COMP5339: Data Engineering

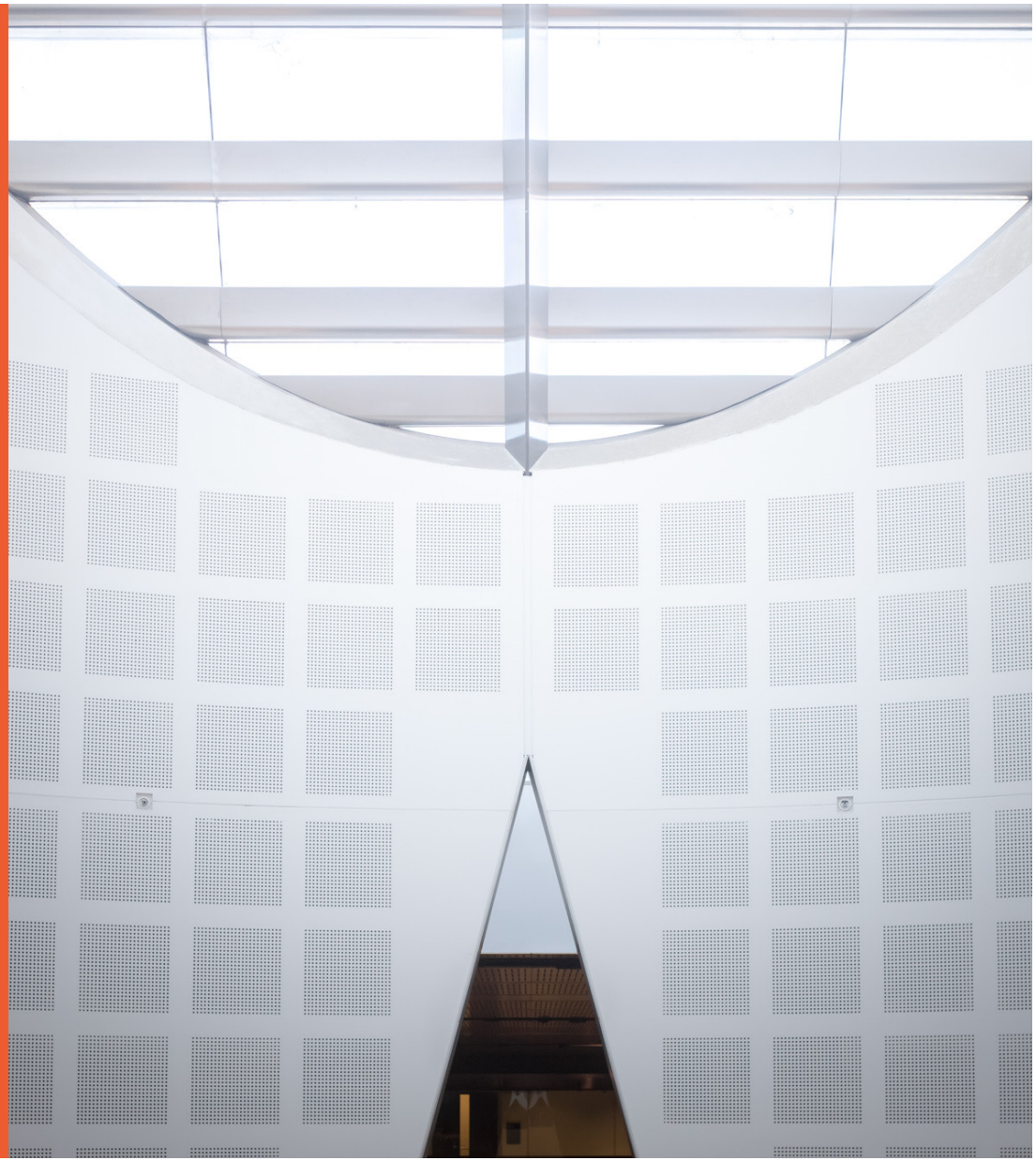
W9: Stream Data Processing

Presented by

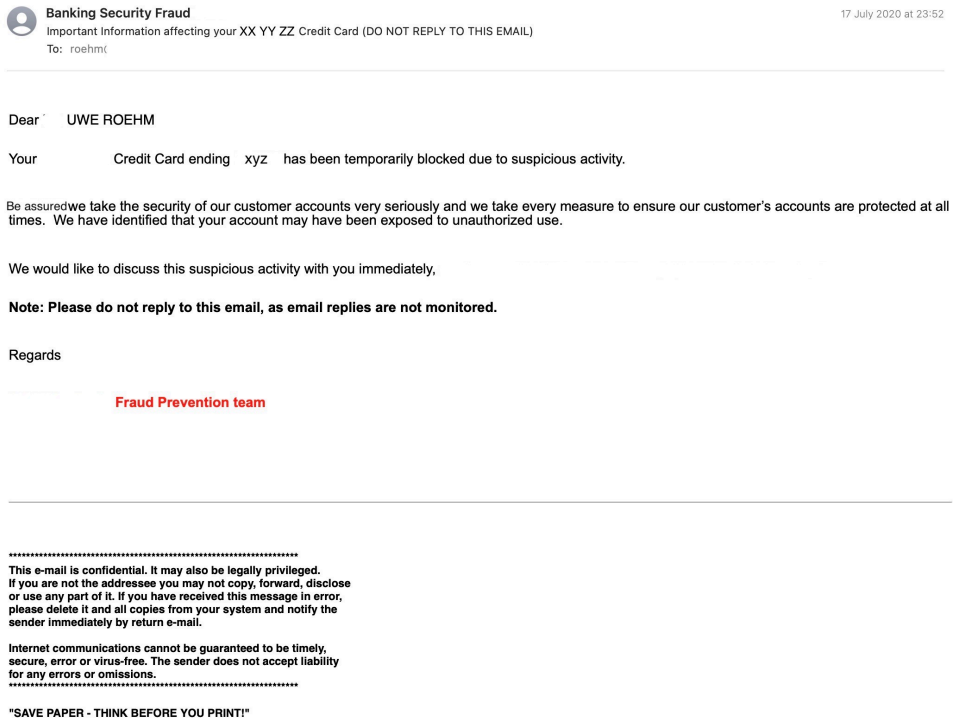
Uwe Roehm

School of Computer Science

THE UNIVERSITY OF SYDNEY



Motivating Example: Real-Time Fraud Detection



Banking Security Fraud
17 July 2020 at 23:52
Important Information affecting your XX YY ZZ Credit Card (DO NOT REPLY TO THIS EMAIL)
To: roehm(

Dear UWE ROEHM

Your Credit Card ending xyz has been temporarily blocked due to suspicious activity.

Be assuredwe take the security of our customer accounts very seriously and we take every measure to ensure our customer's accounts are protected at all times. We have identified that your account may have been exposed to unauthorized use.

We would like to discuss this suspicious activity with you immediately,

Note: Please do not reply to this email, as email replies are not monitored.

Regards

Fraud Prevention team

****
This e-mail is confidential. It may also be legally privileged. If you are not the addressee you may not copy, forward, disclose or use any part of it. If you have received this message in error, please delete it and all copies from your system and notify the sender immediately by return e-mail.

Internet communications cannot be guaranteed to be timely, secure, error or virus-free. The sender does not accept liability for any errors or omissions.
****

"SAVE PAPER - THINK BEFORE YOU PRINT!"

- Financial institutions constantly check for fraudulent transactions
- Has to be fast and with low delay
- At the same time, millions of transactions/sec to be checked for outliers (e.g. usage in a different country than residence)
- How to do so – on scale?

Why Stream Processing?

- Many applications with large streams of live data that needs to be processed immediately ('real-time')
  - Fraud detection
  - Cybersecurity
  - IoT sensor data
  - Industrial IoT
  - Online advertising
  - Log analysis

[Source: https://medium.com/dbconvert/data-stream-processing-b29023f28b47]

- Stream Processing Systems
  - "Data in motion": Process data as it flows without storing it persistently

Data Streams

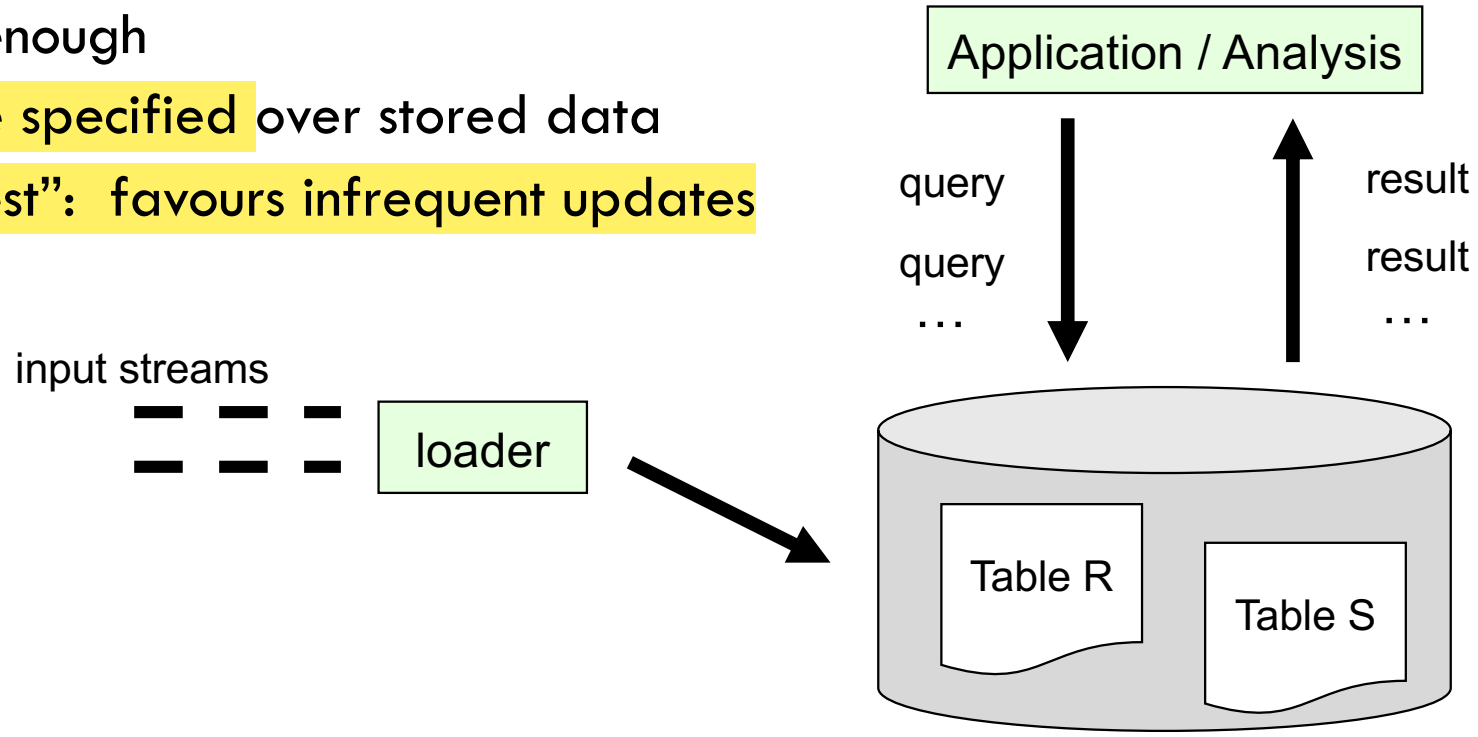
- A **data stream** is a (potentially unbounded) sequence of tuples
  - rapid, time-varying stream of data elements
- Occur in a variety of modern applications
  - **Transactional data streams**: log interactions between entities
    - credit card purchases by consumers from merchants
    - manufacturing processes
    - supply chain monitoring
    - ..etc
  - **Measurement data streams**: monitor evolution of entity states
    - IP network: traffic at router interfaces, network monitoring
    - Sensor networks: physical phenomena, road traffic
    - ..etc

记录实体之间发生的"事件 / 交互"的数据流

持续监测某个实体"状态随时间变化"的数据流

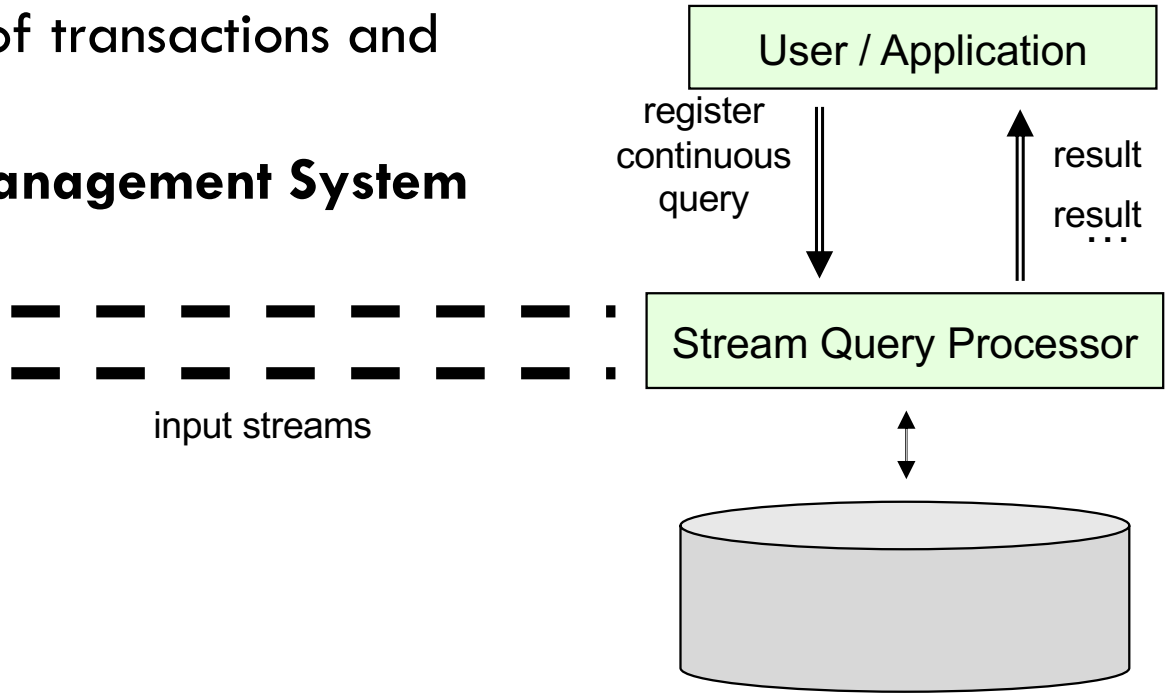
Traditional Data Processing Approach

- Why not use a DBMS?
  - Not agile enough
  - queries are specified over stored data
  - "data at rest": favours infrequent updates



Approach for Data Streams

- Two developments: application- and technology-driven
  - Need for sophisticated near-real-time queries/analyses
  - Massive data volumes of transactions and measurements
  - **DSMS: Data Stream Management System**



DBMS vs. DSMS

Database Systems

- Persistent relations
- Relations: set/bag of tuples
- Data size bound by what's stored
- Data Update: modifications
- Query: transient / one-time
- Query Answer: exact
- Access plan determined by query processor and physical DB design

Data Stream Systems

- Transient streams
- Relations: sequence of tuples have order
- Unbound data
- Data Update: appends
- Query: continuous / persistent
- Query Answer: approximate
- Unpredictable data characteristics and arrival patterns

Data Streaming Platforms

Stream Processing Architectures

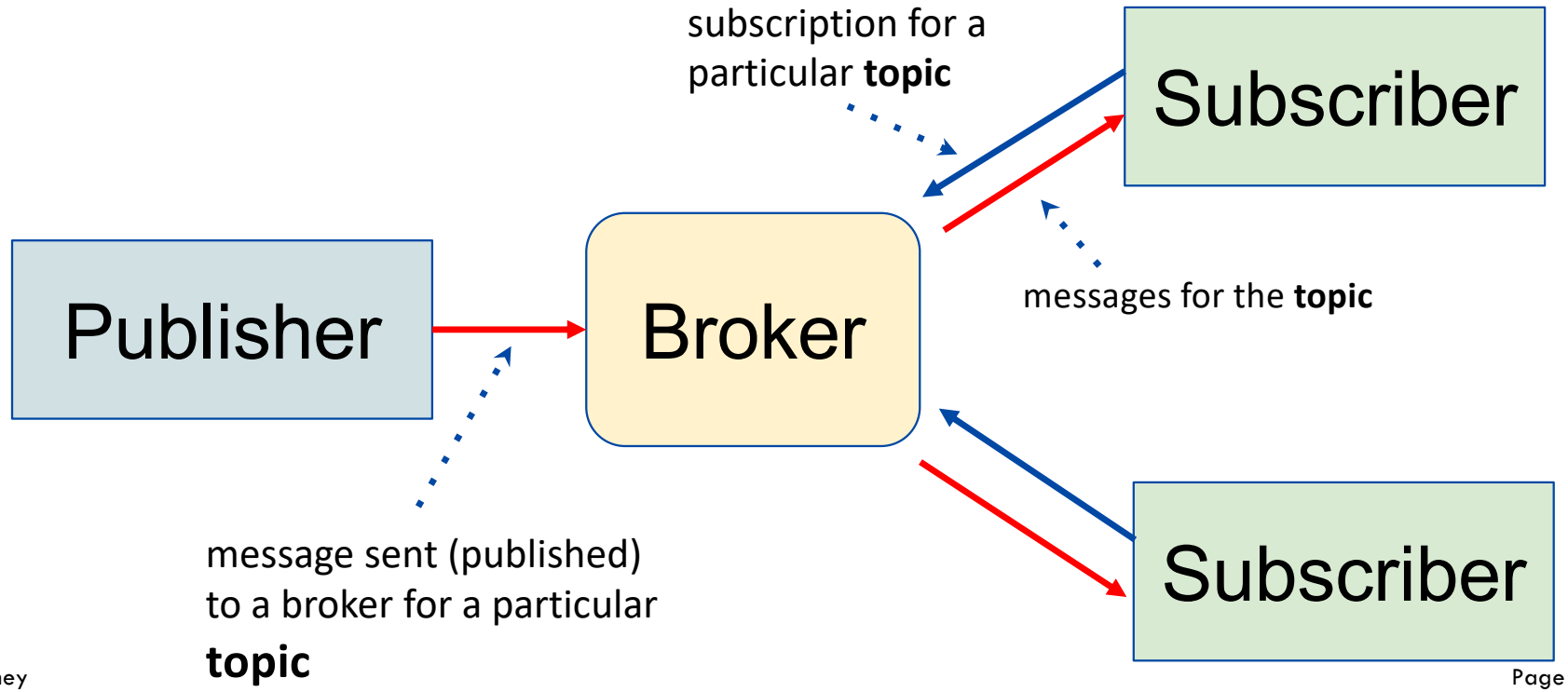
- Stream Processing Architectures need to address two issues:
  1. ability to consume constant stream of events, ideally with QoS guarantees
  2. ability to execute query processing on stream data with clear semantics

- State-of-the-art for a scalable streaming architecture is that these two aspects are handled by separate systems
  - Publish/Subscribe Messaging System
    - Handles one or multiple data streams that applications can subscribe to
  - Data Stream Processors
    - Efficient query processing over data streams

Publish/Subscribe

发布/订阅模式中，发布者不直接把消息发给具体接收者，而是按"主题（topic）"发布；订阅者只接收自己订阅的主题的消息。

- In software architecture, publish/subscribe is a messaging pattern where publishers categorize messages into classes ('topics') that are received by subscribers. This is contrasted to the typical messaging pattern model where publishers send messages directly to subscribers.

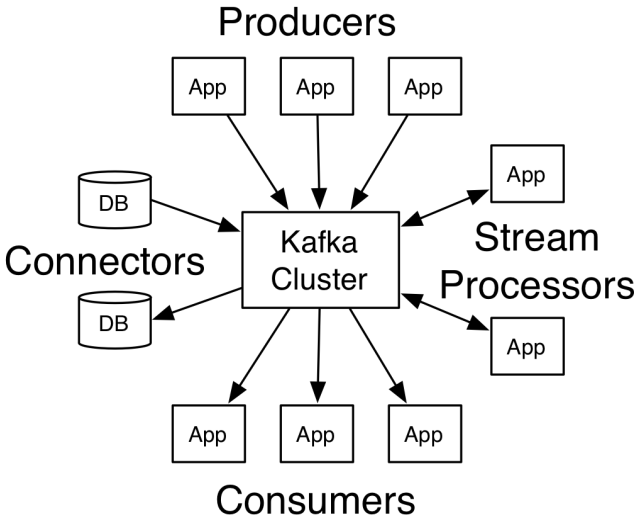


Publish/Subscribe

- Publishers publish messages to topics (or feed) of their choice.
- Data stream is unbound and broken into a sequence of individual data records (aka messages).
- A record is the atomic data item in a data stream.

Example 1: Apache Kafka

- Apache Kafka is a distributed publish-subscribe messaging system
  - maintains feeds of messages from one or more *producers*
  - feeds can be subscribed to by *consumers*
  - Apache Flink and Spark often used as stream processors



- Three key capabilities
  1. publish and subscribe to streams of records
  2. can store streams of records in a fault-tolerant way
  3. lets Apps process streams of records as they occur

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

Example 2: MQTT (Message Queuing Telemetry Transport)

- MQTT is a lightweight pub/sub system designed for streams of sensor data in the Internet of Things (IOT)
  - Many choices for broker and client implementations.

- Broker
  - Eclipse mosquito: https://mosquitto.org/download/
  - Test servers:
    - https://test.mosquitto.org/ (public, not reliable)
    - http://broker.hivemq.com/ (public, not reliable)

- Client (Python): paho-mqtt
  - https://pypi.org/project/paho-mqtt/ (install with pip install paho-mqtt)

MQTT Subscriber

```
import paho.mqtt.subscribe as subscribe

broker = ip-address

def print_msg(client, userdata, message):
    print("%s : %s" % (message.topic, message.payload))

subscribe.callback(print_msg, "MyTopic", hostname=broker)
```

"callback" function definition

"callback" function

Topic

回调函数 = 系统帮你调用的函数，不是你手动调用

订阅 MyTopic 这个主题，当有新消息到来时，用 print_msg 这个函数来处理消息

MQTT Topic

- Topics are case-sensitive

- MQTT accepts multilevel topics
  - CSBldg/level3/temperature
  - Please use a unique topic in your upcoming assignment 2
    - E.g., unikey/ass2

- MQTT accepts wildcard topics
  - Single-level wildcard: +
    - CSBldg/+/temperature
  - Multi-level wildcard: #
    - CSBldg/level3/#: all sensors and subtopics

MQTT Publisher

```
import paho.mqtt.publish as publish

broker = …

publish.single("MyTopic", "TheMessage", hostname=broker)
```

Additional Features: Retain and Quality of Service

MQTT broker 不存历史消息

只有 在线的订阅者 才能收到消息

晚到的订阅者 → 什么也收不到

- By default, MQTT broker does not store any message.
  - When publishing a message you can set the **retain** message flag. This flag tells the broker to store the last message that you sent.

发布消息时设置 retain = true，告诉 broke
把这条"最新消息"存下来

- Quality of Service (Message processing guarantees)
  - *At most once (QoS 0)*: every message is processed at most once, but no guarantee
  - *At least once (QoS 1)*: no message will get ignored
  - *Exactly once (QoS 2)*: system guarantees that every message processed exactly once

尽力而为，发了就不管了，最快

一定送到，但可能送两次

一定送到，而且只送一次，最慢

- By default, MQTT uses QoS 0.

Why Are Message Processing Guarantees So Complex?

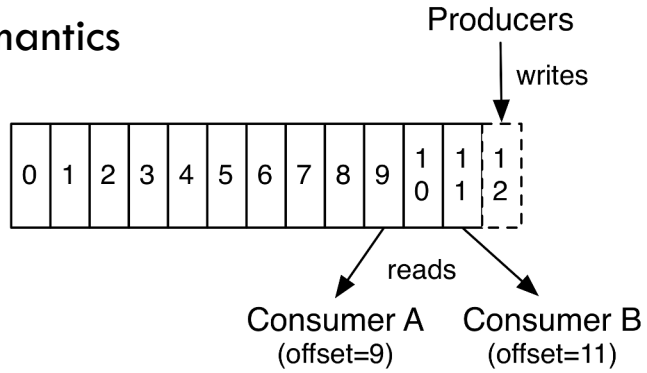
- The complexity arises from system behaviour during failures. Messages can be lost or duplicated due to various failure scenarios, including:
  - Producer publish failure
  - Messaging system failure
  - Consumer processing failure

- Example: consumer crashes during database writes, or brokers running out of disk space.
  - These failures are not hypothetical — they happen in all environments, including production.

Apache Kafka: Distributed Pub/Sub Systems

- For large-scale stream data processing, we need Pub/Sub systems that are scalable. There are many such systems.

- The Apache Kafka system is a *distributed* Pub/Sub system designed to handle large-scale data streaming.
  - Data stream is broken into a sequence of individual data records (aka messages).
    - similar to a structured commit log with append-only semantics



- Major difference between Kafka and MQTT
  - for each topic a stream or queue of data is maintained
  - each consumer is fed items from the topic queue in sequence
  - producers place new data items on the queue
  - can store streams of records in a fault-tolerant way

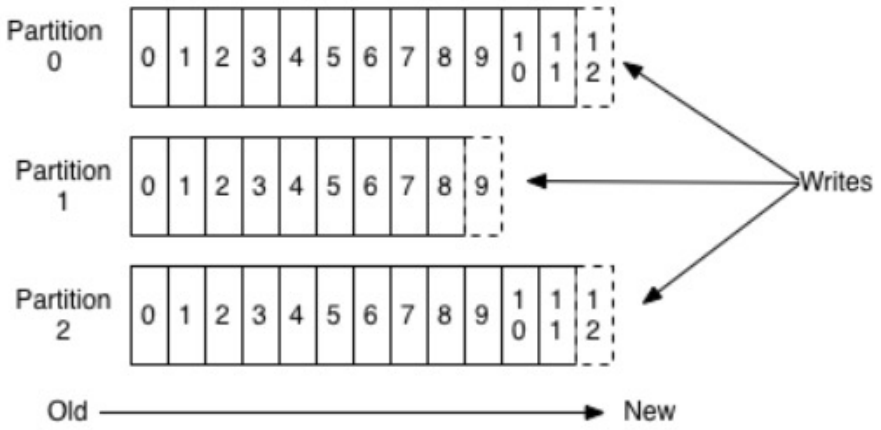
Kafka 是持久化的、顺序的、高可靠的事件日志系统；MQTT 是轻量级的、实时的、解耦的消息传递协议。

Kafka → 日志队列（log queue） + 历史可回

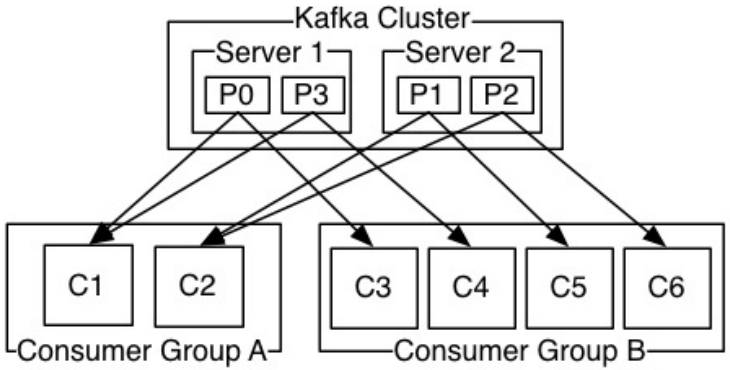
MQTT → 主题订阅（pub/sub） + 实时轻量

Kafka Topics

- Kafka maintains a partitioned log for each topic



- Log is replicated across multiple servers
- Topics in Kafka are multi-subscriber
  - each record published to a topic is delivered to one consumer instance within each subscribing *consumer group*.
  - Consumer instances can be in separate processes or on separate machines.



Guarantees from Kafka

- Messages sent by a producer to a particular topic will be appended in the order sent.
  - Messages are **durably stored** on disk; can be replayed later.
- A consumer sees records in the order they are stored in the queue.
- For a topic with replication factor *N*, it will tolerate up to *N*-1 server failures without losing any records committed to the log.
- Kafka: **at least once** semantics
  - as published messages are first 'committed' to a topic log, it is guaranteed to not get lost
  - but depending on how consumer handles crashes, a message might be read more than once

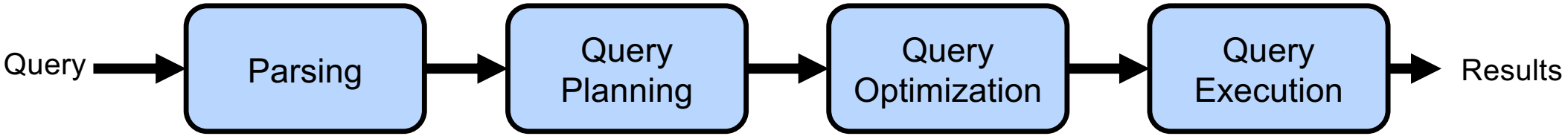
Example Use Case: Real-Time Data Ingestion

- In real-time data ingestion, Kafka acts as a "data pipe" — constantly pulling data from many different sources and pushing it into systems that need it — without waiting for scheduled batch jobs.
- Key components involved:
  - Producers: Systems that generate data (e.g., web servers, mobile apps, IoT devices, sensors, databases).
  - Kafka Brokers: Kafka servers that receive the data from producers and store it in topics.
  - Consumers: Systems that read the data from Kafka (e.g., analytics platforms, machine learning models, alerting systems).
- Advantages of Kafka
  - Guarantees durability, scalability, and low-latency delivery
  - Consumers can replay or reprocess data from any point in time

Stream Query Processing

Querying Stream Data

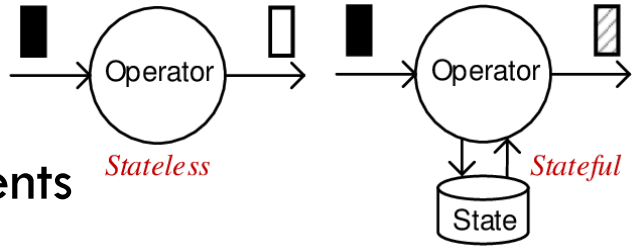
- Query processing assumes fix set of data, both for planning and execution
- Streaming data is however constantly "in motion"
  - Query semantics?
  - Continuous query execution?



Data Stream Query Processing

- *Stateless* operators look at each event individually
  - E.g., selections and (duplicate preserving) projections work locally, per element

```
SELECT sourceIP, time
  FROM IPTraffic
 WHERE length > 512
```



- *Stateful* operators output results based on multiple events
  - Eg. Window operators, aggregation, joins
  - Joins possible via stream-enabled join algorithms
  - Aggregation – need first to define a ***window*** on the data stream to process
- *State* refers to data that is retained and updated across multiple events during stream processing.
  - State is crucial for aggregations, joins, and other operations that rely on data from multiple events.

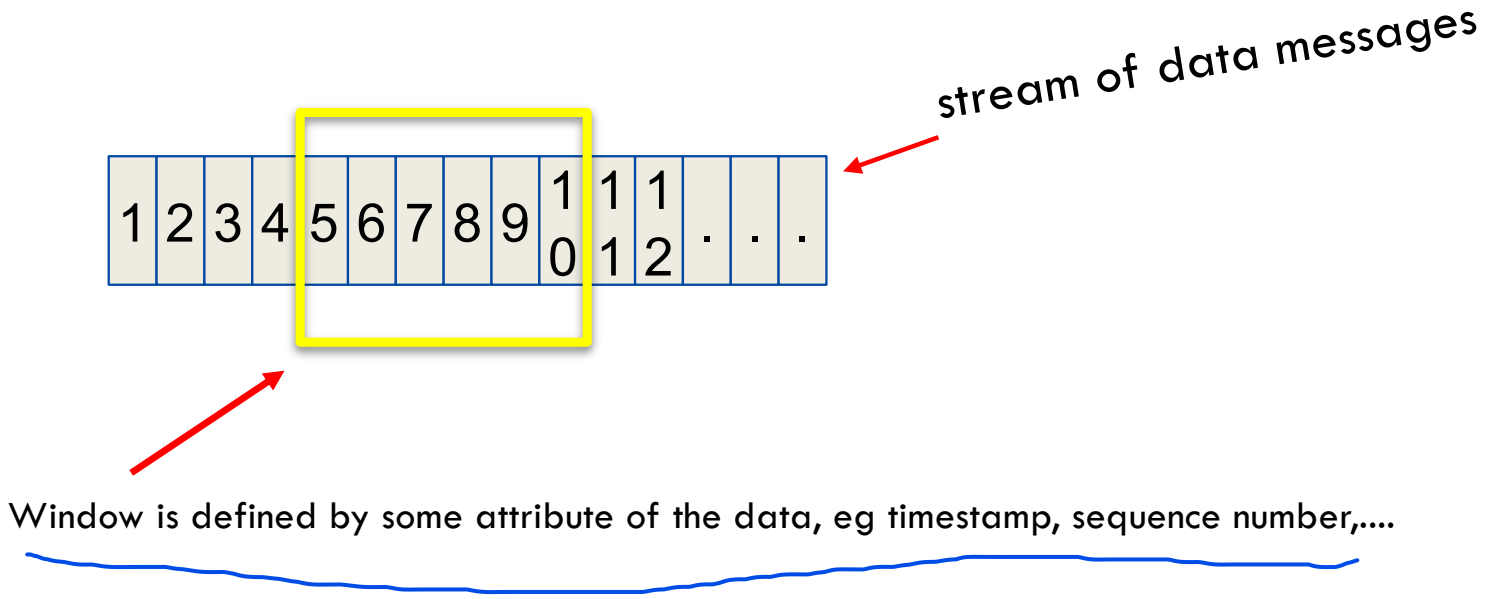
Stateful vs Stateless Applications

The key difference between stateful and stateless applications is whether they retain information about a user's interactions. Specific differences include:

- **Scalability**: Stateless applications are more scalable as requests are independent, while stateful ones need complex load balancing and session management.
- **Fault tolerance**: Stateless apps are more fault-tolerant since losing a server doesn't affect sessions, unlike stateful apps, which need measures like session replication.
- **Resource utilisation**: Stateless apps use fewer resources, as they don't manage session data, while stateful apps require more memory and processing for session handling.
- **Development complexity**: Stateless apps are simpler to develop, as there's no state to manage across requests, whereas stateful apps require careful session management.

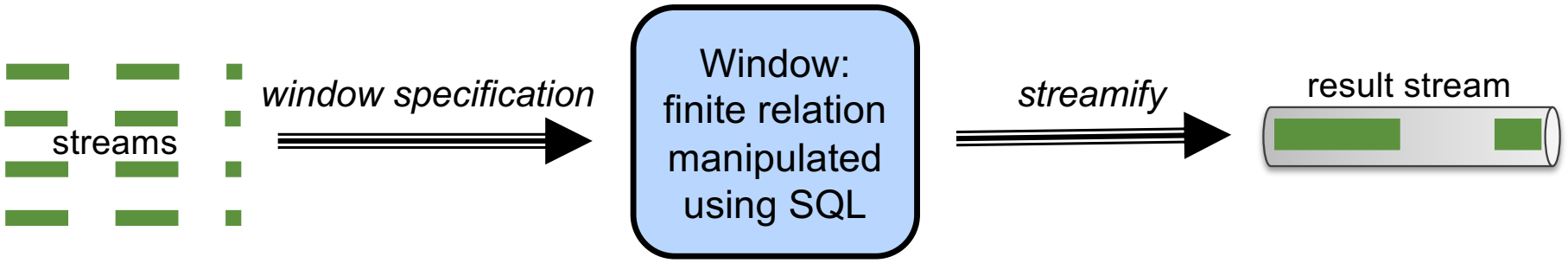
Stream Query Processing: Windowing Technique

- **Window**: mechanism for extracting a finite relation from an infinite stream
  - Enables real-time processing by breaking data into smaller, more actionable segments.



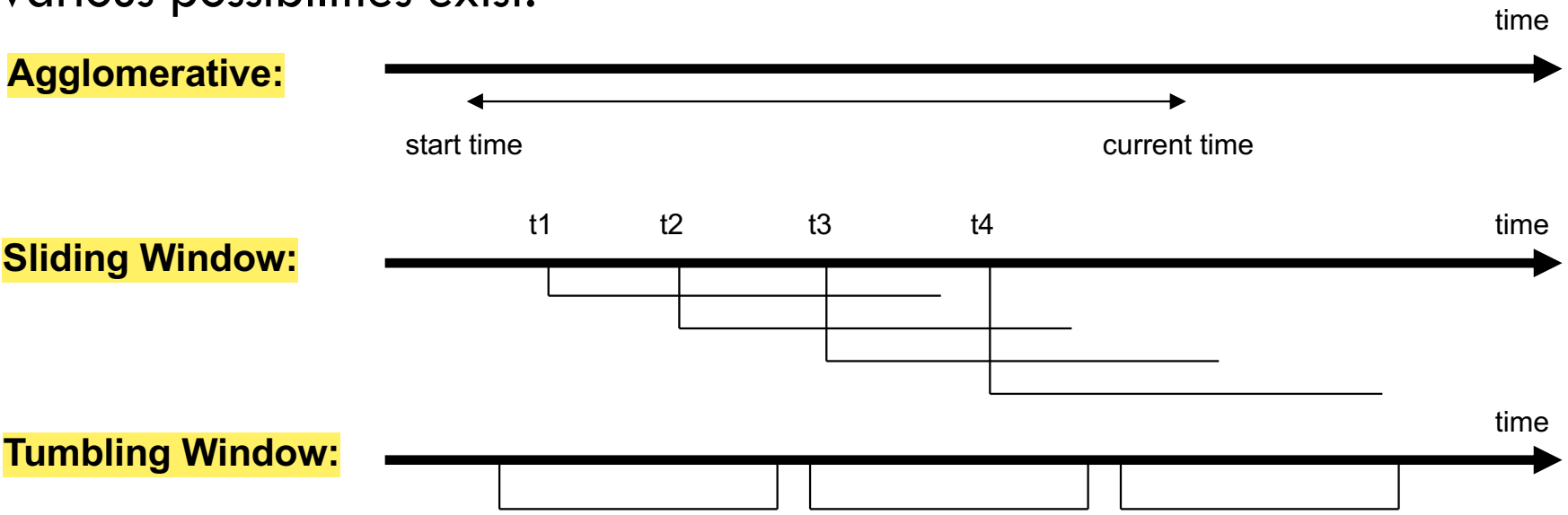
Stream Query Processing: Windowing Technique

- **Window**: mechanism for extracting a finite relation from an infinite stream
- Various window proposals for restricting operator scope
  - Windows based on *ordering attributes* (e.g., time)
  - Windows based on *tuple counts*
  - Windows based on *explicit markers* (e.g., punctuations)
  - Variants (e.g., partitioning tuples in a window)



Ordering-Attribute-Based Windows

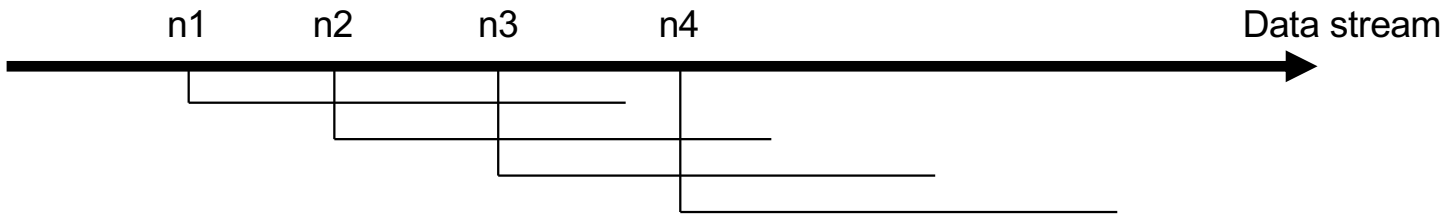
- Assumes existence of an attribute that defines order of stream elements/tuples (e.g. time)
- Let *T* be the window length (size) expressed in units of the ordering attribute (e.g., *T* may be a time window)
- Various possibilities exist:



这个是特殊的sliding window

Tuple-Count-Based Windows

- Window of size *N* tuples (sliding, tumbling) over the stream
- Problematic with non-unique time stamps associated with tuples
- Ties broken arbitrarily may lead to a non-deterministic output



Sessions Windows

- Punctuation-Based Windows:
  - Application inserts ‘end-of-processing’ markers
  - Each data item identifies ‘beginning-of-processing’
  - Enables data item-dependent variable length windows
    - E.g. a stream of auctions
  - Allows to group events based on sessions

Quiz

Sliding: 可以belong to more than 1 window
但是tumbling不行

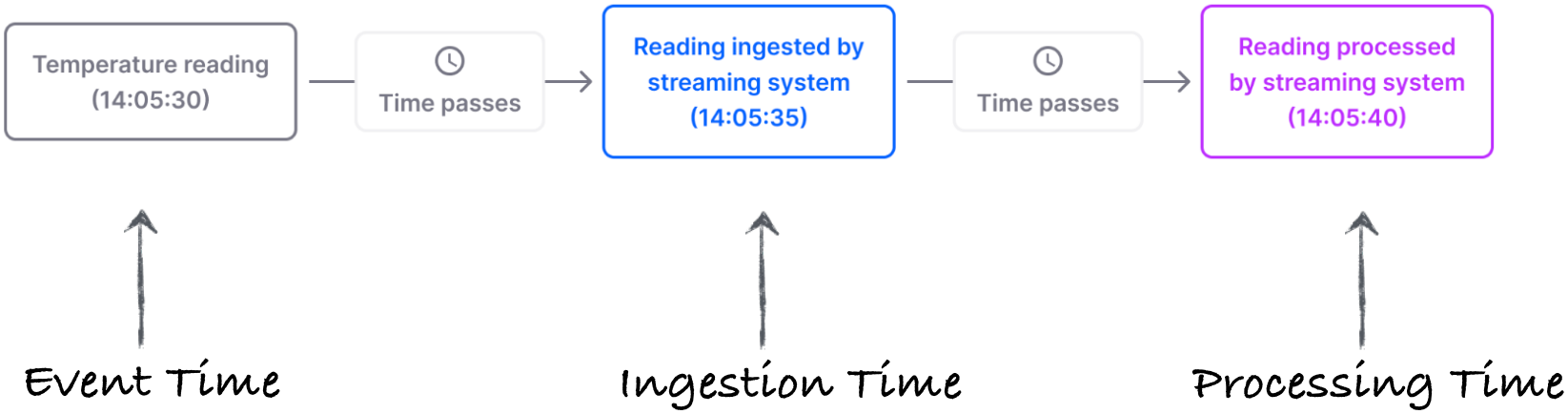
- In which windowing strategies, such as sliding or tumbling windows, can a tuple belong to multiple windows?

- Is punctuation-based windows a type of sliding windows, tumbling windows, or neither?

neither；窗口的边界不是由时间或事件数量决定，而是由流中出现的特殊"标记（punctuation）"决定

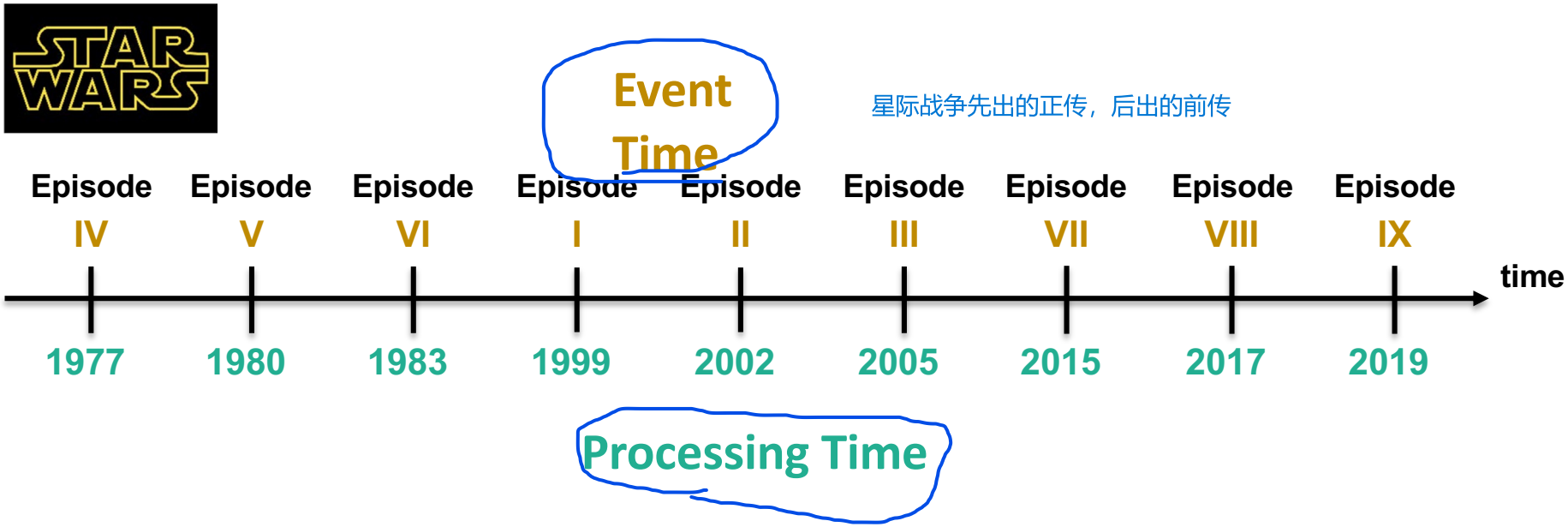
Notions of Time in Stream Processing

- Stream Processing Systems distinguish between **Event time, Ingestion time** and **Processing time**



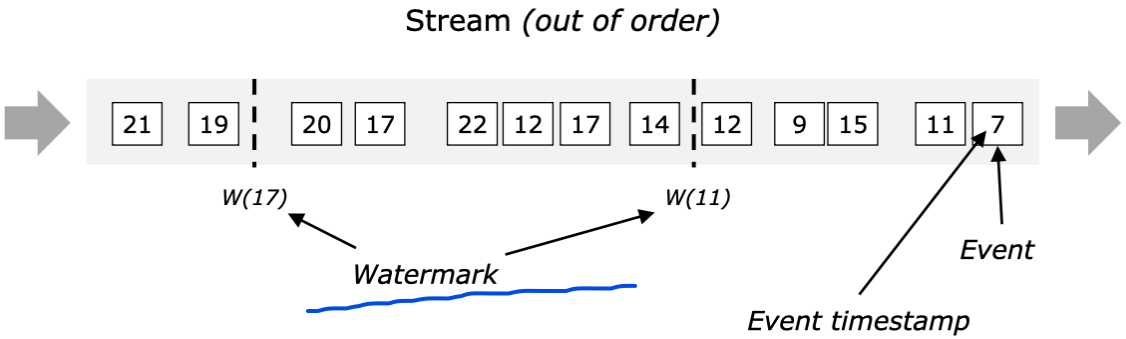
Notions of Time in Stream Processing

- Stream Processing Systems distinguish between **Event time, Ingestion time** and **Processing time**
  - important to be able to define windows by either
  - out-of-order processing possible



Event Time and Watermarks

- A stream processor supporting *event time* needs a way to measure the progress of event time
- *Event time* can progress independently of *processing time* (measured by wall clocks)
  - Key challenges: *Out-of-Order Events* and *Late Data*
- Important to have a mechanism to measure progress in event time: **Watermarks**
  - In general, a watermark is a declaration that by that point in the stream, all events up to a certain timestamp should have arrived.
  - Once a watermark reaches an operator, the operator can advance its internal *event time clock* to the value of the watermark.



Example: Watermark Handling in Apache Flink

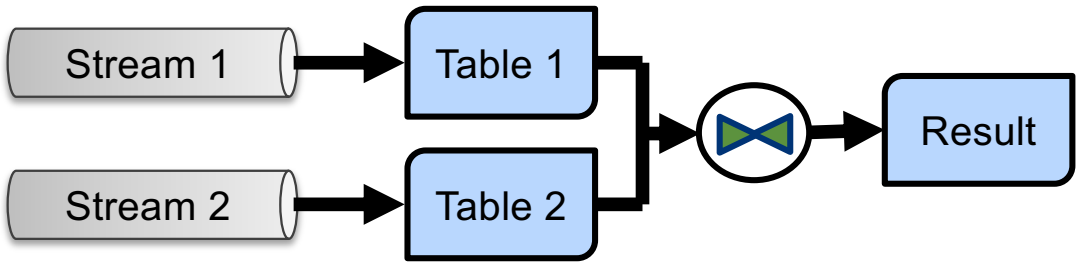
1. Events arrive in a stream into Apache Flink containing an event timestamp.
2. A WatermarkStrategy for the source allows to generate watermarks
   - periodically after some maximum allowed lateness, or
   - when seeing certain punctuation markers in the event data.
3. Events that arrive after a watermark can be discarded or handled using a late strategy.

Example:
```
env.assignTimestampsAndWatermarks(
        WatermarkStrategy.for_bounded_out_of_orderness(Duration.ofSeconds(5)));
result = stream.window(TumblingEventTimeWindows.of(Duration.ofSeconds(10)))
                                                .allowedLateness(Duration.ofSeconds(5))
                                                .sideOutputLateData(lateOutputStream);
```

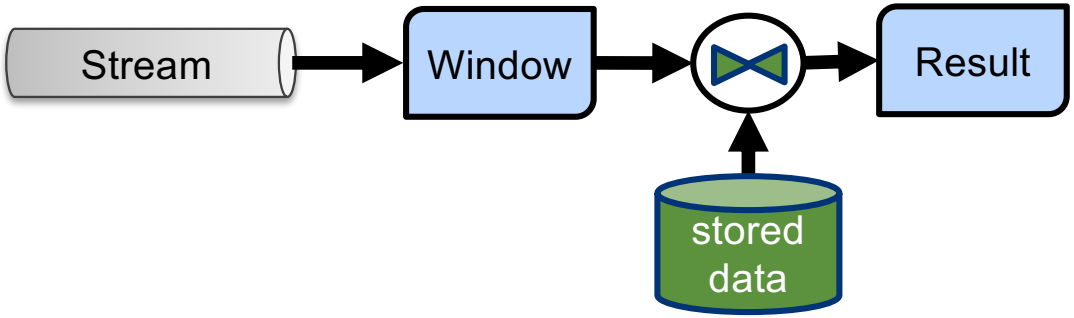
   - system waits for events, allowing a bounded amount of lateness (here: 5 secs).
   - Watermarks are generated periodically, representing the (maximum observed) event time minus the 5-second tolerance window.
   - If an event arrives more than 5 seconds late, it is considered “late” and put aside.

Stream Joins: Combining Streams with other Data

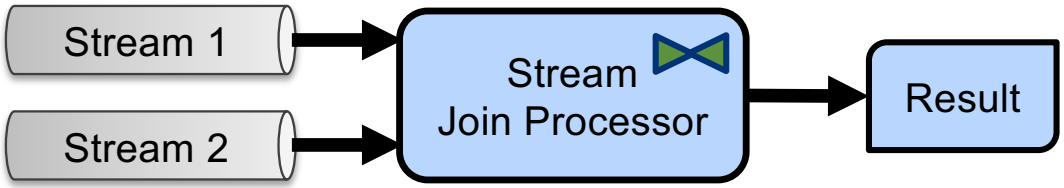
1. Conventional table joins



2. Enrichment



3. Stream-to-stream Joining



Stream-Table Join (Enrichment)

Enrich stream data with static reference data for comprehensive analysis.

- Challenges: Balancing access to up-to-date reference data without impacting performance.
- Solution: Use **in-memory storage** or external data stores (e.g., Redis) for fast lookups.
- Example: Joining a real-time transaction stream with product data stored in Redis.
- Best Practices: Use in-memory storage for frequent lookups, external stores for larger data, and avoid direct reads from production databases.

Stream-Stream Join (Windowed)

- In the table on the right, each row represents a new incoming record. The following assumptions apply:
  - All records have the same value for the join key.
  - All records are processed in timestamp order.
  - The join window is 15 seconds

Timestamp	Left Stream	Right Stream	INNER JOIN
1	null		
2		null	
3	A		
4		a	[A, a]
5	B		[B, a]
6		b	[A, b], [B, b]
7	null		
8		null	
9	C		[C, a], [C, b]
10		c	[A, c], [B, c], [C, c]

Summary

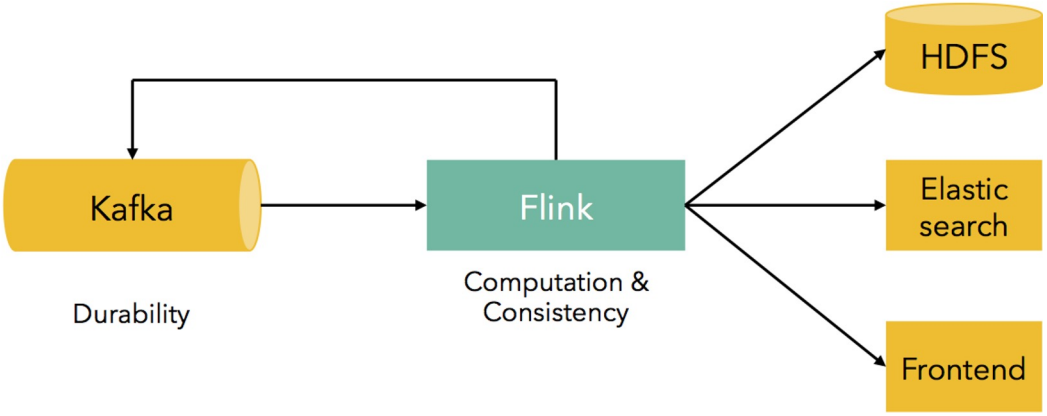
- Data Stream Querying
  - Continuous querying
  - Challenge: unbound nature of data streams with potential out-of-order processing
- Windowing: mechanism for extracting a finite relation from an infinite stream
  - defining context for joins and aggregation
- Must cope with:
  - Stream rates that may be high, variable, and bursty
  - Stream data that may be unpredictable and variable

Tools for Stream Processing

Data Stream Processing

- Recall: Kafka or MQTT are not data stream processors
  - 'just' messaging (Pub/Sub) and storage system for data streams
- Common data stream processors with Kafka are Apache Spark and Flink

Example with Flink:



[https://data-artisans.com/blog/kafka-flink-a-practical-how-to]

Example with Spark:

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

Data Stream Processing with Apache Spark Streaming

- Spark streaming is an extension of the core RDD API
  - RDD is a "Resilient Distributed Dataset"
- Input data stream is divided into *micro batches* which Spark Engine processes
  - Supports fixed, sliding and tumbling windows
  - High-level abstraction of *discretized stream* or *DStream*, which are internally represented as a **sequence** of RDDs

Example: Data Stream WordCount with Spark

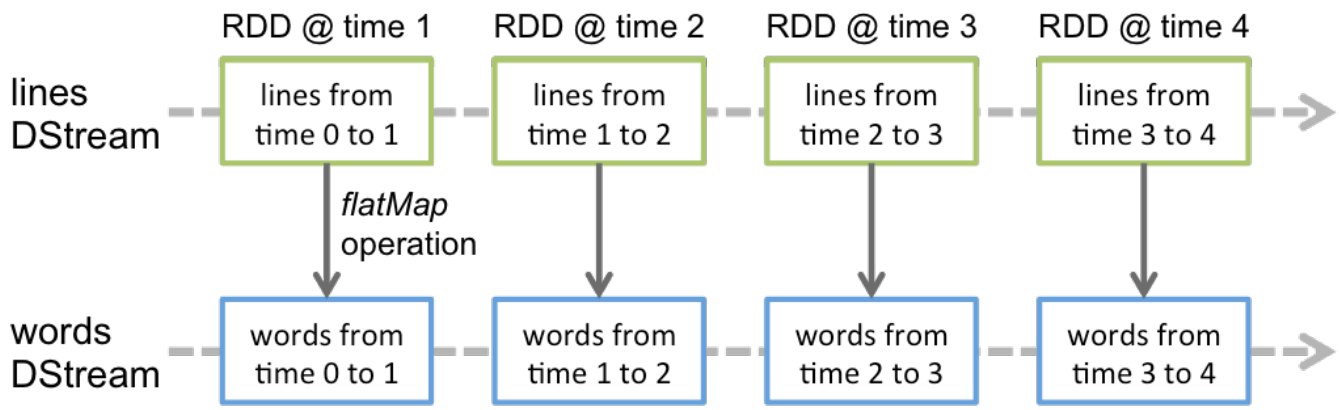
```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)

# Create a DStream that will connect to hostname:port, like localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)
# Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.pprint()

ssc.start() # Start the computation
ssc.awaitTermination() # Wait for the computation to terminate
```

set up computation

execute computation

Source: https://spark.apache.org/docs/latest/streaming-programming-guide.html

Example: Data Stream WordCount with Spark (output)



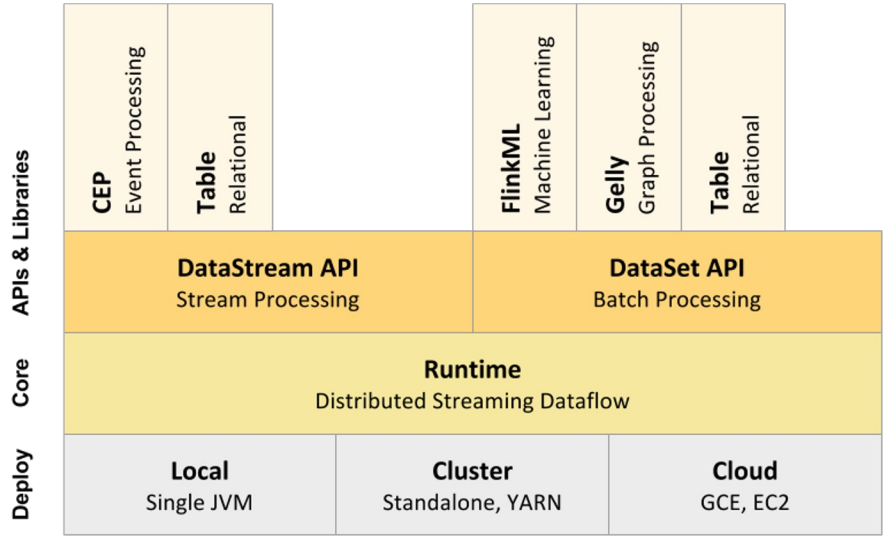

```
# TERMINAL 1:
# Running Netcat

$ nc -lk 9999

hello world



...
```

```
# TERMINAL 2: RUNNING network_wordcount.py

$ ./bin/spark-submit examples/src/main/python/streaming/network_wordcount.py localhost 9999
...
-------------------------------------------
Time: 2014-10-14 15:25:21
-------------------------------------------
(hello,1)
(world,1)
...
```

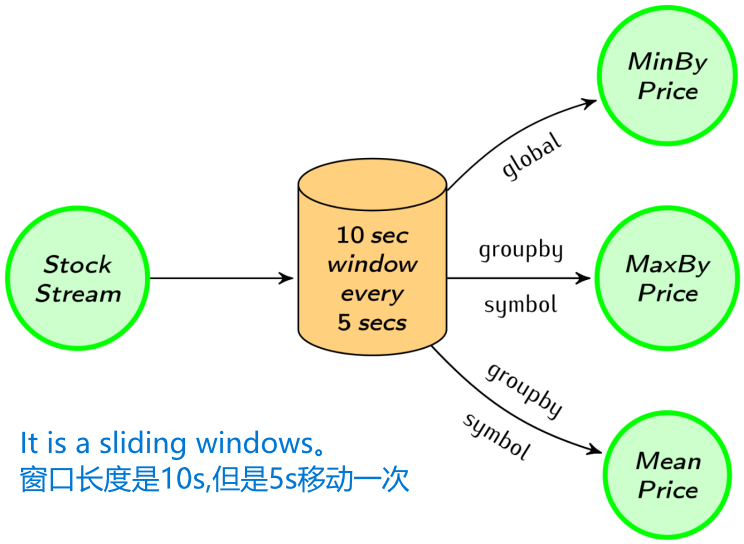

Source: https://spark.apache.org/docs/latest/streaming-programming-guide.html

Data Stream Processing with Apache Flink

- Apache Flink lends itself very well to continuous stream processing because it uses pipelining throughout the whole system
  - tuples are immediately forwarded and consumed by next operator if available
  - major difference to Apache Spark, where processing stages are clearly separated



Example: Window Aggregation with Flink



It is a sliding windows。
窗口长度是10s,但是5s移动一次

```java
//Define the desired time window
WindowedDataStream<StockPrice> windowedStream = stockStream
    .window(Time.of(10, TimeUnit.SECONDS))
    .every(Time.of(5, TimeUnit.SECONDS));

//Compute some simple statistics on a rolling window
DataStream<StockPrice> lowest = windowedStream.minBy("price").flatten();
DataStream<StockPrice> maxByStock = windowedStream.groupBy("symbol")
    .maxBy("price").flatten();
DataStream<StockPrice> rollingMean = windowedStream.groupBy("symbol")
    .mapWindow(new WindowMean()).flatten();

//Compute the mean of a window
public final static class WindowMean implements
    WindowMapFunction<StockPrice, StockPrice> {

    private Double sum = 0.0;
    private Integer count = 0;
    private String symbol = "";

    @Override
    public void mapWindow(Iterable<StockPrice> values, Collector<StockPrice> out)
        throws Exception {

        if (values.iterator().hasNext()) {s
            for (StockPrice sp : values) {
                sum += sp.price;
                symbol = sp.symbol;
                count++;
            }
            out.collect(new StockPrice(symbol, sum / count));
        }
    }
}
```

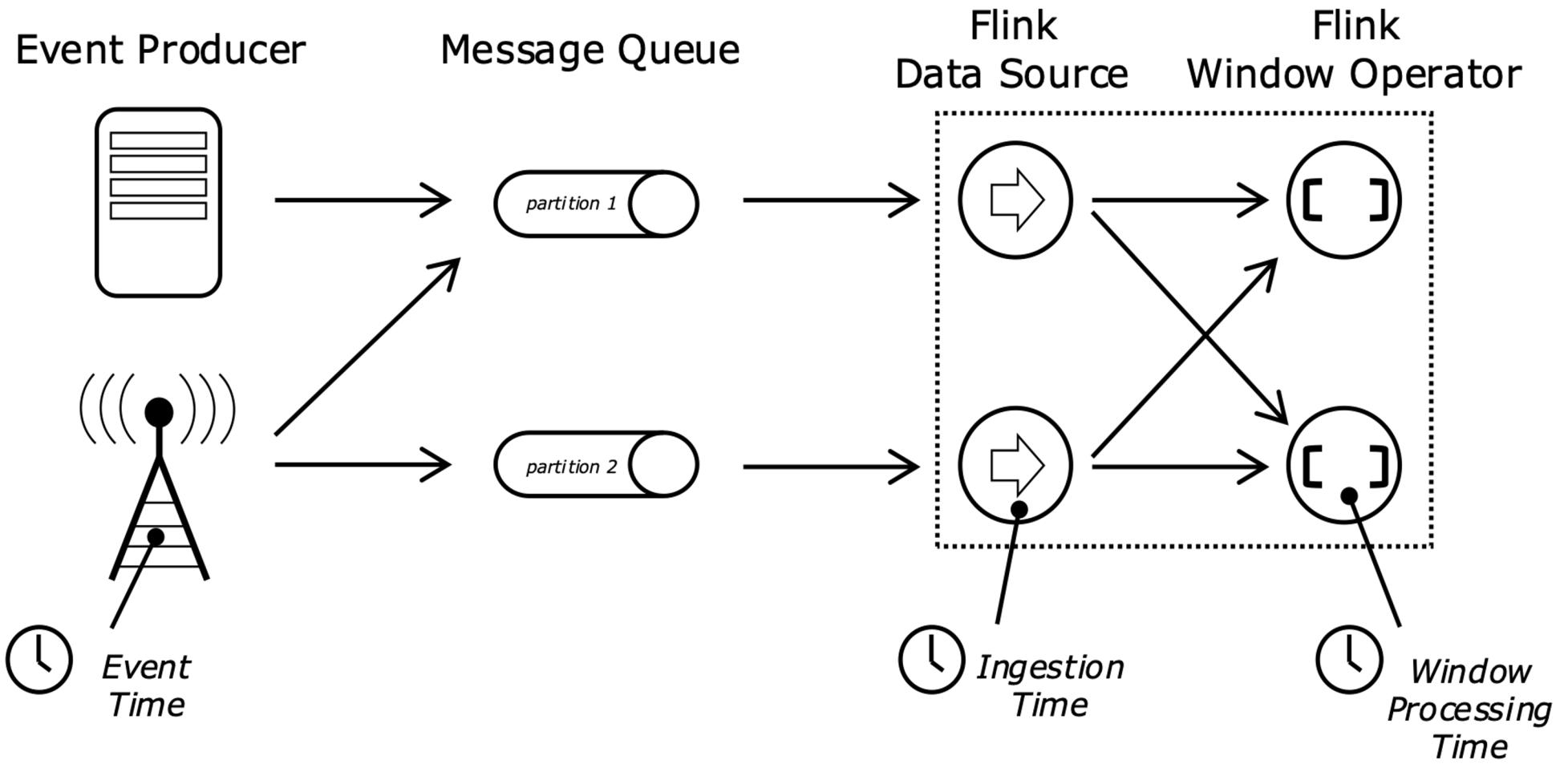
[https://flink.apache.org/news/2015/02/09/streaming-example.html]

支持 三种时间语义Processing Tir

Event Time

Ingestion Time

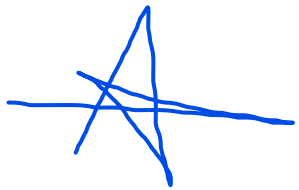
Apache Flink Supports Three Notions of Time



Flink Example: Time Definition

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);

// alternatively:
// env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);
// env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

DataStream<MyEvent> stream = env.addSource(new FlinkKafkaConsumer09<MyEvent>(topic, schema, props));

stream
    .keyBy( (event) -> event.getUser() )
    .timeWindow(Time.hours(1))
    .reduce( (a, b) -> a.add(b) )
    .addSink(...);
```

Differences between Spark and Flink



	Apache Spark	Apache Flink
Principle	set-oriented data transformations in stages	transformations by iterating over collections with pipelining
Data Abstraction	RDD	DataSet
Processing Stages	separate stages	overlapping stages
Optimiser	with SparkSQL	integrated into API
Batch Processing	RDD	DataSet
Stream Processing	micro-batching	pipelining; DataStream API

Summary

- **Data Stream Processing Principles**
  - window **aggregation,** different notions of time
- **Stream Processing Architectures**
  - stream-oriented Publish/Subscribe messaging systems
    - such as **Apache Kafka** or **MQTT**
  - Stream processors
    - **Apache Spark:** Spark Streaming for stream processing, mini-batches on-top of Spark RDDs
    - **Apache Flink:** Optimised for stream data processing by emphasizing pipelined processing which allows for low latency processing
- More details on Spark and Flink next week.