Aalto University
School of Science

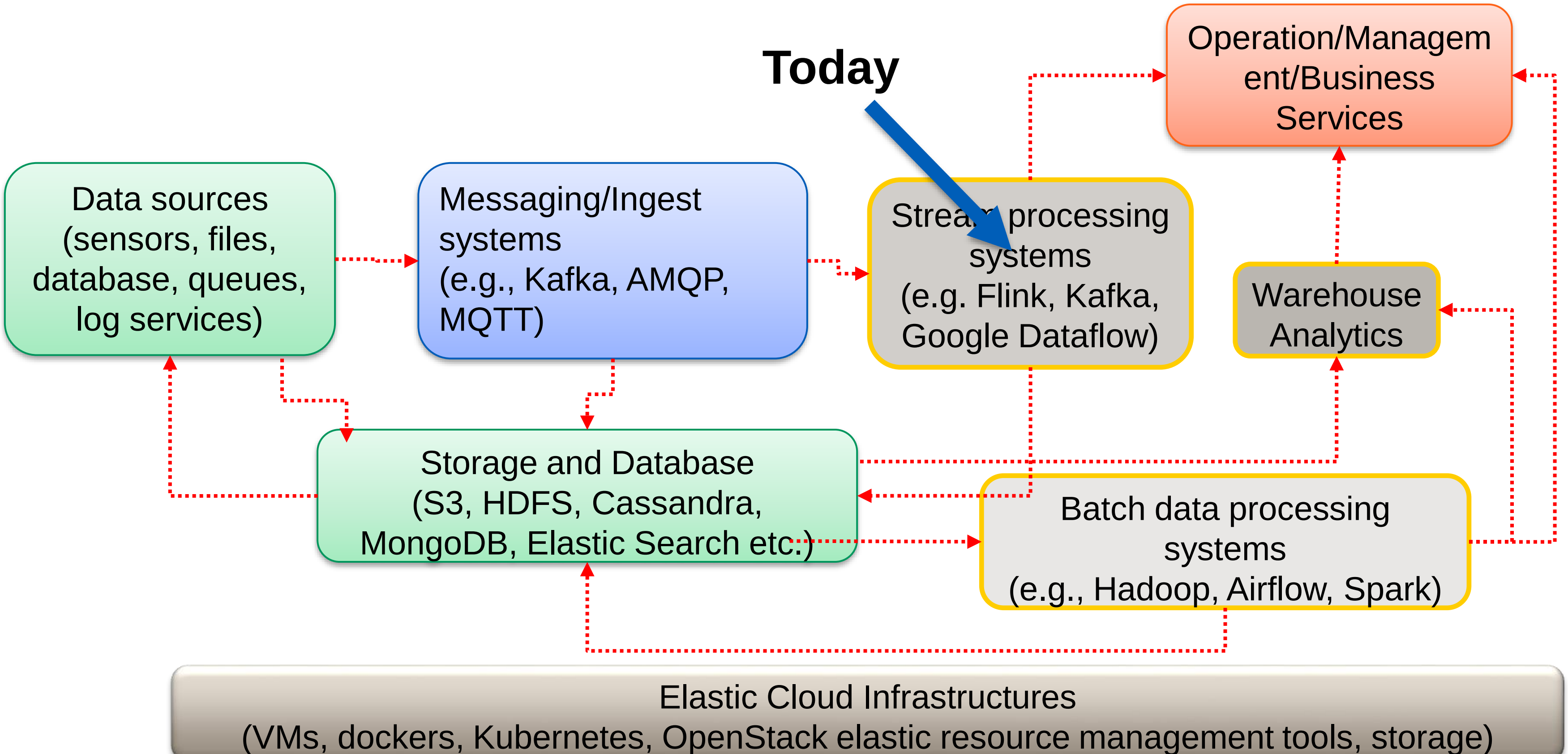
Stream Processing and Big Data Platforms

Hong-Linh Truong
Department of Computer Science
linh.truong@aalto.fi, *https://rdsea.github.io*

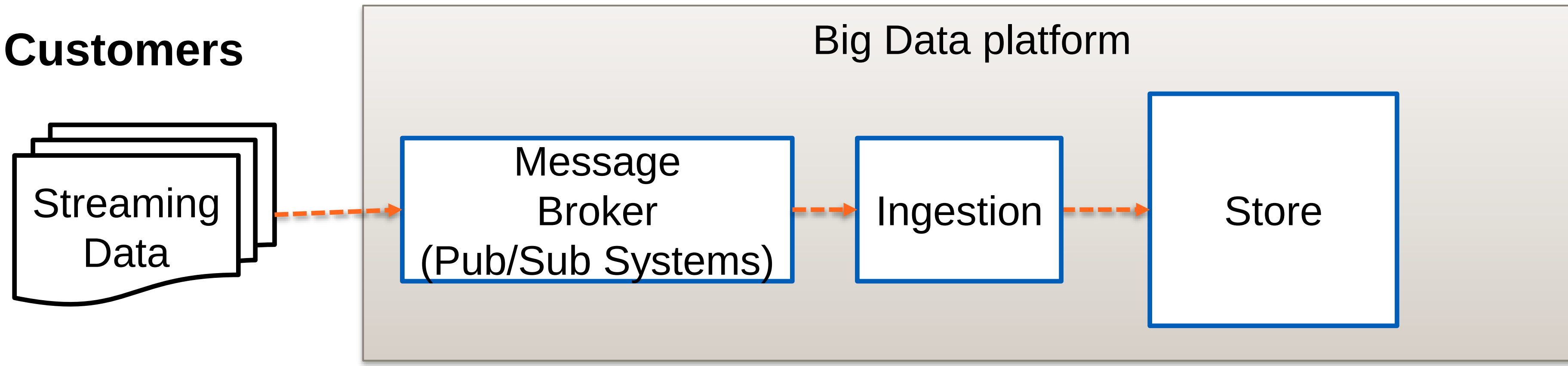
Schedule

- **Do you see the role of stream processing in big data platforms?**
- **What should we learn if you want to support stream processing in your platforms?**
  - Basic concepts and important aspects
  - Apache Flink case
- **Summary**

Big data at large-scale

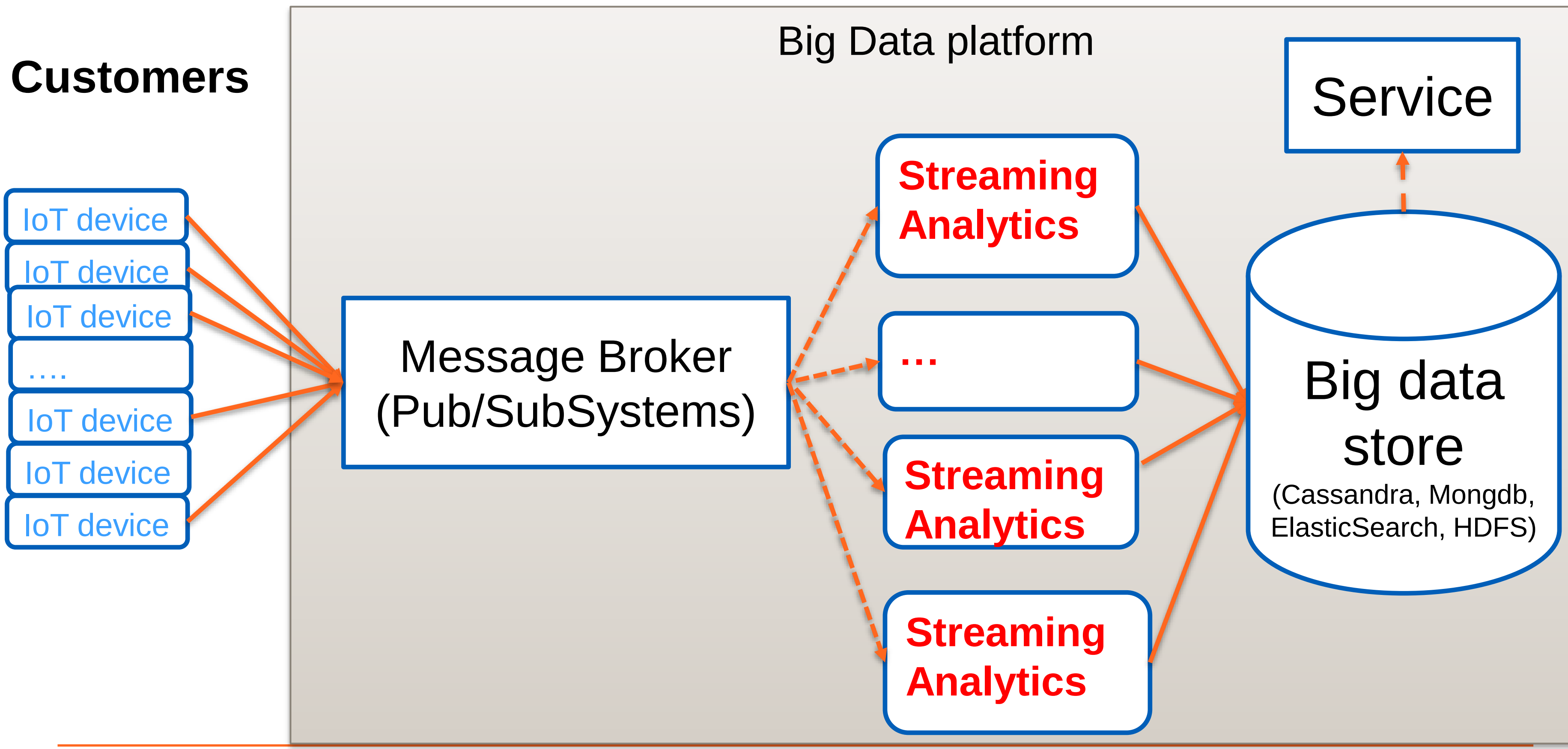


Recall: near-real time streaming data ingestion

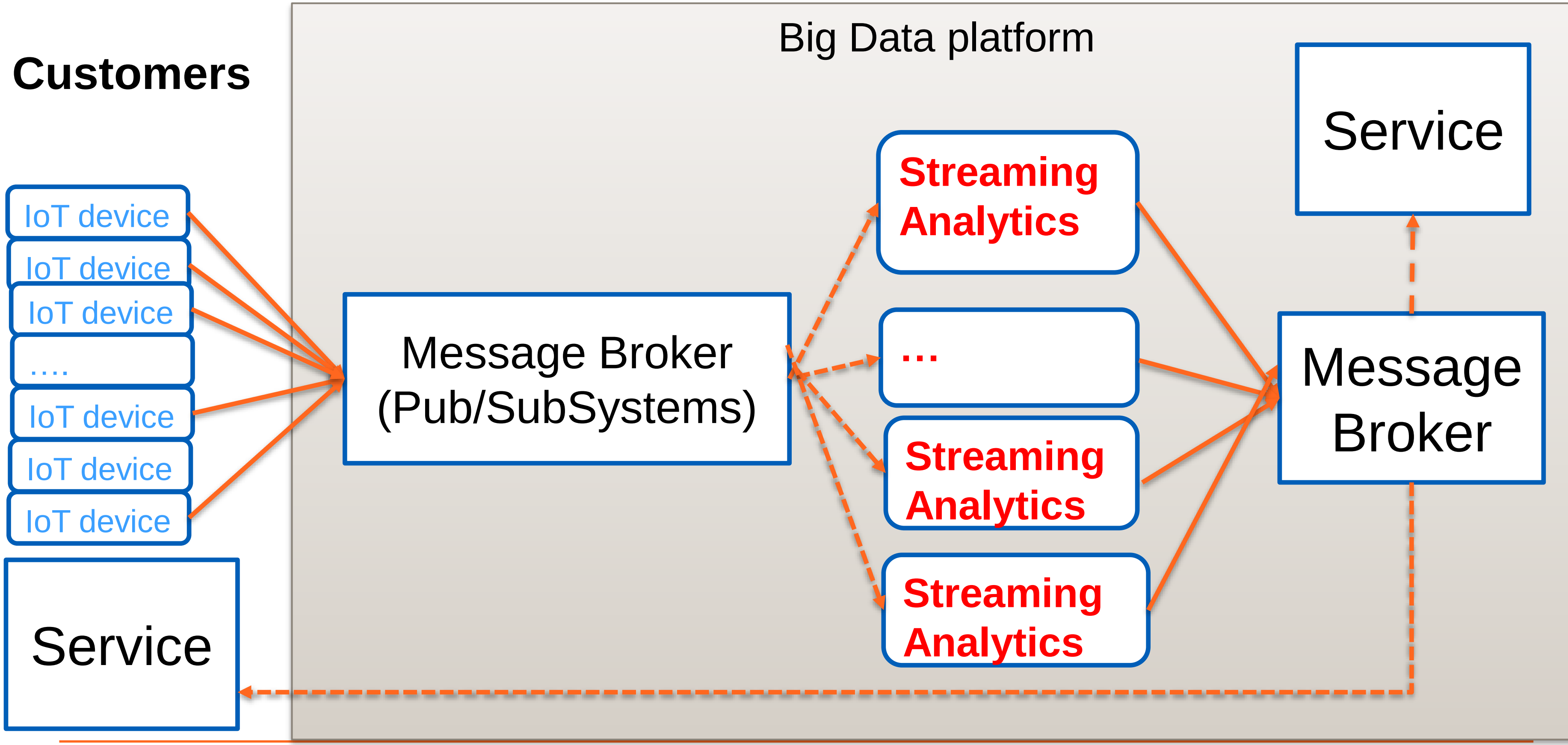


- **Mostly we ingest raw data without/little processing**
- **Data is unbounded from different places in different orders!**

Near realtime streaming data processing



Near realtime streaming data processing



Stream processing and big data platforms

- **Stream processing is part of big data platforms!**
  - A big data technology
  - Pre-processing, ingestion and high-level analytics
- **Stream processing services as big data platforms**
  - We can build a big data platform mainly based on stream processing services
  - Analytics on the fly as the first class
    - *Historical analytics results as the second class*
  - E.g., IoT analytics, e-commerce user activities, fraud detection

Example in the cloud – stream processing and big data platforms

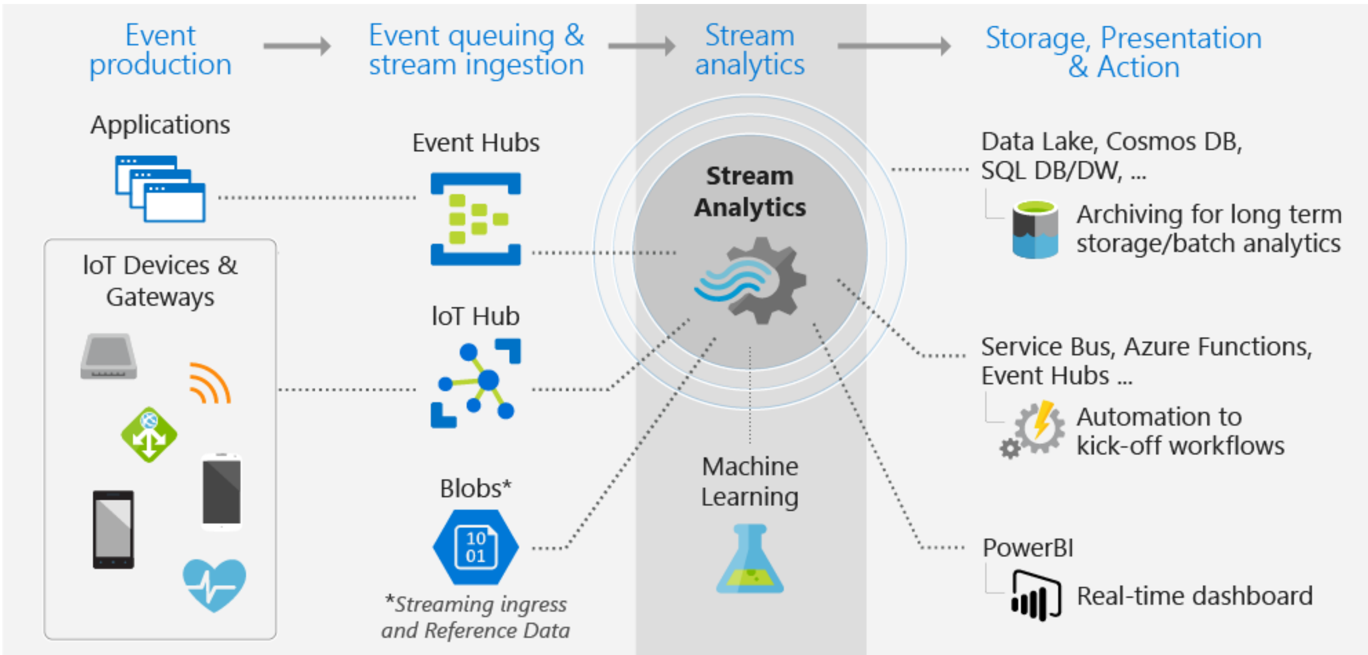


Figure source: https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-introduction

Example of frameworks for study

- **Apache Flink**
- **Apache Kafka**
- **Apache Spark**
- **Apache Storm**
- **Others:**
  - Azure Stream Analytics
  - MQTT + Node-RED

They are not toy examples! They are used in business systems and big cloud platforms

Streaming Data Processing - Concepts

Data stream programming

Data stream: a sequence/flow of data units

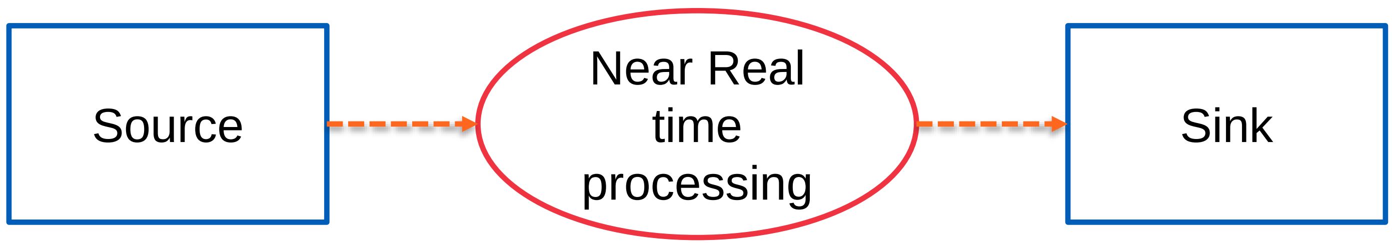
Data units are defined by applications: a data unit can be data described by a primitive data type or by a complex data type, a serializable object, etc.

Streaming data: produced by (near)realtime data sources as well as (big) static data sources → unbounded and bounded

- Examples of data streams
  - Continuous media (e.g., video for video analytics)
  - Discrete media (e.g., stock market events, twitter events, system monitoring events, comments, notifications)

Events/Records

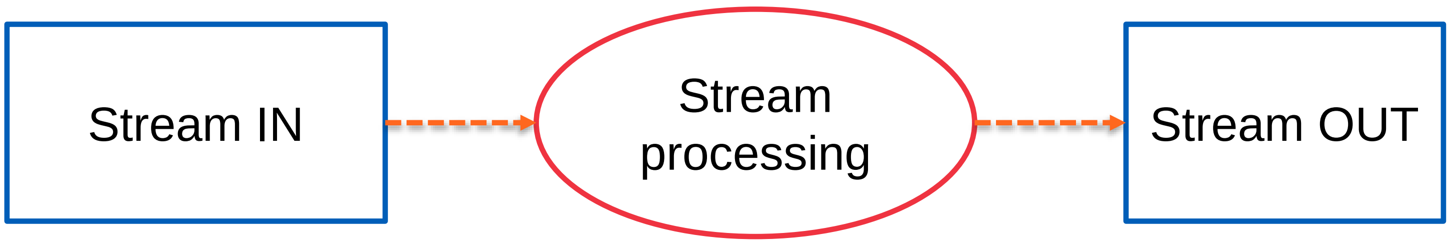
In many applications: data is generated continuously and needs to be processed in near real-time



We focus on unbounded discrete events/records/messages

Stream processing

High level view

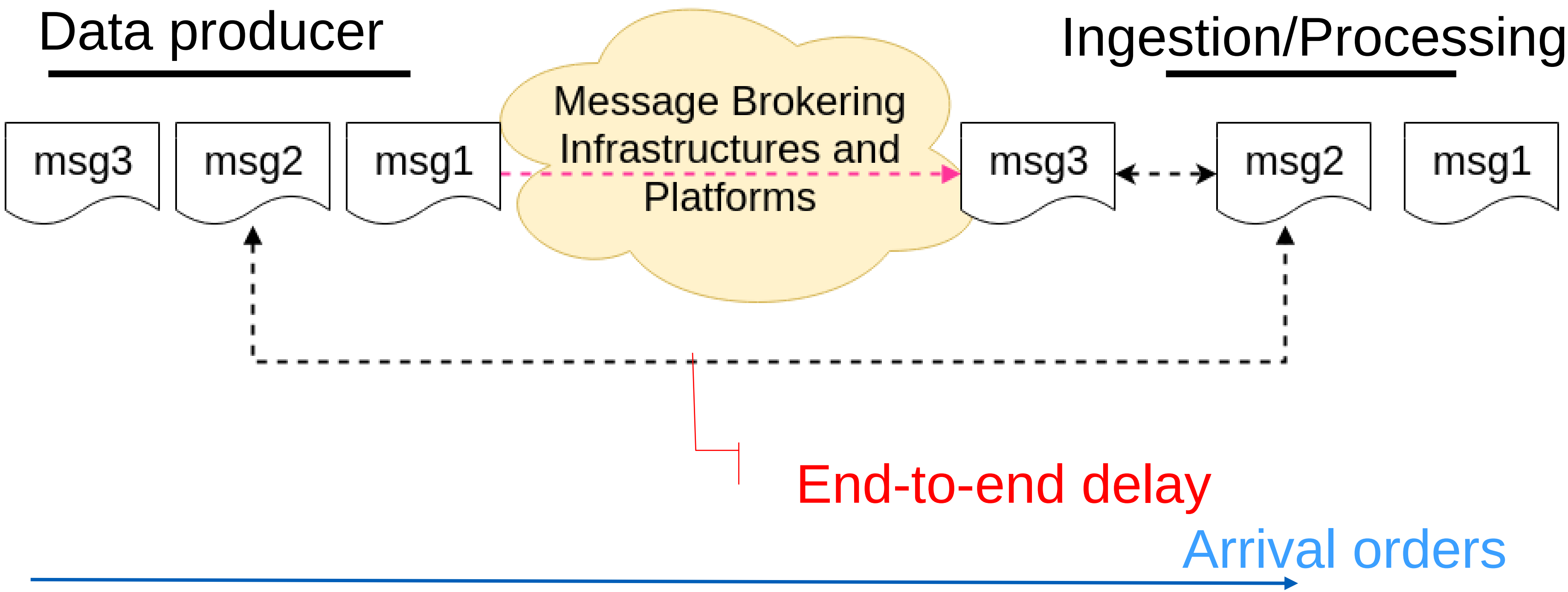


Multiple streams, a set of events

Some key issues

- **Late data, out of order data?**
- **Exactly once?**
- **Times associated with events**
- **Key-based data processing**

Key issues in streaming data: delay and out of order

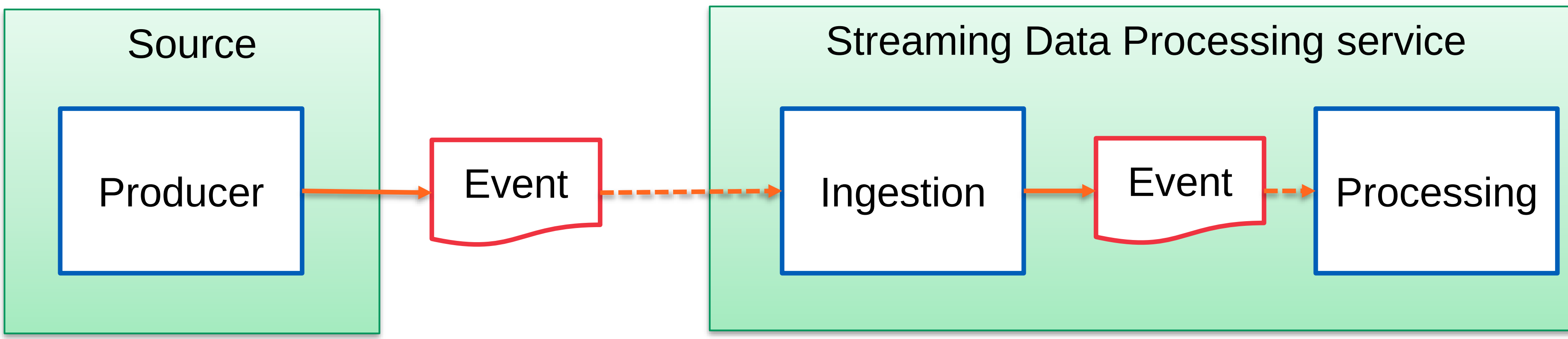


Without event time, do we know the delay or out of order?

What is the consequence of delay/out of order for processing?

Key issues in streaming data: the notion of times

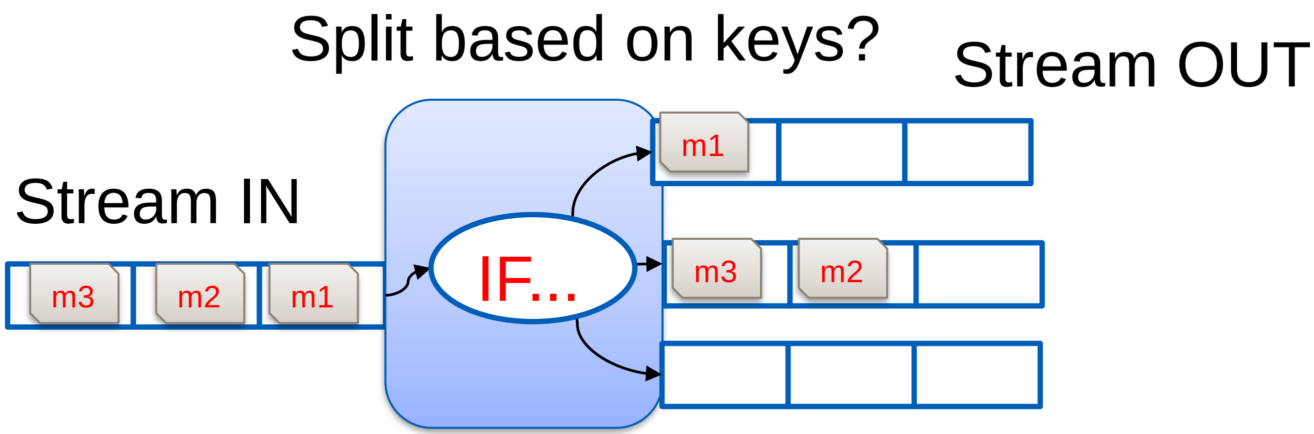
Times associated with data and processing



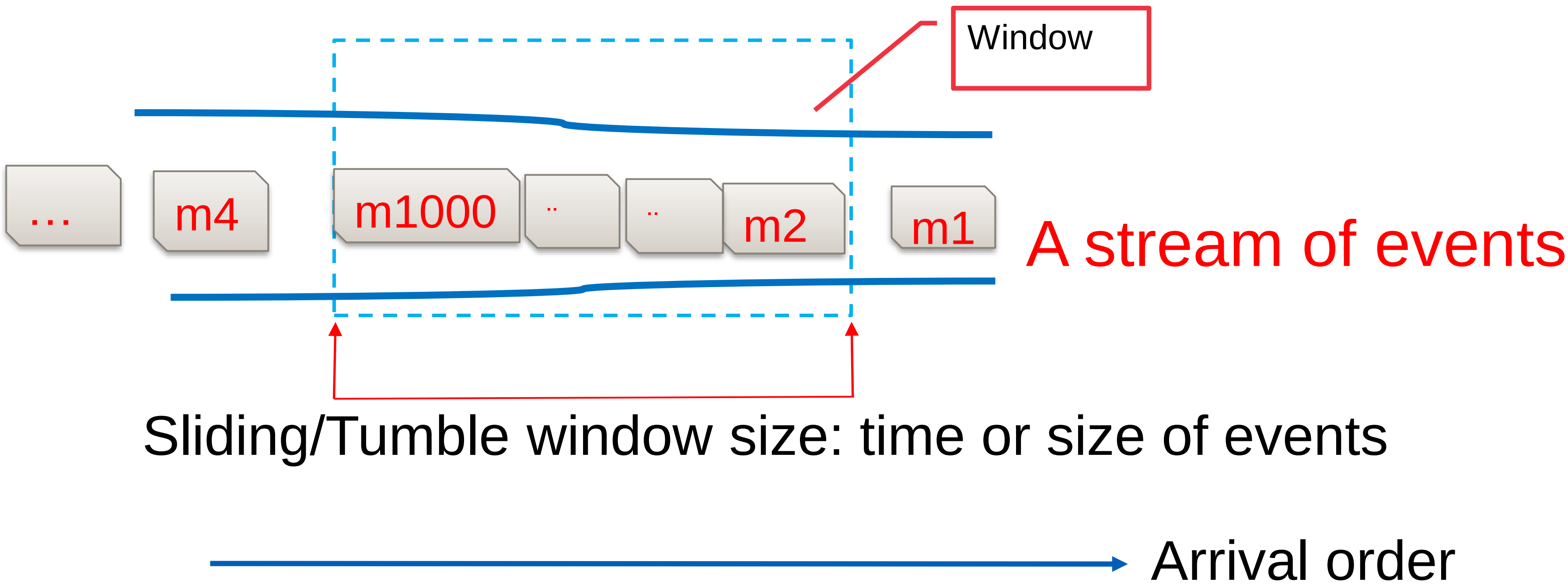
Event time **Ingestion time** **Processing time**

Which time is important for analytics (from business viewpoint)?

Partition stream data based on some keys for analytics



Windows of data



Windowing

- **Windows size: time or number of records (not popular)**
- **Tumbling window:**
  - identified by size, no gap between windows
- **Sliding window:**
  - identified by size and a sliding internal
- **Session Window:**
  - identified by "gap" between windows

Functions applied to Windows of data

If we

- specify a set of conditions for the window and events within the window

then we can

- Apply functions to events in the window that match these conditions

Example

Monitoring working hours of taxi drivers:

- **Windows: 12 hours**
- **Partitioning data/Keyed streams: licenseID**
- **Function: determine working and break times and check with the law/regulation**

Source: https://www.infoworld.com/article/3293426/how-to-build-stateful-streaming-applications-with-apache-flink.html

What if events come late into the windows?

Do we need to deal with late, out of order events?

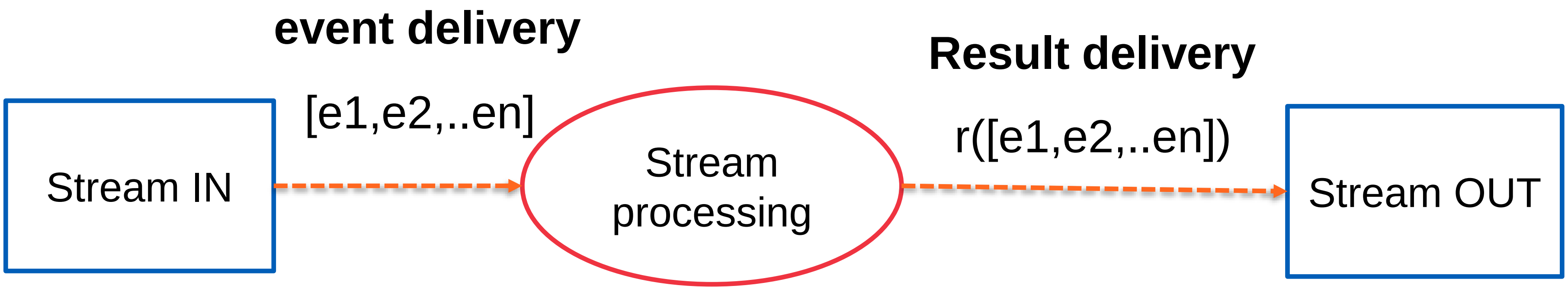
Support lateness

- **Identify timestamp of events**
- **Identify watermark in streams**
  - A watermark is a timestamp
  - A watermark indicates that no events which are older that the watermark should be processed
  - Enable the delay of processing functions to wait for late events

Delivery guarantees

Exactly once? at least once? or at-most-once

End-to-end?

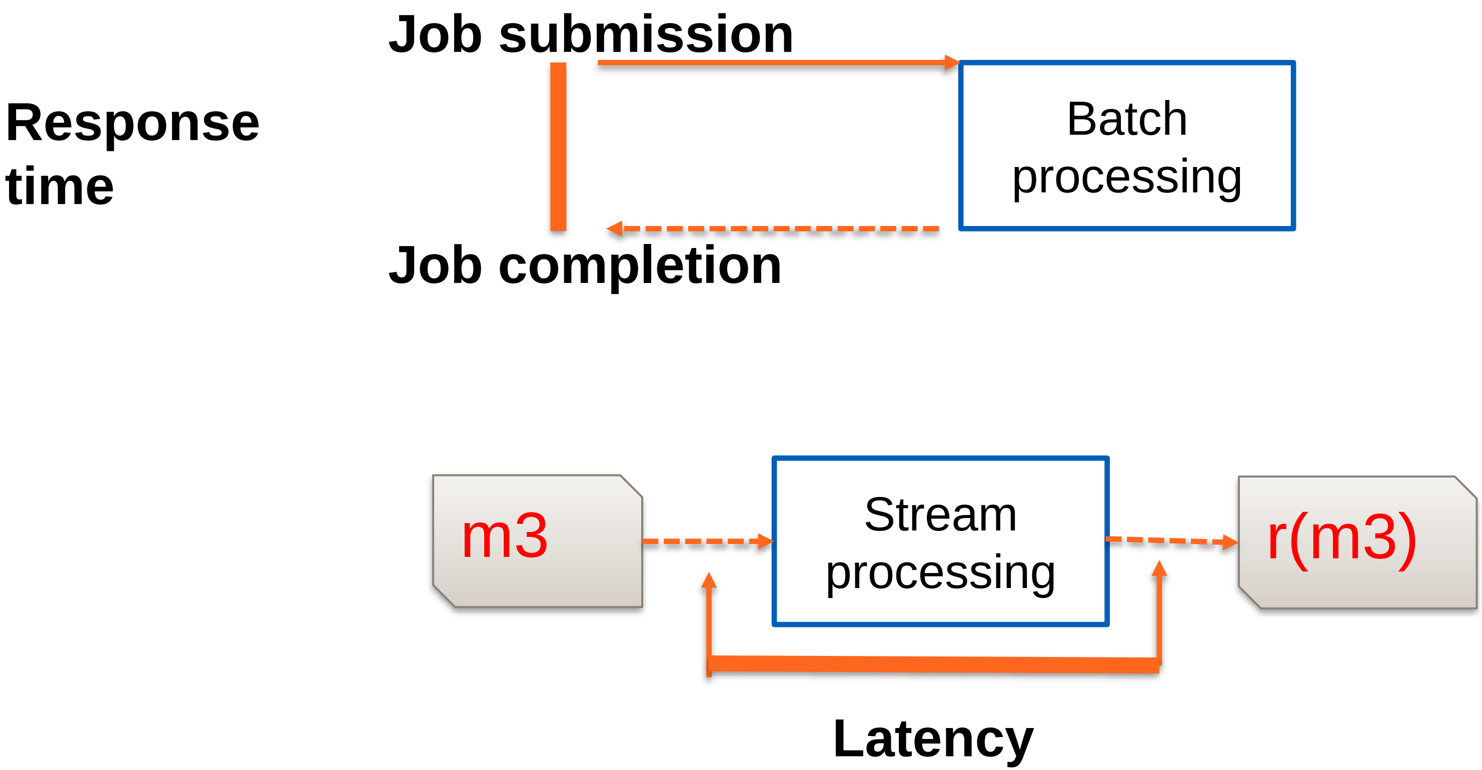


What if the stream processing fails and restarts

Message and processing guarantees

- **Message guarantees are the job of the broker**
- **Processing guarantees are the job of the stream processing frameworks**
- **They are highly connected if brokers and processing frameworks are tightly coupled (e.g. Kafka case)**

Performance metrics



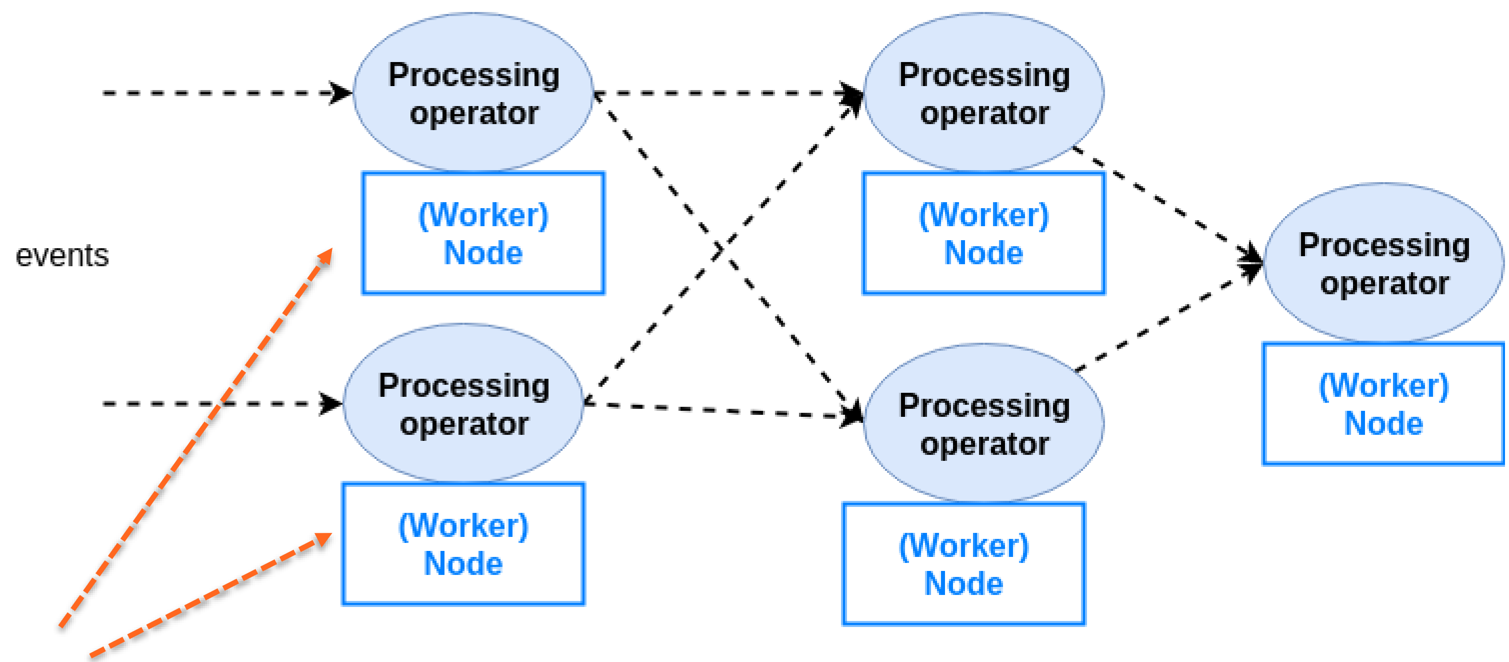
Latency and Throughput

- **Latency**
  - Subseconds! E.g., milliseconds
  - Max, min or percentile? → up to application requirements
- **Throughput**
  - How many events can be processed per second?
- **Goal: low latency and high throughput!**

Dataflow programming and stream processing

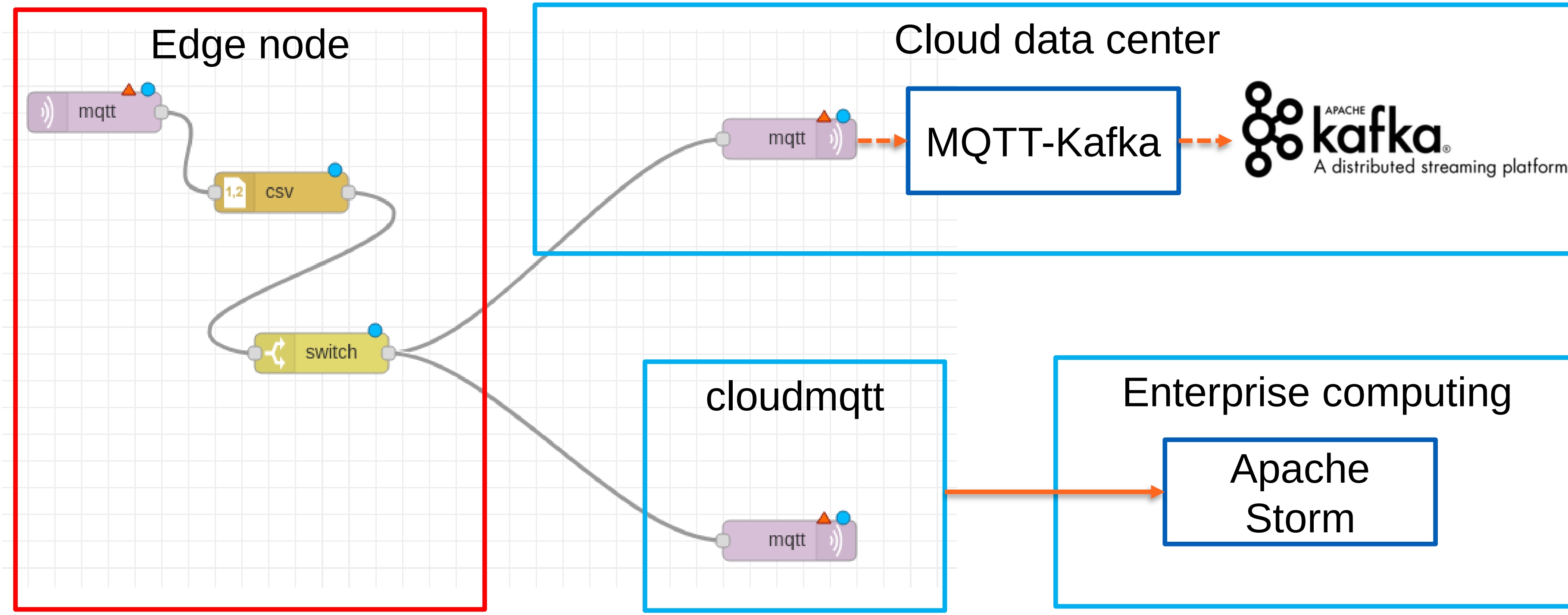
- **Data exchange between tasks**
  - Links in task graphs reflect data flows
- **Stream processing**
  - Centralized or distributed (in terms of computing resources)

Distributed processing topology in a cluster

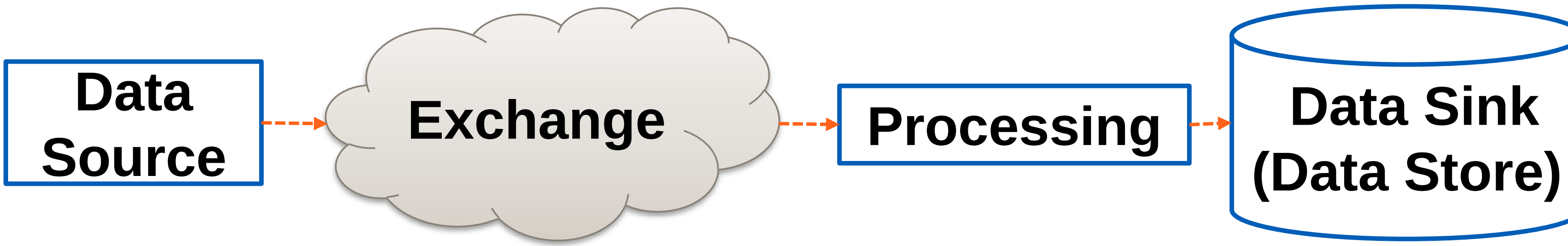


Nodes of a cluster (VMs, containers, Kubernetes)

Distributed processing topology in cross distributed systems



Recall: syntax and semantic problems

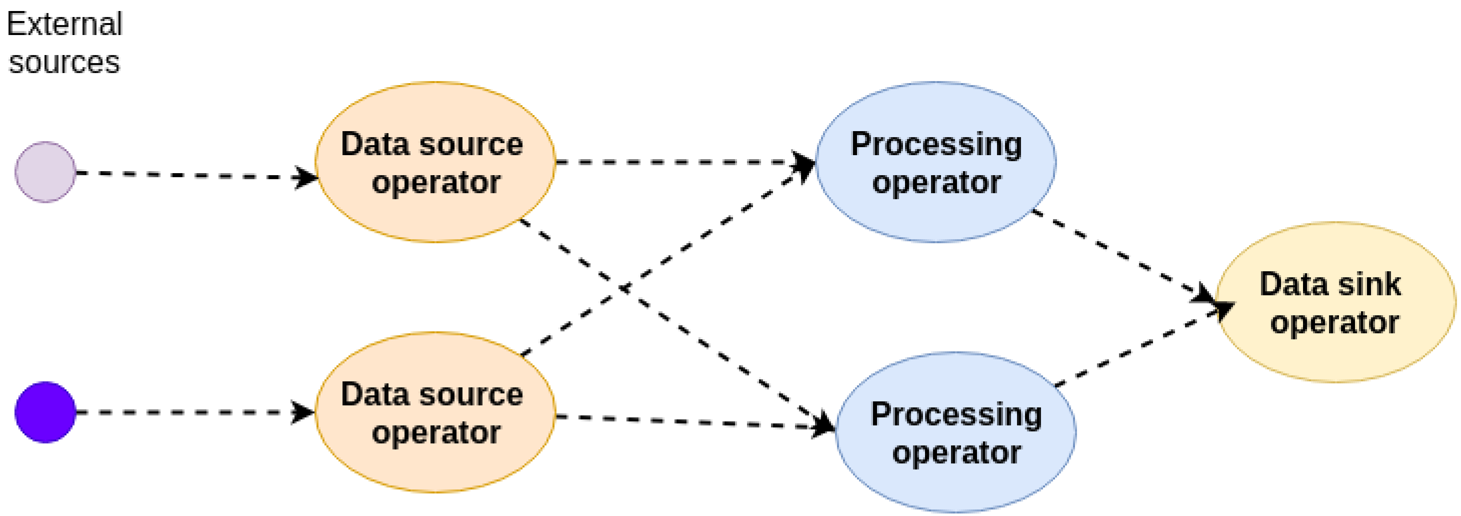


- Like Ingestion: processing might not understand the semantics of the data
- Solution?

Event pepresentation and streams

- **Event sources: via message brokers, database, websocket, different IO adapters/connectors, etc.**
- **Event representation & views**
  - POJO (Plain Old Java Object), CSV, Arvo format, etc.
  - SQL-alike tables
- **Event Stream**
  - Feed events from sources
- **Event Sink**
  - Feed results into sinks

Structure of streaming data processing programs



- **Data source operator: represents a source of streams**
- **Processing operators: represents processing functions**
- ***Native* versus *micro-batching***

Common concepts in existing frameworks

- **The way to connect data streams and obtain events**
  - Focusing very much on connector concepts and well-defined event structures (e.g., can be described JSON, POJO)
  - Assume that existing systems push the data
- **The way to specify "analytics"**
  - Statements and how they are glued together to process flows of events
  - High-level, easy to use
- **The engine to process analytics requests**
  - Centralized in the view of the user → so the user does not have to program complex distributed applications
  - Underlying it might be complex (for scalability purposes)
- **The way to push results to external components**

Common concepts in existing frameworks - programming level

- **How to streaming program?**
- **With programming languages**
  - Low level APIs
  - DSL
  - Java, Scala, Python (Spark, Flink, Kafka)
- **High-level data models**
  - KSQL
- **Flow description**
  - Node-RED

Common concepts in existing frameworks - key common concepts

- **Abstraction of streams**
- **Connector library for data sources/sinks**
  - Very important for application domains
- **Runtime elasticity**
  - Add/remove (new) operators (and underlying computing node)
- **Fault tolerance**

Why are the richness and diversity of connectors important?

Implementations

- **Apache Storm**
  - *https://storm.apache.org/*
- **Apache Spark**
  - *https://spark.apache.org/*
- **Apache Kafka Streams and KCQL**
  - strongly bounded to Kafka messaging
- **Apache Flink**
  - native, clustered, better data sources/sinks

Practical learning paths

- **Path 1: if you don’t have a preference and need challenges**
  - Apache Flink Stream API (e.g., with RabbitMQ/Kafka connectors)
- **Path 2: many of you have worked with Kafka**
  - Kafka Streams DSL (everything can be done with Kafka)
- **Path 3: for those of you who are working with Spark (and Python is the main programming language)**
  - Apache Spark Streaming
- **Path 4: for those who deal with MQTT brokers**
  - Apache Storm (but also Kafka, …): Spout and Bolt API or Stream API

Examples of Apache Flink

Apache Flink

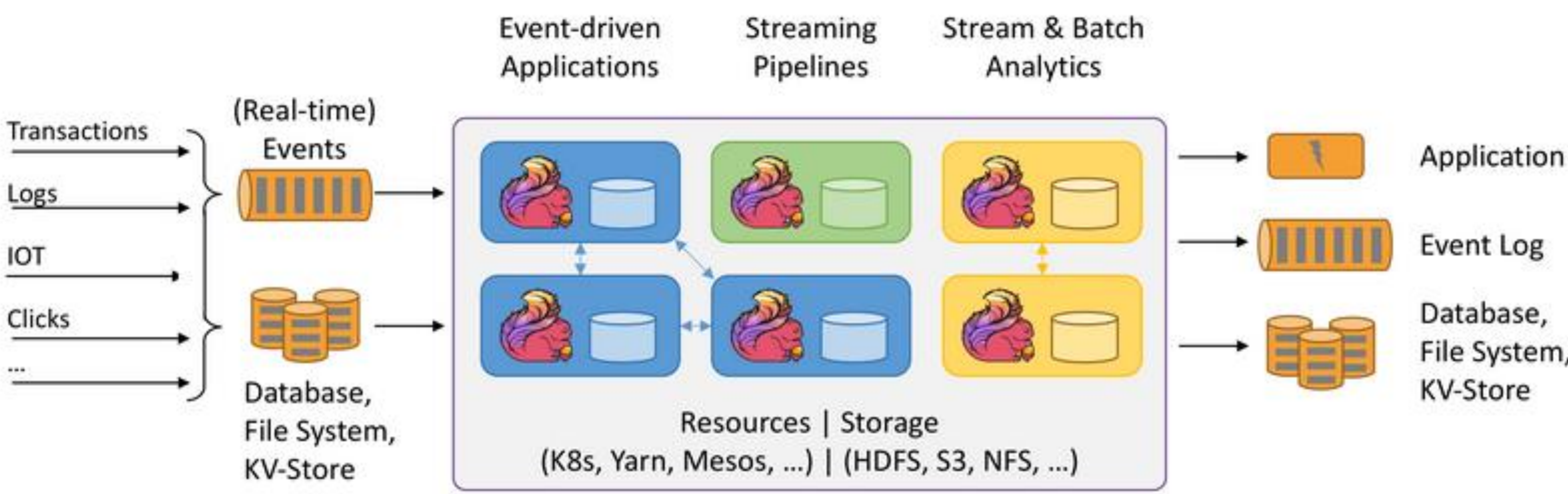
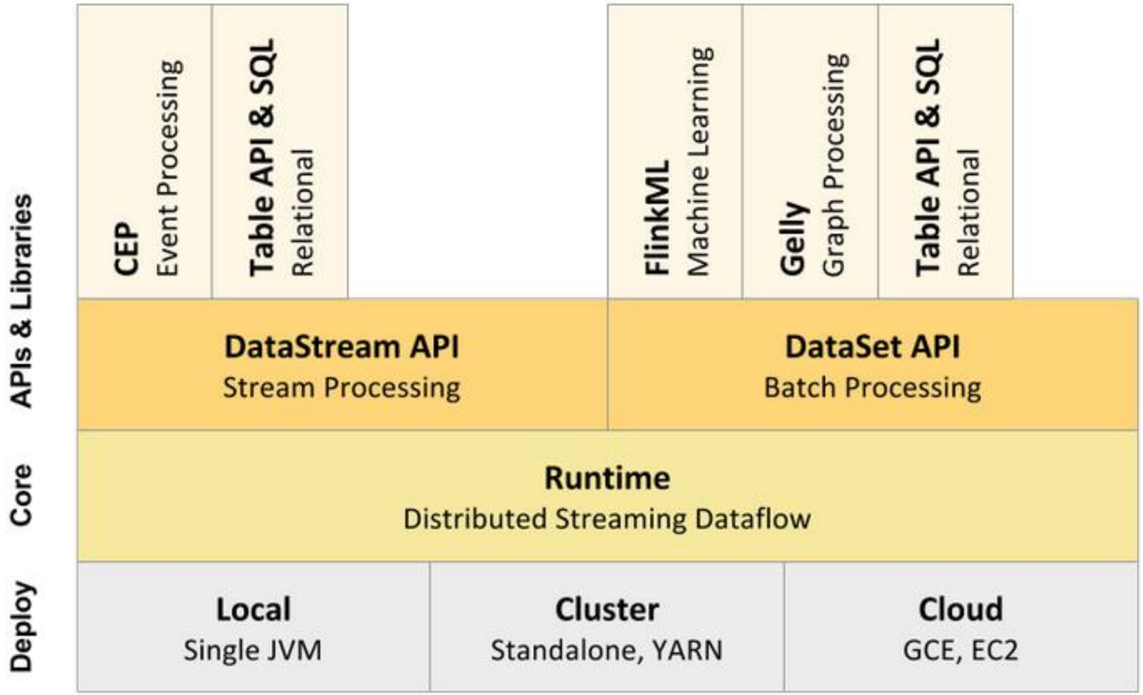


Figure source: https://flink.apache.org/

Apache Flink

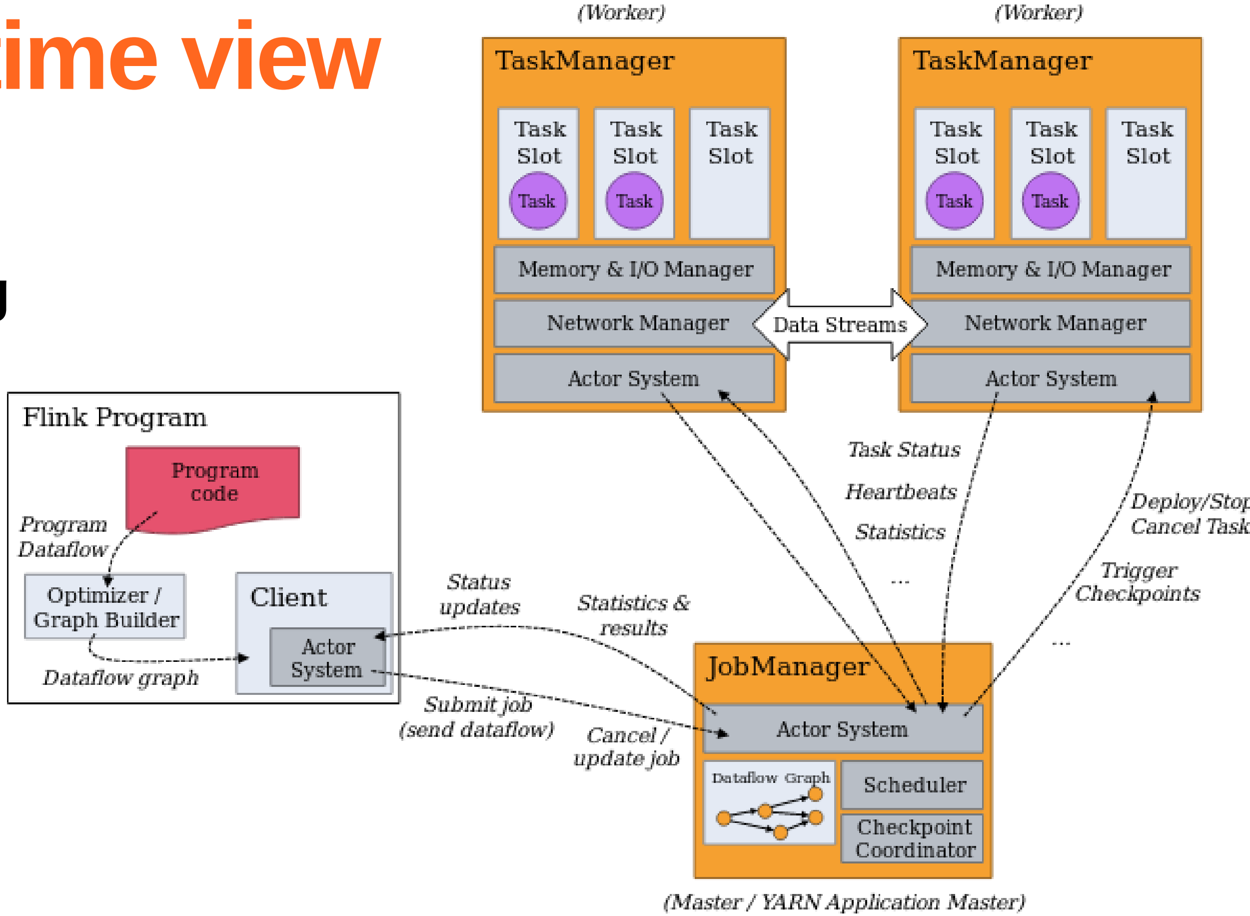


We focus only on DataStream API for understanding studied concepts

Source: https://ci.apache.org/projects/flink/flink-docs-release-1.9/internals/components.html

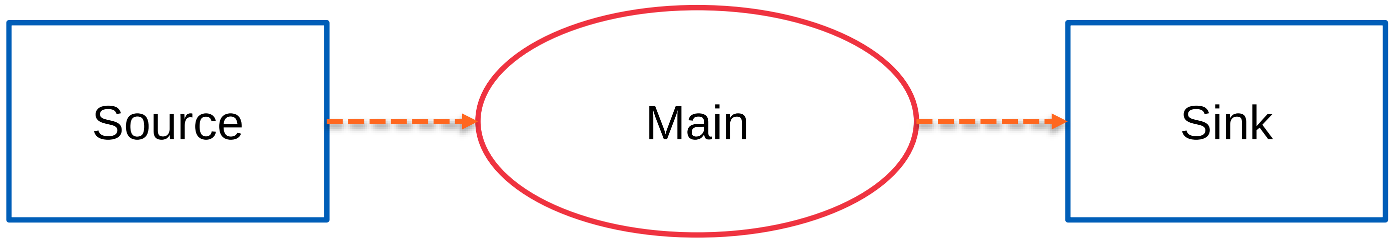
Flink runtime view

- **Parallelism**
- **Checkpointing**
- **Monitoring**



Source: https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/programming-model.html

Main elements in Flink applications



- **Rich set of sources and sinks via many connectors**

Connectors

- **Major systems in big data**
- **We have used many of them in our study**
  - Apache Kafka
  - Apache Cassandra
  - Elasticsearch (sink)
  - Hadoop FileSystem
  - RabbitMQ
  - Apache NiFi

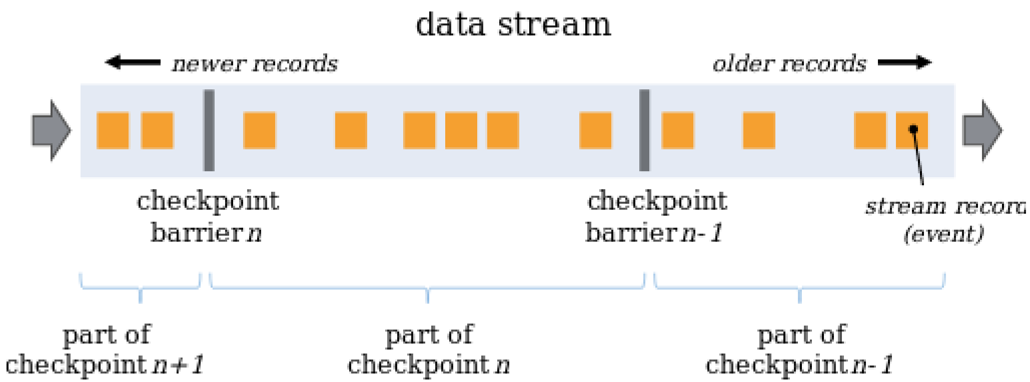
Main

- **Setting environments**
- **Handling inputs and outputs via data streams**
- **Key functions for processing data**
- **Stream processing flows**

Bounded and unbounded streams

Fault tolerance

Can be exactly once



Principles: checkpointing, restarts operators from the latest successful checkpoints
Need support from data stream sources/sinks w.r.t. (end-to-end) exactly once message receiving and result delivery

Stream processing flows

Split streaming data into different windows with a key for analytics purposes

Keyed Windows

```
stream
       .keyBy(...)               <-  keyed versus non-keyed windows
       .window(...)              <-  required: "assigner"
      [.trigger(...)]            <-  optional: "trigger" (else default trigger)
      [.evictor(...)]            <-  optional: "evictor" (else no evictor)
      [.allowedLateness(...)]    <-  optional: "lateness" (else zero)
      [.sideOutputLateData(...)] <-  optional: "output tag" (else no side output for late data)
       .reduce/aggregate/fold/apply()      <-  required: "function"
      [.getSideOutput(...)]      <-  optional: "output tag"
```

Source: https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/stream/operators/windows.html

Stream processing flows

Handling streaming data without a key for analytics purposes

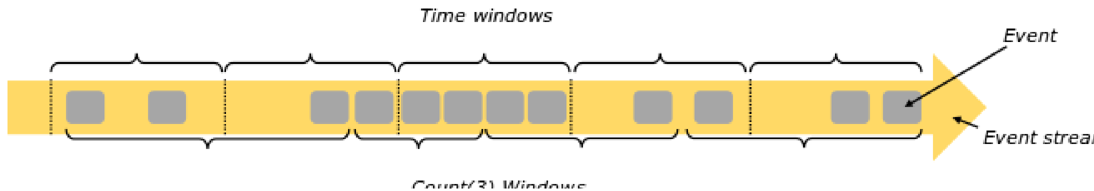
Non-Keyed Windows

```
stream
       .windowAll(...)           <-  required: "assigner"
      [.trigger(...)]            <-  optional: "trigger" (else default trigger)
      [.evictor(...)]            <-  optional: "evictor" (else no evictor)
      [.allowedLateness(...)]    <-  optional: "lateness" (else zero)
      [.sideOutputLateData(...)] <-  optional: "output tag" (else no side output for late data)
       .reduce/aggregate/fold/apply()      <-  required: "function"
      [.getSideOutput(...)]      <-  optional: "output tag"
```

Source: https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/stream/operators/windows.html

Windows and Times

Windows



Times

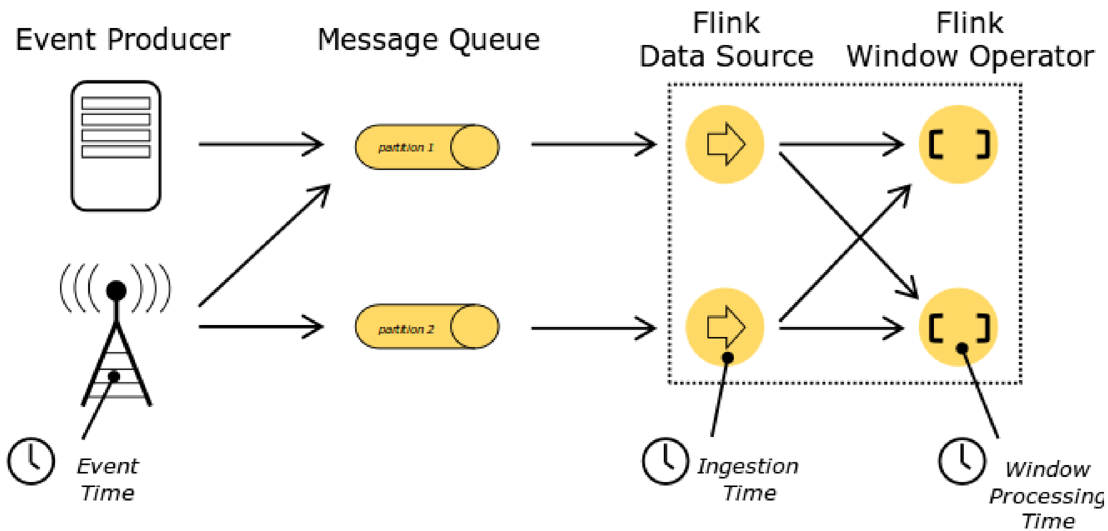
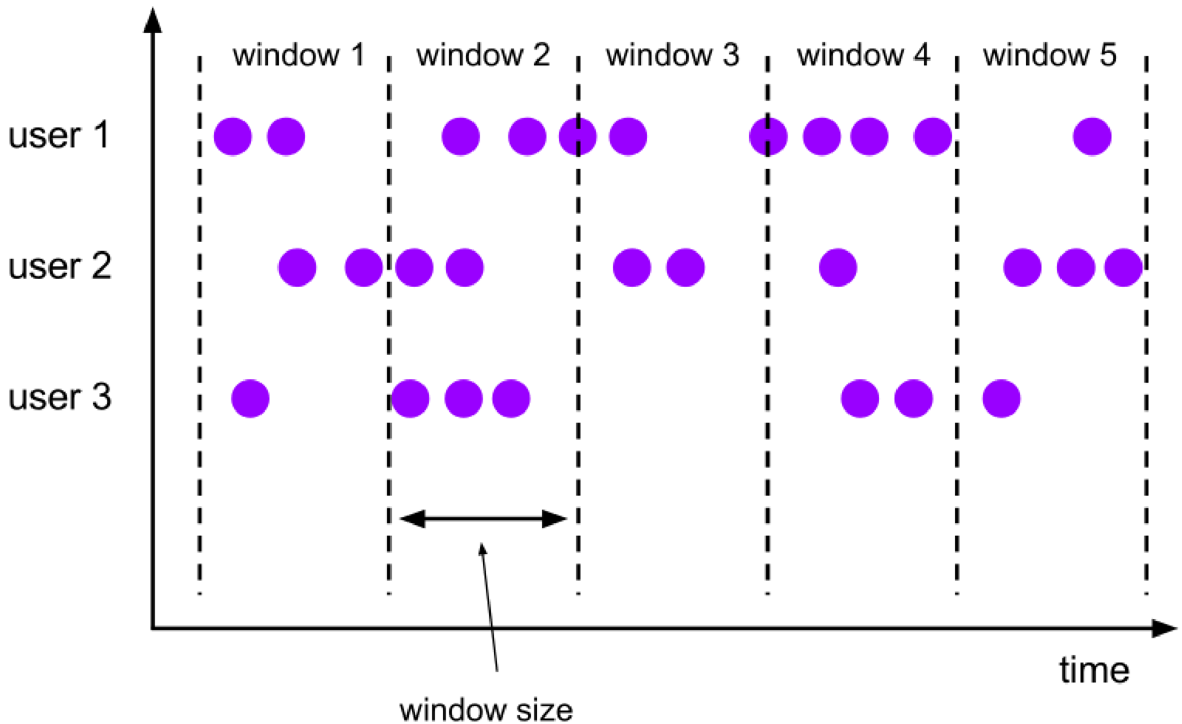


Figure source: https://ci.apache.org/projects/flink/flink-docs-release-1.9/concepts/programming-model.html

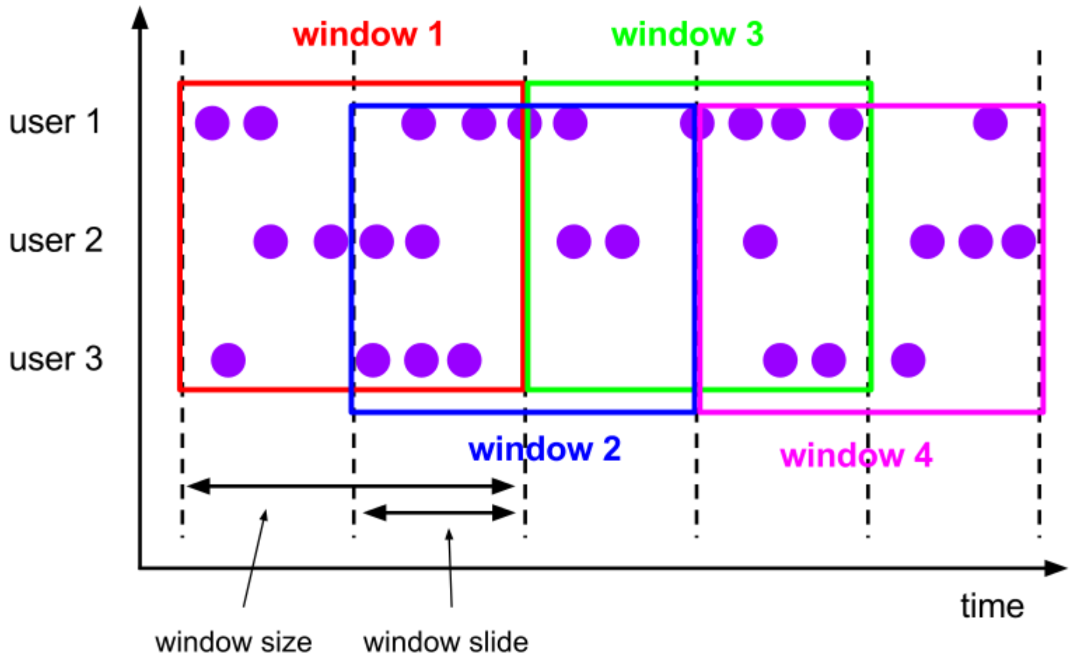
Batch/Tumbling Windows



Use cases: Period computation (e.g. stock, temperature, IoT data)

Source: https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/stream/operators/windows.html

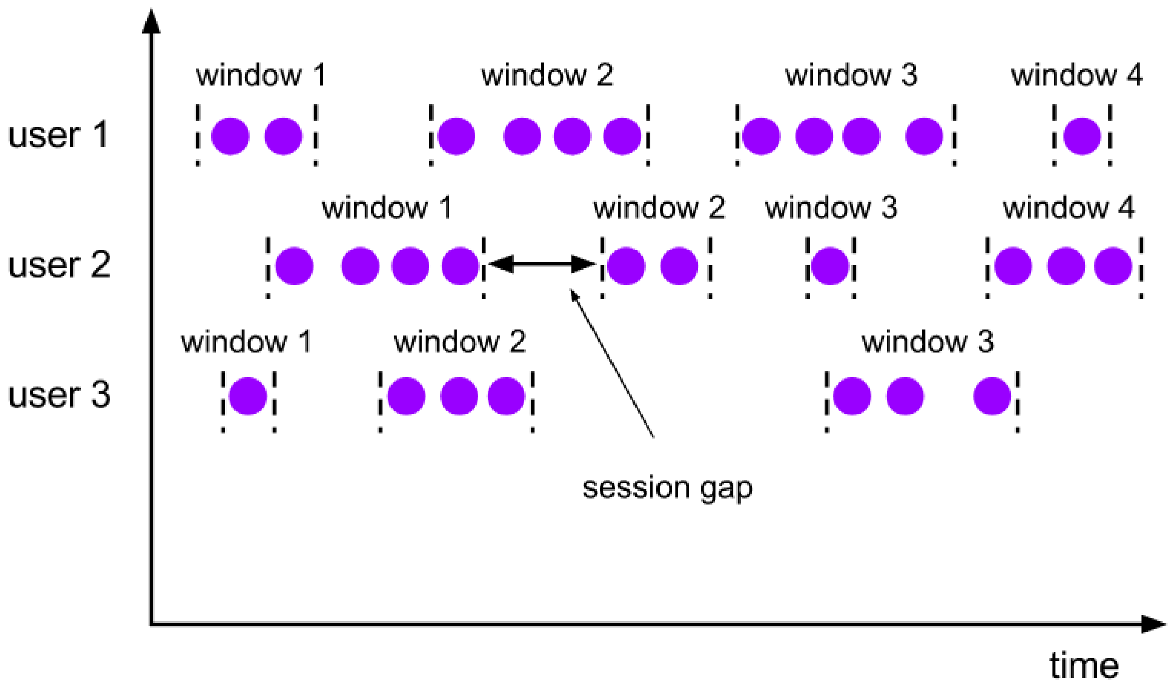
Sliding windows



Use cases: Moving average

Source: https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/stream/operators/windows.html

Session Windows



Use cases: Web/user activities clicks

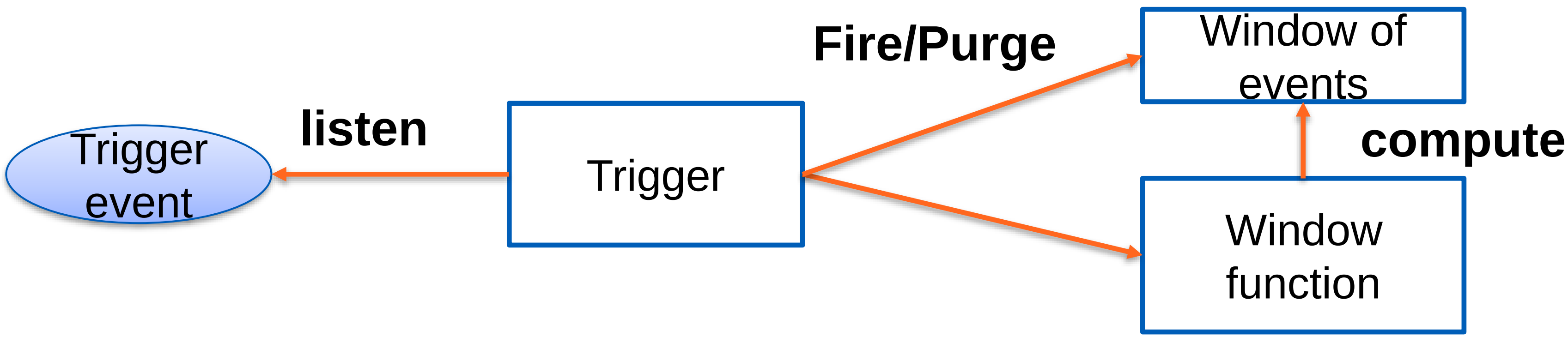
Source: https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/stream/operators/windows.html

Window Functions

- **Reduce Function**
  - Reduce two inputs
- **Aggregate Function**
  - Add an input into an accumulator
- **Fold Function**
  - Combine input with an output
- **ProcessWindow Function**
  - Get all elements of the windows and many other information so that you can do many tasks

Triggers & Evictor

- **Trigger: determine if a window is ready for window functions**



Evictor: actions after the trigger fires and before and/or after the windows function is called

Example with Base Transceiver Station

Data in our git

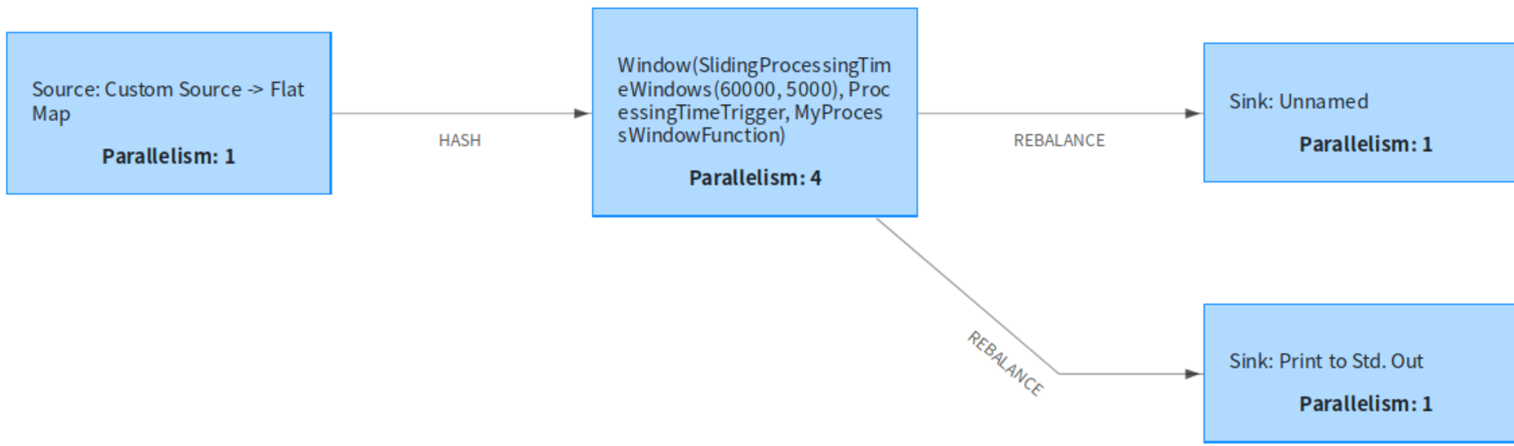
```
station_id,datapoint_id,alarm_id,event_time,value,valueThreshold,isActive,storedtime
1161115016,121,308,2017-02-18 18:28:05 UTC,240,240,false,
1161114050,143,312,2017-02-18 18:56:20 UTC,28.5,28,true,
1161115040,141,312,2017-02-18 18:22:03 UTC,56.5,56,true,
1161114008,121,308,2017-02-18 18:34:09 UTC,240,240,false,
1161115040,141,312,2017-02-18 18:20:49 UTC,56,56,false,
```

Simple example

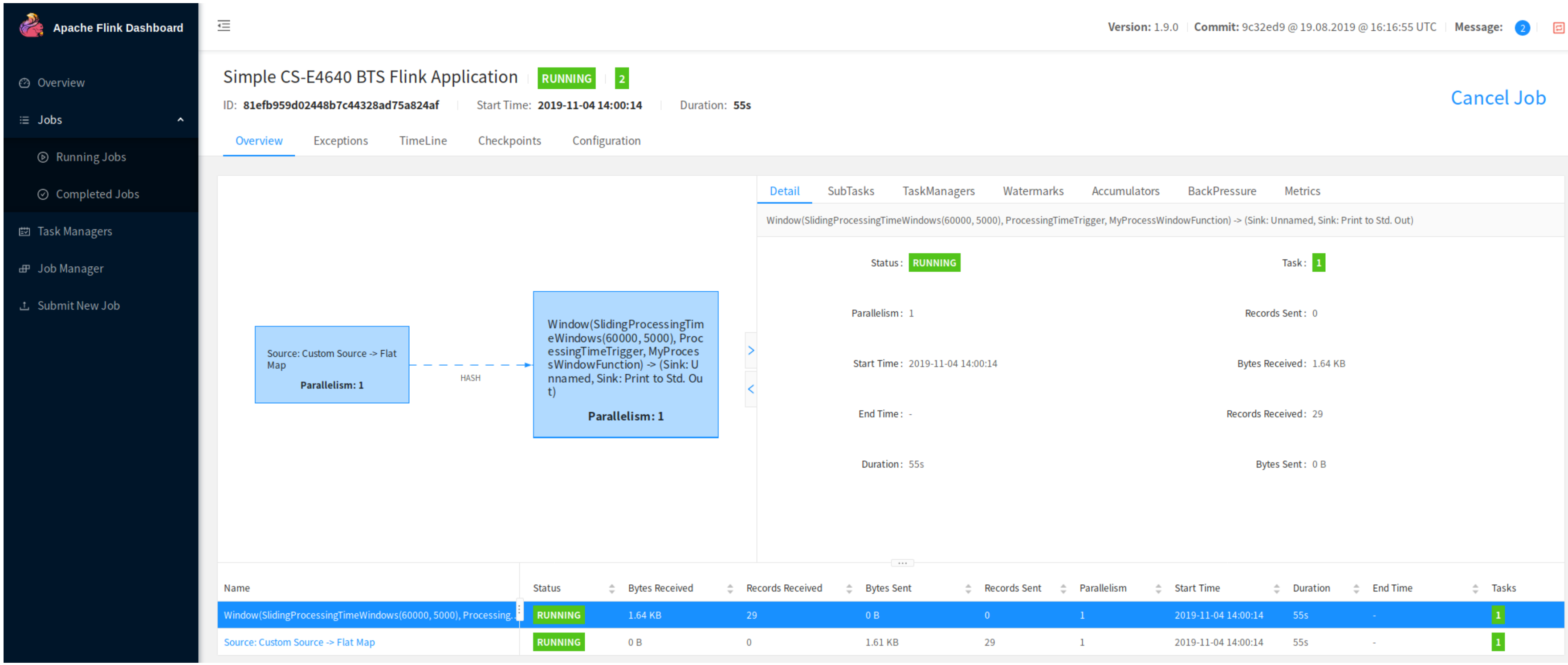
See the code in our git:

https://version.aalto.fi/gitlab/bigdataplatforms/cs-e4640-2019/blob/master/tutorials/streamingwithflink/simplebts/src/main/java/fi/aalto/cs/cse4640/SimpleAlarmAnalysis.java

Simple example



Monitoring



Apache Flink Dashboard
Overview
Jobs
Running Jobs
Completed Jobs
Task Managers
Job Manager
Submit New Job
Version: 1.9.0
Commit: 9c32ed9 @ 19.08.2019 @ 16:16:55 UTC
Message: 2
Simple CS-E4640 BTS Flink Application
RUNNING
2
ID: 81efb959d02448b7c44328ad75a824af
Start Time: 2019-11-04 14:00:14
Duration: 55s
Cancel Job
Overview
Exceptions
TimeLine
Checkpoints
Configuration
Source: Custom Source -> Flat Map
Parallelism: 1
HASH
Window(SlidingProcessingTimeWindows(60000, 5000), ProcessingTimeTrigger, MyProcessWindowFunction) -> (Sink: Unnamed, Sink: Print to Std. Out)
Parallelism: 1
Detail
SubTasks
TaskManagers
Watermarks
Accumulators
BackPressure
Metrics
Window(SlidingProcessingTimeWindows(60000, 5000), ProcessingTimeTrigger, MyProcessWindowFunction) -> (Sink: Unnamed, Sink: Print to Std. Out)
Status: RUNNING
Task: 1
Parallelism: 1
Records Sent: 0
Start Time: 2019-11-04 14:00:14
Bytes Received: 1.64 KB
End Time: -
Records Received: 29
Duration: 55s
Bytes Sent: 0 B
Name
Status
Bytes Received
Records Received
Bytes Sent
Records Sent
Parallelism
Start Time
Duration
End Time
Tasks
Window(SlidingProcessingTimeWindows(60000, 5000), Processing...
RUNNING
1.64 KB
29
0 B
0
1
2019-11-04 14:00:14
55s
-
1
Source: Custom Source -> Flat Map
RUNNING
0 B
0
1.61 KB
29
1
2019-11-04 14:00:14
55s
-
1

One of the successful project from Europe: originally from TU Berlin

Alibaba cloud:

"Flink can process over 472 million transactions per second during business peaks, which is truly astronomical"

Source: https://www.alibabacloud.com/blog/why-did-alibaba-choose-apache-flink-anyway_595190

"Amazon Kinesis Data Analytics includes open source libraries based on Apache Flink" (From https://aws.amazon.com/kinesis/data-analytics/)

Summary

- **Facts:**
  - Stream processing is important in big data platforms
  - There are many frameworks, but they have quite common concepts
    - *You should focus on the concept and pickup a suitable one for your work*
  - Core concepts can be implemented at different levels
    - *Programming languages, DSL, SQL-style*
- **Thoughts:**
  - Think about combining stream with batch analytics to produce a comprehensive platform!
  - There are many real world-big data needs stream processing

Summary

- **Focus:**
  - Practical programming with one of the stacks:
    - *Apache Flink Stream API (with different connectors)*
    - *Apache Spark*
    - *Kafka Streams*
    - *Apache Storm*
  - Check the common concepts in other tools/systems
- **Action:**
  - Work on use cases where you can use stream analytics (as a user/developer) → there are many interesting analytics
  - Provision services for stream processing (as a platform)

Thanks!

Hong-Linh Truong
Department of Computer Science

rdsea.github.io