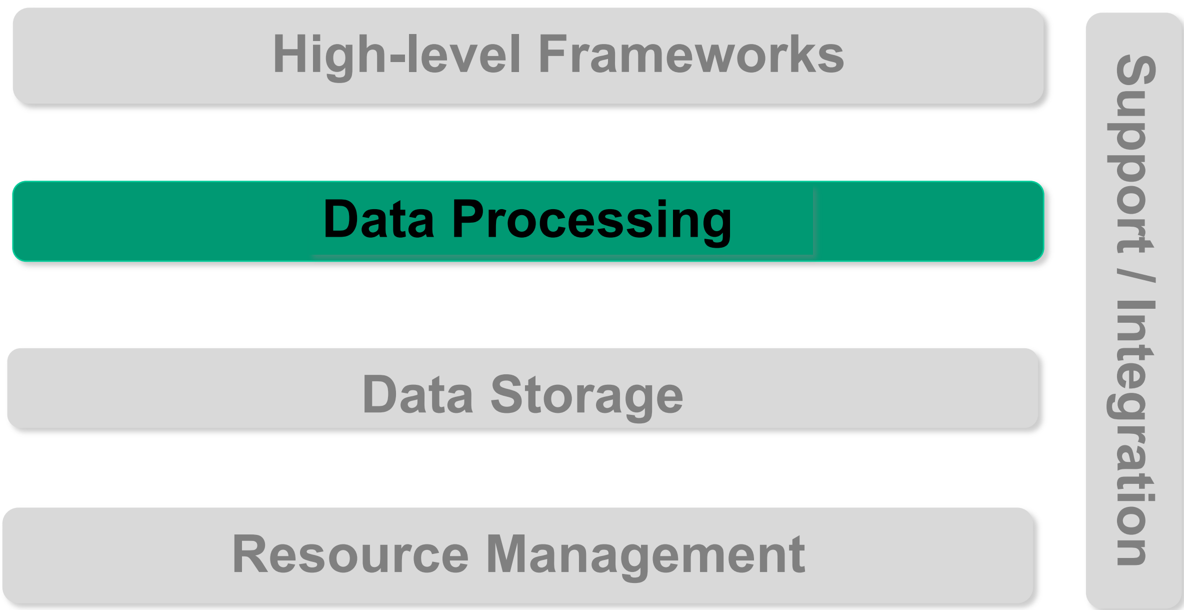
TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA
Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Introduction to Data Stream Processing

Corso di Sistemi e Architetture per Big Data
A.A. 2024/25
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



Why data stream processing?

- Applications such as:
  - Sentiment analysis on tweets @Twitter
  - User profiling @Yahoo!
  - Tracking of query trend evolution @Google
  - Fraud detection in financial transactions
  - Real-time advertising
  - Healthcare analytics involving IoT medical sensors
- Require:
  - Continuous **processing** of unbounded **data streams** generated by multiple and distributed sources
  - In (near) **real-time** fashion

Why data stream processing?

- In the early years data stream processing (**DSP**) was considered a solution for very specific problems (e.g., financial tickers)
- Now we have more general settings
  - E.g., social media, Internet of Things

Why data stream processing?

- Decrease latency to obtain results and improve data freshness
  - Events are processed close to the time they are generated
  - Applications respond to events as they occur
  - No delays involved with batch processing
  - No data persistence on stable storage
- Simplify data analytics pipelines and underlying infrastructure

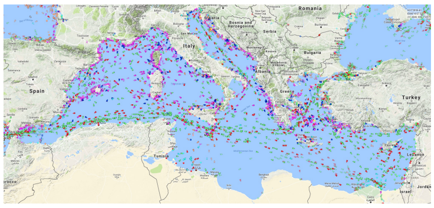
Data stream

- "A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety. Queries over streams run continuously over a period of time and incrementally return new results as new data arrive."

  Golab and Özs, Issues in data stream management, ACM *SIGMOD Rec.*, 2003.
- A data stream refers to both velocity and variety of Big data
- A stream is an unbounded sequence of tuples, where a tuple is an ordered list of values

Data stream: example

- Data stream related to maritime traffic in the Mediterranean



```
0x3b62baab6210a8e69d3e7f9df53d000c83d00fd0,2,
15.247220,37.287770,163,511,01-06-15 0:00,AUGUSTA,12
0x0fe9acdb3675a8a2942fafbd4af61bc37e44c0ec,146,
23.694910,37.313620,13,15,01-06-15 0:00,SALERNO,88
0xb35dc6acdc29f2241296c44384fa2b0f7044d257,20,
15.669920,38.387740,339,339,01-06-15 0:00,MESSINA,66
…
```

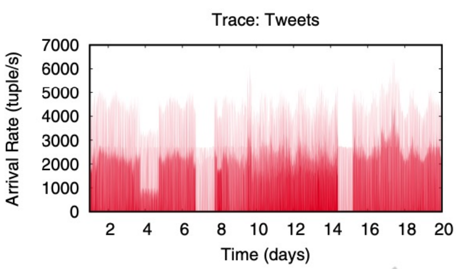
tuples

Each tuple contains the fields:

`SHIP_ID,SPEED,LON2,LAT2,COURSE,HEADING,TIMESTAMP, departurePortName,Reported_Draught`

Traditional DSP challenges

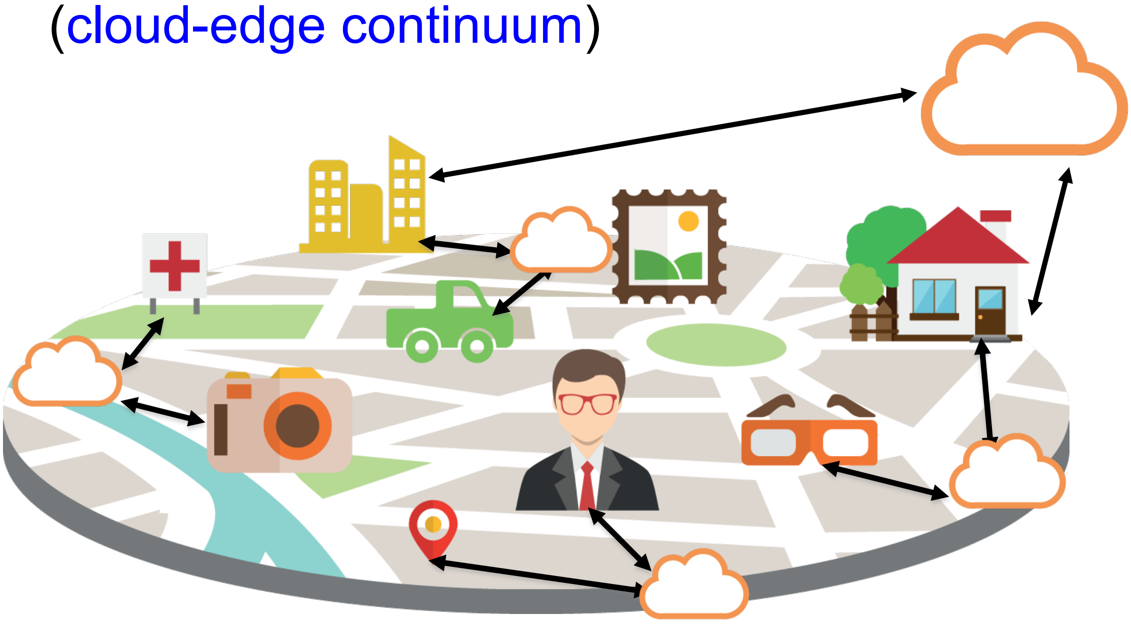
- Stream data can arrive at high velocity, with high volumes and highly variable arrival patterns
  - High resource requirements for processing



- Processing stream data has real-time aspects
  - Stream processing applications have QoS requirements, e.g., end-to-end latency
  - Must be able to react to events as they occur
- Faults can happen during processing

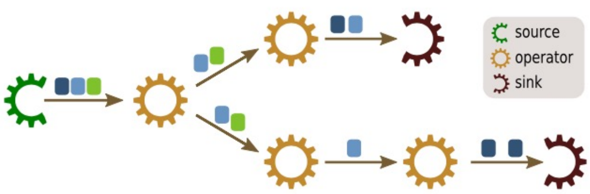
Challenges for DSP in Cloud-Edge continuum

- Goals: increase scalability and reduce latency
- How? Rely not only on Cloud resources but also on distributed and near-edge computation (cloud-edge continuum)



DSP application model

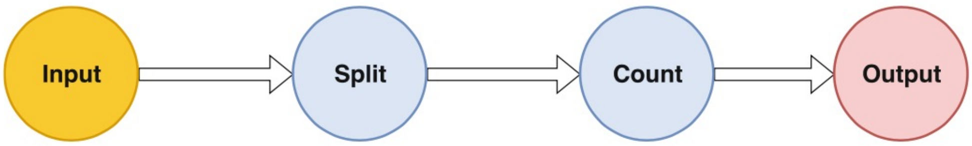
- A DSP application is made of a network of *operators* (processing elements) connected by *streams*, at least one *data source* and at least one *data sink*
- Represented by a directed dataflow graph
  - Graph vertices: operators
  - Graph edges: streams
  - Graph is often referred to as *topology*



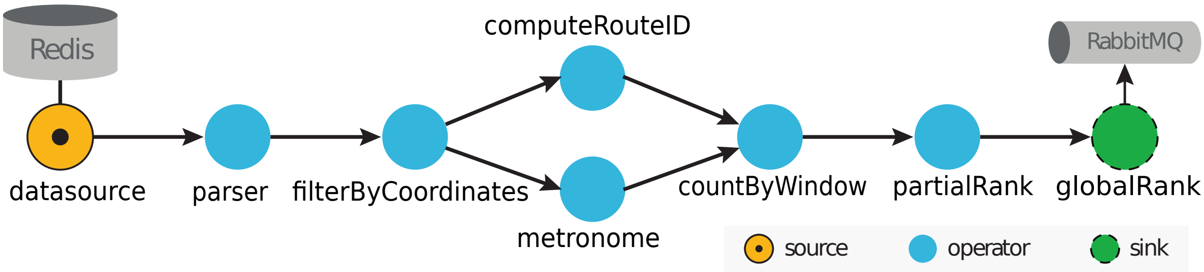
- Graph is typically acyclic: directed acyclic graph (DAG)
  - In DAGs, data can only move from upstream tasks to downstream task
  - Most DSP systems support only DAGs, few systems (e.g., Flink) support also loops
- Topology does not usually change during processing

DSP application model: examples

- DAG for WordCount application in DSP salsa



- DAG for NYC taxi streaming analysis: data streams from NYC taxis are processed to find the top-10 most frequent routes during the last 30 minutes



DSP programming model

- Dataflow programming
  - Programming paradigm that models a program as a directed graph of data (dataflow) flowing between operations
  - Pioneered by Jack Dennis and his students at MIT in the 1960s
- Examples
  - Apache NiFi: automates dataflow between systems
  - Apache Flink: stream and batch processing
  - Apache Beam: unifies batch and streaming data processing on top of several execution engines
  - TensorFlow: ML library based on dataflow programming

DSP programming model

- What to we need?
- **Dataflow composition**: create the topology associated with the DAG for a DSP application
- **Dataflow manipulation**: use processing elements (i.e., operators) to perform data transformations

Dataflow composition: How to define a DSP application

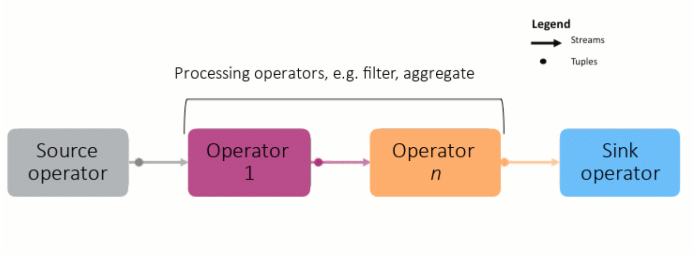
- Explicit way: describe topology
  - Explicitly defines operators (built-in or user-defined) and how they are connected in the DAG
  - Used in many DSP systems (e.g., Flink, Storm, Spark Streaming)
- Implicit way: use formal language
  - Declarative languages that specify query result (SQL-like)
    - Streams Processing Language (SPL) in IBM Cloud Pak https://community.ibm.com/community/user/viewdocument/resources-for-streams-developers
    - SQL support in Flink provided by Apache Calcite https://calcite.apache.org
  - Procedural languages that specify composition of operators
    - e.g., SQuAl (Stream Query Algebra) used in Aurora/Borealis
- The first offers more flexibility, the latter more rigor and expressiveness

Dataflow manipulation

- How streaming data is manipulated by the operators in the DAG?
- Operator properties:
  - Operator type
  - Operator state
  - Windowing

DSP operator

- Self-contained processing element that
  - Transforms one or more input streams into another stream
  - Can execute a generic user-defined code
    - Algebraic operation (filter, aggregate, join, ..)
    - User-defined and possibly complex operation (e.g., part-of-speech-tagging, machine learning algorithm)
  - Multiple operators execute at the same time on different streams



DSP operator: types

- Edge adaptation: converting data from external sources into tuples that can be consumed by downstream operators
- Aggregation: collecting and summarizing a subset of tuples from one or more streams
- Splitting: partitioning a stream into multiple streams
- Merging: combining multiple input streams (e.g., join)

DSP operator: types

- Logical and mathematical operations: applying different logical processing, relational processing, and mathematical functions to tuple attributes
- Sequence manipulation: reordering, delaying, or altering the temporal properties of stream
- Custom data manipulations: applying data mining, machine learning, ...

DSP operator: state

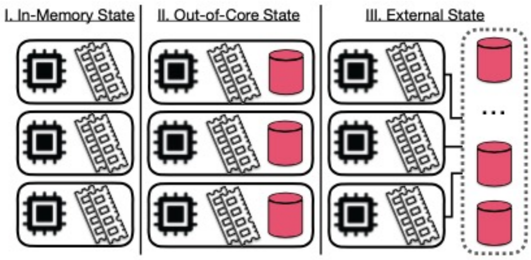
- Operator can be either stateless or stateful
- **Stateless**: processing depends only on current input
  - Operator knows nothing about state and thus processes tuples independently of each other, independently of prior history or even from tuple arrival order
  - E.g., filter, map
  - Easily parallelizable
  - No synchronization in a multi-threaded context
  - Easy restart upon failures (no need to recover state)
  - In a nutshell: easy to manage

DSP operator: state

- **Stateful**: keeps some sort of state (i.e., information across multiple tuples) that operator can read and modify during execution,
- Examples of stateful operator
  - Aggregation or summary of tuples per minute/hour
  - When an application searches for certain patterns, the state will store the sequence of events encountered so far
  - When training a machine learning model over a stream of data points, the state holds the current version of the model parameters
- State makes management more complex

DSP operator: state

- State may be stored in different ways:
  - Entirely stored within in-memory data structures and replicated to disk only for fault tolerance
  - Entirely stored on non-volatile memory (e.g., disk)



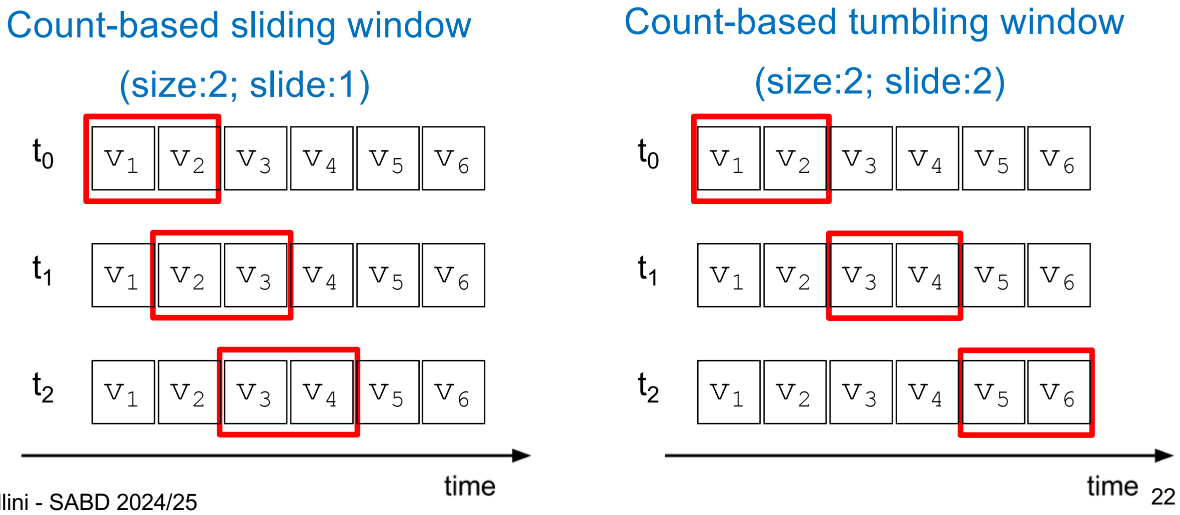
  - Hybrid solution: partially stored in memory for improved performance and flushed to disk to scale in size
  - Stored on external storage service (e.g., Redis)
- State is mostly private to operator but in some system can be shared between operators
  - Shared state makes execution even more complex

Windowing

- **Window**: buffer associated with an operator input port that retains incoming tuples, allowing computation to be applied on the set of tuples in the buffer
  - E.g., to find the most frequently purchased items over the last hour
- Window is characterized by:
  - **Size**: amount of data that should be buffered before triggering the operator execution
    - Statically defined: time-based (e.g., 30 seconds) or count-based (e.g., the last 100 tuples)
    - Dynamically defined: session-based, where the window is defined by session boundaries
  - **Sliding interval**: how the window moves forward
    - Time-based or count-based

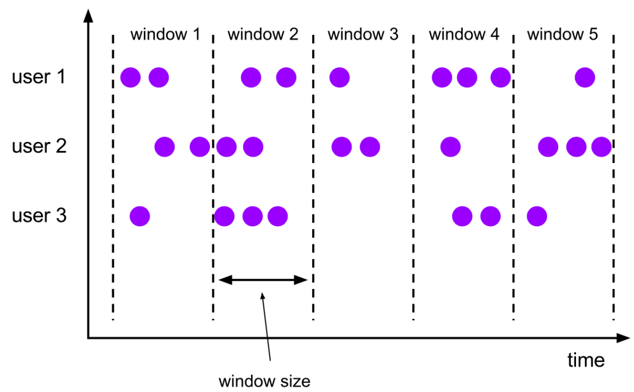
Windowing: patterns

- Different windowing patterns by combining window size and sliding interval
  - **Sliding window**: window size and sliding interval are different, single tuples may be included in multiple consecutive windows
  - **Tumbling** (or fixed) **window**: sliding interval is equal to window size, consecutive windows do not overlap

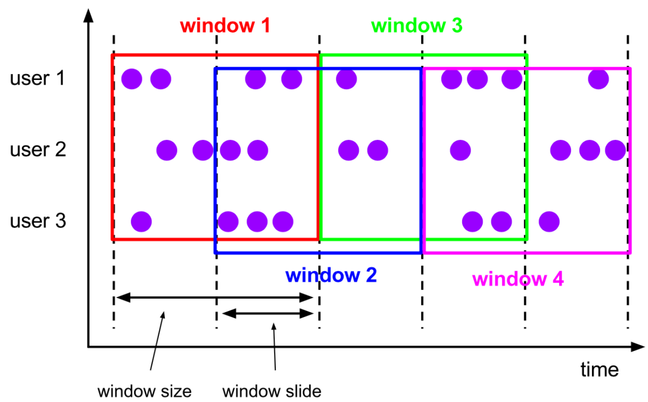


Windowing: patterns

Tumbling windows



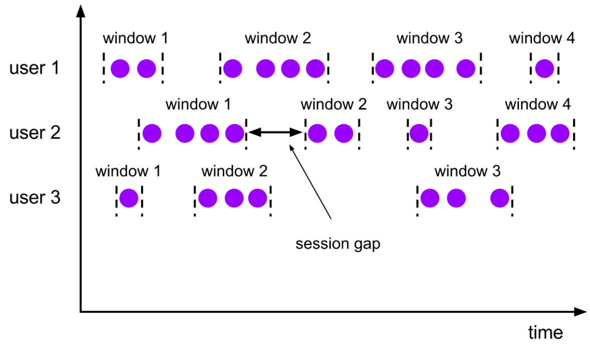
Sliding windows



Windowing: patterns

- Window can be also dynamically defined: session window
  – Dynamic size of window length, depending on inputs
  – Starts with an input and expands itself if the following input has been received within the gap duration
  – Closes when there's no input received within the gap duration after receiving the latest input
  – Enables to group events until there are no new events for specified time duration (inactivity)

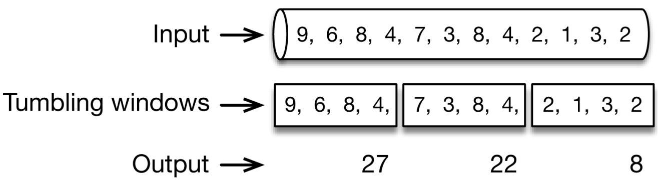
Session windows



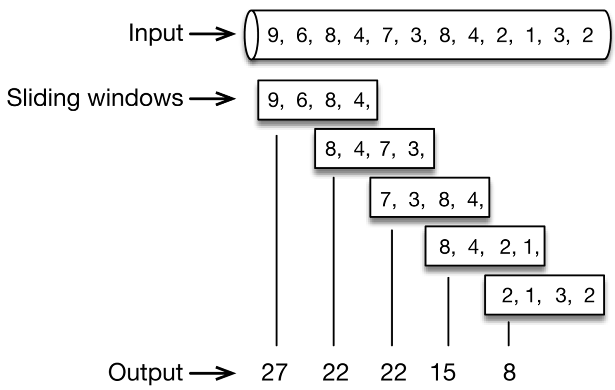
Windowing: emit

- Once a trigger determines that a window is ready for processing, it fires, i.e., emits the results of the current window
- Example: tumbling/sliding time window of 1 minute that sums the values

Tumbling window of 1 minute that sums the values



Sliding window of 1 minute that sums the values every half minute

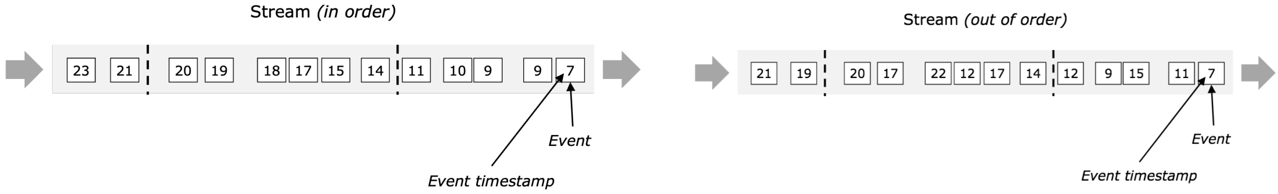


Windowing: which pattern?

- Choosing the appropriate window type requires careful consideration of data and processing requirements
- A few rules of thumb
  - Use tumbling windows to segment a data stream into distinct, non-overlapping segments, and perform computiation on each segment
  - Keep in mind that sliding windows can produce overlapping results
    - Sliding windows are ideal for use cases where you need to closely monitor changes over time or compare them to previous readings
  - Consider that windows can span long periods (such as days, weeks, or months), thus accumulating very large state
    - Depending on DSP system, sliding windows may be more memory-consuming

Windowing: out-of-order tuples

- Out-of-order tuples: tuples in a DSP system may arrive later than expected relative to the order they should appear
- Why? Network delays, asynchronous data sources, latency issues, computing delays, specific operations on streams (e.g., join)
- Issues caused by out-of-order tuples
  - Incorrect computation
  - State inconsistency
  - Increased complexity: additional logic in DSP system to handle them, which can increase computational complexity and memory usage



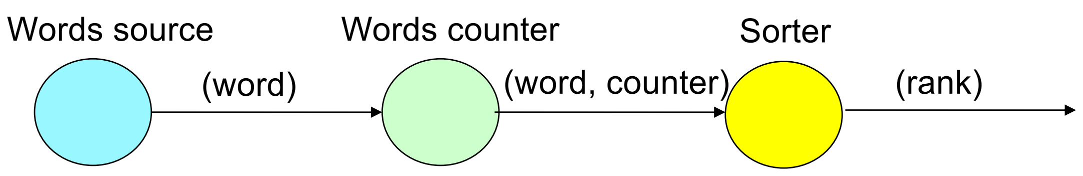
Windowing: out-of-order tuples

- How can DSP system handle out-of-order tuples?
- Multiple solutions, including:

1. Discard
2. Buffer and reorder
   - Systems can buffer out-of-order tuples temporarily until they are able to fit into the correct window
   - Systems can reorder tuples as they arrive, can be complex and memory-intensive
3. Admit late data without reordering up to a lateness bound
4. Use watermarks
   - Special markers to track progress of data processing: "no tuple older than this timestamp will arrive"

"Hello World": a variant of WordCount

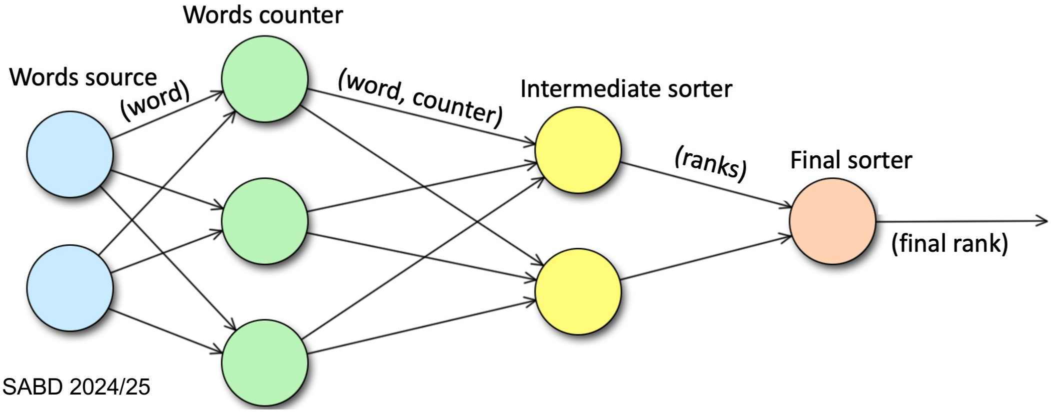
- Goal: emit top-k words in terms of occurrence when there is a rank update



- Which operators can be a performance bottleneck?
- How to scale DSP application in order to sustain a traffic load increase?

“Hello World”: a variant of WordCount

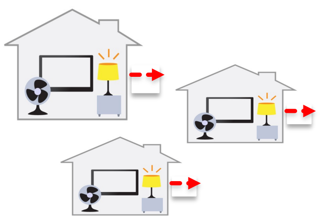
- The usual approach: replicate operators whenever possible
- Use data parallelism (aka *operator fission*) and redesign application by splitting sorting into two stages: multiple intermediate sorters followed by a final sorter
- To partition the downstream flow across multiple replicas, use key-based partitioning, so that operator state is also partitioned among replicas



Example of DSP application: DEBS’14 GC

https://debs.org/grand-challenges/2014

- Real-time analytics over high volume sensor data: analysis of energy consumption measurements for smart homes
  - Smart plugs deployed in households and equipped with sensors that measure values related to power consumption



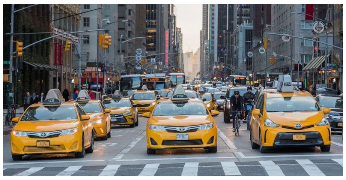
- Input data stream: `id, timestamp, value, property, plug_id, household_id, house_id,`
  e.g., `2967740693, 1379879533, 82.042, 0, 1, 0, 12`
- *Query 1*: make load forecasts based on current load measurements and historical data
  - Output data stream:
    `ts, house_id, predicted_load`
- *Query 2*: find outliers concerning energy consumption
  - Output data stream:
    `ts_start, ts_stop, household_id, percentage`

Example of DSP application: DEBS’15 GC

https://debs.org/grand-challenges/2015

- Real-time analytics over high volume spatio-temporal data streams: analysis of taxi trips based on data streams originating from New York City taxis
- Data stream composed of tuples
- Each tuple includes: pickup and drop-off points (longitude and latitude), corresponding timestamps plus information related to payment

```
07290D3599E7A0D62097A346EFCC1FB5,E7750A37CAB07D0DFF0AF
7E3573AC141,2013-01-01 00:00:00,2013-01-01
00:02:00,120,0.44,-73.956528,40.716976,-
73.962440,40.715008,CSH,3.50,0.50,0.50,0.00,0.00,4.50
```



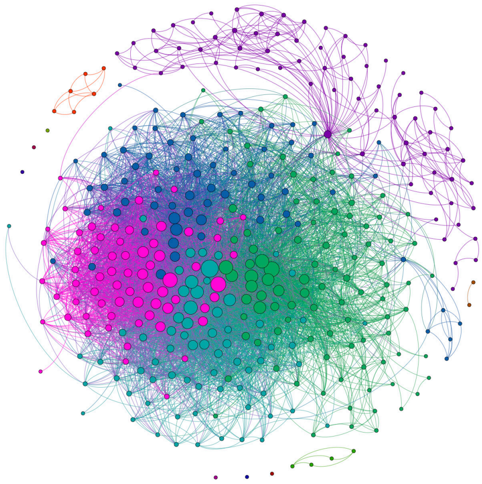
Example of DSP application: DEBS’15 GC

- *Query 1*: identify top-10 most frequent routes during the last 30 minutes
- *Query 2*: identify areas that are currently most profitable for taxi drivers
- Both queries rely on sliding window operators
  - Continuously evaluate query results

Example of DSP application: DEBS'16 GC

https://debs.org/grand-challenges/2016

- Real-time analytics for a dynamic (evolving) social-network graph
- *Query 1*: identify the posts that currently trigger the most activity in the social network
- *Query 2*: identify large communities that are currently involved in a topic
- Require continuous analysis of dynamic graph considering multiple streams that reflect graph updates



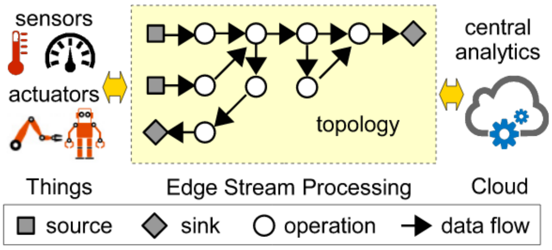
Distributed DSP system

- Distributed system that executes DSP applications
  - Continuously calculates results for long-standing queries
  - Over potentially infinite data streams
  - Using stateless or stateful operators
- System nodes may be heterogeneous
  - Computing capacity, network bandwidth, …
- Must be highly optimized and with minimal overhead so to deliver real-time response
- Must manage a number of issues
  - Place operators on computing nodes (application deployment)
  - Hide node and operator failures
  - …

Distributed DSP system

- Traditionally runs in a locally distributed cluster within a data center (also Cloud-based)
- Assumptions:
  - Scale out
    - Commodity servers
    - Data-parallelism (operator parallelism) is king
  - Designed to handle failures

- Newer environments: edge computing and Cloud-edge continuum



Top DSP frameworks

- All having a distributed architecture
- Apache Storm
- Apache Flink
- Apache Spark Streaming
- Kafka Streams
- Cloud-based services
  - Amazon Kinesis https://aws.amazon.com/kinesis
  - Azure Stream Analytics https://azure.microsoft.com/products/stream-analytics
  - Google Cloud Dataflow https://cloud.google.com/products/dataflow

Distributed DSP systems: processing models

- Processing models:
  - One-at-a-time: each individual incoming tuple is processed sequentially, one by one, as it arrives
    - E.g, Apache Storm
  - Windowed (or *micro-batched*): multiple tuples are grouped in a buffer before being processed
    - E.g, Apache Spark Streaming
- One-at-a-time: pros and cons

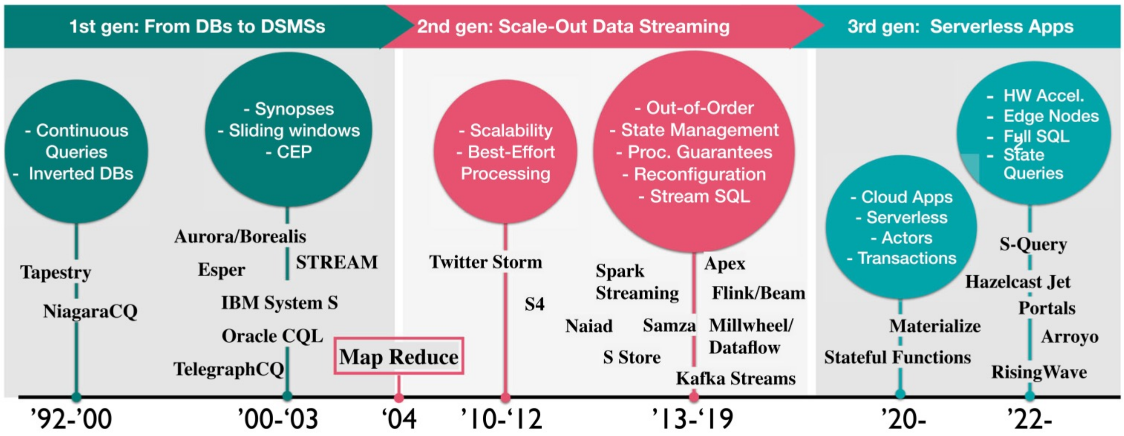
Low latency: immediate processing upon tuple arrival	Limited context: no global or historical view of data
Simple state management: often stateless or minimal	Not easy to perform aggregations over time/count
Real-time responsiveness, e.g., alerts	No temporal grouping (e.g., trends)
Efficient for simple event-driven logic	No built-in tolerance for out-of-order tuples

Distributed DSP systems: processing models

- Windowed: pros and cons

Supports complex analytics (e.g., average, trends)	Higher latency: waits for window to fire
Stateful processing: retains data over time/counts	More memory-consuming
Temporal awareness: enables time-sensitive logic	Increased complexity: requires managing window logic, including trigger and eviction
Handles out-of-order tuples (in some systems)	

Distributed DSP systems: evolution

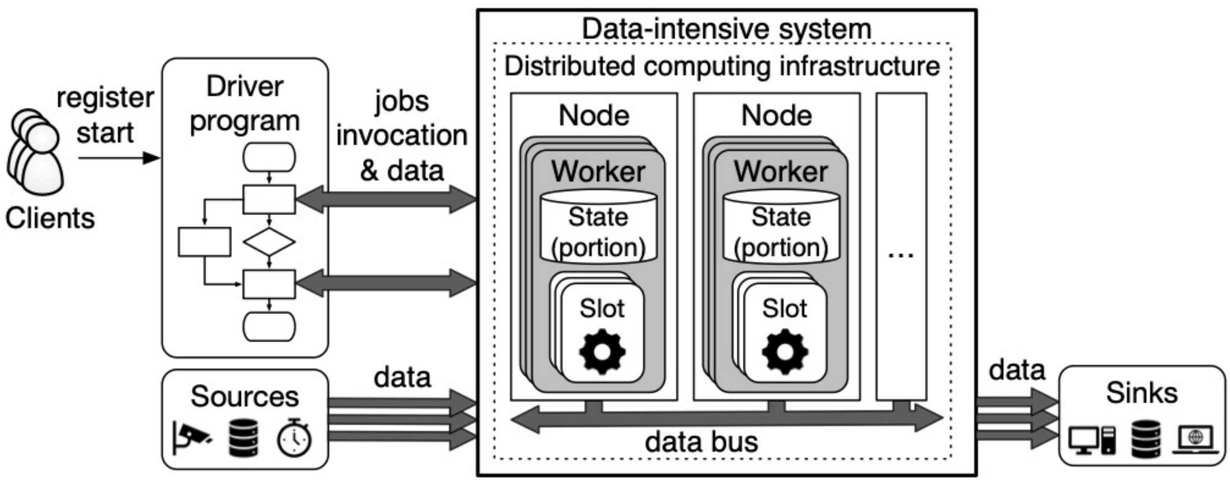


- Early systems were designed as extensions of relational execution engines plus windows
- Modern systems have evolved considering completeness and ordering (e.g., out-of-order computation) and witnessed architectural paradigm shifts (e.g., processing guarantees, reconfiguration and state management)
- Recent shift towards general event-driven architectures, actor-like programming models and microservices, and growing use of hw accelerators

Fragkoulis et al., A Survey on the Evolution of Stream Processing Systems, 2024

Data-intensive systems: a common view

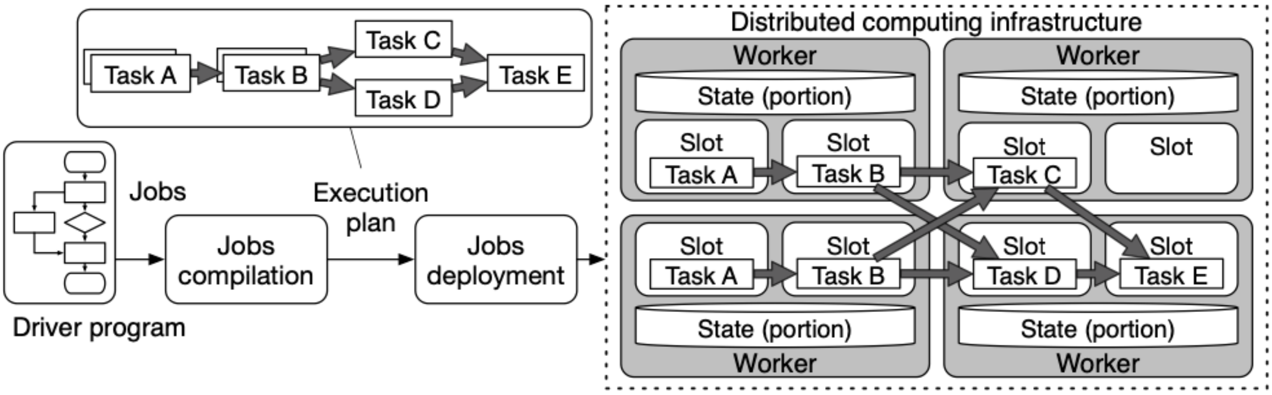
- Distributed data-intensive systems for batch and stream processing share some common characteristics in terms of architecture



Margara et al., A Model and Survey of Distributed Data-Intensive Systems, 2023

Data-intensive systems: a common view

- Applications (i.e., jobs) and their lifecycle
  - Job lifecycle includes: definition using API, compilation into an *execution plan*, deployment, and execution
  - Jobs are compiled into elementary units of execution (i.e., tasks) and run on slots offered by worker nodes
  - Each task can be replicated (data parallelism)
  - Tasks must be deployed onto the slots of the underlying infrastructure through a placement algorithm



References

- Akidau, Streaming 101: The world beyond batch, 2015 https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101
- Kleppman, Designing Data-Intensive Applications, chapter 11
- Margara et al., A model and survey of distributed data-intensive systems, *ACM Comp. Surv.*, 2023 https://dl.acm.org/doi/pdf/10.1145/3604801
- Fragkoulis et al., A survey on the evolution of stream processing systems, *VLDB J.*, 2024 https://link.springer.com/article/10.1007/s00778-023-00819-8