UNIT III- REAL-TIME STREAM PROCESSING AND ANALYTICS

Distributed Stream Processing – Stateful and Stateless Processing – Windowing Techniques – Event Time vs Processing Time – Watermarks – Apache Spark Structured Streaming – Apache Flink – Kafka Streams – Apache Beam – Stream Joins – CEP (Complex Event Processing) – Real-Time Machine Learning – Online Learning Algorithms – AI-powered Stream Analytics – NoSQL Databases (MongoDB, Cassandra, Redis) – Time-Series Databases (InfluxDB, TimescaleDB).

DISTRIBUTED STREAM PROCESSING

Distributed Stream Processing is a technique used to process **continuous streams of data in real time** across multiple computers (nodes).
Instead of storing data first and processing later (batch processing), stream processing analyzes data as soon as it arrives.
Distributed Stream Processing (DSP) is the method of ingesting, transporting, and analyzing data as it arrives, rather than waiting for batch accumulation.
It treats **events** (sensor readings, clicks, transactions) as the central input/output objects.
DSP systems are designed for **scalability, fault tolerance, and real-time responsiveness**.

Why Distributed?

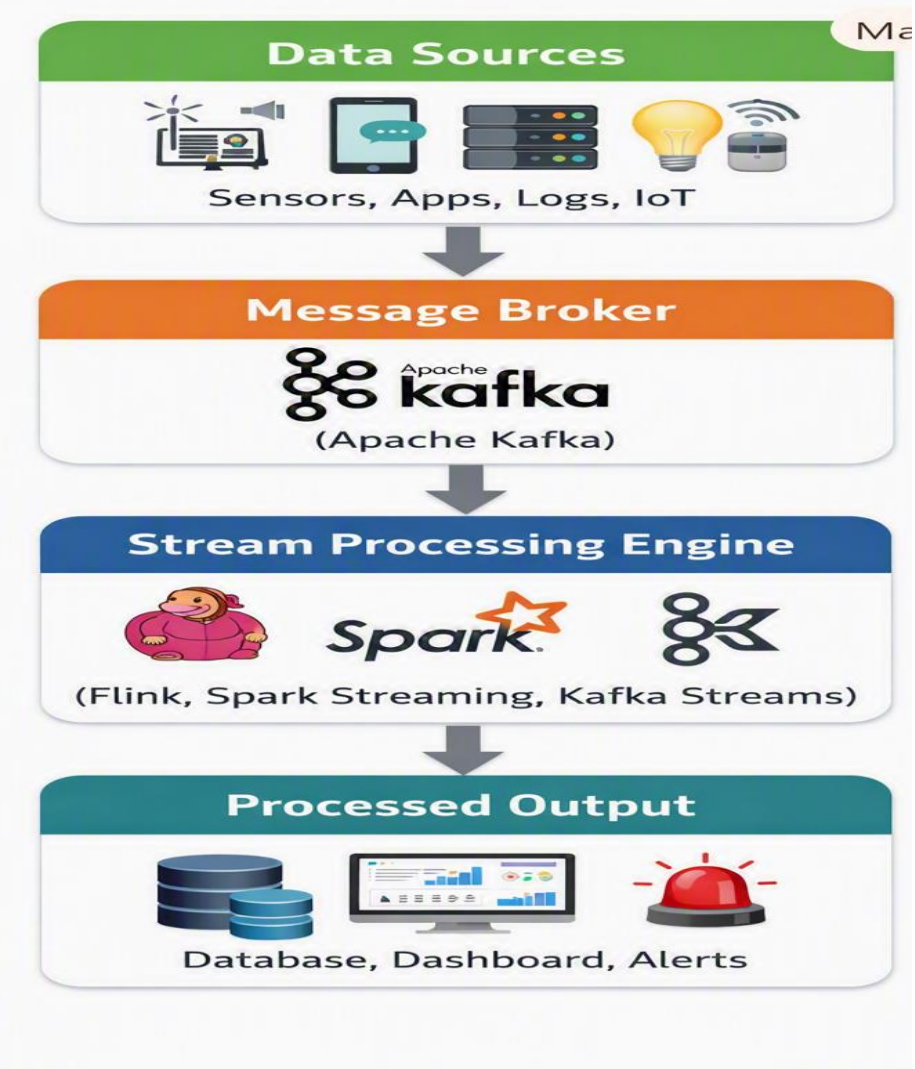
A single machine cannot efficiently process huge volumes of streaming data.
Therefore, the workload is distributed across multiple servers to achieve:

- High scalability
- Fault tolerance
- Low latency
- High throughput

Key Characteristics

Continuous Processing: Each event is processed immediately upon arrival.
Parallelism: Workload distributed across multiple nodes for high throughput.
Low Latency: Results delivered in milliseconds.
Fault Tolerance: Checkpointing and replication ensure reliability.
Scalability: Can handle millions of events per second.

Stream Processing Architecture



Step 1: Data Sources

These are the origins of streaming data.
Examples: IoT sensors, mobile apps, server logs, social media feeds, or transaction systems.
Data is generated continuously and often in large volumes.
Each event (e.g., temperature reading, user click, log entry) is sent to the next layer for ingestion.

Step 2: Message Broker

The **message broker** acts as a buffer and transport layer between producers (data sources) and consumers (processing engines).
Apache Kafka is the most common broker used.
It stores streams of records in **topics** and ensures reliable delivery.

Kafka decouples data producers and consumers, enabling scalability and fault tolerance.

Step 3: Stream Processing Engine

This layer performs **real-time computation** on incoming data.
Frameworks like **Apache Flink**, **Spark Structured Streaming**, and **Kafka Streams** process events as they arrive.

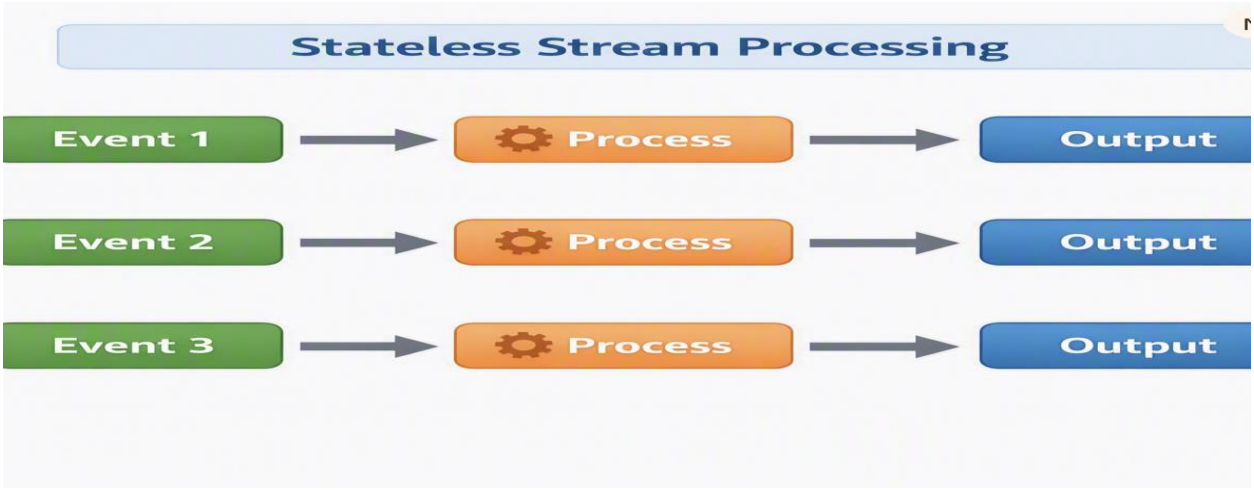
Operations include:

Filtering and transformation
Aggregation (sum, average, count)
Windowing (time-based grouping)
State management (tracking sessions or trends)
The engine applies logic defined by developers to derive insights or trigger actions.

Step 4: Processed Output

The results of stream processing are sent to output systems.
Common destinations:
Databases → for storage and further analysis
Dashboards → for visualization and monitoring
Alerts → for real-time notifications or automated responses
This enables immediate decision-making based on live data.

Stateless Processing
Stateless Processing processes each incoming event independently, without remembering any previous events.
Each record is treated separately.



No information is stored between events.

Characteristics

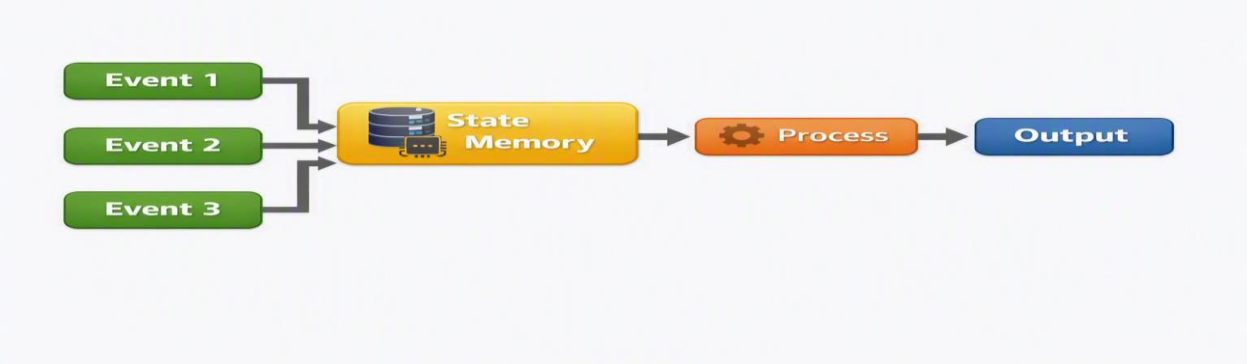
- No memory of previous events
- Easy to scale
- Fast processing
- Low resource usage

Examples

- Convert temperature from Celsius to Fahrenheit
- Filter records with age > 18
- Convert text to uppercase
- Data validation

Stateful Processing

Stateful Processing remembers previous events while processing new ones.
The system stores intermediate information (called **state**) to perform calculations across multiple events.

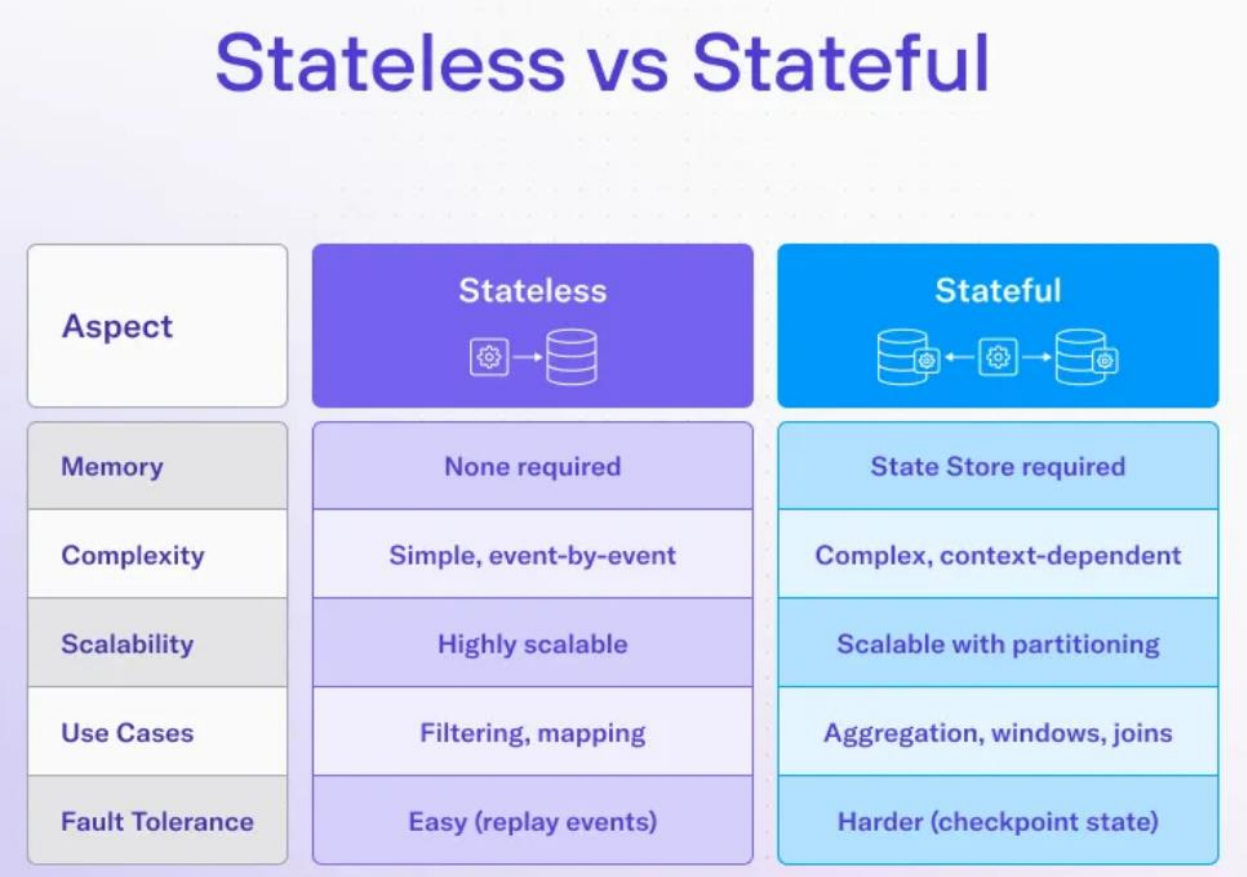


Characteristics

- Maintains state information
- Supports aggregations
- Enables window operations
- Detects patterns over time

Examples

- Count website visitors
- Average temperature over the last 10 minutes
- Detect fraudulent transactions
- Count product sales
- Session management

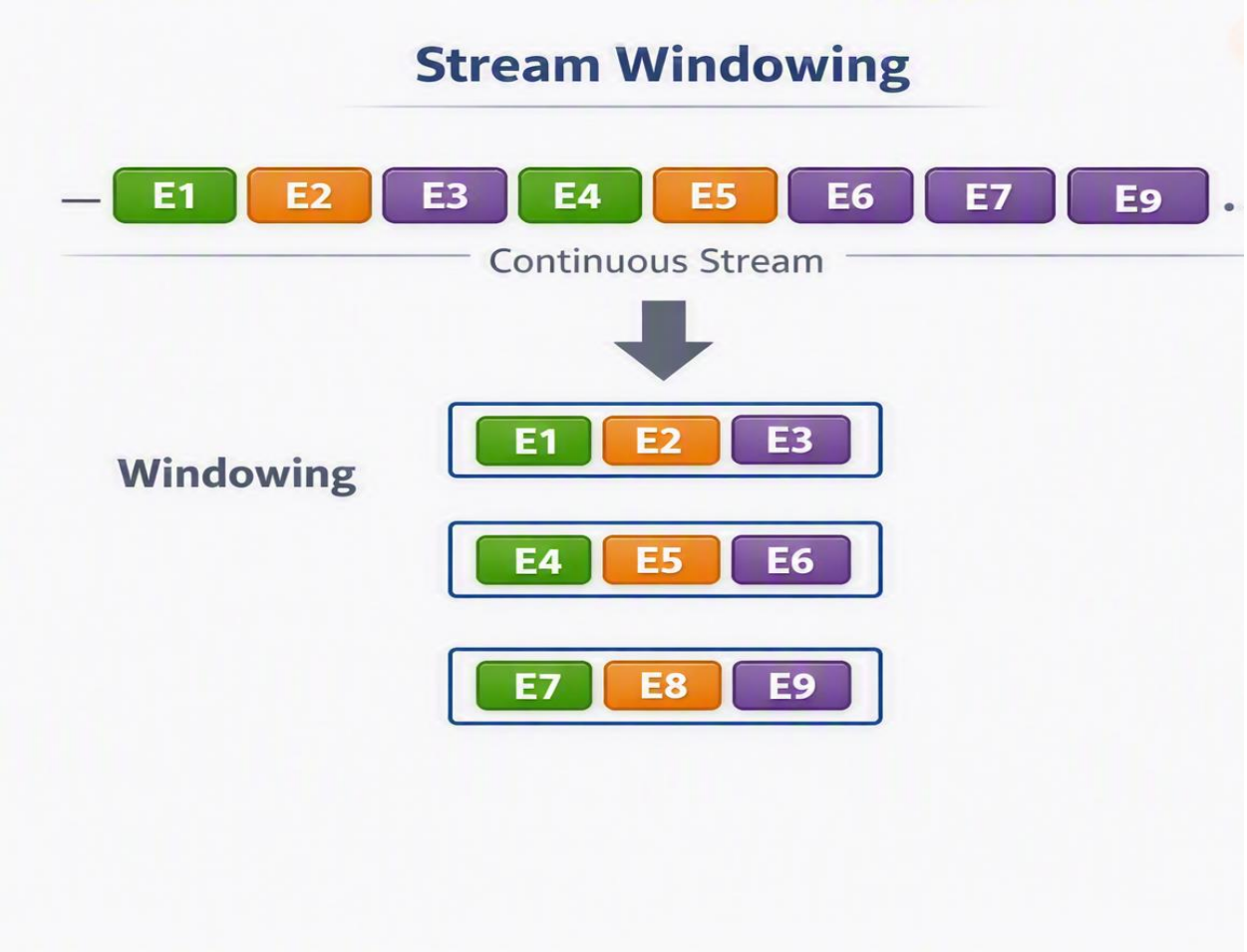


When to Use Stateless vs. Stateful?

Stateless: Opt for stateless processing when your application involves simple transformations, filtering, or enrichment without historical context. It's ideal for lightweight, high-throughput pipelines where speed and simplicity matter.
Stateful: Choose stateful processing for analytics requiring memory — aggregations (e.g., running averages), sessionization, or pattern matching. It's essential when the "why" behind an event depends on "what came before."

WINDOWING TECHNIQUES

Windowing is a technique used in **stream processing** to divide an **infinite stream of data into finite chunks (windows)** so that operations like counting, averaging, summing, and analysis can be performed efficiently.
Since streaming data never ends, windowing enables meaningful computations over a specific time interval or group of events.



2. Time Semantics
Time semantics determine **how events are assigned to windows**.

Event Time

The **actual time when the event occurred**, stored in the event itself.

Characteristics

- Uses timestamp inside the event
- Handles delayed and out-of-order events
- Most accurate for analytics

Challenges

- Out-of-order events
- Clock synchronization
- Late-arriving data

Applications

- Financial transactions

IoT sensor analysis
Historical analytics

Processing Time

The **time at which the stream processing engine processes the event**.

Characteristics
Uses system clock
Simple implementation
Low latency
Advantages
Fast processing
Easy to implement
Applications
Real-time dashboards
Live monitoring
Instant alerts

Ingestion Time

The **time when the event enters the streaming platform** (e.g., Kafka).

Characteristics
Assigned by the message broker
More consistent than processing time
Simpler than event time

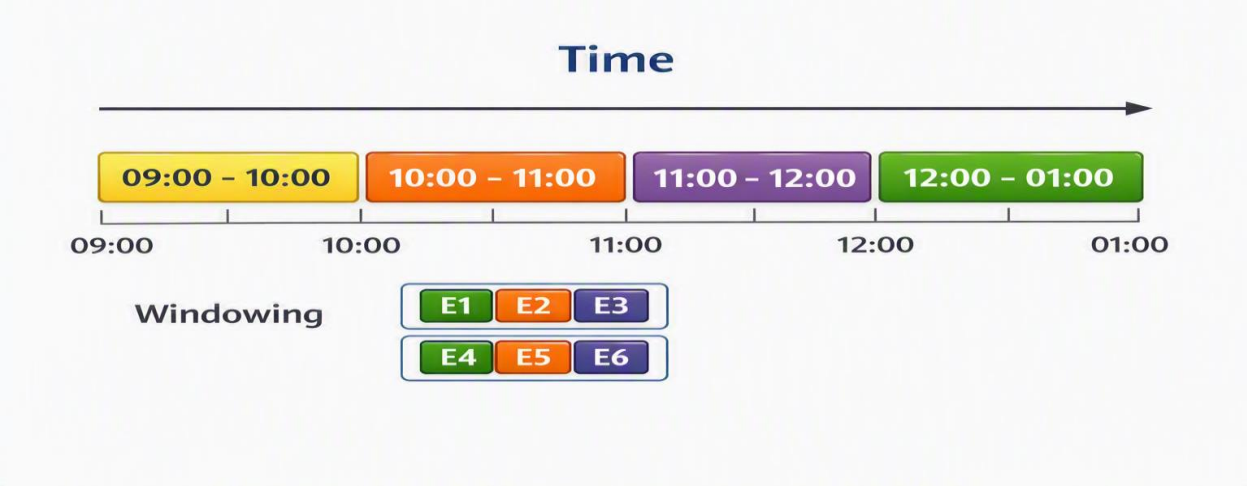
Applications
Log processing
Approximate real-time analytics

Comparison of Time Semantics

Feature	Event Time	Processing Time	Ingestion Time
Timestamp Source	Event	System Clock	Message Broker
Accuracy	High	Medium	Medium
Handles Late Data	Yes	No	Limited
Complexity	High	Low	Medium
Typical Use	Historical Analytics	Live Monitoring	Log Processing

3. TYPES OF WINDOWS

Tumbling Window
A **fixed-size, non-overlapping** window.
Each event belongs to **exactly one window**.



Characteristics

- Fixed duration
- No overlap
- Easy to implement

Applications

- Hourly sales

Daily reports
Monthly billing

Example
Sales

9-10 AM = ₹5000
10-11 AM = ₹6500
11-12 PM = ₹7200

Sliding Window
A **fixed-size window** that **slides** at regular intervals.
Events may belong to **multiple windows**.



Characteristics
Overlapping windows
Better trend analysis
More computations

Applications
Moving average
CPU monitoring
Stock prices
Temperature analysis

Session Window
A **dynamic window** based on user activity.
A new window starts after a specified **inactivity gap**.



Characteristics

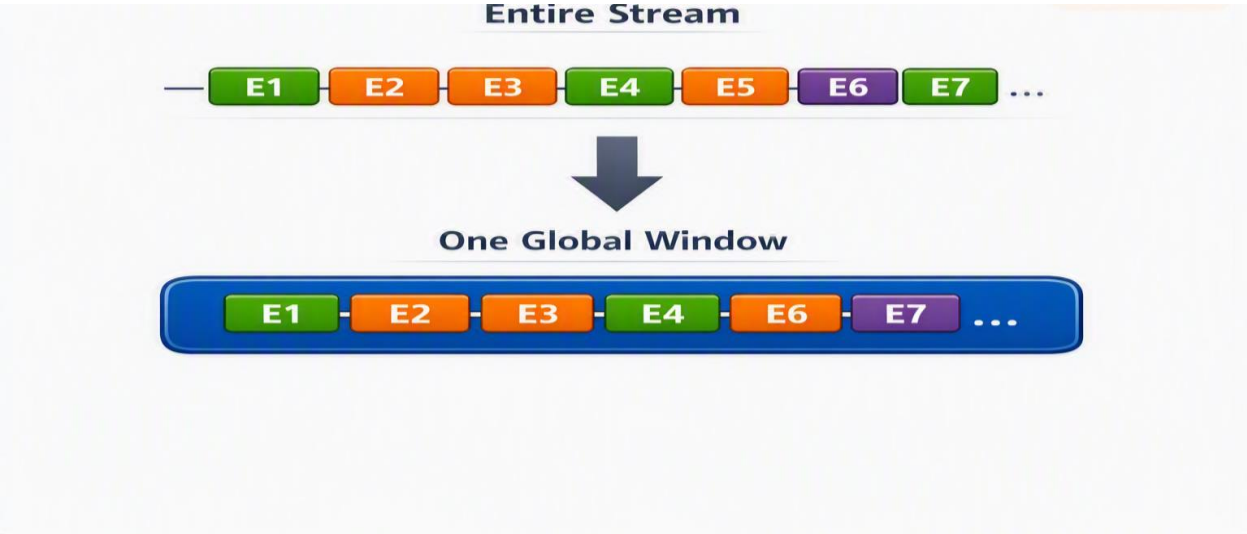
- Variable window size
- Based on inactivity timeout
- Groups related events

Applications

- Website sessions
- Mobile app usage
- IoT device activity
- Fraud detection

Global Window

All events belong to **one single window**.
Processing occurs only when manually triggered.



Applications

Batch analytics
End-of-day reports
Machine learning model updates

4. Advanced Windowing Concept: **Watermarking**

Definition

A **watermark** is a **time-based marker** used in stream processing that indicates **how far the system has progressed in event time**. It tells the processing engine that **all events with timestamps earlier than the watermark are expected to have arrived**. Any event arriving after this point is considered a **late event**.

In simple terms, a watermark helps the system decide **when it is safe to process and close a window**, even if some events arrive out of order.

How Do Watermarks Work?

In real-world streaming systems, events **do not always arrive in the order they were generated**. This is called **out-of-order event arrival**. It occurs because of:

Network delays
Queueing delays
Different processing speeds
Clock differences between devices
Delays in upstream systems

For example, an event that occurred at **10:02** may arrive after an event that occurred at **10:04**.

Disorder Threshold (d)

To handle out-of-order events, streaming systems define a **disorder threshold**, represented by **d**.

The **disorder threshold (d)** is the **maximum amount of time an event is allowed to be late**.

If an event has timestamp **t**, then the system waits for possible late events that occurred between **t − d** and **t**.

Watermark Calculation

Suppose:

T = Maximum event timestamp seen so far

d = Maximum allowed delay (disorder threshold)

Then,

$$\textbf{Watermark} = T - d$$

This watermark indicates that the system believes all events with timestamps **earlier than or equal to T−d** should already have arrived.

Example

Assume:

Maximum event time observed (**T**) = **10:20**

Disorder threshold (**d**) = **5 minutes**
Then,
Watermark=10:20−5 minutes=10:15

This means:
Events with timestamps **up to 10:15** are expected to have arrived.
The system can safely process windows ending at **10:15**.
Any event arriving later with a timestamp before **10:15** is considered a **late event**.

Why Watermarks Are Important
Without watermarks:
- The system would wait forever for delayed events.
- Window results would never be finalized.

With watermarks:
- Windows can close at the right time.
- Processing remains accurate and efficient.
- Memory usage is reduced.
- Real-time analytics become possible.

Watermarks and Deduplication
Watermarks are also useful for **removing duplicate events** in streaming systems.

Example
Incoming stream:
Order 101
Order 102
Order 101 (Duplicate)
Order 103

The system waits until the watermark passes before deciding that no more copies of **Order 101** are expected.

After the watermark:
Order 101
Order 102
Order 103

Duplicates arriving after the watermark can be safely ignored or handled as late data.

Advantages of Watermarks

Handle out-of-order events effectively.
Enable event-time processing.
Decide when windows should close. Support deduplication and aggregations.
Improve real-time analytics while balancing latency and accuracy.