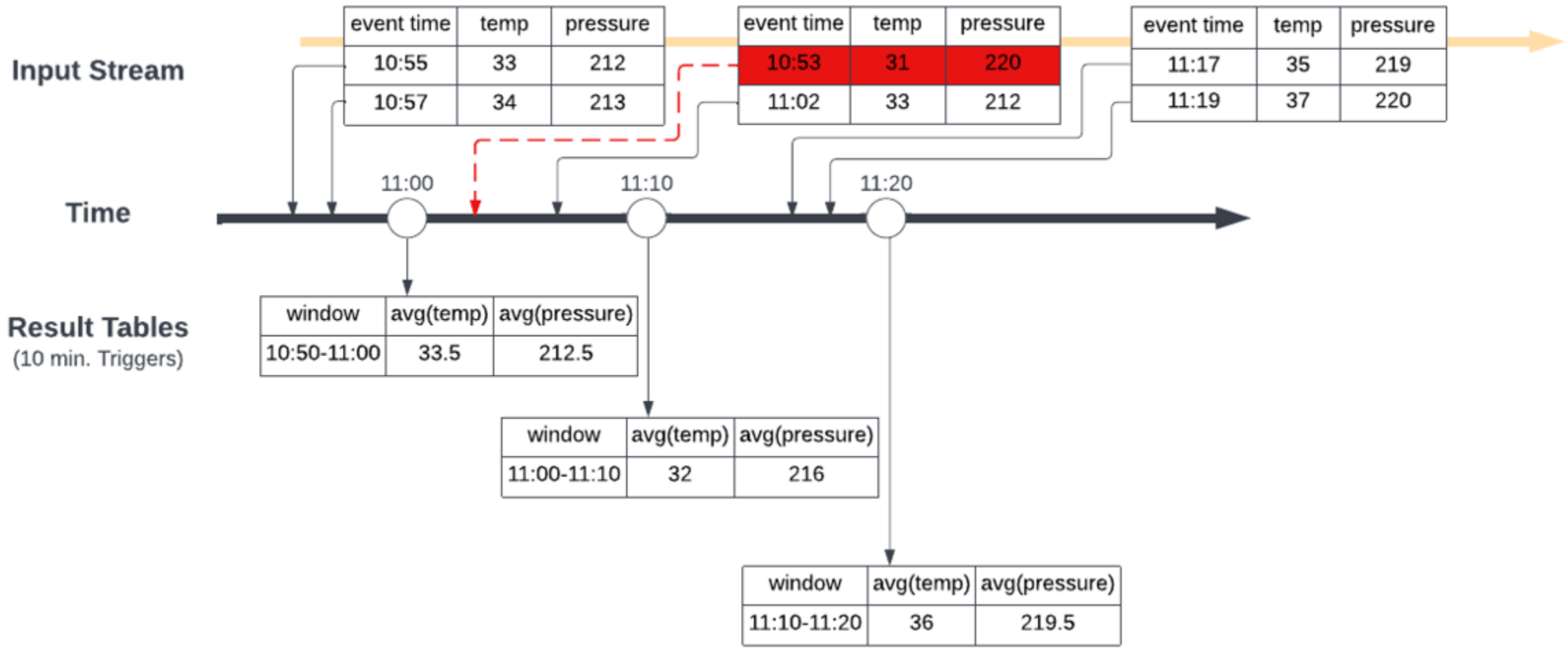
EVENT-TIME AND WATERMARKING

- Event-time refers to the time at which an event occurs, as opposed to the time it is processed.
- In streaming applications, handling event-time is crucial for accurate processing, especially when dealing with out-of-order data.

- Watermarking is a technique used to manage late data in streaming applications. It allows you to specify a threshold for how late data can arrive and still be processed.
- By using watermarks, you can ensure that your computations remain efficient and that you do not wait indefinitely for late data, thus maintaining the integrity of your results



EVENT-TIME AND WATERMARKING



Data processing – Deep Dive into Stream Processing

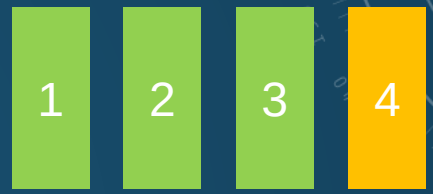
RECAP

Output modes

Append

This mode appends new rows to the existing output data.

It's suitable for use cases where you want to accumulate data over time.



Complete

This mode overwrites the entire output dataset with each new batch of data.

It's useful when you want to get the latest state of the data.

Update

This mode updates existing rows based on a key and updates the output dataset.

It's suitable for use cases where you want to modify existing data based on new information.

Data processing – Deep Dive into Stream Processing

NEW

Output Sinks and Supported Output Modes

Format	Append	Complete	Update
Delta	Yes	Yes	Yes
Parquet	Yes	No	No
JSON	Yes	No	No
CSV	Yes	No	No
ORC	Yes	No	No
Text	Yes	No	No
Kafka	Yes	Yes	Yes
Console	Yes	Yes	Yes
Memory	Yes	Yes	Yes

DIFFERENCES BETWEEN STRUCTURED STREAMING, DSTREAMS, AND BATCH PROCESSING

Criteria	Structured Streaming	Dstreams (Discretized Streams)	Batch Processing
Data Processing Model	Continuous processing of streaming data as a table.	Micro-batch processing of data streams.	Processing of static datasets in batches.
API	Uses DataFrame and Dataset APIs for both batch and streaming.	Based on RDDs (Resilient Distributed Datasets).	Primarily uses RDDs or DataFrames for batch jobs.
Latency	Low latency with options for continuous processing.	Higher latency due to micro-batch intervals.	High latency; results are available after batch completion.
Fault Tolerance	Provides exactly-once processing guarantees.	Offers at-least-once processing guarantees.	Typically allows for reprocessing of failed batches.

DIFFERENCES BETWEEN STRUCTURED STREAMING, DSTREAMS, AND BATCH PROCESSING

Criteria	Structured Streaming	Dstreams (Discretized Streams)	Batch Processing
Complexity	More complex due to continuous data flow and state management.	Simpler than Structured Streaming but less flexible.	Generally simpler as it deals with finite data chunks.
Scalability	Highly scalable, designed for both high throughput and low latency.	Scalable but less efficient for high-velocity data.	Scalable for large volumes but may require more resources for batch jobs.
Use Cases	Real-time analytics, monitoring, and event-driven applications.	Streaming data processing for applications like log analysis.	ETL processes, reporting, and large-scale data analysis.
Event Time Handling	Supports event-time processing and watermarking for late data.	Limited support for event-time; primarily processes based on arrival time.	Does not handle event-time; processes data as it arrives in batches.

APACHE KAFKA – ARCHITECTURE & CORE CONCEPTS

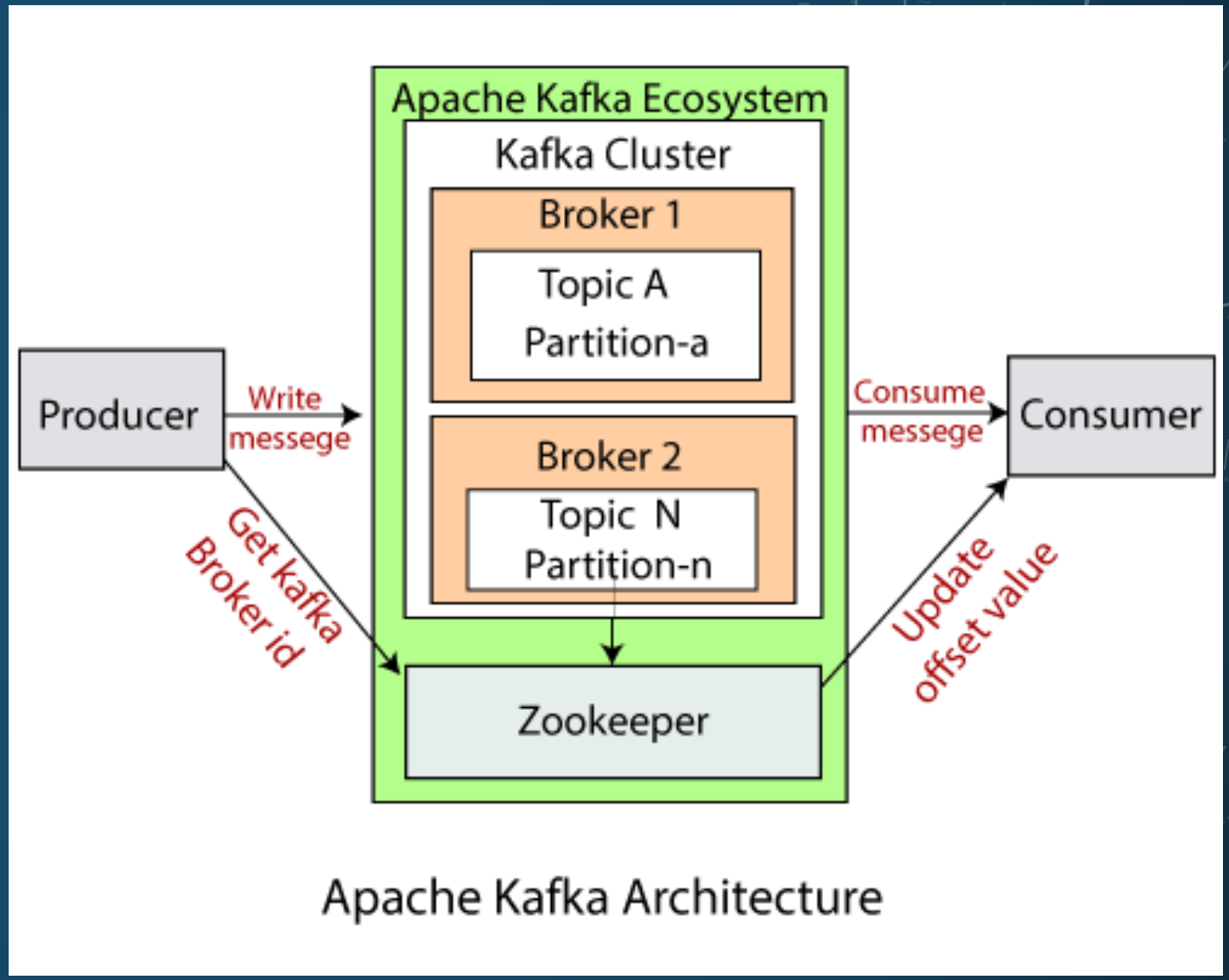
1. A **topic** is a category or feed name to which records are published. Topics are partitioned for scalability.

2. **Producers** are applications that publish messages to Kafka topics. They can choose which partition to send messages to.

3. **Consumers** read messages from topics. They can be part of a consumer group, allowing for load balancing and fault tolerance.

4. A **consumer group** consists of one or more consumers that work together to consume topics. Each message is delivered to one consumer within the group.

5. Each topic can be divided into **partitions**, which are ordered and immutable sequences of messages. Partitions allow for parallel processing.



Apache Kafka Architecture

APACHE KAFKA – ARCHITECTURE & CORE CONCEPTS

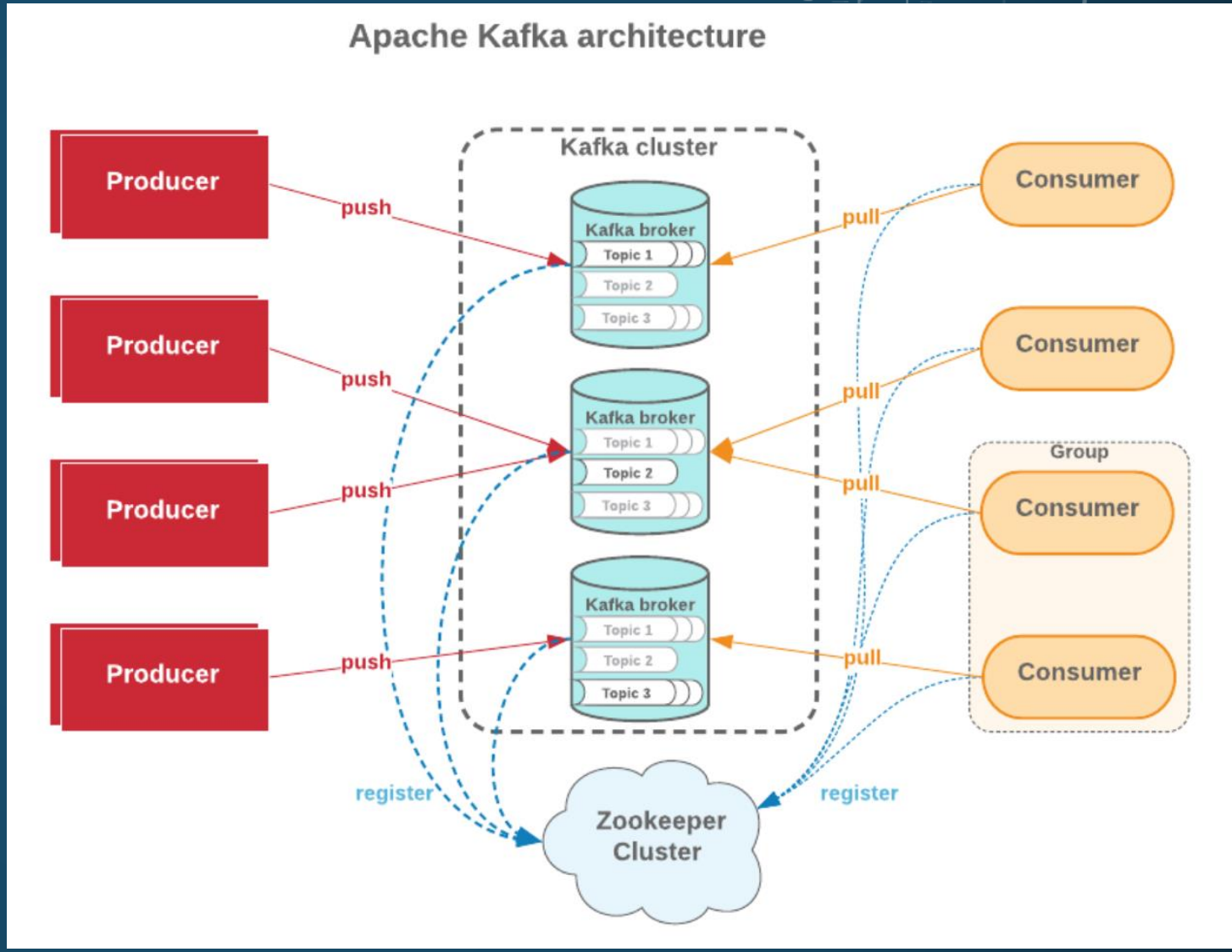
6. Each message within a partition has a unique **offset** that denotes its position. Consumers track offsets to know which messages have been processed.

7. Kafka runs in a cluster of servers, known as **brokers**. Each broker is responsible for storing and serving partitions of topics.

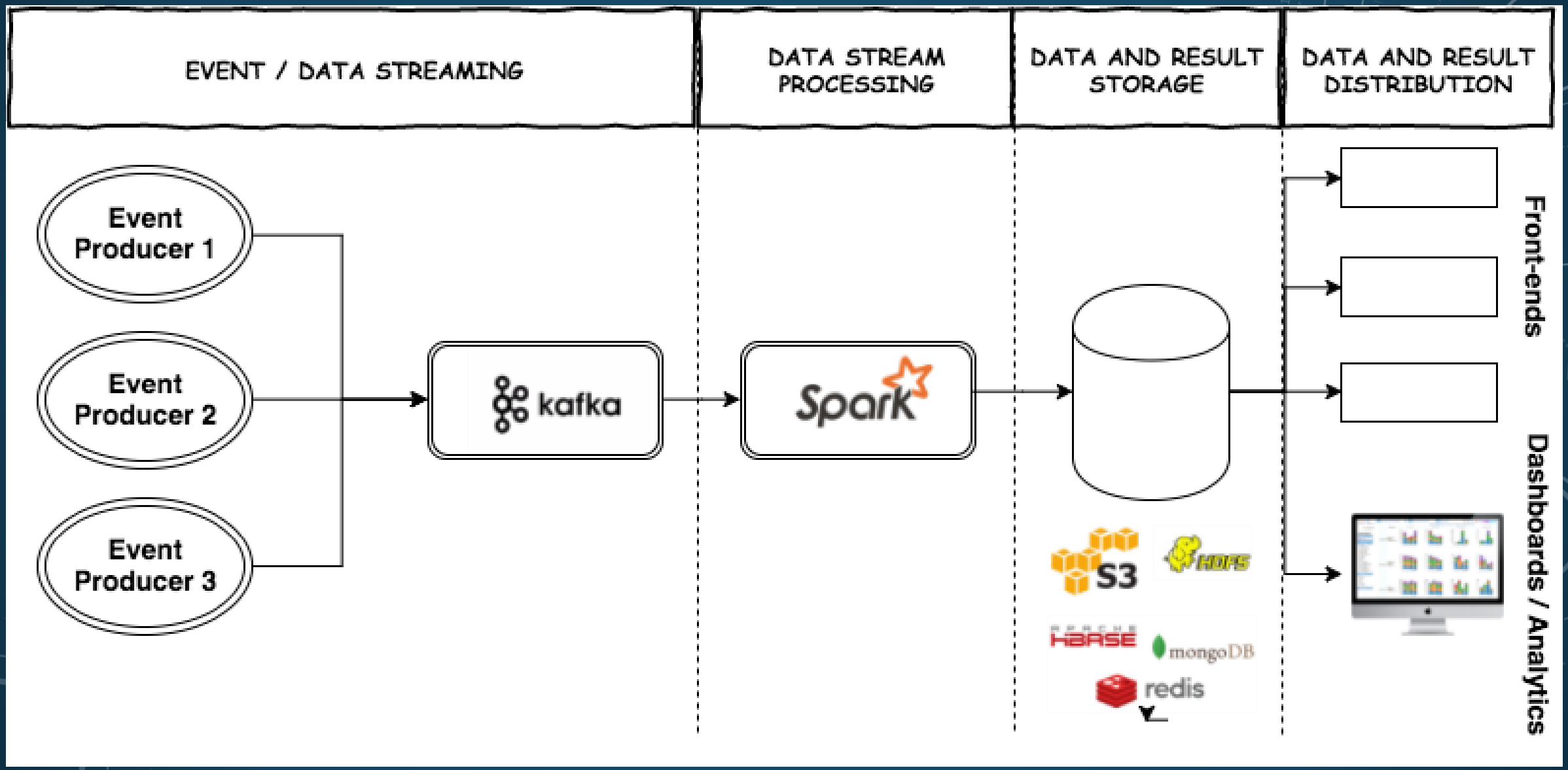
8. **Replication**: Kafka replicates partitions across multiple brokers to ensure fault tolerance. Each partition has a leader and multiple followers.

9. **Zookeeper** is used for cluster coordination, maintaining metadata about the Kafka cluster, and managing broker configurations.

10. **Kafka Connect**: A tool for connecting Kafka with external systems, such as databases and file systems.



APACHE KAFKA – SPARK INTEGRATION



LAB PRACTICE