Beyond Analytics

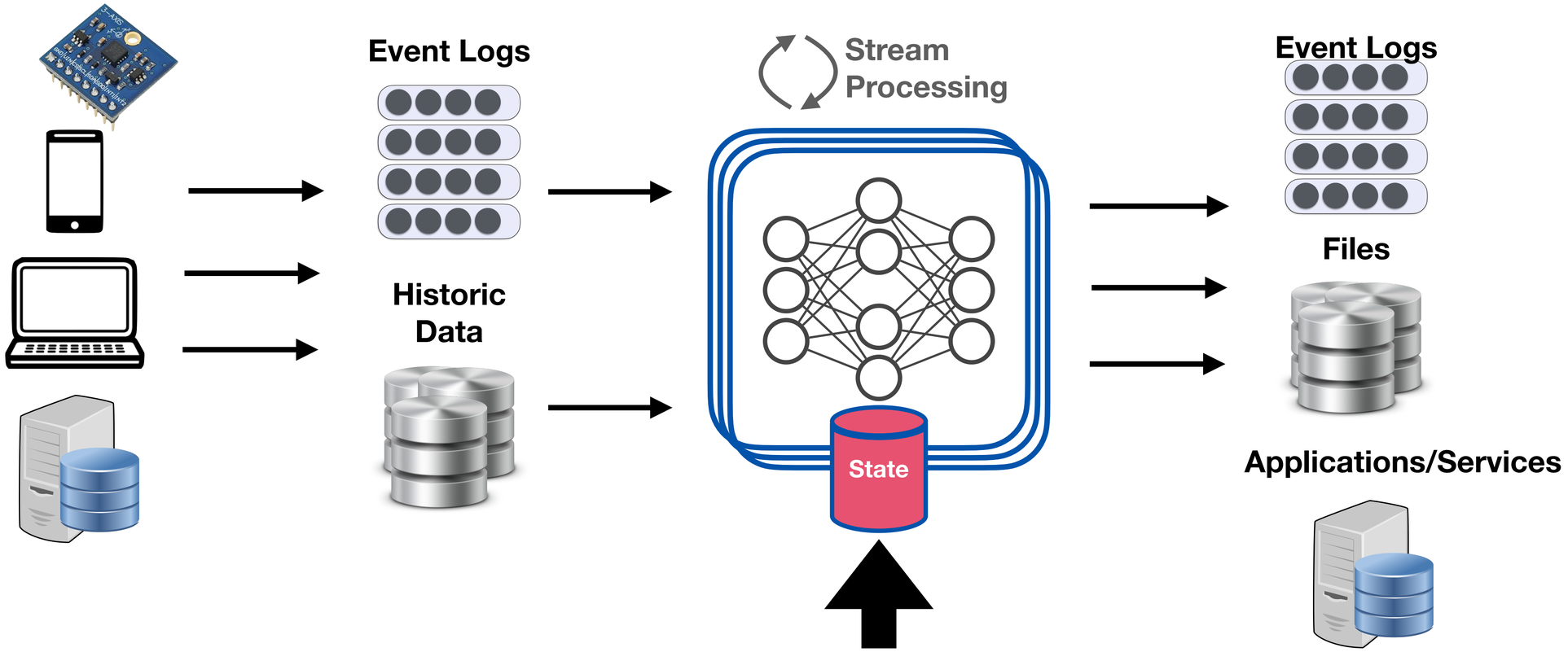
The Evolution of Stream Processing Systems

State Management

Paris Carbone, Marios Fragkoulis, Vasia Kalavri, Asterios Katsifodimos

RI.SE BOSTON UNIVERSITY TU Delft

What is State

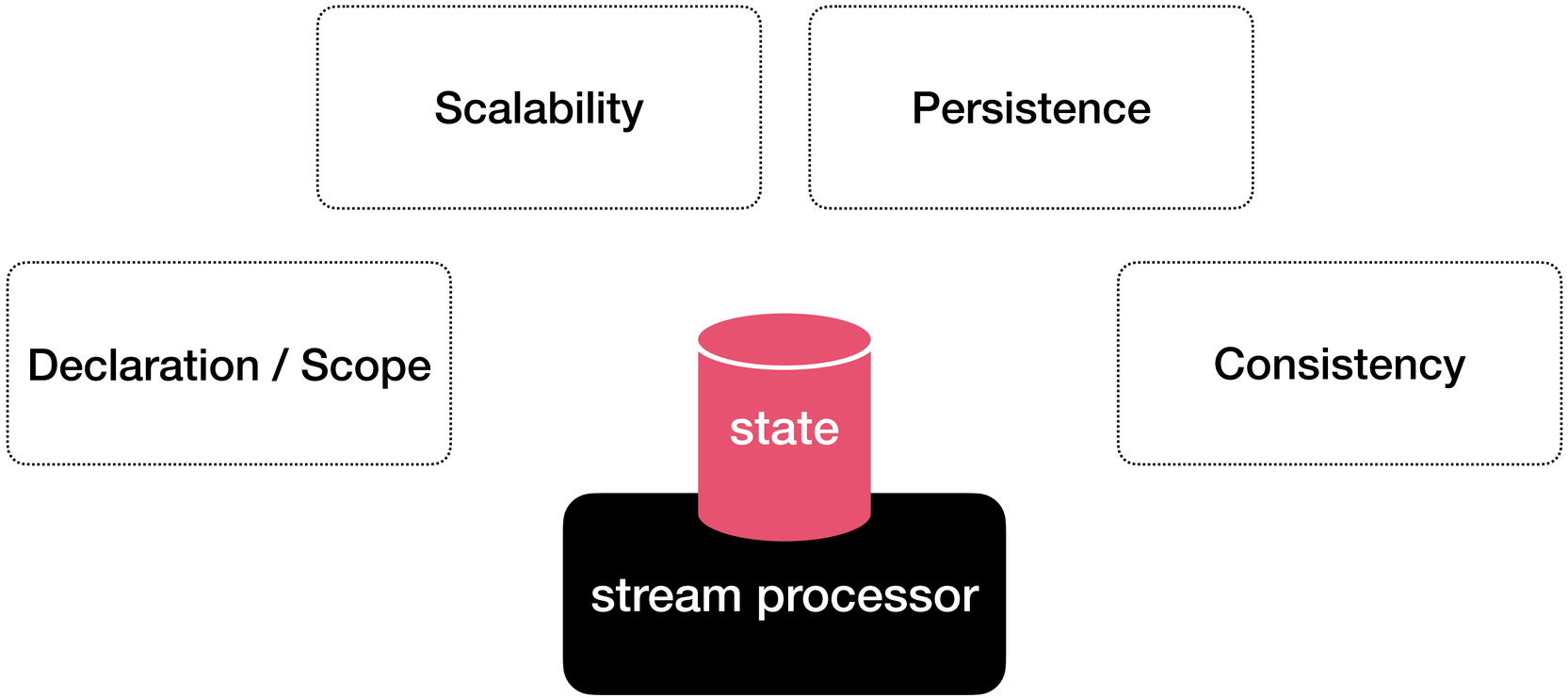


windows

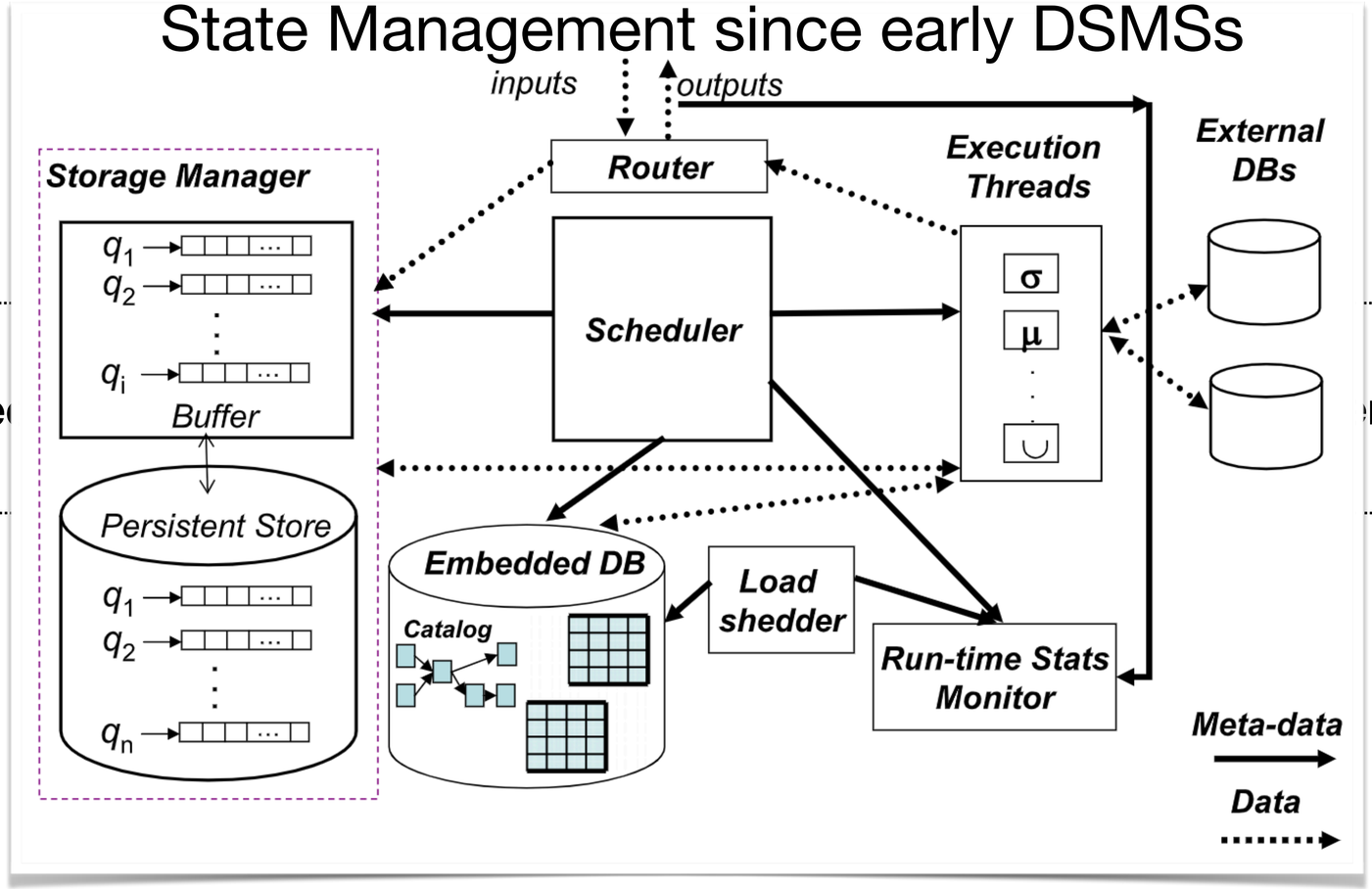
aggregates

user-defined variables

What is Stream State Management



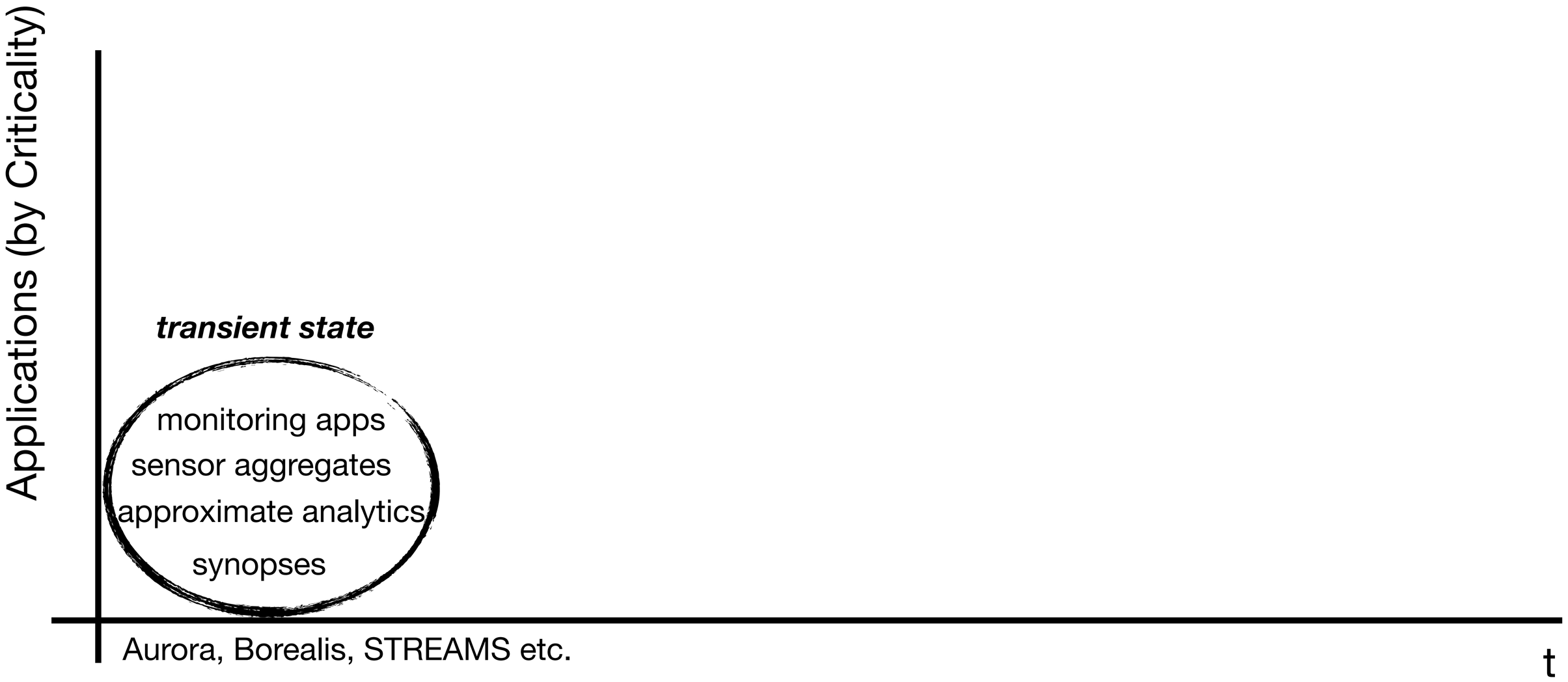
What is Stream State Management



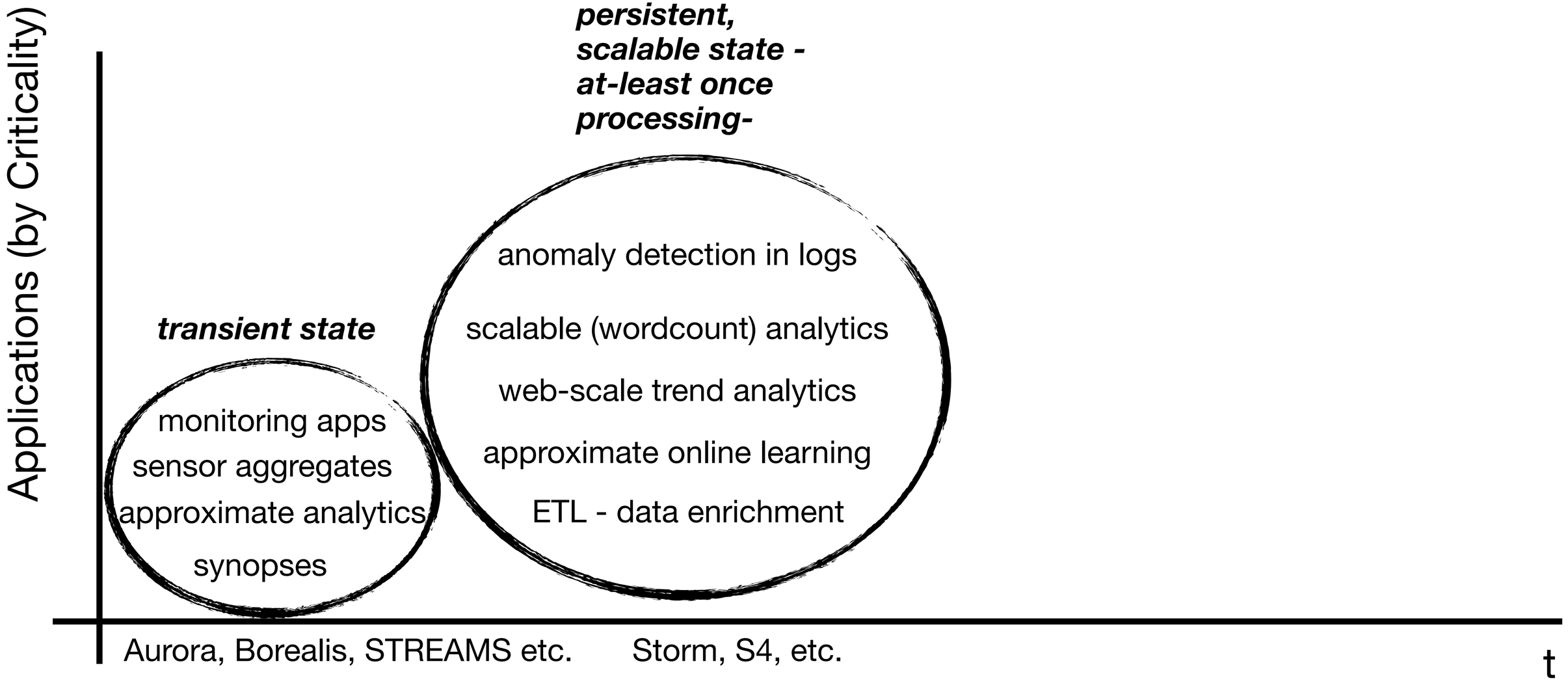
The Aurora and Borealis Stream Processing Engines

Çetintemel U, Abadi D, Ahmad Y, et. al. (2016)

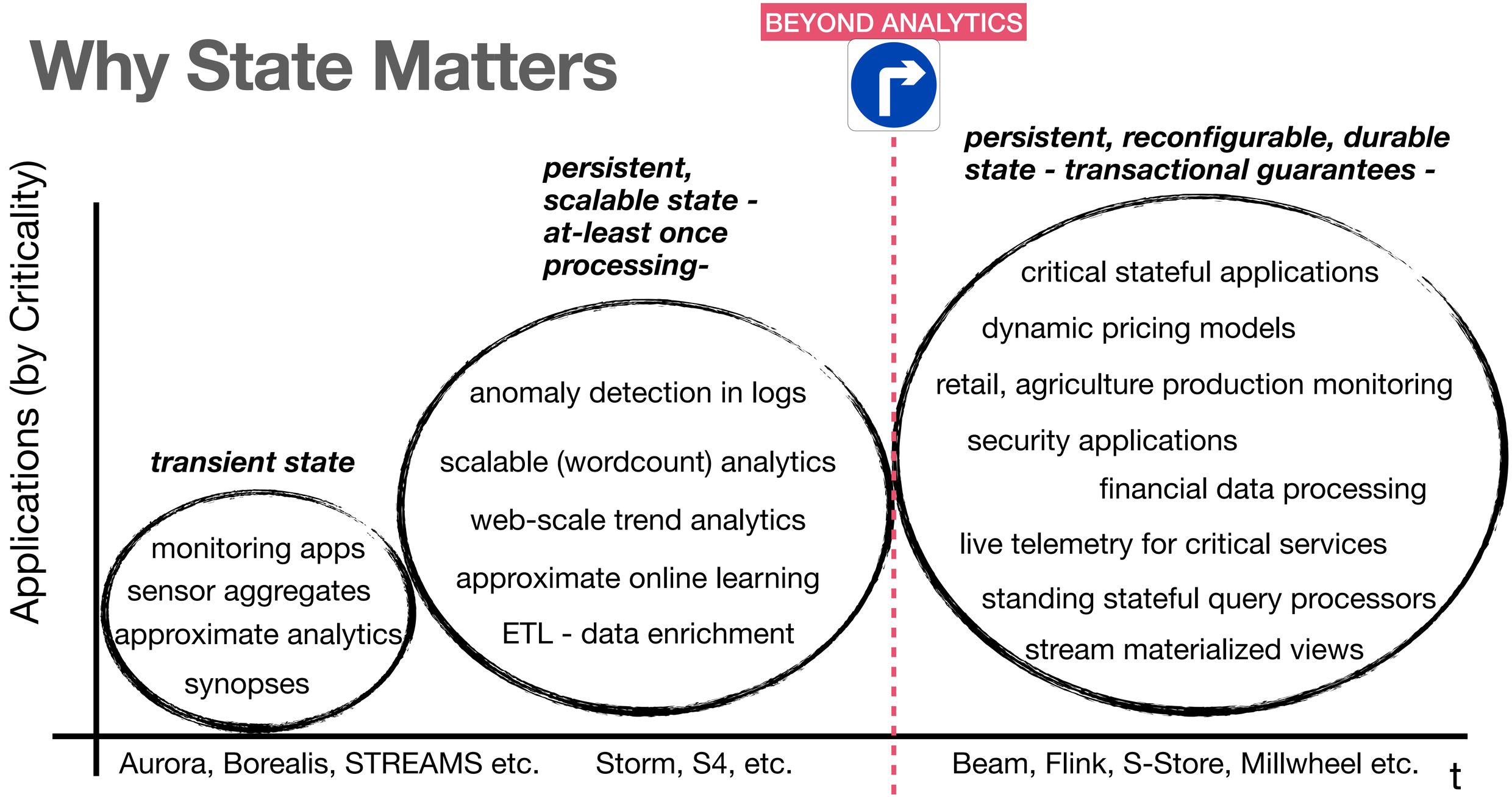
Why State Matters



Why State Matters



Why State Matters

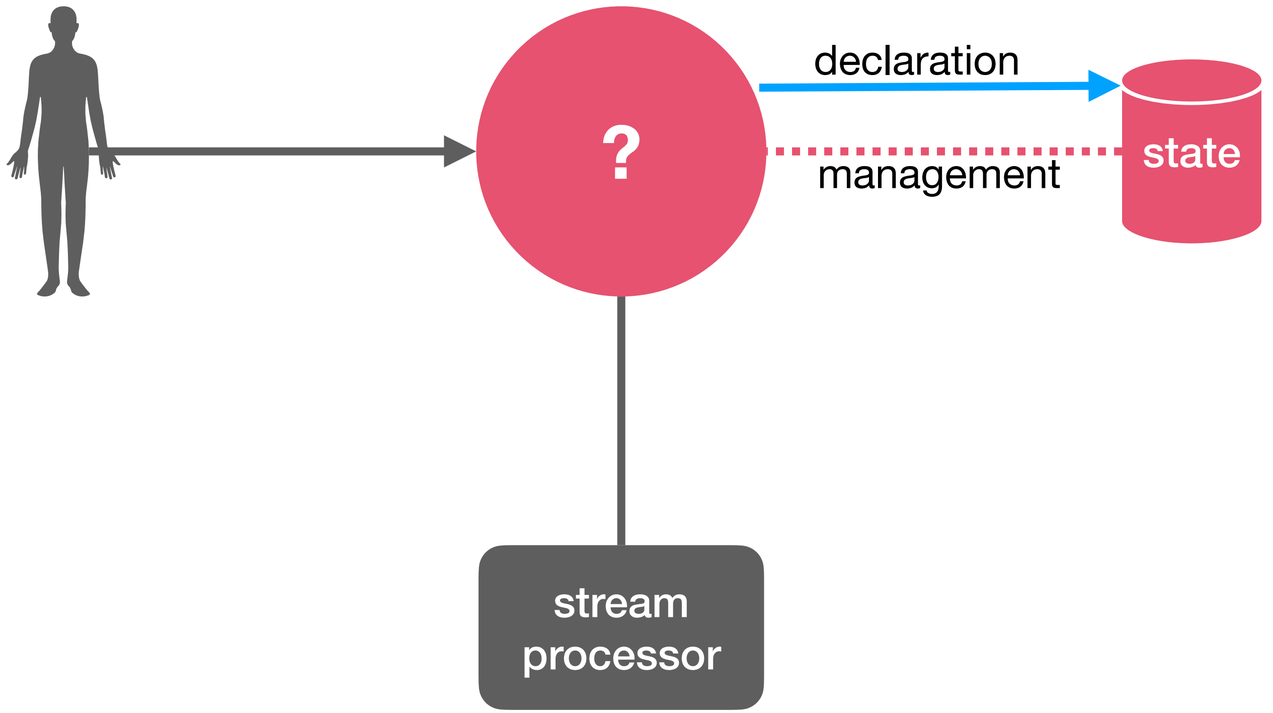


State Management

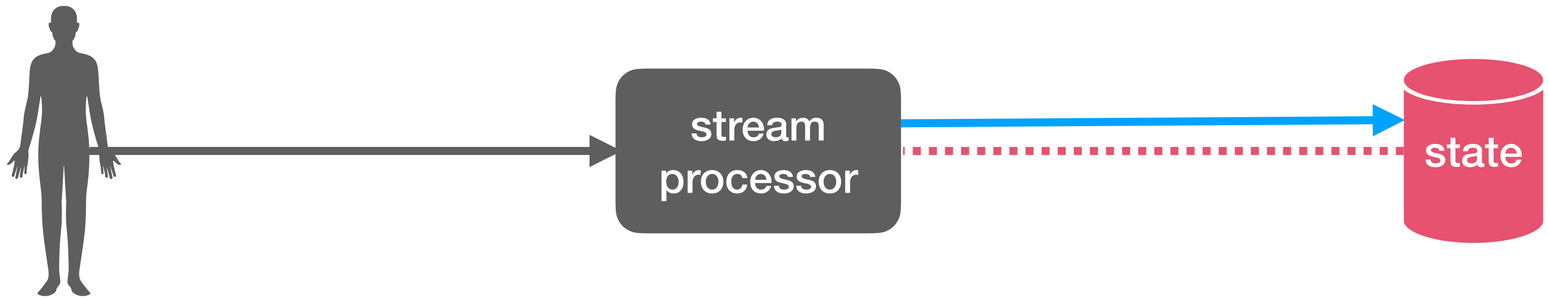
Agenda

1. **Scope** - Synopses, Application- and System-Managed State
2. **Scalability** - Partitioned Task and Key-level State
3. **Persistency** - Locally and Externally Managed State
4. **Consistency** - Guarantees and Transactional Processing
5. Prospects and Summary

1. Scope



State as a Synopsis



"synopsis" : short system-internal summarisation of an input / computation

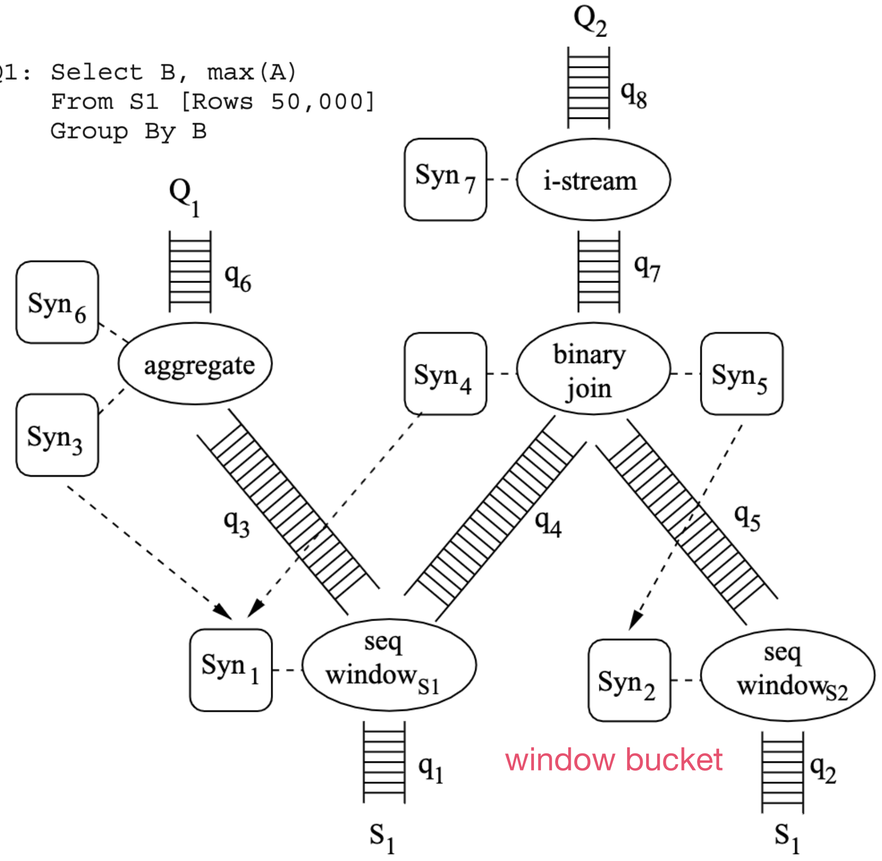
aka system-defined and system-managed state

- A *implicit* form of State that has been an integral component of DSMSs (2000s)
- It facilitates implementation of a set of **operators** (e.g., join, filter, aggregate, sort, window etc.)
- It is usually hidden from the user (System-Defined)
- Often Transient and Memory-bound - **Approximation**

Examples

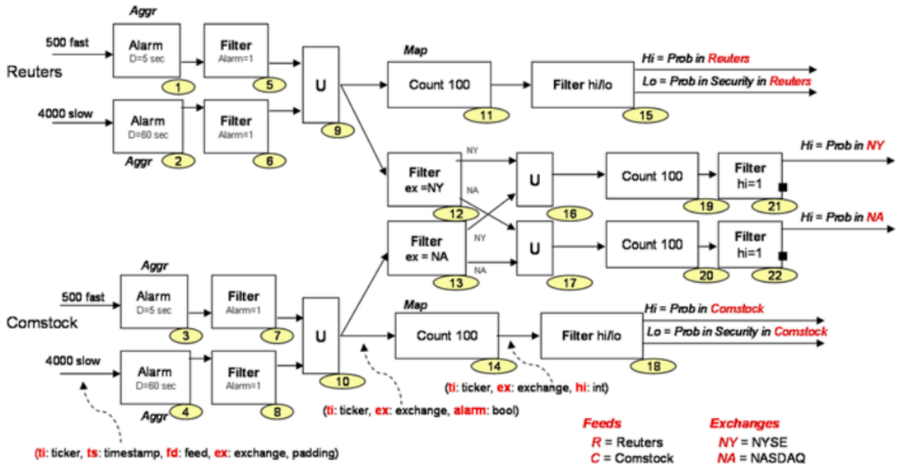
```
Q2: Select Istream(*)
    From   S1 [Rows 40,000],
           S2 [Range 600 Seconds]
    Where  S1.A = S2.A
```

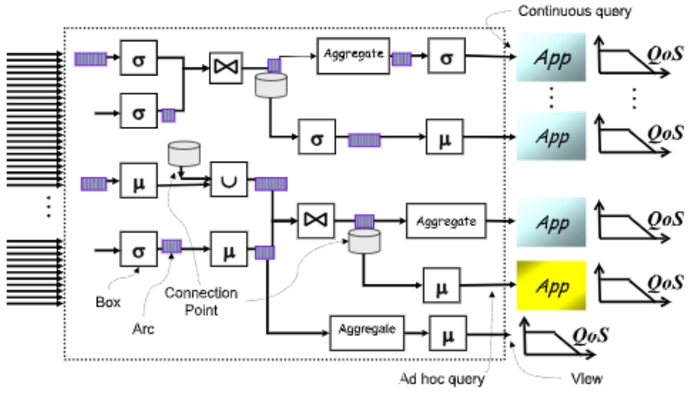
```
Q1: Select B, max(A)
    From S1 [Rows 50,000]
    Group By B
```



The CQL continuous query language: Semantic foundations and query execution
Arasu A, Babu S, Widom J in VLDB Journal (2006)

```
Aggregate(Group by ticker,
          Order on arrival,
          Window (Size = 2 tuples,
                  Step = 1 tuple,
                  Timeout = 5 sec))
```



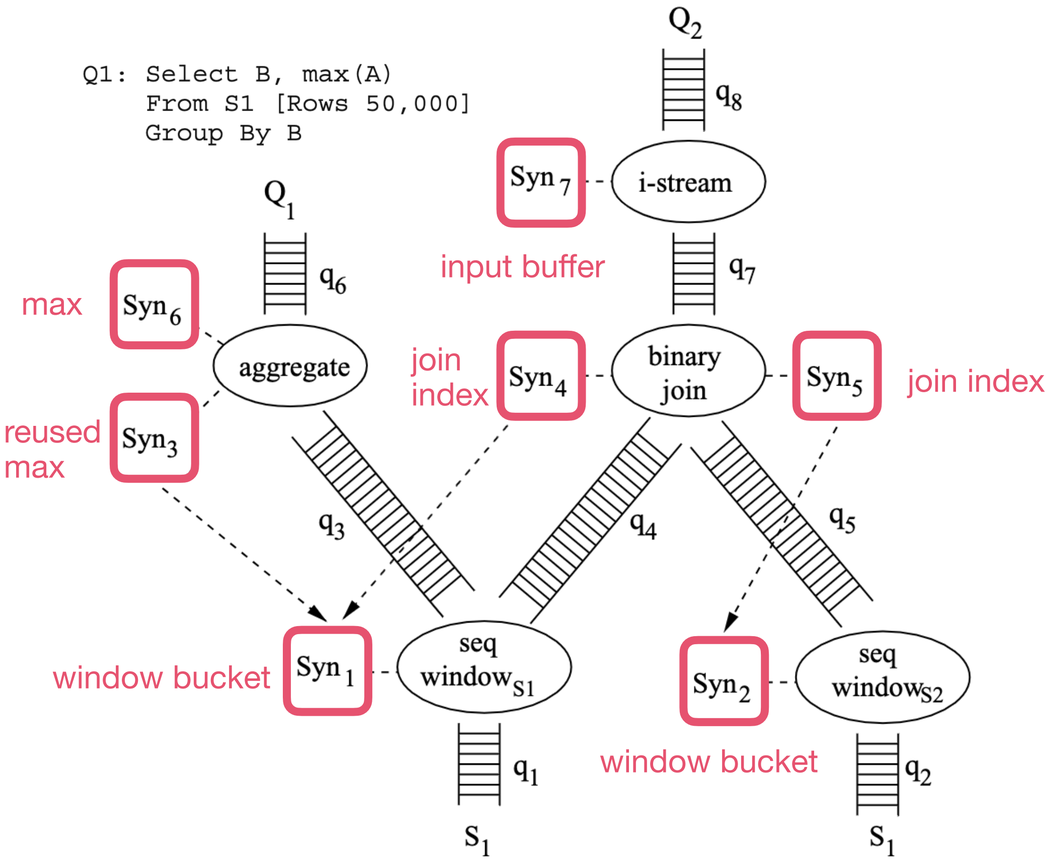


The Aurora and Borealis Stream Processing Engines
Çetintemel U, Abadi D, Ahmad Y, et. al. (2016)

Examples

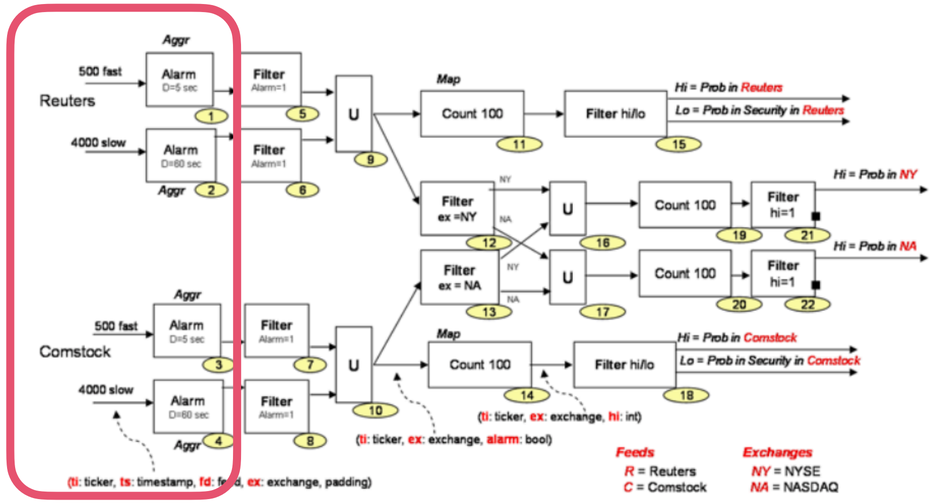
```
Q2: Select Istream(*)
    From   S1 [Rows 40,000],
           S2 [Range 600 Seconds]
    Where  S1.A = S2.A
```

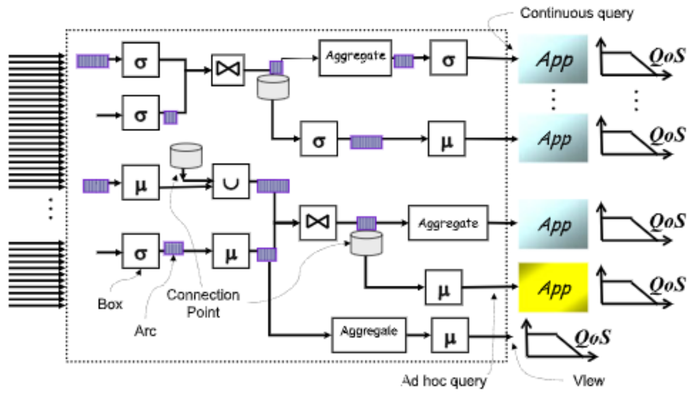
```
Q1: Select B, max(A)
    From S1 [Rows 50,000]
    Group By B
```



The CQL continuous query language: Semantic foundations and query execution
Arasu A, Babu S, Widom J in VLDB Journal (2006)

```
Aggregate(Group by ticker,
          Order on arrival,
          Window (Size = 2 tuples,
                  Step = 1 tuple,
                  Timeout = 5 sec))
```





The Aurora and Borealis Stream Processing Engines
Çetintemel U, Abadi D, Ahmad Y, et. al. (2016)

Synopses Overview

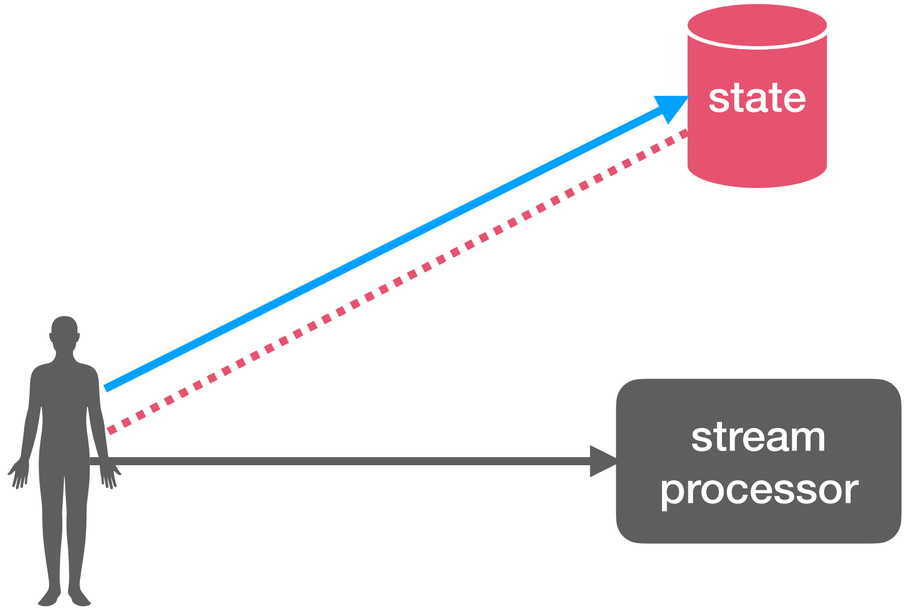
Benefits

- System can manage synopses efficiently as internal data structures
- Synopses can be combined and reused across operators internally (STREAM)
- Powerful approach in *shared-memory* system implementations

Downsides

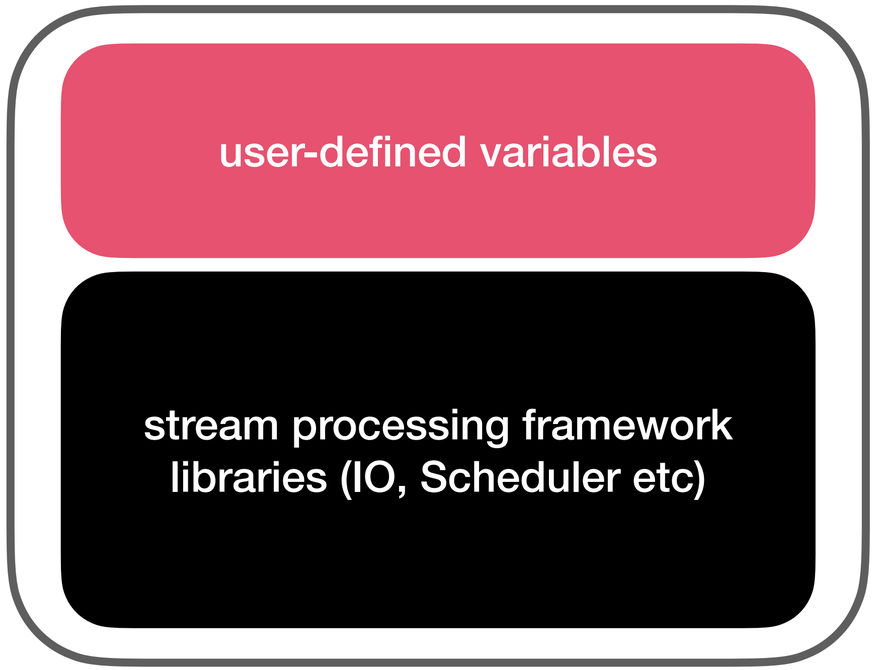
- Tight coupling of state to limited set of operator semantics (implementation)
- No general, declarative model for underlying user-defined state
- Over-specialization complicates auxiliary operations (fault tolerance, reconfiguration, elasticity etc.)

Application-Managed User-Defined State



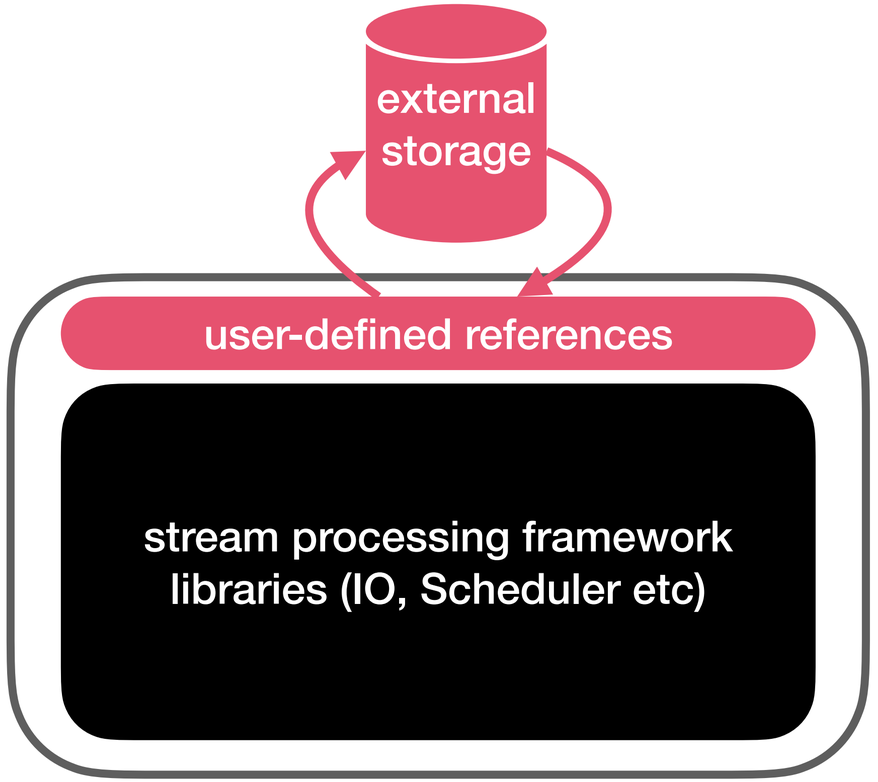
Application-Managed User-Defined State

Variant 1 - Process-Local State



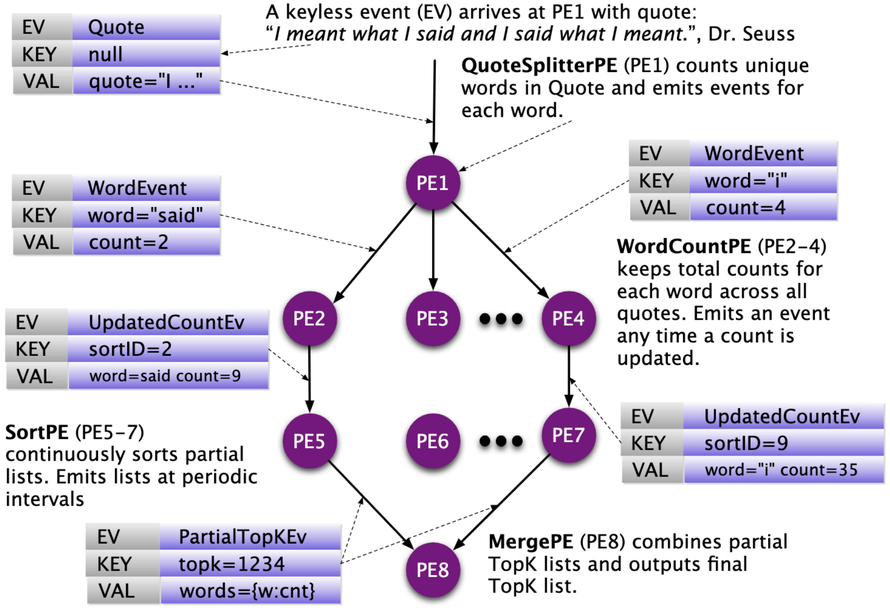
Stream Processing Task

Variant 2 - External State



Stream Processing Task

Examples

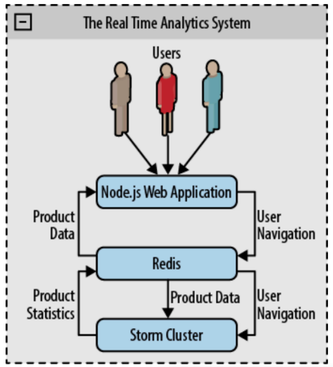


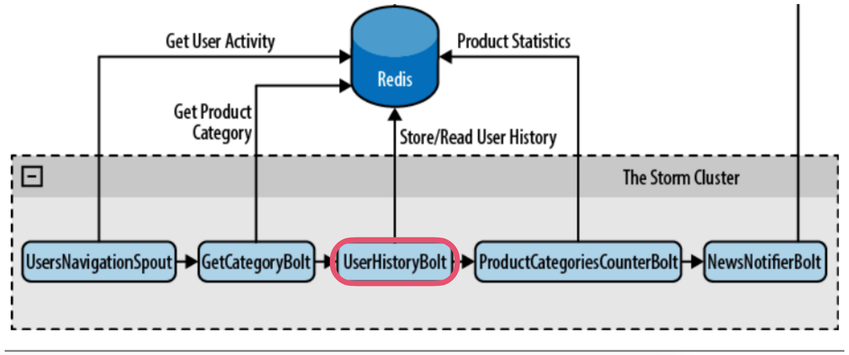

```
private queryCount = 0;

public void processEvent(Event event)
{
  queryCount ++;
}

public void output()
{
  String query = (String) this.getKeyValue().get(0);
  persister.set(query, queryCount ;
}
```

S4: Distributed Stream Computing Platform
Neumeyer L, Robbins B, Nair A, et. al. (2010)






```
public class UserHistoryBolt extends BaseRichBolt{

    @Override
    public void execute(Tuple input) {
        String user = input.getString(0);
        String prodKey = input.getString(1)+":"+input.getString(2);
        Set<String> productsNavigated = getHistoryRedis(user);
        if(!productsNavigated.contains(prodKey)) {
            ...
            addHistoryRedis(user, prodKey);
        }
    }
    private Set<String> getHistoryRedis(String user) {
        Set<String> userHistory = redisNavItems.get(user);
        if(userHistory == null) {
            userHistory = jedis.smembers(buildKey(user));
            if(userHistory == null)
                userHistory = new HashSet<String>();
            redisNavItems.put(user, userHistory);
        }
        return userHistory;
    }
    private void addHistoryRedis(String user, String product) {
        Set<String> userHistory = getHistoryRedis(user);
        userHistory.add(product);
        jedis.sadd(buildKey(user), product);
    }
    ...
}
```

Getting Started with Storm
Leibiusky J, Eisbruch G, Simonassi D, et. al. (2012)

Summary

Benefits

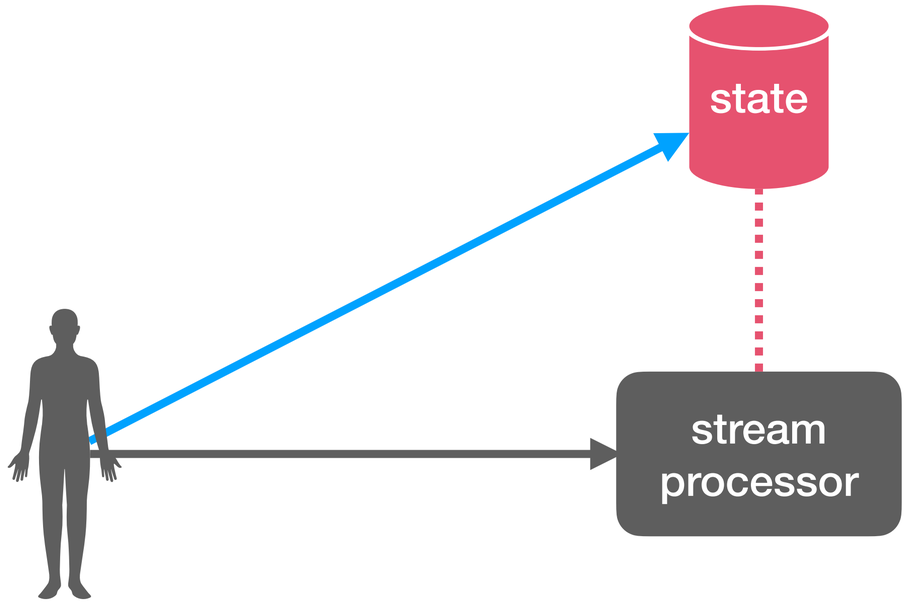
- Flexibility to use any complex state type (e.g., graphs, matrices, arrays etc.)
- Decoupling of state and computing in application-logic

Downsides

- Manual application state management is necessary
- Third-party system dependencies are required for persistence and FT
- Requires deep user expertise on state management
- Missed opportunities for system-level state access optimisation

System-Managed User-Defined State

State as a first-class Citizen



Examples

```
// the source data stream
val stream: DataStream[…] = ...
val result: DataStream[…] = stream
  .keyBy(0)
  .process(new MyCustomLogic())
```

```
class MyCustomLogic extends KeyedProcessFunction[…] {

  /** The state that is maintained by this process function */
  lazy val state: ValueState[…] = getRuntimeContext
    .getState(new ValueStateDescriptor[…](“myState”, classOf[…]))


  override def processElement(element: …)
    …
    state.update(…)
    …
    ctx.timerService.registerEventTimeTimer(…)
    …
  }

  override def onTimer( timestamp: Long,  StreamContext, TimerContext,
out: …): Unit = {

    state.value match {
      case foo => out.collect((key, count))
      case _ =>
    }
  }
}
```

https://flink.apache.org/

```
class IndexAssigningStatefulDoFn(DoFn):
  INDEX_STATE = CombiningStateSpec('index', sum)

  def process(self, element, index=DoFn.StateParam(INDEX_STATE)):
    unused_key, value = element
    current_index = index.read()
    yield (value, current_index)
    index.add(1)
```

```
# Events is a collection of (user, event) pairs.
events = (p | ReadFromEventSource() | beam.WindowInto(....))

user_models = beam.pvalue.AsDict(
                  events
                  | beam.core.CombinePerKey(ModelFromEventsFn()))

def event_prediction(user_event, models):
  user = user_event[0]
  event = user_event[1]

  # Retrieve the model calculated for this user
  model = models[user]

  return (user, model.prediction(event))

# Predictions is a collection of (user, prediction) pairs.
predictions = events | beam.Map(event_prediction, user_models)
```

https://beam.apache.org/blog/stateful-processing/

Examples

```
// the source data stream
val stream: DataStream[…] = ...
val result: DataStream[…] = stream
  .keyBy(0)
  .process(new MyCustomLogic())
```

```
class MyCustomLogic extends KeyedProcessFunction[…] {

  /** The state that is maintained by this process function */
  lazy val state: ValueState[…] = getRuntimeContext
    .getState(new ValueStateDescriptor[…]("myState", classOf[…]))

  override def processElement(element: …)
    …
    state.update(…)
    …
    ctx.timerService.registerEventTimeTimer(…)
    …
  }

  override def onTimer( timestamp: Long,  StreamContext, TimerContext,
out: …): Unit = {

    state.value match {
      case foo => out.collect((key, count))
      case _ =>
    }
  }
}
```

https://flink.apache.org/

```
class IndexAssigningStatefulDoFn(DoFn):
  INDEX_STATE = CombiningStateSpec('index', sum)

  def process(self, element, index=DoFn.StateParam(INDEX_STATE)):
    unused_key, value = element
    current_index = index.read()
    yield (value, current_index)
    index.add(1)
```

```
# Events is a collection of (user, event) pairs.
events = (p | ReadFromEventSource() | beam.WindowInto(....))

user_models = beam.pvalue.AsDict(
                  events
                  | beam.core.CombinePerKey(ModelFromEventsFn()))

def event_prediction(user_event, models):
  user = user_event[0]
  event = user_event[1]

  # Retrieve the model calculated for this user
  model = models[user]

  return (user, model.prediction(event))

# Predictions is a collection of (user, prediction) pairs.
predictions = events | beam.Map(event_prediction, user_models)
```

https://beam.apache.org/blog/stateful-processing/

Examples

```
// the source data stream
val stream: DataStream[…] = ...
val result: DataStream[…] = stream
  .keyBy(0)
  .process(new MyCustomLogic())
```

```
class MyCustomLogic extends KeyedProcessFunction[…] {

  /** The state that is maintained by this process function */
  lazy val state: ValueState[…] = getRuntimeContext
    .getState(new ValueStateDescriptor[…]("myState", classOf[…]))

  override def processElement(element: …)
    …
    state.update(…)
    …
    ctx.timerService.registerEventTimeTimer(…)
    …
  }

  override def onTimer( timestamp: Long,  StreamContext, TimerContext,
out: …): Unit = {

    state.value match {
      case foo => out.collect((key, count))
      case _ =>
    }
  }
}
```

https://flink.apache.org/

```
class IndexAssigningStatefulDoFn(DoFn):
  INDEX_STATE = CombiningStateSpec('index', sum)

  def process(self, element, index=DoFn.StateParam(INDEX_STATE)):
    unused_key, value = element
    current_index = index.read()
    yield (value, current_index)
    index.add(1)
```

```
# Events is a collection of (user, event) pairs.
events = (p | ReadFromEventSource() | beam.WindowInto(....))

user_models = beam.pvalue.AsDict(
                    events
                    | beam.core.CombinePerKey(ModelFromEventsFn()))

def event_prediction(user_event, models):
  user = user_event[0]
  event = user_event[1]

  # Retrieve the model calculated for this user
  model = models[user]

  return (user, model.prediction(event))

# Predictions is a collection of (user, prediction) pairs.
predictions = events | beam.Map(event_prediction, user_models)
```

https://beam.apache.org/blog/stateful-processing/

Summary

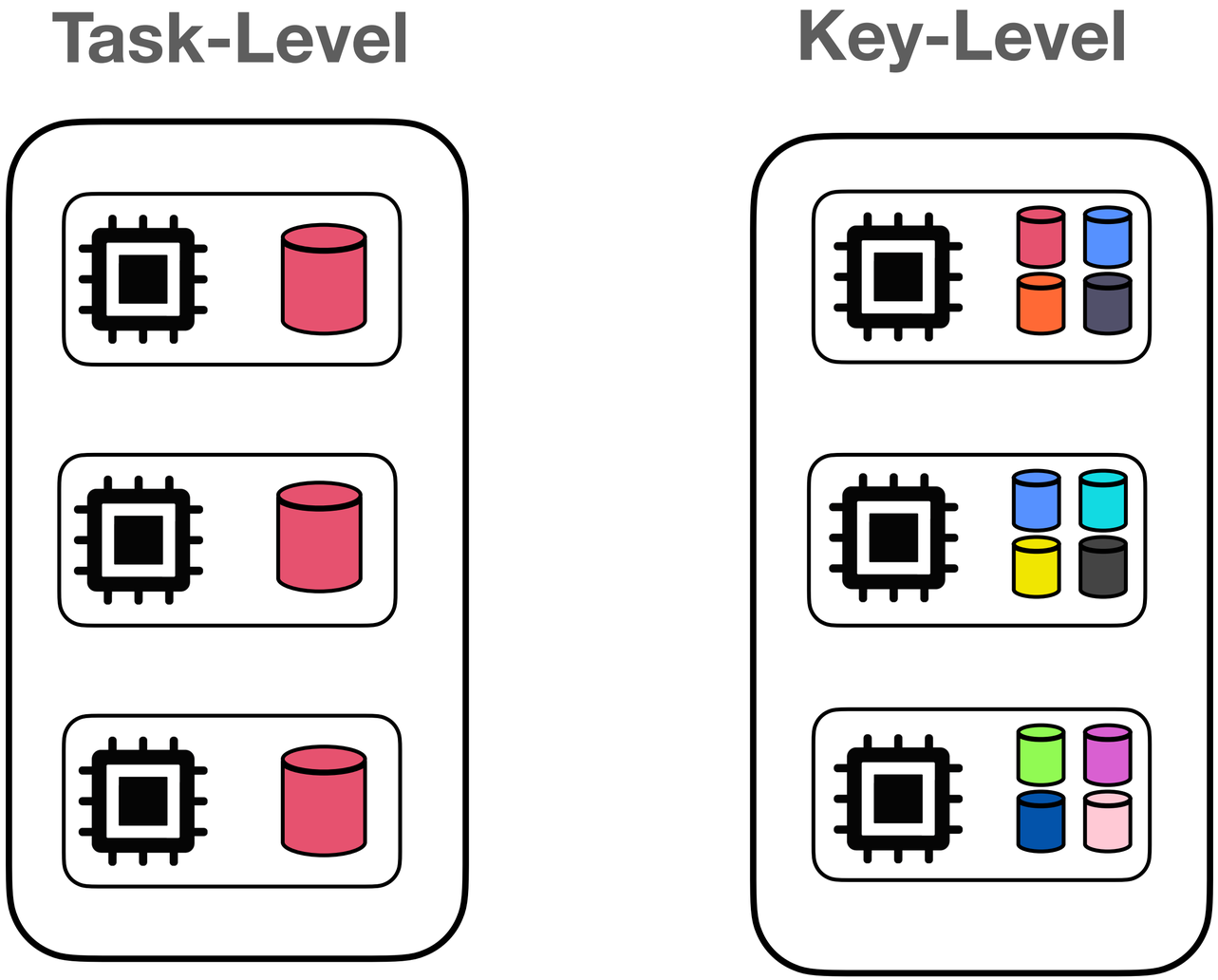
Benefits

- Stream Processor can scale, persist, reconfigure and make state durable.
- No external data storage systems to be maintained by the user.

Downsides

- Restrictions on supported state types (e.g., Flink’s value, append-list, map)
- Limited set of user-level state optimisations

Partitioned State



Task-Level State Examples

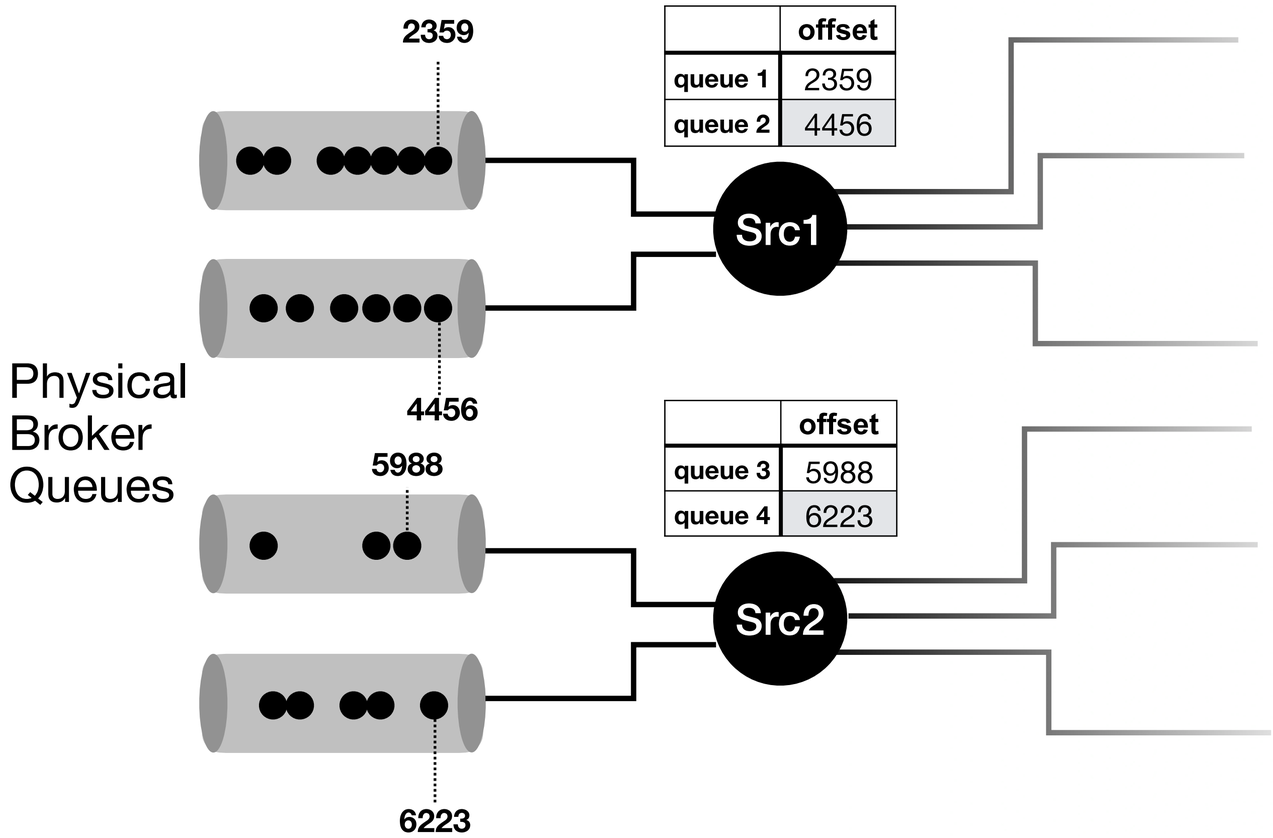
topK Window Counts in Samza

```
skillTags.merge(jobTags).map(MyCounter)
      .window(10, TopKFinder).sendto(topTags);

  class MyCounter implements Map<In, Out>{
  //state definition
   Store<String, int> counts = new Store();
   public Out apply (In msg){
      int cur = counts.get(msg.id) + 1;
      counts.put(msg.id, cur);
      return new Out(msg.id, cur)
   }
```

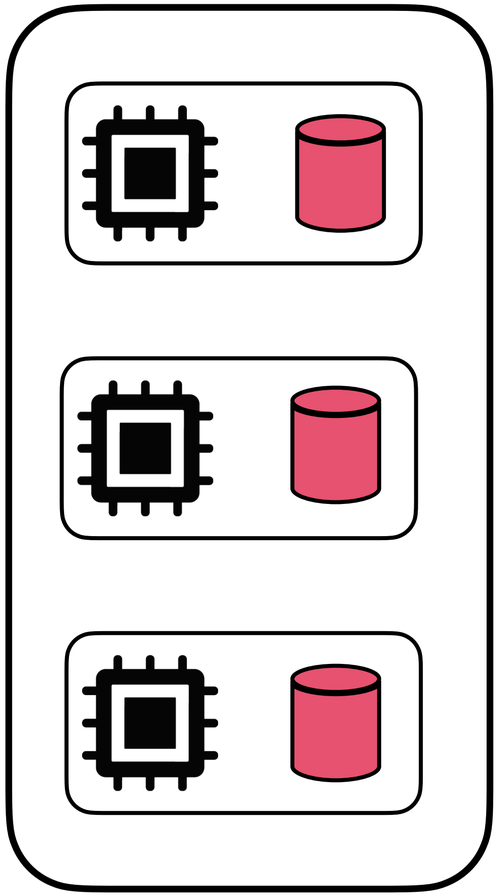
Samza: Stateful Scalable Stream Processing at LinkedIn
Noghabi S, Paramasivam K, Pan Y, et. al. in Proc. VLDB Endow. (2017)

Kafka Source (Flink OperatorState)



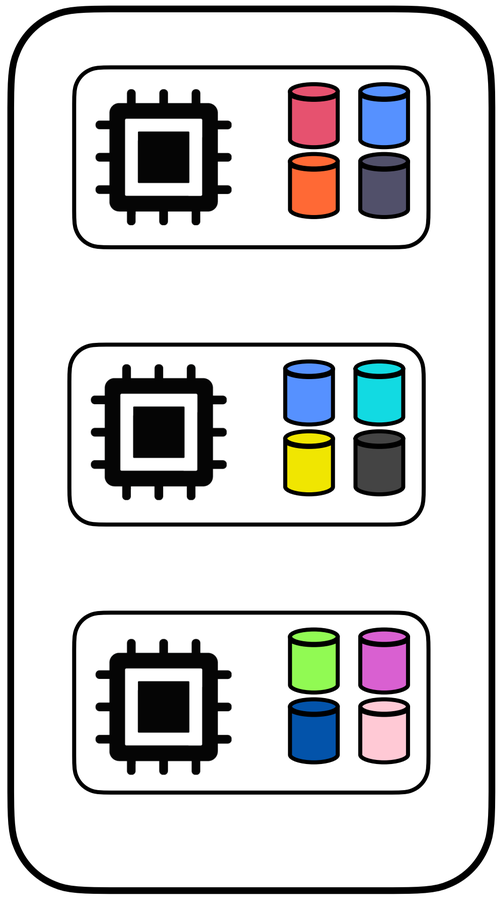
Summary

Task-Level



- Maps to physical partitioning of compute tasks.
- Relevant for continuous task-parallel computing (e.g., online ML)
- Typically *non-growing* state
- Common on message-broker native stream proc. frameworks (e.g., Samza)
- **Hard to re-partition**

Key-Level State

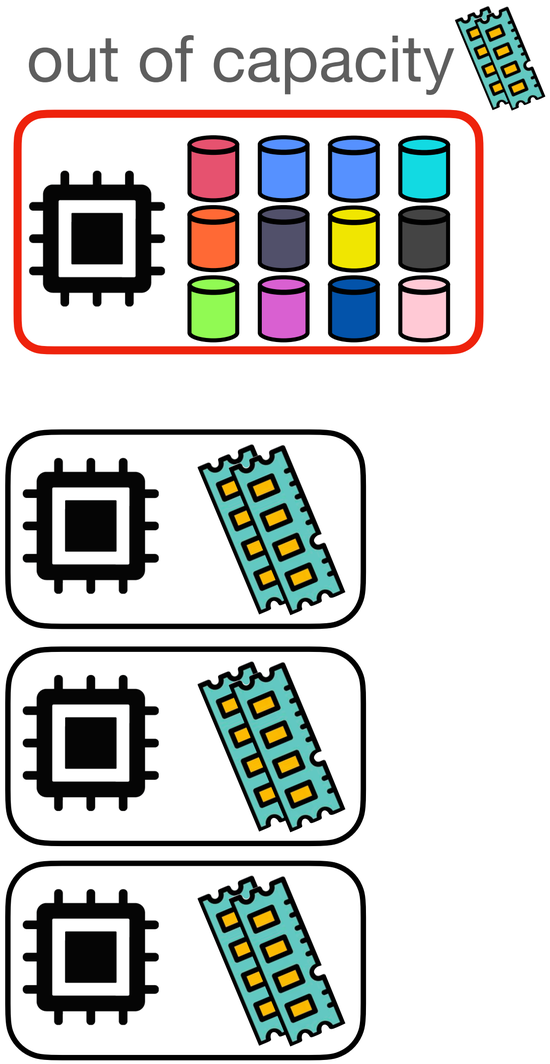


keyby(…)

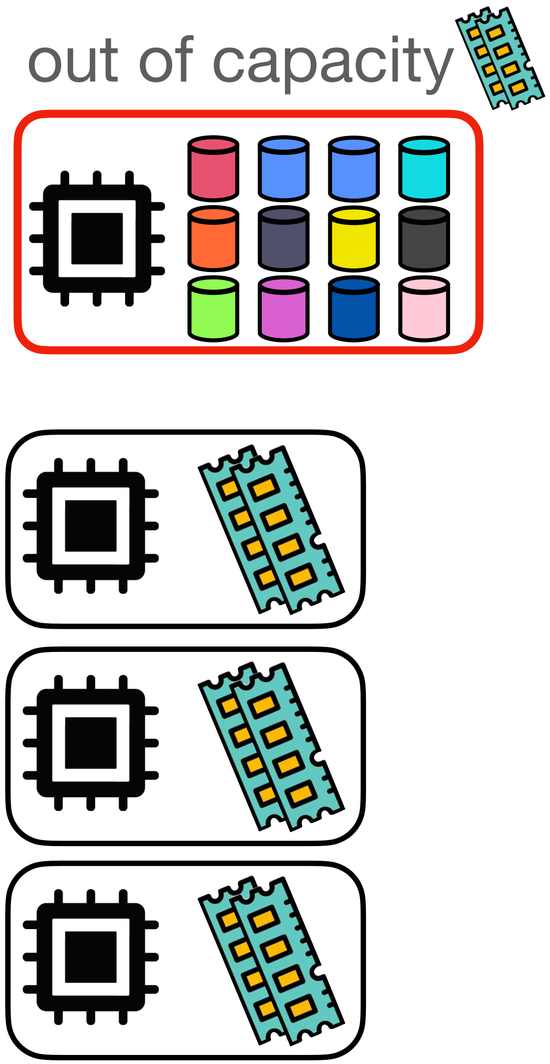
groupBy(..)

- Maps to logical partitioning of compute tasks (multiple keys handled by each task).
- Typically *growing* state
- Common for data parallel stream computation
- **Easy to partition**

Scalable (Out-of-Core) State Architectures

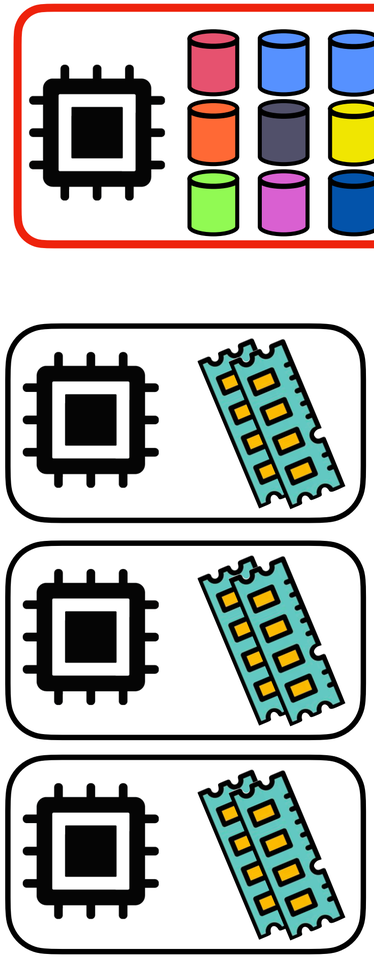


Scalable (Out-of-Core) State Architectures

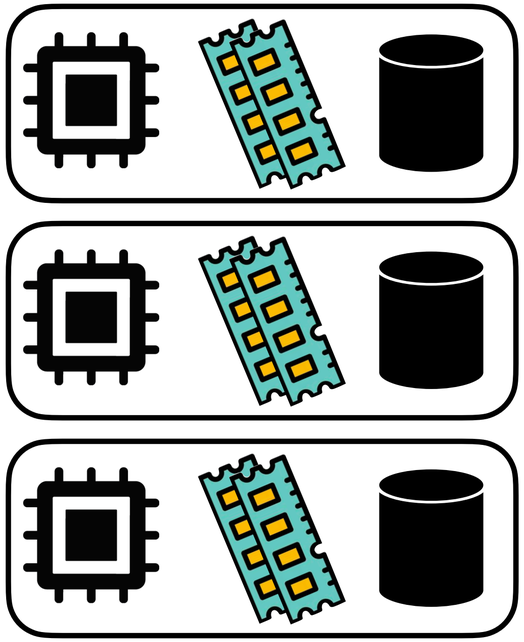


Scalable (Out-of-Core) State Architectures

out of capacity

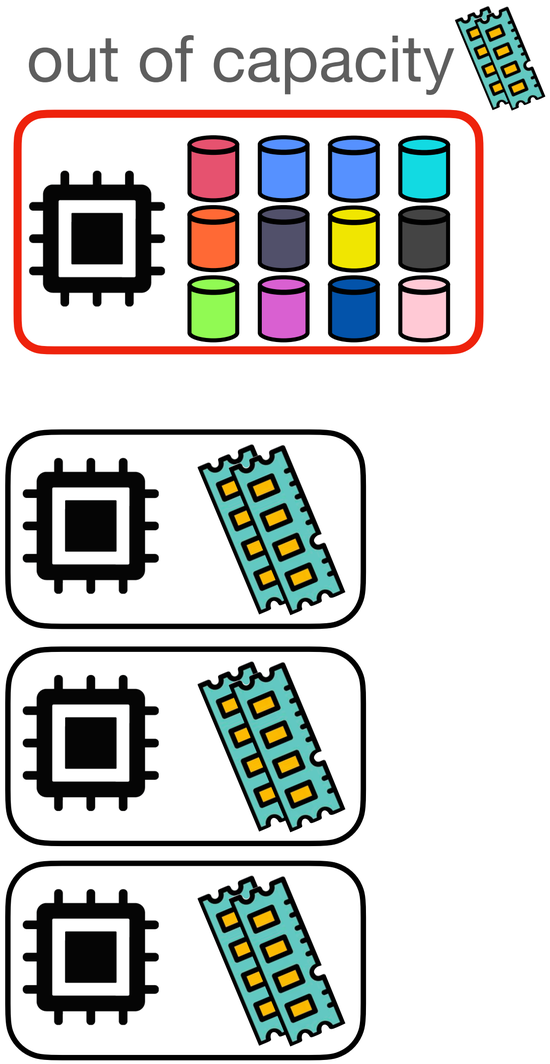


I. Embedded State

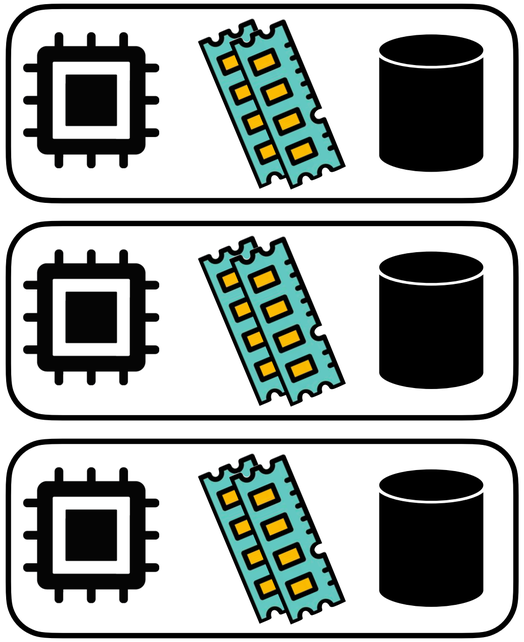


Flink	Apex
Seep	Samza
IBM-Streams	Spark Structured Streaming

Scalable (Out-of-Core) State Architectures



I. Embedded State



Flink

Seep

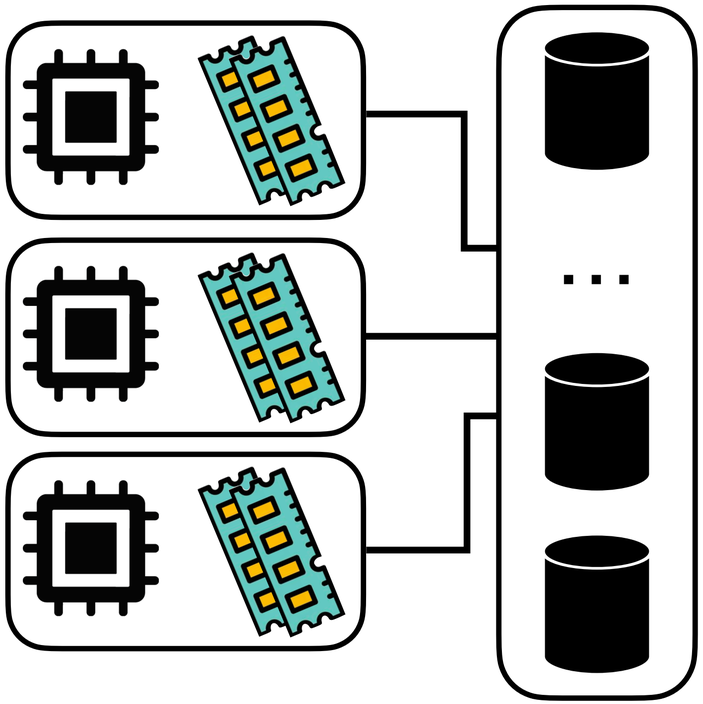
IBM-Streams

Apex

Samza

Spark Structured Streaming

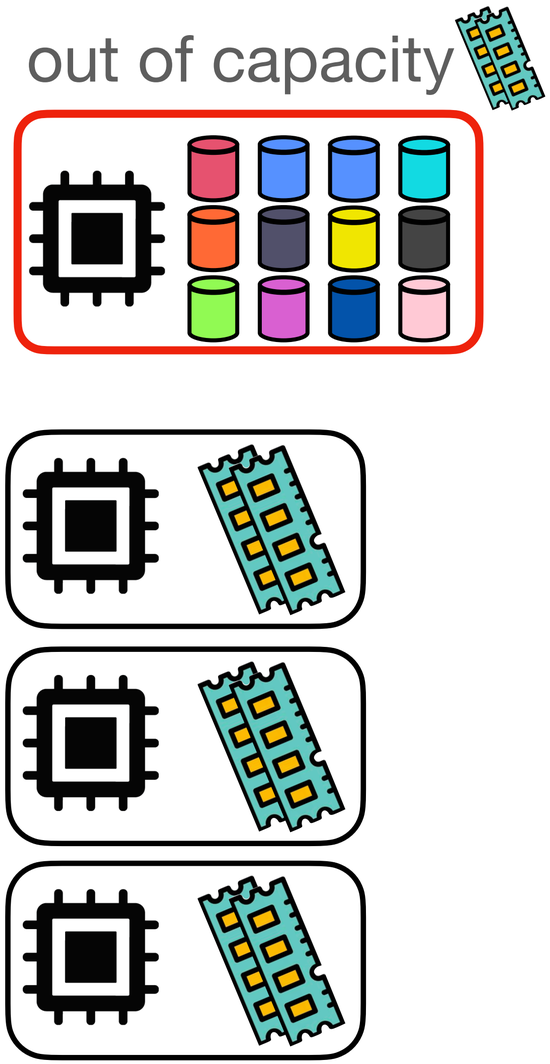
II. External State



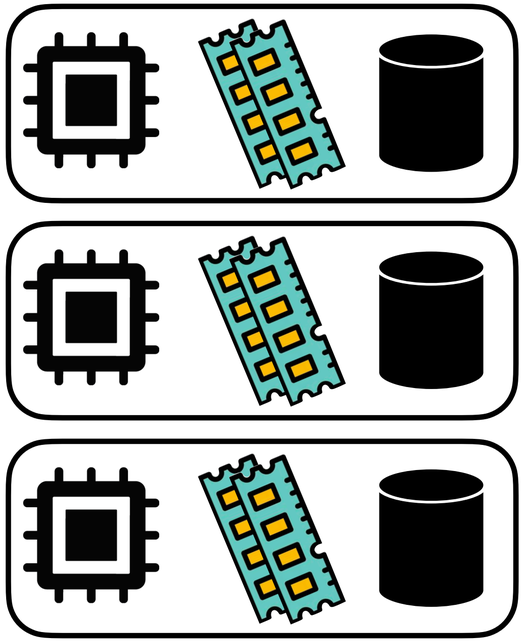
Millwheel (Google Dataflow)

Scalable (Out-of-Core) State Architectures

out of capacity



I. Embedded State



Flink

Seep

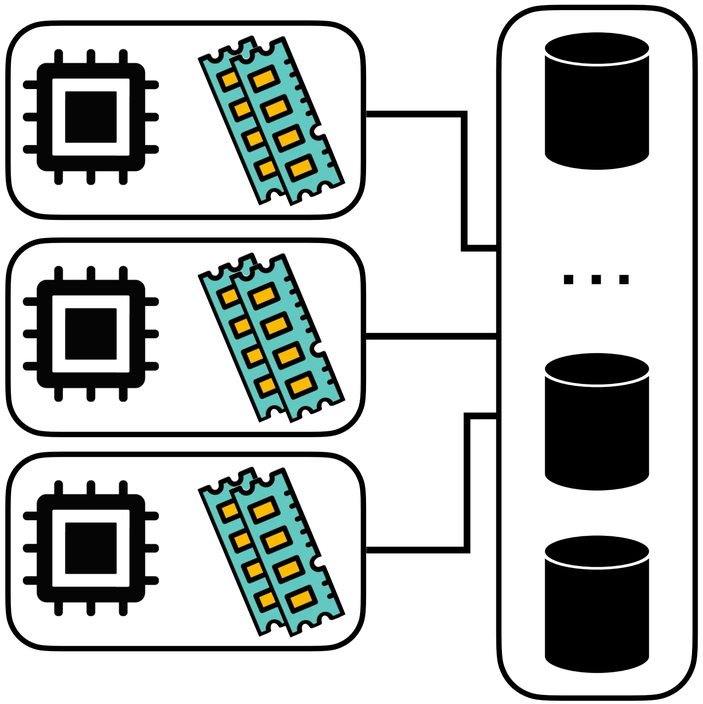
IBM-Streams

Apex

Samza

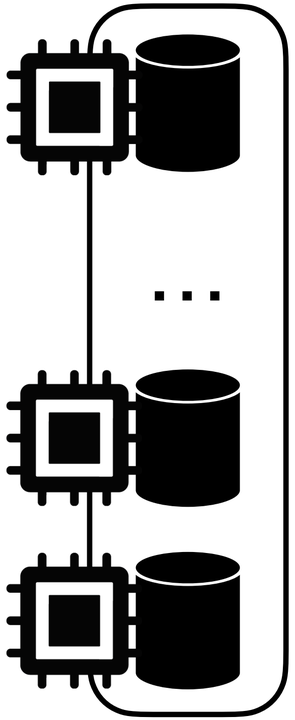
Spark Structured Streaming

II. External State



Millwheel (Google Dataflow)

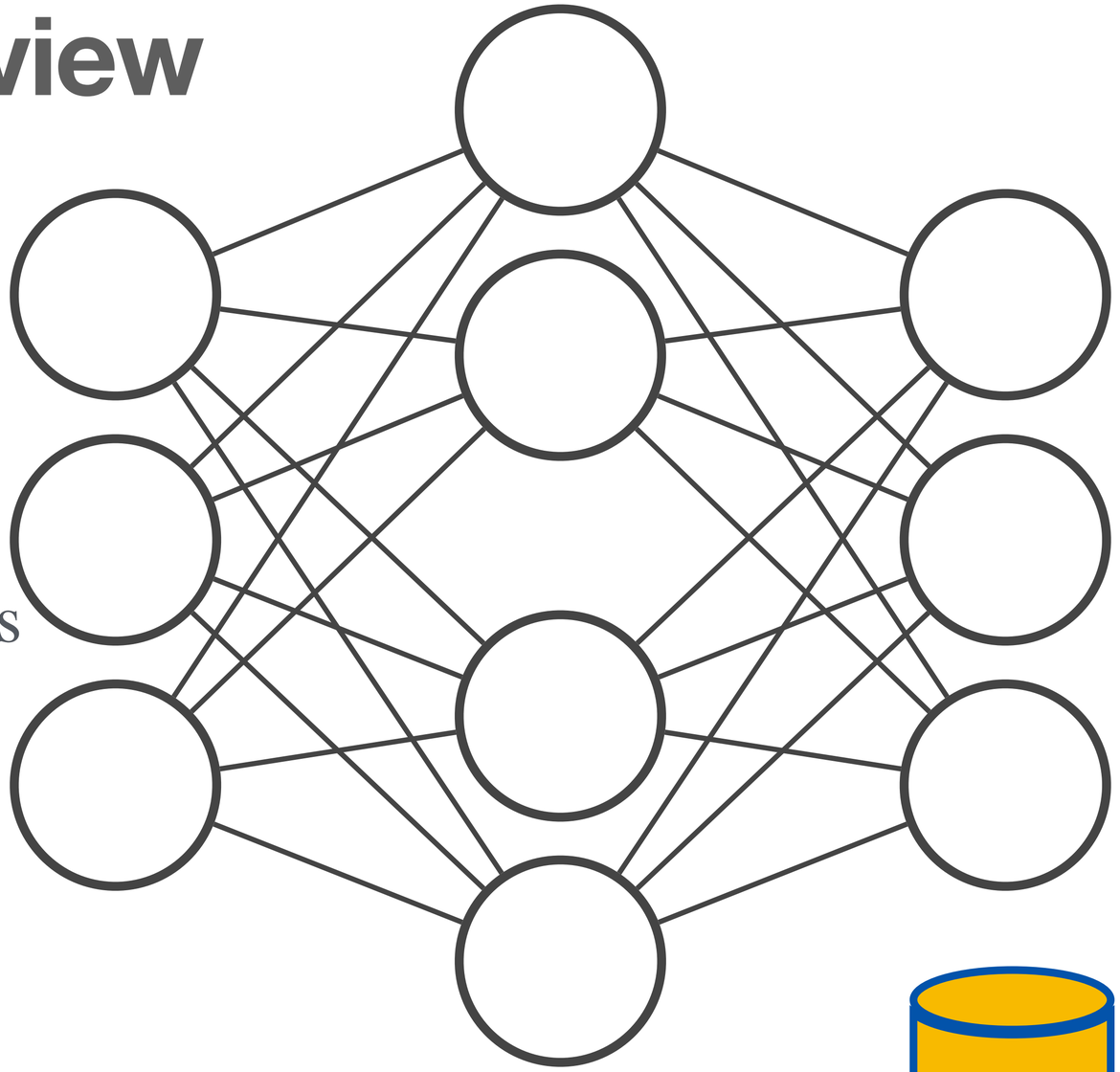
III. Embedded Compute



S-Store (on H-Store)

Kafka-Streams (on Kafka)

Embedded State - Overview



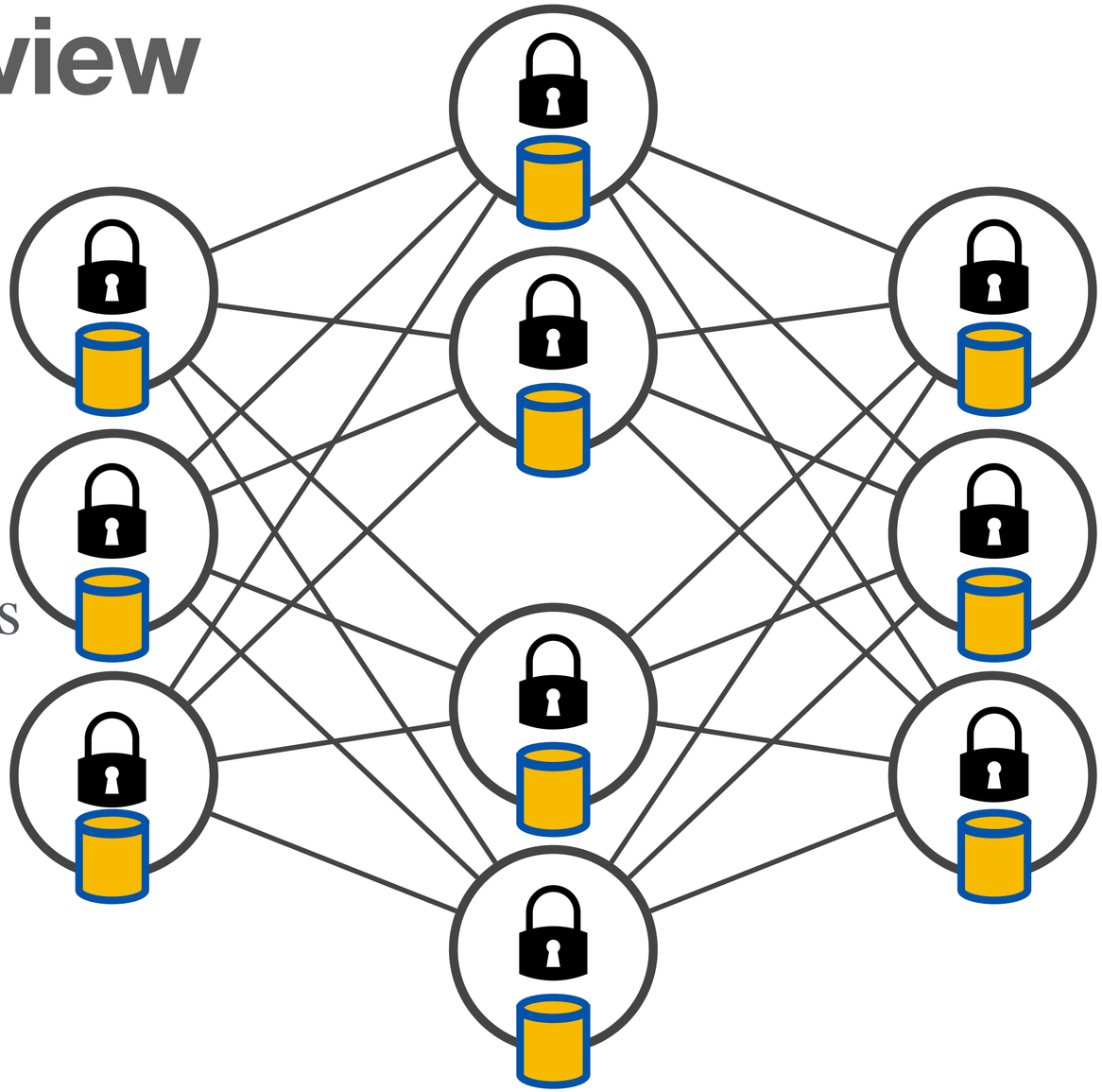
Pros

- Direct, fast state access
- No external calls
- [Prospect] Flexibility on local data structures

Cons

- Strong Coupling on Scalability
- Challenging Transactional Processing
- Complex/Slow Reconfiguration

Embedded State - Overview



Pros

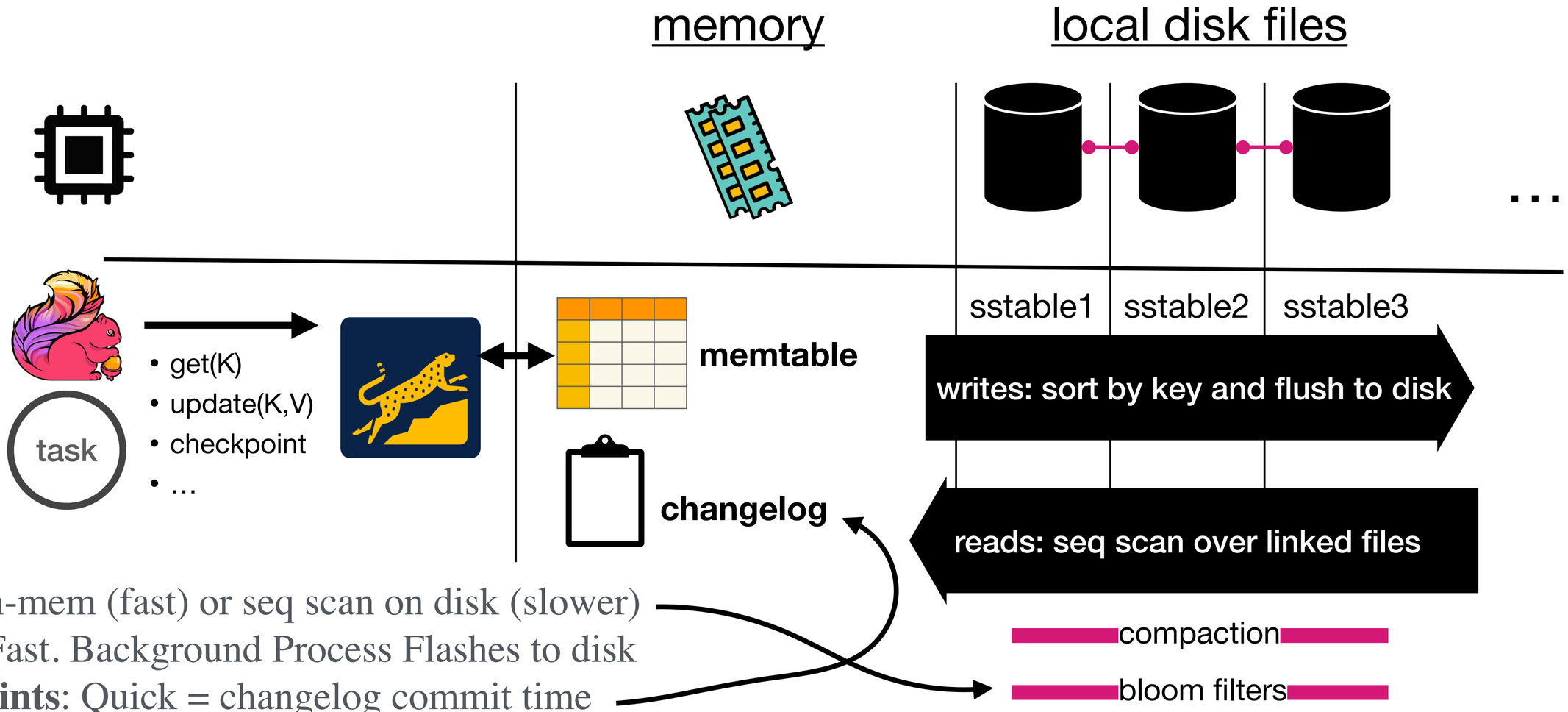
- Direct, fast state access
- No external calls
- [Prospect] Flexibility on local data structures

Cons

- Strong Coupling on Scalability
- Challenging Transactional Processing
- Complex/Slow Reconfiguration

Embedded State Example

Flink with Rocksdb (Log-Structured-Merge)



- **Reads**: in-mem (fast) or seq scan on disk (slower)
- **Writes**: Fast. Background Process Flashes to disk
- **Checkpoints**: Quick = changelog commit time

Embedded State Trends

- *Out-of-Core* is nearly synonymous to the embedded state architecture
- LSM-based storage is the norm but there is more…
  - Unnecessary low-level latching for shared stream access
  - Index-based backends more appropriate for fast stream reads
  - Modern Hardware on the rise: NVMe, RDMA etc.

Embedded State Trends

FASTER: An Embedded Concurrent Key-Value Store for State Management*

Badrish Chandramouli‡, Guna Prasaad◇†, Donald Kossmann‡, Justin Levandoski‡,
James Hunter‡, Mike Barnett‡
‡Microsoft Research ◇University of Washington
badrishc@microsoft.com, guna@cs.washington.edu, {donaldk, justin.levandoski, jahunter, mbarnett}@microsoft.com

ABSTRACT

Over the last decade, there has been a tremendous growth in data-intensive applications and services in the cloud. Data is created on a variety of edge sources such as devices, and is processed by cloud applicatio… are typica… beyond w… nificant t… FASTER, … current h… record sto… fast in-pla… magnitud… It is built… dynamic … such as l… on: (1) th… deeply in… logic at lo… resulting … ities, dura…

PVLDB R…
Badrish C… doski, Jam… rent Key-V… 2018.
DOI: https…

may also issue a mix of record inserts, updates, and reads, usually supported via caching object stores such as Redis and Memcached.

State management is a difficult challenge for such applications and services, and has several unique characteristics:

Research 3: Transactions and Indexing SIGMOD'18, June 10-15, 2018, Houston, TX, USA

FASTER: A Concurrent Key-Value Store with In-Place Updates

Badrish Chandramouli†, Guna Prasaad‡*, Donald Kossmann†, Justin Levandoski†,
James Hunter†, Mike Barnett†
†Microsoft Research ‡University of Washington
badrishc@microsoft.com, guna@cs.washington.edu, {donaldk, justin.levandoski, jahunter, mbarnett}@microsoft.com

ABSTRACT

Over the last decade, there has been a tremendous growth in data-intensive applications and services in the cloud. Data is created on a variety of edge sources, e.g., devices, browsers, and servers, and processed by cloud applications to gain insights or take decisions. Applications and services either work on collected data, or monitor and process data in real time. These applications are typically update intensive and involve a large amount of state beyond what can fit in main memory. However, they display significant temporal locality in their access pattern. This paper presents FASTER, a new key-value store for point read, blind update, and read-modify-write operations. FASTER combines a highly cache-optimized concurrent hash index with a *hybrid log*: a concurrent log-structured record store that spans main memory and storage, while supporting fast

1.1 Challenges and Existing Solutions

State management is an important problem for all these applications and services. It exhibits several unique characteristics:

- ***Large State***: The amount of state accessed by some applications can be very large, far exceeding the capacity of main memory. For example, a targeted search ads provider may maintain per-user, per-ad and clickthrough-rate statistics for billions of users. Also, it is often cheaper to retain state that is infrequently accessed on secondary storage [18], even when it fits in memory.
- ***Update Intensity***: While reads and inserts are common, there are applications with significant update traffic. For example, a monitoring application receiving millions of CPU readings every second from sensors and devices may need to update a per-device aggregate for each reading.

Research 17: Scalability SIGMOD '19, June 30–July 5, 2019, Amsterdam, Netherlands

FISHSTORE: Faster Ingestion with Subset Hashing

Dong Xie§* Badrish Chandramouli† Yinan Li† Donald Kossmann†
†Microsoft Research §University of Utah
dongx@cs.utah.edu,{badrishc,yinali,donaldk}@microsoft.com

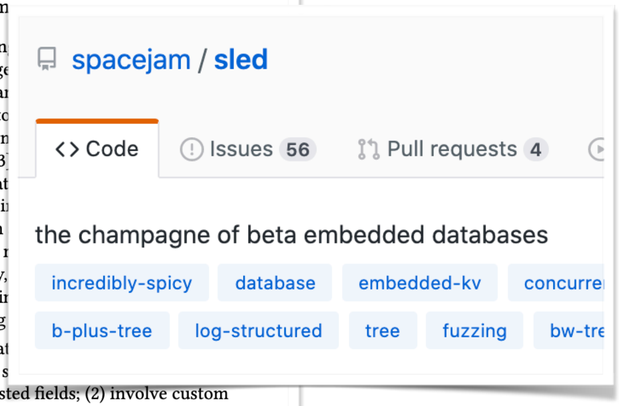
ABSTRACT

The last decade has witnessed a huge increase in data being ingested into the cloud, in forms such as JSON, CSV, and binary formats. Traditionally, data is either ingested into …n, indexed ad-hoc using range indices, …ytics-friendly columnar formats. None …s able to handle modern requirements … the data available immediately for ad-…ueries while ingesting at extremely high …aper builds on recent advances in parsing …ques to propose FISHSTORE, a concurrent …yer for data with flexible schema, based … indexing of dynamically registered *pred-*… We find predicated subset hashing to be …e that supports a broad range of queries …d admits a high-performance concurrent …r detailed evaluation on real datasets and

huge increase in data being in… variety of data sources. The inge… ranging from JSON (a popular… with high expressive power) to… (comma-separated values) form… as Google Protocol Buffers [13]…

Given the huge ingested dat… tion has traditionally been to i… saturating storage bandwidth … overhead. These goals usually r… data on storage. More recently, … ing need [17, 33] to make the i… diately" for an ever-increasing …

- *Ad-hoc analysis* queries that… (e.g., last hour of data). The s… predicates over possibly nested fields; (2) involve custom logic to select a varying (but usually small) number of

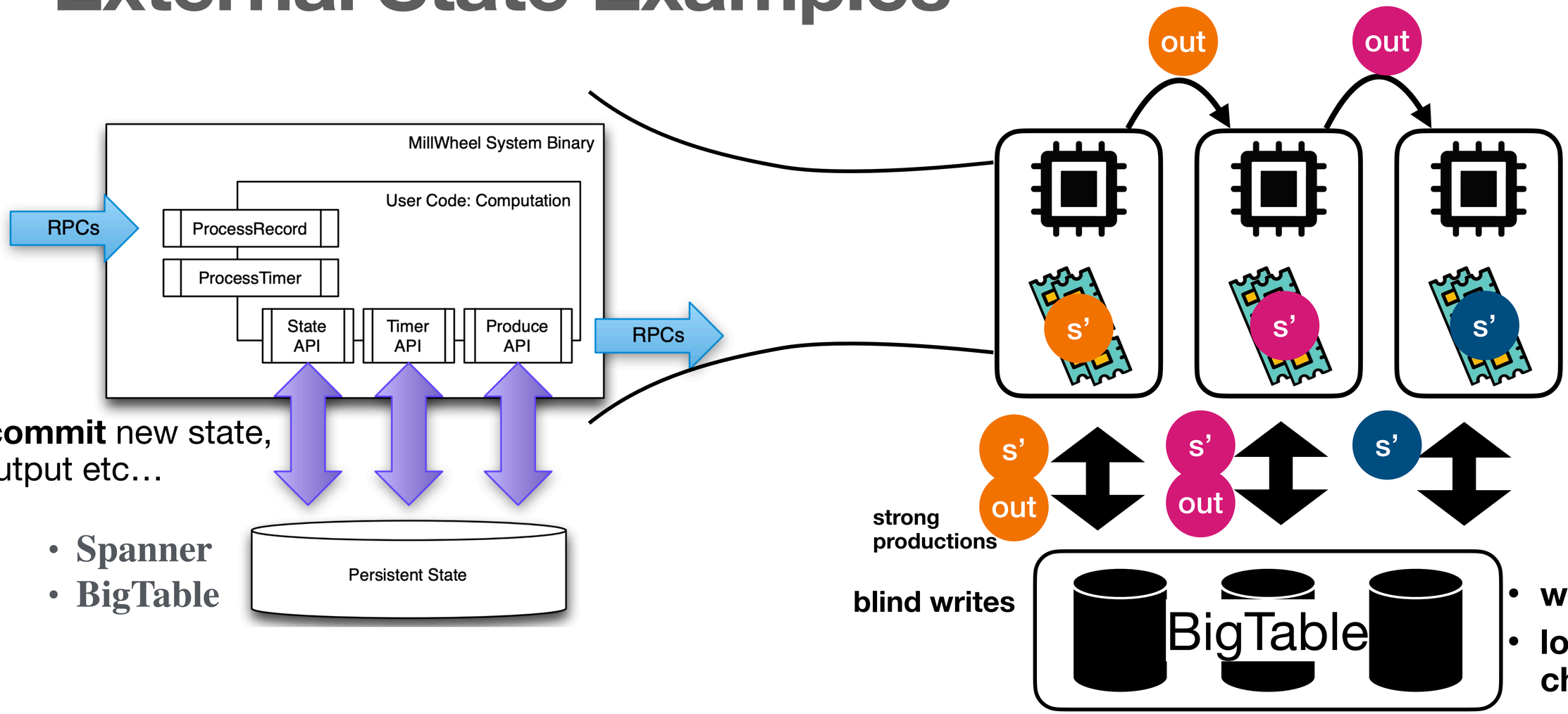


LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans

Xiaowei Zhu[1], *Guanyu Feng*[1], *Marco Serafini*[2], *Xiaosong Ma*[3], *Jiping Yu*[1], *Lei Xie*[1], *Ashraf Aboulnaga*[3], *and Wenguang Chen*[1]

[1]Tsinghua University
[2]University of Massachusetts Amherst
[3]Qatar Computing Research Institute

External State Examples



- Spanner
- BigTable

Millwheel: Fault-tolerant stream processing at internet scale

Akidau T, Balikov A, Bekiroğlu K, et. al. in Proceedings of the VLDB Endowment (2013)

External State Overview

Pros

- Decoupling of persistent and transient state
- Unified Approach for Operation Logging + Active state
- Compute Scalability ⧄ Storage Scalability
- Fast Reconfiguration, Failover etc.

Cons

- High-performance requirements on external storage
- Higher Read latency compared to embedded state (prefetching etc. needed)
- Complex maintenance/config. of >2 systems (outside managed cloud services)

Embedded Compute Example

S-Store: A small H-Store extension using scheduled Transactions

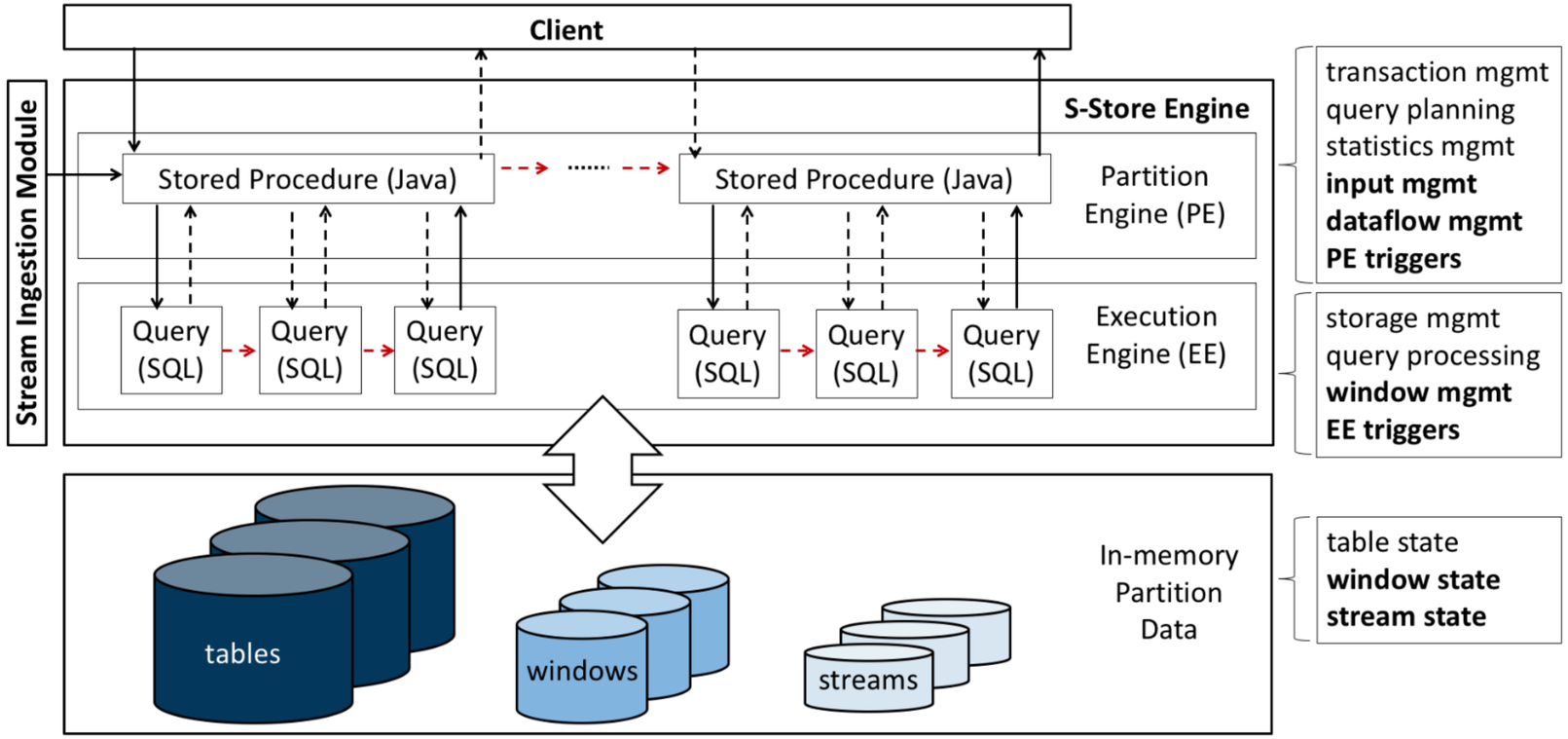


Figure 4: S-Store Architecture

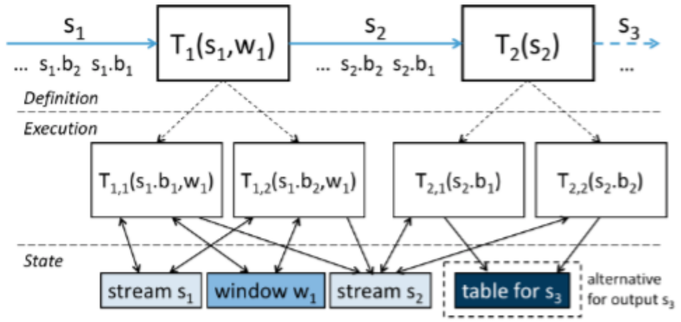
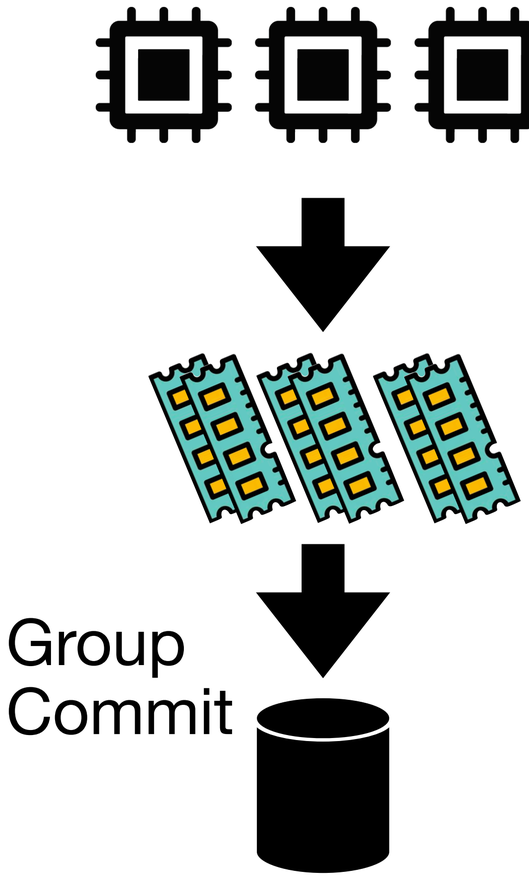


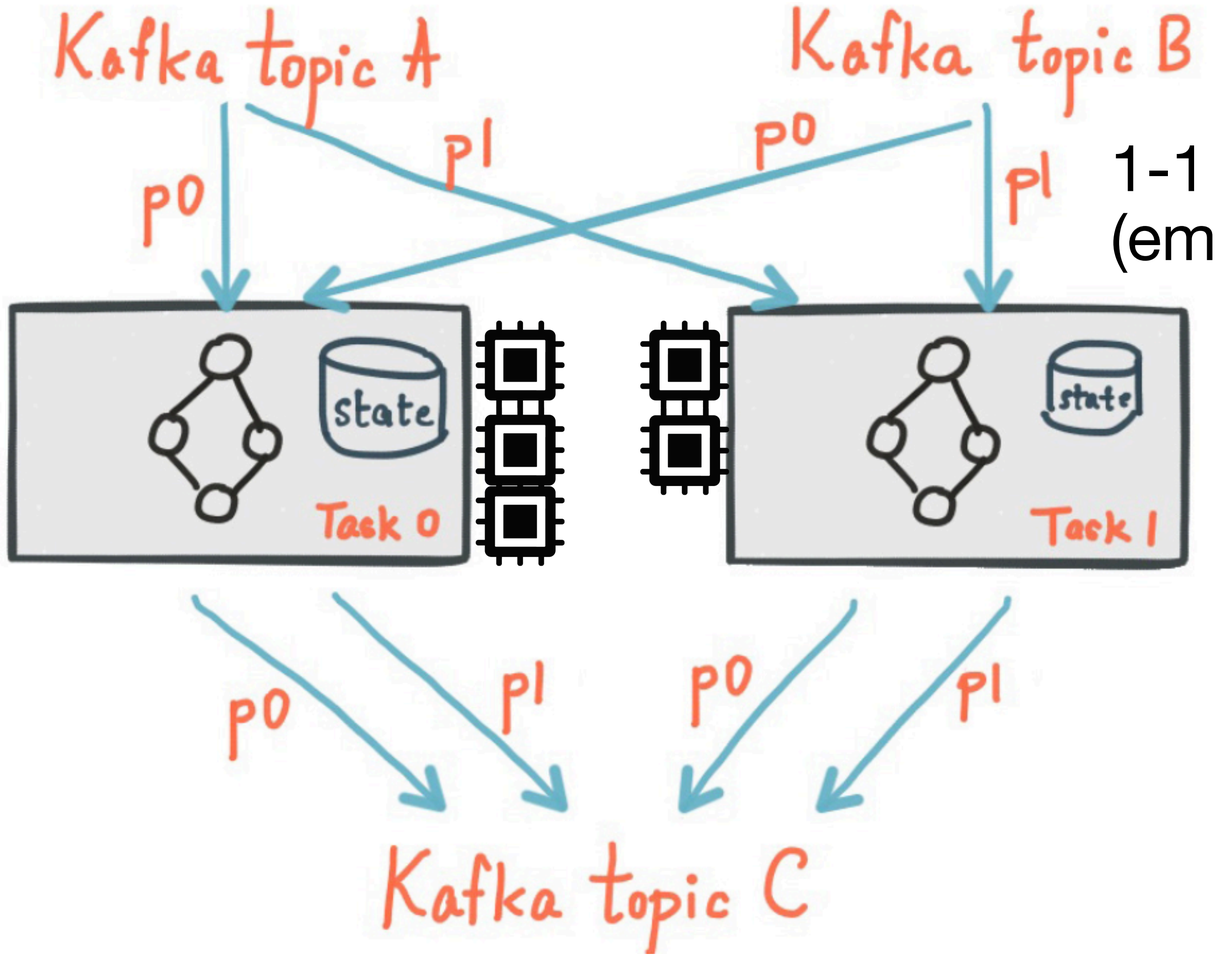
Figure 2: Transaction Executions in a Dataflow Graph



S-Store: A Streaming NewSQL System for Big Velocity Applications
Cetintemel U, Du J, Kraska T, et. al. (2014)

A Hybrid Approach (Embedded compute+state)

Kafka-Streams - Stream apps on top of partitioned logs



1-1 mapping to kafka brokers (embedding compute)

<- embedded state (rocksdb)

https://kafka.apache.org/25/documentation/streams/architecture

Embedded Compute Overview

Pros

- Enhance compute capabilities of a storage system (e.g., kafka, h-store)
- Built-in optimisations
- Only solutions that allow forms of *Shared Mutable State*

Cons

- Performance capabilities not in par with dedicated stream processing systems
- Tight coupling of stream framework to underlying storage technology

The Holy Grail of Stream Processing?

Consistency Guarantees Means…

handling failures!

consistent application updates

trusting external output

adding more/less workers

reconfiguring/upgrading the system correctly

Co ns…

handling fa ates

trusting /less workers

rrectly

Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management

Raul Castro Fernandez
Imperial College London
rc3011@doc.ic.ac.uk

Matteo Migliavacca
University of Kent
mm53@kent.ac.uk

Evangelia Kalyvianaki
Imperial College London
ekalyv@doc.ic.ac.uk

Peter Pietzuch
Imperial College London
prp@doc.ic.ac.uk

ABSTRACT

As users of "big data" applications expect fresh results, we witness a new breed of stream processing systems (SPS) that are designed to scale to large numbers of cloud-hosted machines. Such systems face new challenges: (i) to benefit from the "pay-as-you-go" model of cloud computing, they must *scale out* on demand, acquiring additional virtual machines (VMs) and parallelising operators when the workload increases; (ii) failures are common with deployments on hundreds of VMs—systems must be *fault-tolerant* with fast recovery times, yet low per-machine overheads. An open question is how to achieve these two goals when stream queries include *stateful* operators, which must be scaled out and recovered without affecting query results.

Our key idea is to expose internal operator state explicitly to the SPS through a set of state management primitives. Based on them, we describe an integrated approach for dynamic scale out and recovery of stateful operators. Externalised operator state is checkpointed periodically by the SPS and backed up to upstream VMs. The SPS identifies individual operator bottlenecks and automatically scales them out by allocating new VMs and partitioning the checkpointed state. At any point, failed operators are recovered by restoring checkpointed state on a new VM and replaying unprocessed tuples. We evaluate this approach with the Linear Road Benchmark on the Amazon EC2 cloud platform and show that it can scale automatically to a load factor of L=350 with 50 VMs, while recovering quickly from failures.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems

low latency results. For example, web companies such as Facebook and LinkedIn execute daily data mining queries to analyse their latest web logs [25]; online marketplace providers such as eBay and BetFair run sophisticated fraud detection algorithms on real-time trading activity [24]; and scientific experiments require on-the-fly processing of data.

Therefore *stream processing systems* (SPSs) have evolved from cluster-based systems, deployed on a few dozen machines [1], to extremely scalable architectures for big data processing, spanning hundreds of servers. Scalable SPSs such as Apache S4 [23] and Twitter Storm [29] parallelise the execution of stream queries to exploit *intra-query parallelism*. By scaling out partitioned query operators horizontally, they can support high input stream rates and queries with computationally demanding operators.

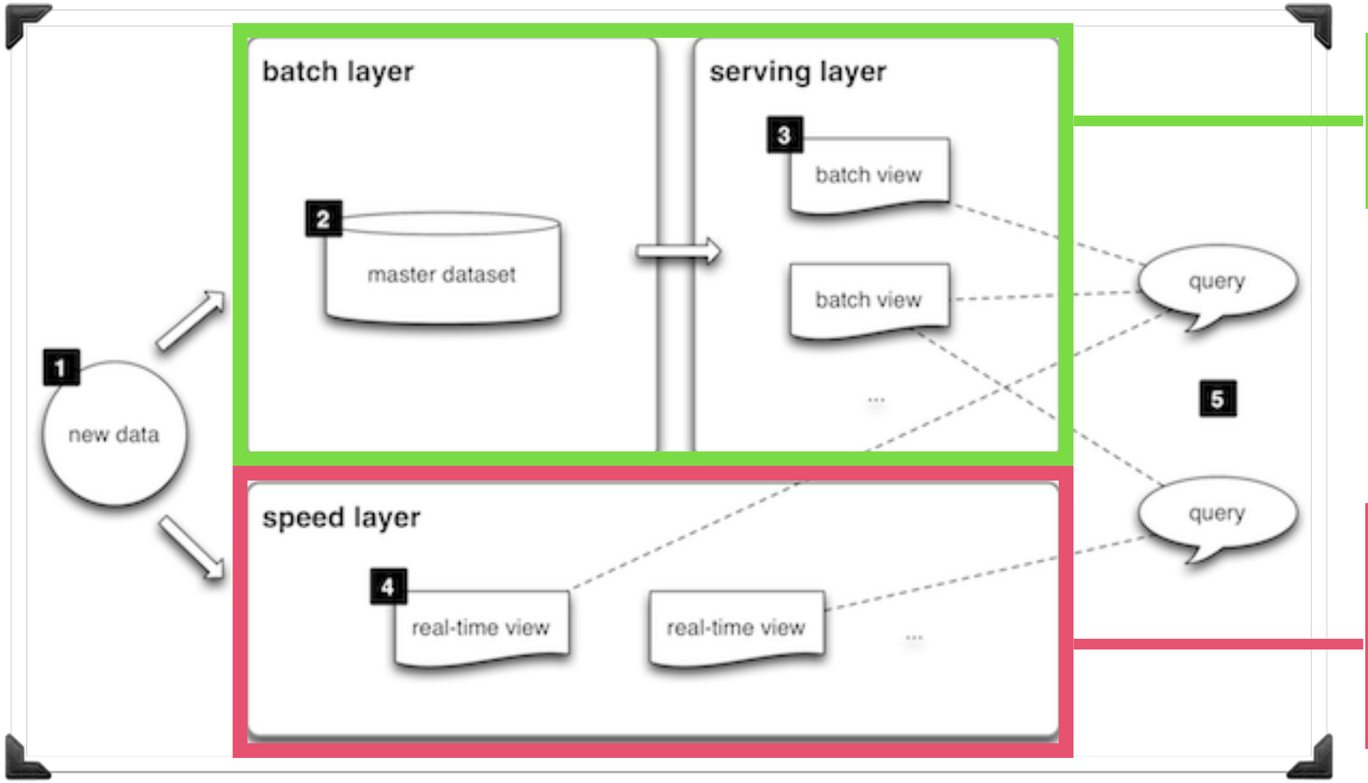
A cloud computing model offers SPSs access to a virtually unlimited number of virtual machines (VMs). To gain widespread adoption, however, cloud-hosted systems must hide the complexity of data parallelism and failure recovery from users, as evidenced by the popularity of parallel batch processing systems such as MapReduce. Therefore cloud-hosted SPSs face the same two fundamental challenges:

1. On-demand parallelism. To reduce financial costs under "pay-as-you-go" pricing models in public cloud environments such as Amazon EC2 and Rackspace, an SPS should acquire resources *on demand*. It should request additional VMs at runtime, reacting to changes in the processing workload and repartitioning query operators accordingly.

2. Resource-efficient failure recovery. A cloud-deployed SPS with hundreds of VMs is likely to suffer from failure. It must therefore be fault-tolerant, i.e. be able to recover from failures without affecting processing results. Due to the size of deployments, the per-machine resource overhead of any

The Lambda Architecture (~2013)

An artificial bound to stream processing



- Production-Ready
- Critical computation

- Experimental
- Non-critical computation

http://lambda-architecture.net/

What Consistency ?

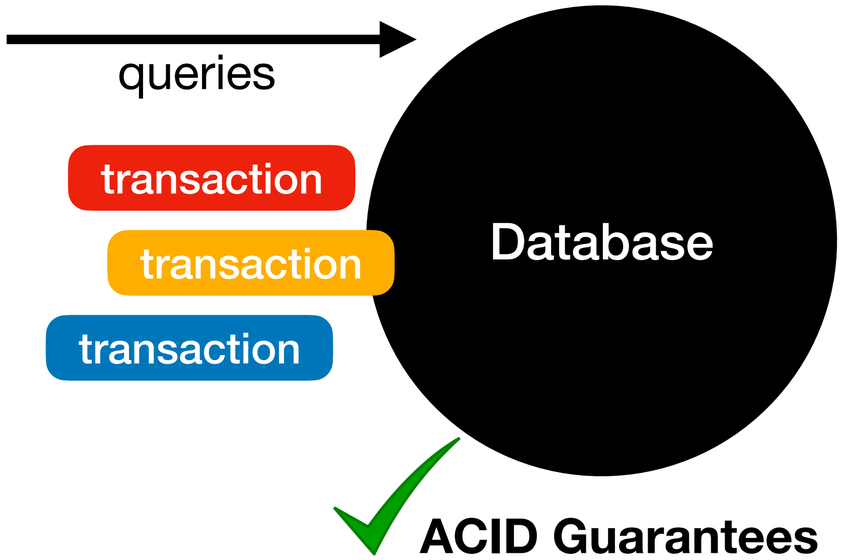
Problem #1 - No concrete proof of concept in reliable S.P. up to 2013

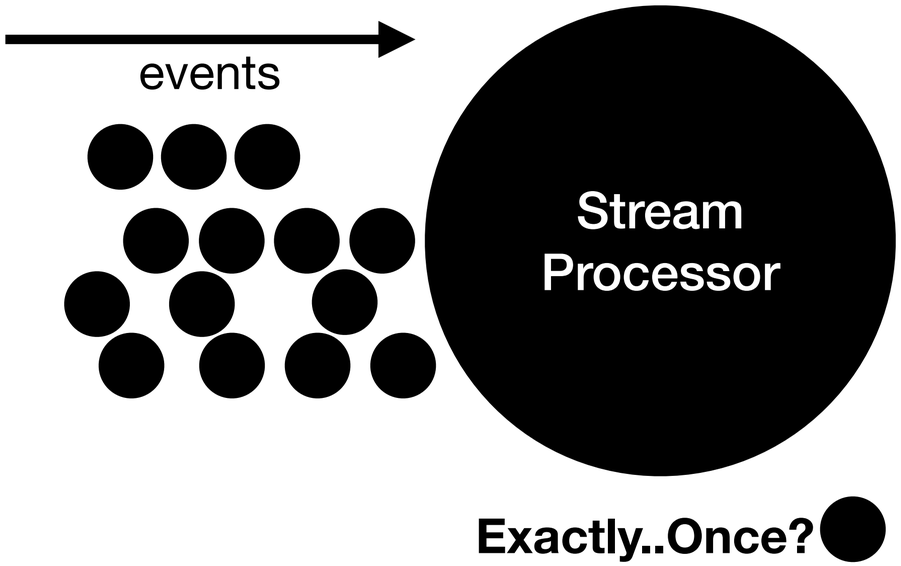
- Hadoop, Spark etc. had set an industry standard: batch atomic commits

- Existing Scalable Stream Processors (Storm) focused on input replays.
- Borealis etc. defined task but not application-wide failure-recovery protocols.
- No formal specification of consistency properties

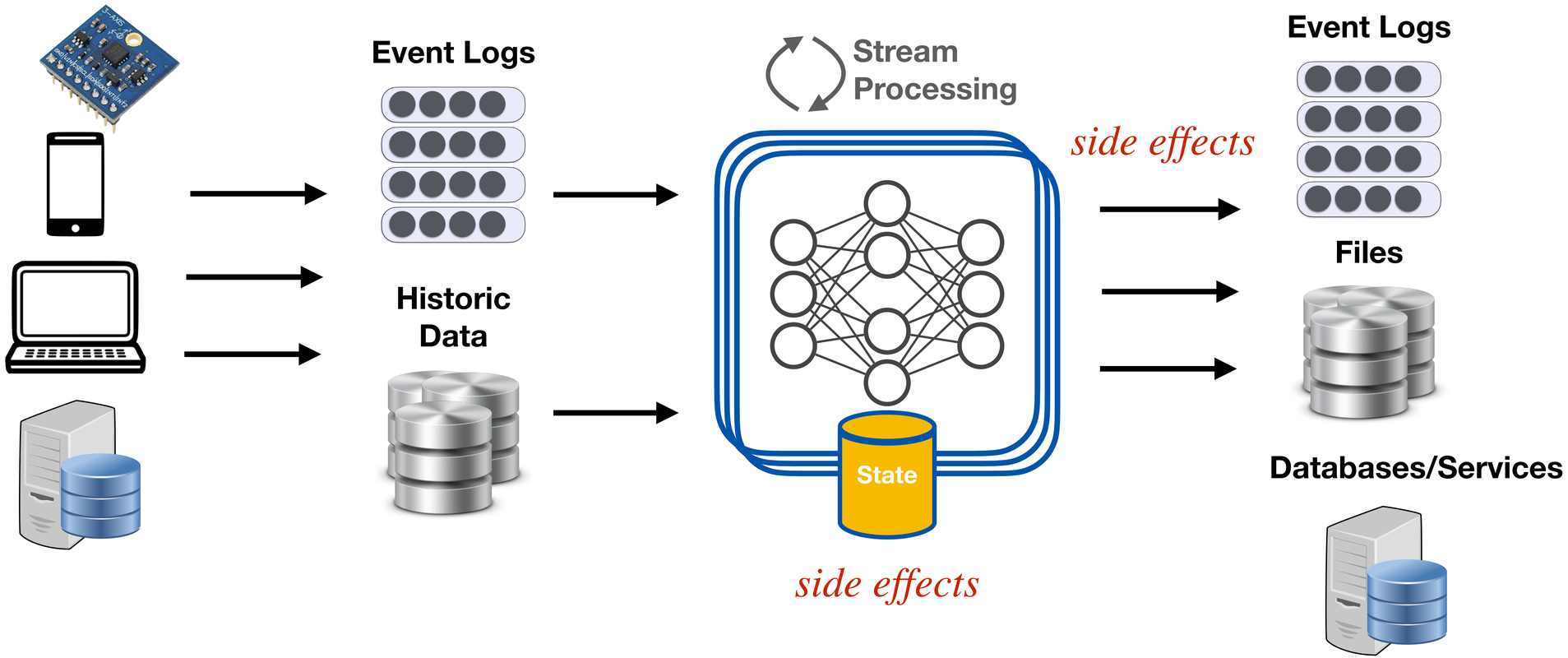
What is Consistency ?

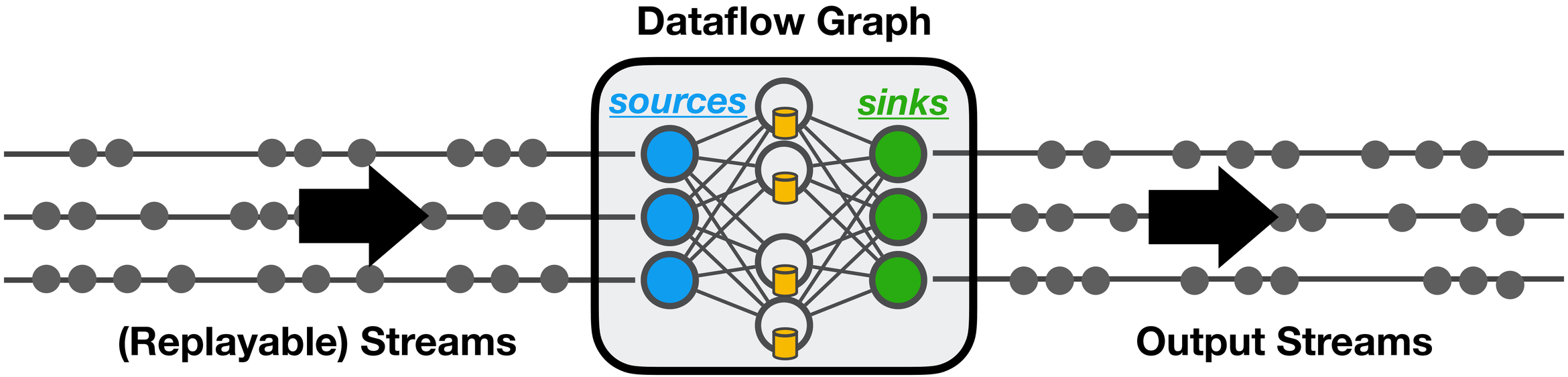
Problem #2 - No clear picture of what consistency even means



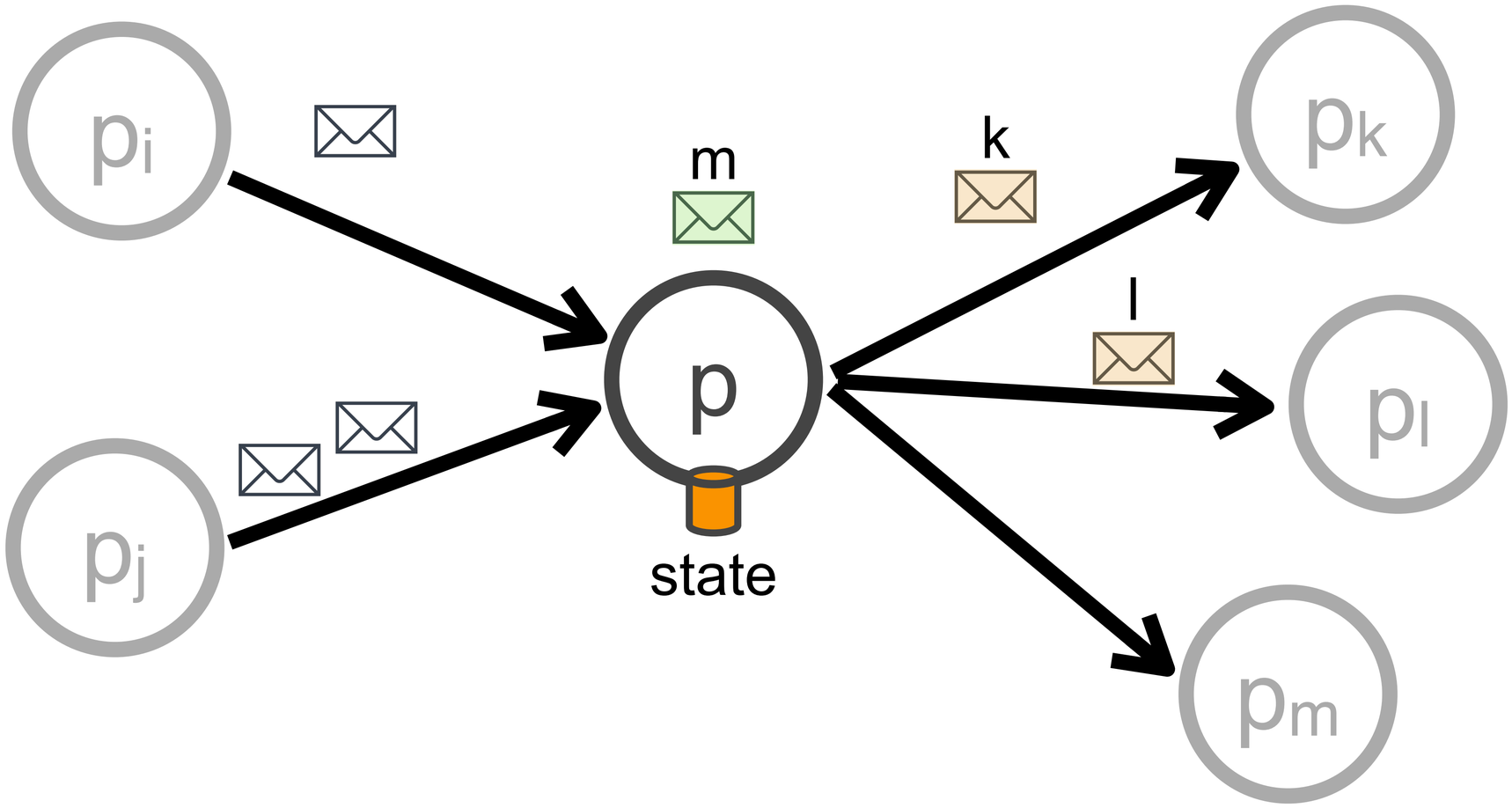


Foundational Step: Define “Transaction” in Data Stream Processing?

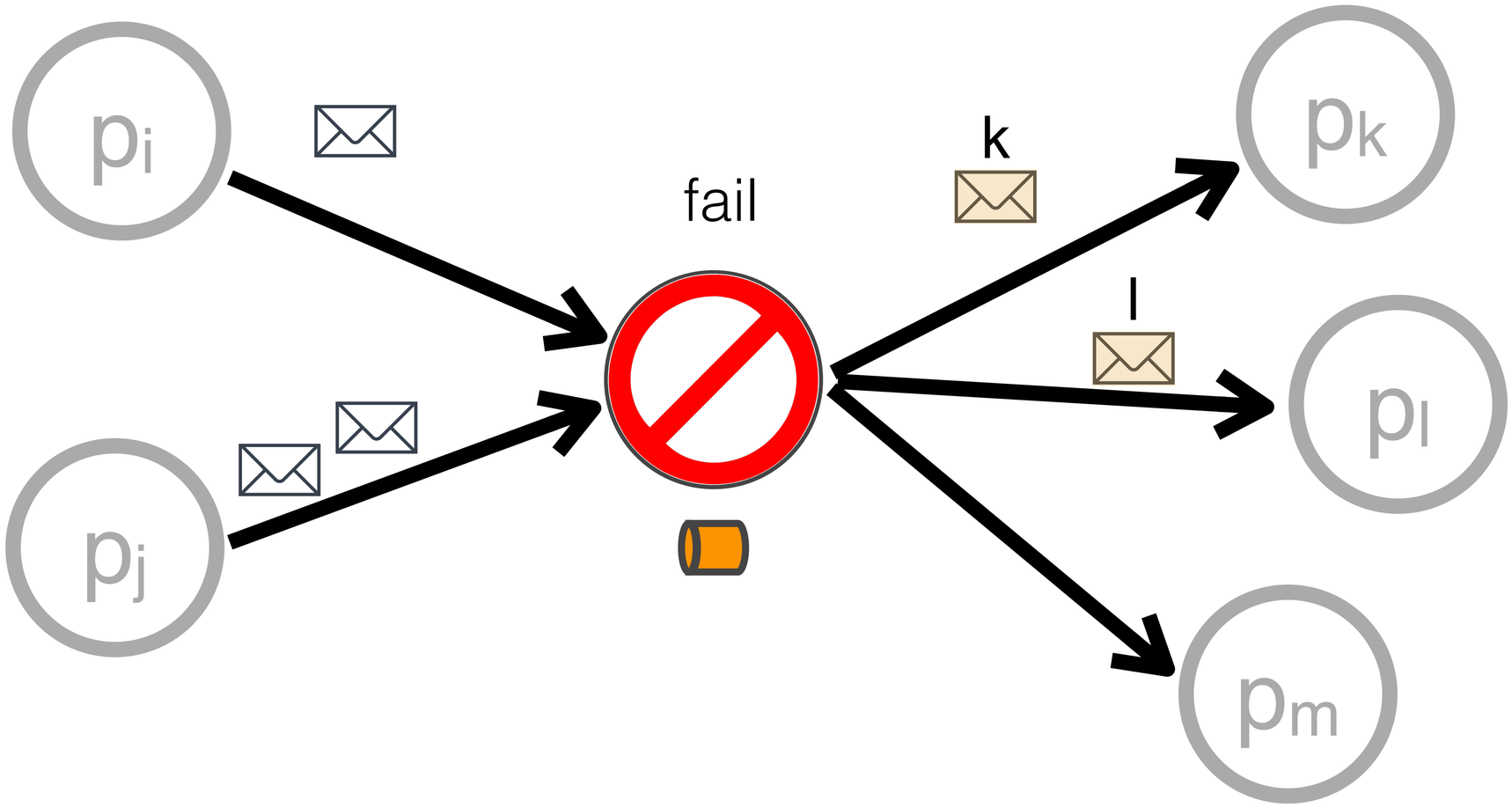

Event Logs
Historic Data
Stream Processing
State
side effects
side effects
Event Logs
Files
Databases/Services


Dataflow Graph
sources
sinks
(Replayable) Streams
Output Streams

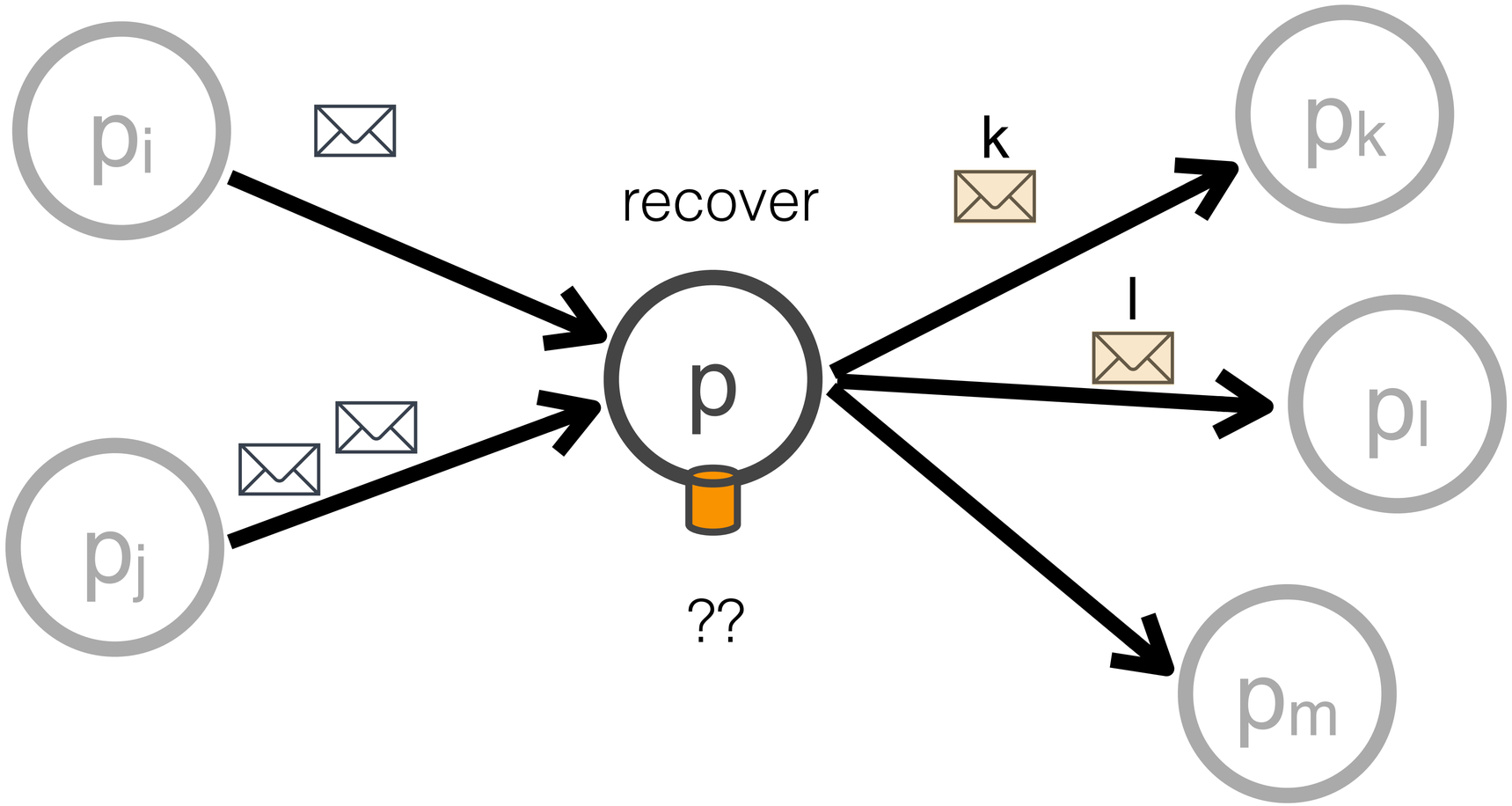
Actions and Failures



Actions and Failures



Actions and Failures

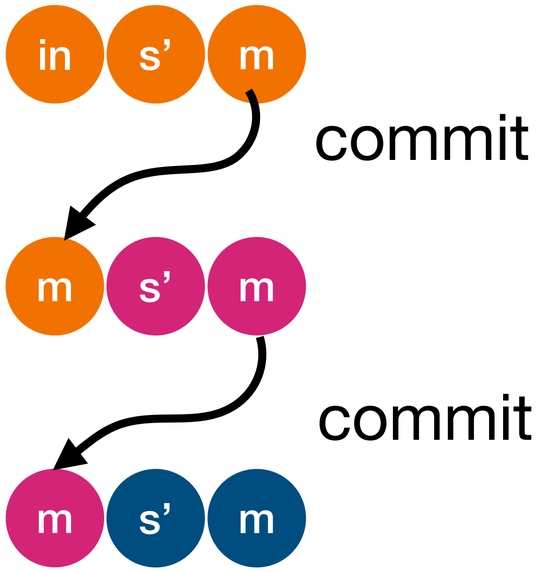


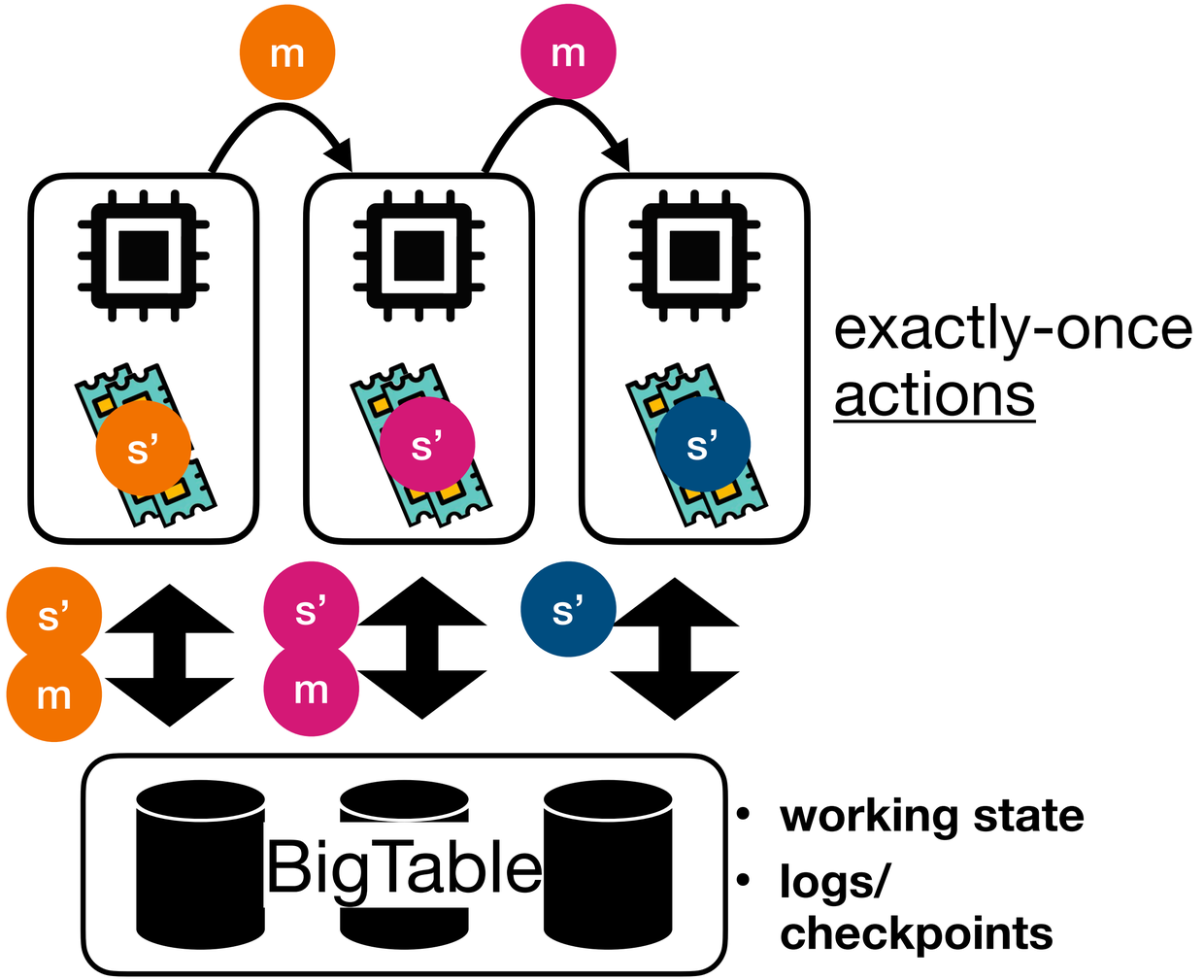
- Has m been fully processed?
- Have k and l been delivered? Have they caused other actions?

- Transaction =
  - processing **action (in,out,state) - Google Millwheel**
  - stream **epoch/batch - Spark Streaming, Flink, S-Store, IBM Streams, etc.**

Action-Level Transactional Stream Processing

Millwheel's strong productions

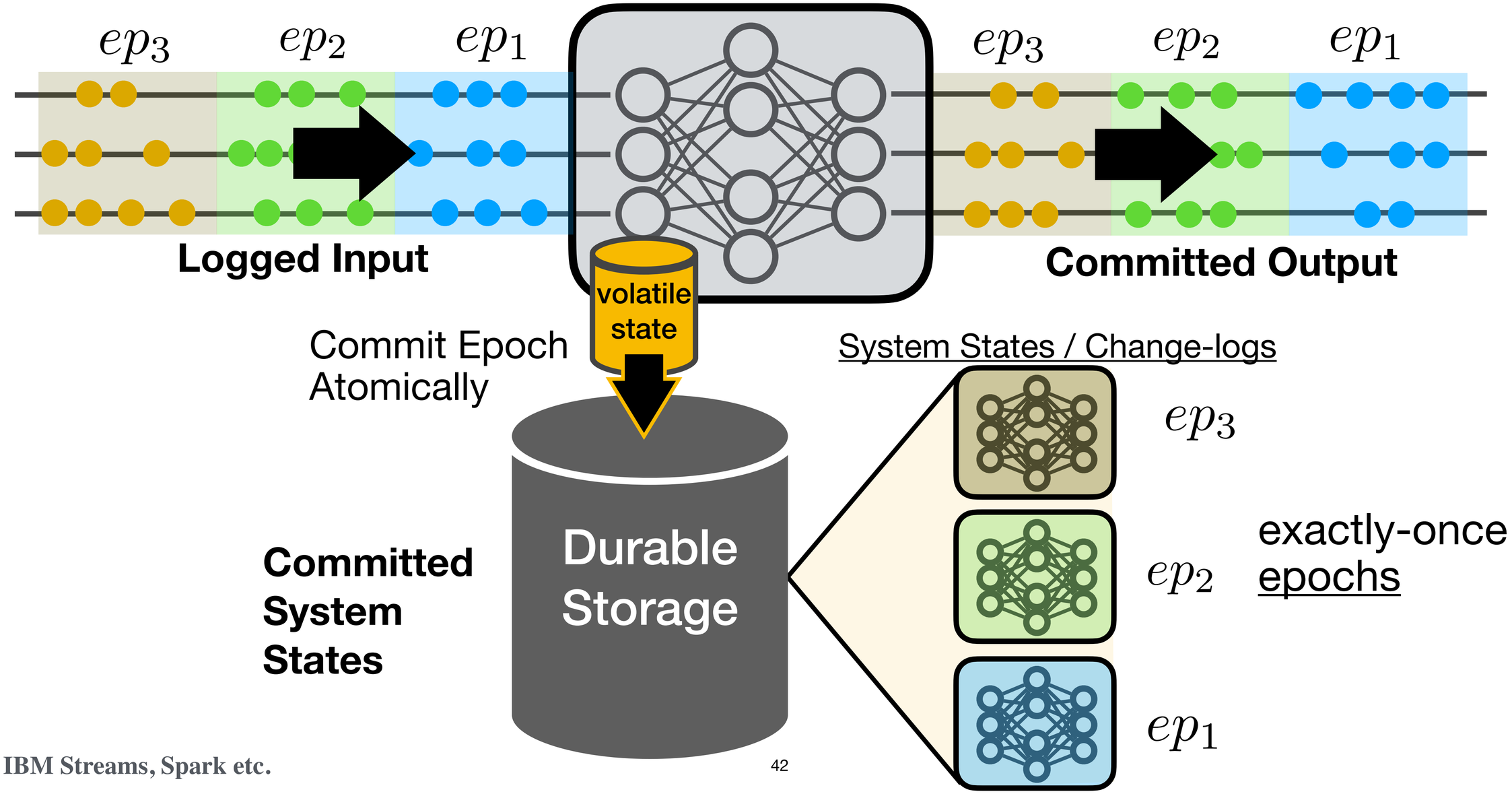




Millwheel: Fault-tolerant stream processing at internet scale

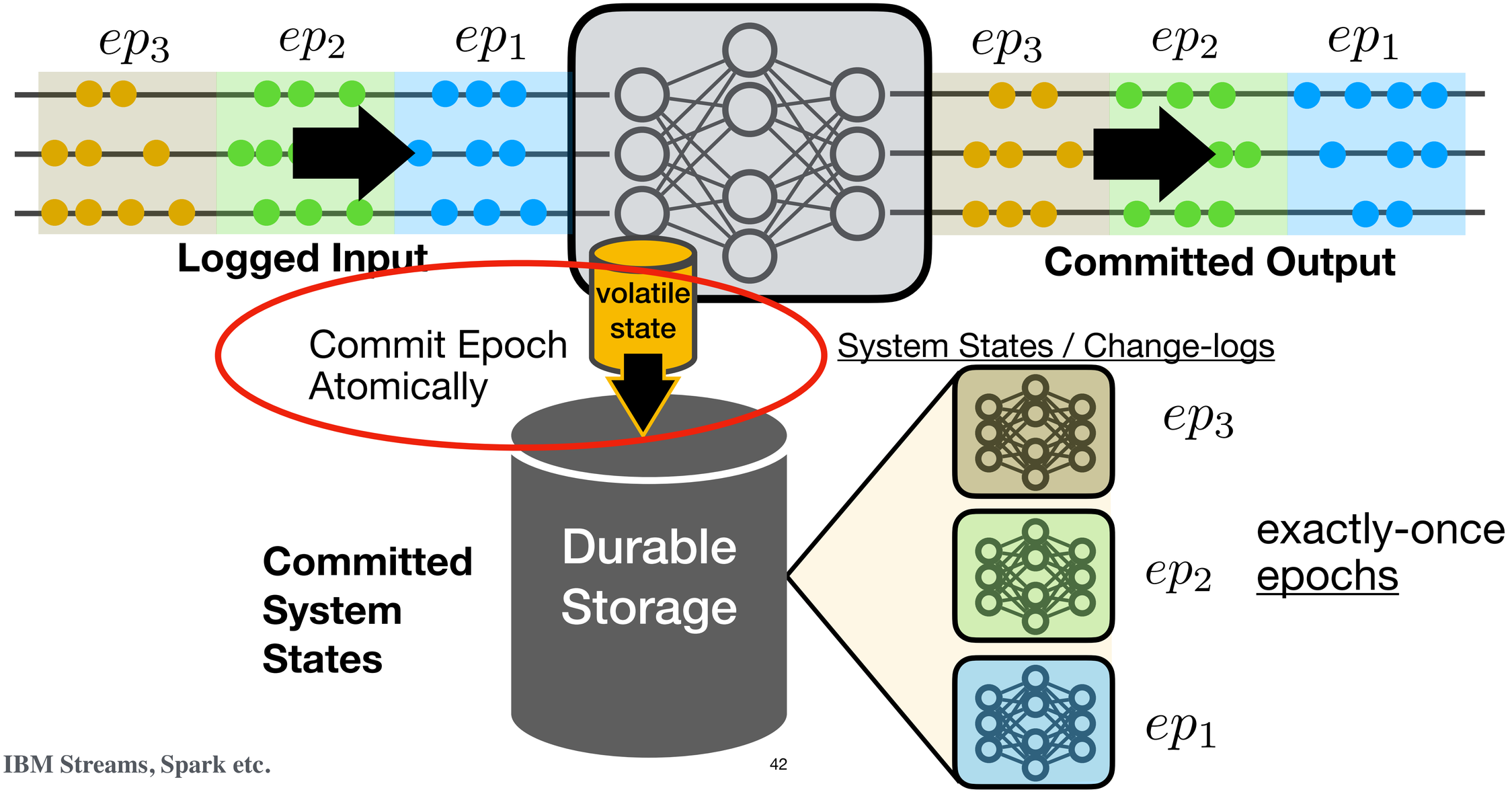
Akidau T, Balikov A, Bekiroğlu K, et. al. in Proceedings of the VLDB Endowment (2013)

Epoch-Level Transactional Stream Processing



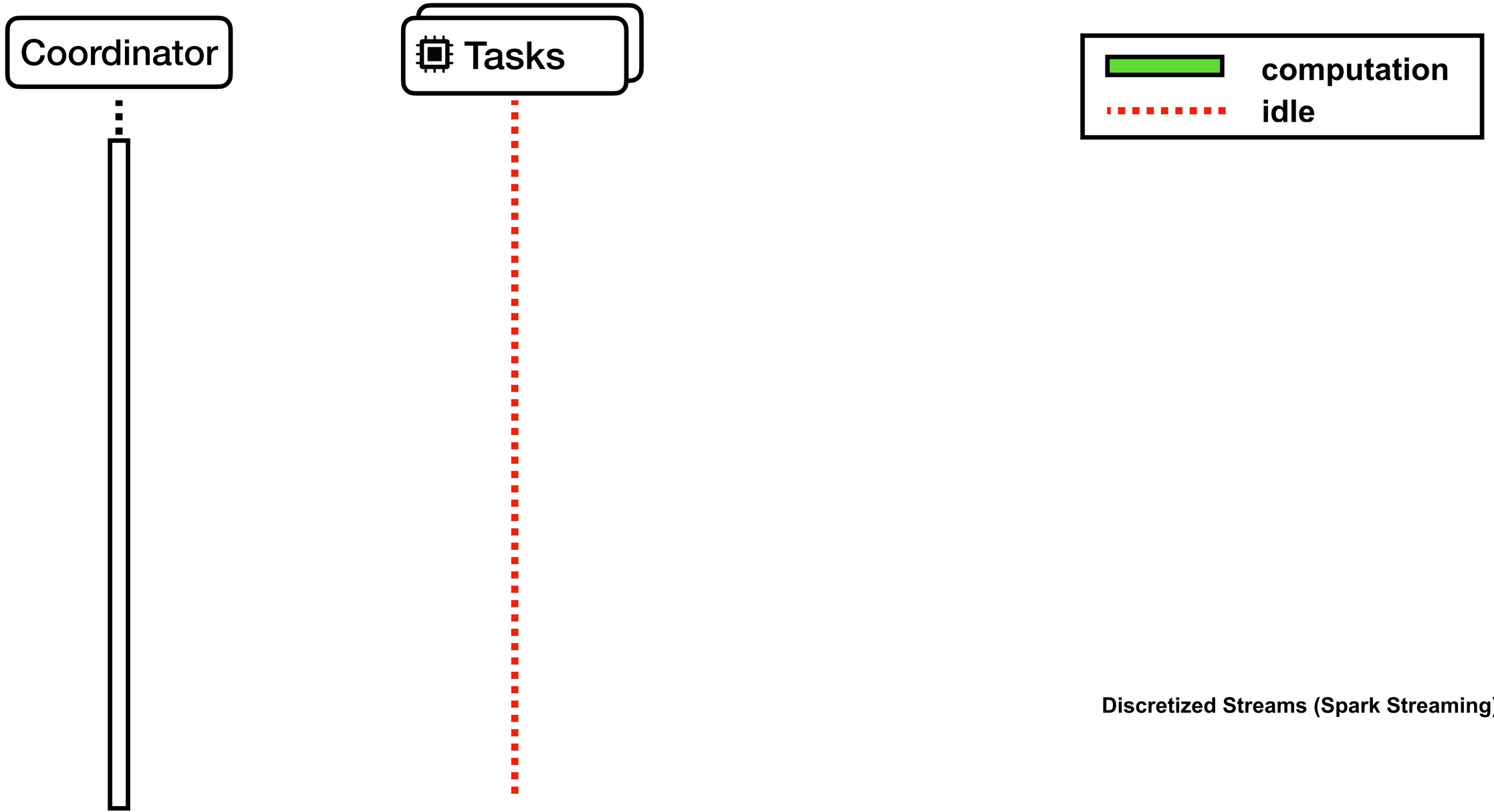
Flink, IBM Streams, Spark etc.

Epoch-Level Transactional Stream Processing



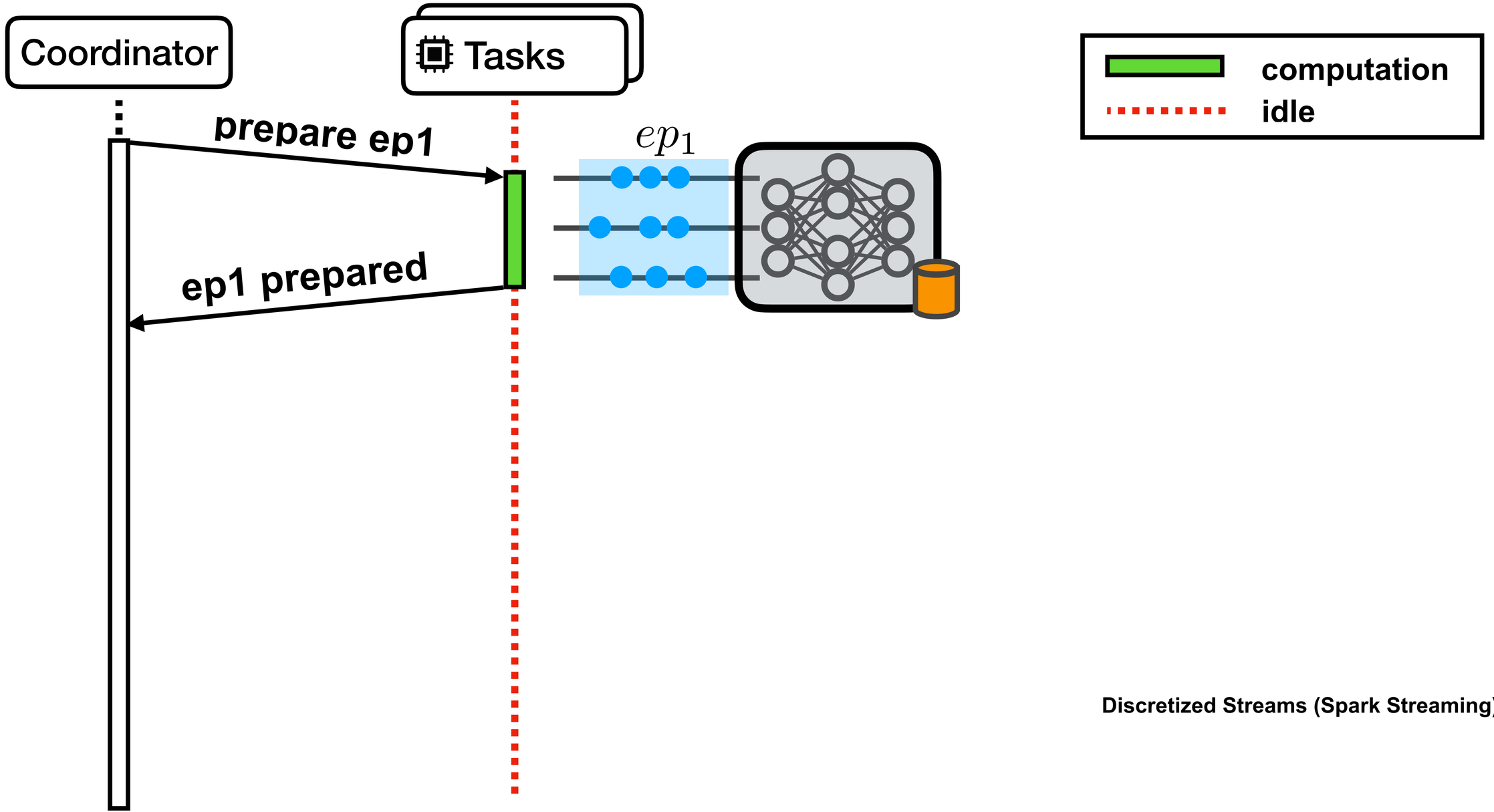
Flink, IBM Streams, Spark etc.

Variant 1: Synchronous Epoch Commits



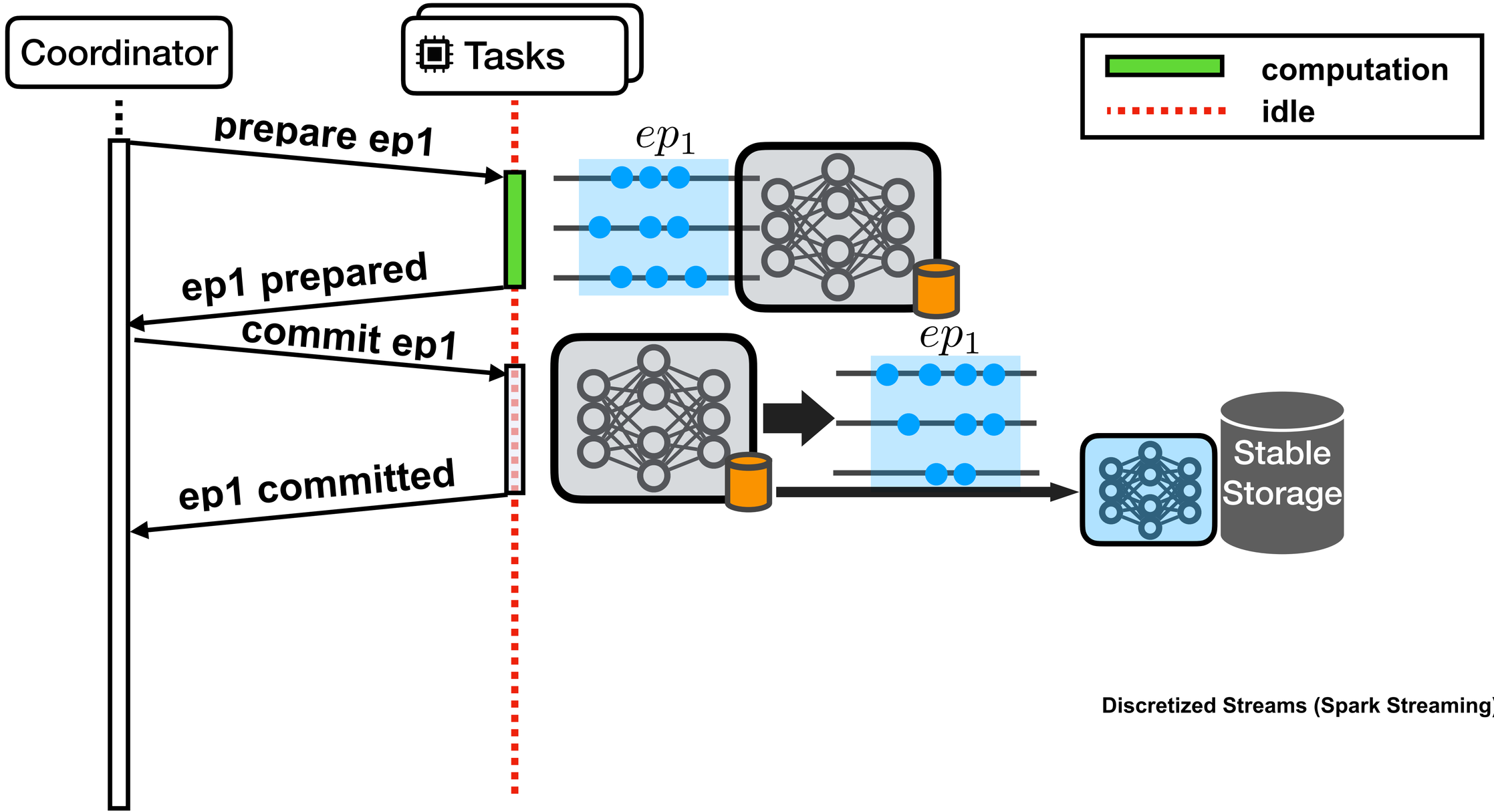
Discretized Streams (Spark Streaming)

Variant 1: Synchronous Epoch Commits



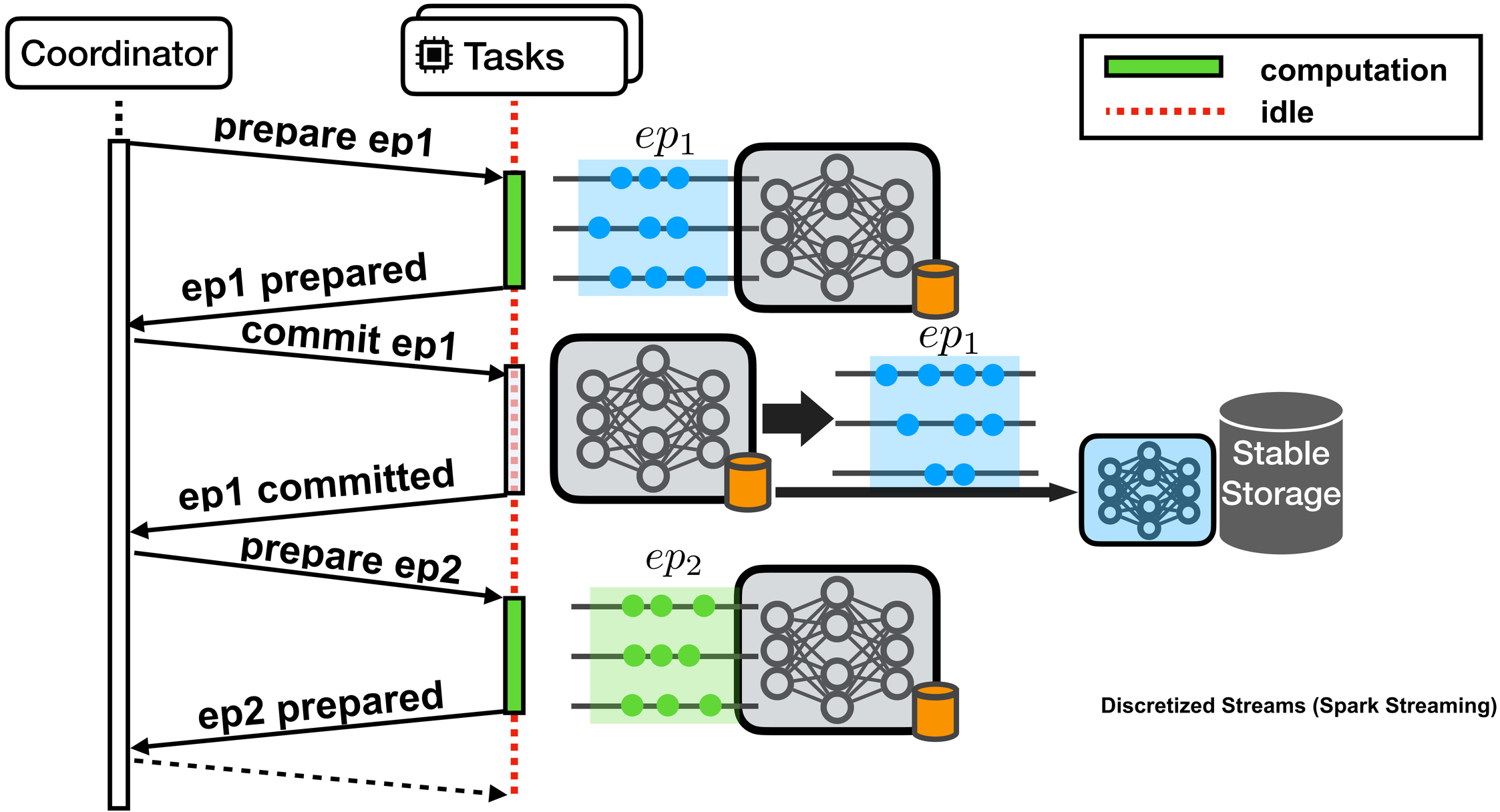
Discretized Streams (Spark Streaming)

Variant 1: Synchronous Epoch Commits



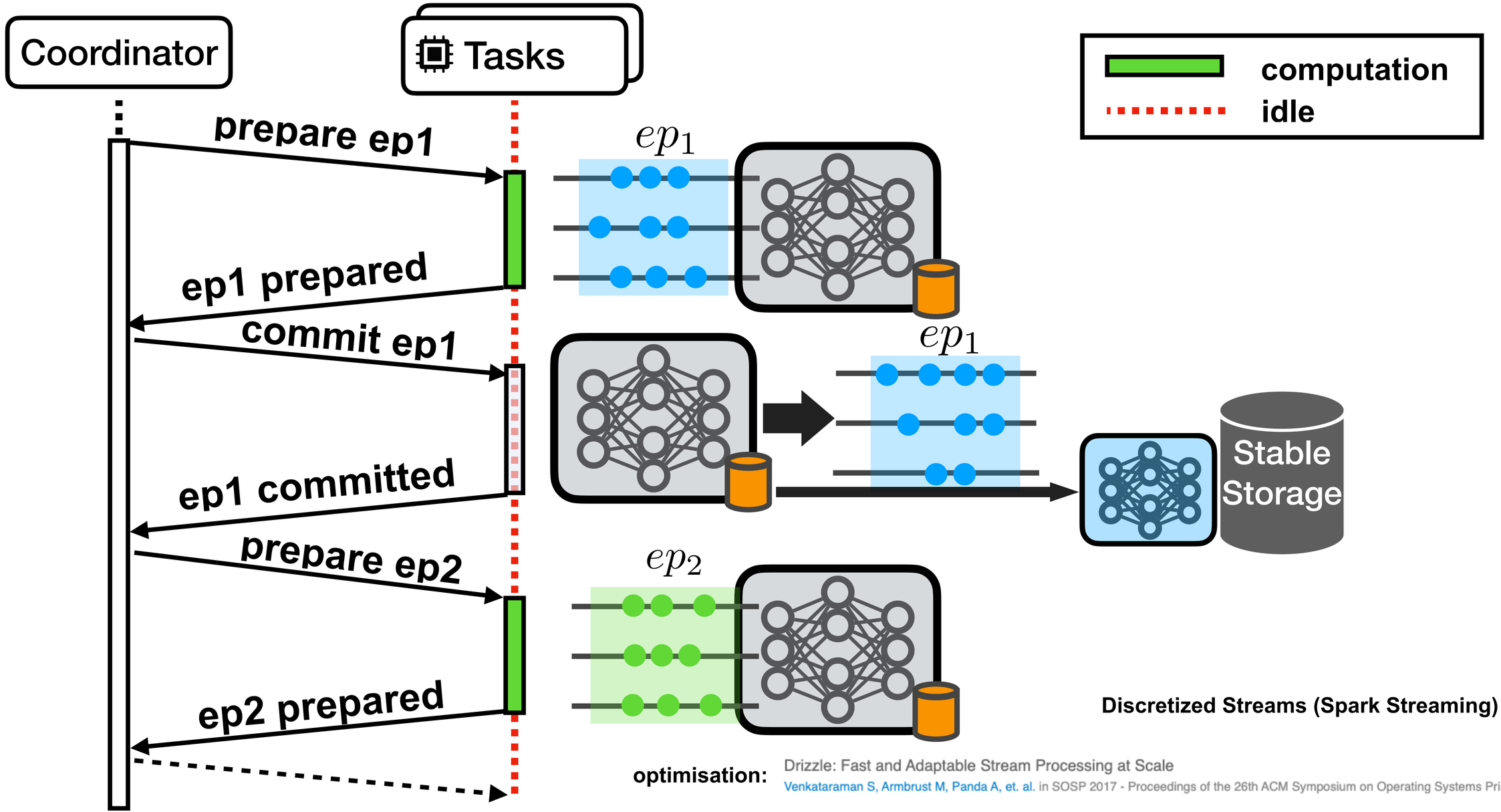
Discretized Streams (Spark Streaming)

Variant 1: Synchronous Epoch Commits



Discretized Streams (Spark Streaming)

Variant 1: Synchronous Epoch Commits



Discretized Streams (Spark Streaming)

optimisation: Drizzle: Fast and Adaptable Stream Processing at Scale
Venkataraman S, Armbrust M, Panda A, et. al. in SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles (2017)

Variant 1: Synchronous Epoch Commits

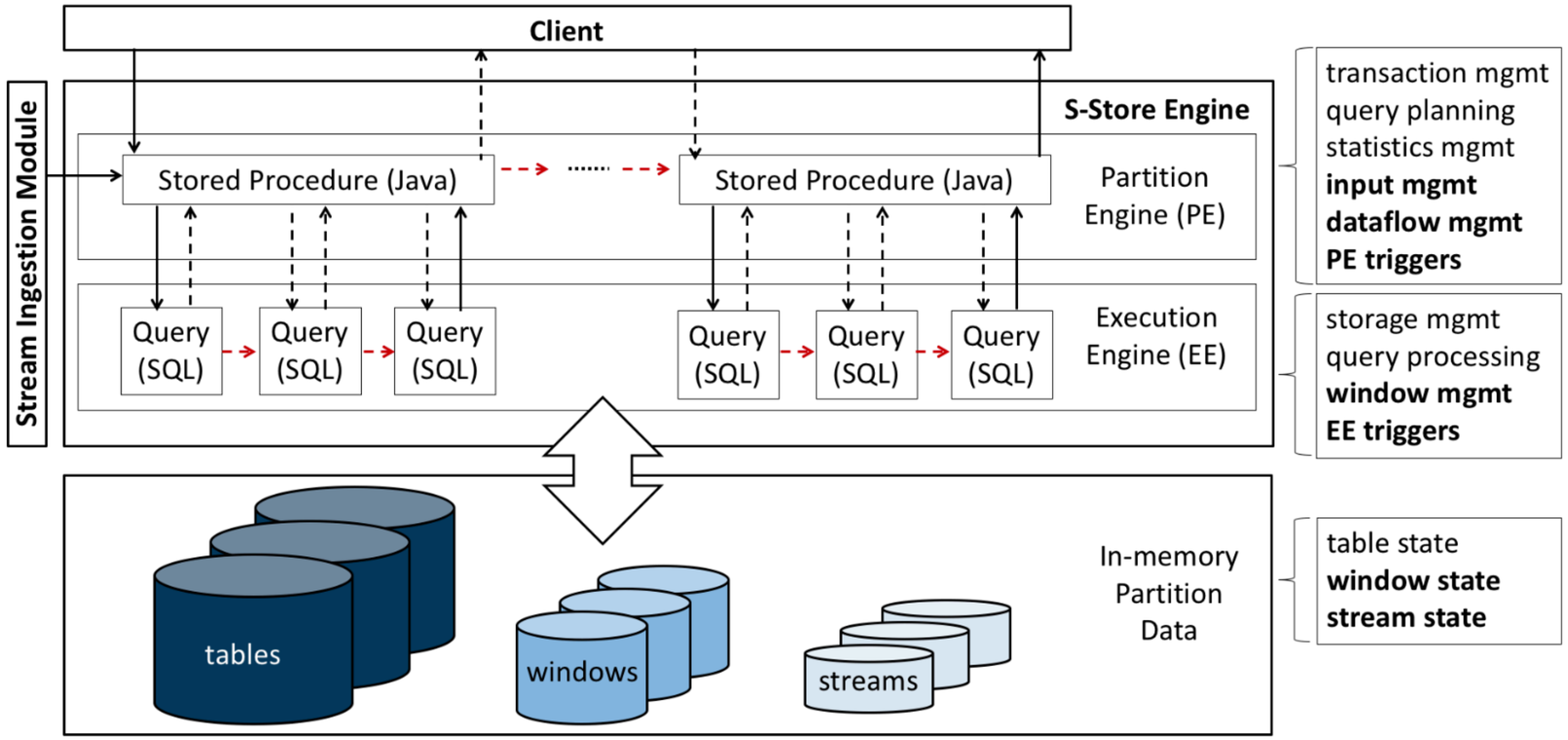


Figure 4: S-Store Architecture

S-Store Transactions executed by the DBMS

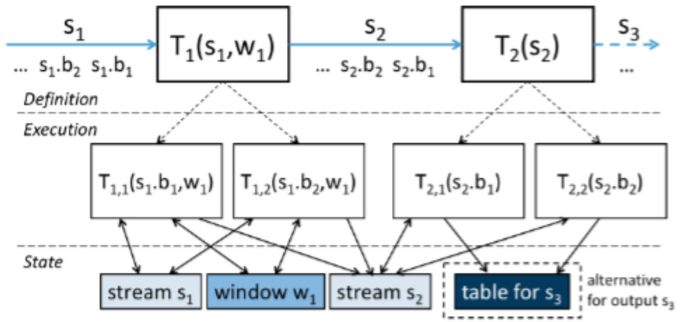
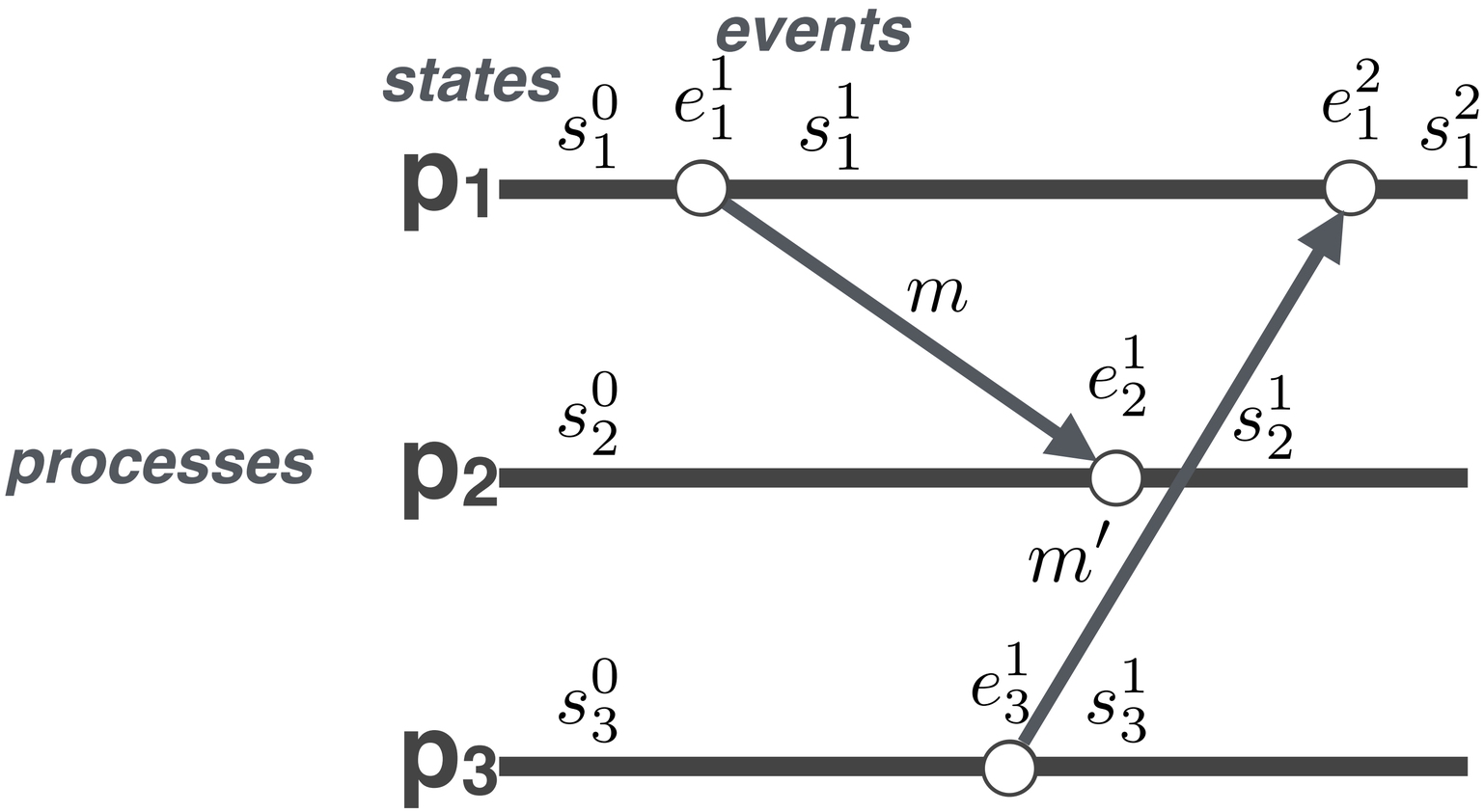


Figure 2: Transaction Executions in a Dataflow Graph

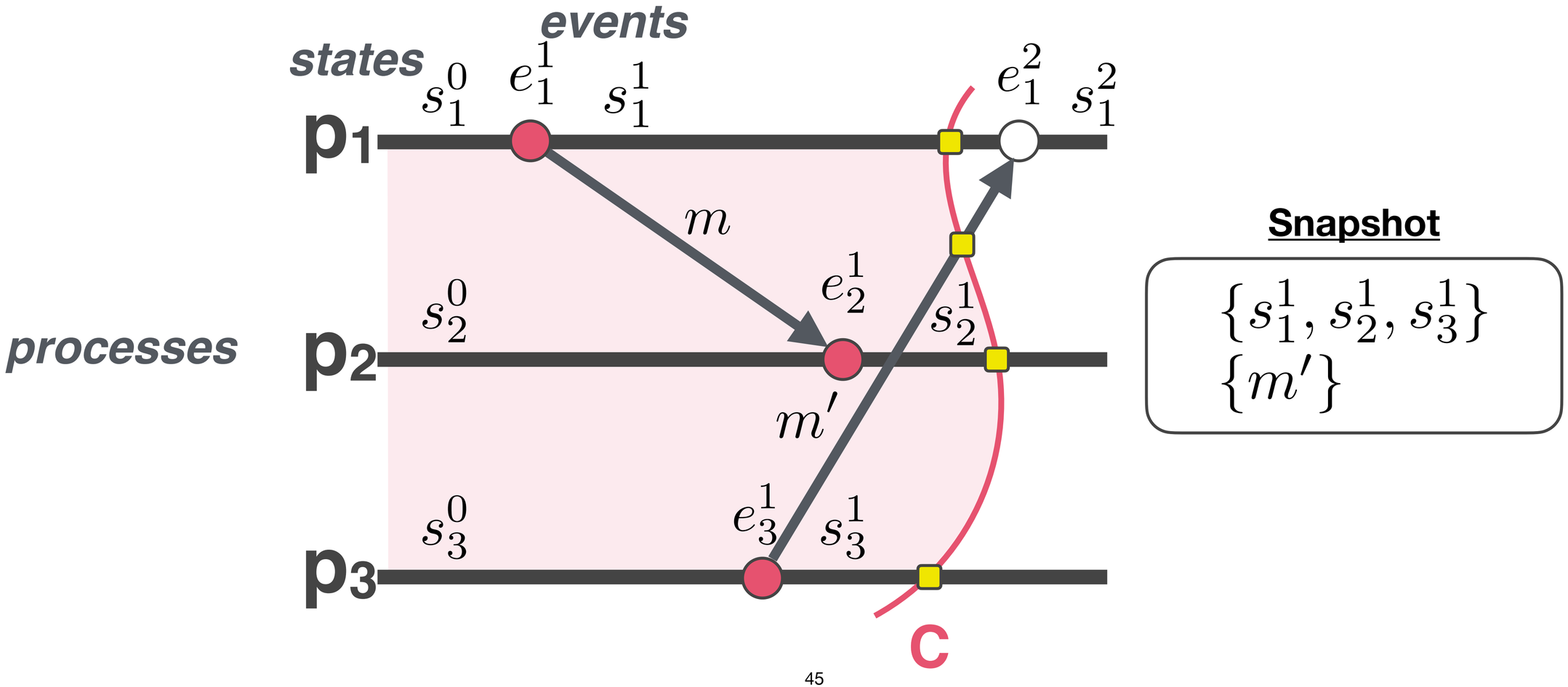
Distributed Snapshotting

Based on Consistent Snapshotting Protocols: Distributed Algorithms that capture causality-compliant system states (**consistent cuts** in a system execution)



Distributed Snapshotting

Based on Consistent Snapshotting Protocols: Distributed Algorithms that capture causality-compliant system states (**consistent cuts** in a system execution)

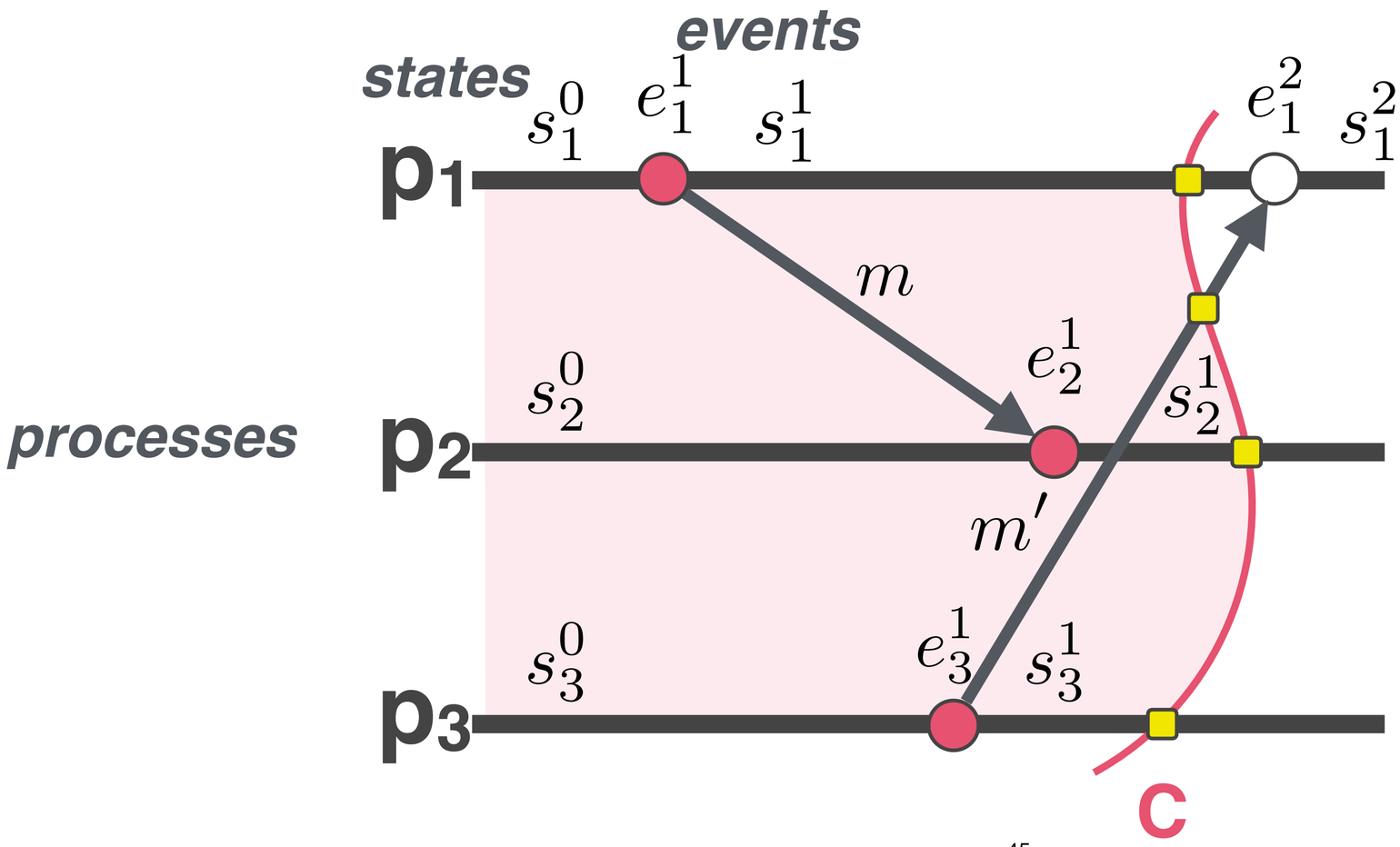


Snapshot

$\{s_1^1, s_2^1, s_3^1\}$
$\{m'\}$

Distributed Snapshotting

Based on Consistent Snapshotting Protocols: Distributed Algorithms that capture causality-compliant system states (**consistent cuts** in a system execution)



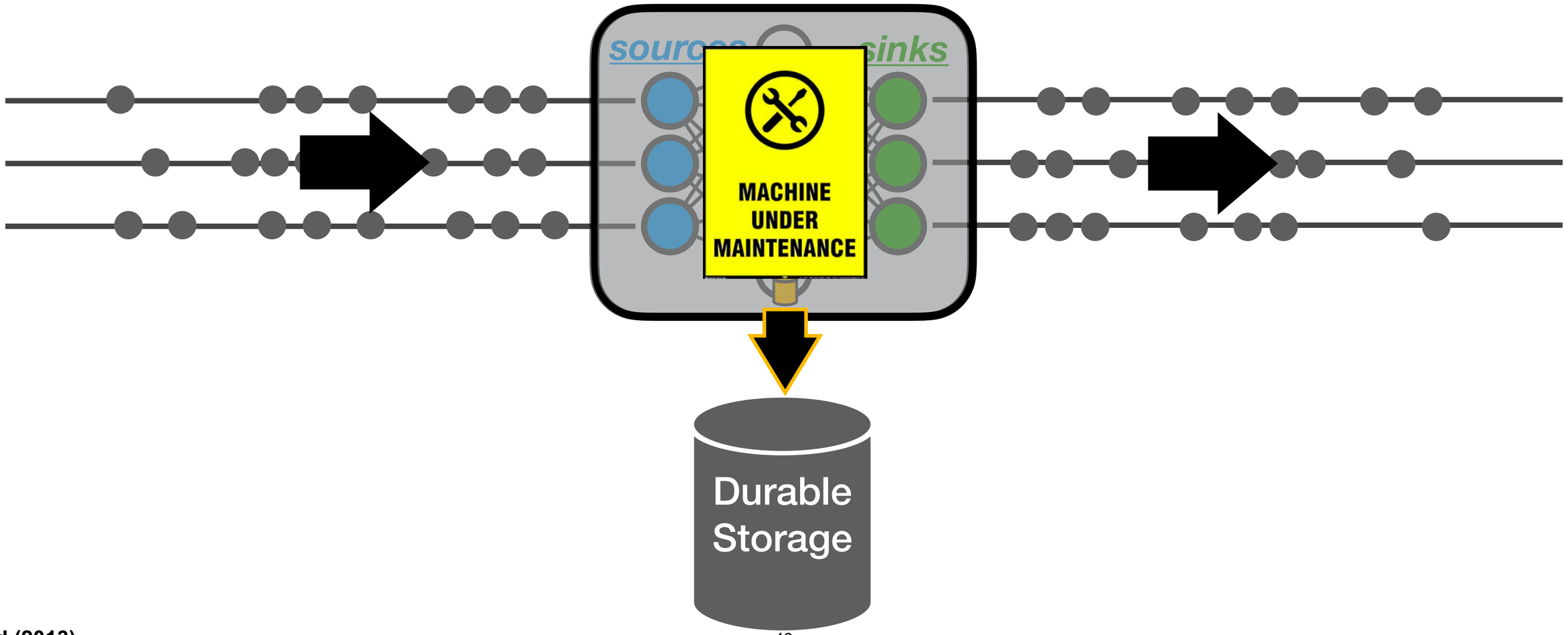
Snapshot

$\{s_1^1, s_2^1, s_3^1\}$

$\{m'\}$

m occurred during snapshot (~redo log)

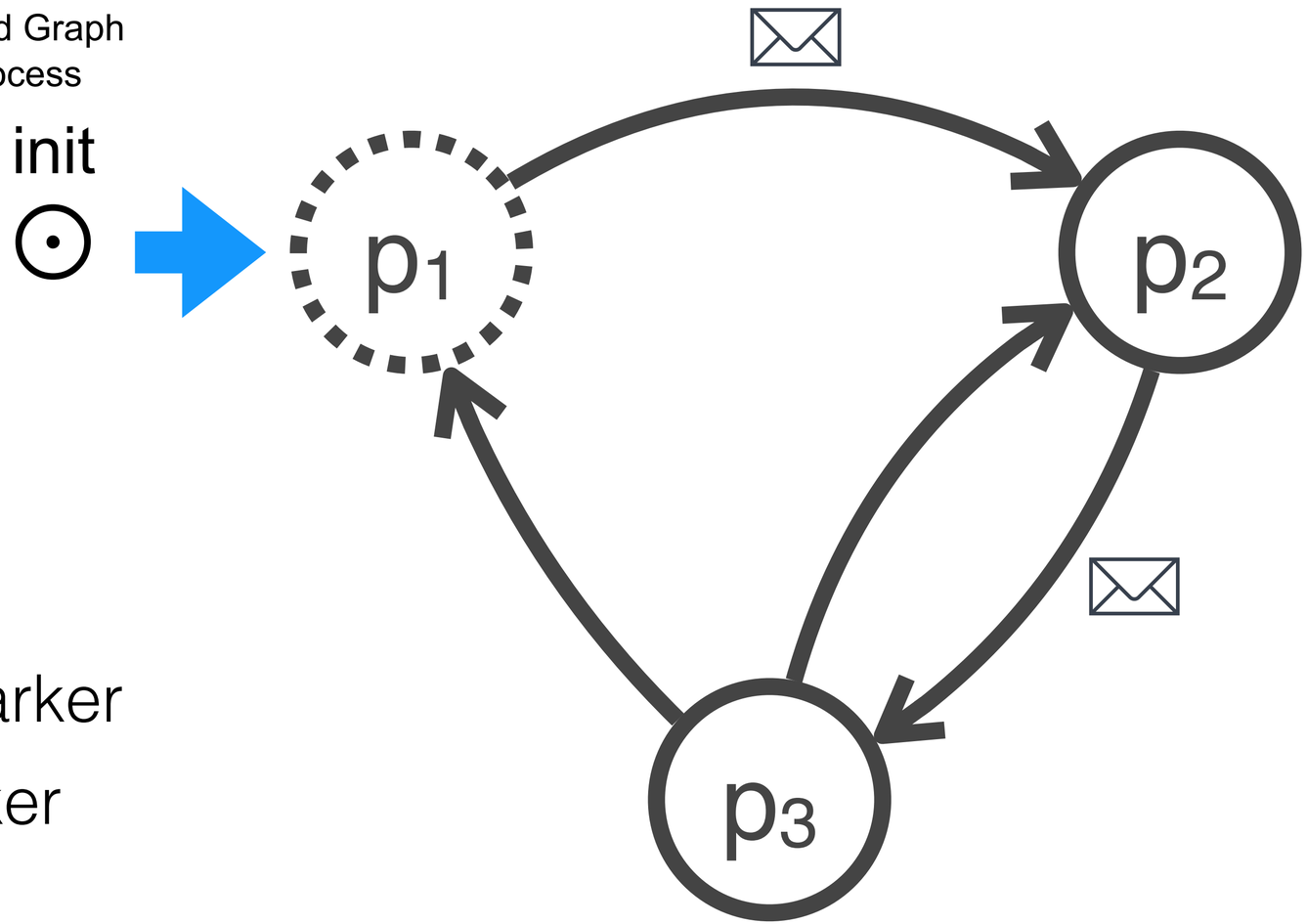
Variant 2.1 - Pause All Tasks/IO and Snapshot



Naiad (2013)

Variant 2.2 - Unaligned Epoch Snapshots

- Assumptions:
  - FIFO Reliable Channels
  - Strongly Connected Graph
  - Single Initiating Process



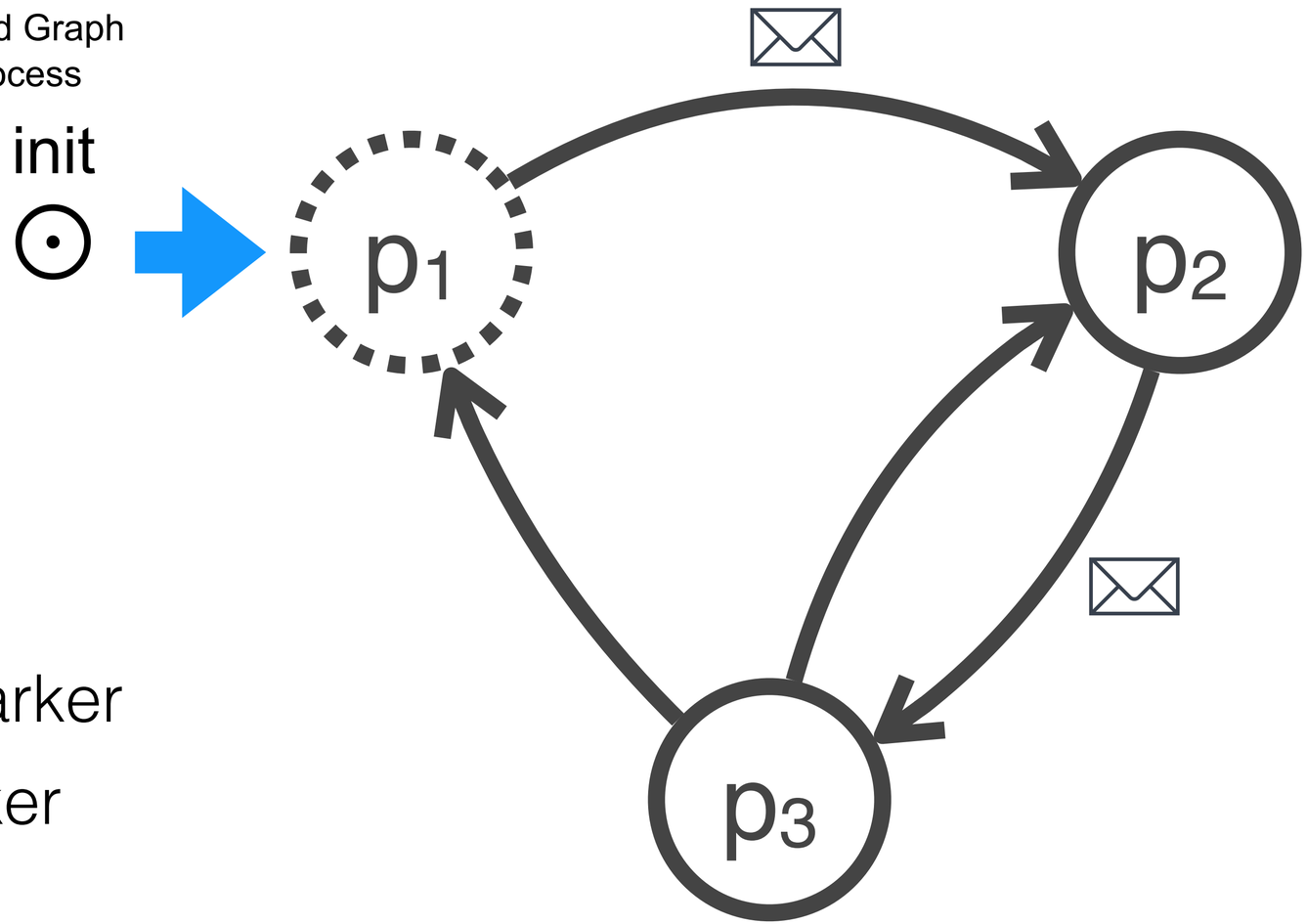
before marker

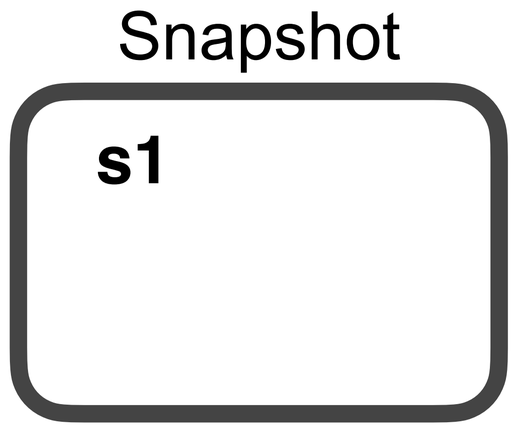
after marker

~IBM Streams
~Flink Unaligned Checkpoints (experimental)

Variant 2.2 - Unaligned Epoch Snapshots

- Assumptions:
  - FIFO Reliable Channels
  - Strongly Connected Graph
  - Single Initiating Process





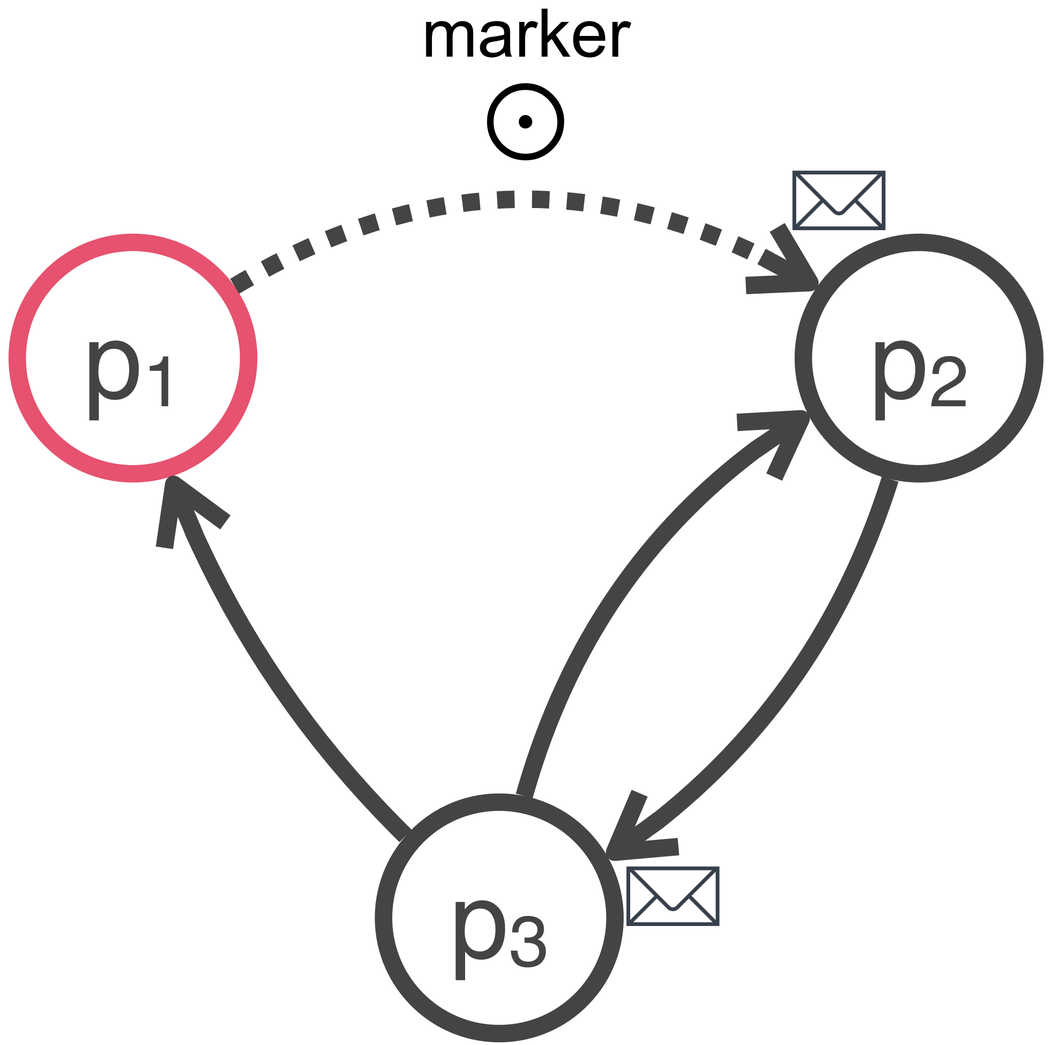
before marker

after marker

~IBM Streams
~Flink Unaligned Checkpoints (experimental)

Variant 2.2 - Unaligned Epoch Snapshots

- Assumptions:
  - FIFO Reliable Channels
  - Strongly Connected Graph
  - Single Initiating Process



Snapshot

s1

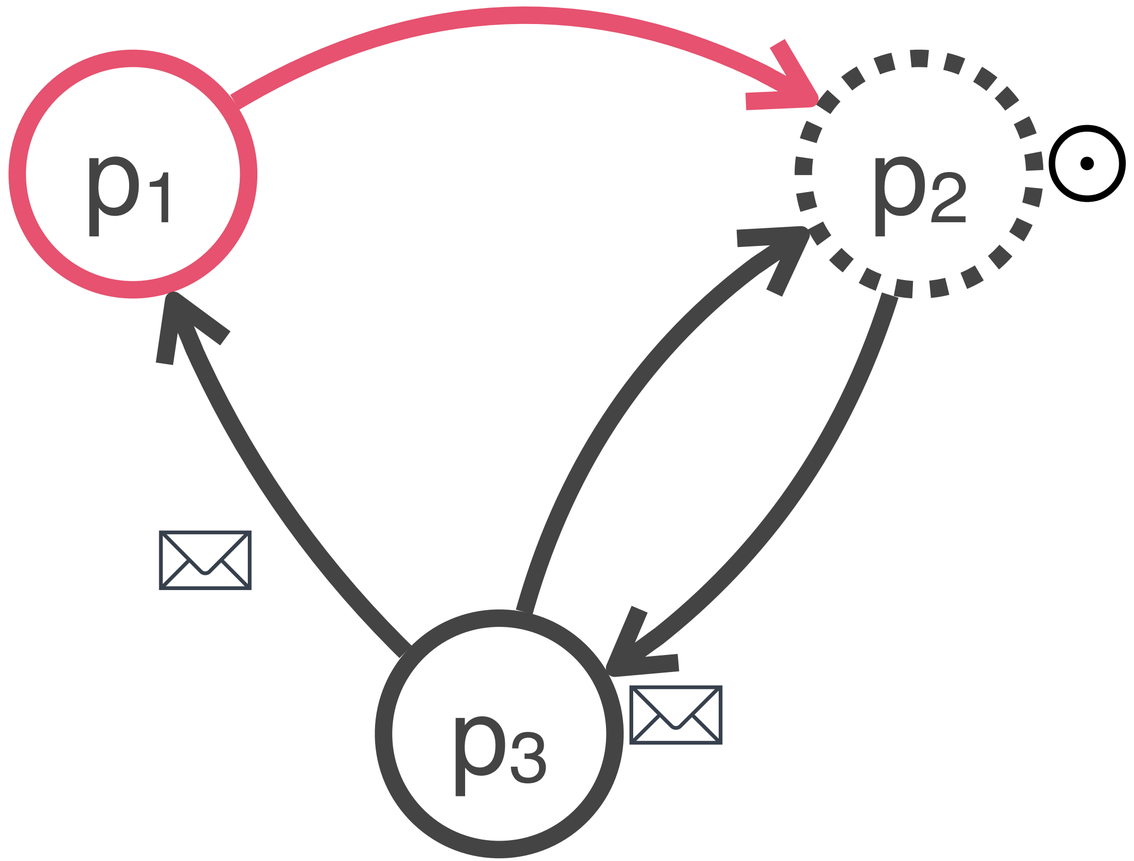
before marker

after marker

~IBM Streams
~Flink Unaligned Checkpoints (experimental)

Variant 2.2 - Unaligned Epoch Snapshots

- Assumptions:
  - FIFO Reliable Channels
  - Strongly Connected Graph
  - Single Initiating Process



Snapshot

s1, s2

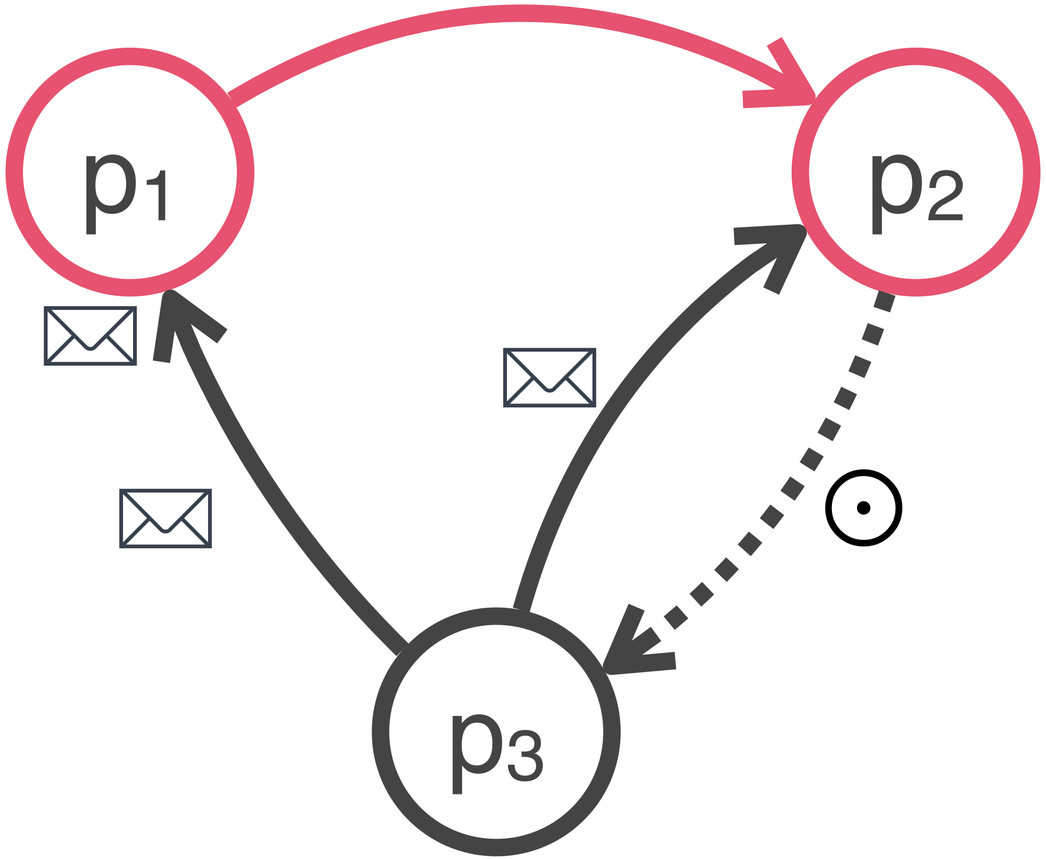
before marker

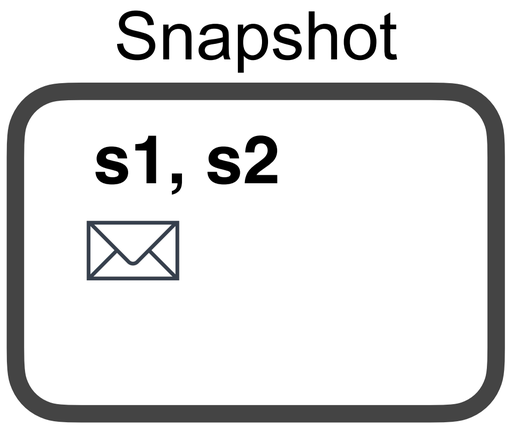
after marker

~IBM Streams
~Flink Unaligned Checkpoints (experimental)

Variant 2.2 - Unaligned Epoch Snapshots

- Assumptions:
  - FIFO Reliable Channels
  - Strongly Connected Graph
  - Single Initiating Process





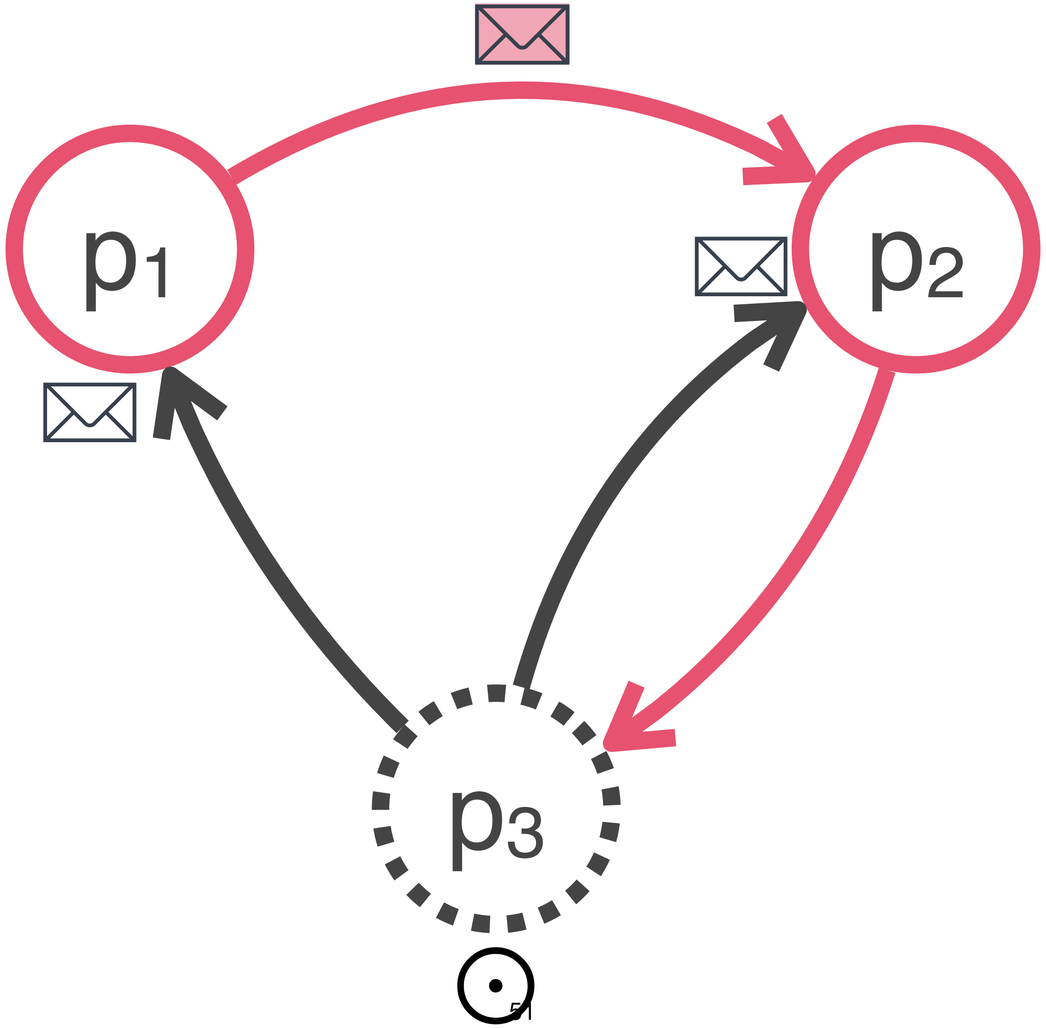
before marker

after marker

~IBM Streams
~Flink Unaligned Checkpoints (experimental)

Variant 2.2 - Unaligned Epoch Snapshots

- Assumptions:
  - FIFO Reliable Channels
  - Strongly Connected Graph
  - Single Initiating Process



Snapshot

s1, s2, s3

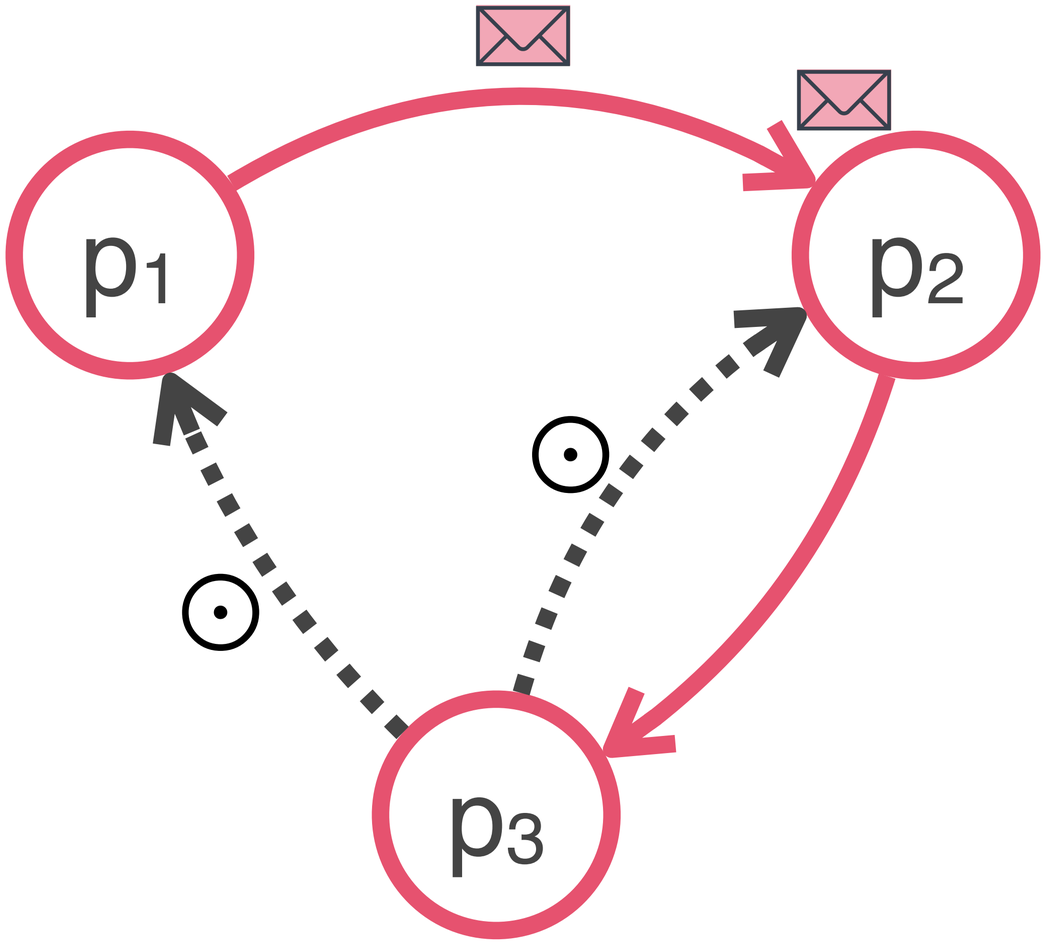
before marker

after marker

~IBM Streams
~Flink Unaligned Checkpoints (experimental)

Variant 2.2 - Unaligned Epoch Snapshots

- Assumptions:
  - FIFO Reliable Channels
  - Strongly Connected Graph
  - Single Initiating Process



Snapshot

s1, s2, s3

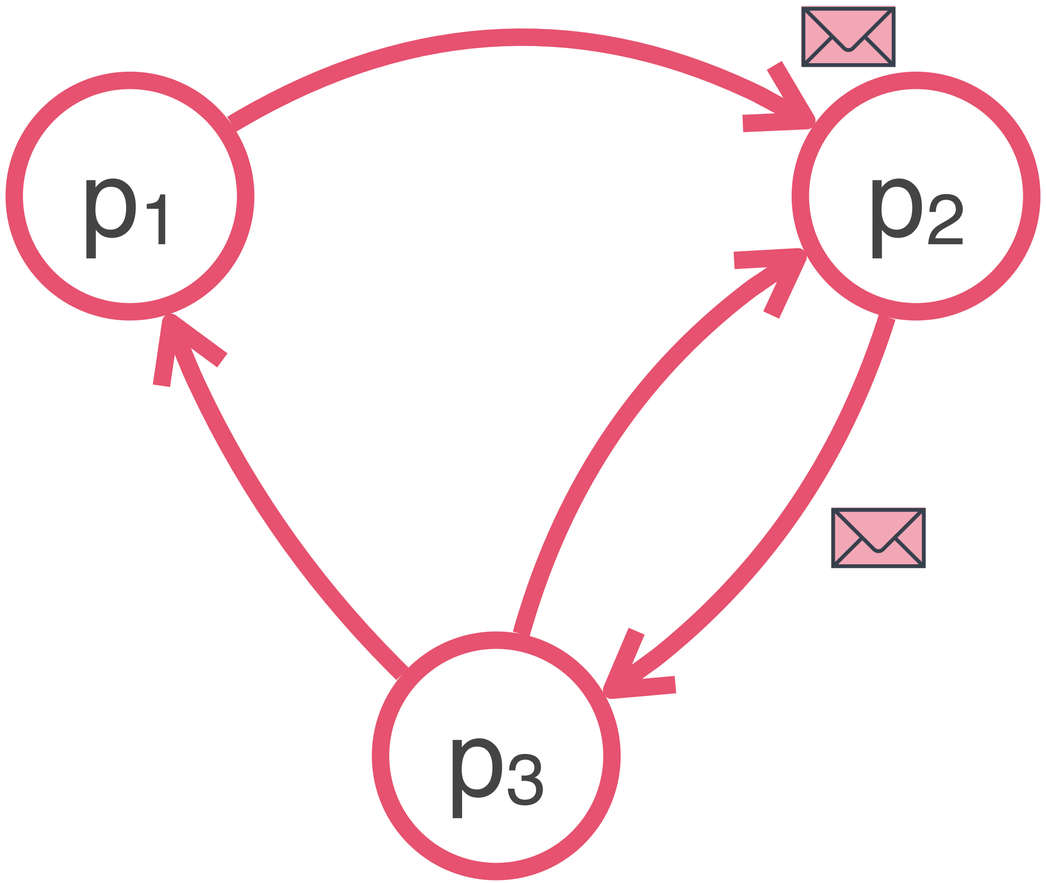
before marker

after marker

~IBM Streams
~Flink Unaligned Checkpoints (experimental)

Variant 2.2 - Unaligned Epoch Snapshots

- Assumptions:
  - FIFO Reliable Channels
  - Strongly Connected Graph
  - Single Initiating Process



Snapshot

s1, s2, s3

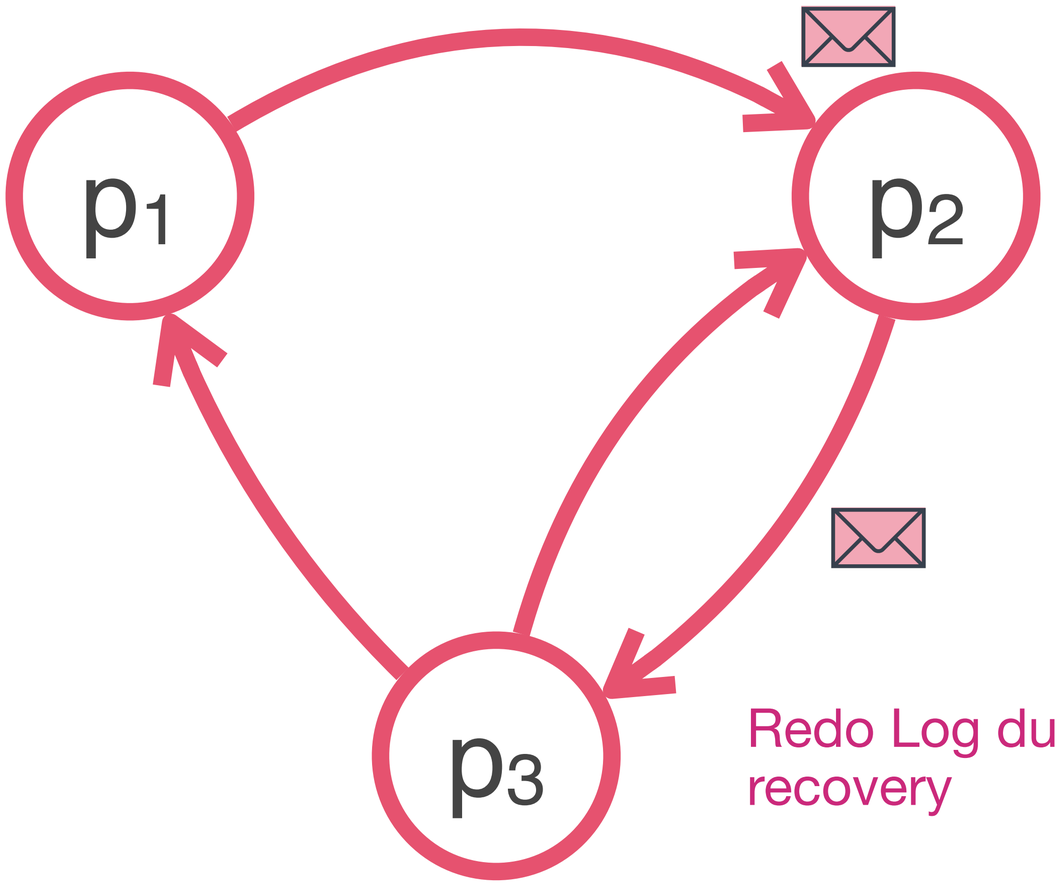
before marker

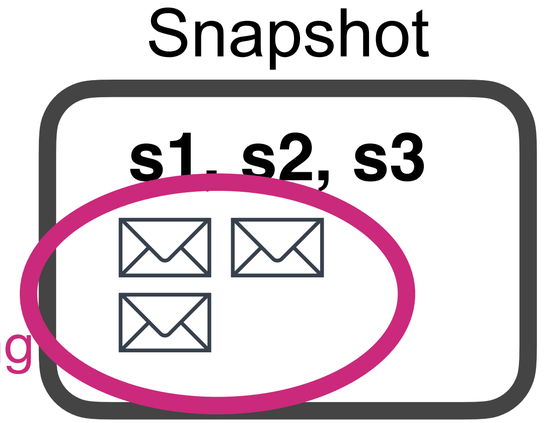
after marker

~IBM Streams
~Flink Unaligned Checkpoints (experimental)

Variant 2.2 - Unaligned Epoch Snapshots

- Assumptions:
  - FIFO Reliable Channels
  - Strongly Connected Graph
  - Single Initiating Process





Redo Log during recovery

before marker

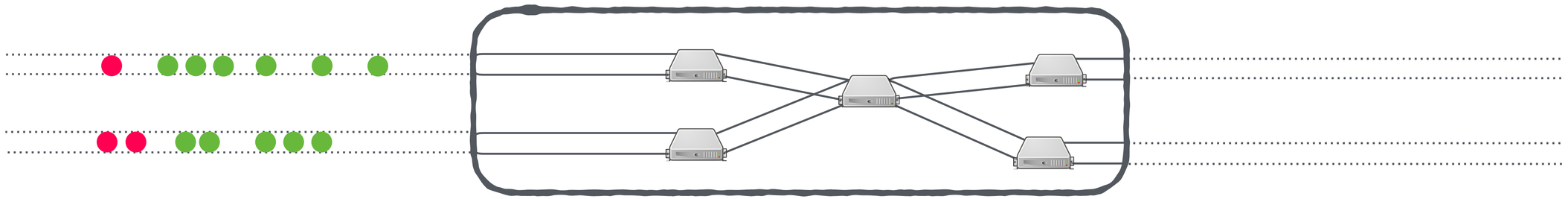
after marker

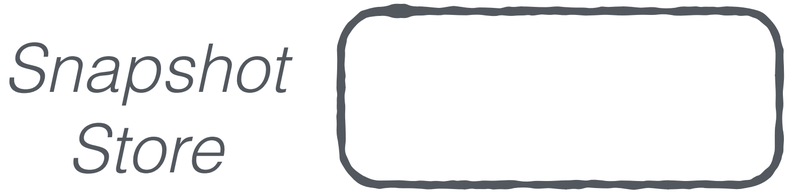
~Distributed Fuzzy Database Snapshotting

~IBM Streams
~Flink Unaligned Checkpoints (experimental)

Variant 2.3 - Aligned Epoch Snapshots

~Asynchronous without a redo log



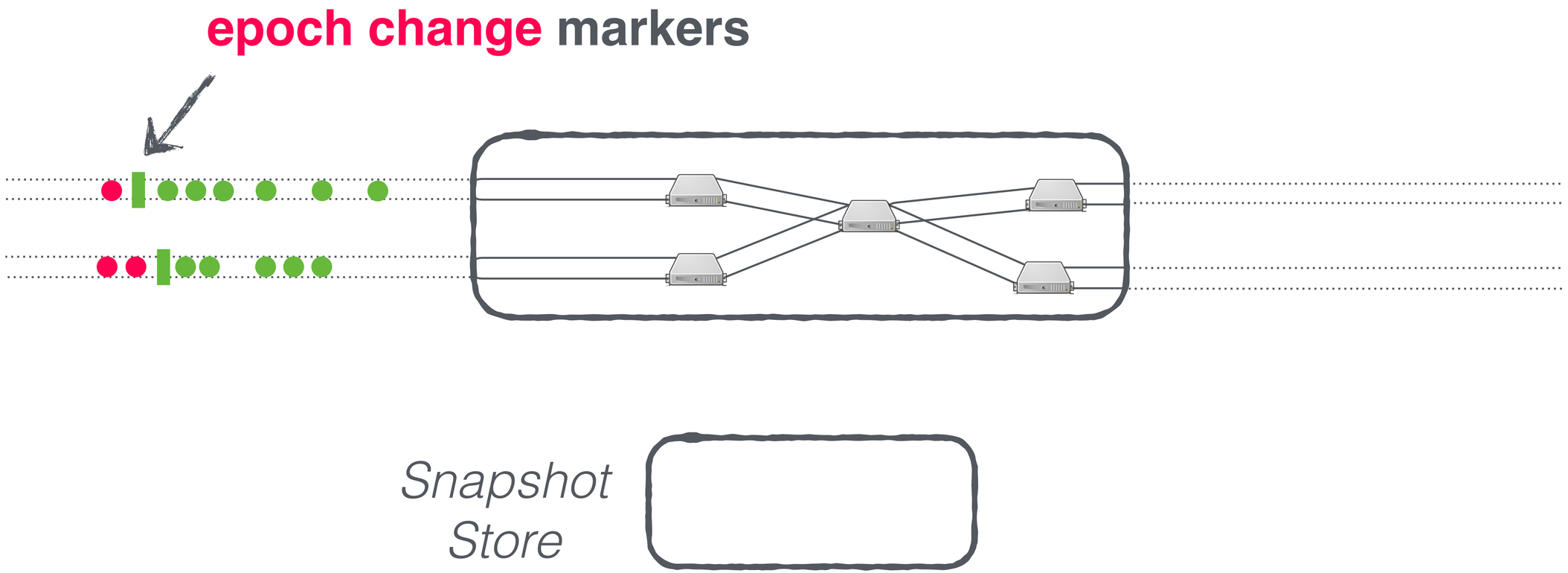


Flink Savepoints

State management in Apache Flink® : Consistent Stateful Distributed Stream Processing
Carbone P, Ewen S, Fóra G, et. al. in Proceedings of the VLDB Endowment (2017)

Variant 2.3 - Aligned Epoch Snapshots

~Asynchronous without a redo log

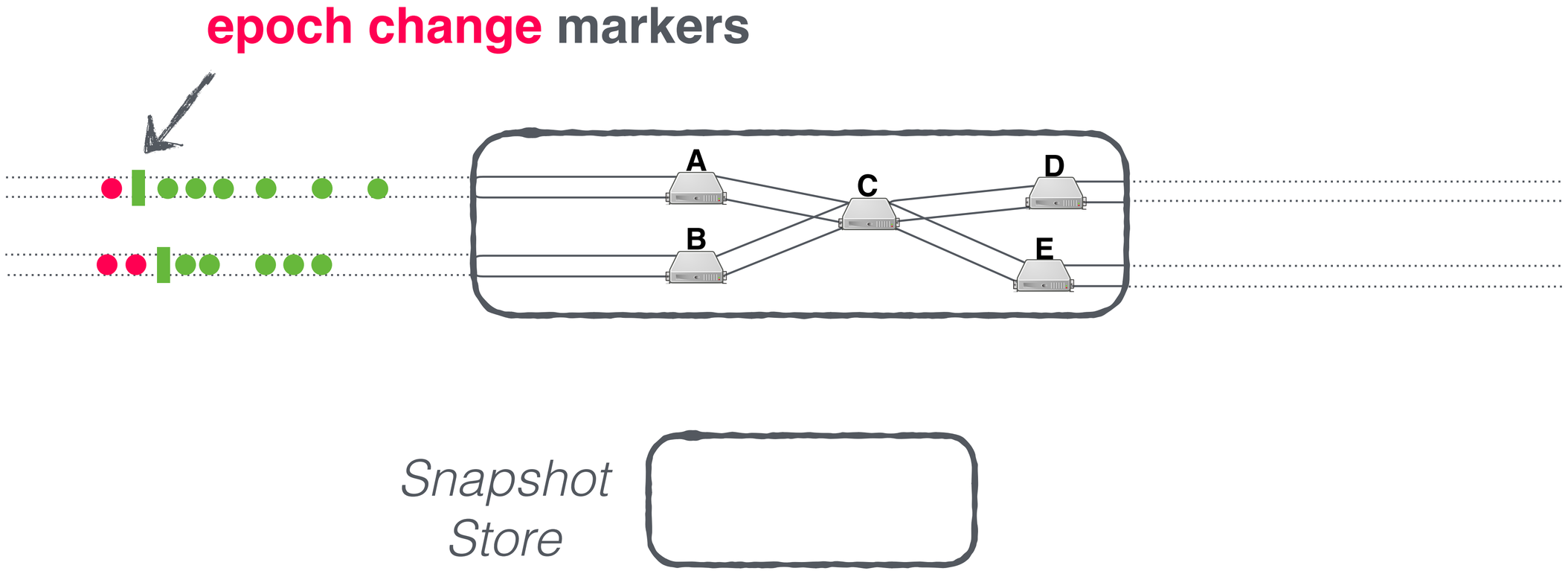


Flink Savepoints

State management in Apache Flink® : Consistent Stateful Distributed Stream Processing
Carbone P, Ewen S, Fóra G, et. al. in Proceedings of the VLDB Endowment (2017)

Variant 2.3 - Aligned Epoch Snapshots

~Asynchronous without a redo log

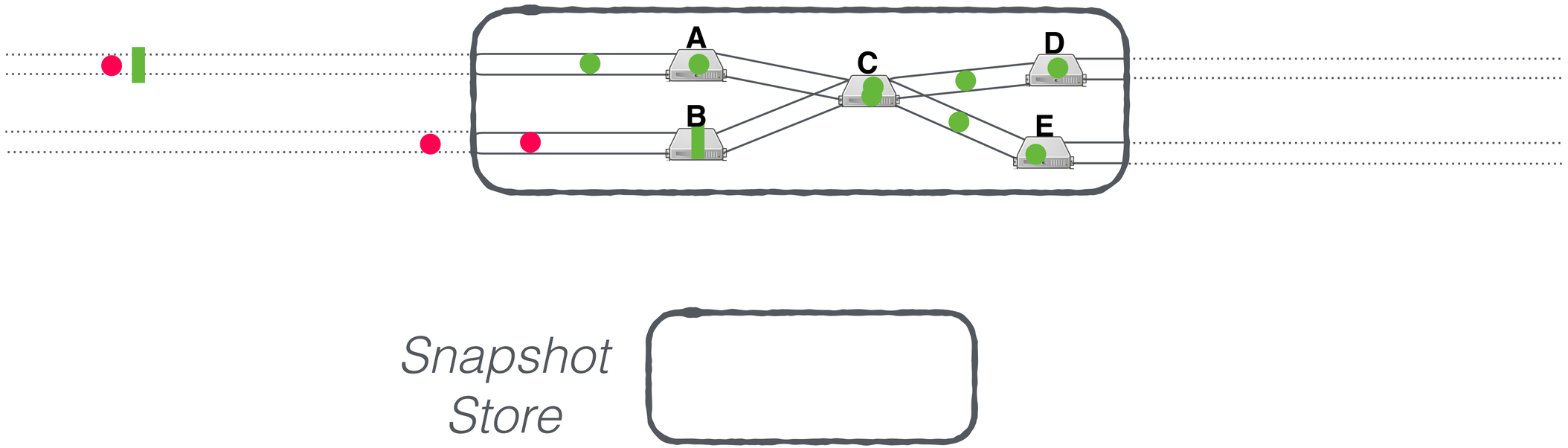


Flink Savepoints

State management in Apache Flink® : Consistent Stateful Distributed Stream Processing
Carbone P, Ewen S, Fóra G, et. al. in Proceedings of the VLDB Endowment (2017)

Variant 2.3 - Aligned Epoch Snapshots

~Asynchronous without a redo log

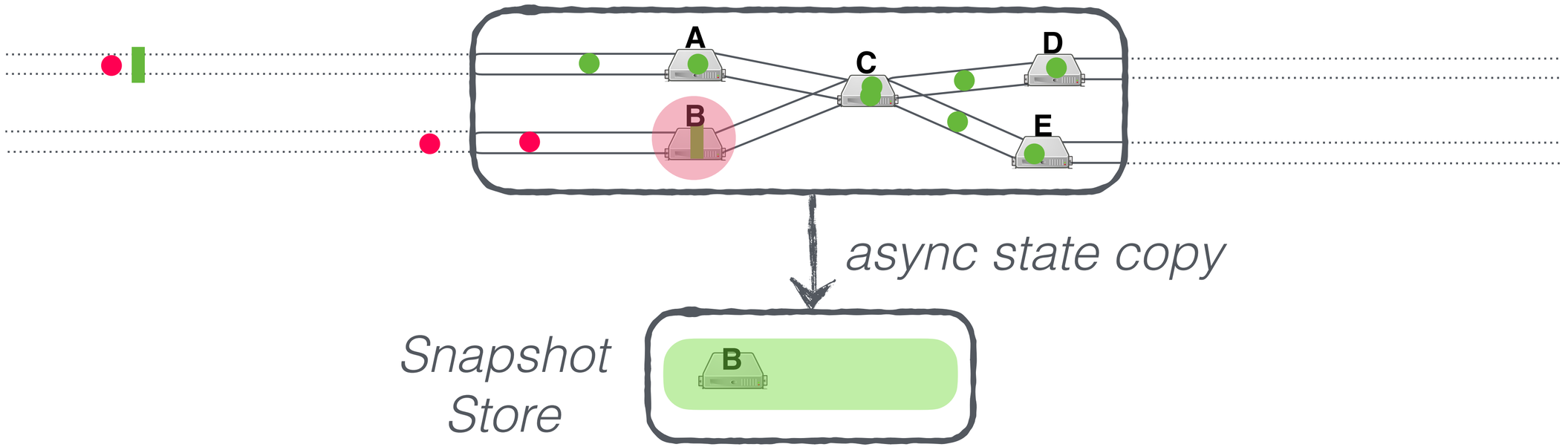


Flink Savepoints

State management in Apache Flink® : Consistent Stateful Distributed Stream Processing
Carbone P, Ewen S, Fóra G, et. al. in Proceedings of the VLDB Endowment (2017)

Variant 2.3 - Aligned Epoch Snapshots

~Asynchronous without a redo log

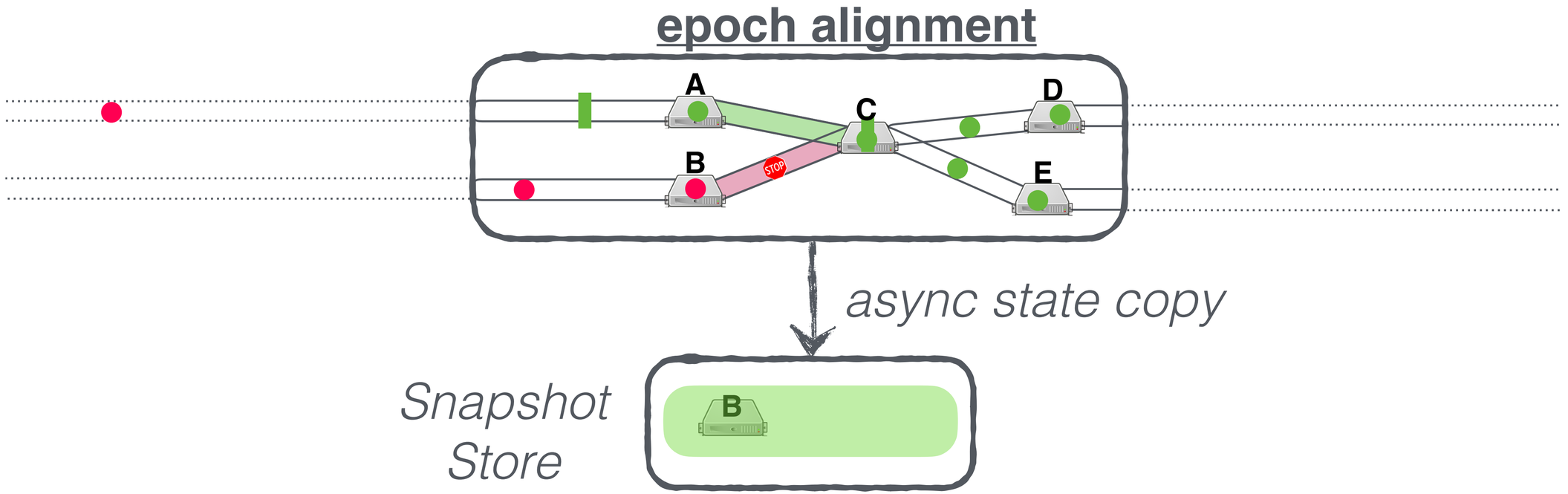


Flink Savepoints

State management in Apache Flink® : Consistent Stateful Distributed Stream Processing
Carbone P, Ewen S, Fóra G, et. al. in Proceedings of the VLDB Endowment (2017)

Variant 2.3 - Aligned Epoch Snapshots

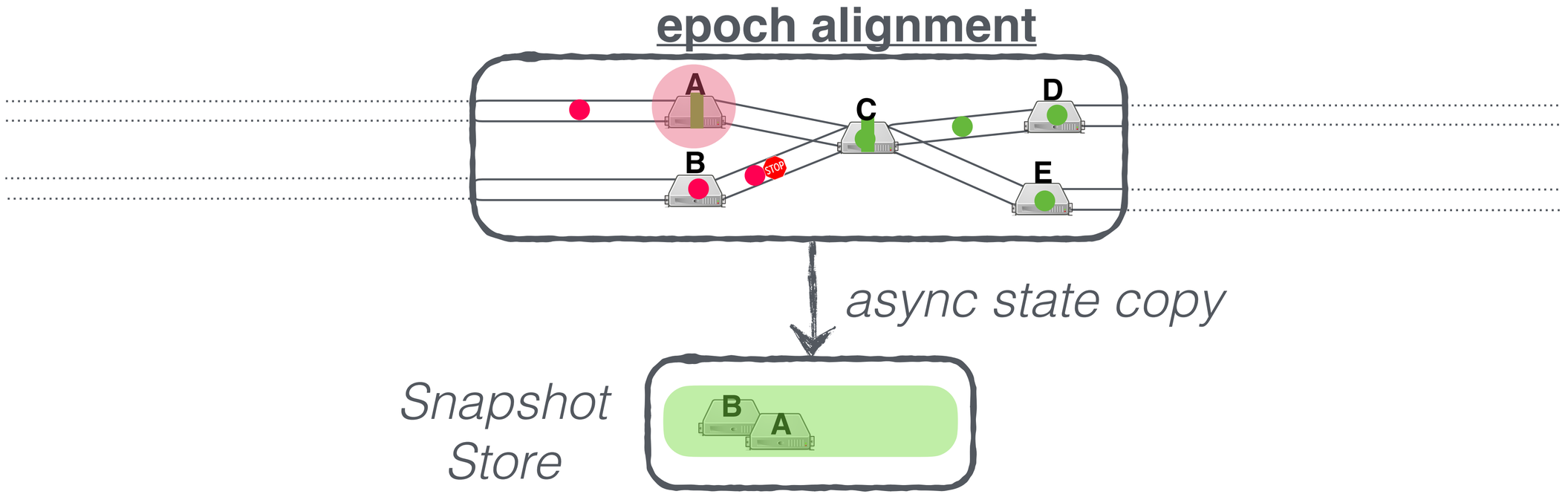
~Asynchronous without a redo log



State management in Apache Flink® : Consistent Stateful Distributed Stream Processing
Carbone P, Ewen S, Fóra G, et. al. in Proceedings of the VLDB Endowment (2017)

Variant 2.3 - Aligned Epoch Snapshots

~Asynchronous without a redo log

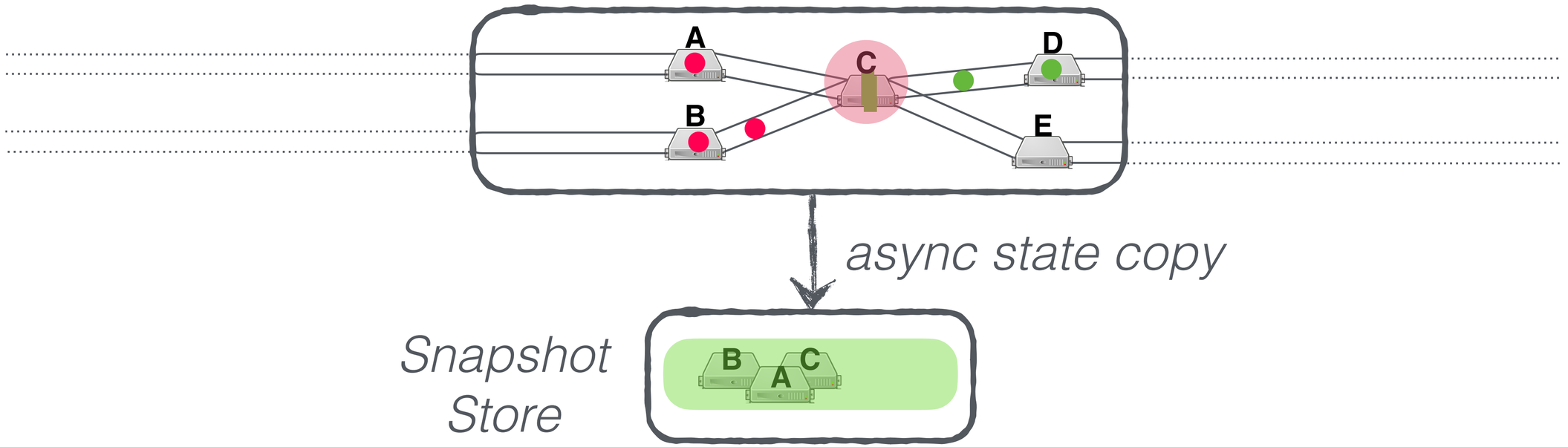


Flink Savepoints

State management in Apache Flink® : Consistent Stateful Distributed Stream Processing
Carbone P, Ewen S, Fóra G, et. al. in Proceedings of the VLDB Endowment (2017)

Variant 2.3 - Aligned Epoch Snapshots

~Asynchronous without a redo log

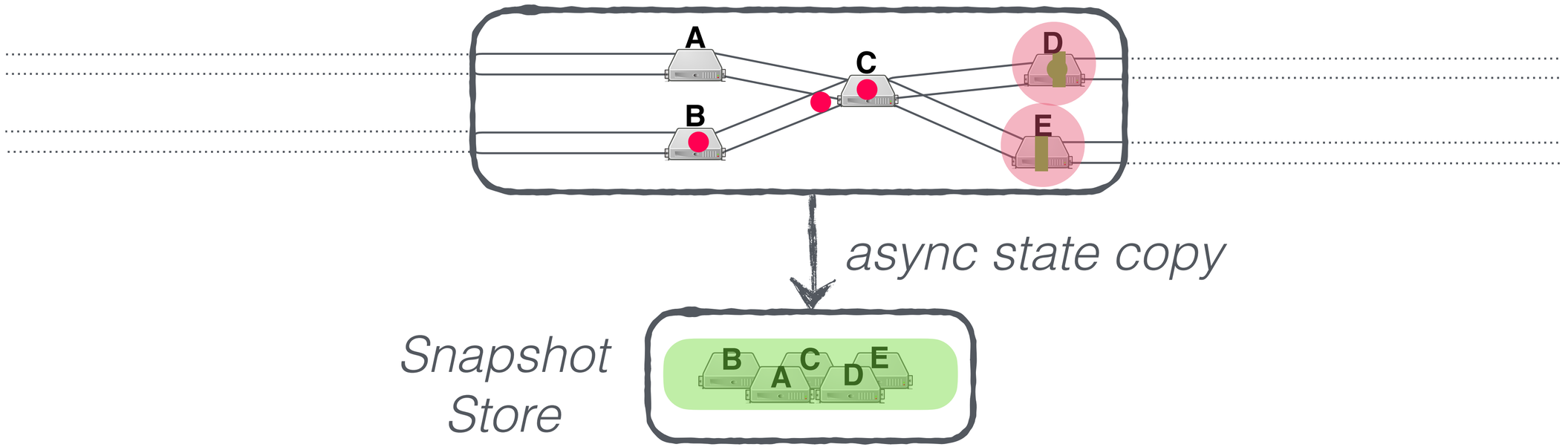


Flink Savepoints

State management in Apache Flink® : Consistent Stateful Distributed Stream Processing
Carbone P, Ewen S, Fóra G, et. al. in Proceedings of the VLDB Endowment (2017)

Variant 2.3 - Aligned Epoch Snapshots

~Asynchronous without a redo log

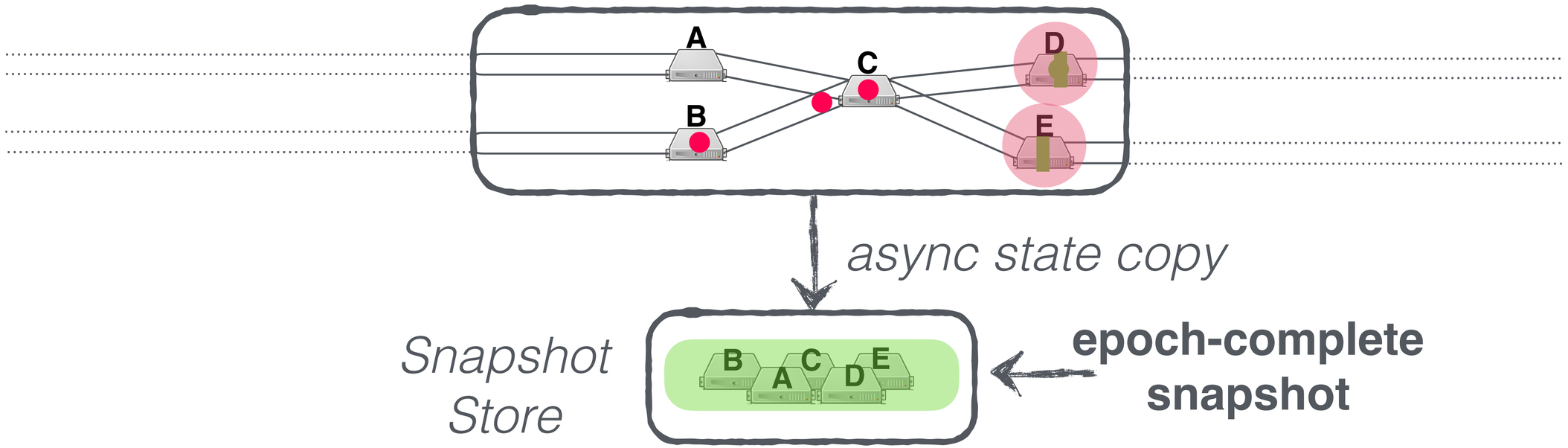


State management in Apache Flink® : Consistent Stateful Distributed Stream Processing
Carbone P, Ewen S, Fóra G, et. al. in Proceedings of the VLDB Endowment (2017)

Flink Savepoints

Variant 2.3 - Aligned Epoch Snapshots

~Asynchronous without a redo log



Flink Savepoints

State management in Apache Flink® : Consistent Stateful Distributed Stream Processing
Carbone P, Ewen S, Fóra G, et. al. in Proceedings of the VLDB Endowment (2017)

Summary

- **Unligned Snapshots**: lowest runtime overhead with reconfiguration costs (redo log)
- **Aligned Snapshots**: enforce flow control, yet offer:
  - **completeness** (snapshots reflect a complete epoch transaction)
  - **fast reconfiguration** (no redo log, no message shuffling and replaying)

- **marker based** consistent hot reconfiguration also feasible

Summ

- Unligne
- Aligned
  - comp
  - fast re
  - mark

(redo log)

Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems

Luo Mai[1], Kai Zeng[2], Rahul Potharaju[2], Le Xu[3], Steve Suh[2], Shivaram Venkataraman[2], Paolo Costa[1,2], Terry Kim[2], Saravanan Muthukrishnan[2], Vamsi Kuppa[2], Sudheer Dhulipalla[2], Sriram Rao[2]

[1]Imperial College London, [2]Microsoft, [3]UIUC

[1]luo.mai11@imperial.ac.uk, [2]{kaizeng, rapoth, stsuh, shivaram.venkataraman, pcosta, terryk, sarmut, vamsik, sudheerd, sriramra}@microsoft.com, [3]lexu1@illinois.edu

ABSTRACT

Stream-processing workloads and modern shared cluster environments exhibit high variability and unpredictability. Combined with the large parameter space and the diverse set of user SLOs, this makes modern streaming systems very challenging to statically configure and tune. To address these issues, in this paper we investigate a novel control-plane design, Chi, which supports continuous monitoring and feedback, and enables dynamic re-configuration. Chi leverages the key insight of embedding control-plane messages in the data-plane channels to achieve a low-latency and flexible control plane for stream-processing systems.

Chi introduces a new reactive programming model and design mechanisms to asynchronously execute control policies, thus avoiding global synchronization. We show how this allows us to easily implement a wide spectrum of control policies targeting different use cases observed in production. Large-scale experiments using production workloads from a popular cloud provider demonstrate the flexibility and efficiency of our approach.

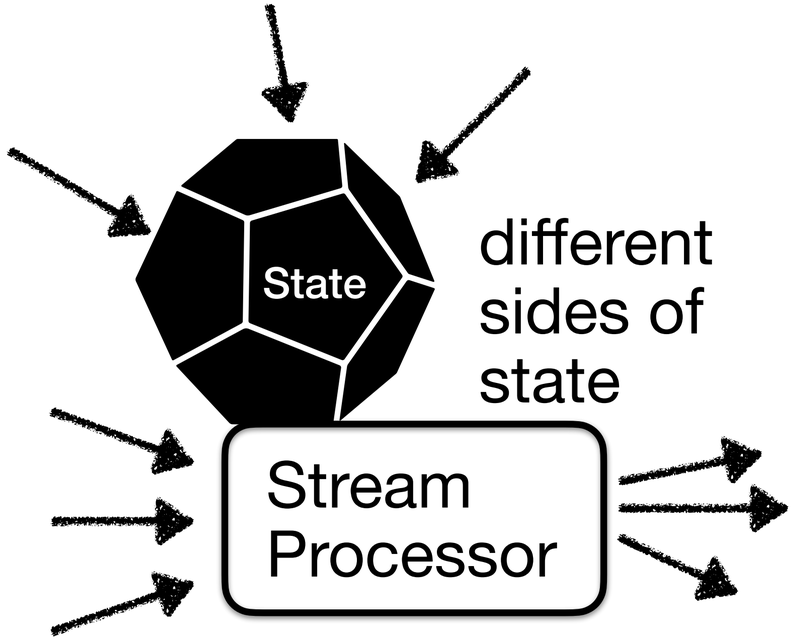
PVLDB Reference Format:
Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, Sriram Rao. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *PVLDB*, 11 (10): 1303-1316, 2018.
DOI: https://doi.org/10.14778/3231751.3231765

Fully achieving the benefits promised by these online systems, however, is particularly challenging. First, streaming workloads exhibit high *temporal* and *spatial* variability, up to an order of magnitude compared to the average load [27, 31]. Second, large shared clusters exhibit high hardware heterogeneity and unpredictable concurrent usage. Third, modern streaming systems expose a large parameter space, which makes tuning hard, even for the most experienced engineers. Further, different users and jobs have a diverse set of Service Level Objectives (SLOs), leading to divergent configuration settings. Addressing these issues requires introducing continuous monitoring and feedback as well as dynamic re-configuration into all aspects of streaming systems, ranging from query planning and resource allocation/scheduling to parameter tuning. We call the system layer in charge of such control mechanisms the *control plane*, to distinguish it from the *data plane* (the layer in charge of data processing).

Through our interaction with product teams and cloud operators, we identified the following requirements that a control plane should satisfy. First, it should be possible to define new custom control operations, tailored to different scenarios [24]. Second, this should be achieved with only minimal effort by the developers and through a simple and intuitive API to minimize the chance of bugs, which are particularly challenging to detect in a distributed environment. Finally, the control overhead should be kept to a minimum, even in the presence of high event throughput and large computation graphs. This is particularly critical in today's landscape with different cloud providers competing hard to offer the best SLOs to their

[Prospects] What is left to solve?

- Standardization of Snapshots (cross-system interoperability)
- State is still a black box outside the dataflow graph
  - Can it facilitate analytics/services?
  - db materialized views
  - live ML features
- Cross-partitioned shared state access?
  - Snapshot Isolation on Live State



Beyond Analytics

The Evolution of Stream Processing Systems

State Management

Paris Carbone, Marios Fragkoulis, Vasia Kalavri, Asterios Katsifodimos

RI.SE BOSTON UNIVERSITY TU Delft