CS 591 K1:
Data Stream Processing and Analytics

Spring 2020

1/23: Stream Processing Fundamentals

Vasiliki (Vasia) Kalavri
vkalavri@bu.edu

What is a stream?

- In traditional data processing applications, we know the entire dataset in advance, e.g. tables stored in a database.

A data stream is a data set that is produced incrementally over time, rather than being available in full before its processing begins.

- Data streams are **high-volume**, **real-time data** that might be **unbounded**
  - we cannot store the entire stream in an accessible way
  - we have to process stream elements on-the-fly using limited memory

Properties of data streams

- They **arrive continuously** instead of being available a-priori.
- They bear an *arrival* and/or a *generation* **timestamp**.
- They are produced by external sources, i.e. the DSMS has **no control** over their **arrival order** or the **data rate**.
- They have **unknown**, possibly **unbounded length**, i.e. the DSMS does not know when the stream *ends*.

Data Management Approaches

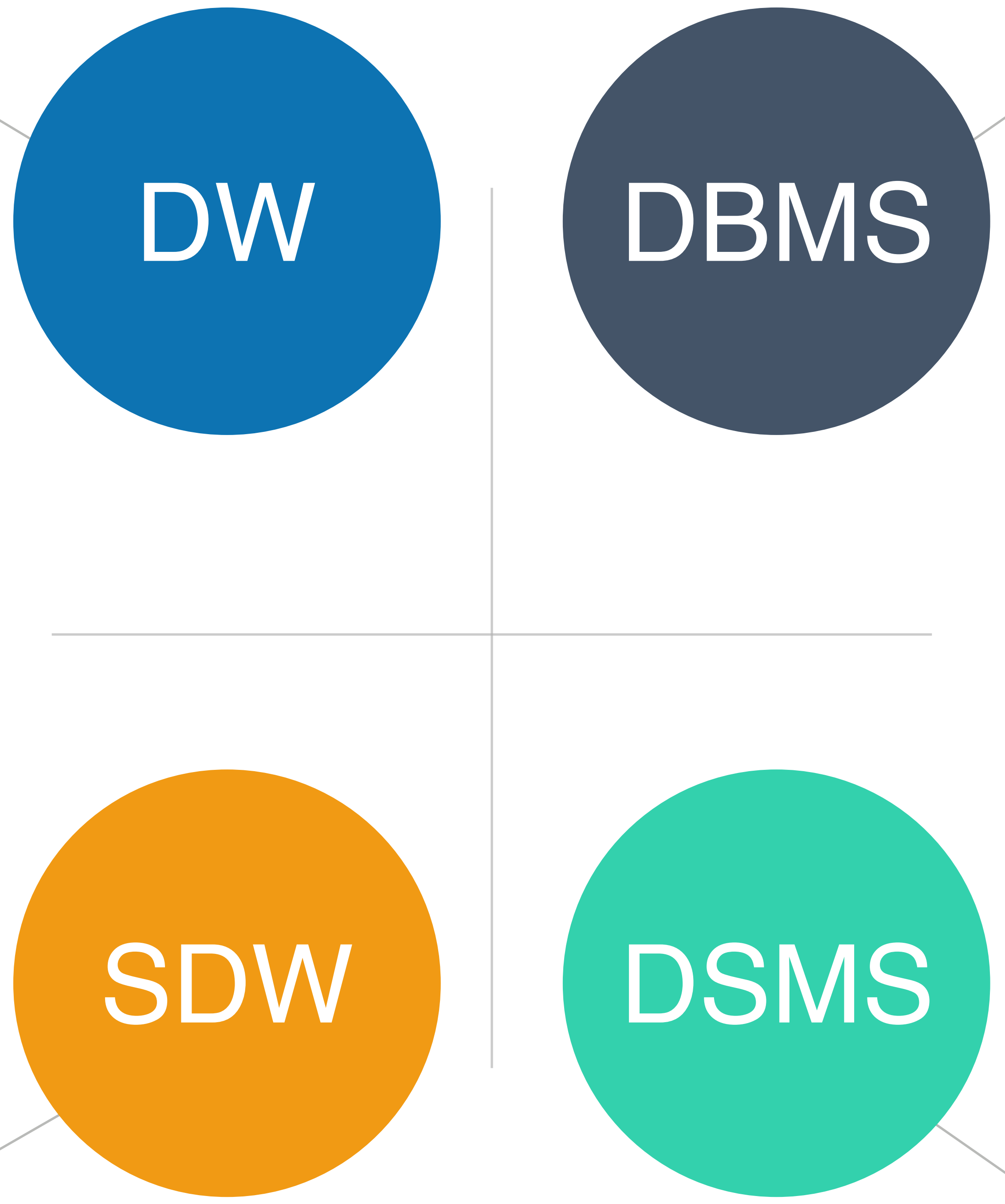
static data

Data Warehouse

- complex, offline analysis
- large and relatively static and historical data
- batched updates during downtimes, e.g. every night

Database Management System

- ad-hoc queries, data manipulation tasks
- insertions, updates, deletions of single row or groups of rows



storage

analytics

Streaming Data Warehouse

- low-latency materialized view updates
- pre-aggregated, pre-processed streams and historical data

Data Stream Management System

- continuous queries
- sequential data access, high-rate append-only updates

streaming data

DBMS vs. DSMS

	DBMS	DSMS
Data	persistent relations	streams
Data Access	random	sequential, single-pass
Updates	arbitrary	append-only
Update rates	relatively low	high, bursty
Processing Model	query-driven / pull-based	data-driven / push-based
Queries	ad-hoc	continuous
Latency	relatively high	low

Traditional DW vs. SDW

	Traditional DW	SDW
Update Frequency	low	high
Update propagation	synchronized	asynchronous
Data	historical	recent and historical
ETL process	complex	fast and light-weight

ETL: Extract-Transform-Load

e.g. unzipping compressed files, data cleaning and standardization

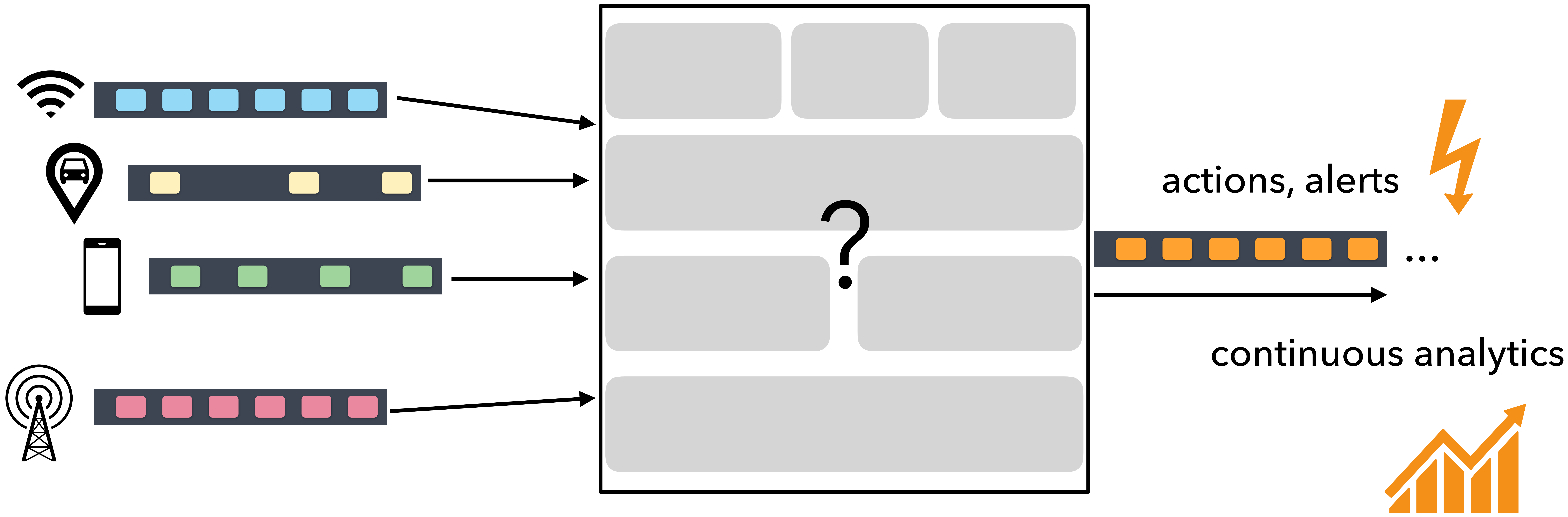
2005

The 8 Requirements of Real-Time Stream Processing

Michael Stonebraker   Uğur Çetintemel   Stan Zdonik

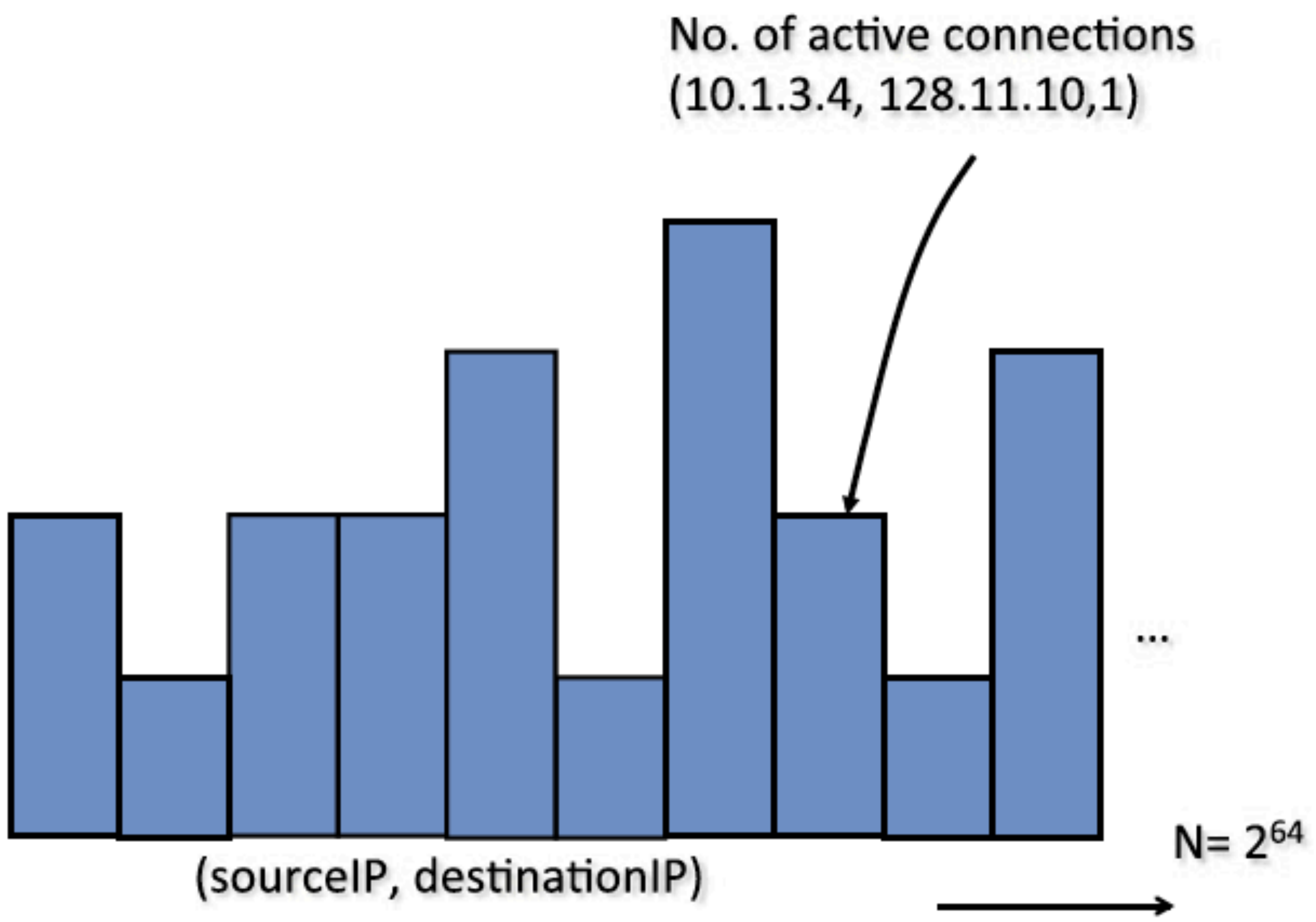
1. Process events *online* without storing them
2. Support a high-level language (e.g. StreamSQL)
3. Handle missing, out-of-order, delayed data
4. Guarantee deterministic (on replay) and correct results (on recovery)
5. Combine batch (historical) and stream processing
6. Ensure availability despite failures
7. Support distribution and automatic elasticity
8. Offer low-latency

Building a stream processor…



Basic Stream Models

A stream can be viewed as a massive, dynamic, one-dimensional *vector* ***A***[1…N].



up-to-date frequencies for specific (source, destination) pairs observed in IP connections that are currently active

The size N of the streaming vector is defined as the product of the attribute domain size(s).
Note that N might be unknown.

The vector is updated by a continuous stream events where the j_{th} update has the general form (k, c[j]) and modifies the k_{th} entry of ***A*** with the operation ***A***[k]←***A***[k] + c[j].

Time-Series Model: The j_{th} update is (j, ***A***[j]) and updates arrive in increasing order of j, i.e. we observe the entries of ***A*** by increasing index.

This can model time-series data streams:

- a sequence of measurements from a temperature sensor
- the volume of NASDAQ stock trades over time

This model poses a severe limitation on the stream: updates cannot change past entries in ***A***.

Useful in theory for the development of streaming algorithms
With limited practical value in distributed, real-world settings

Cash-Register Model: In this model, multiple updates can *increment* an entry ***A***[j]: In the j_{th} update (k, c[j]), it must hold that c[j] ≥ 0.

This can model insertion-only streams:

- monitoring the total packets exchanged between two IP addresses
- the collection of IP addresses accessing a web server

With some practical value for use-cases with append-only data
It preserves all history without the option to discard old events

Turnstile Model: The j_{th} update (k, c[j]), can be either positive or negative. Events can be continuously inserted and deleted from the stream.

It can model fully dynamic situations:

- Monitoring active IP network connections is a Turnstile stream, as connections can be initiated or terminated between any pair of addresses at any point in the stream.

It is the most general model

Hard to develop space-efficient and time-efficient algorithms

Relational Streaming Model

Streams as evolving relations

- A **stream** is interpreted as describing a changing relation.
- Stream elements bear a **valid timestamp**, V_s, after which they are considered valid and they can contribute to the result.
  - alternatively, events can have **validity intervals**.
- The contents of the relation at time t are all events with $V_s \leq t$.

Types of streams

- **Base stream**: produced by an external source
  - e.g. TCP packet stream

```
<timestamp, src_IP, src_port, dest_IP, dest_port, size>
```

packet generation time

bytes in packet

- **Derived stream**: produced by a continuous query and its operators, e.g. *total traffic from a source every minute*

```
<minute, src_IP, SUM(size)>
```

minute start or end

total bytes this minute

new events

<t_k, 15.1.2.5, 11.4.5.5, 64K>

⋮

<t_3, 16.2.3.7, 11.8.6.2, 28K>

<t_2, 13.5.6.7, 12.4.0.3, 32K>

<t_1, 16.2.3.7, 10.1.0.2, 20K>

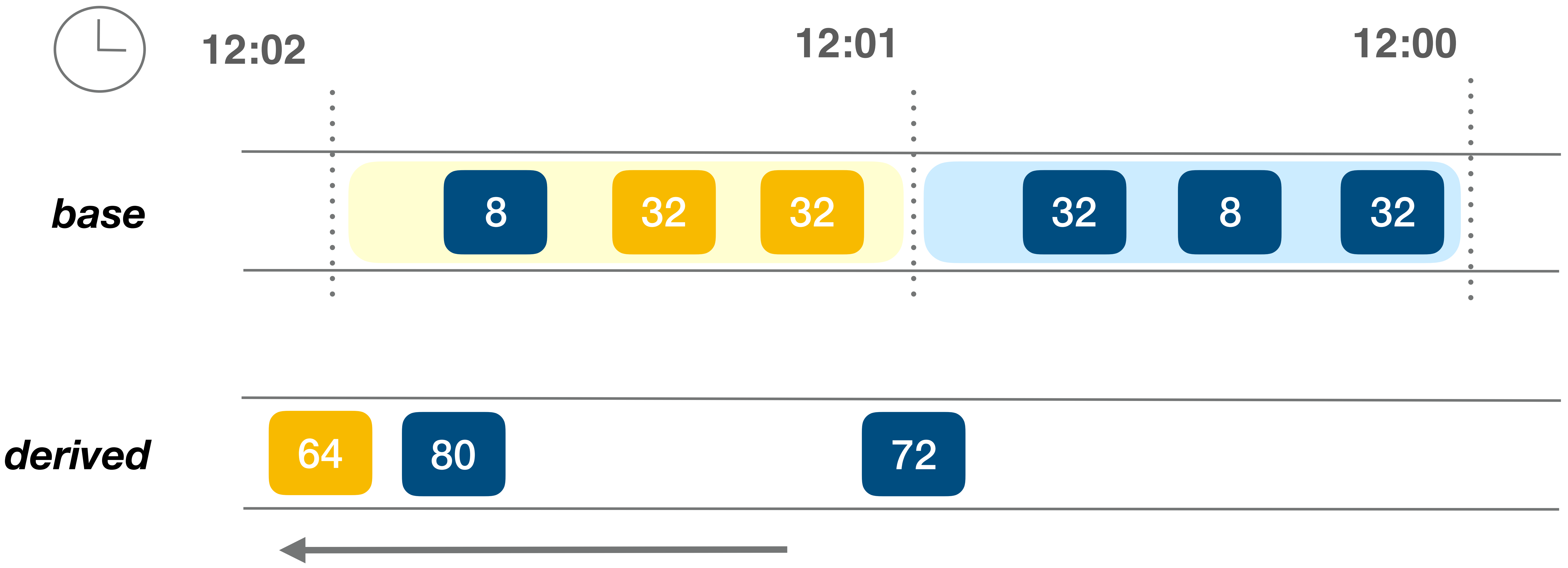
old events

append

	src_IP	dest_IP	bytes
$R(t_1)$	16.2.3.7	10.1.0.2	20K
$R(t_2)$	13.5.6.7	12.4.0.3	32K
$R(t_3)$	16.2.3.7	11.8.6.2	28K
	⋮	⋮	⋮
$R(t_k)$	15.1.2.5	11.4.5.5	64K

- Base streams are typically **append-only**
  - previously arrived items are not modified
- Derived streams **may not be** append-only
  - what if packets arrive late?
  - we might need to revise the computed total traffic, i.e. output stream might contain updates to previously emitted items

Which basic models do base and derived streams correspond to?



Results as continuously updated materialized views

- Base streams update relation tables and derived streams update materialized views.
- An **operator** outputs event streams that describe the *changing view* computed over the input stream according to the relational semantics of the operator.

src	dest	bytes
1	2	20K

sum

src	dest	total
1	2	20K

Results as continuously updated materialized views

- Base streams update relation tables and derived streams update materialized views.
- An **operator** outputs event streams that describe the *changing view* computed over the input stream according to the relational semantics of the operator.

src	dest	bytes
1	2	20K
2	5	32K

sum

src	dest	total
1	2	20K
2	5	32K

Results as continuously updated materialized views

- Base streams update relation tables and derived streams update materialized views.
- An **operator** outputs event streams that describe the *changing view* computed over the input stream according to the relational semantics of the operator.

src	dest	bytes
1	2	20K
2	5	32K
1	2	28K

sum

src	dest	total
1	2	**48K**
2	5	32K

Results as continuously updated materialized views

- Base streams update relation tables and derived streams update materialized views.
- An **operator** outputs event streams that describe the *changing view* computed over the input stream according to the relational semantics of the operator.

src	dest	bytes
1	2	20K
2	5	32K
1	2	28K
2	3	32K

sum

src	dest	total
1	2	48K
2	5	32K
2	3	32K

Results as continuously updated materialized views

- Base streams update relation tables and derived streams update materialized views.
- An **operator** outputs event streams that describe the *changing view* computed over the input stream according to the relational semantics of the operator.

src	dest	bytes
1	2	20K
2	5	32K
1	2	28K
2	3	32K
2	5	64K

sum

src	dest	total
1	2	48K
2	5	**96K**
2	3	32K

Stream representation matters

Consider streams of sensor readings from a temperature probe

How would you compute the average temperature over all sensors if the probe emits:

1. a reading of the current temperature every 1s?
2. the difference from the previous reading every 1s?
3. a reading of the current temperature only if it differs significantly from the last emitted reading?

Stream denotation

An abstract **interpretation** of the stream as a mathematical structure, e.g.

a *sequence* of (finite) relation states over a common schema R: [r_1(R), r_2(R), ...,], where the individual relations are unordered sets.

src	dest	bytes
1	2	20K
2	5	32K
1	2	28K

{(1, 2, 20K), (2, 5, 32K), (1, 2, 28K)}

Such a relation sequence could be **represented** in various ways:

- as the **concatenation** of serializations of the relations.
- as a list of **tuple-index pairs**, where <t, j> indicates that $t \in r_j$
- as a serialization of r_1 followed by a series of **delta tuples** that indicate updates to make to obtain r_2, r_3, ..., etc.
- as a **replacement sequence** where some attribute A denotes a key and an arriving tuple t replaces any existing tuple with the same t(A) value to form a new relation state.
- as a **sliding window** with length k in which each subsequence of k tuples represents a relation state in the sequence.

R1

src	dest	bytes
1	2	20K
2	5	32K

R2

src	dest	bytes
1	2	20K
2	5	32K
2	3	28K

R3

src	dest	bytes
2	5	32K
2	3	28K
1	2	28K

- **concatenation**

(1, 2, 20K), (2, 5, 32K) EOR (1, 2, 20K), (2, 5, 32K), (2, 3, 28K) EOR (2, 5, 32K), (2, 3, 28K), (1, 2, 28K) EOR

- **tuple-index pairs**

<(1, 2, 20K), 1>, <(2, 5, 32K), 2>, <(1, 2, 20K), 2>, <(2, 5, 32K), 1>, <(2, 3, 28K), 3>, <(2, 5, 32K), 3>, <(2, 3, 28K), 2>, <(1, 2, 28K), 3>, …

- **delta tuples**

+(1, 2, 20K), +(2, 5, 32K) EOR +(2, 3, 28K) EOR -(1, 2, 20K), +(1, 2, 28K) EOR

What are the advantages and disadvantages of each representation?

Reconstitution functions

Insert (append-only): The reconstitution function **ins** starts with an empty bag and then inserts each successive stream item:

- ins([]) = Ø
- ins(P:i) = insert(i, ins(P)), where P:i denotes the sequence P extended by item i.

Insert-Unique (distinct): The reconstitution function **ins_u** checks for duplicates:

- ins_u([]) = Ø
- ins_u(P:i) = if i $\notin$ ins_u(P) then insert(i, ins_u(P)) else ins_u(P).

Insert-Replace: If the stream has a *key*, the reconstitution function **ins_r** guarantees that only the most recent item with a given key is included:

- ins_r([]) = Ø
- ins_r(P:i) = insert(i, {j | j $\in$ ins_r(P) ^ j.A ≠ i.A}).

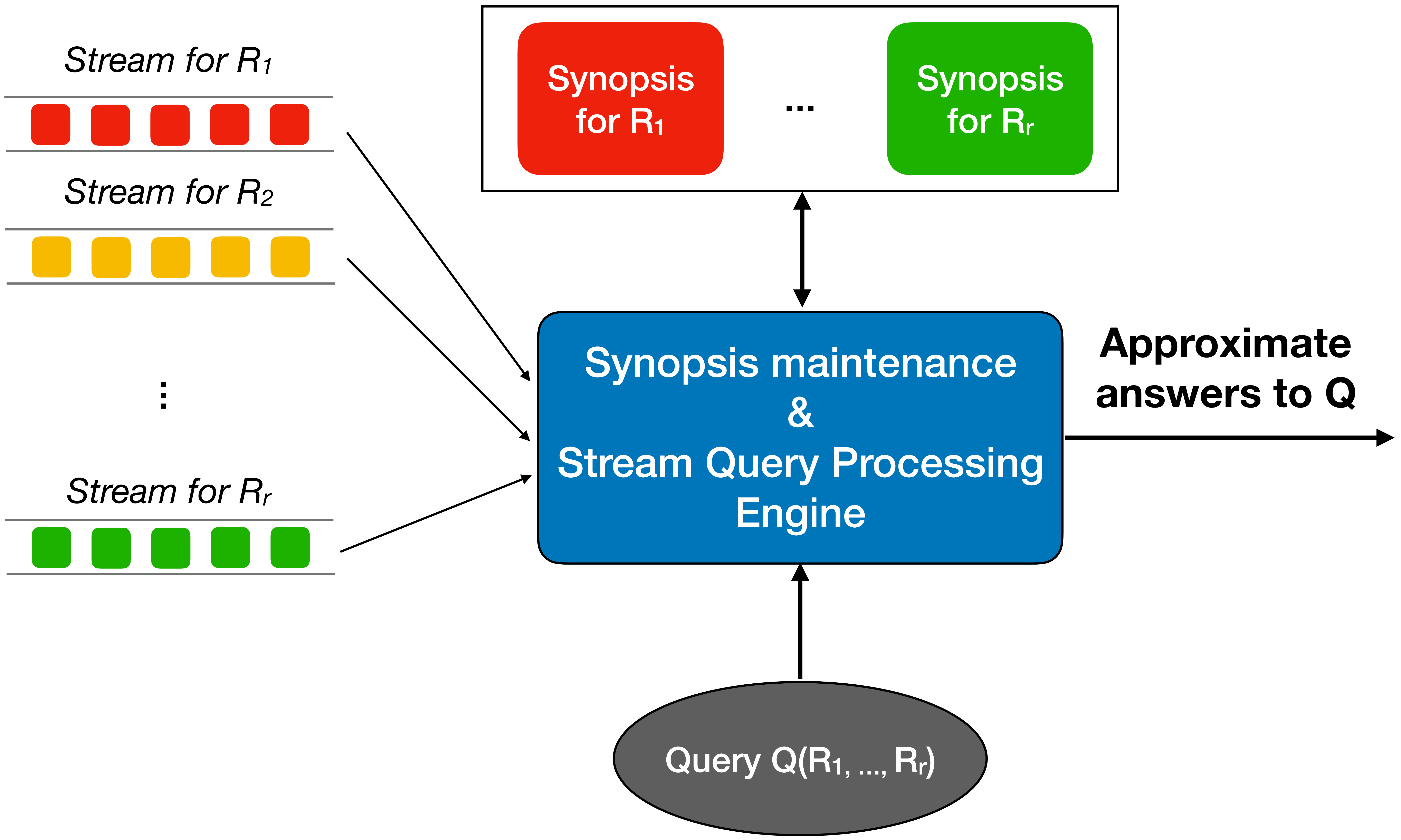
Query processing challenges

- **Memory requirements:** we cannot store the whole stream history.
- **Data rate:** we cannot afford to continuously update indexes and materialized views for high rates.
- **Incremental computation**: do we recompute the result from scratch whenever a new record is appended to the stream table?

Synopses: Maintain *summaries* of streaming data instead of the complete history.

Stream synopses requirements

- **Single-pass**: synopses can be easily updated with a single pass over streaming tuples in their arrival order
- **Small space**: memory footprint poly-logarithmic in the stream size
- **Low time**: fast update and query times
- **Delete-proof**: synopses can handle both insertions and deletions in an update stream
- **Composable**: synopses can be built independently on different parts of the stream and composed/merged to obtain the synopsis of the whole stream



Stream for R_1
Stream for R_2
⋮
Stream for R_r
Synopsis for R_1
...
Synopsis for R_r
Synopsis maintenance & Stream Query Processing Engine
Approximate answers to Q
Query $Q(R_1, ..., R_r)$

What synopsis would you use to compute:

- The average of a stream on integers?
- The number of distinct users who have visited a website?
- The top-10 queries inserted in a search engine?
- The connected components of accounts in a stream of financial transactions?

Issues with synopses

- They are *lossy* compressions of streams
    - trade-off memory footprint for accuracy
- Query results are approximate with either deterministic or probabilistic error bounds
- There is no *universal* synopsis solution
- They are purpose-built and query-specific
    - different synopsis to count distinct elements than to keep track of top-K events

Dataflow Streaming Model

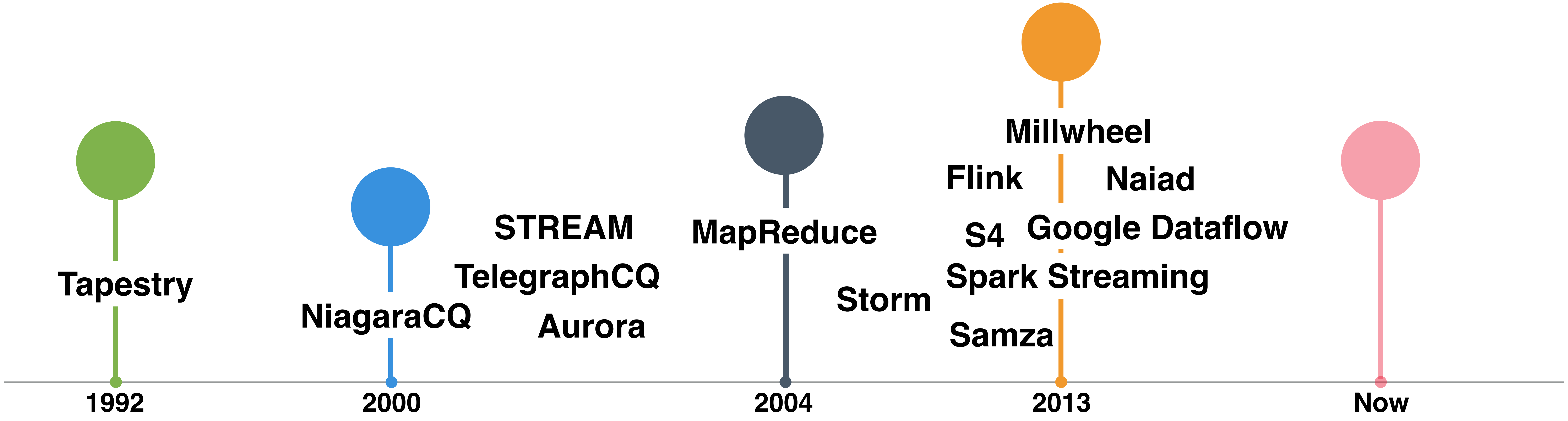
Evolution of Stream Processing

Stream Database Systems

Single-node execution
Synopses and sketches
Approximate results
In-order data processing

Dataflow Systems

Distributed execution
Partitioned state
Exact results
Out-of-order support



Vasiliki Kalavri | Boston University 2020

Distributed dataflow systems

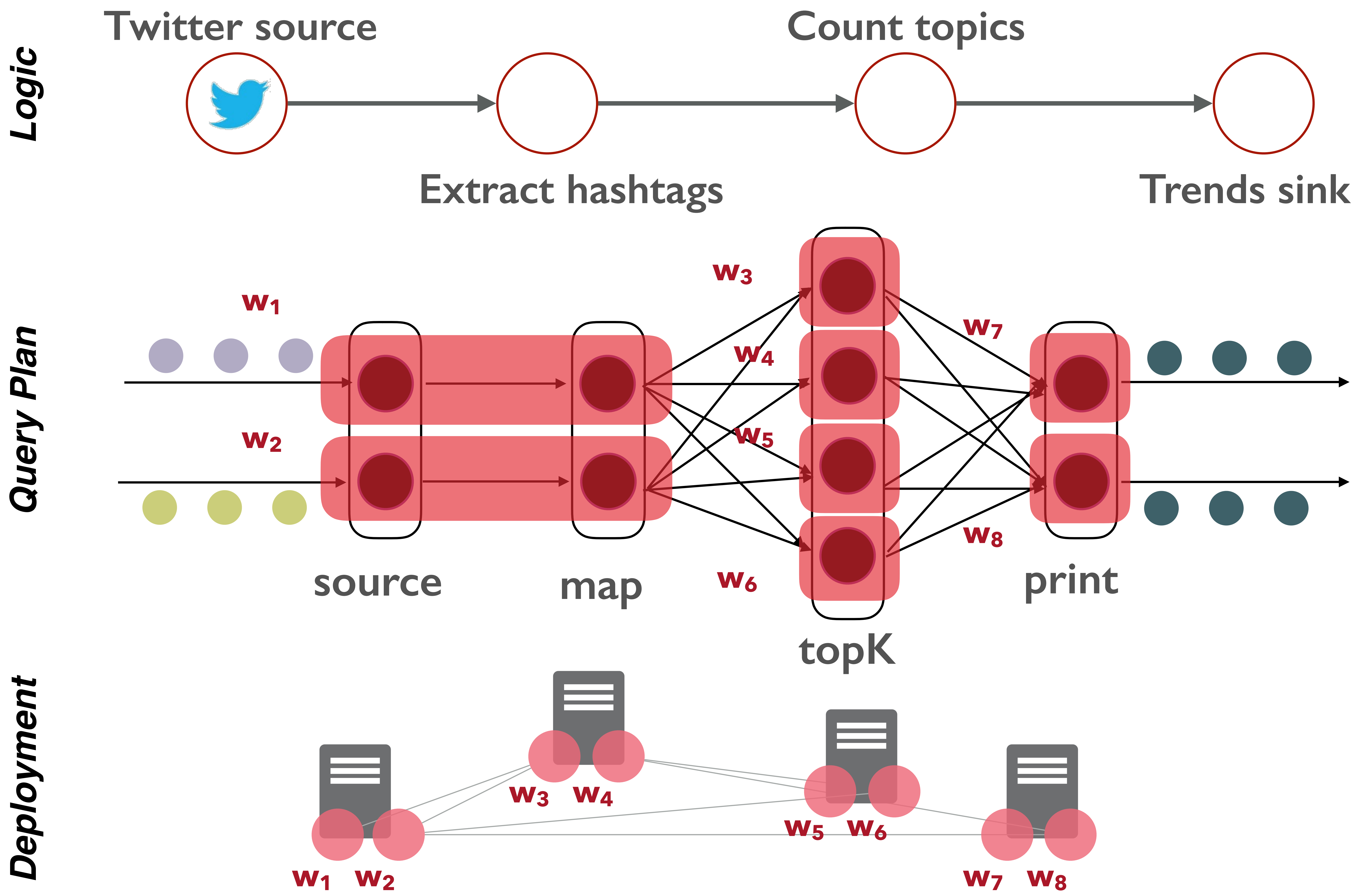
- Computations as **Directed Acyclic Graphs** (DAGs)
  - nodes are operators and edges are data channels
  - operators can accumulate state, have multiple inputs, express event-time custom window-based logic
  - some systems, like Timely Dataflow support **cyclic dataflows** and **iterations** on streams
- Operators are **data-parallel**
  - distributed workers (threads) execute one parallel instance of one of more operators on *disjoint data partitions*

Distributed dataflow model

- Exploit **data parallelism** and **shared-nothing** architectures to scale stream processing to high-volume streams and large state
    - Streams do not correspond to states one-on-one, i.e. state can be the result of one or more base and/or derived streams
    - Each query (operator) maintains its own state
    - Queries process raw streams, not synopses => results are typically exact
- **Challenges**: computation progress, fault-tolerance and result guarantees, automatic scaling and state migration, out-of-order processing

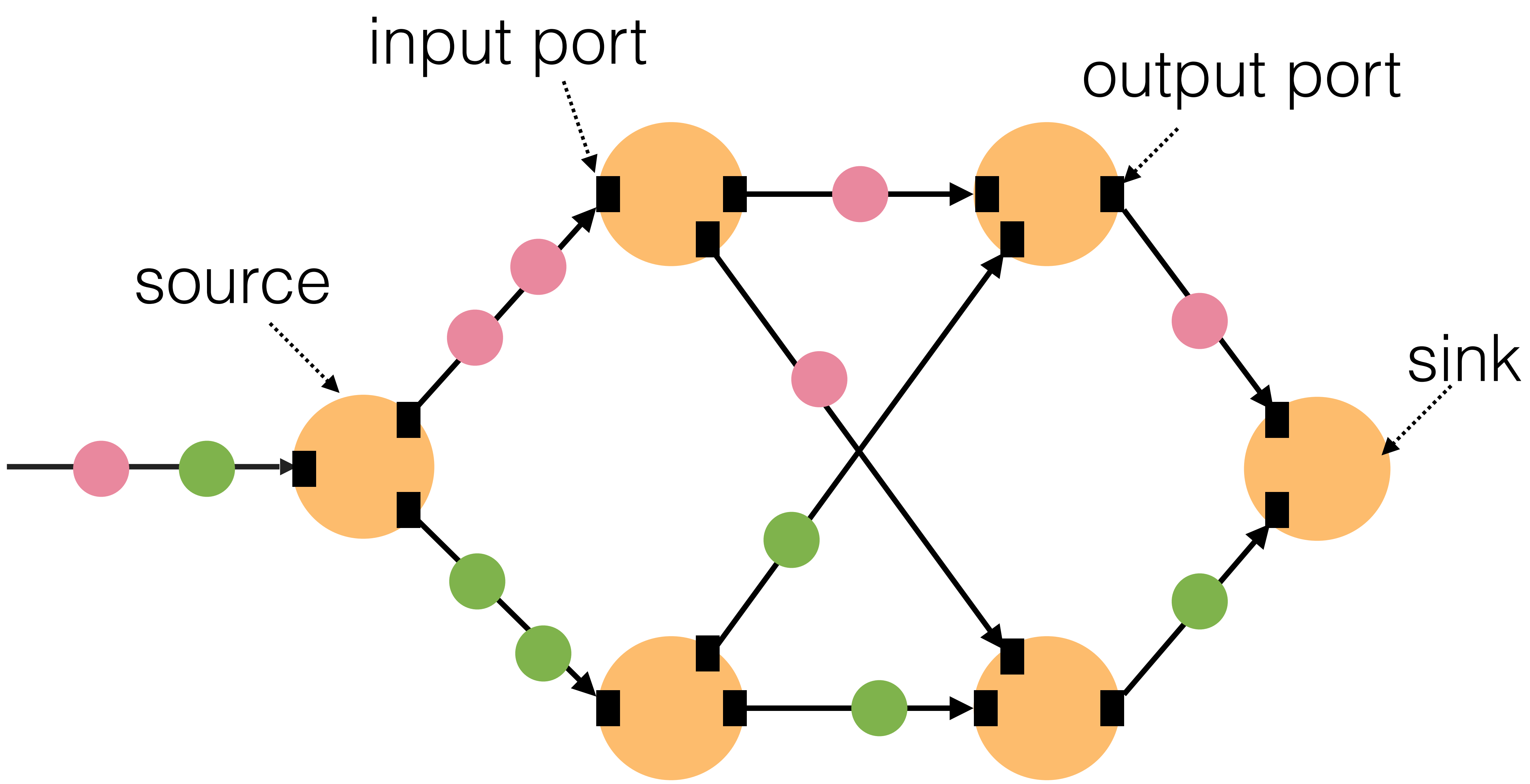
Distributed dataflow model

- No particular basic stream model (time-series, turnstile…) is imposed by the dataflow execution engine.
- The burden of representation and denotations if left to the application developer/user.
- The programmer needs to design and maintain appropriate state synopses.
- In order to parallelize operations, events must have associated keys.


Logic
Twitter source
Count topics
Extract hashtags
Trends sink
Query Plan
w1
w2
w3
w4
w5
w6
w7
w8
source
map
topK
print
Deployment
w3 w4
w5 w6
w1 w2
w7 w8

A series of transformations
on streams in
Stream SQL, Scala, Python,
Rust, Java…

dataflow graph



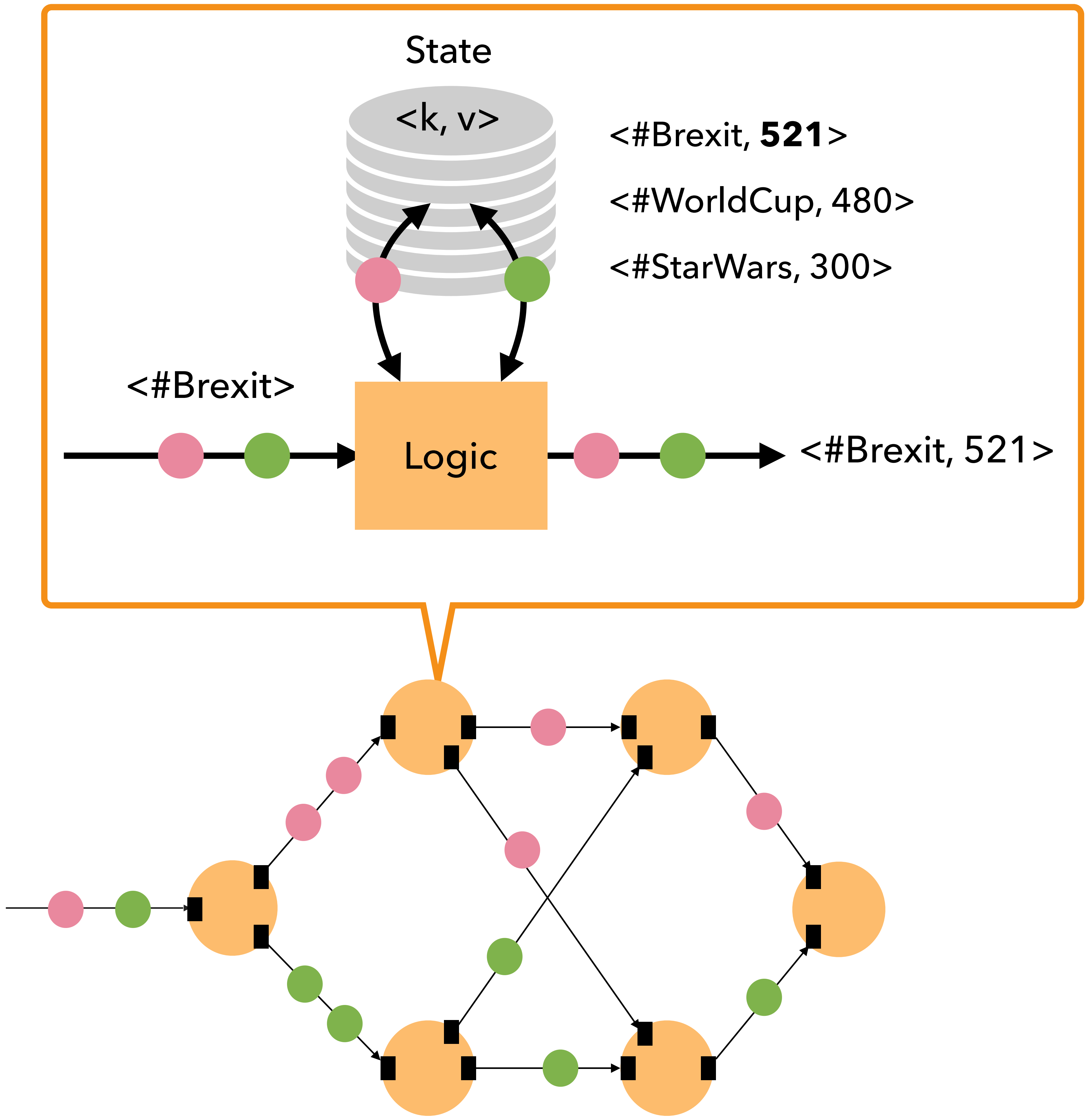
Dataflow graph

- operators are nodes, data channels are edges
- channels have FIFO semantics
- streams of data elements flow continuously along edges

Operators

- receive one or more input streams
- perform tuple-at-a-time, window, logic, pattern matching transformations
- output one or more streams of possibly different type

Stateful operators



- Stateful operators **maintain state** that reflect part of the stream history they have seen
    - windows, continuous aggregations, distinct…
- State is commonly **partitioned by key**
- State can be cleared based on watermarks or punctuations
    - window fires, post becomes inactive

Example: Apache Flink DataStream API

```
case class Reading(id: String, time: Long, temp: Double)

object MaxSensorReadings {
  def main(args: Array[String]) {
  val env = StreamExecutionEnvironment.getExecutionEnvironment
    val sensorData = env.addSource(new SensorSource)
    val maxTemp = sensorData
      .map(r => Reading(r.id,r.time,(r.temp-32)*(5.0/9.0)))
      .keyBy(_.id)
      .max("temp")
    maxTemp.print()
    env.execute("Compute max sensor temperature”)
  }
}
```

Relational Streaming vs. Dataflow Streaming

	Relational	Dataflow
Input	in-order	out-of-order
Results	approximate	exact
Language	SQL extensions, CQL	Java, Scala, Python, SQL
Execution	centralized	distributed
Parallelism	pipeline	pipeline, task, data
State	limited, in-memory	partitioned, virtually unlimited, persisted to backends
Load management	shedding	backpressure, elasticity
Fault tolerance	limited support, high availability	full support, exactly-once

Summary

Today you learned:

- stream representations, stream processing models
- streaming applications and use-cases
- different approaches to data management
- the relational streaming model vs. the dataflow streaming model

Lecture references

Some material in this lecture was assembled from the following sources:

- Minos Garofalakis, Johannes Gehrke, and Rajeev Rastogi. **Data Stream Management: Processing High-Speed Data Streams**. Springer-Verlag, Berlin, Heidelberg.

- Lukasz Golab and M. Tamer Özsu. **Issues in data stream management**. SIGMOD Rec. 32, 2 (June 2003).

- David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. **Semantics of data streams and operators**. In Proceedings of the 10th international conference on Database Theory (ICDT'05).

- Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. **The 8 requirements of real-time stream processing**. SIGMOD Rec. 34, 4 (December 2005).