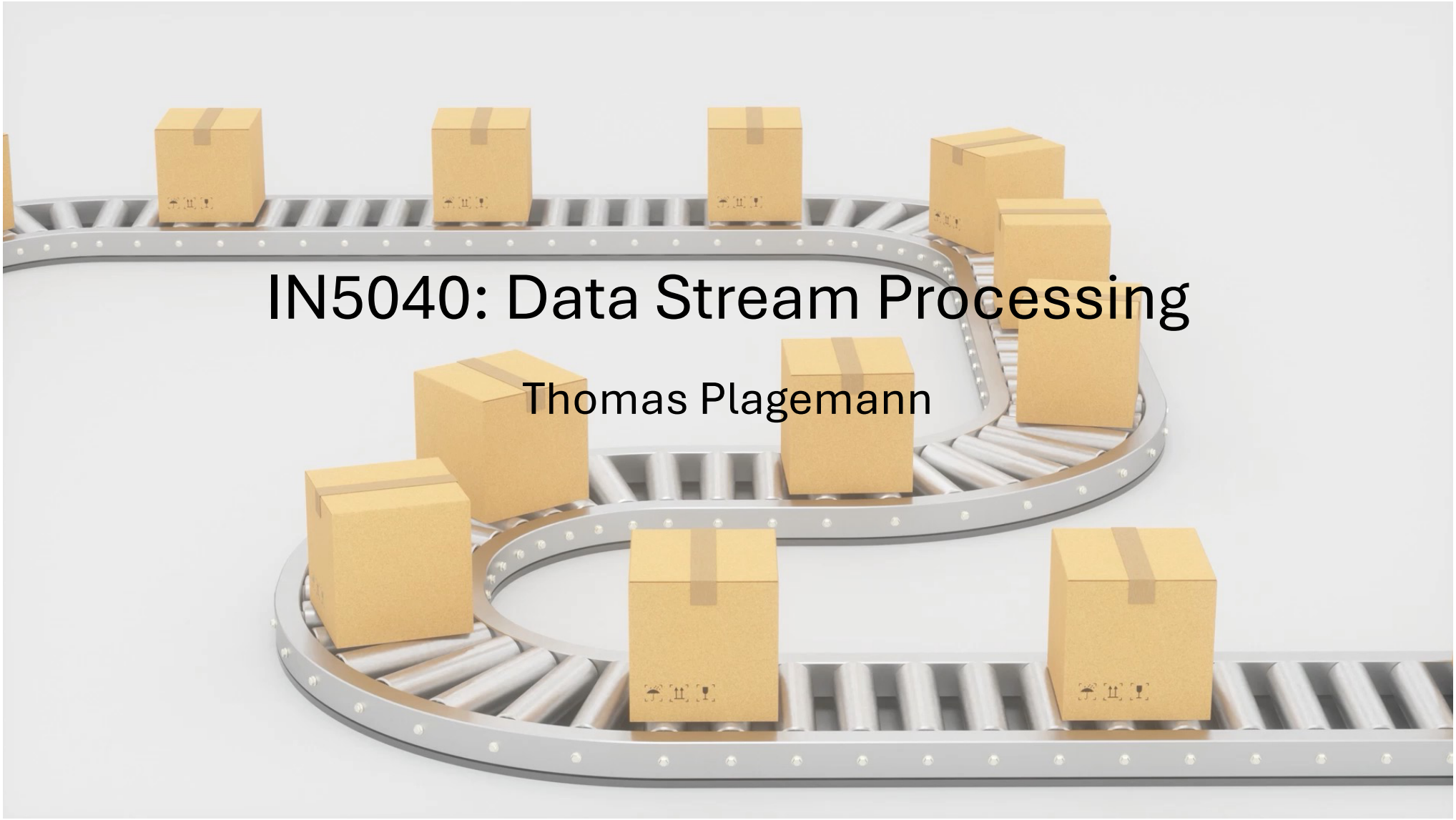
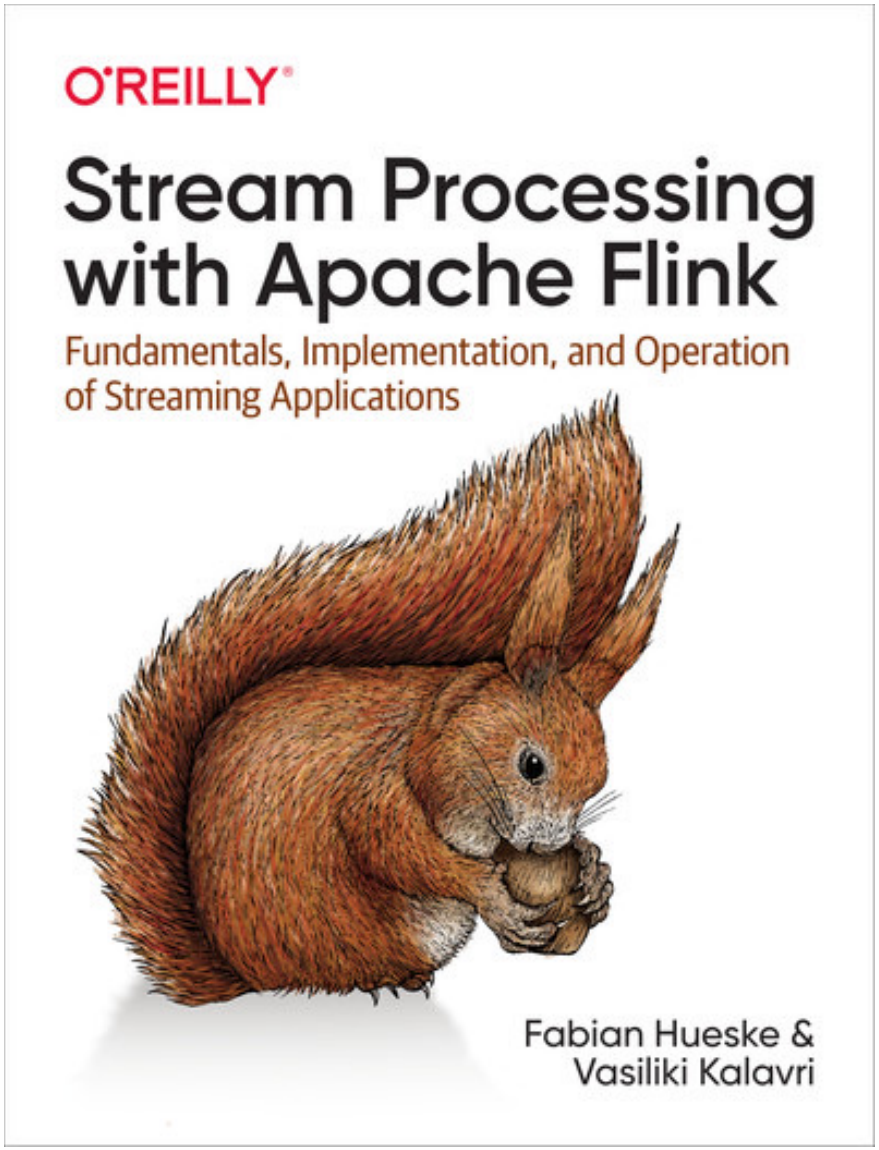


IN5040: Data Stream Processing

Thomas Plagemann

Outline

- Motivation
- Historical development
- Fundamentals
- Windows
- Time
- Queries
- Load shedding
- State management
- Result guarantees
- Checkpoint, savepoint and recovery
- Parrot an insighte into our current research



Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Motivation for DSP

- Large amounts of interesting data:
  - deploy transactional data observation points, e.g.,
    - AT&T long-distance: ~300M call tuples/day
    - AT&T IP backbone: ~10B IP flows/day
  - generate automated, highly detailed measurements
    - NOAA: satellite-based measurement of earth geodetics
    - Sensor networks: huge number of measurement points
- Near real-time queries/analyses
  - ISPs: controlling the service level
  - NOAA: tornado detection using weather radar data

DSP Applications

- Sensor Networks:
  - Monitoring of sensor data from many sources, complex filtering, activation of alarms, aggregation and joins over single or multiple streams
- Network Traffic Analysis:
  - Analyzing Internet traffic in near real-time to compute traffic statistics and detect critical conditions
- Financial Tickers:
  - On-line analysis of stock prices, discover correlations, identify trends
- On-line auctions
- Transaction Log Analysis, e.g., Web, telephone calls, …
- Credit card fraud detection

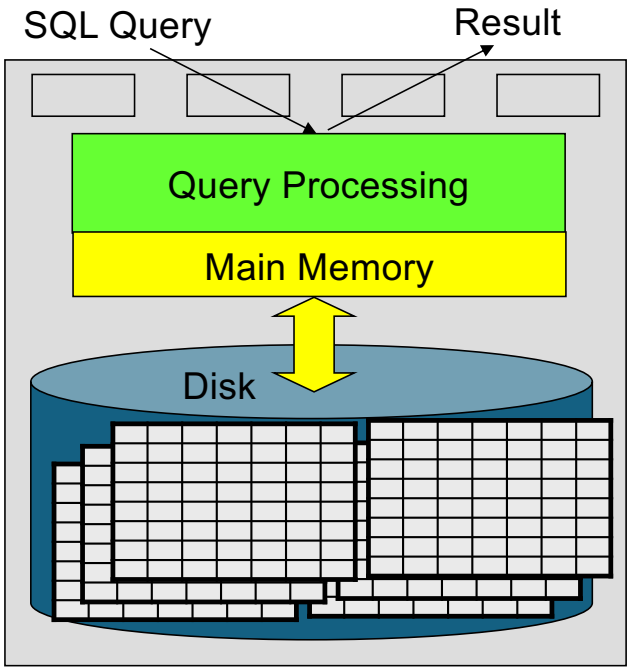
Motivation for DSP (cont.)

- Performance of disks:

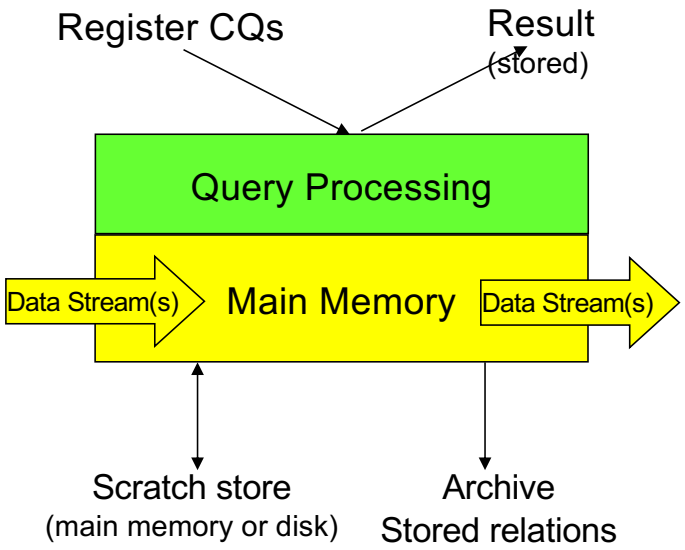
	1987	**2004**	**2016 HD (SSD)**
CPU Performance	1 MIPS	2 Mill. MIPS	1-n Giga/Peta/Exa MIPS
Memory Size	16 Kbytes	32 Gbytes	1-n TBytes
Memory Performance	100 usec	2 nsec	2 nsec
Disc Drive Capacity	20 Mbytes	300 Gbytes	10-16 TBytes
Disc Drive Performance	60 msec	5.3 msec	2-5 msec HD (SSD: 25-250 usec read, 2 msec write)

Handle Data Streams in DBS?

Traditional DBS



DSP



DBS vs DSP

Database Systems (DBS)

- Persistent data (relations)
  (relatively static, stored)
- Transactions (ACID properties)
- One-time queries
- Random access
- "Unbounded" disk store
- Only current state matters
- No real-time services
- Relatively low update rate
- Data at any granularity
- Assume precise data
- Access plan determined by query processor, physical DB design

DSP

- Read-only (append-only) data
- No transaction management
- Transient streams (on-line analysis)
- Continuous queries (CQs)
- Sequential access
- Bounded main memory
- Historical data is important
- Real-time requirements
- Possibly multi-GB arrival rate
- Data at fine granularity
- Data stale/imprecise
- Unpredictable/variable data arrival and characteristics

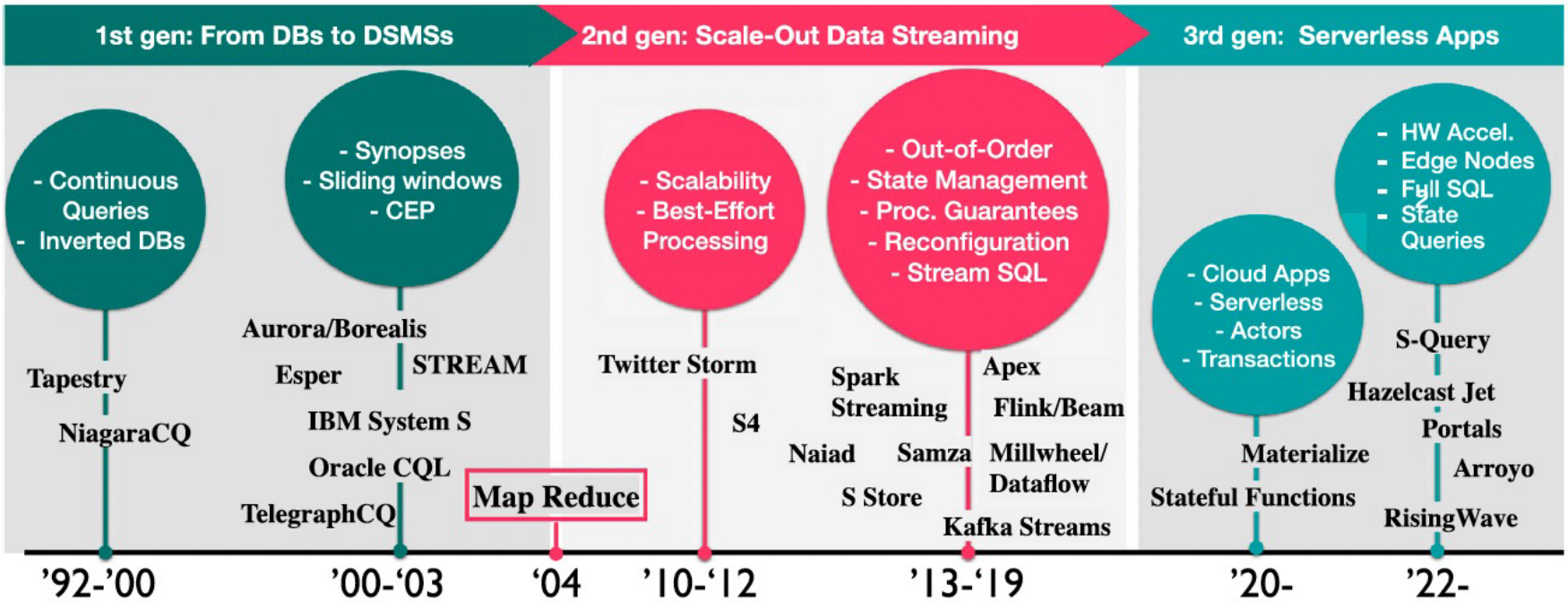
Motivation for DSP (cont.)

- Take-away points:
  - Large amounts of raw data
  - Analysis needed as fast as possible
  - Data feed problem

DSMS versus CEP

- DSMS: Data Stream Management System
- CEP: Complex Event Processing

Data Stream Processing Evolution



[Source: Fragkoulis, M., Carbone, P., Kalavri, V. *et al.* A survey on the evolution of stream processing systems. *The VLDB Journal* **33**, 507–541 (2024). https://doi.org/10.1007/s00778-023-00819-8]

Fundamentals

Fundamentals

- Data stream: Unbounded sequence of tuples/events/records generated continuously over time

  $S = t_1, t_2, t_3, t_4, ...$

- Data tuple: ordered list of attribute-value pairs including a time stamp

  t_1 = (symbol, AAPL), (price, 150.23), (time, 2022-02-14 10:30:00)

Fundamentals

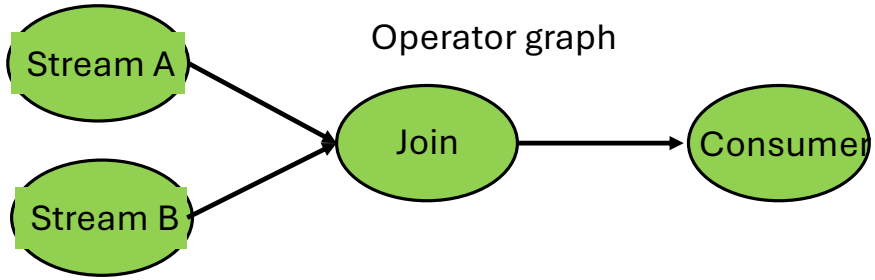
- Latency
  - How long does it take to process a tuple/event?
  - $t_{latency} = t_{output} - t_{input}$
  - Latency in modern systems a few ms
  - Batch processing requires all tupels before processing starts
- Throughput
  - Capacity of the system
  - #events per unit of time

Fundamentals (cont.)

- Query: a function that processes a data stream and returns a result

```
select A.id, B.amount
from Auction A
join Bid B on B.auction = A.id
```

- Operator graph: directed acyclic graph, nodes are operators and vertices are data flows



Fundamentals (cont.)

- Operators:
  - consume data from inputs
  - perform a computation on them
  - produce data to outputs for further processing
- Operators without input port = data source
- Operators without output port = data sink
- Examples:
  - Data source: sensor, stock ticker, credit card transactions, ...
  - Operator: transform, aggregate, join, followed by, ...
  - Data sink: application, data warehouse, ...

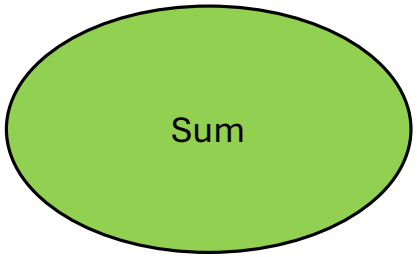
Fundamentals (cont.)

- Non-blocking operators process incoming data as soon as it arrives
- For every individual tuple it creates a result
  - Pass through operators like filter
  - Transform operators, e.g., temperature F -> C

- The easy case ☺

Fundamentals (cont.)

- Blocking operators can only produce results when they received all data tuples



We do not want to wait forever => windows

We need to manage state

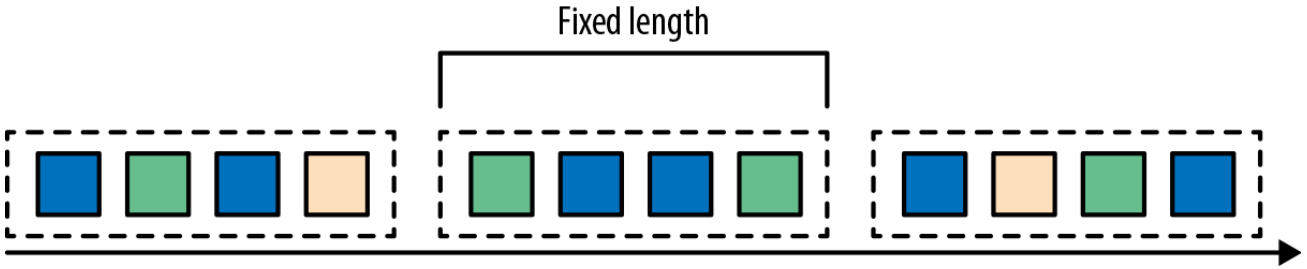
Important issues to understand

- Windows
- Time
- Out-of-order processing
- Queries
- Load shedding
- Processing semantics
- State management
- Fault tolerance

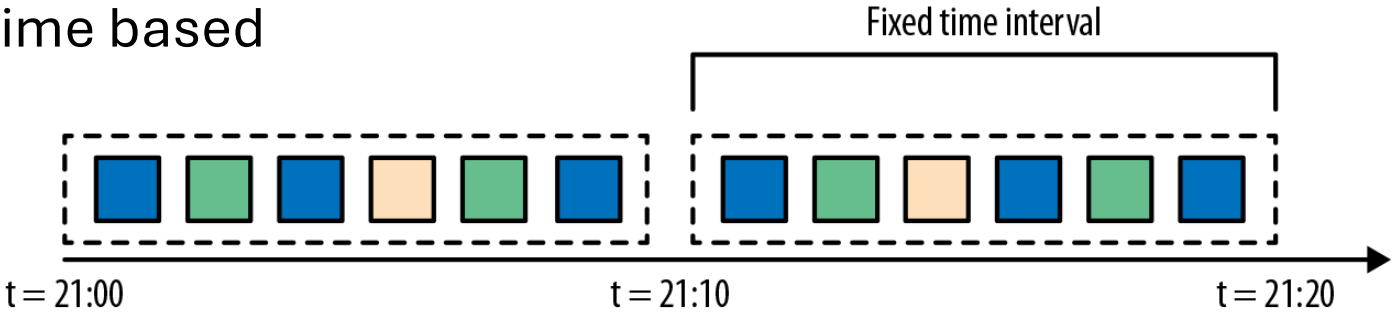
Windows

Tumbling Window

- Count based

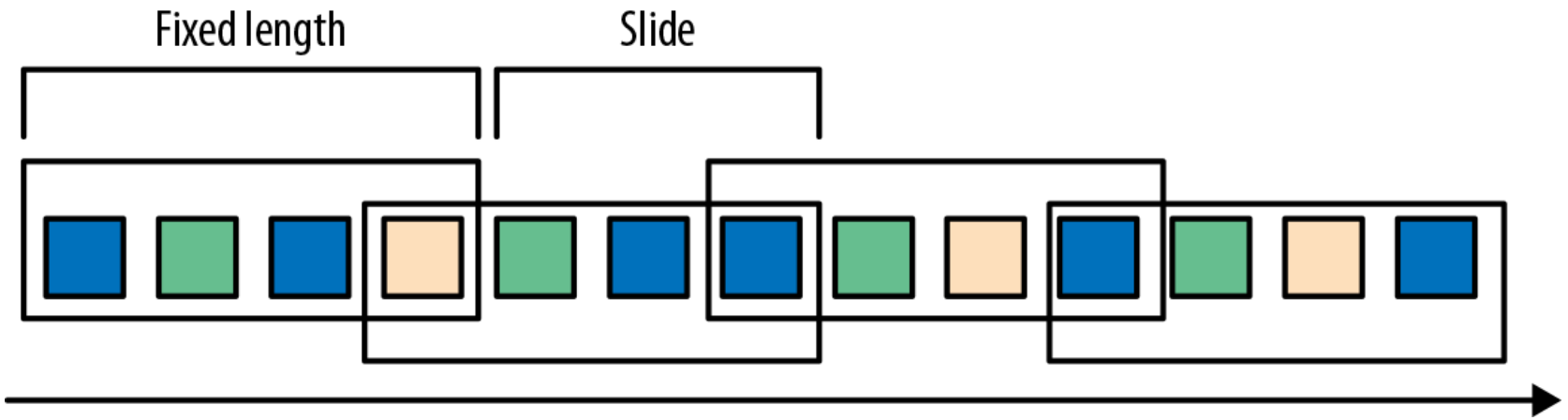


- Time based



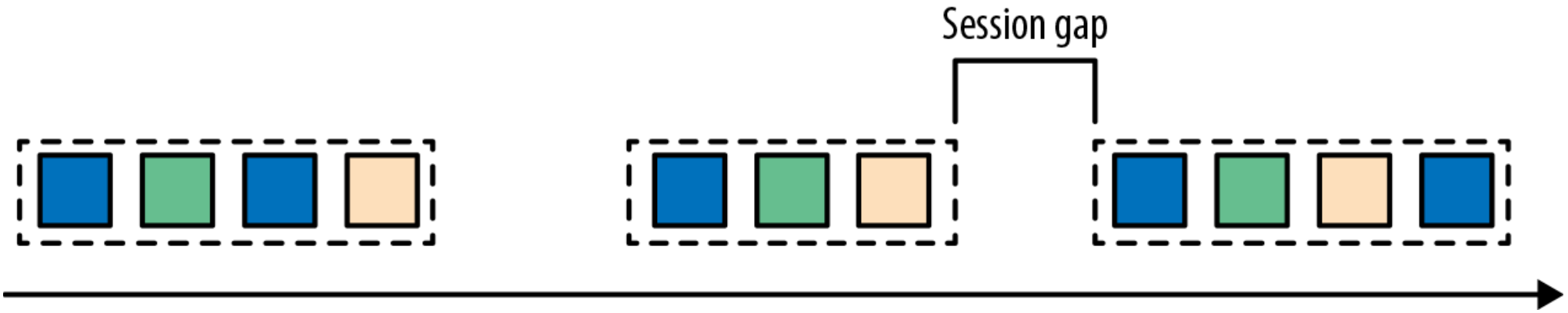
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Sliding Window



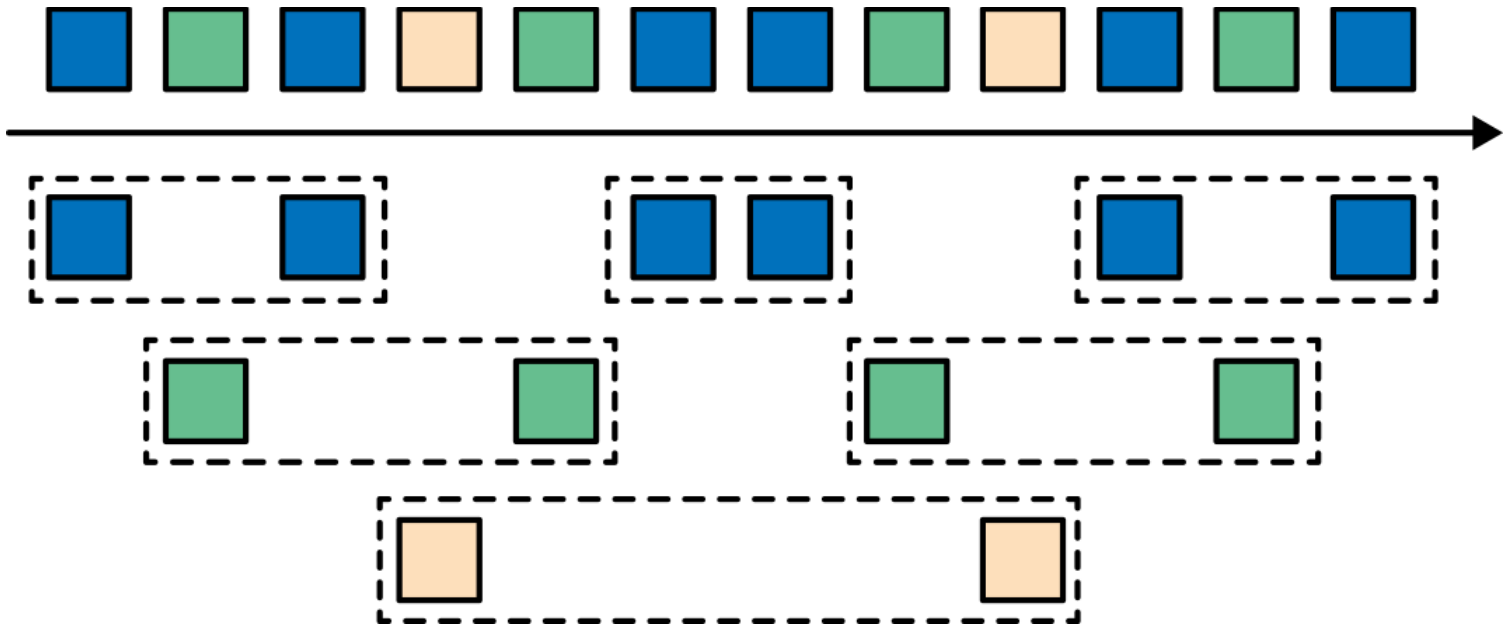
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Session Window



Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

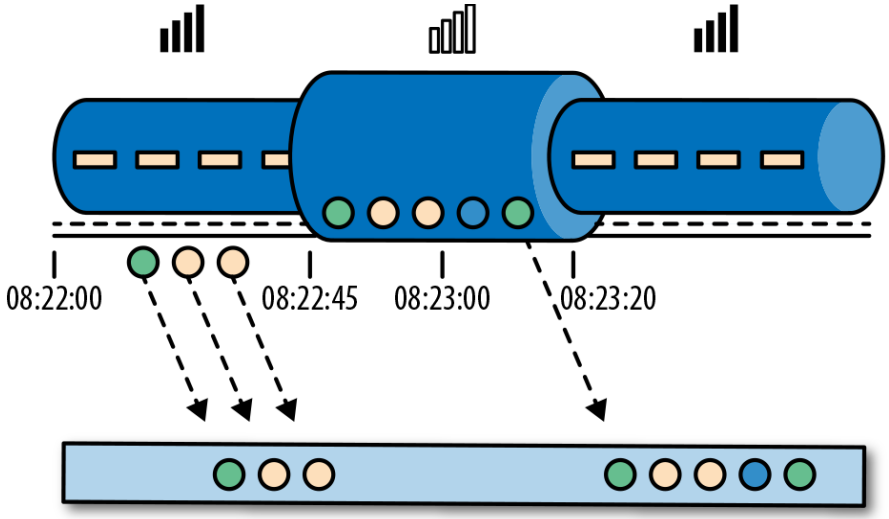
Parallel Windows



Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Time

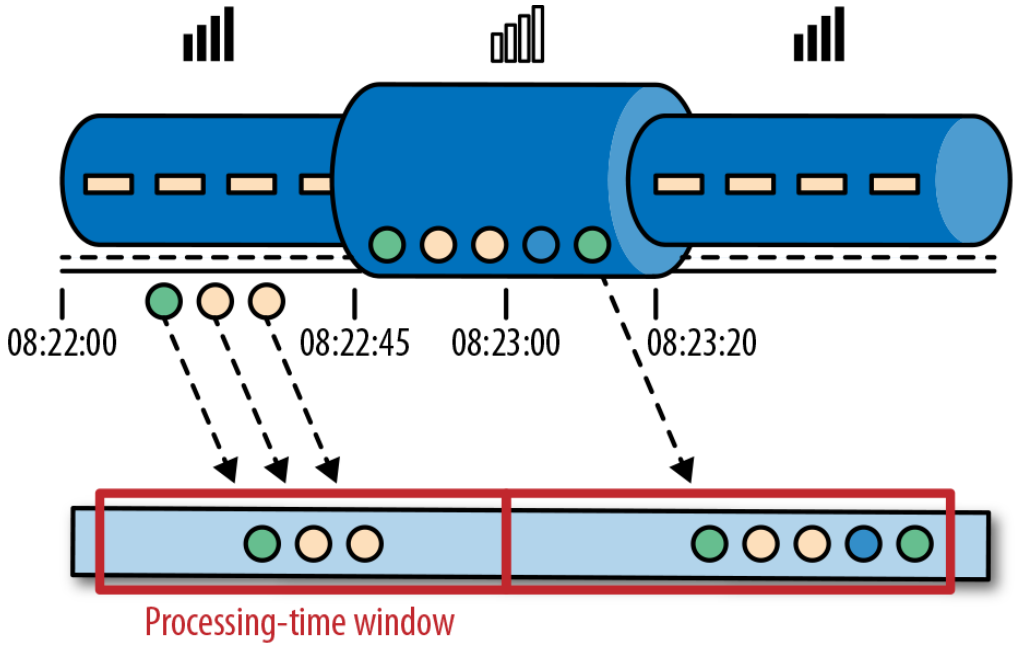
- Example: online mobile game
  - Alice plays an interactive game
  - Enters the sub-way and the Internet connection breaks
  - Game events are stored locally and send when connection is up again



Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Time (cont.)

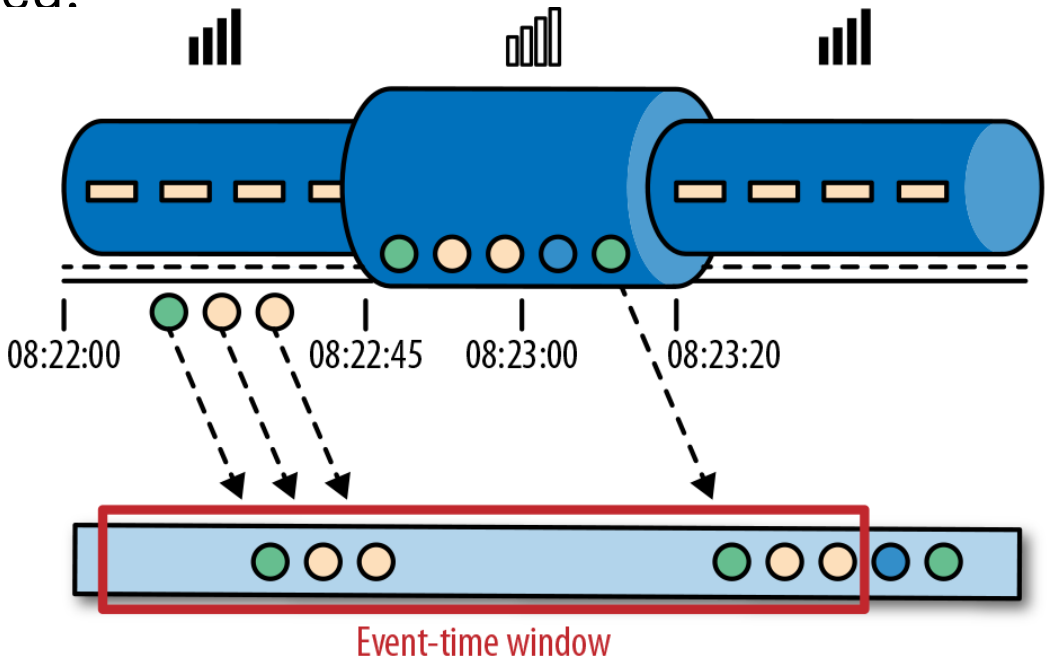
- Processing time is the time of the local clock on the machine where the operator processing the stream is being executed



Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Time (cont.)

- Event time is the time when an event in the stream actually happened.



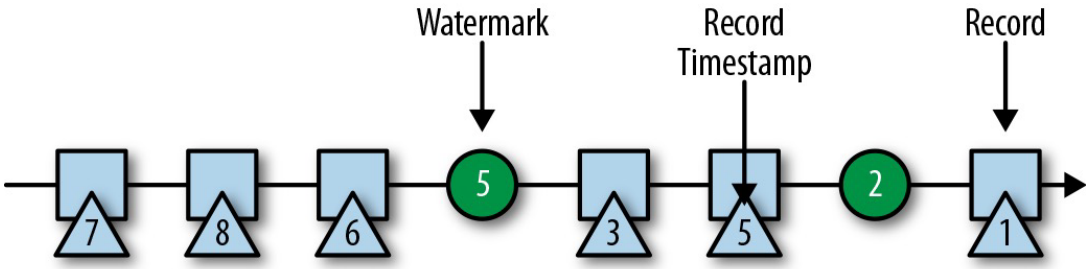
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Disorder in Data Streams

- Many queries over data streams rely on some kind of order on the input data items
  - A -> B
- What causes disorder in streams?
  - Items from the same source may take different routes
  - Many sources with varying delays
  - May have been sorted on different attribute
- Sorting a stream with watermarks

Watermarks in Apache Flink

- Events can be delayed and/or out-of-order
- When to trigger an event-time window?
- When can an operator order events?
- Watermarks provide a logical clock that informs the system about the current event time
  - Operator receives a watermark with time T => no further events with timestamp < T will be received
  - Trigger computation or order events or detect late events



- Eager watermarks => low latency & low confidence
- Relaxed watermarks => high latency & high confidence

Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Timestamp Assignment & Watermark Generation in Flink

- At the source with `sourcefunction`
- Records can be emitted together with an associated timestamp
- Watermarks can be emitted at any point in time as special records

Punctuations in Apache Samza or Kafka Streams

- Conceptually similar to watermarks in Flink
- Both being mechanisms for managing event processing in the presence of out-of-order events
- Specifics of their implementation and usage can differ
- Punctuations can serve broader signalling purposes

Queries

Queries - I

- DBS: one-time (transient) queries
- DSP: continuous (persistent) queries
  - Support persistent and transient queries
  - Predefined and ad hoc queries (CQs)
  - Examples (persistent CQs):
    - Tapestry: content-based email, news filtering
    - OpenCQ, NiagaraCQ: monitor web sites
    - Chronicle: incremental view maintenance
    - Esper: EPL, SQL like can specify windows and patterns
    - Apache Flink: SQL, DataStream API, Tabel API, CEP Library
- Unbounded memory requirements
- Blocking operators: window techniques
- Queries referencing past data

Queries - II

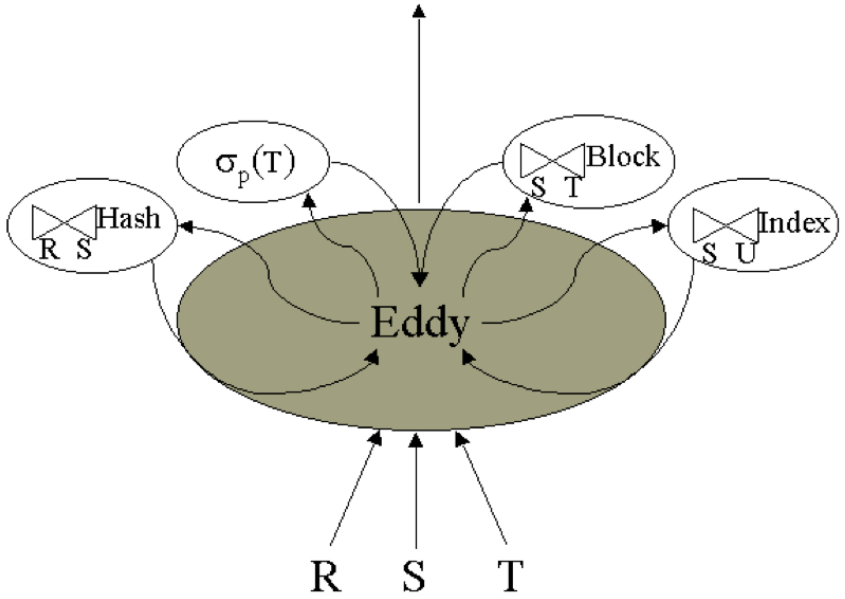
- DBS: (mostly) exact query answer
- 1. generation DSMS: (mostly) approximate query answer
  - Approximate query answers have been studied:
    - Synopses construction: histograms, sampling, sketches
    - Approximating query answers: using synopses structures
    - Approximate joins: using windows to limit scope
    - Approximate aggregates: using synopses structures
- Batch processing
- Data reduction: sampling, synopses, sketches, wavelets, histograms, ...
- Current DSP can give precise query answers
  - Scale out in cloud environments
  - Checkpointing & event logs -> processing guarantees

One-pass Query Evaluation

- DBS:
  - Arbitrary data access
  - One/few pass algorithms have been studied:
    - Limited memory selection/sorting: *n*-pass quantiles
    - Tertiary memory databases: reordering execution
    - Complex aggregates: bounding number of passes
- DSP:
  - Per-element processing: single pass to reduce drops
  - Block processing: multiple passes to optimize I/O cost

Query Plan

- DBS: fixed query plans optimized at beginning
- DSMS: adaptive query operators
  - Eddies: volatile, unpredictable environments



Ron Avnur and Joseph M. Hellerstein. 2000. Eddies: continuously adaptive query processing. S IGMOD Rec. 29, 2 (June 2000)

Query Languages & Processing

- Stream query language issues (compositionality, windows)
- SQL-like proposals suitably extended for a stream environment:
  - Composable SQL operators
  - Queries reference relations or streams
  - Queries produce relations or streams
- Query operators (selection/projection, join, aggregation)
- Examples:
  - GSQL (Gigascope)
  - CQL (STREAM)
- Optimization objectives
- Multi-query execution

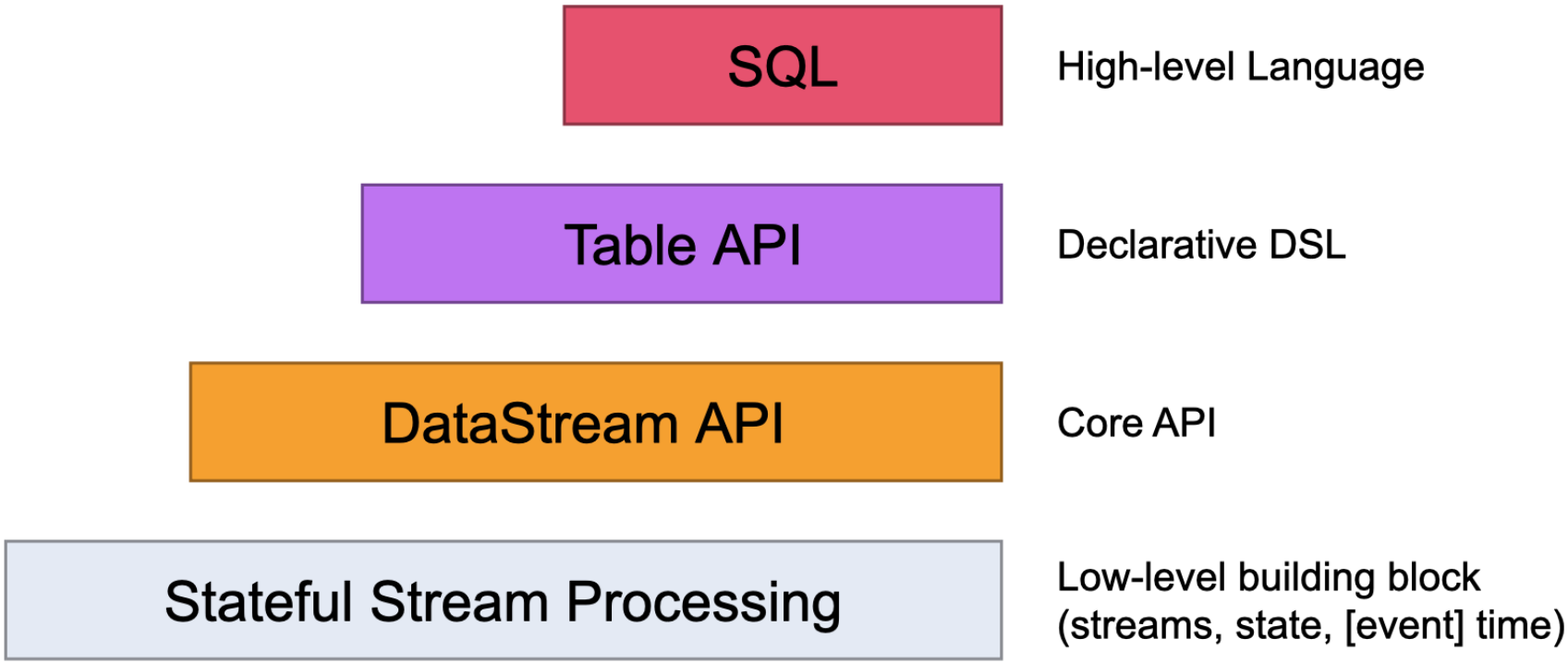
Query Languages

3 querying paradigms for streaming data:

1. Relation-based: SQL-like syntax and enhanced support for windows and ordering, e.g., Esper, CQL (STREAM), StreaQuel (TelegraphCQ), AQuery, GigaScope
2. Object-based: object-oriented stream modeling, classify stream elements according to type hierarchy, e.g., Tribeca, or model the sources as ADTs, e.g., COUGAR
3. Procedural: users specify the data flow, e.g., Aurora, users construct query plans via a graphical interface

(1) and (2) are declarative query languages,
currently, the relation-based paradigm is mostly used.

Apache Flink APIs



Source: https://nightlies.apache.org/flink/flink-docs-release-2.1/docs/concepts/overview/

Sample Stream

```
Traffic  (      sourceIP -- source IP address
         sourcePort -- port number on source
                destIP -- destination IP address
                destPort -- port number on destination
                length -- length in bytes
                time -- time stamp
           );
```

Selections, Projections

- Selections, (duplicate preserving) projections are straightforward
  - Local, per-element operators
  - Duplicate eliminating projection is like grouping
- Projection needs to include ordering attribute
  - No restriction for position ordered streams

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

Join Operators

- General case of join operators problematic on streams
  - May need to join arbitrarily far apart stream tuples
  - Equijoin on stream ordering attributes is tractable
- Majority of work focuses on joins between streams with windows specified on each stream

```
SELECT A.sourceIP, B.sourceIP
FROM Traffic1 A [window T1], Traffic2 B [window T2]
WHERE A.destIP = B.destIP
```

Aggregation

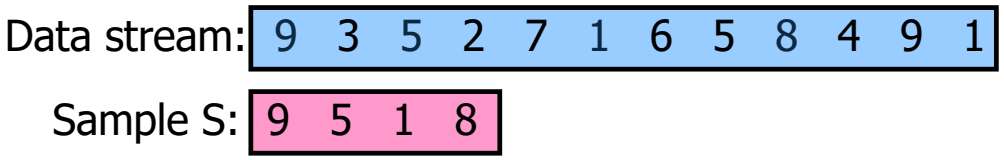
- General form:
  - **select** G, F1 **from** S **where** P **group by**G **having** F2 op θ
  - G: grouping attributes, F1,F2: aggregate expressions
- Aggregate expressions:
  - distributive: sum, count, min, max
  - algebraic: avg
  - holistic: count-distinct, median

Aggregation & Approximation

- When aggregates cannot be computed exactly in limited storage, approximation may be possible and acceptable
- Examples:
  - select G, median(A) from S group by G
  - select G, count(distinct A) from S group by G
  - select G, count(*) from S group by G having count(*) > f|S|
- Data reduction: use summary structures
  - samples, histograms, sketches …

Sampling

- A small random sample S of the data often well-represents all the data
  - Example: select agg from R where R.e is odd (n=12)



  - If agg is avg, return average of odd elements in S

answer: 5

  - If agg is count, return average over all elements e in S of
    - n if e is odd
    - 0 if e is even

answer: 12*3/4 =9 Unbiased!

Histograms

- Histograms approximate the frequency distribution of element values in a stream
- A histogram (typically) consists of
  - A partitioning of element domain values into buckets
  - A count C_B per bucket B (of the number of elements in B)
- Long history of use for selectivity estimation within a query optimizer

C_B

For details about wavelets check out
https://minosng.github.io/Papers/eds09streamwav.pdf

Wavelets

- For hierarchical decomposition of functions/signals
- Haar wavelets
  - Simplest wavelet basis => Recursive pairwise averaging and differencing at different resolutions

Resolution	Averages	Detail Coefficients
3	[2, 2, 0, 2, 3, 5, 4, 4]	----
2	[2, 1, 4, 4]	[0, -1, -1, 0]
1	[1.5, 4]	[0.5, 0]
0	[2.75]	[-1.25]

Haar wavelet decomposition: [2.75, -1.25, 0.5, 0, 0, -1, -1, 0]

Query Optimization

- DBS: table based cardinalities used in query optimization
  => Problematic in a streaming environment
- Cost metrics and statistics: accuracy and reporting delay vs. memory usage, output rate, power usage
- Query optimization: query rewriting to minimize cost metric, adaptive query plans, due to changing processing time of operators, selectivity of predicates, and stream arrival rates
- Query optimization techniques
  - stream rate based
  - resource based
  - QoS based
- Continuously adaptive optimization
- Possibility that objectives cannot be met:
  - resource constraints
  - bursty arrivals under limited processing capability

Load shedding

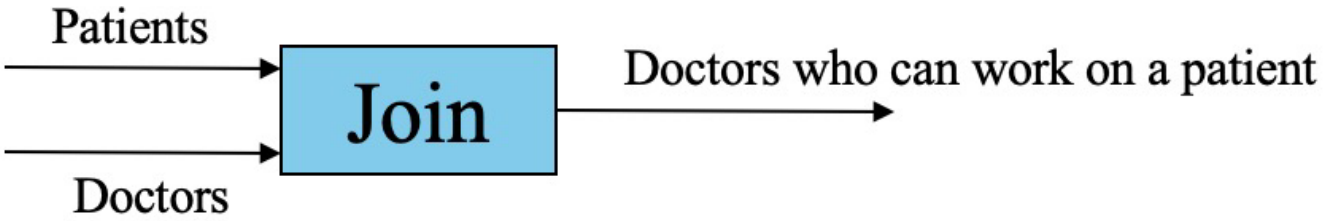
Load shedding

- Systems have a limit to how much fast data can be processed
- When the rate is too high, Queues will build up waiting for system resources
- Loadshedding discards some data so the system can flow
- Different from networking loadshedding
  - Data has semantic value in DSMS
  - QoS can be used to find the best stream to drop

Kevin Hoeschele, Anurag Shakti Maskey: Load Shedding in a Data Stream Manager, https://www.cs.brandeis.edu › slides › lsaurora

Hospital - Network

- Hospital - Network
  - Stream of free doctors locations
  - Stream of untreated patients locations, their condition (dieing, critical, injured, barely injured)
  - Output: match a patient with doctors within a certain distance



Kevin Hoeschele, Anurag Shakti Maskey: Load Shedding in a Data Stream Manager, https://www.cs.brandeis.edu › slides › lsaurora

Too many Patients, what to do?

- Loadshedding based on condition
  - Official name “Triage”
  - Most critical patients get treated first
  - Filter added before the Join
    - Selectivity based on amount of untreated patients

Kevin Hoeschele, Anurag Shakti Maskey: Load Shedding in a Data Stream Manager, https://www.cs.brandeis.edu › slides › lsaurora

Aurora Overview

- Push based data from streaming sources
- 3 kinds of Quality of Service
  - Latency
    - Shows utility drop as answers take longer to achieve
  - Value-based
    - Shows which output values are most important
  - Loss-tolerance
    - Shows how approximate answers affect a query

Kevin Hoeschele, Anurag Shakti Maskey: Load Shedding in a Data Stream Manager, https://www.cs.brandeis.edu › slides › lsaurora

Loadshedding Techniques

- Filters (semantic drop)
  - Chooses what to shed based on QoS
  - Filter with a predicate in which selectivity = 1-p
  - Lowest utility tuples are dropped

- Drops (random drop)
  - Eliminates a random fraction of input
  - Has a p% chance of dropping each incoming tuple

Kevin Hoeschele, Anurag Shakti Maskey: Load Shedding in a Data Stream Manager, https://www.cs.brandeis.edu › slides › lsaurora

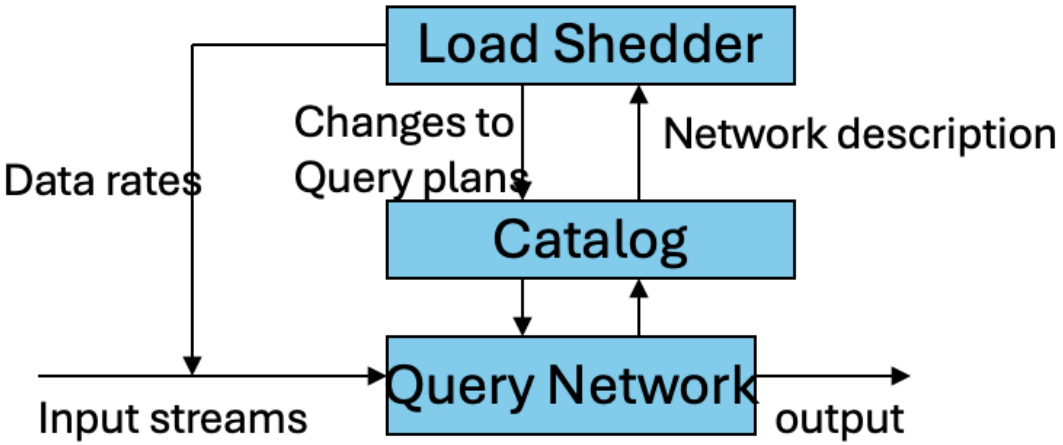
3 Questions of Load Shedding

- When
  - Load of system needs constant evaluation
- Where
  - Dropping as early as possible saves most resources
    - Can be a problem with streams that fan out and are used by multiple queries
- How much
  - the percent for a random drop
  - Make the predicate for a semantic drop(filter)

Kevin Hoeschele, Anurag Shakti Maskey: Load Shedding in a Data Stream Manager, https://www.cs.brandeis.edu › slides › lsaurora

Load Shedding in Aurora

- Aurora Catalog
  - Holds QoS and other statistics
  - Network description
- Load shedder monitors these and input rates: makes loadshedding decisions
  - Inserts drops/filters into the query network, which are stored in the catalog

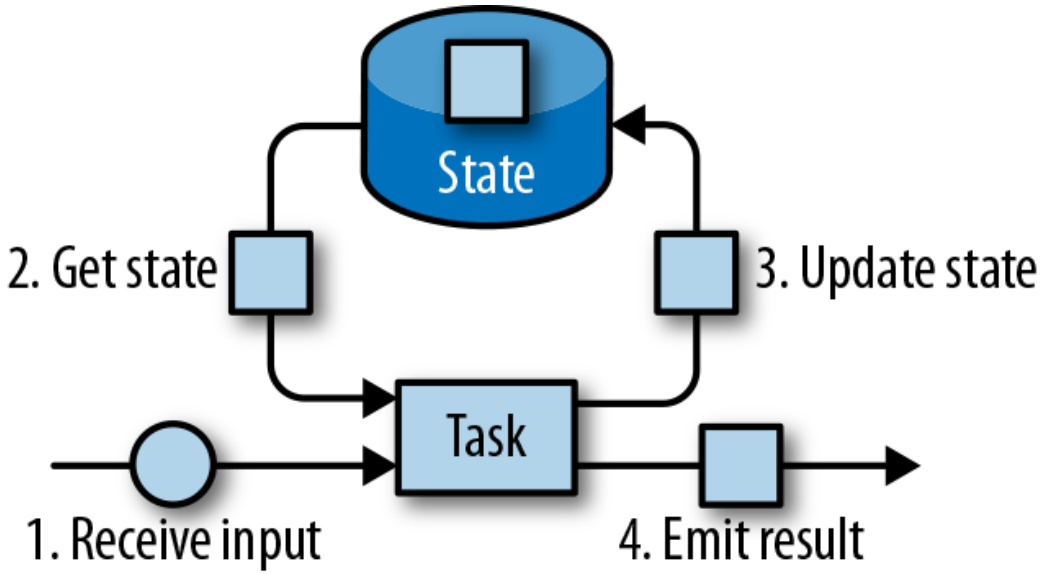


Kevin Hoeschele, Anurag Shakti Maskey: Load Shedding in a Data Stream Manager, https://www.cs.brandeis.edu › slides › lsaurora

Operator State

State Management in Apache Flink

- State can grow very large
- State must not be lost in case of failure
- Flink takes care of state consistency, failure handling, and efficient storage and access
- Developers focus on the logic of the application



Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Example: State of `sum` Operator

- Sum operator with count based tumbling window size 3
  - S = (5, t_1), (3, t_2), (4,t_3), (7, t_4), ...
  - State over time

t_1: 5
t_2: 5, 3
t_3: 5, 3, 4 -> 12
t_4: 7

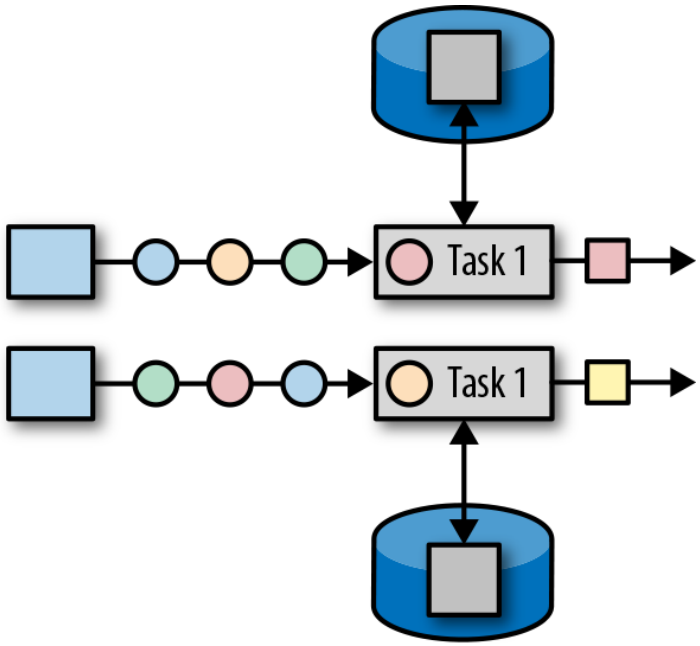
Example: State of `followed by` (=>) operator

- A => B Count-based tumbling window size 4
- S1: (C, t_1), (B, t_2), (A, t_3), (H, t_4), (C, t5)

- State over time

	S_1	
t_1:		
t_2:		
t_3:	A	
t_4:	A	-> no output
t_5:		

State of a Parallel Task in Flink

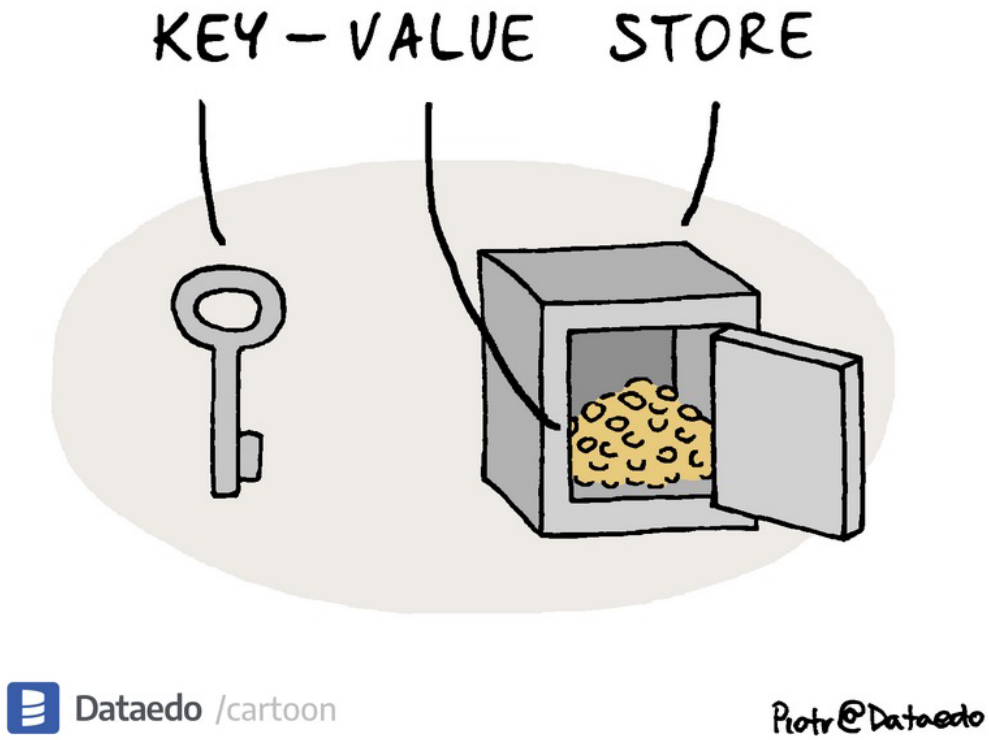


Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Primitives for Operator State in Flink

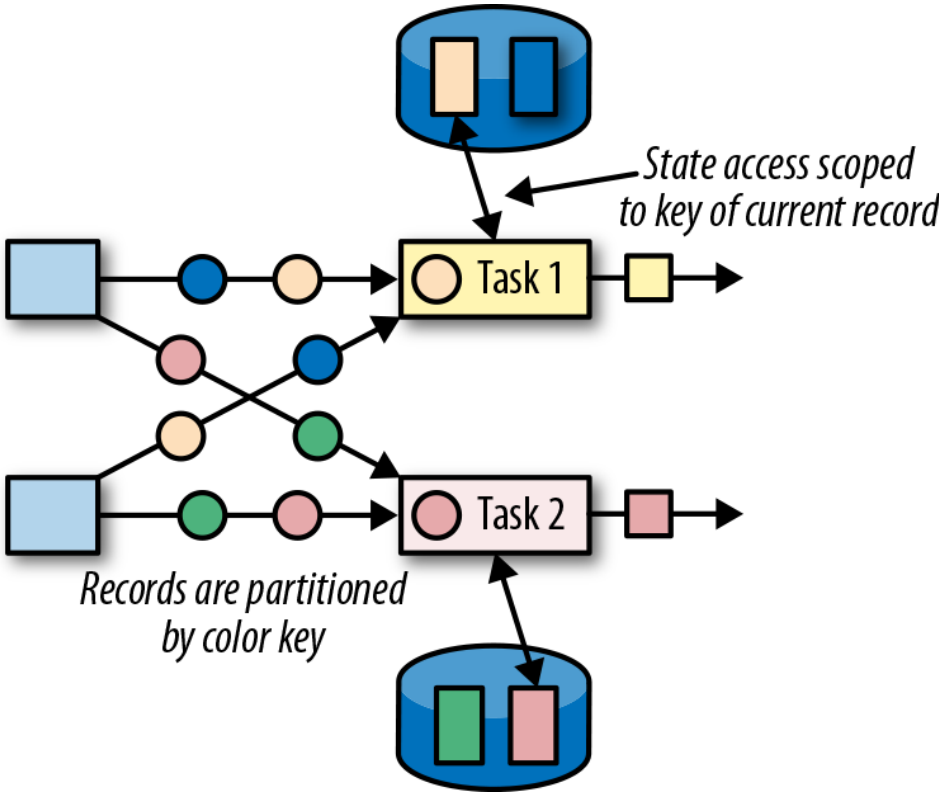
- List state
  - Represents state as a list of entries
- Union list state
  - Represents state as a list of entries as well,
  - but it differs from regular list state in how it is restored in the case of a failure or when an application is started from a savepoint. We discuss this difference later in this chapter.
- Broadcast state
  - Designed for the special case where the state of each task of an operator is identical. This property can be leveraged during checkpoints and when rescaling an operator. Both aspects are discussed in later sections of this chapter.

What is a Key-Value Store?



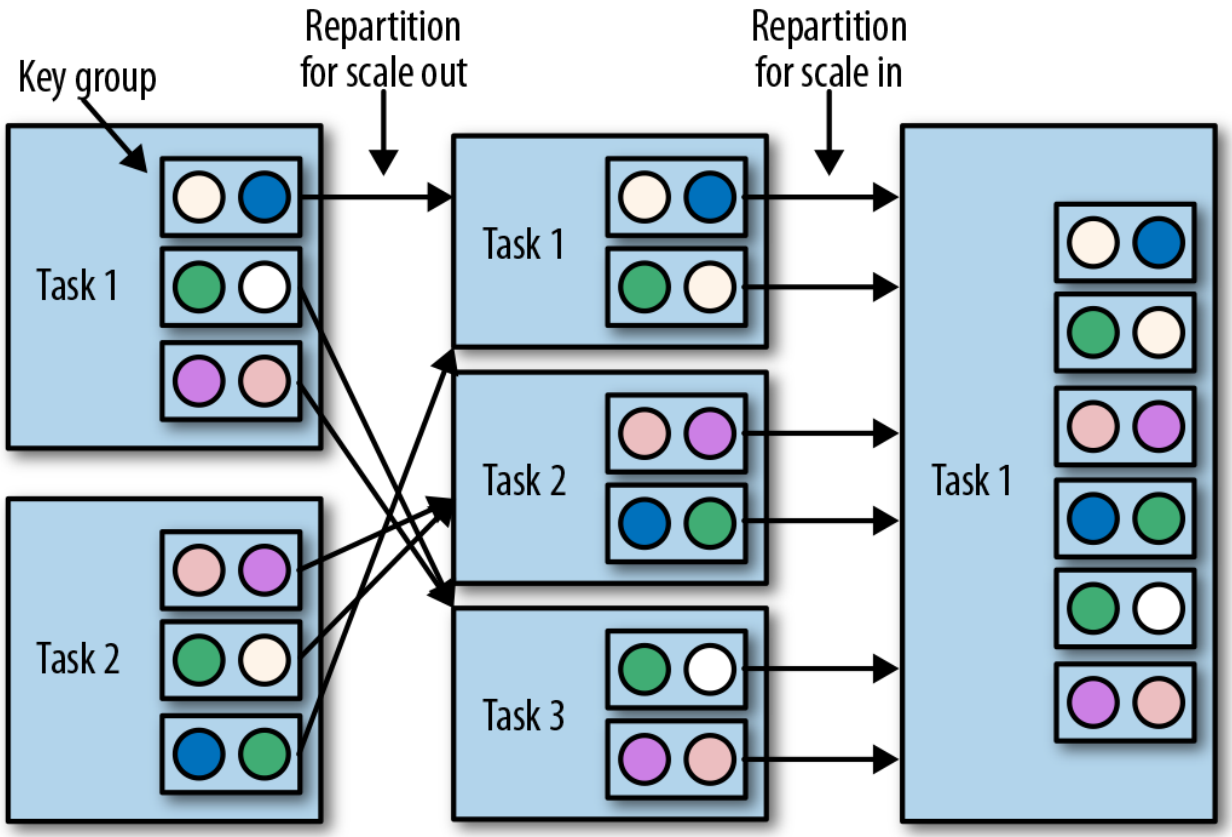
Source: https://dataedo.com/cartoon/key-value-store, Artist: Piotr Kononow

Keyed State in Flink



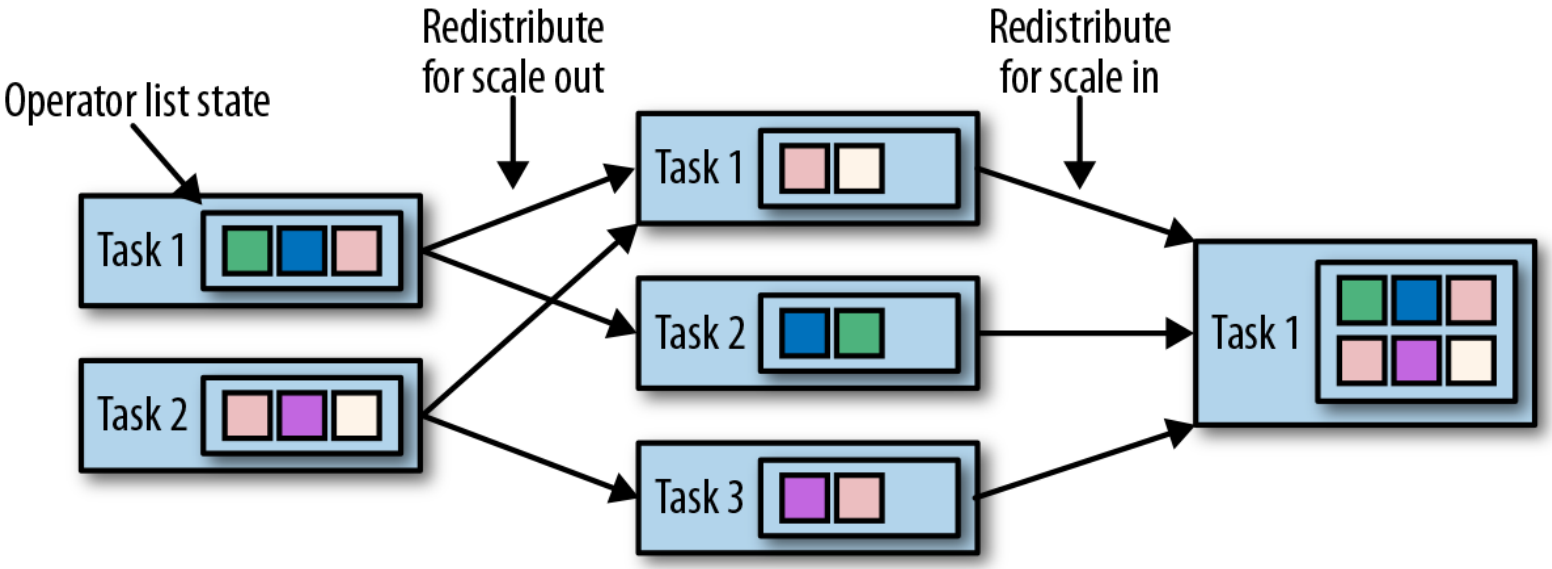
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Scaling Stateful Operators - keyed State



Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Scaling Stateful Operators– list State



Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

State – Implementation Challenges

- State management
  - concurrent updates
- State partitioning
  - Parallelization
- State recovery
  - Correct results in presence in failure

State Backends in Flink

- Delegating state management to 3rd party backends
- In-memory: data structures on the JVM heap
  - Very fast access
  - limited memory
- Hard disk: serialize state objects, put them into RocksDB, which writes them to local hard disks
  - Slower access
  - Large memory space

.

.... and if something breaks?

Task Failure & Result Guarantees

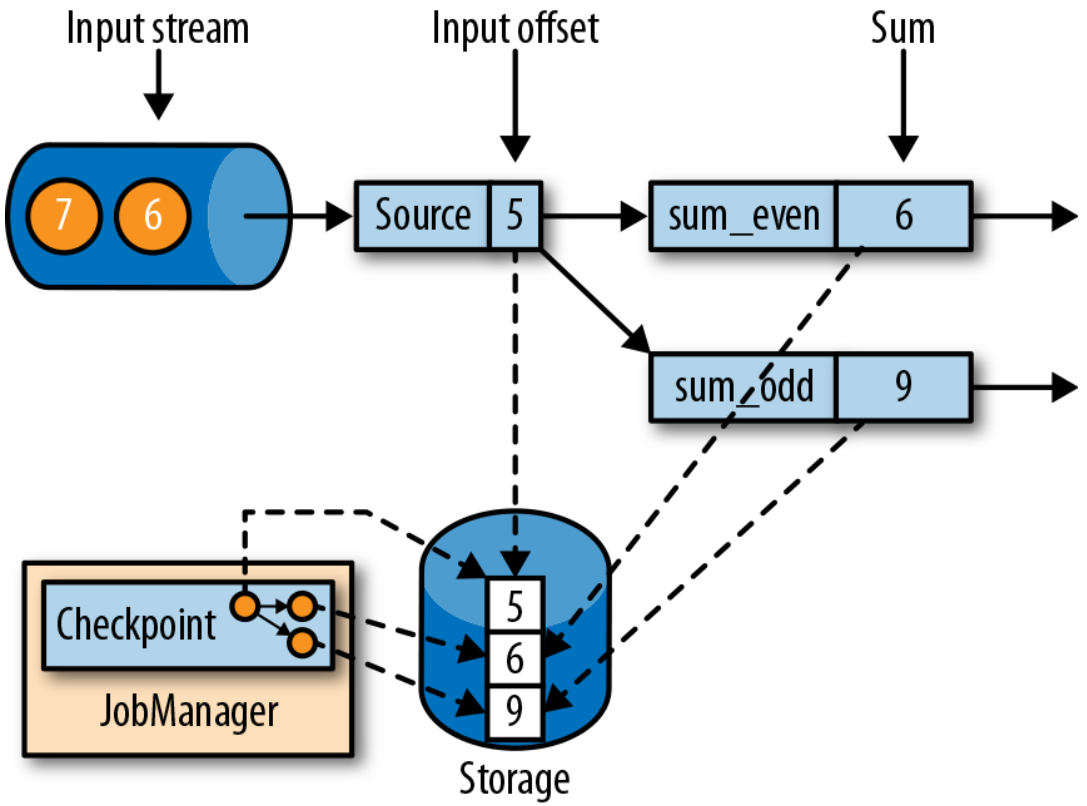
- Processing steps of operators
  - Receive event & buffer it locally
  - Process event & possibly update internal state
  - Produce result and output it (e.g., at end of window)
- Failures can occur in any step
- Result guarantees
  - At-most-once
  - At-least-once
  - Exactly-once
- Checkpoints, savepoints, and state recovery

At-least-once and Exactly-once

- Use checkpoint and recovery to go back to a consistent point
- At-least-once: replay events that where send after the checkpoint -> select appropriate source connector, e.g., Apache Kafka
- Exactly-once:
  - At-least-once plus
  - Ensure internal state consistency through
    - Transactional updates
    - Lightweight snapshotting mechanisms
- End-to-end exactly-once requires additionally
  - special sink connectors, or
  - Idempotent writes, or
  - Transactional writes: send only results to the outside that have been calculated before the last checkpoint

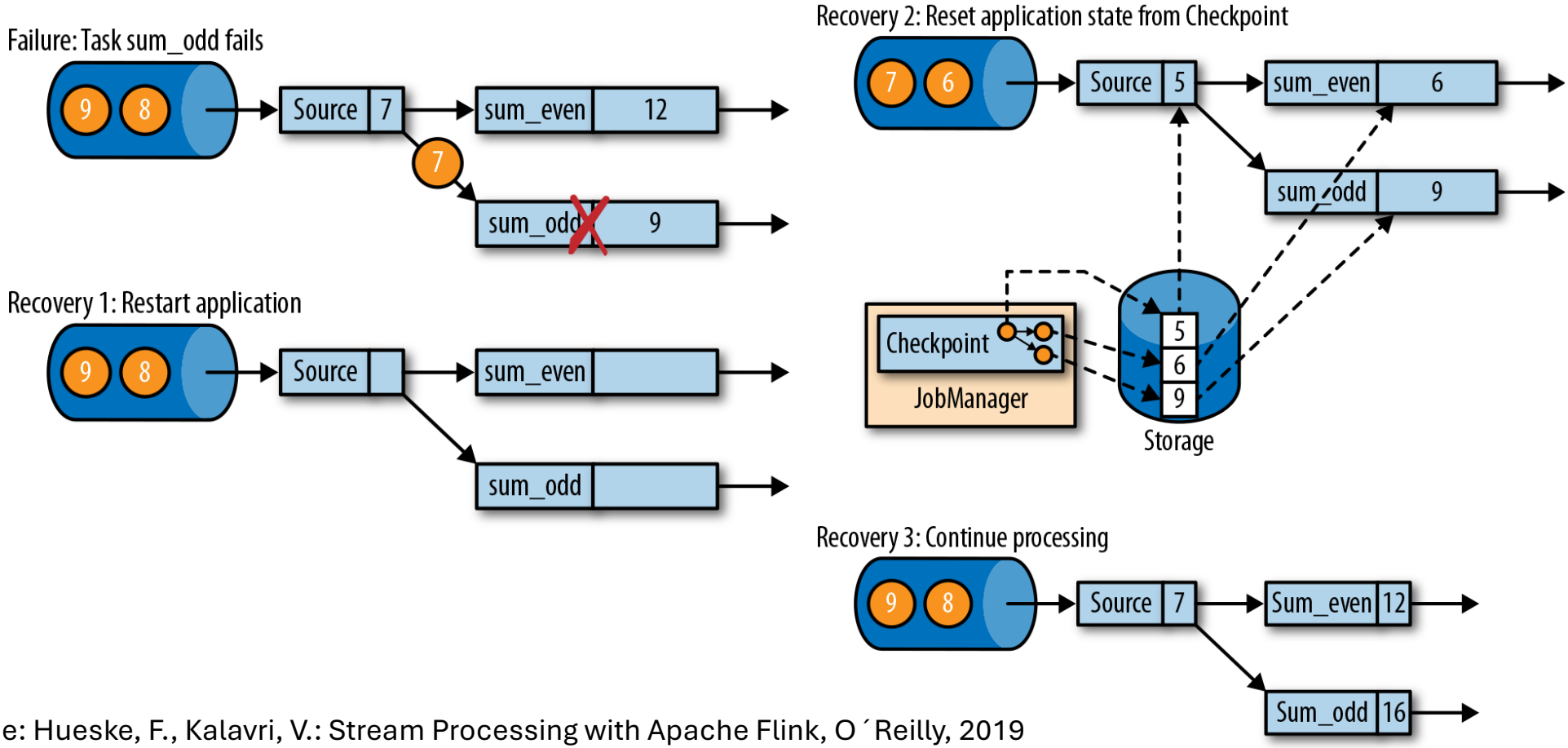
Consistent Checkpoints in Flink

- The naïve approach (not done in Flink):
  - Pause input
  - Finish processing all in-flight tuples
  - Checkpoint
  - Resume input



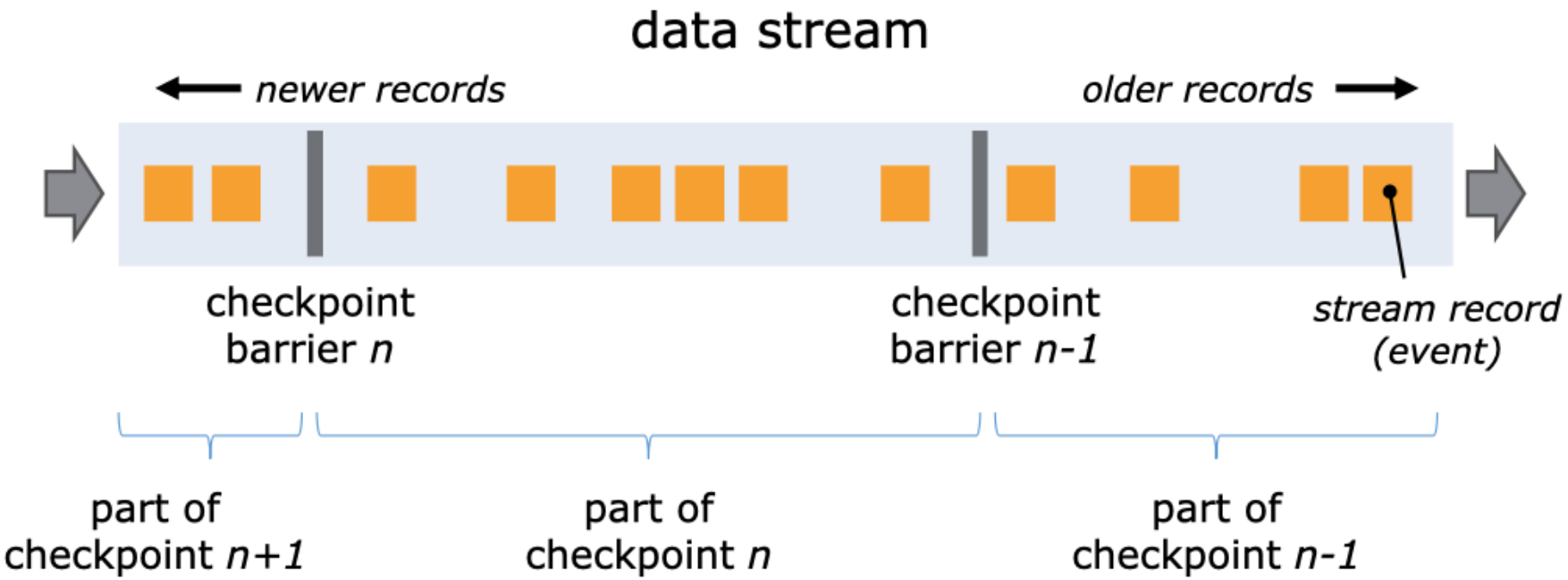
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Recovery from Consistent Checkpoint



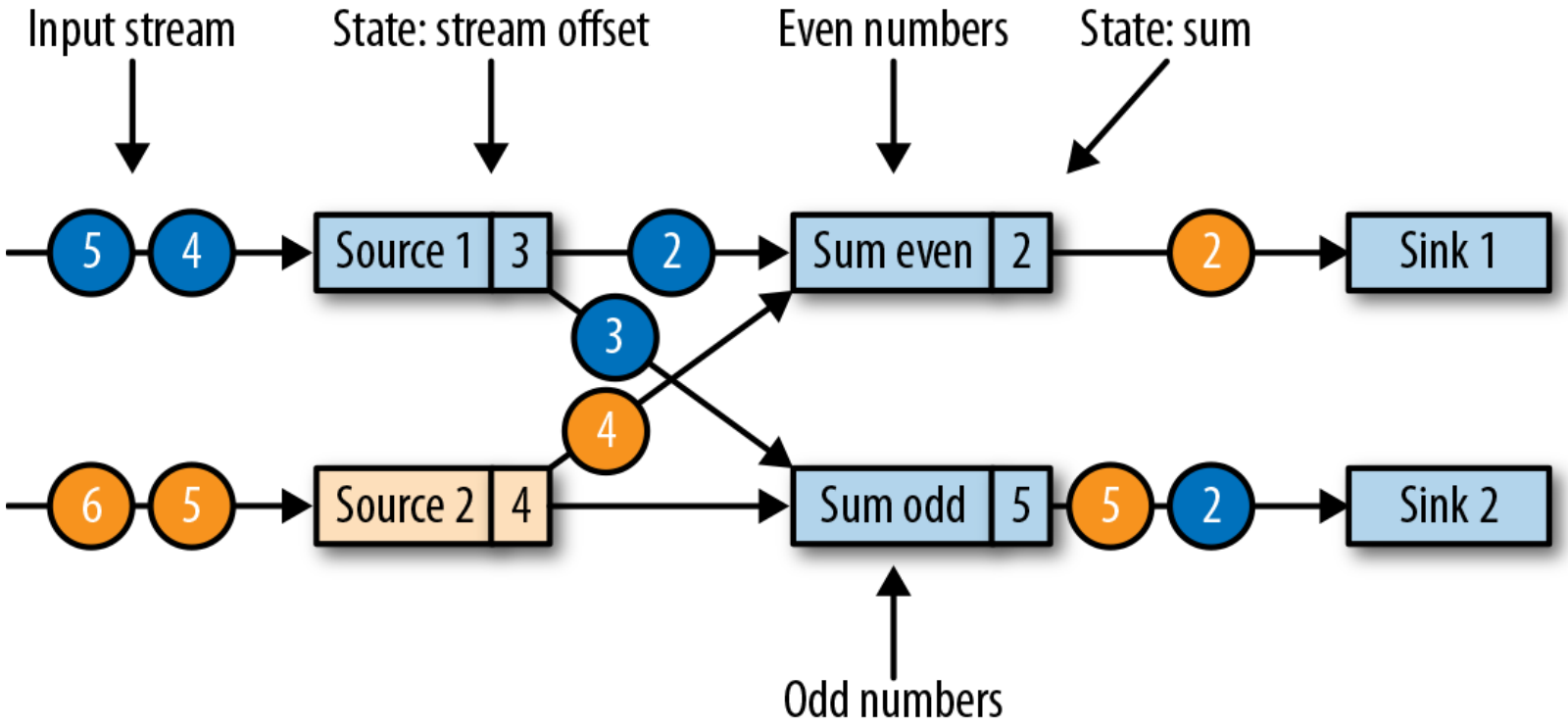
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Checkpoint Barrier



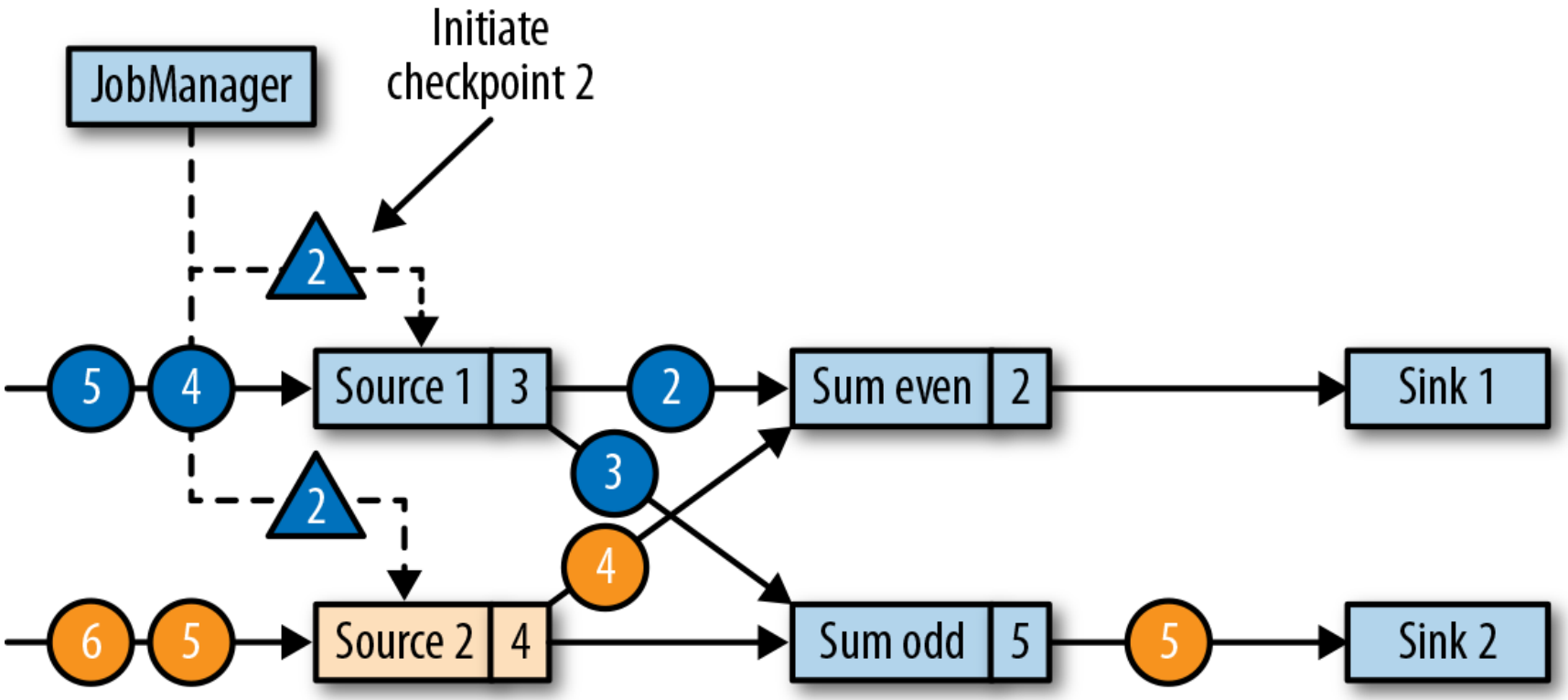
Source: https://nightlies.apache.org/flink/flink-docs-release-2.1/docs/concepts/stateful-stream-processing/

Chandy-Lamport Algorithm for Checkpointing 1



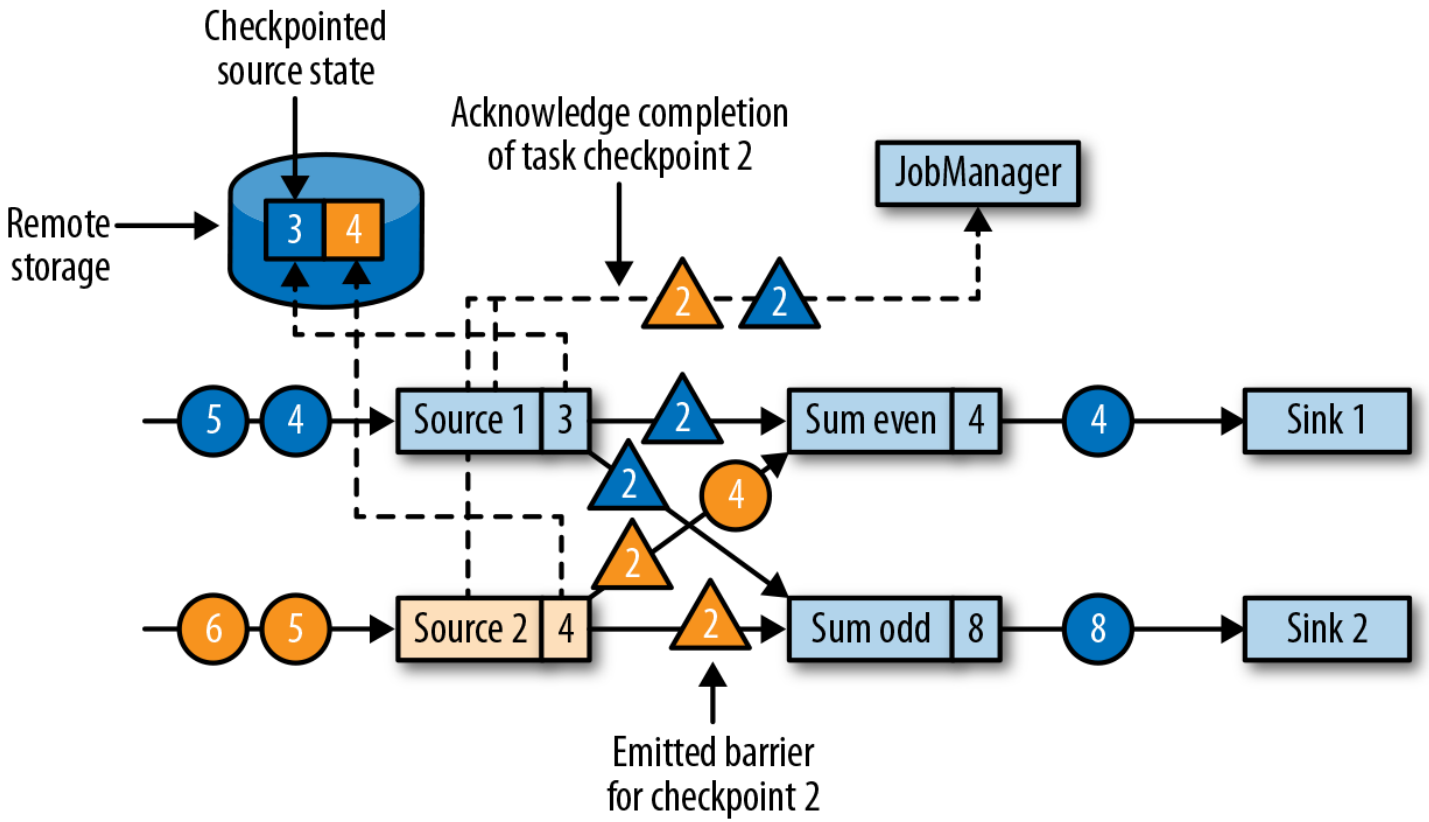
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Chandy-Lamport Algorithm for Checkpointing 2



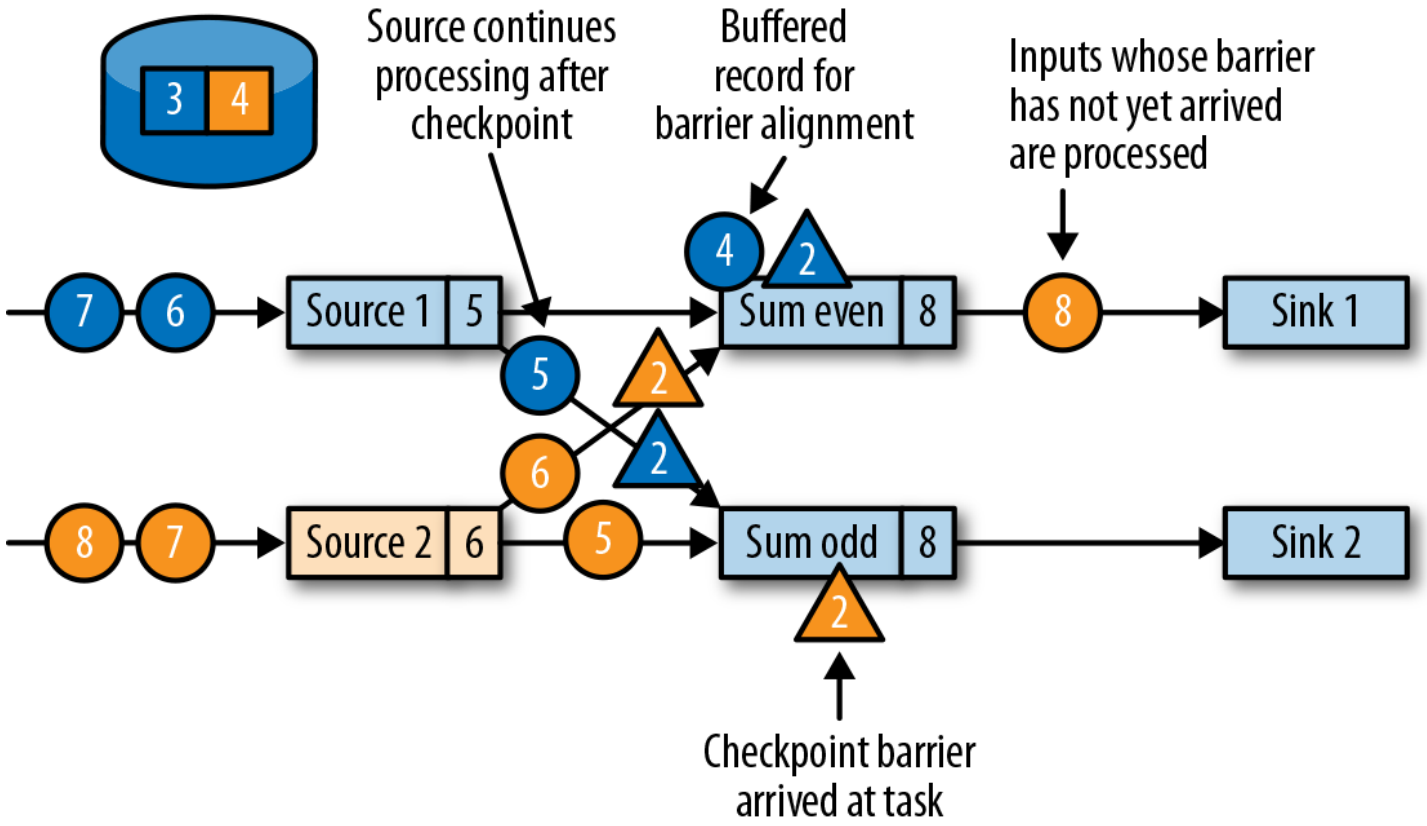
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Chandy-Lamport Algorithm for Checkpointing 3



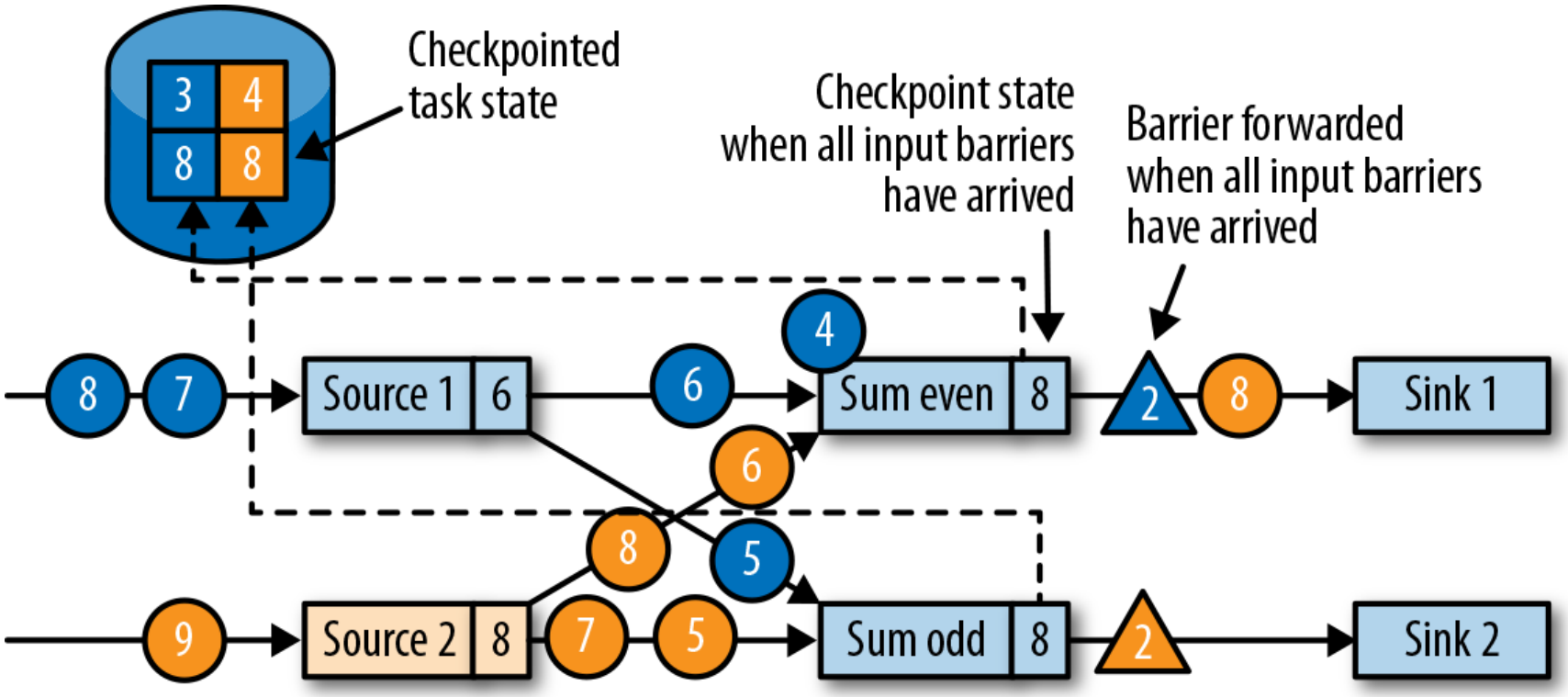
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Chandy-Lamport Algorithm for Checkpointing 4



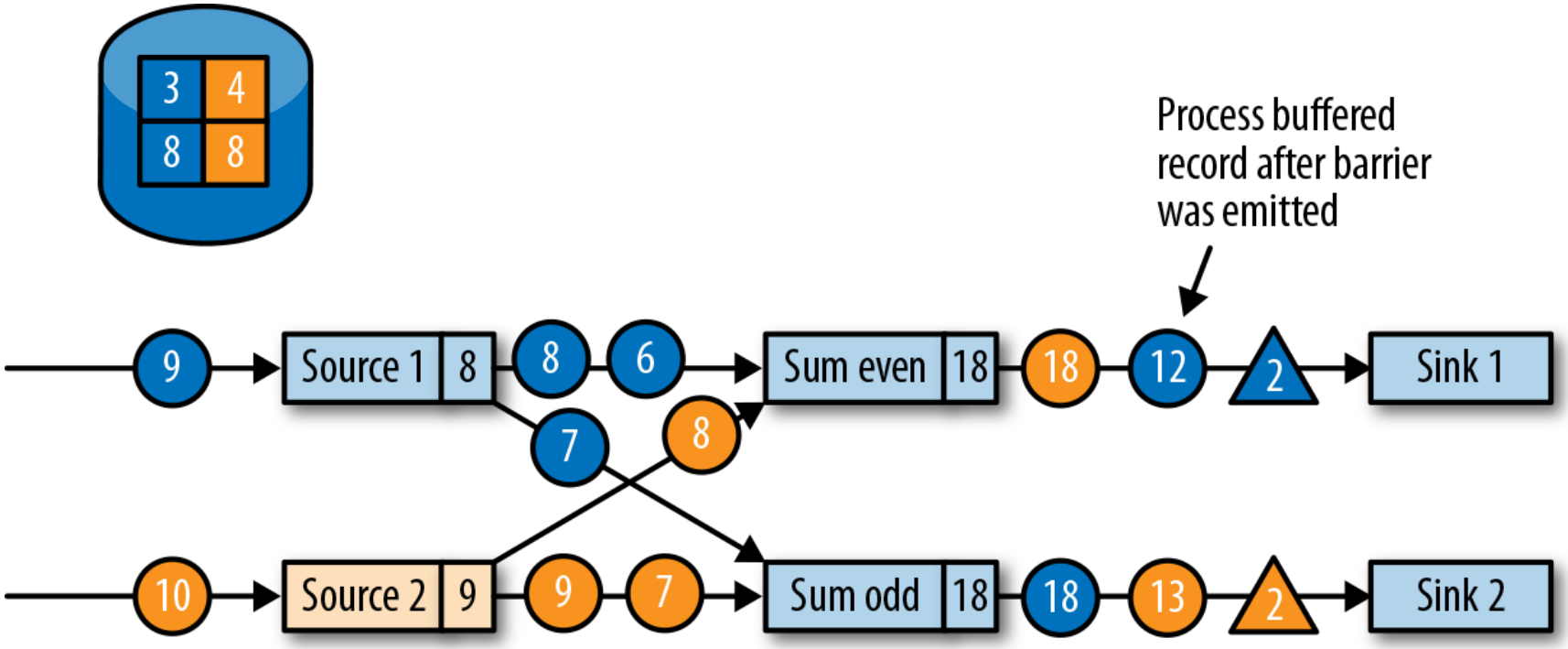
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Chandy-Lamport Algorithm for Checkpointing 5



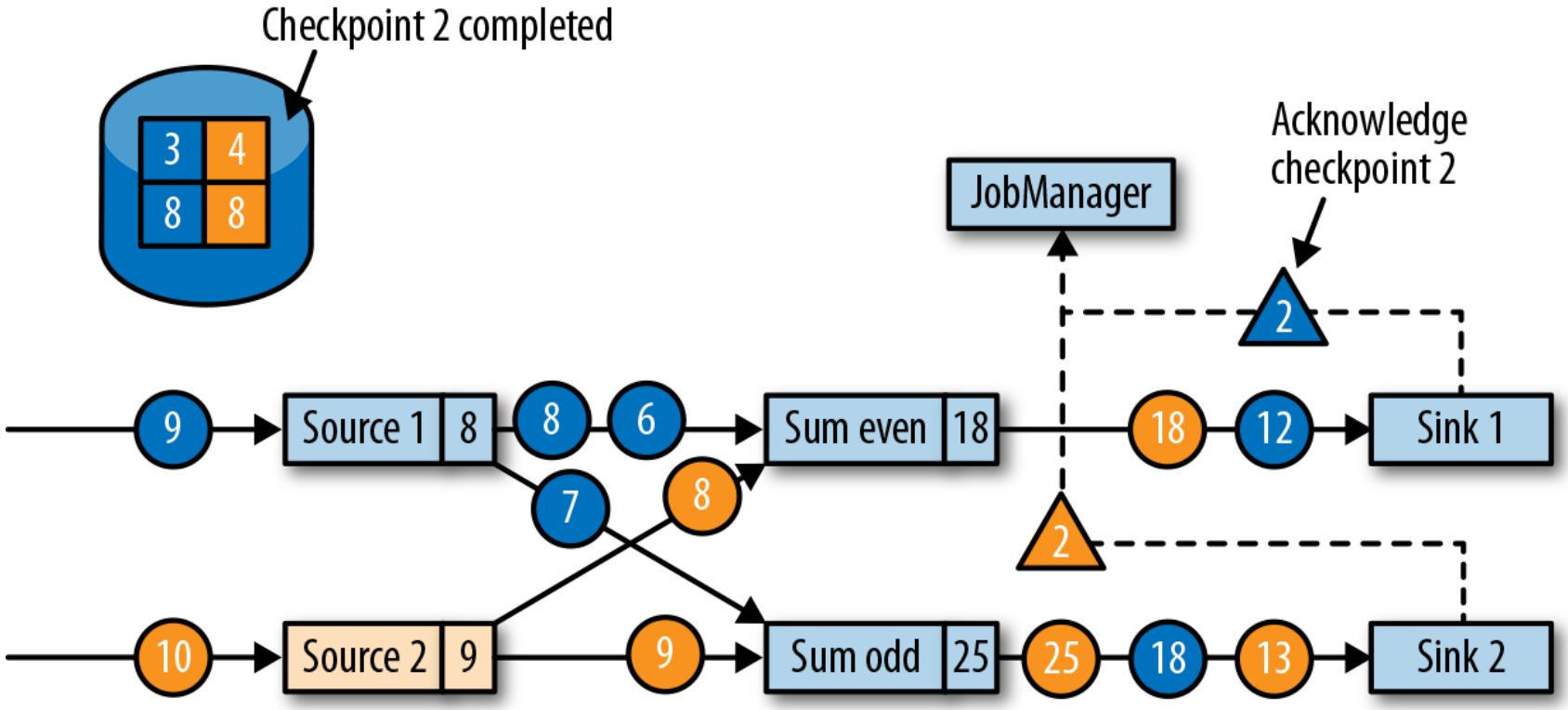
Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Chandy-Lamport Algorithm for Checkpointing 6



Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Chandy-Lamport Algorithm for Checkpointing 7

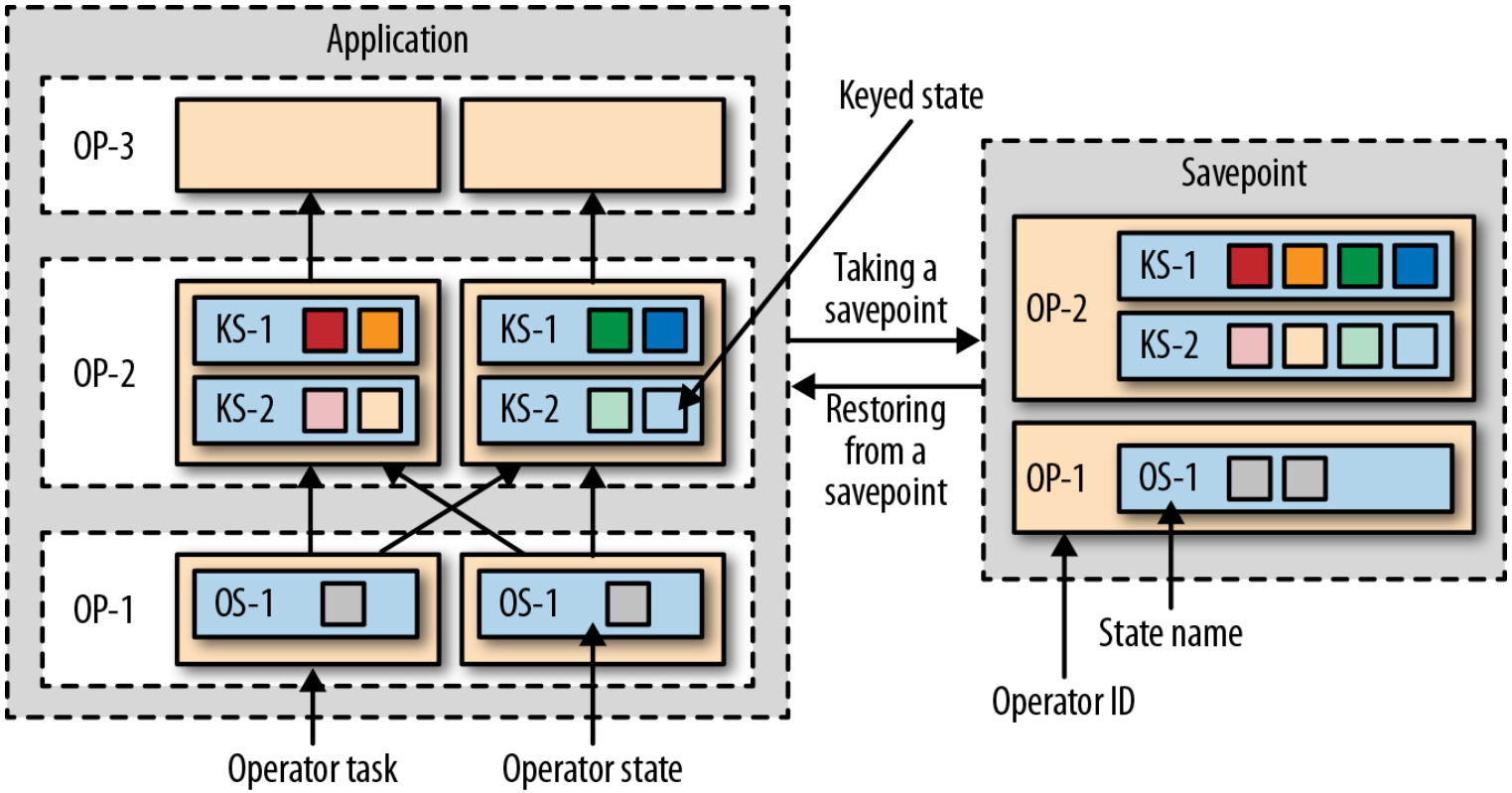


Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Savepoints in Flink

- User must trigger creation
- Savepoints are Checkpoints with additional meta data to also
  - Start a different but compatible application from a savepoint for bug fixing, A/B test,…
  - Start the same application with a different parallelism
  - Start the same application on a different cluster
  - Pause an application and resume it later.
  - Version and archive the state of an application

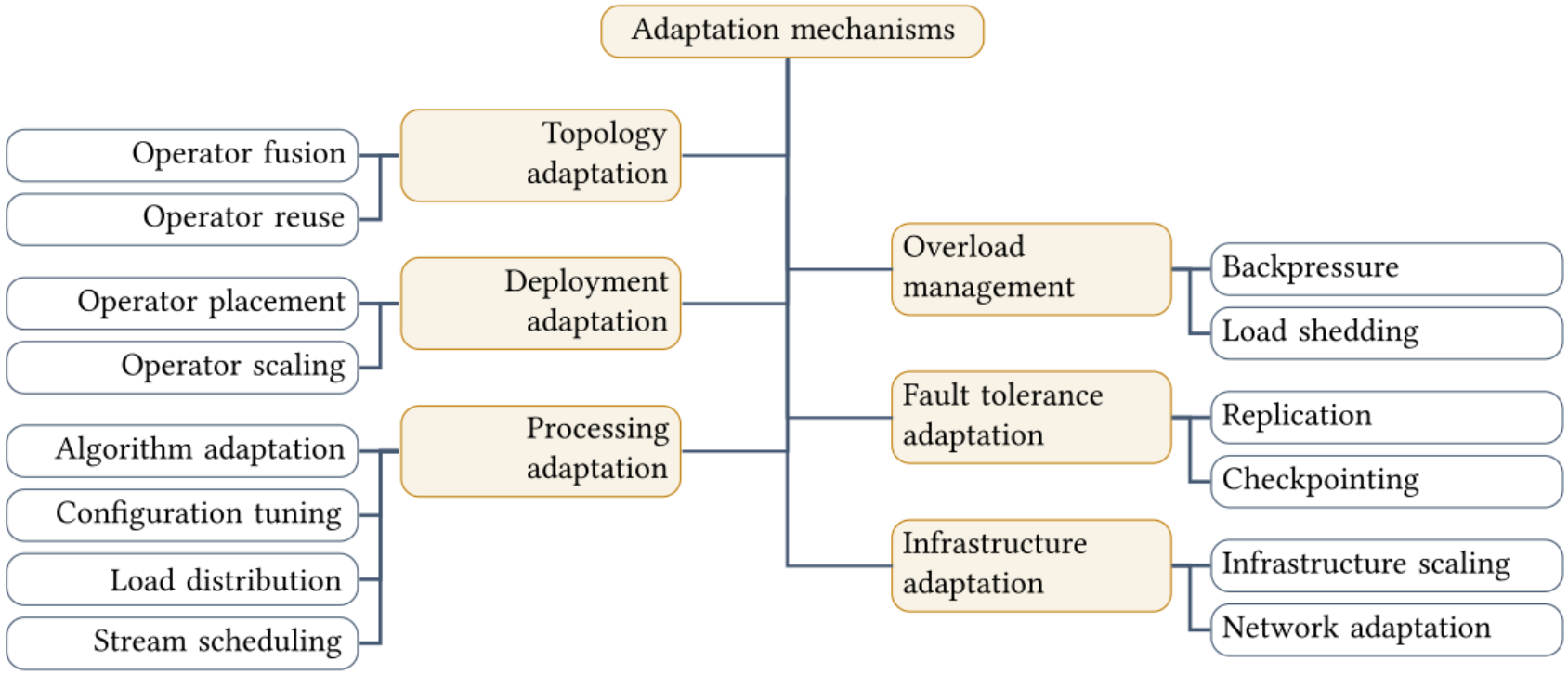
Savepoints in Flink



Source: Hueske, F., Kalavri, V.: Stream Processing with Apache Flink, O´Reilly, 2019

Big pictures from recent surveys

Runtime Adaptation of Data Stream Processing Systems



Cardellini, V., Lo Presti, , F., Nardelli, M., Russo, G.. 2022. Runtime Adaptation of Data Stream Processing Systems: The State of the Art. ACM Comput. Surv. 54, 11s, Article 237 (January 2022)

Evolution of streaming systems

	1st generation	2nd–3rd generation
Results	Approximate or exact	Exact
Programming interface	SQL extensions, CQL	UDF-heavy—Java, Scala, Python, SQL-like, etc.
Query plans	Global, optimized, with pre-defined operators	Independent, with custom operators
Query execution	(Mostly) scale-up	Distributed
Parallelism	Pipeline	Data, pipeline, task
Time and progress	Heartbeats, slack, punctuations	Low-watermark, frontiers
State management	Shared synopses, in-memory	Per query, partitioned, persistent, larger-than-memory
Fault tolerance	HA-focused, limited correctness guarantess	Distributed snapshots, exactly-once
Load management	Load shedding, load-aware scheduling	Backpressure, elasticity

[Source: Fragkoulis, M., Carbone, P., Kalavri, V. *et al.* A survey on the evolution of stream processing systems. *The VLDB Journal* **33**, 507–541 (2024). https://doi.org/10.1007/s00778-023-00819-8]

A bit about our research

Parrot

TECHNISCHE UNIVERSITÄT ILMENAU

Prof. B. Koldehofe

Alexrk2, CC BY-SA 3.0 <https://creativecommons.org/licenses/by-sa/3.0>, via Wikimedia Commons

UiO : University of Oslo

Prof. V. Goebel
Prof. T. Plagemann
PhD candidate H. Gu
PhD candidate T. Javel

TECHNISCHE UNIVERSITÄT DARMSTADT

Prof. M. Hollick
PostDoc M. Fomichev

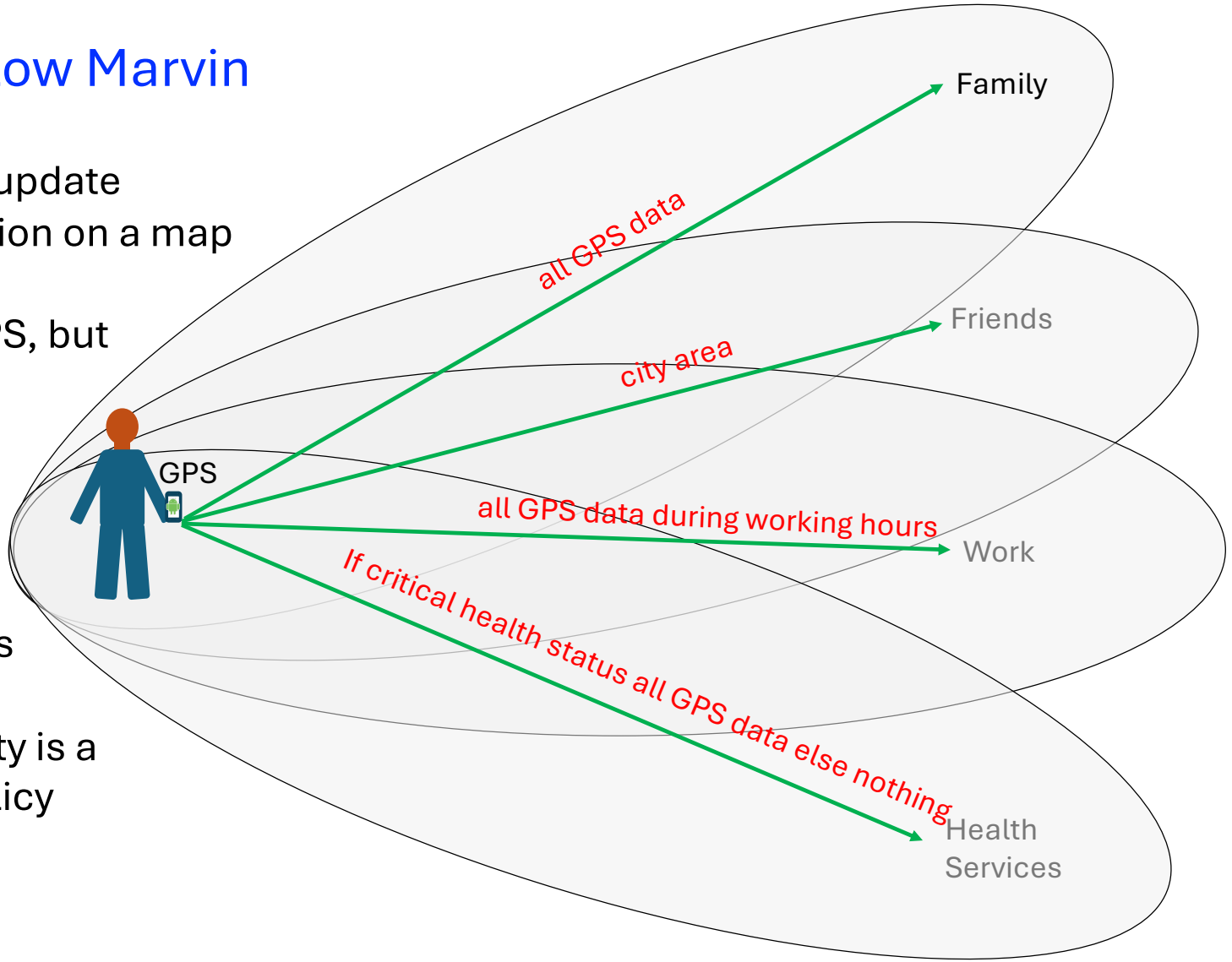
Use Case: Follow Marvin

App: continuously update
Marvin ´s position on a map

Marvin provides GPS, but not all should see “everything”

Marvin is member of 4 closed groups called communities

For each community is a specific privacy policy

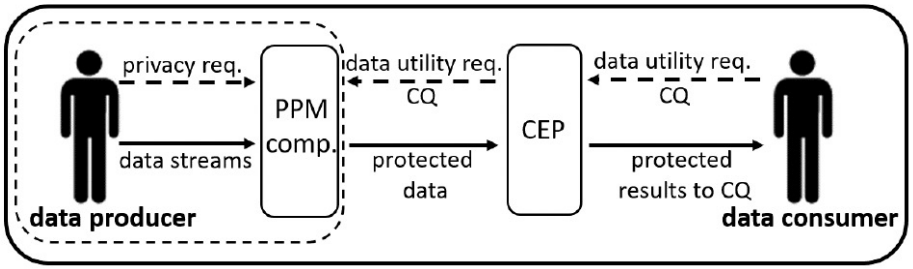


Problem and Core Idea

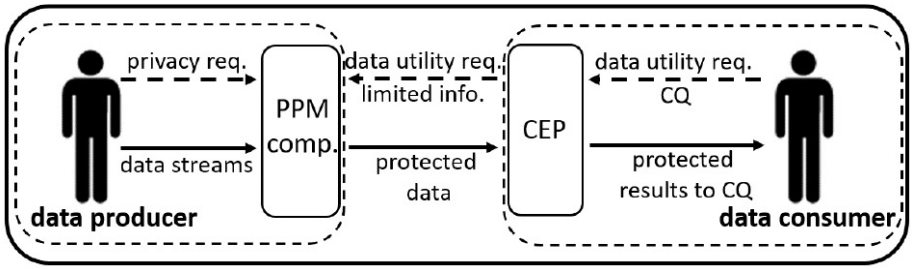
- Privacy utility tradeoff
  - Data producer wants max privacy
  - Data consumer wants max utility
- Privacy protection (PP) reduces utility (e.g., adds noise)
- PP in data streams ≠ PP in databases

- Not all data that is shared needs protection
- Focus protection on sensible data
- Try to avoid obfuscating data that is not private, but important for the data concumer

The System Model



(a) Privacy model for transparent data stream analysis



(b) Privacy model for opaque data stream analysis

- Data producer:
  - queries that identify private complex events
- Data consumer:
  - Data utility needs
  - Query
- PP Mechanism (PPM) based on differential privacy for streams:
  - Use producer query to identify in the stream private complex events
  - Consider consumer query to optimally distribute the privacy budget

Next Problem

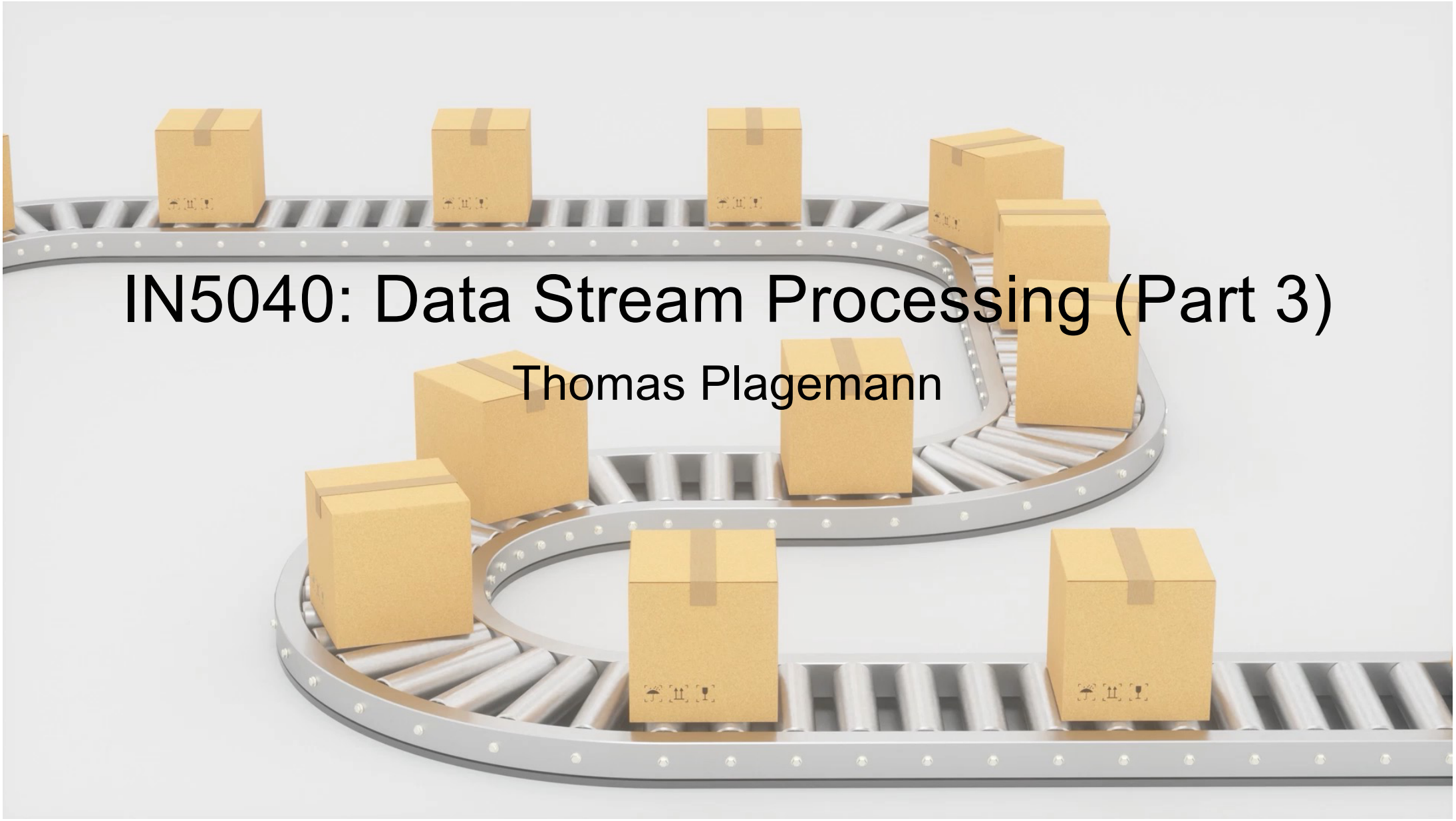
- Which data producer is able to specify those queries?
- Even “simple” anonymization is hard

- -> pArborist to help the data producer
  - Just some seeds needed from data producer
  - Use combinatorics and statistic to generate “all” privacy revealing queries

- How to evaluate such a solution
- -> We need your help!

More on our DSP research in the Cloud lecture

But now let´s go to your first assignment



IN5040: Data Stream Processing (Part 3)

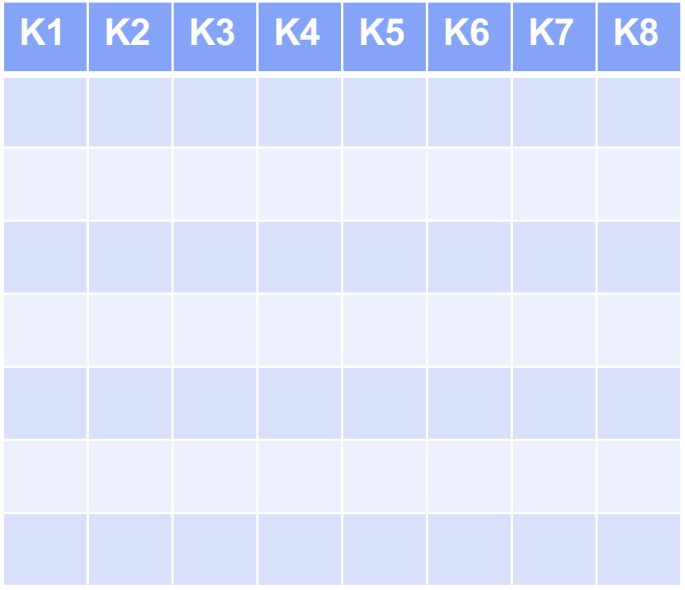
Thomas Plagemann

Apache Flink in Cloud Environments - Storage

- State-backends for Flink
  - Manage state
  - Checkpointing
- Pluggable file systems
  - Amazon S3
  - Aliyun Object Storage Service
  - Azur Data Lake Store Gen 2
  - Azur Blob Storage
  - Google Cloud Storage

Apache Flink in Cloud Environments - CPU

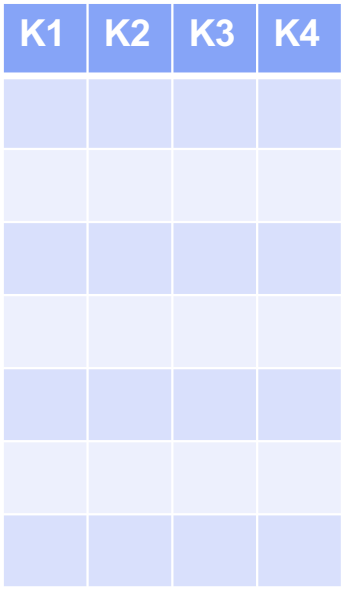
Host 1



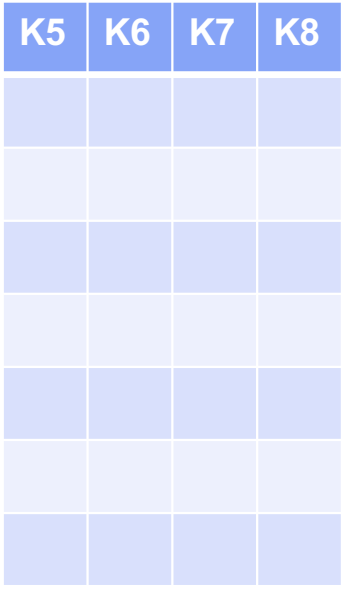
Overload

- scale out
- Extract some state
- Migrate it to other host(s)
- Update operator graph

Host 2



Host 1



UNIVERSITY OF OSLO

Semantic and utility aware operator migration for Distributed Stream Processing

Thomas Plagemann

UiO : University of Oslo

Espen Volnes

Distributed Stream Processing: Performance Evaluation and Enhanced Operator Migration

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics
Faculty of Mathematics and Natural Sciences

2023

In collaboration with Vera Goebel, Boris Koldehofe, Thomas Plagemann

Supported by the Parrot Project (Norwegian Research council, project number 311197)

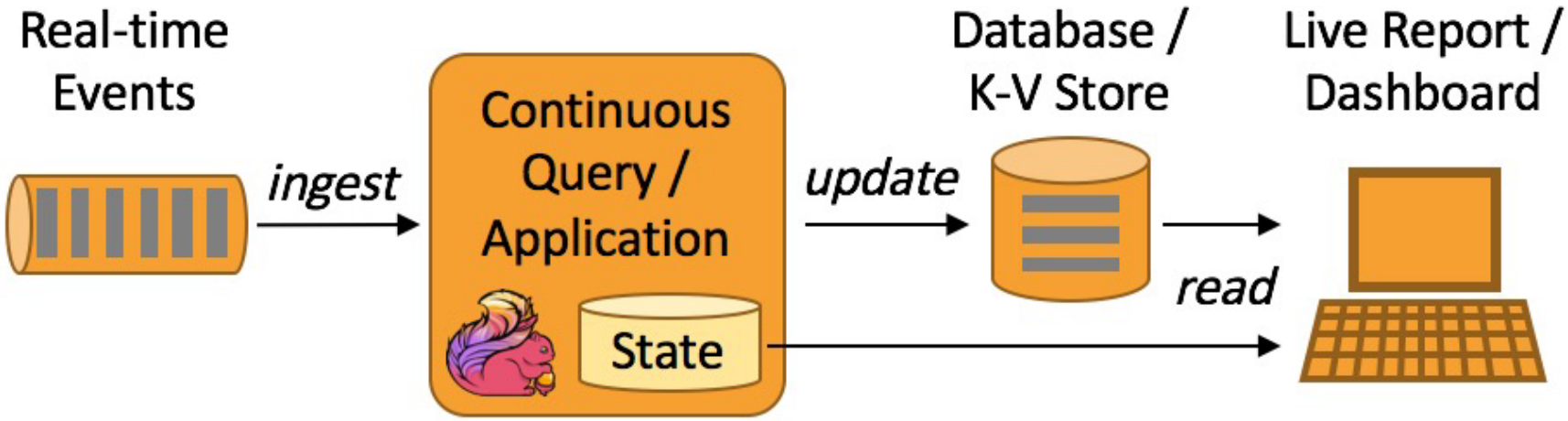
Outline

- Background
  - Stream processing
  - Operator migration
- Core idea
- Implementation
- Evaluation
- Conclusions

Data stream sources and applications

[Source: https://www.icterra.com/iot-is-coming-over-but-are-we-ready-for-it/]

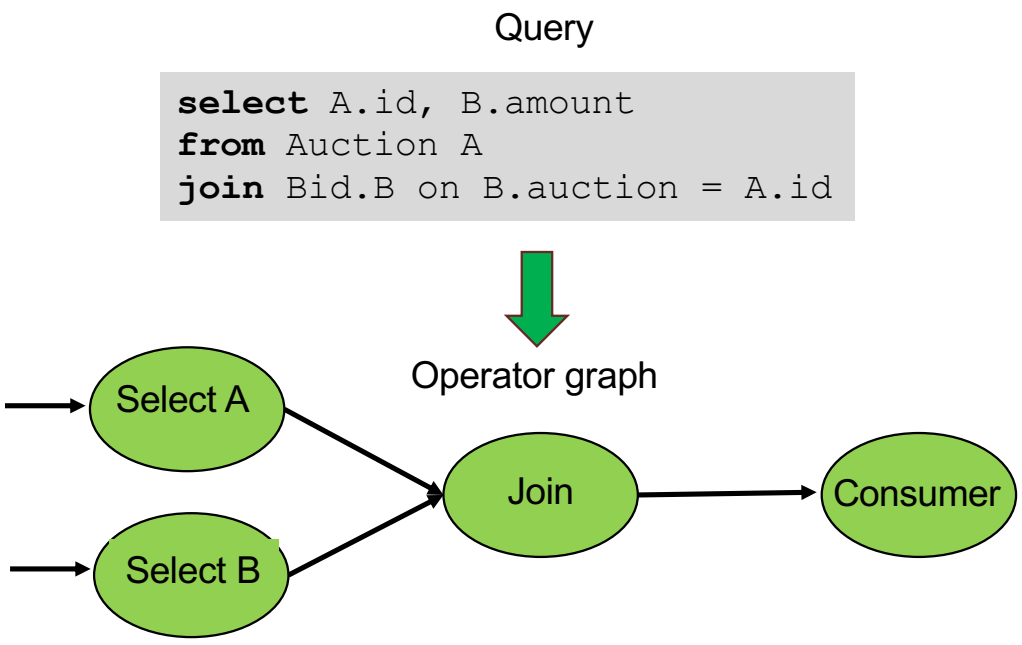
Data stream processing

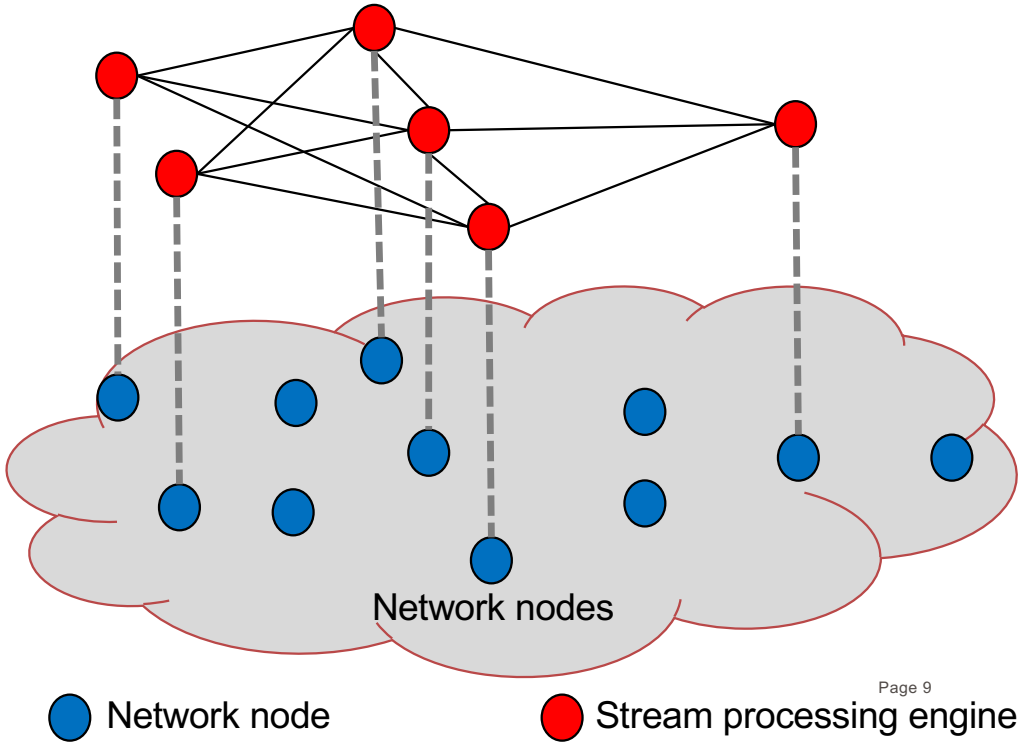


[Source: https://dz2cdn1.dzone.com/storage/temp/11630297-batch-stream-processing-apache-flink-real-time-dat.png]

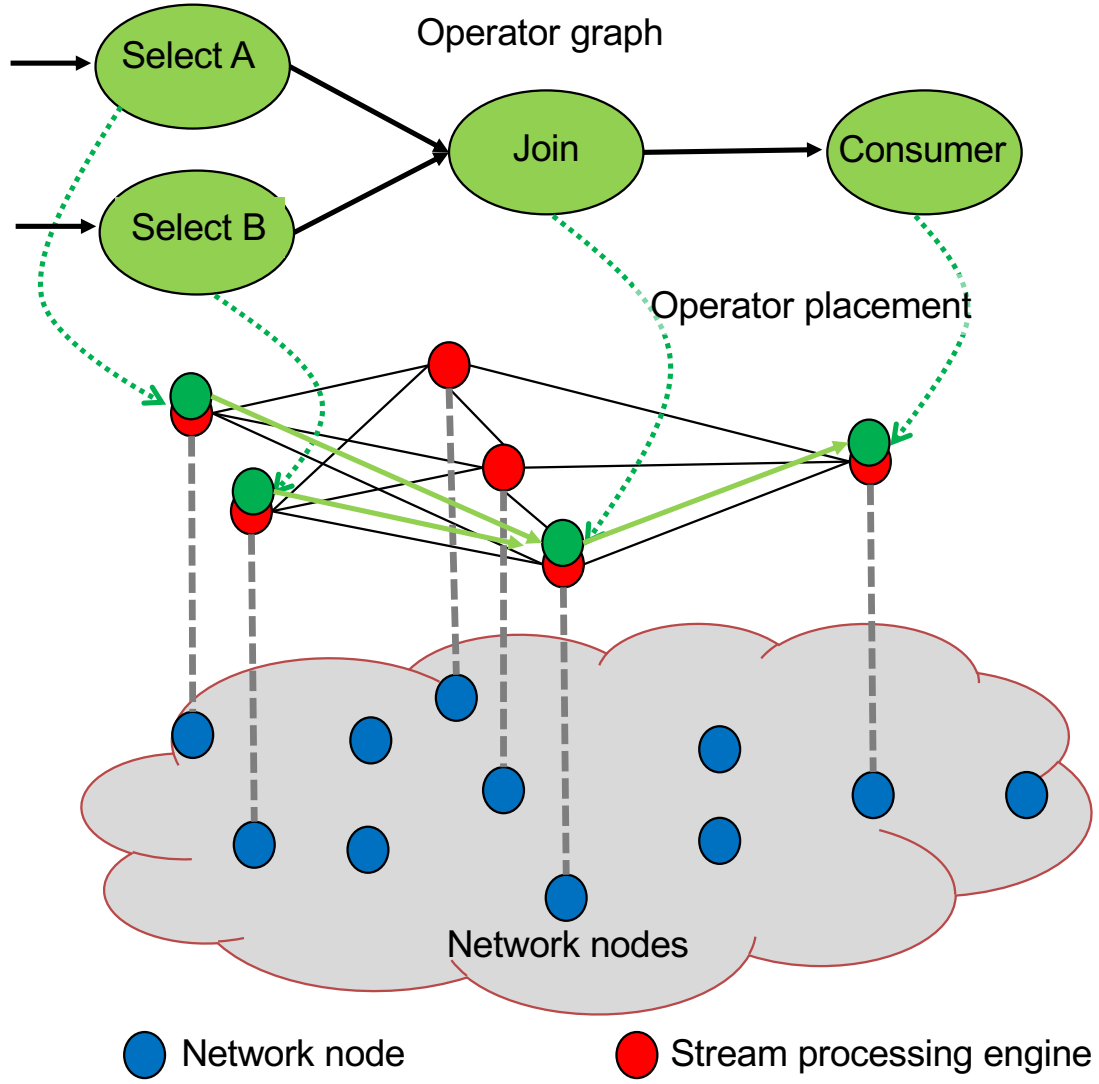
Distributed stream processing

- Distribute the load over multiple networked nodes
- Operator the basic “unit” of distribution
- Query → operator graph

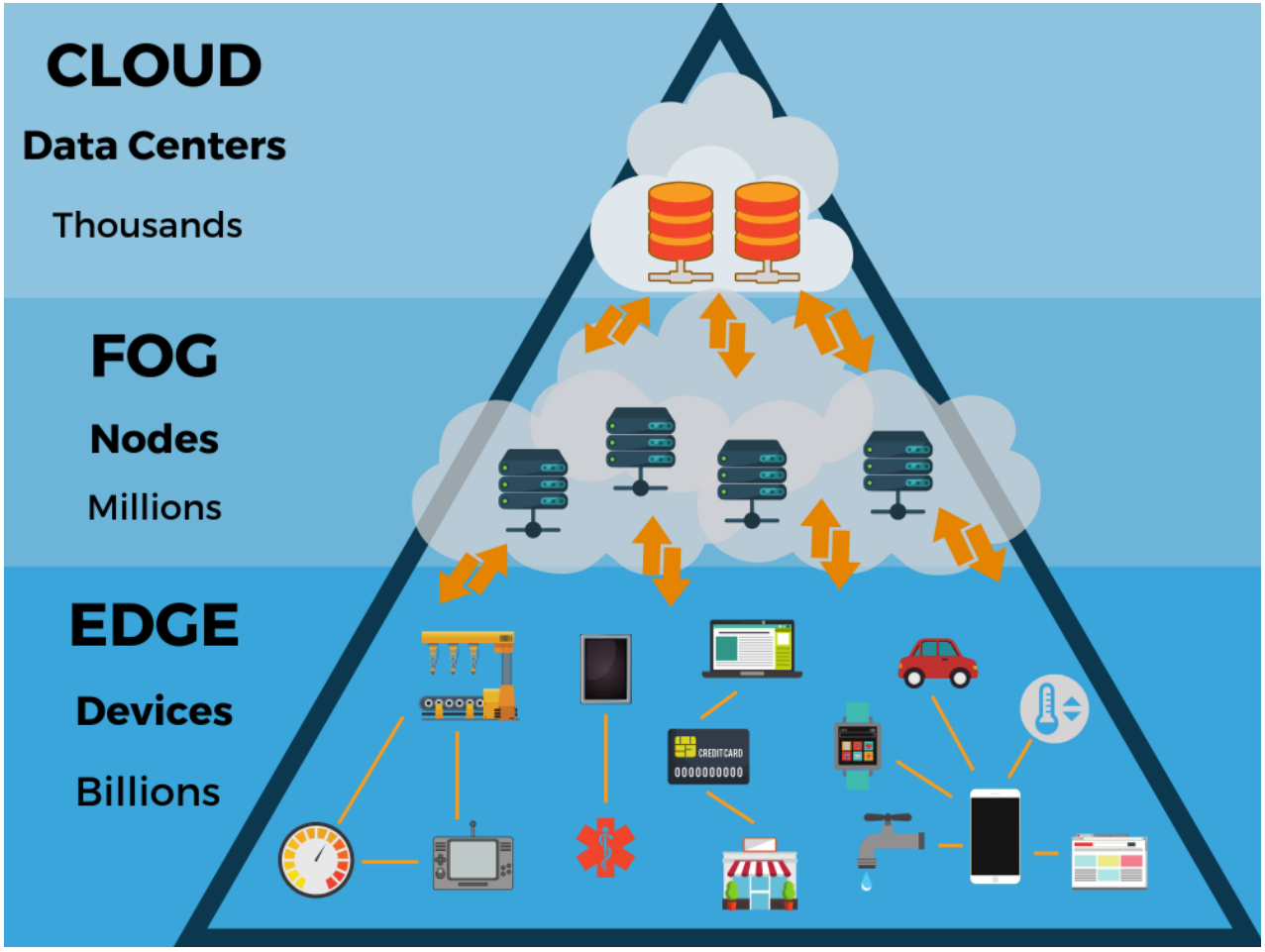




Initial placement



Processing environments



Adaptation

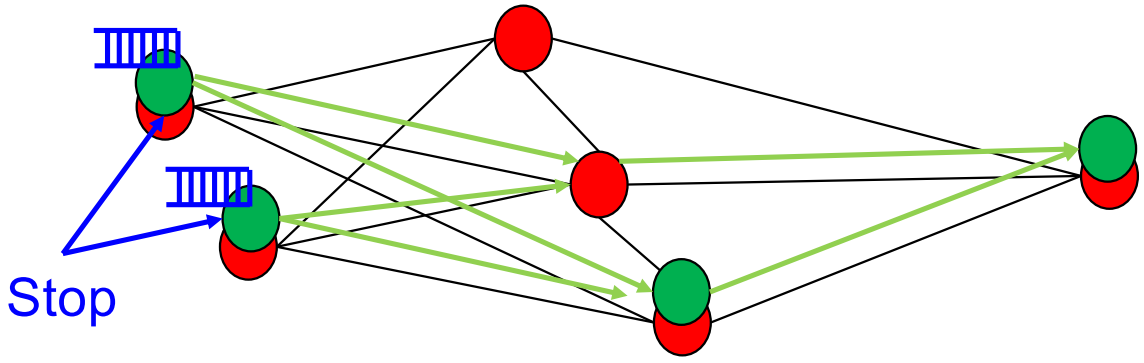
- Queries are long running
- Amount of streaming data can change
- Load of operator hosts can change
- Network QoS can change
- Mobile nodes can change the location

⇒ Adapt distributed stream processing overlay through operator migration

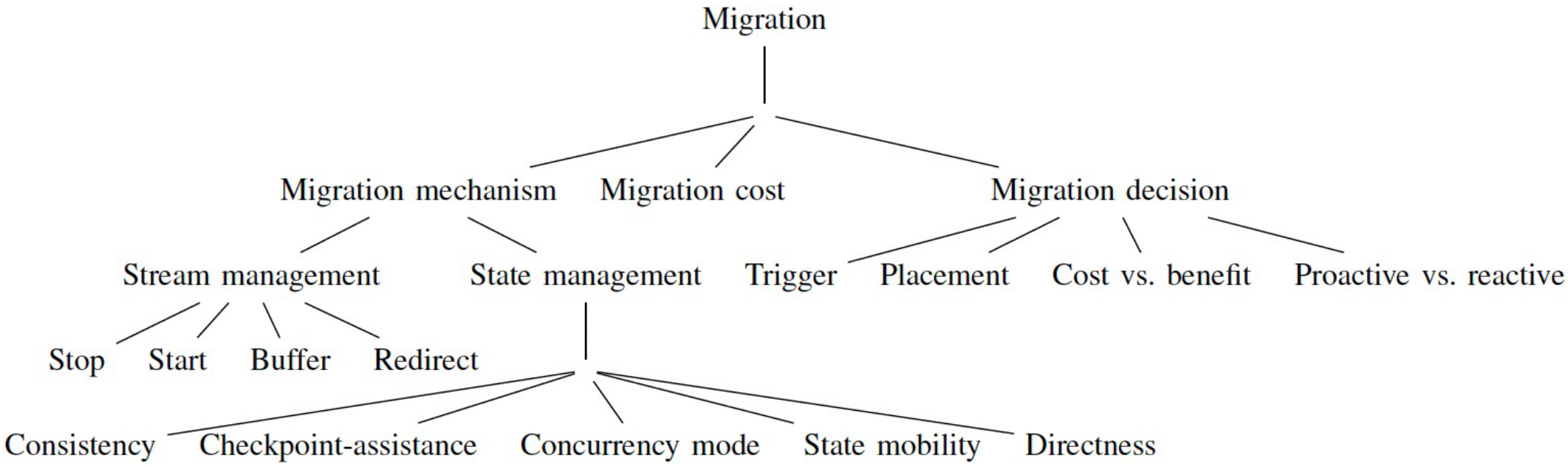
⇒ Load balancing/elasticity, fault tolerance, Quality of Service

Basic steps of all-at-once operator migration

- Stop streams from upstream nodes
- Buffer tuples at upstream nodes
- Extract operator state at old host
- Transfer operator state to new host
- Redirect streams
- Start streaming of tuples from upstream nodes



Concepts of migration



Espen Volnes, Thomas Plagemann, and Vera Goebel. 2023. To Migrate or not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing. IEEE Communications Surveys & Tutorials, Volume: 26, Issue: 1, Firstquarter 2024

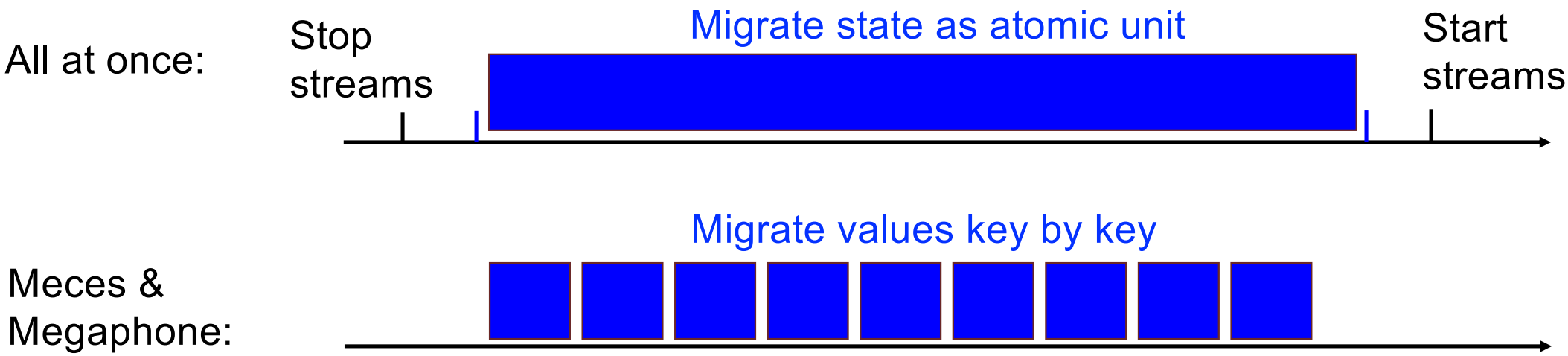
Operator state

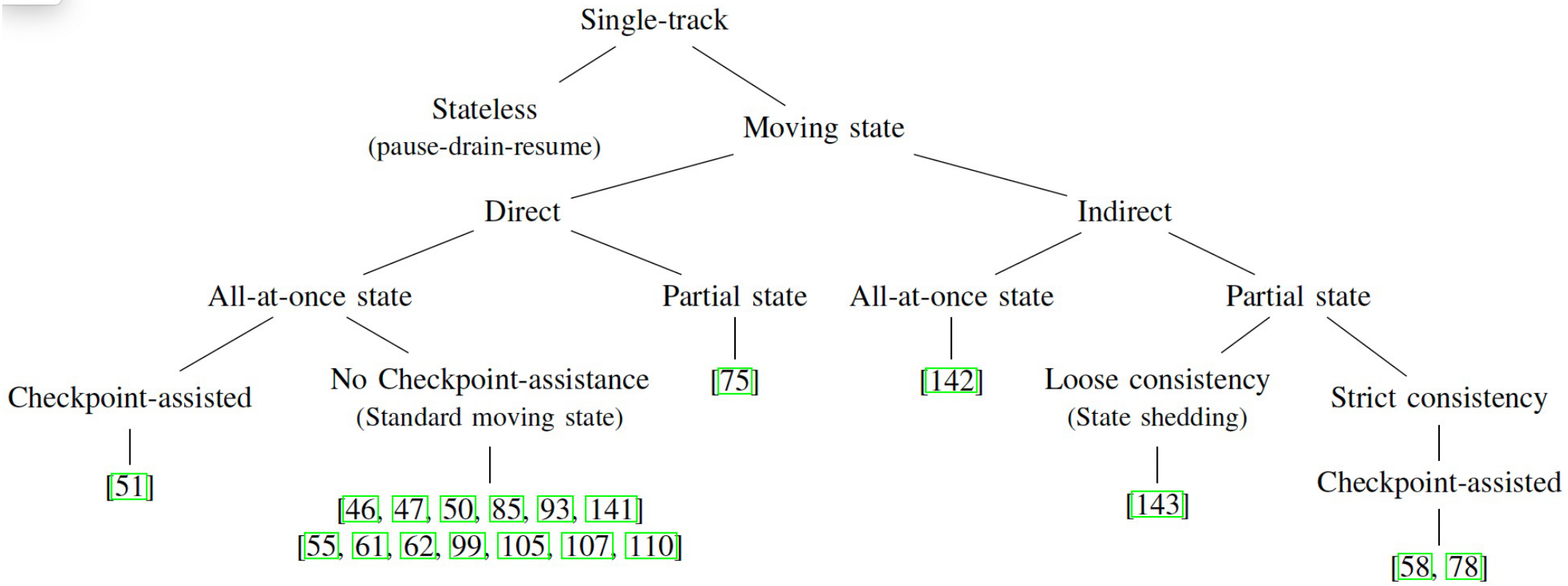
- Processing State: historical data necessary for processing incoming tuples
- Buffer State: holds output tuples that are not yet forwarded downstream
- Routing State: next operator for output tuples

- Examples of processing state:
  - Sum: all tuples that have arrived in a window or sum of all tuples that have arrived so far in the window or partial aggregate
  - Join: all tuples that have arrived in a window so far

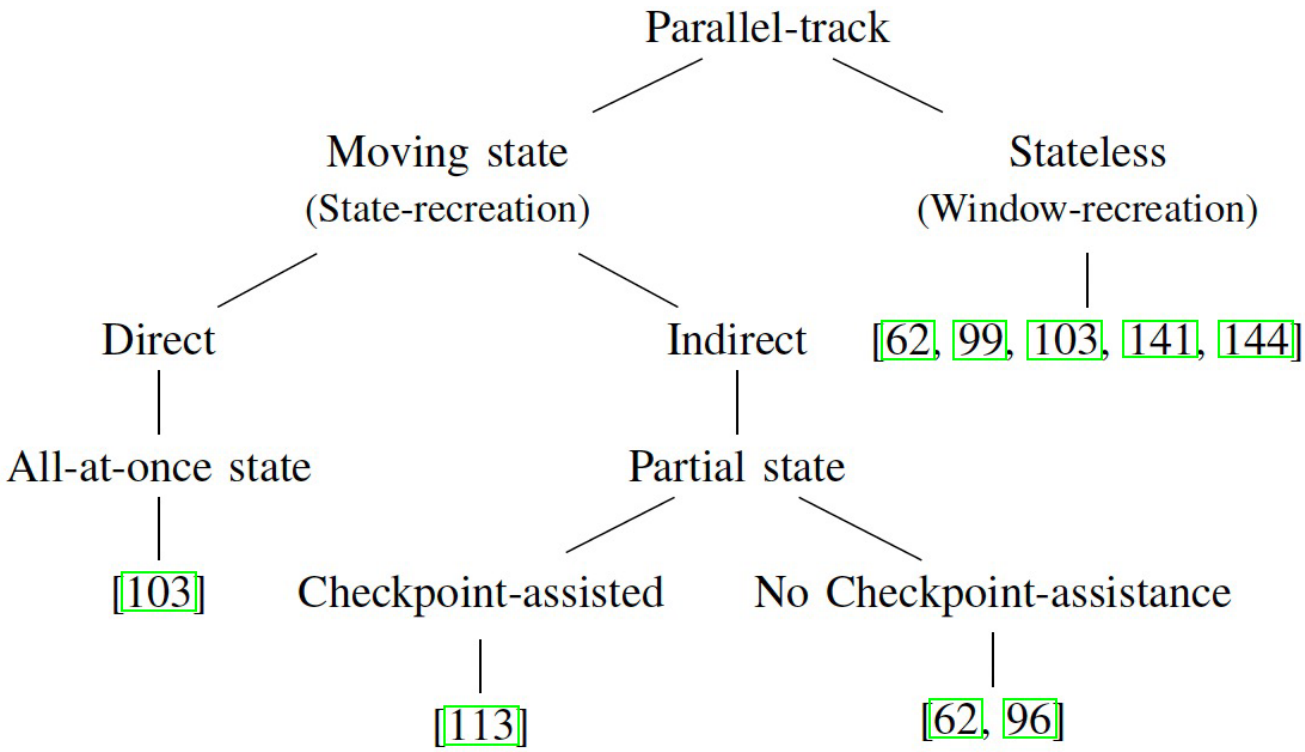
- In modern systems, like Apache Flink, state is managed in key-value stores

Timeline operator migration mechanism (simplified)





Espen Volnes, Thomas Plagemann, and Vera Goebel. 2023. To Migrate or not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing. IEEE Communications Surveys & Tutorials, Volume: 26, Issue: 1, Firstquarter 2024



Espen Volnes, Thomas Plagemann, and Vera Goebel. 2023. To Migrate or not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing. IEEE Communications Surveys & Tutorials, Volume: 26, Issue: 1, Firstquarter 2024

Trend in modern solutions & core ideas

- Recent solutions: state is not an atomic unit
  - Static key – value partitions
- Core ideas of Lazy Migration:
  - Fine granular static and dynamic state partitions
  - Semantic awareness: understand which part of the state is needed when
  - Utility awareness: what are the most important parts of the state

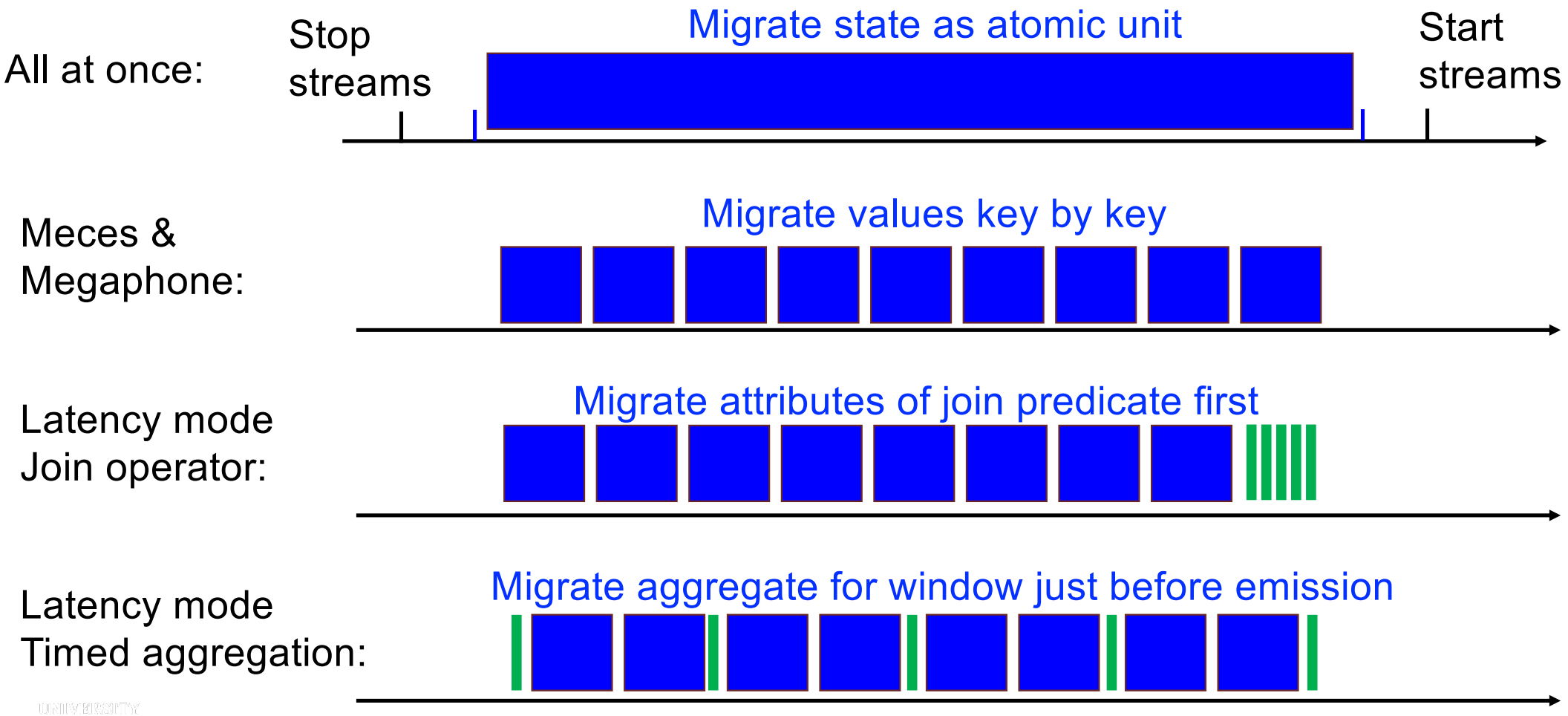
Perspective	State units	Consideration
Partial state	Tuple in join operator (op.)	Selectivity
	Sequence in pattern matching op.	Selectivity, Minimum number of events before match
	Tuple in aggregation op.	Effect on aggregated results, Emission times
	Partial window in aggregation op.	Emission time
Key-Value	Key Value	Number of Key-Value pairs, Size of each Key-Value pair
All-at-once	Entire state	State size

Lazy migration: core approach

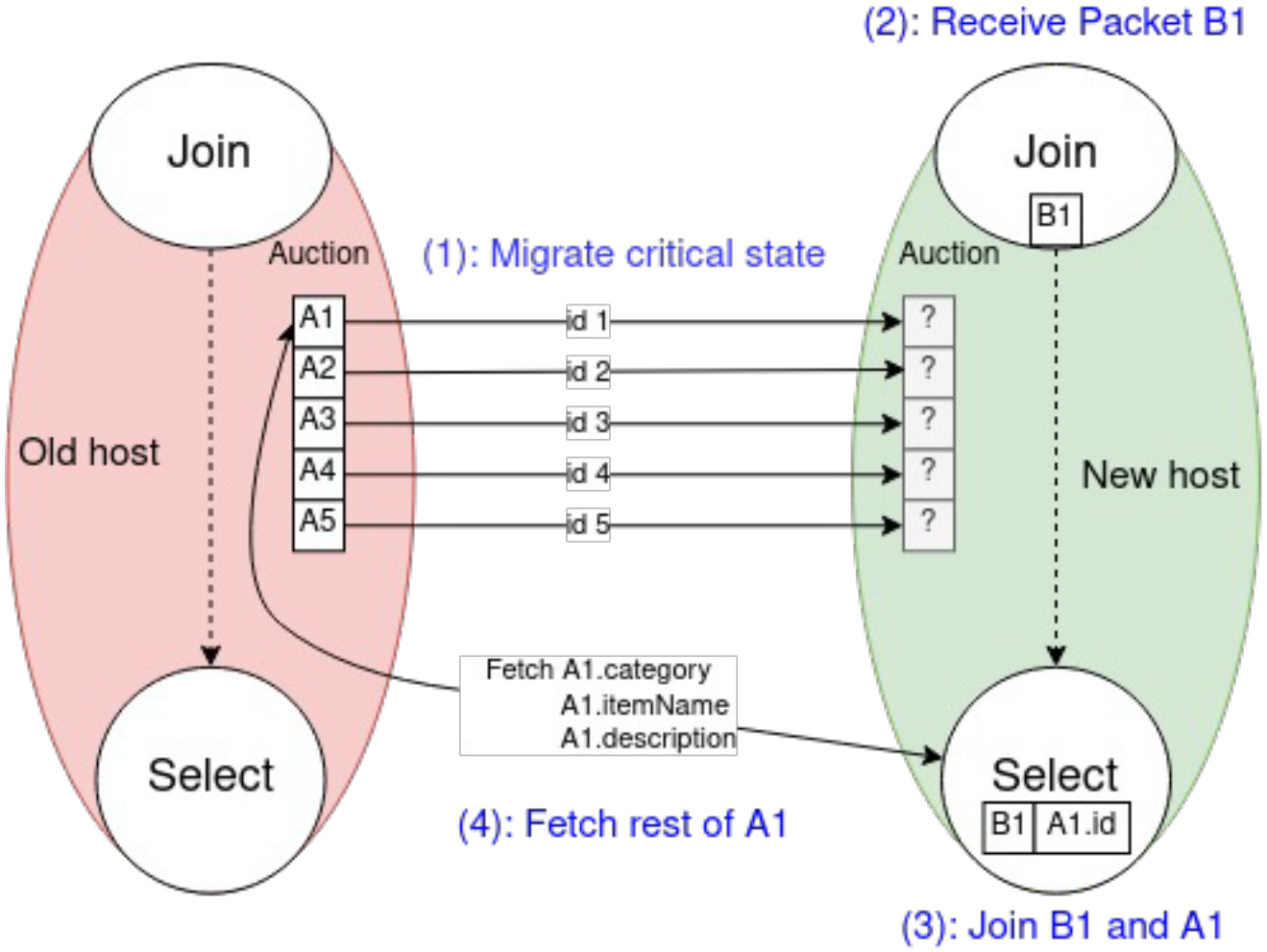
- Fine granular state partition, static & dynamic
- Transfer state partitions to the new host just before they are needed
- State partitions that are essential for processing have higher priority
- Latency mode:
  - Dynamic state partition
  - Timely scheduling
- Utility mode:
  - Essential and unessential attributes
  - User defined utility functions

Espen Volnes, Thomas Plagemann, Boris Koldehofe, and Vera Goebel. 2022. Travel light: state shedding for efficient operator migration. In Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems (DEBS '22)

Timeline operator migration mechanism (simplified)

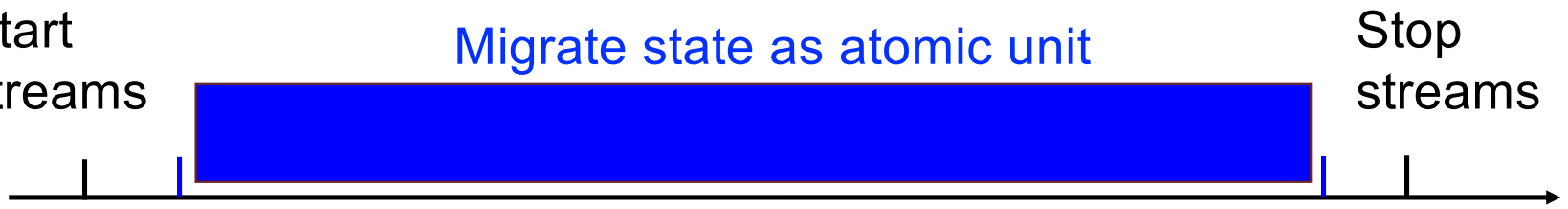


Utility mode – join operator

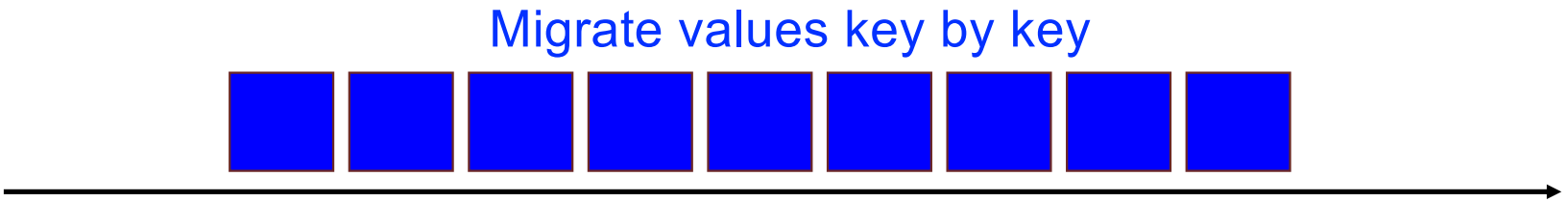


Timeline operator migration mechanism (simplified)

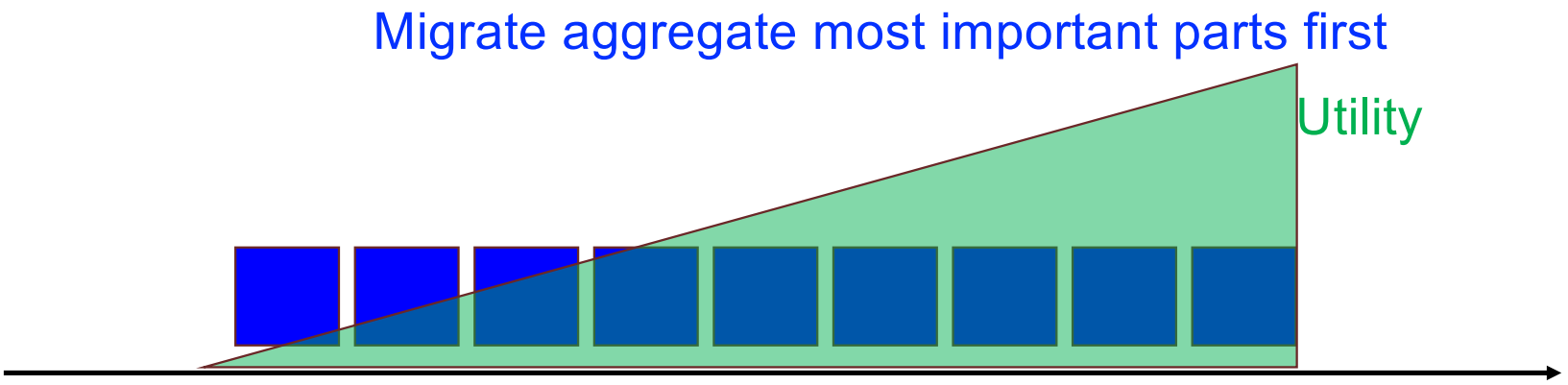
All at once:



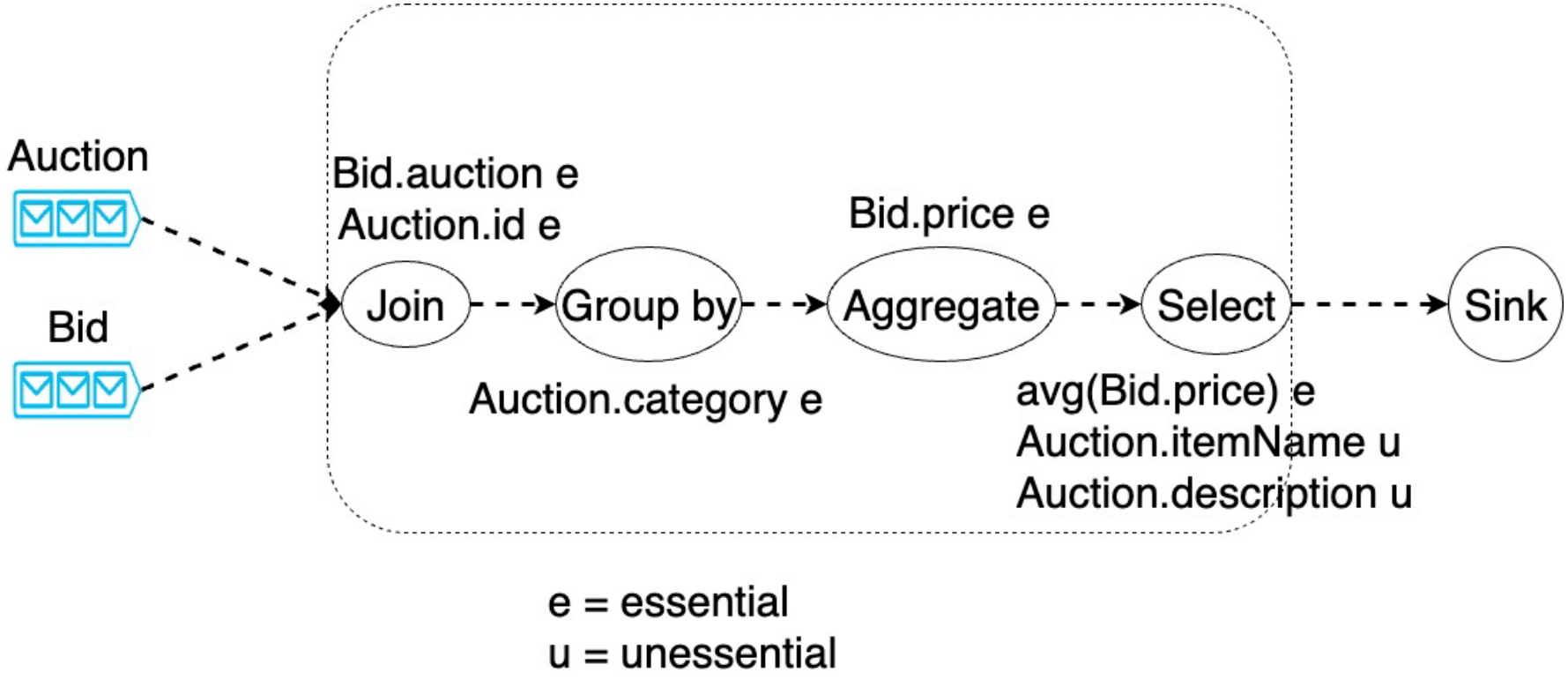
Meces &
Megaphone:



Lazy Migration
Utility mode:



Operator graph with essential and unessential attributes



How to evaluate?

- Compare performance with state-of-the art systems and baseline: all-at-once, Meces, Rhino, Megaphone
- Not all systems are available
- How to isolate the operator migration overhead from the rest of the system

⇒ Implement all operator migration mechanism in the same system

- Apache Flink???
  - Core idea of Lazy Migration is based on fine grained state partitions
  - Not (yet) supported in Flink

How to evaluate? (cont.)

- Fair comparison requires repeatable experiments
- All solutions must be evaluated under exactly the same conditions
- Hard to guarantee in real-world experiments

⇒ Implementation and simulation in DCEP-Sim

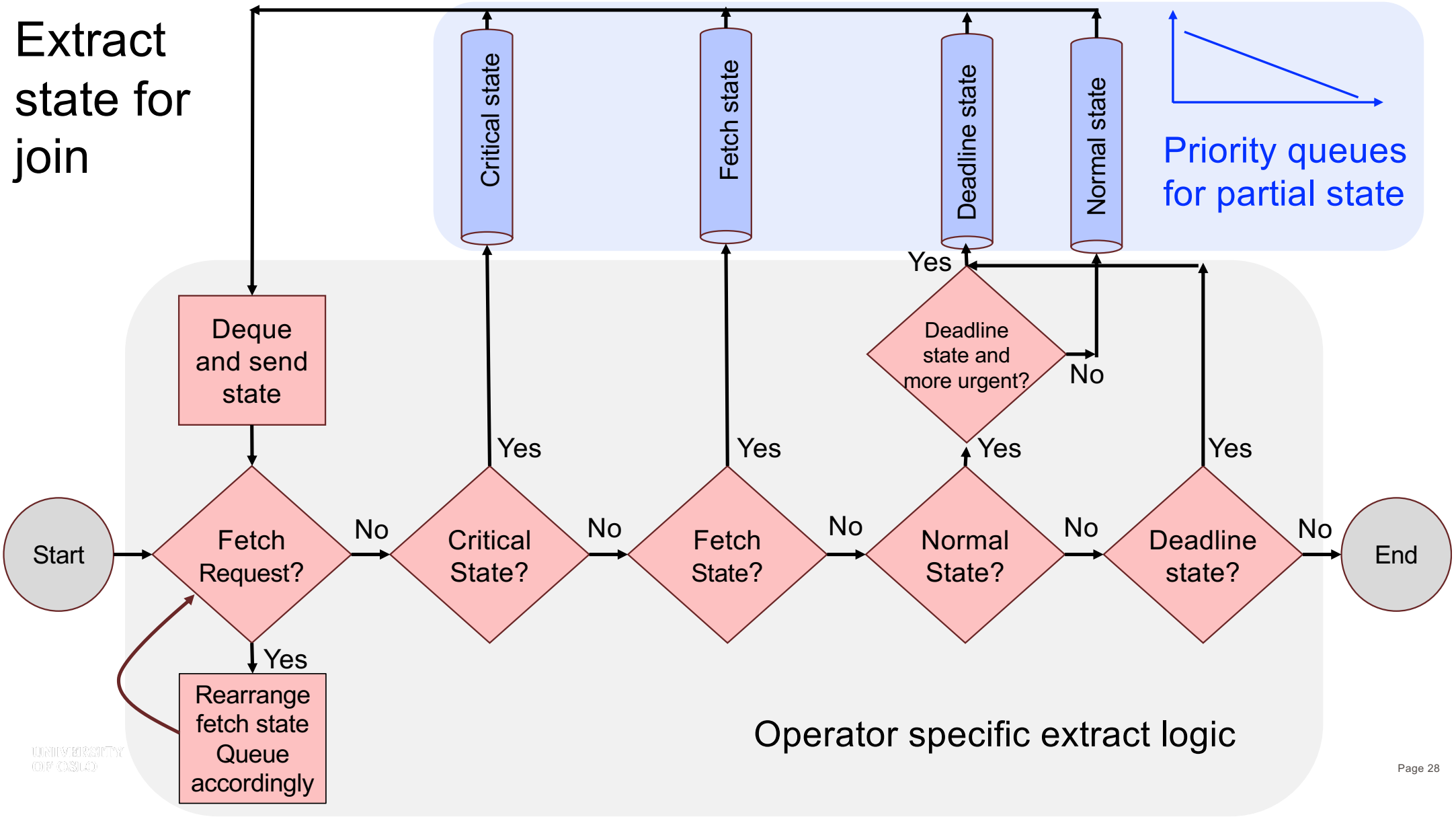
- DCEP-Sim is a distributed stream processing system in ns-3
  - Accurate models for communication behaviour exist
  - Models for processing delay
  - Migration mechanism framework

Migration mechanism framework

- Import state
- Extract state and place operator state partitions into priority queues
  - Critical state queue: if this state is not available the operator on the new host blocks
  - Fetch state queue: increase priority when it is necessary
  - Normal state queue: lowest priority
  - Deadline state queue: partitions that need to be delivered at a certain deadline
- Schedule partitions according to priority queues and deadlines

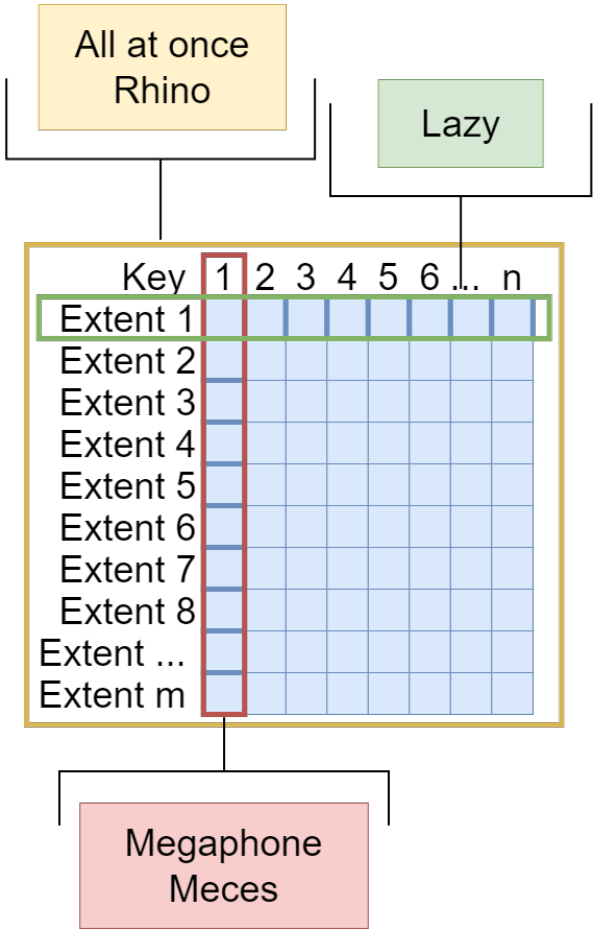
- The framework facilitated the implementation of all migration mechanisms

Extract state for join

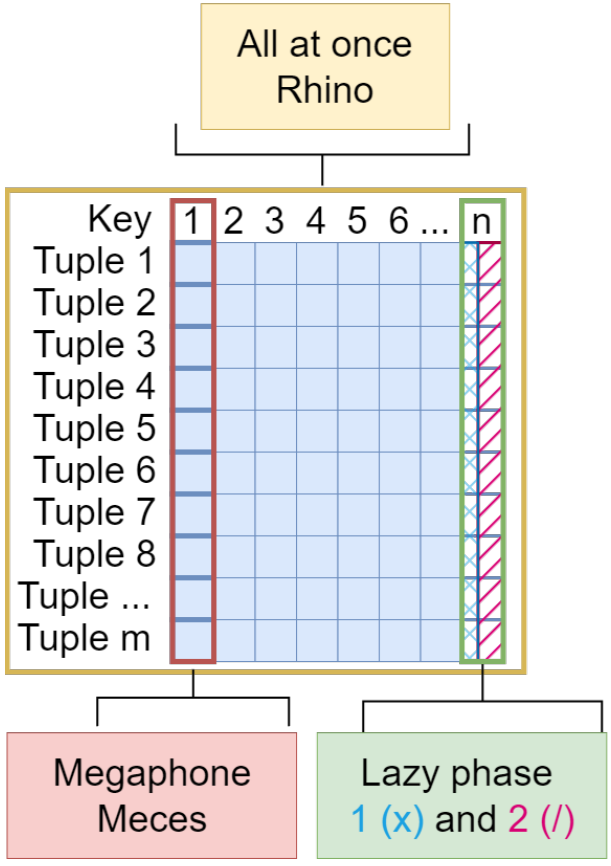


State movement priority for stateful operators

Timed aggregation



Join



Workload: NextMark benchmark

Join:

```
select A.itemName, B.price
from Auction A
join Bid B
on A.id > B.auction - 10
and A.id < B.auction + 10
```

Timed aggregartion:

```
select avg(B.price)
from Bid B
group by B.auction, hop(100 seconds, 1 second)
```

Join followed by timed aggregation:

```
select A.itemName, avg(B.price)
from Auction A
join Bid B
on A.id > B.auction - 10
and A.id < B.auction + 10
group by B.auction, hop(100 seconds, 1 second)
```

10.000 Auction tuples
1000.000 Bid tuples
Network bandwidth: 10 MB/s

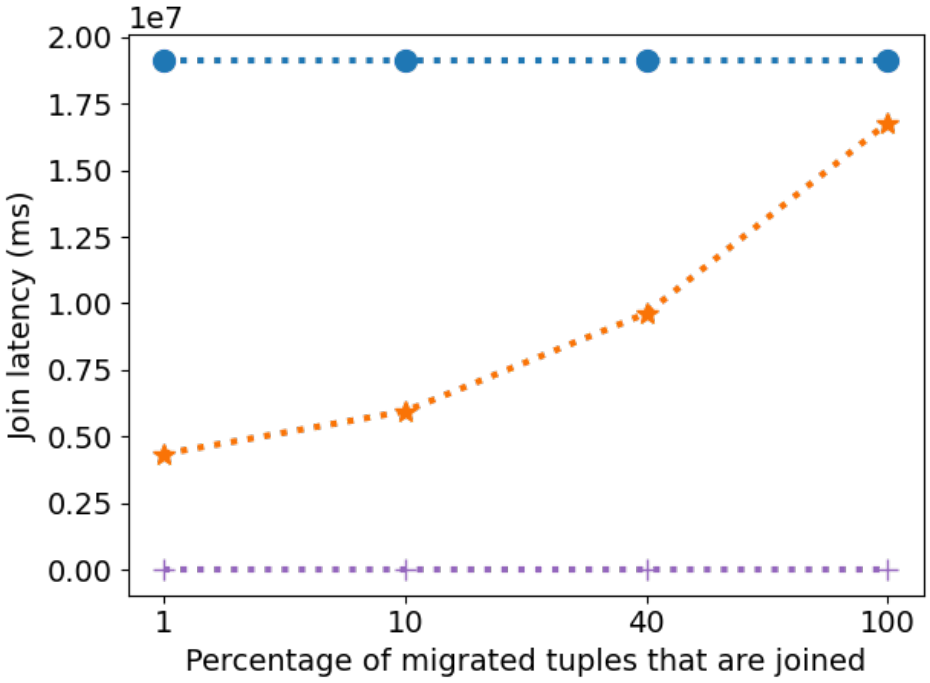
Metrics

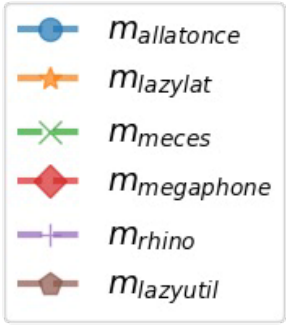
- Join latency: comparing join time with and without migration
- Aggregation latency: comparing aggregation time with and without migration
- Emission delay: how long after emission deadline a tuple is emitted
- Missed window extents: tuples that could not be processed in the window they should
- Utility: number of tuples that are joined

- Parameters:
  - Join selectivity: 1%, 10%, 40%, 100%
  - Window jump: 200 ms to 10.000 ms in ten steps
  - Operational migration period (OMP): 1ms, 1000 ms, 2000ms, …, 15.000ms

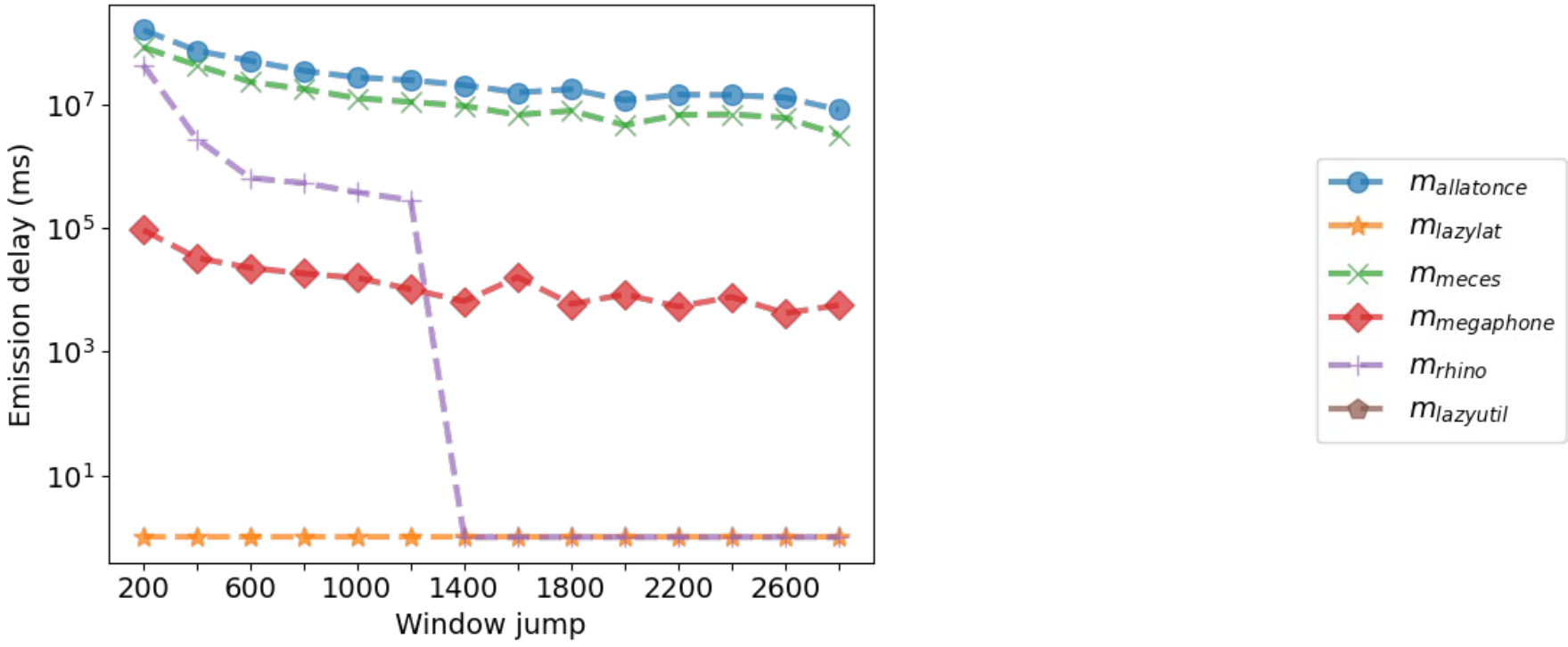
Experiment 1: Join latency for non-equijoin

migrated tuples that are joined: 1%, 10%, 40%, 100%

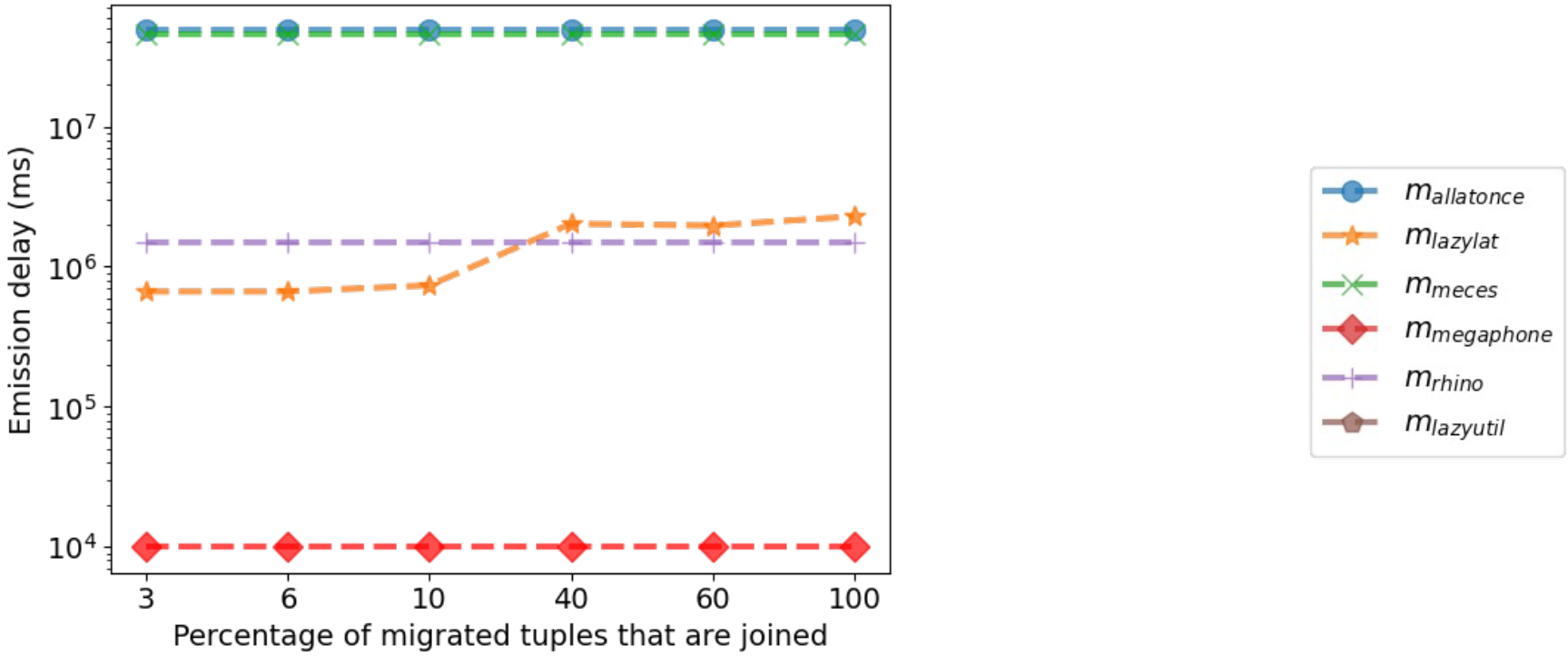




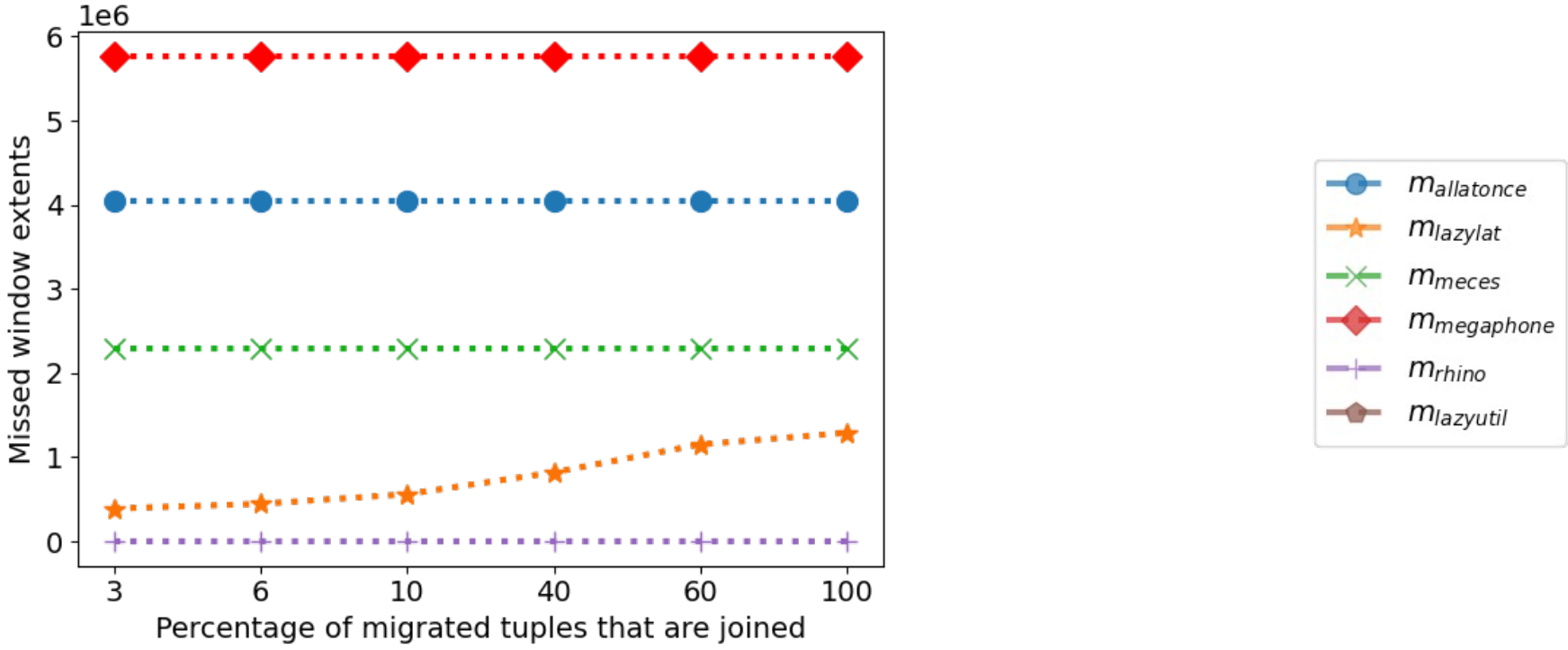
Experiment 2: Emission delay for timed aggregation



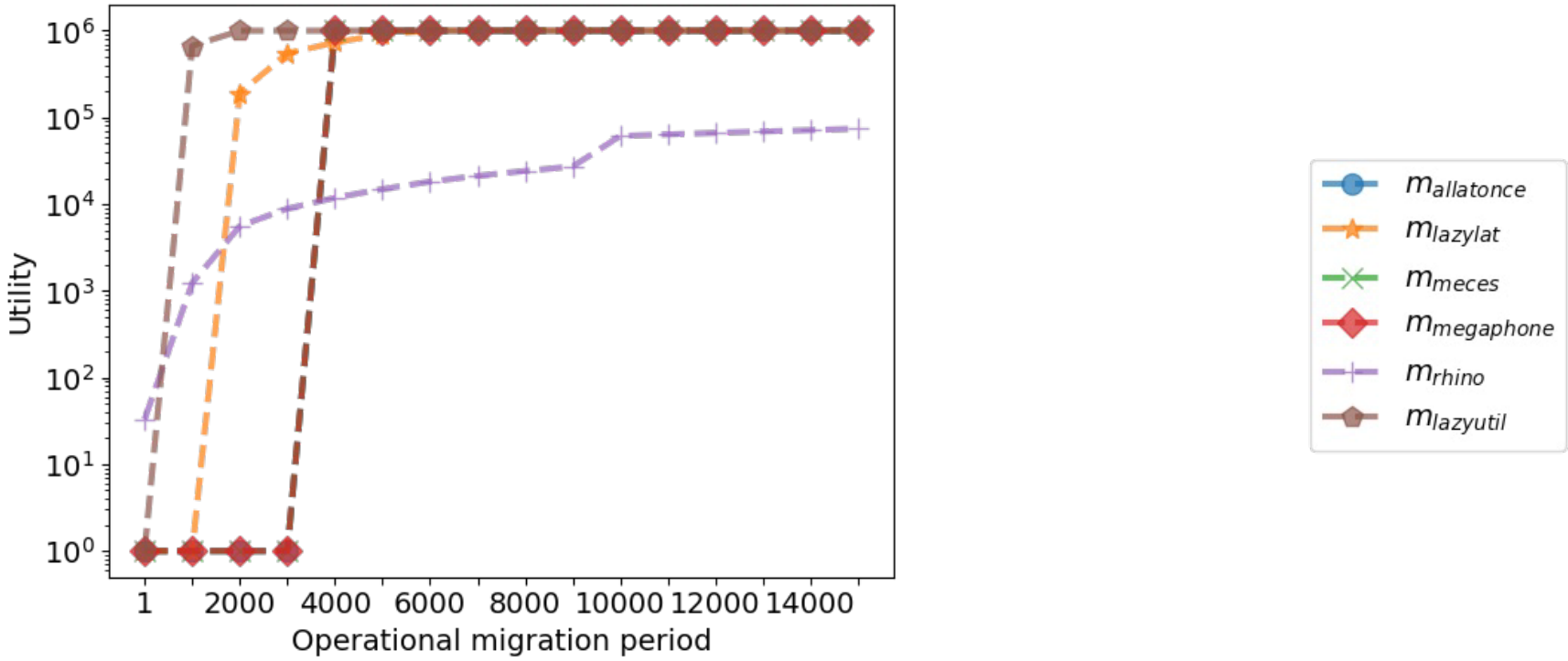
Experiment 3: Emission delay for join → timed aggregation



Experiment 3: Missed window extents for join → timed aggregation



Experiment 5: Utility for non-equijoin and different OMPs



Conclusions

- Main contributions:
  - Latency mode:
    - Superior with non-equijoin and sliding window timed aggregation
    - For word count and equijoin Meces and Megaphone are better optimized
  - Migration that is interrupted:
    - Utility mode superior
    - Latency mode works also well
  - Migration Framework

- How useful is an evaluation based on DCEP-Sim

Questions?

UNI
OF