Data Stream Computing with Apache Flink

Paris Carbone <paris.carbone@ri.se>
Lead Researcher @ **RISE**
Committer @ **Apache Flink**

Unbounded Analytics Stack

High Level Models	Stream SQL, CEP…
Compute	Flink, Beam, Kafka-Streams, Apex, Storm, Spark Streaming…
Storage	Kafka, Pub/Sub, Kinesis, Pravega…

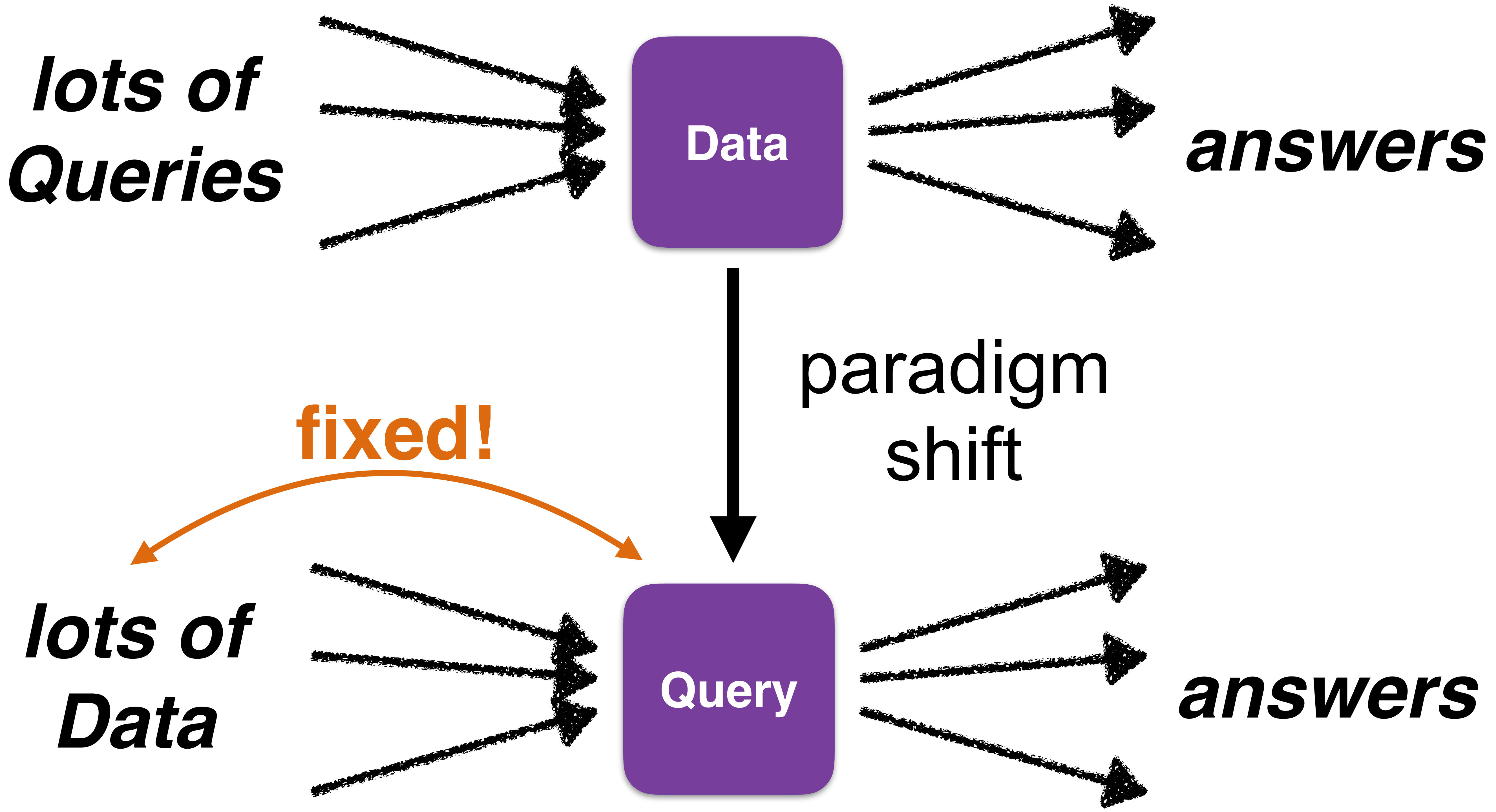
Unbounded Analytics Stack

High Level Models	Stream SQL, CEP…
Compute	Flink, Beam, Kafka-Streams, Apex, Storm, Spark Streaming…
Storage	Kafka, Pub/Sub, Kinesis, Pravega…

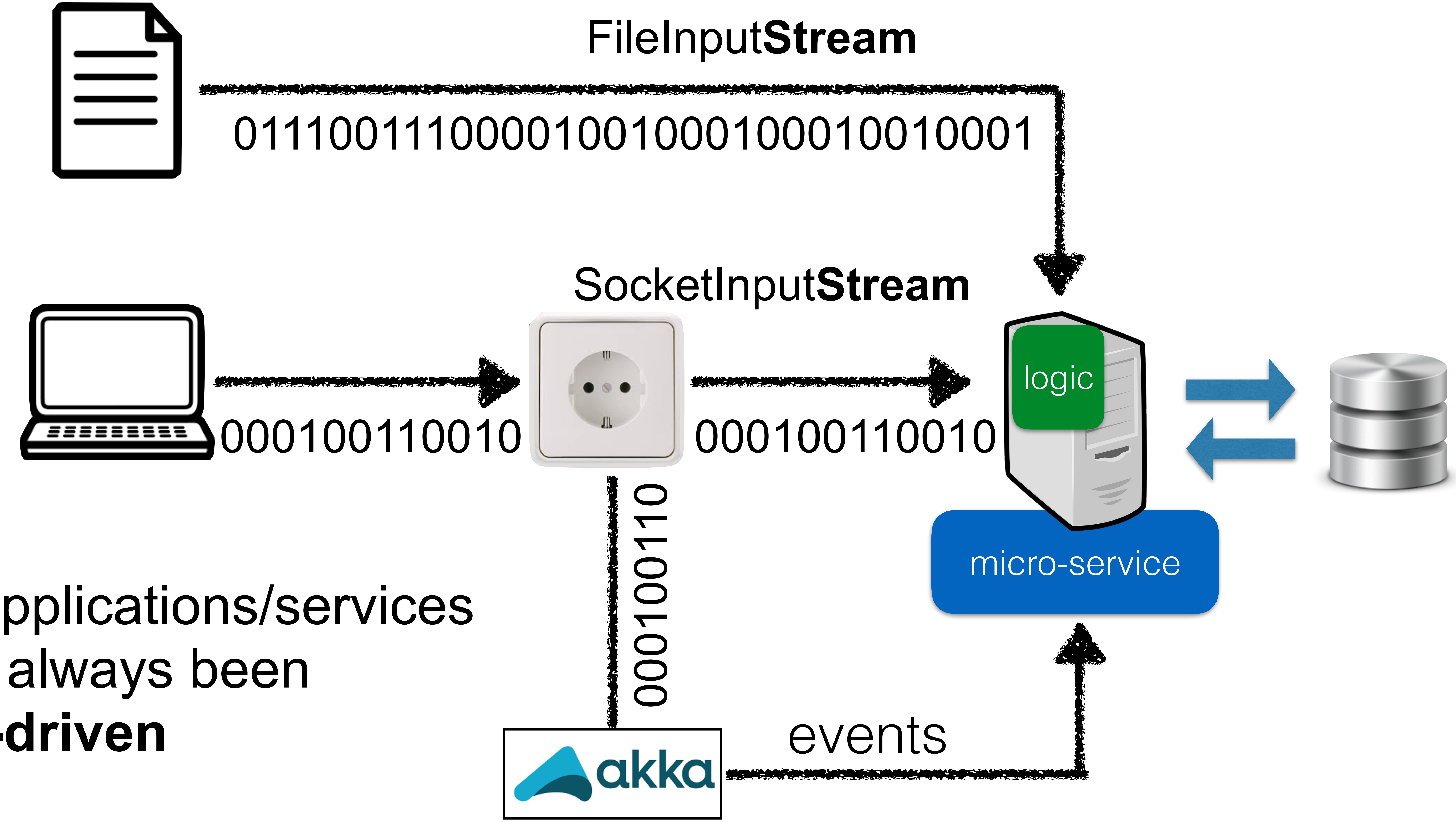
Overview

A Paradigm Shift

- **Data Stream Processing** as a standing query execution paradigm



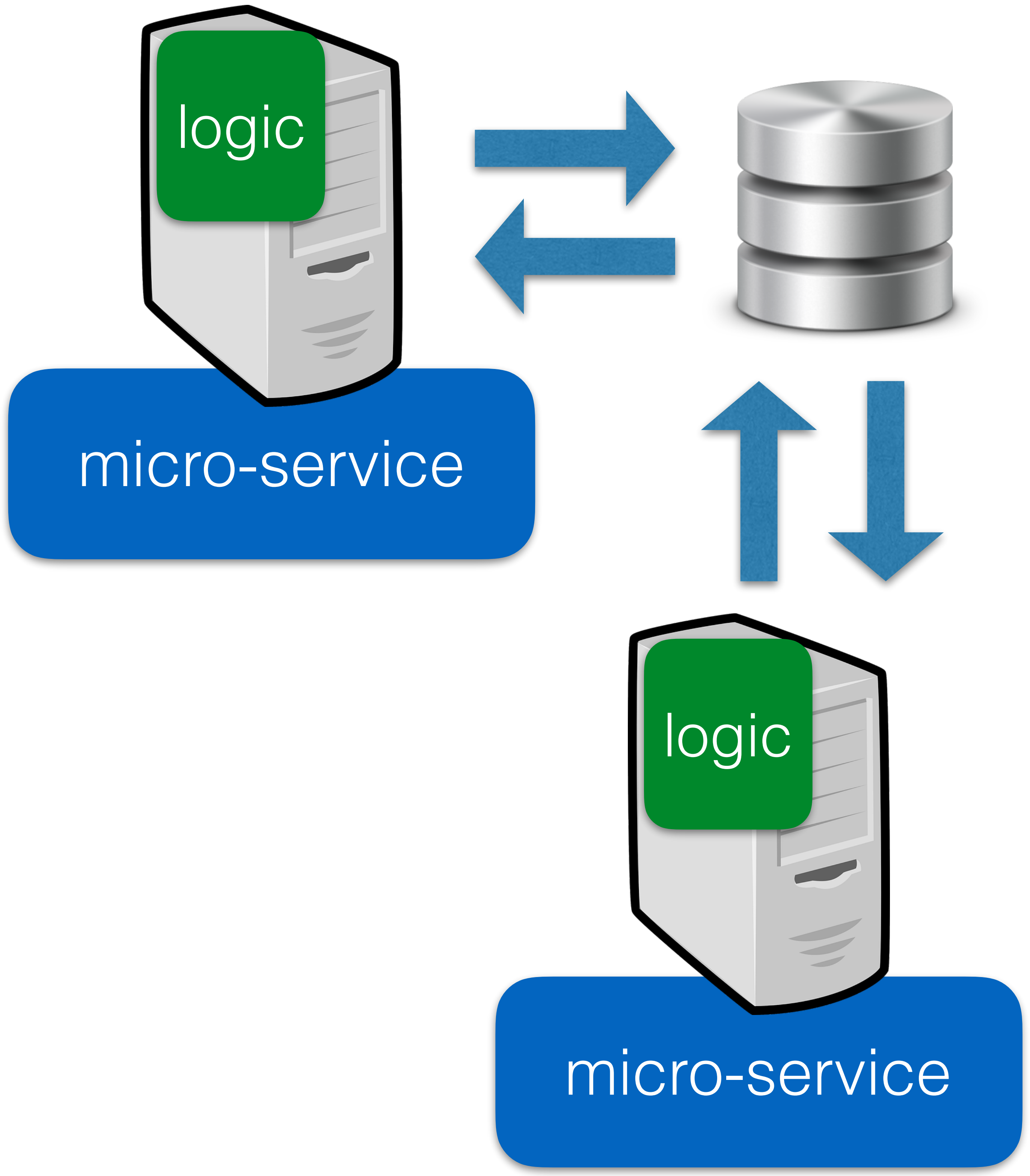
Similar Technologies



live applications/services have always been **data-driven**

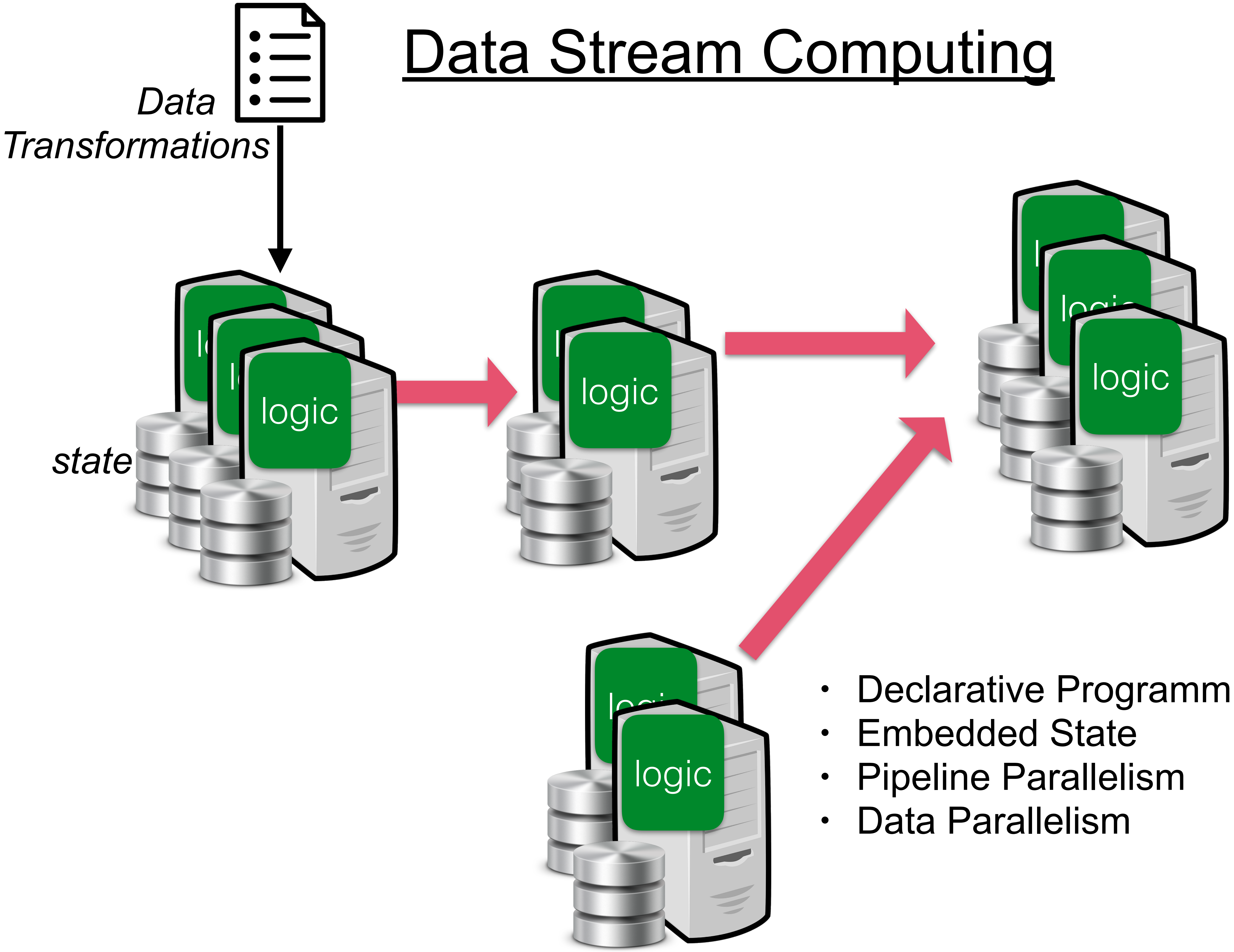
What Streams do Better

Traditional Event Processing



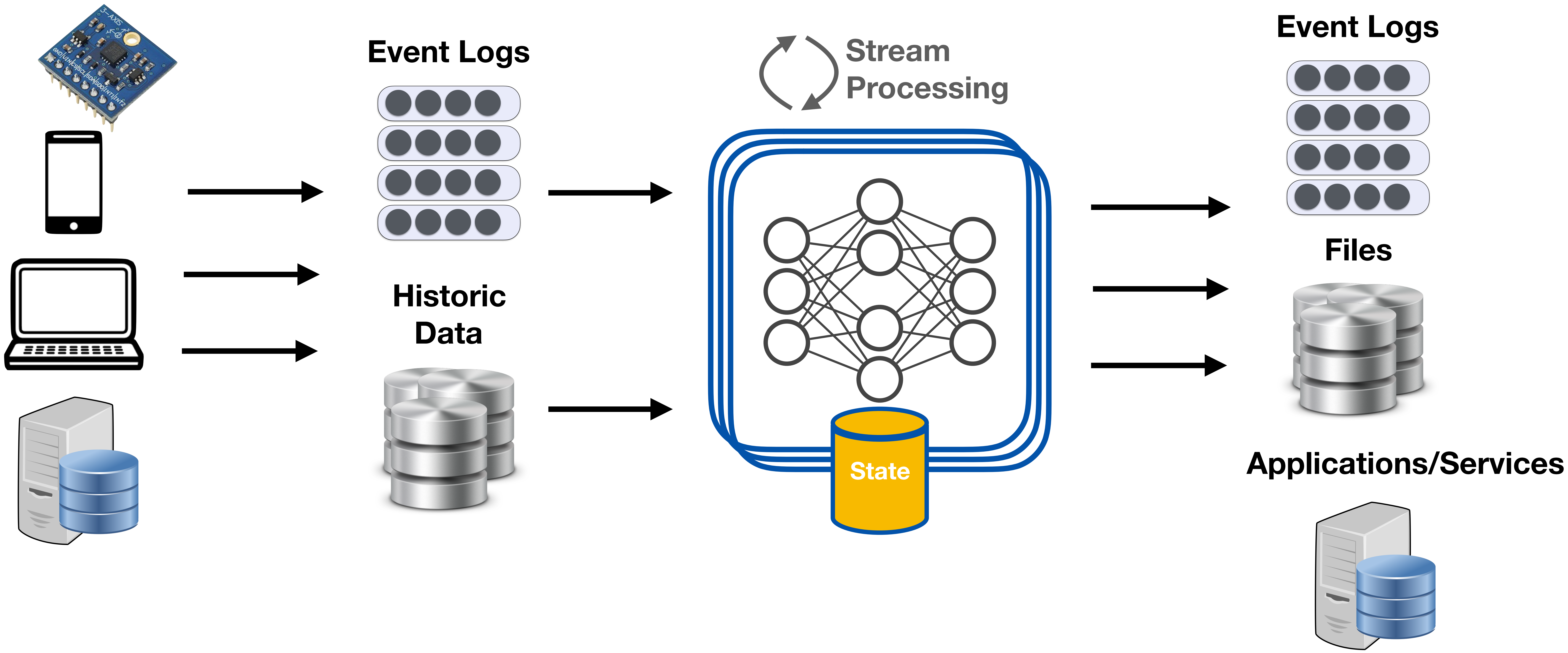
vs

Data Stream Computing



- Declarative Programming
- Embedded State
- Pipeline Parallelism
- Data Parallelism

The End-To-End Picture



Why Flink

Data Streams, Fault Tolerance, Window Aggregation, Iterations

- Top-level Apache Project
- #1 stream processor (2019)
- Production-Proof

- > 400 contributors
- 100s of deployments

state management, windows, sql

influenced

production deployments

Declarative Data Streaming

```
val windowCounts = text.flatMap { w => w.split("\\s") }
  .map { w => WordWithCount(w, 1) }
  .keyBy("word")
  .timeWindow(Time.seconds(5))
  .sum("count")
```

Window Word Count (Apache Flink)

Flink

Building Blocks of Flink

CEP

```
Pattern.begin("start").where(_.getName().equals("c"))
  .followedBy("middle").where(_.getName().equals("a"))
                       .oneOrMore().consecutive()
  .followedBy("end1").where(_.getName().equals("b"));
```

Stream SQL

```
|SELECT
|  user,
|  SESSION_START(rowtime, INTERVAL '12' HOUR) AS sStart,
|  SESSION_END(rowtime, INTERVAL '12' HOUR) AS sEnd,
|  SUM(amount)
| FROM Orders
| GROUP BY SESSION(rowtime(), INTERVAL '12' HOUR), user
```

Domain-Specific APIs

uses

DataStream API → window, flatmap, filter etc.

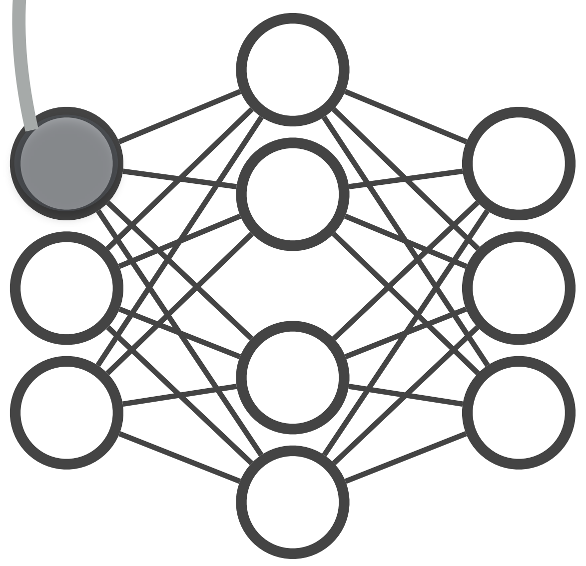
composes

Event Processing Model → f(event, state, time)

Out-of-order Processing

executes

Dataflow Task Execution



- Task Schedulling/IO/Monitoring etc
- Fault Tolerance
- Reconfiguration
- Savepoints

State Management

Part I

Stream Programming in Apache Flink

Technologies Behind Flink

- Flink runs on the **JVM.**
- **Master/Slave** architecture ~ Hadoop **(JobManager, TaskManagers)**
- **Java** and **Scala** 100% supported.
- **Depends** on: *Zookeeper, Akka, RocksDB (state).*
- **Supports**: *Kafka, Cassandra, Kinesis, Elasticsearch, HDFS, RabbitMQ, NiFi, Google Cloud PubSub, Twitter API* etc.

- Two Underlying Execution Modes:
  - **DataSet**: Batch programs (to be deprecated)
  - **DataStream**: Unbounded programs (and batch soon)

Types

- Automatic Support (Flink Serializer) for:
  - **Basic Types** (String, Long , Integer etc.)
  - **Composite Types:** Flink Tuples, POJOs / Scala Case Classes

```java
Tuple2<String, Integer> person = new Tuple2<>("Fred", 35);
// zero based index!
String name = person.f0;
Integer age = person.f1;
```

flink tuple

```scala
case class Person(name:String, age: Int)
Person("Fred Flintstone", 35)
```

case class

```java
public class Person {
    public String name;
    public Integer age;
    public Person() {};
    public Person(String name, Integer age) {
        …
    };
}

Person person = new Person("Fred Flintstone", 35);
```

pojo

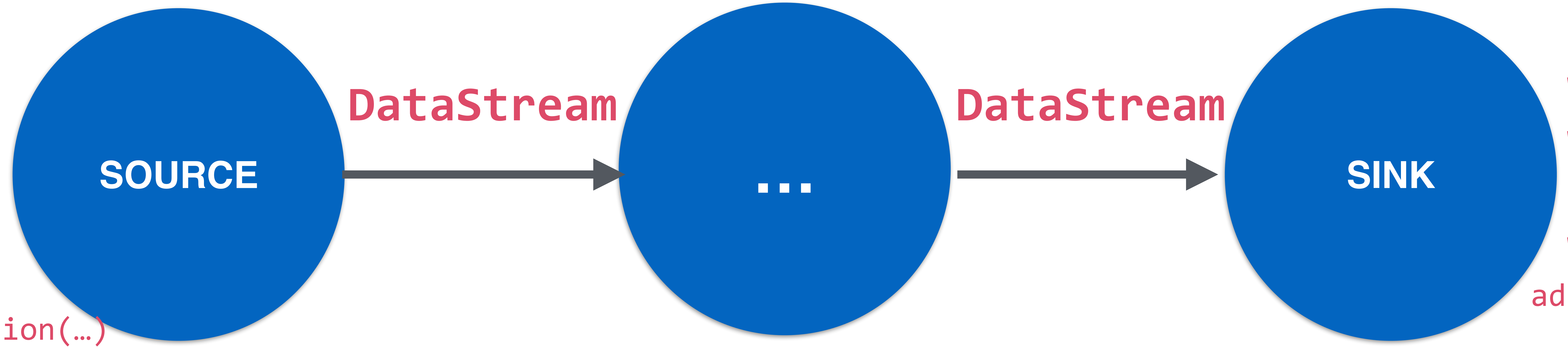
https://training.ververica.com/intro/intro-2.html

Program Composition

- A Flink Program has a *beginning* (**Source**) and an *end* (**Sink**).
- Programs are **lazily** executed (compiled, optimised and executed all-together)

`environment.`

`readTextFile(…)`
`readFile(…)`
`socketTextStream(…)`
`fromElements(…)`
`fromCollection(…)`
`fromParallelCollection(…)`
`addSource(connector) - Kafka etc.`



`writeAsText(…)`
`writeAsCSV(…)`
`print(…)`
`writeToSocket(…)`
`addSink(connector)`

`environment.execute();`

Example

```java
public class Example {

  public static void main(String[] args) throws Exception {
    final StreamExecutionEnvironment env =
      StreamExecutionEnvironment.getExecutionEnvironment();

    DataStream<Person> flintstones = env.fromElements(
        new Person("Fred", 35),
        new Person("Wilma", 35),
        new Person("Pebbles", 2));

    DataStream<Person> adults = flintstones
      .filter(new FilterFunction<Person>() {
      @Override
      public boolean filter(Person person){
        return person.age >= 18;
      }
    });

    adults.print();

    env.execute();
  }
```

```java
  public static class Person {
    public String name;
    public Integer age;
    public Person() {};

    public Person(String name, Integer age) {
      this.name = name;
      this.age = age;
    };

    public String toString() {
      return this.name.toString() + ": age "
      + this.age.toString();
    };
  }
}
```


https://training.ververica.com/intro/intro-3.html

DataStream CheatSheet

DataStream

Map
```
dataStream.map { x => x * 2 }
```

FlatMap
```
dataStream.flatMap {
str => str.split(" ") }
```

Filter
```
dataStream.filter { _ != 0 }
```

KeyBy
```
dataStream.keyBy("someKey")
dataStream.keyBy(0)
```

Union
```
dataStream.union(stream1, ...)
```

Connect
```
someStream : DataStream[Int] = ...
otherStream : DataStream[String] = ...

someStream.connect(otherStream)
```

Split
```
val split = someDataStream.split(
  (num: Int) =>
    (num % 2) match {
      case 0 => List("even")
      case 1 => List("odd")
    } )
```

KeyedStream

Reduce
```
keyedStream.reduce { _ + _ }
```

Fold
```
keyedStream.fold("start")((str, i)
     => { str + "-" + i })
```

Aggregations
```
keyedStream.sum(0)
```

Window
```
<goto next slide>
```

ConnectedStream

CoMap, CoFlatMap
```
connectedStreams.map(
    (_ : Int) => true,
    (_ : String) => false
)
connectedStreams.flatMap(
    (_ : Int) => true,
    (_ : String) => false
)
```

SplitStream

Select
```
val even = split select "even"
val odd = split select "odd"
val all = split.select("even","odd")
```


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

Stream Windows

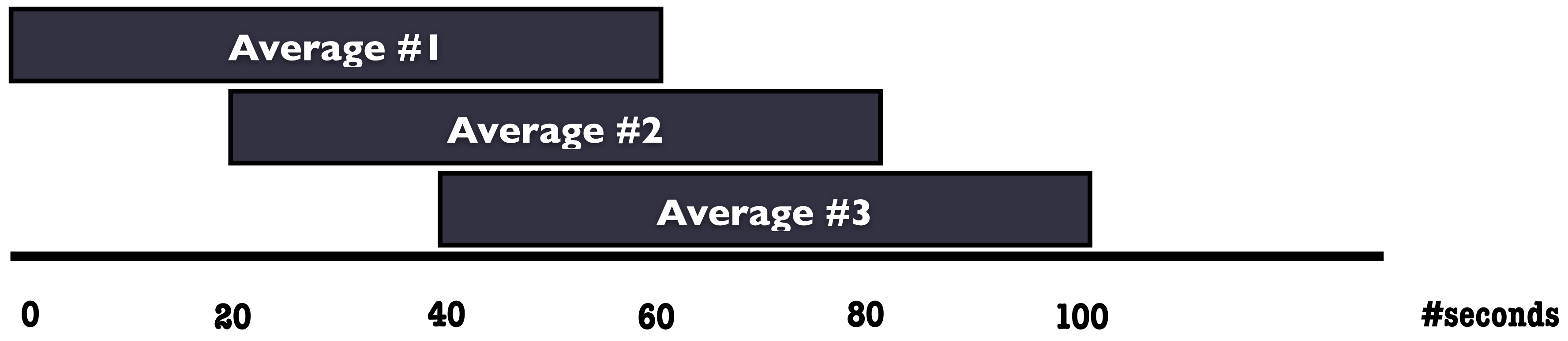
- We often need to do **analytics/aggregations** on relevant sets of records (e.g. a user session).

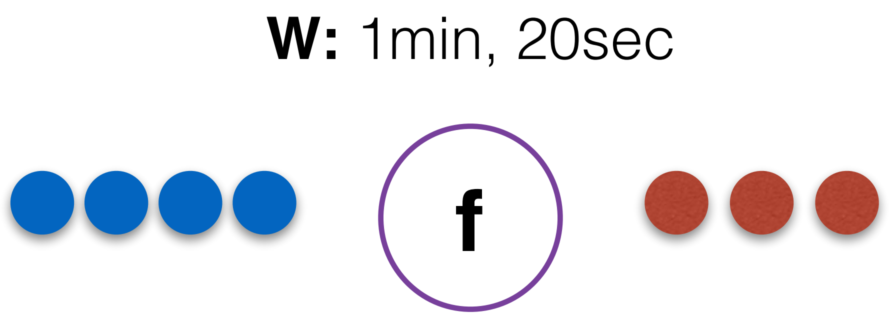
- A stream window is a **relevant slice** in the space-time continuum

  "*location temperature over the last minute every 20 sec*"

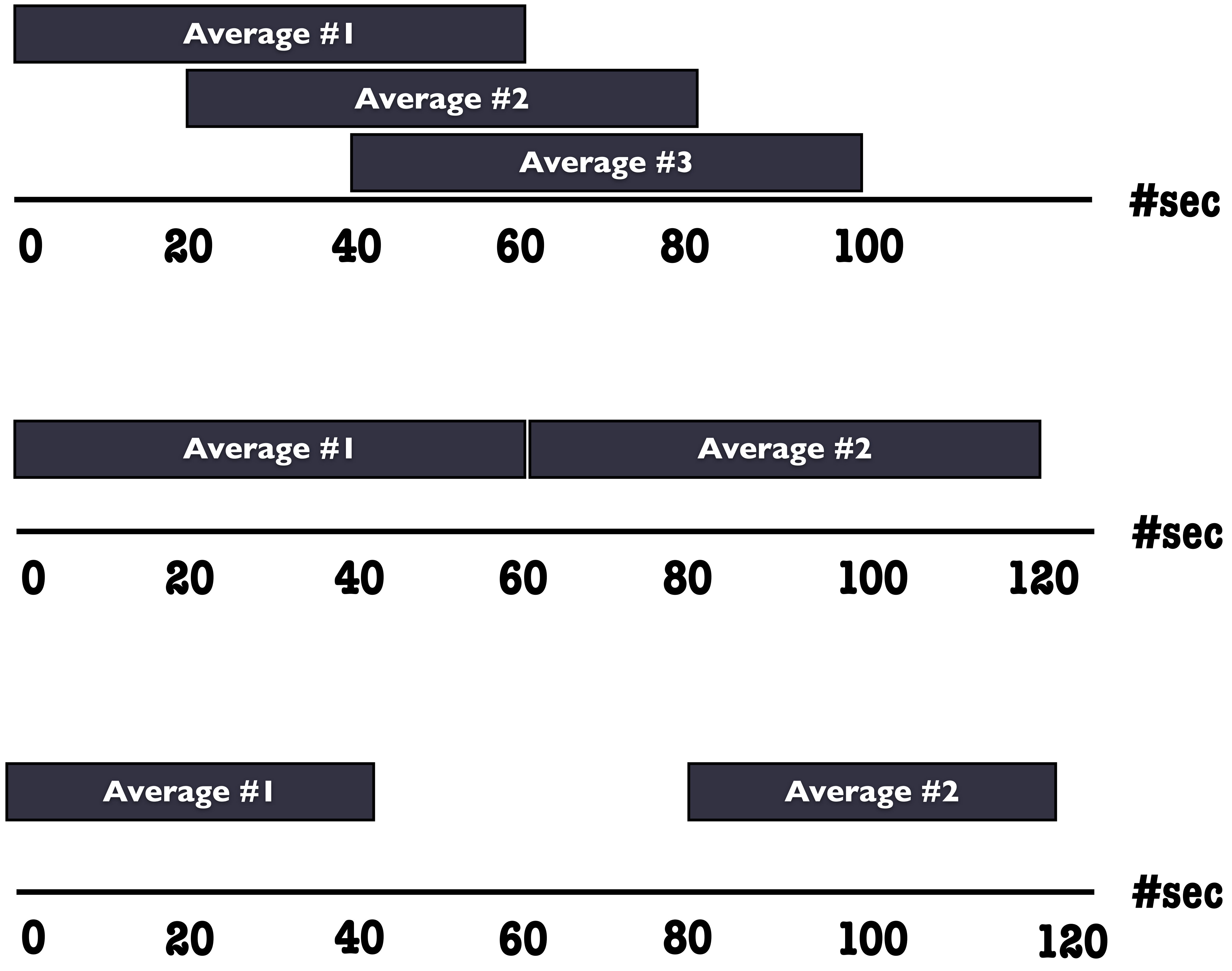
- **Range**: How **big** a window is (eg. 1 minute, 1000 tuples)

- **Trigger/Slide**: How **often** we need analysis on a window





Stream Window Types



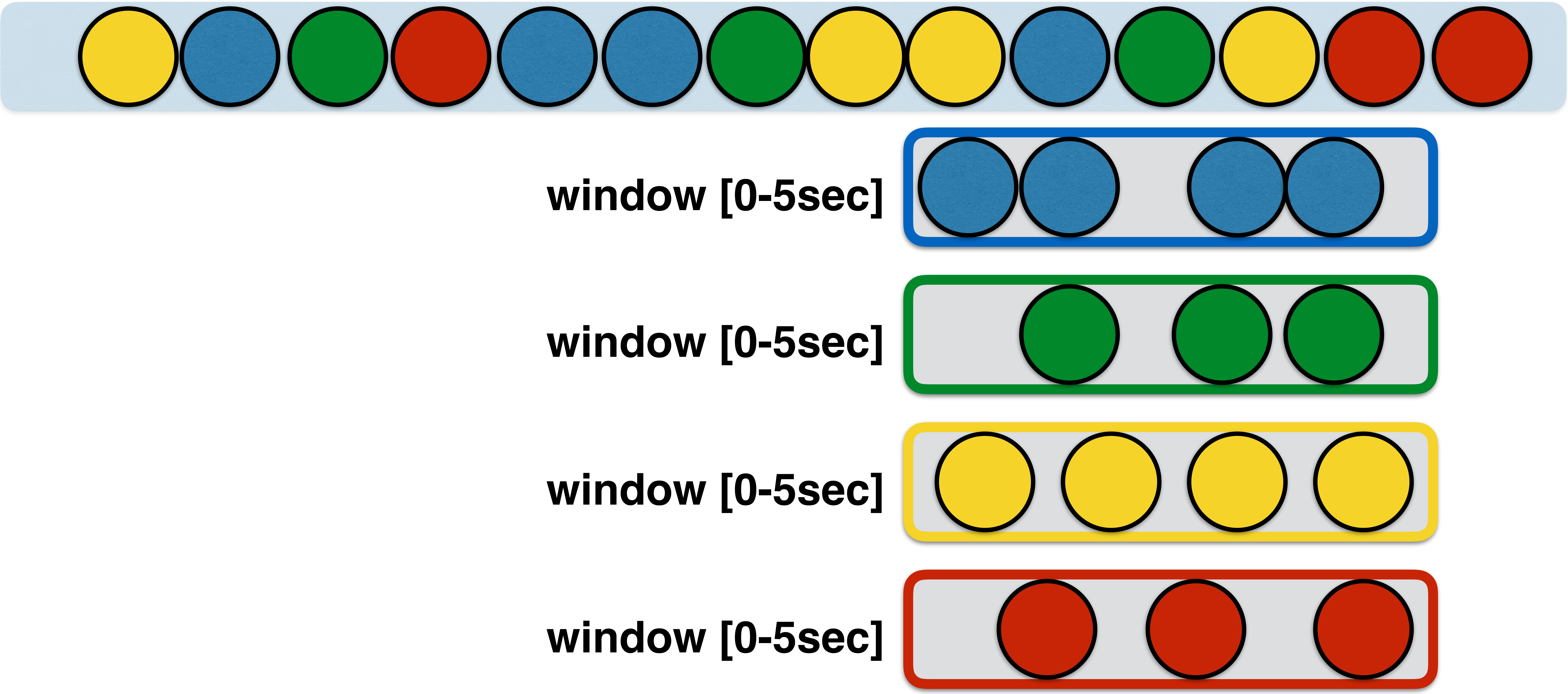
Sliding *range > slide*

Tumbling *range = slide*

Jumping *range < slide*

Data Parallelism and Windows

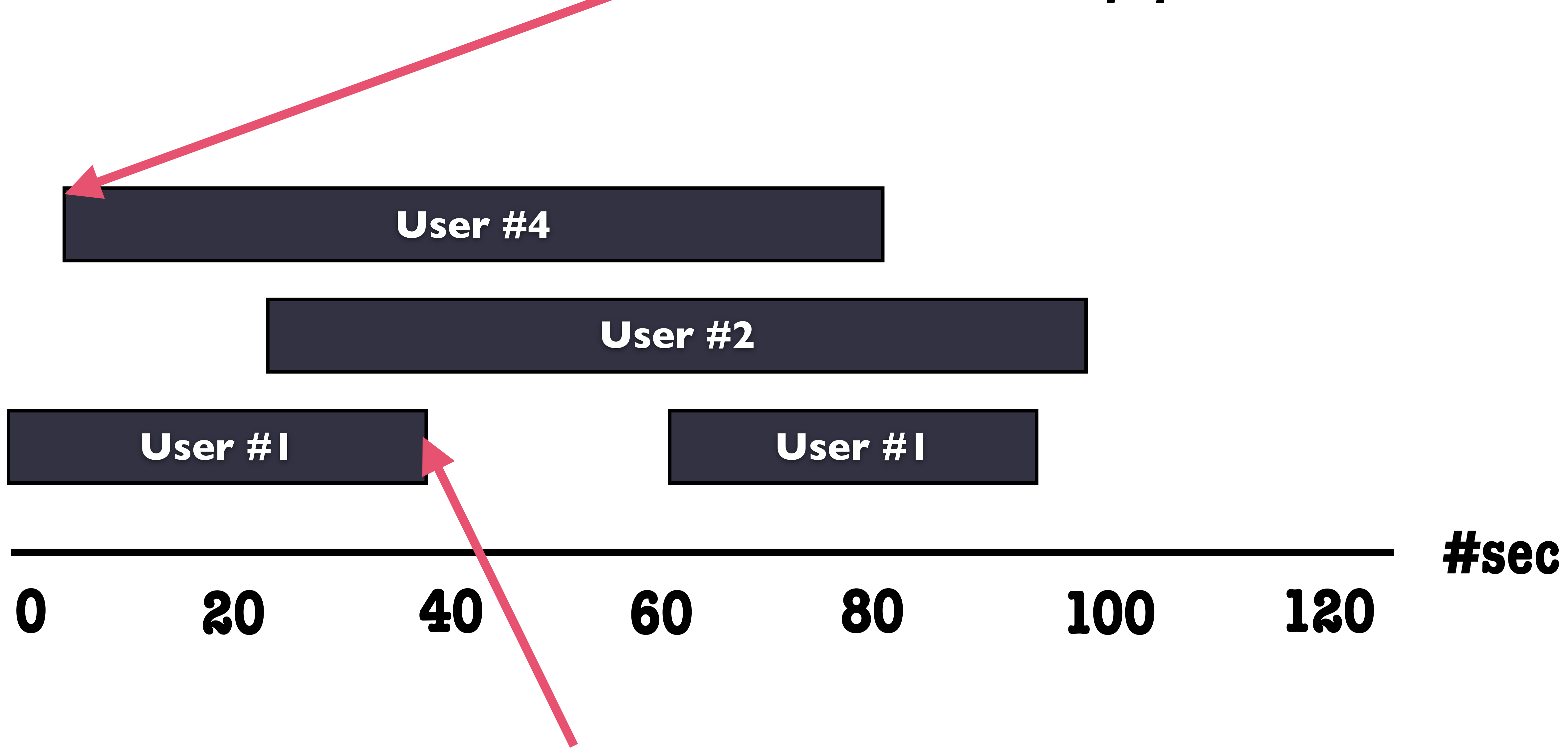
***Remember** Windows are defined on a KeyedStream*



*Note: Flink also supports a non-keyed **windowAll** with the cost of a **single task execution***

Session Windows

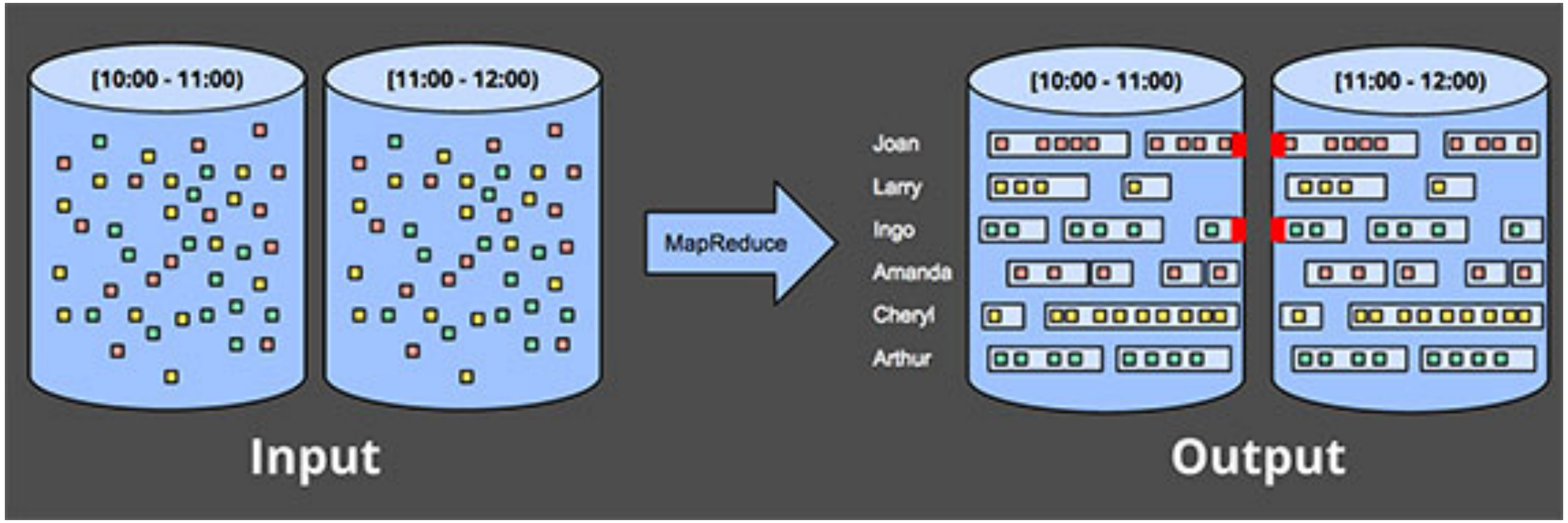
first event of user #4 happens here



user#1 becomes inactive (session times-out)

Session Windows

- Hard problem for Batch Processing engines
- Only suitable for a Continuous Execution



CheatSheet Continued

KeyedStream

Reduce
```
keyedStream.reduce { _ + _ }
```

Fold
```
keyedStream.fold("start")((str, i)
    => { str + "-" + i })
```

Aggregations
```
keyedStream.sum(0)
```

Window
```
keyedStream.window(

TumblingEventTimeWindows.of(Time.seconds(5))

SlidingProcessingTimeWindows.of(
  Time.seconds(10), Time.seconds(5))

EventTimeSessionWindows.withGap(Time.minutes(10))

)
```

WindowedStream

Reduce
```
wstream,reduce { (v1, v2) =>
  (v1._1, v1._2 + v2._2)
```

Fold
```
wstream.fold(“”) { (acc, v) =>
  acc + v._2 }
```

Aggregate(Associative)
```
wstream.aggregate(Sum..)
```

ProcessWindowFunction:
```
.process(new MyProcessWindowFunction())

class MyProcessWindowFunction
extends ProcessWindowFunction[(String, Long), String,
String, TimeWindow] {

 def process(key: String, context: Context,
 input: Iterable[(String, Long)], out: Collector[String]): ()
 = { var count = 0L
      for (in <- input) {
        count = count + 1
      }
      out.collect(s"Window ${context.window} count: $count")}}
```

DataStream

The Process Function

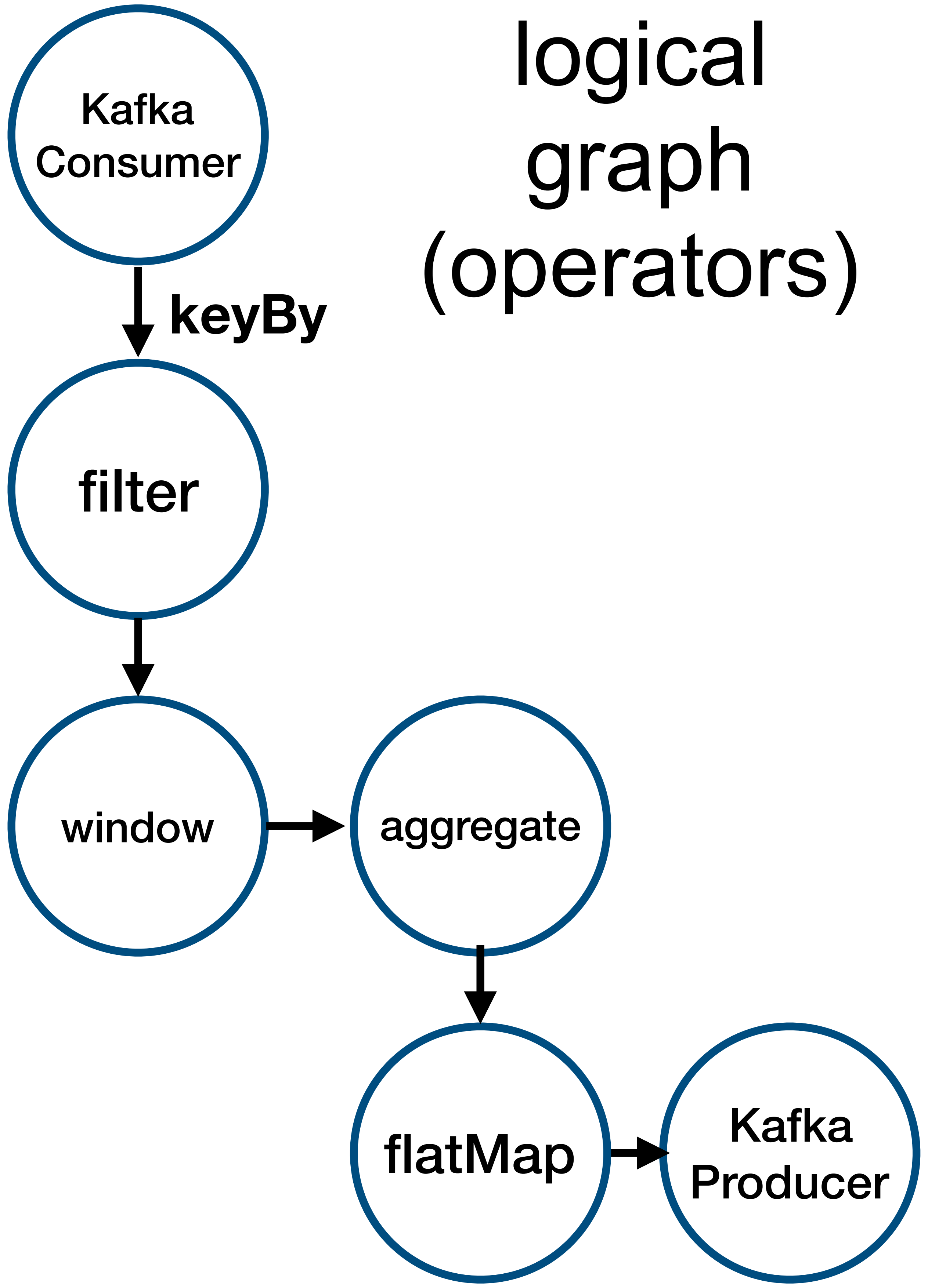
- Encapsulates **any** Event-Processing Logic as: **f(event, state, time)**

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

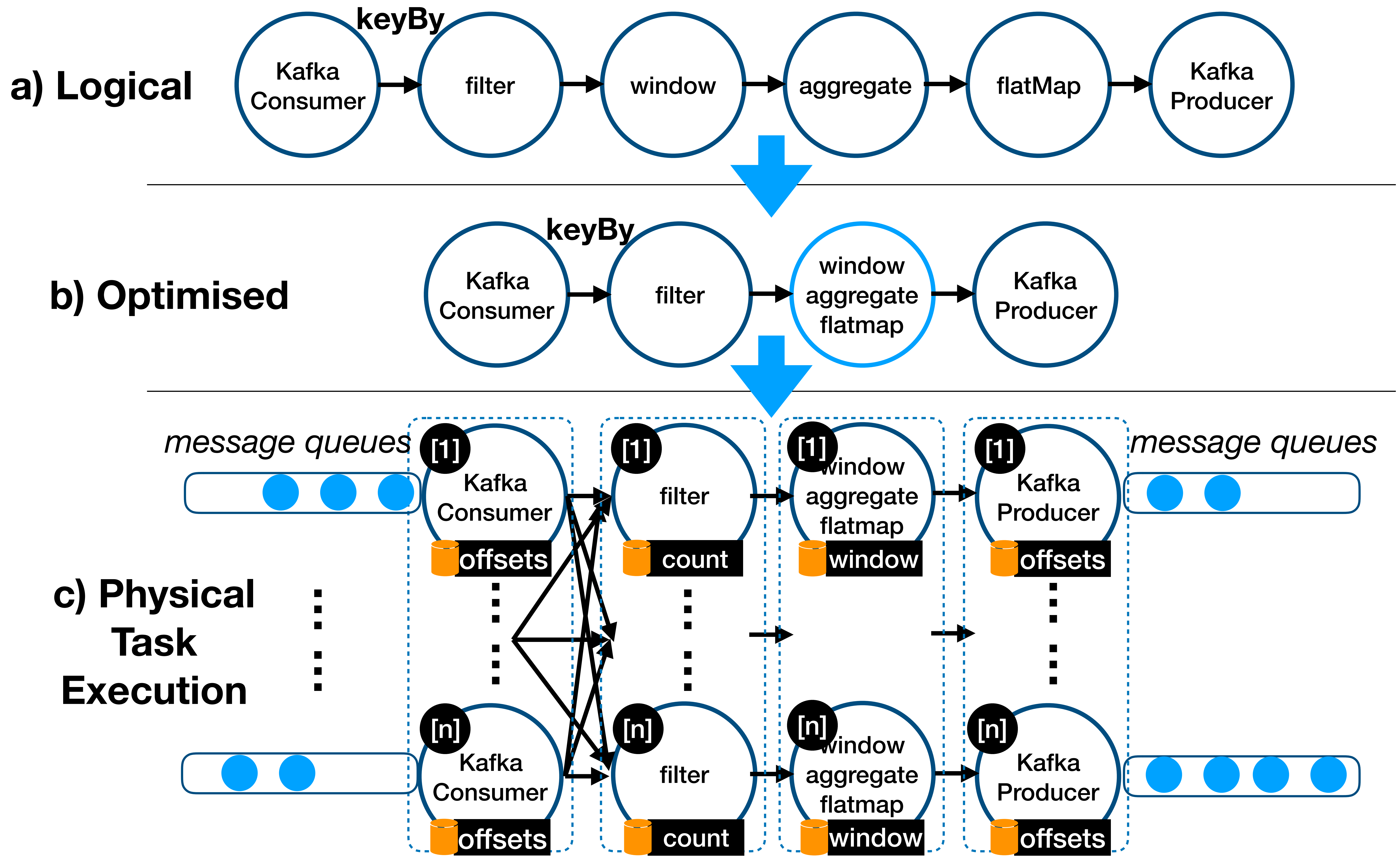
```
class MyCustomLogic extends KeyedProcessFunction[…] {

  /** 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 _ =>
    }
  }
}
```

Fire Detection with the DataStream API)

```
case class SensorEvent(sensorID: Long, temperature: Int);
case class TemperatureWarning(sensorID: Long, temperature: Int);
```



Fire Detection with the DataStream API)



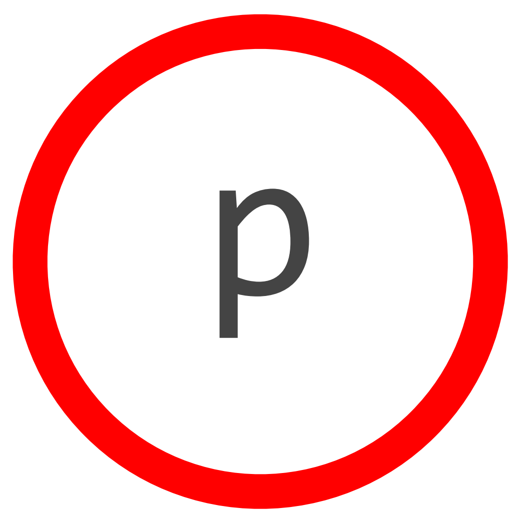
Task computation is not staged but can go on **indefinitely**.

How can we achieve **reliable processing** at the presence of failures, reconfiguration, migration etc.?

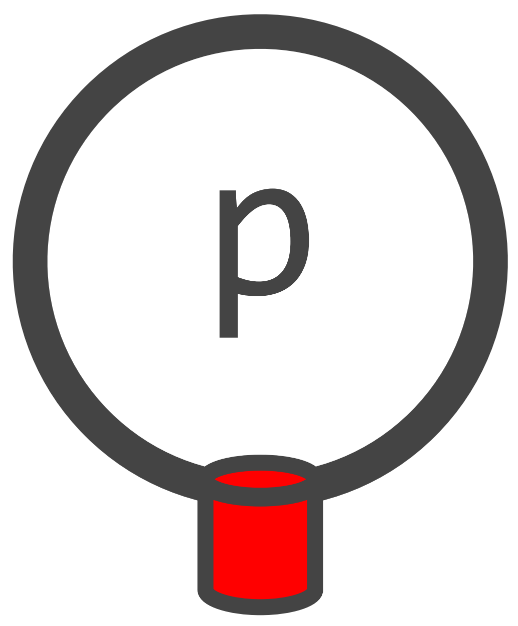
Part II

State Management in Apache Flink

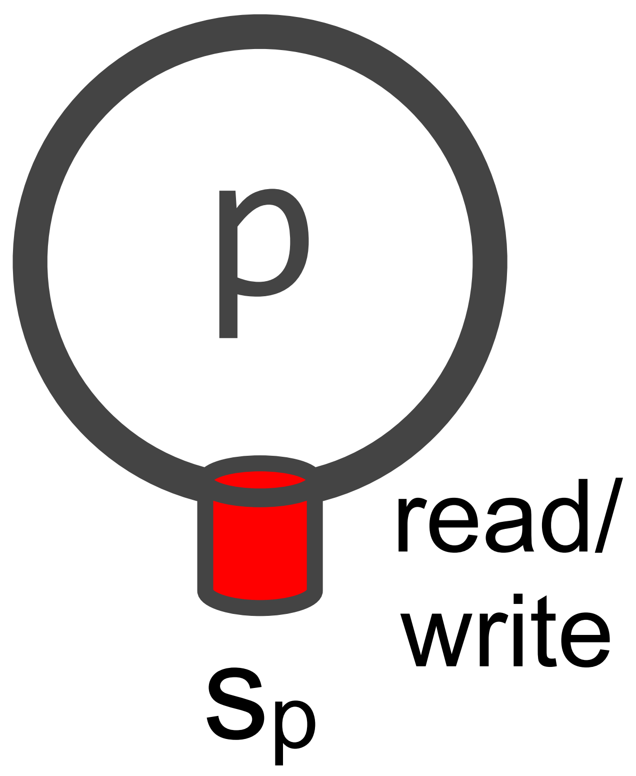
Event Processing Model



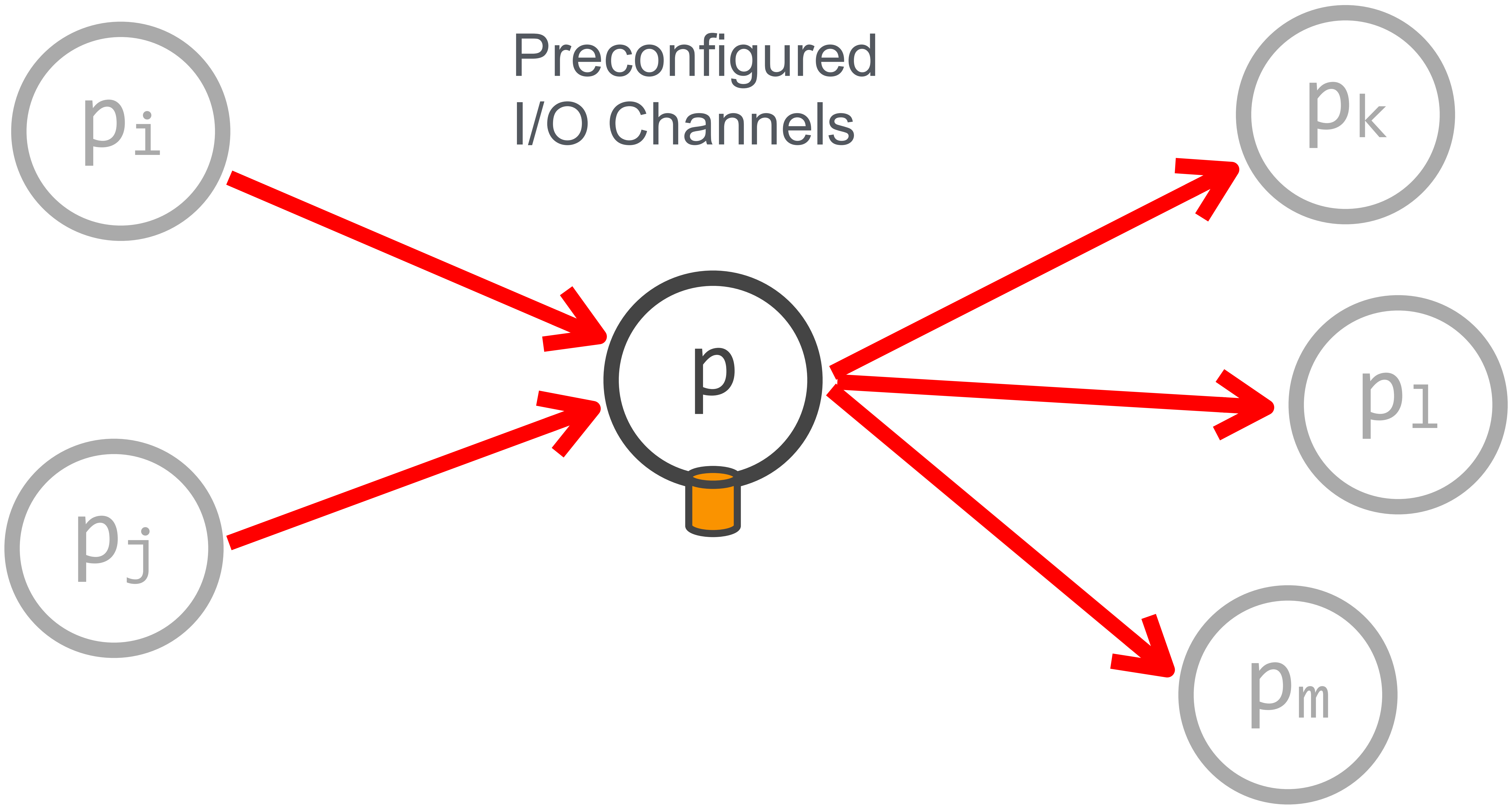
Event Processing Model



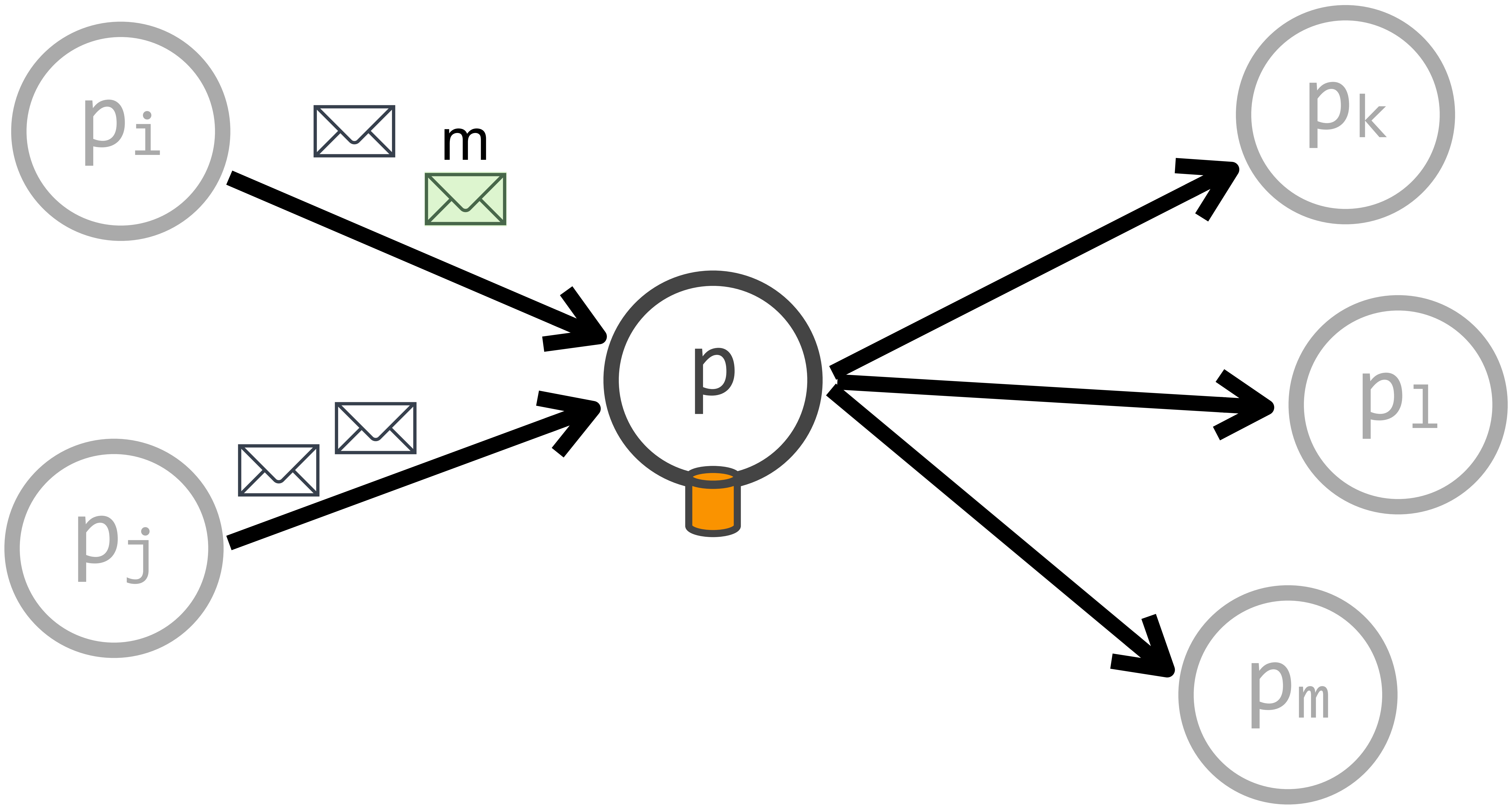
Event Processing Model



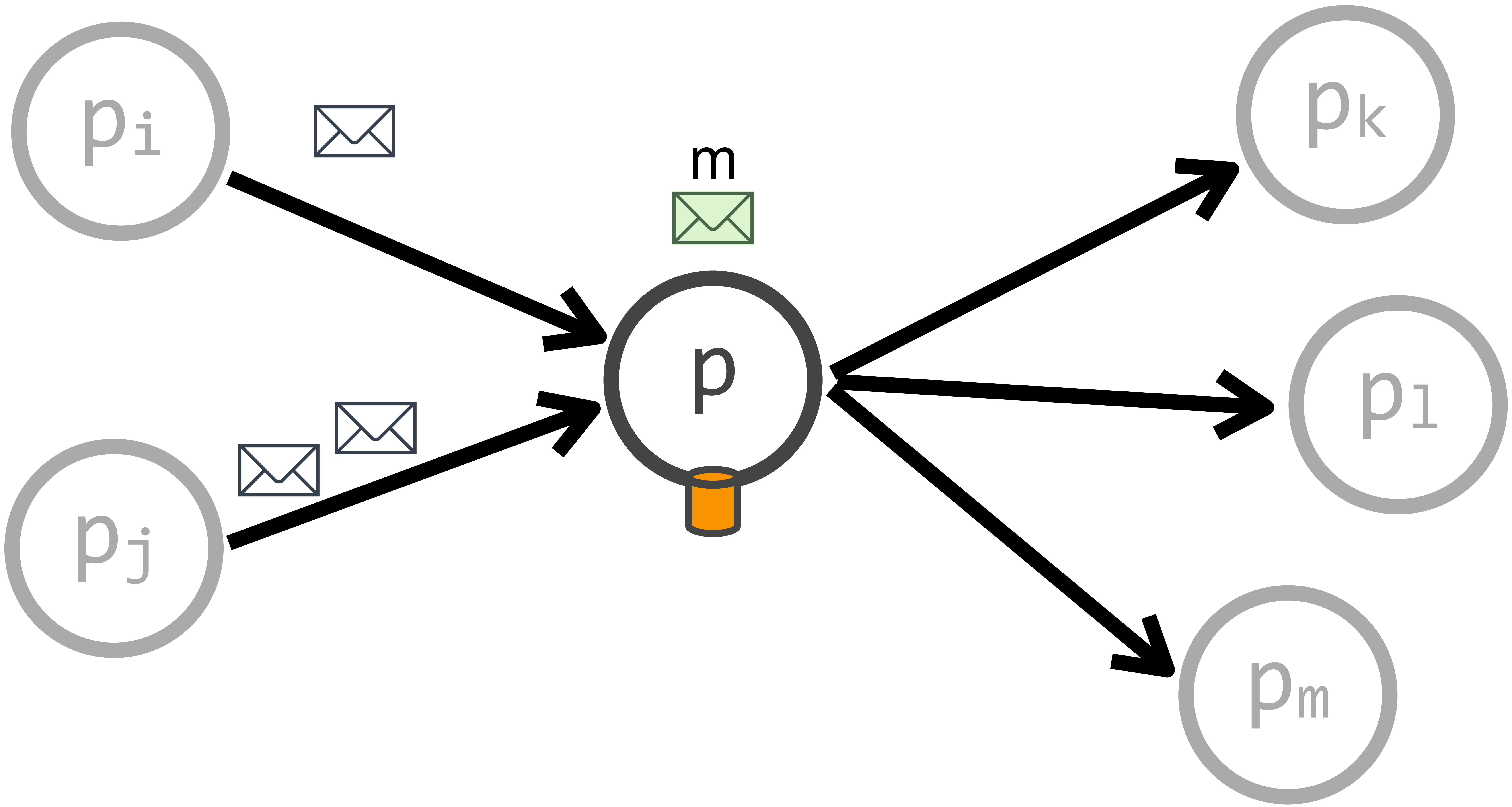
Event Processing Model



Event Processing Model

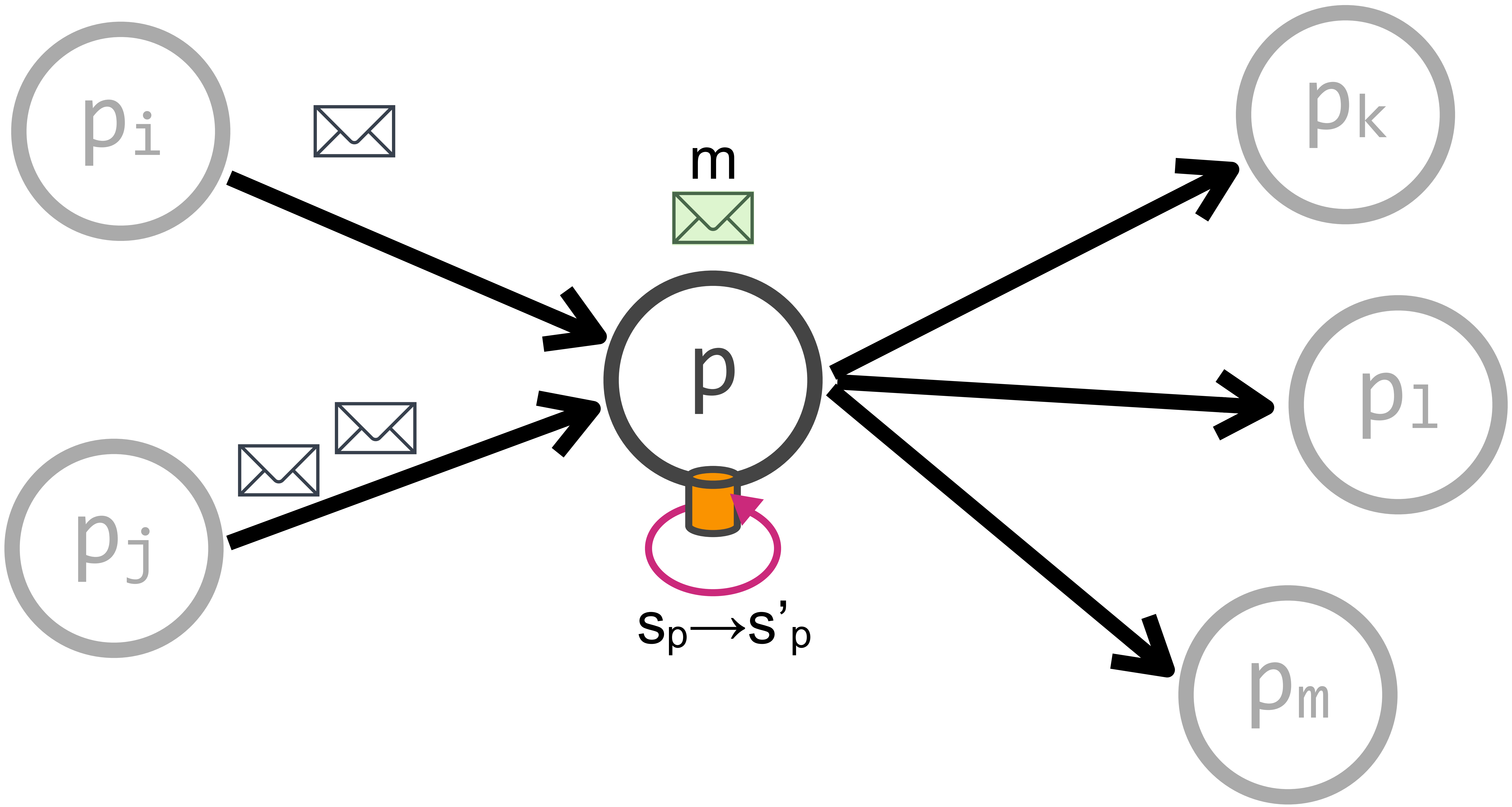


Event Processing Model



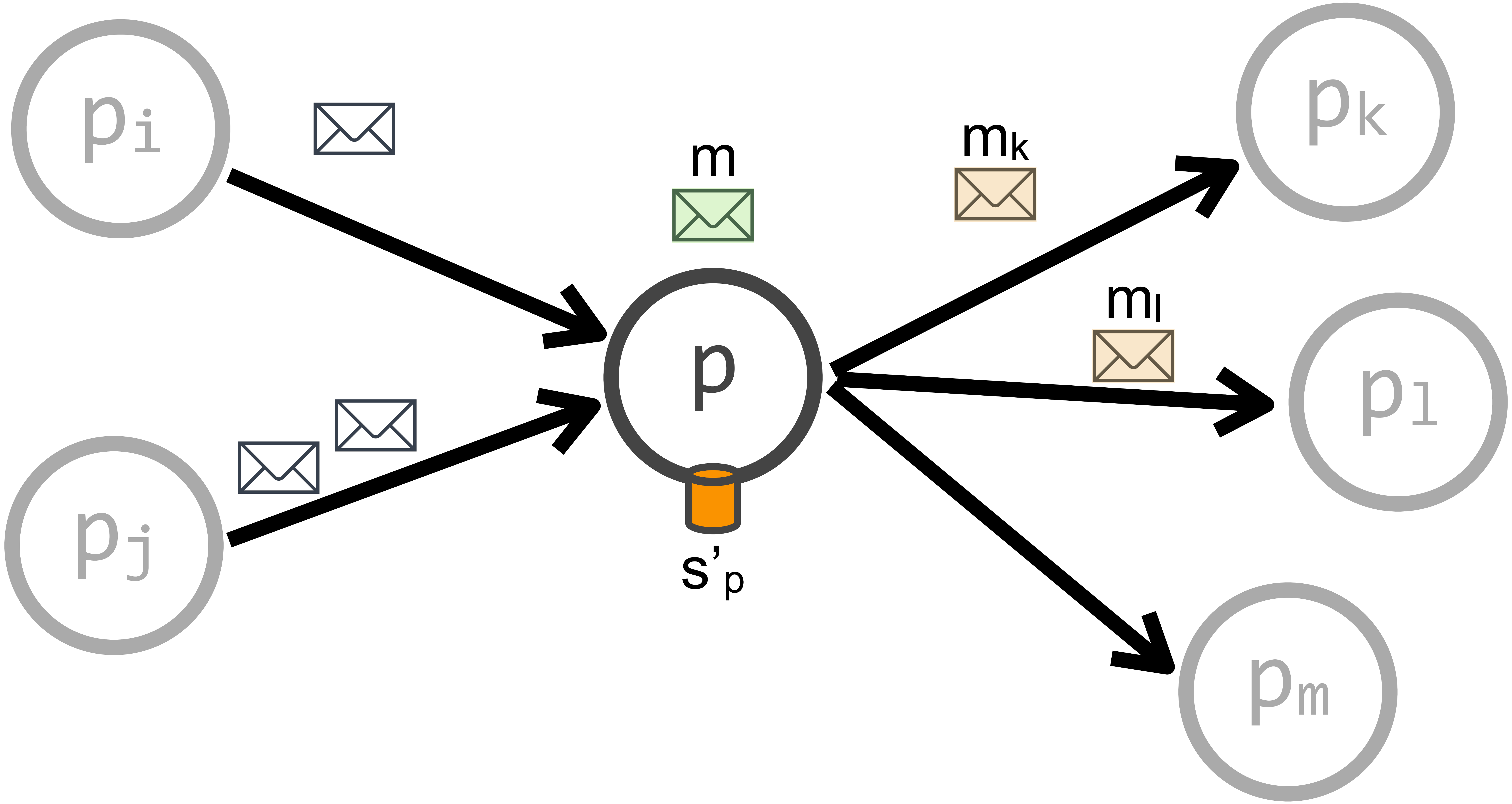
Action:{ ⟨recv,m⟩}

Event Processing Model



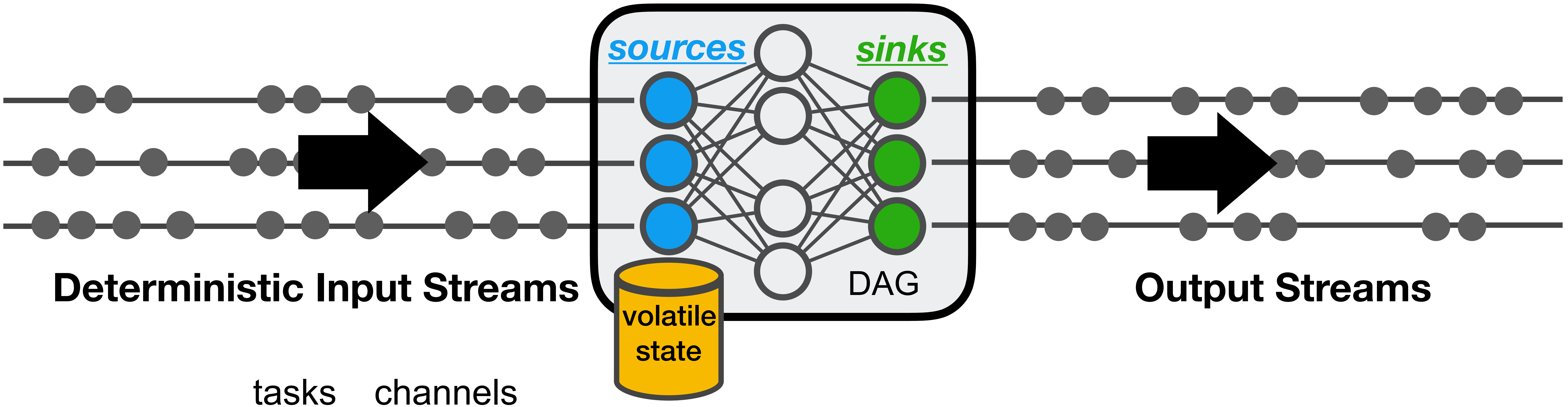
Action:{ ⟨recv,m⟩, ⟨$s_p \rightarrow s'_p$⟩ }

Event Processing Model



Action:{ $\langle$recv,m$\rangle$, $\langle s_p \longrightarrow s'_p \rangle$, $\langle$send,$m_k\rangle$, $\langle$send,$m_l\rangle$ }

Stream Process Graphs

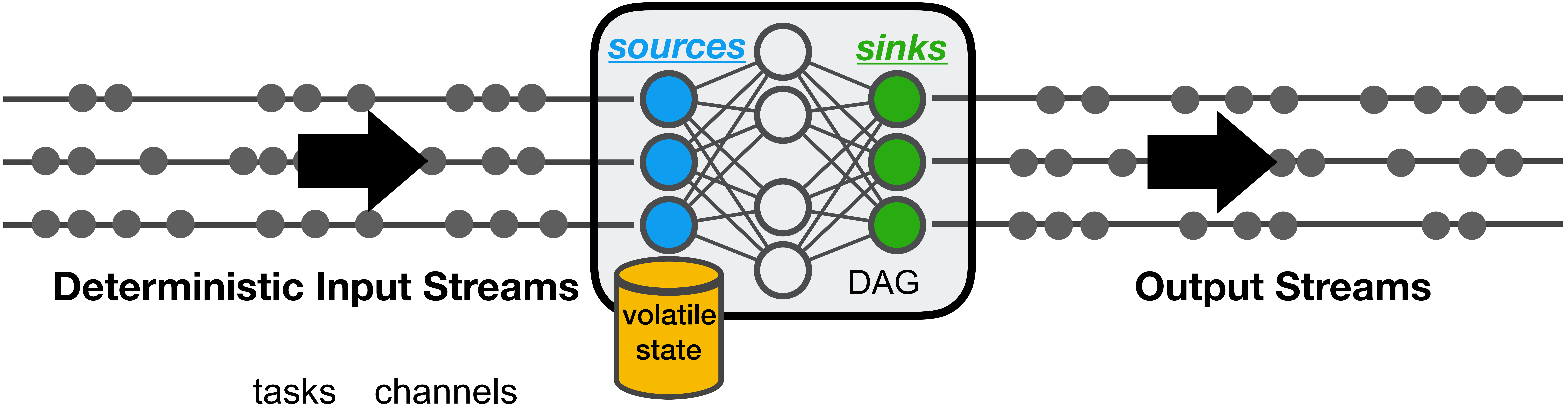


tasks channels

System : $\{\Pi, \mathbb{E}\}$

System Execution : $\ldots \rightarrow \{\Pi_*, M\} \rightarrow \{\Pi'_*, M'\} \rightarrow \ldots$

Stream Process Graphs



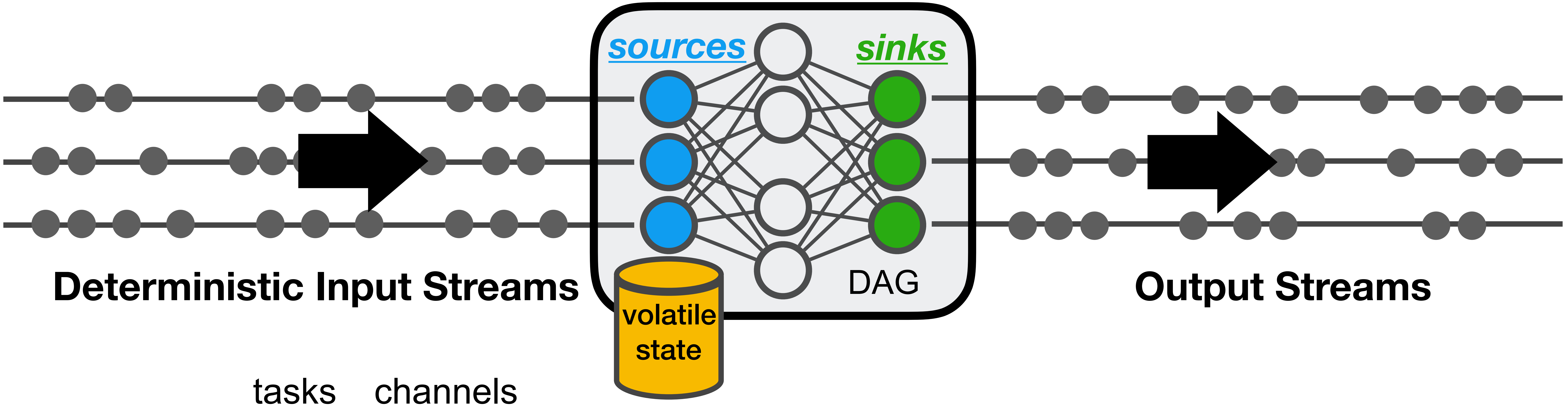
tasks channels

System : $\left\{\Pi, \mathbb{E}\right\}$

Task Actions

System Execution : $\ldots \boxed{\rightarrow} \{\Pi_*, M\} \boxed{\rightarrow} \{\Pi'_*, M'\} \boxed{\rightarrow} \ldots$

Stream Process Graphs



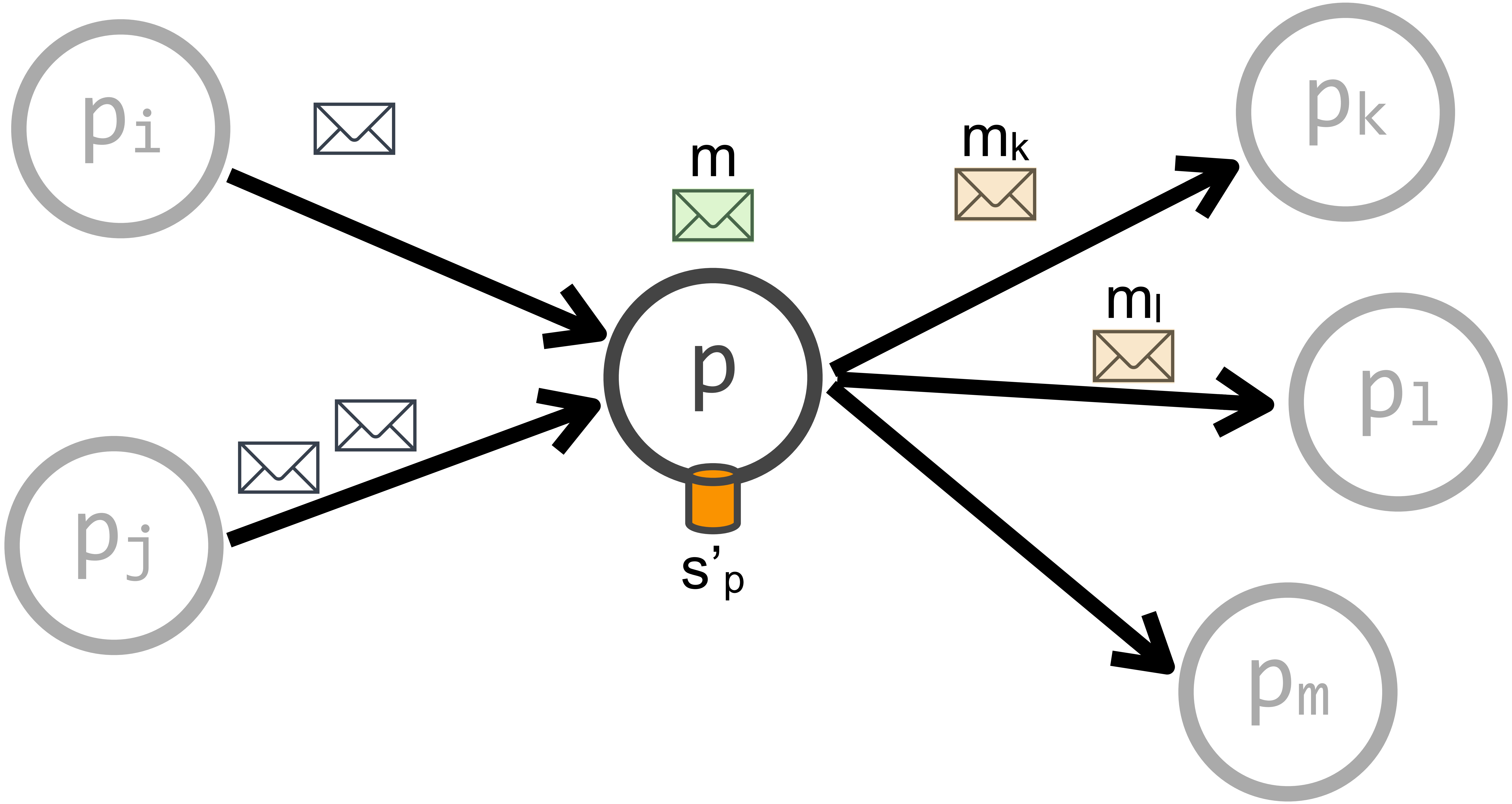
tasks channels

System : $\{\Pi, \mathbb{E}\}$

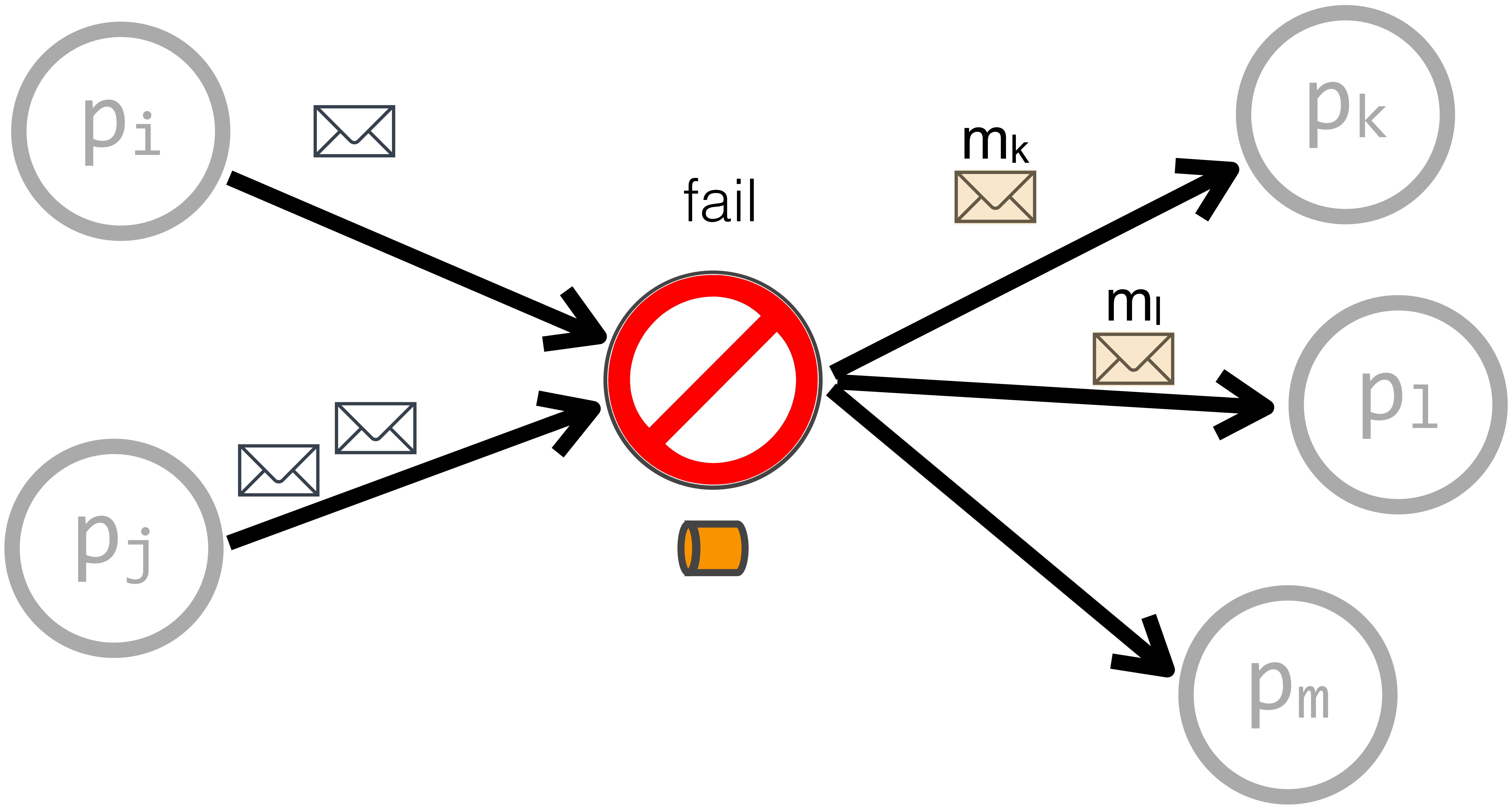
System Configurations (states, messages in-transit)

System Execution : $\ldots \rightarrow \boxed{\{\Pi_*, M\}} \rightarrow \boxed{\{\Pi'_*, M'\}} \rightarrow \ldots$

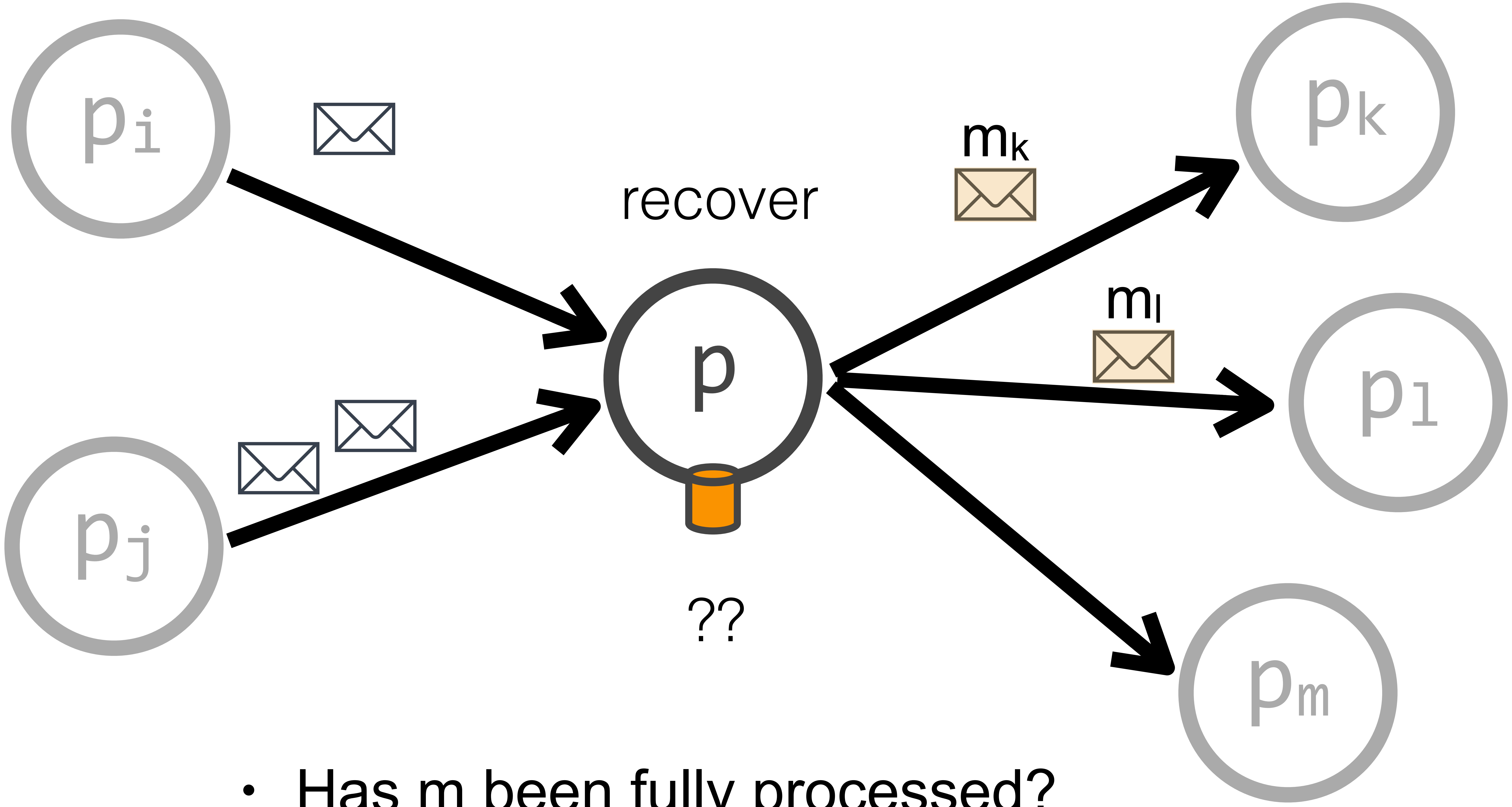
Fault Tolerance



Fault Tolerance



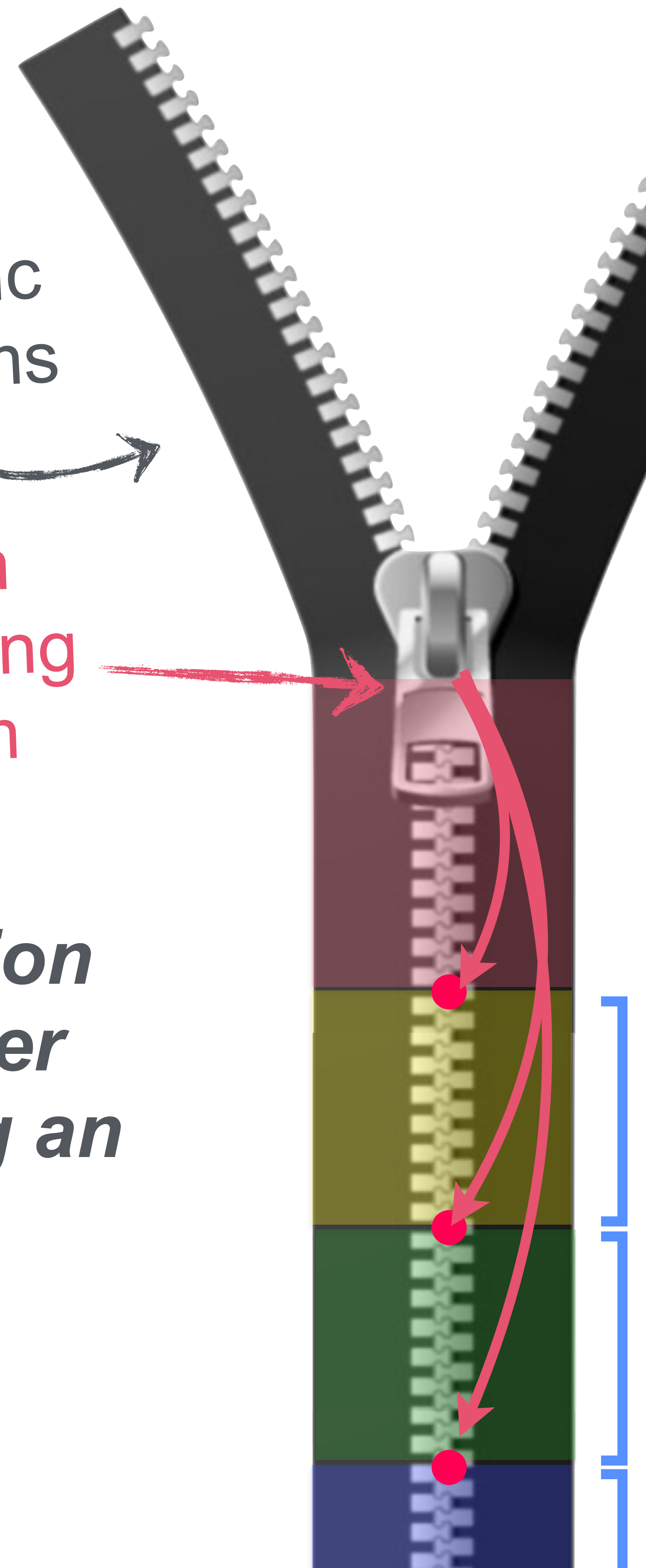
Fail Recovery



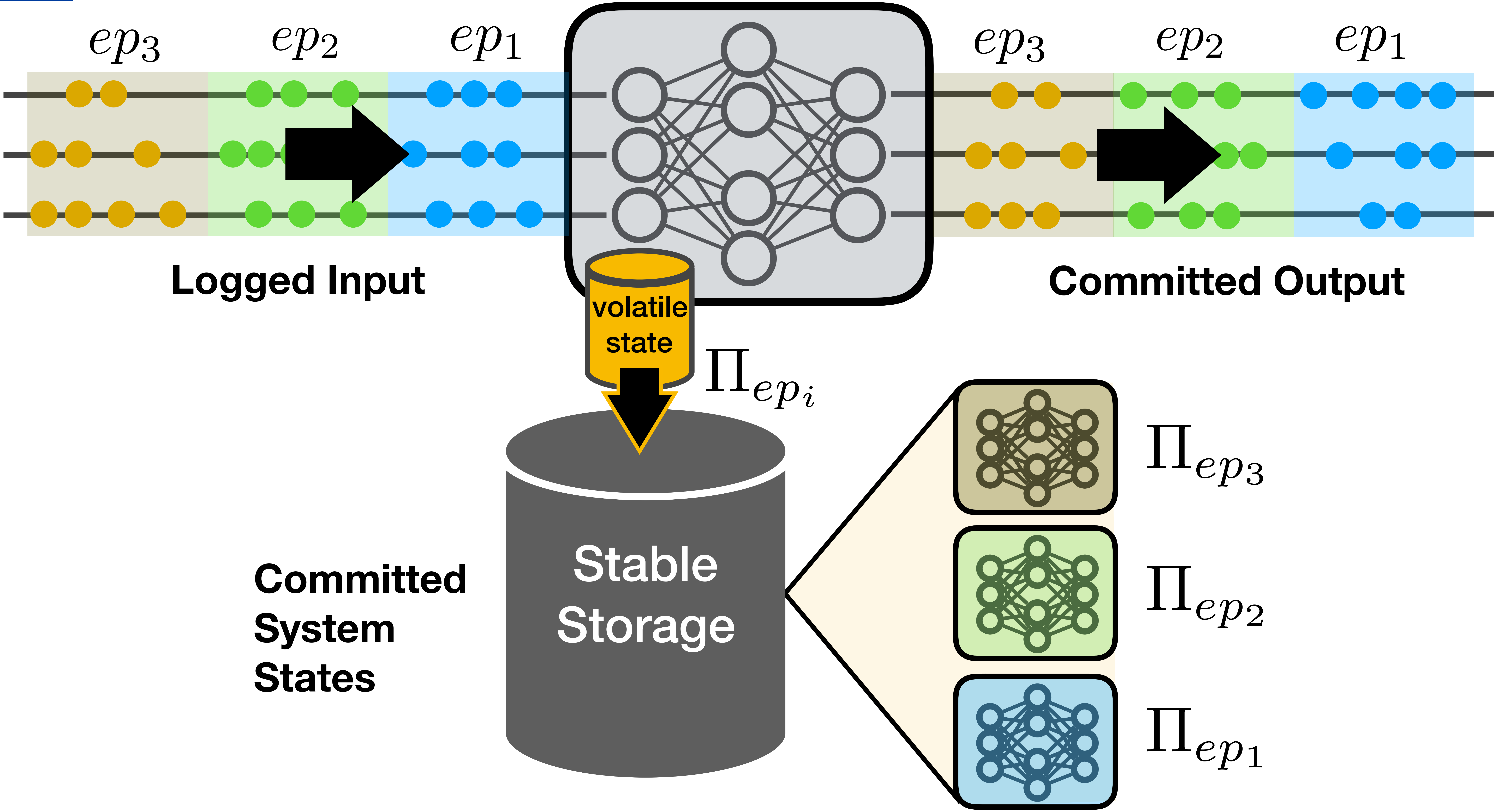
- Has m been fully processed?
- Have m_k and m_l been delivered?

Transactional Stream Processing

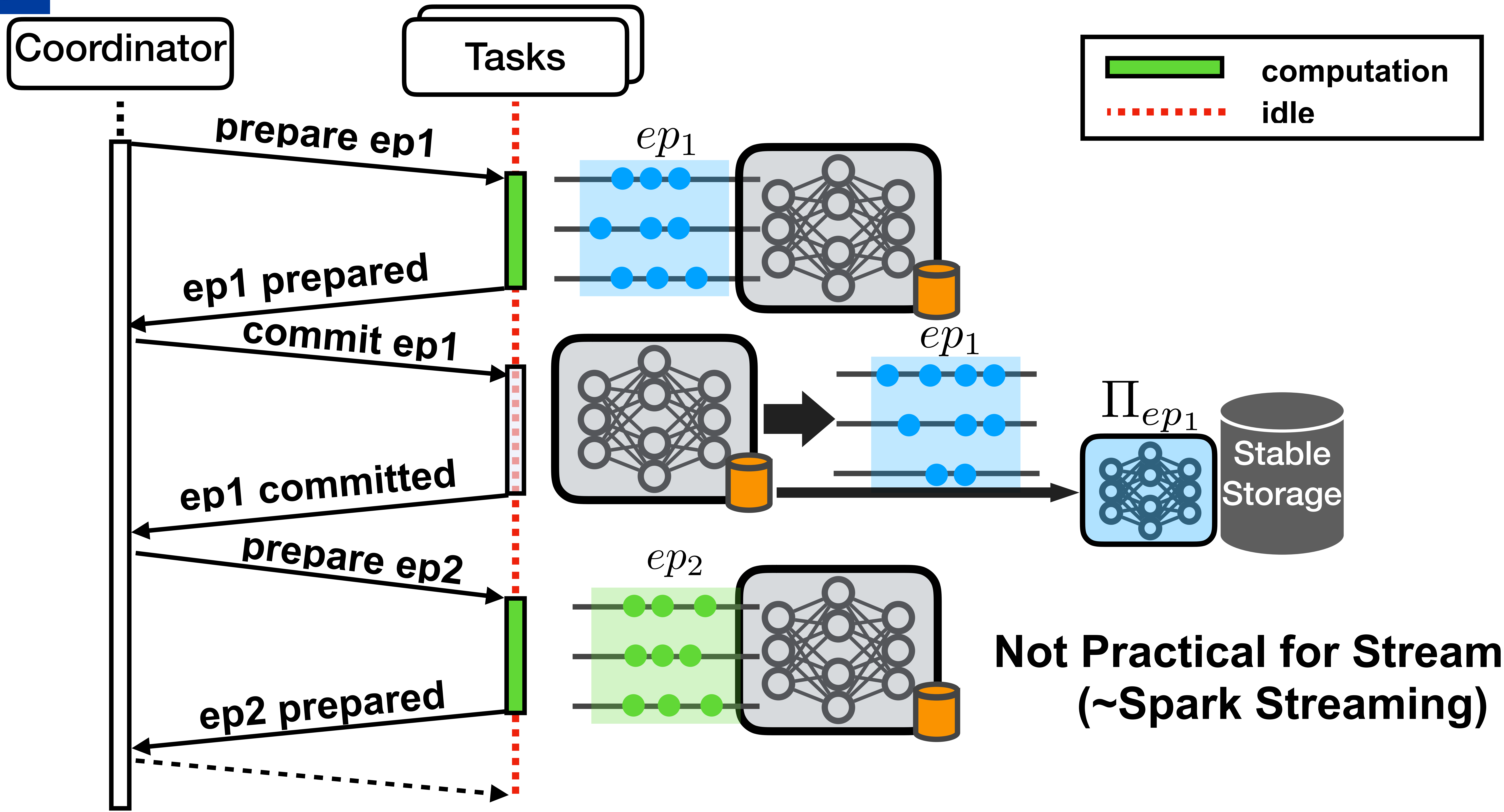
The Intuition



Approach Overview

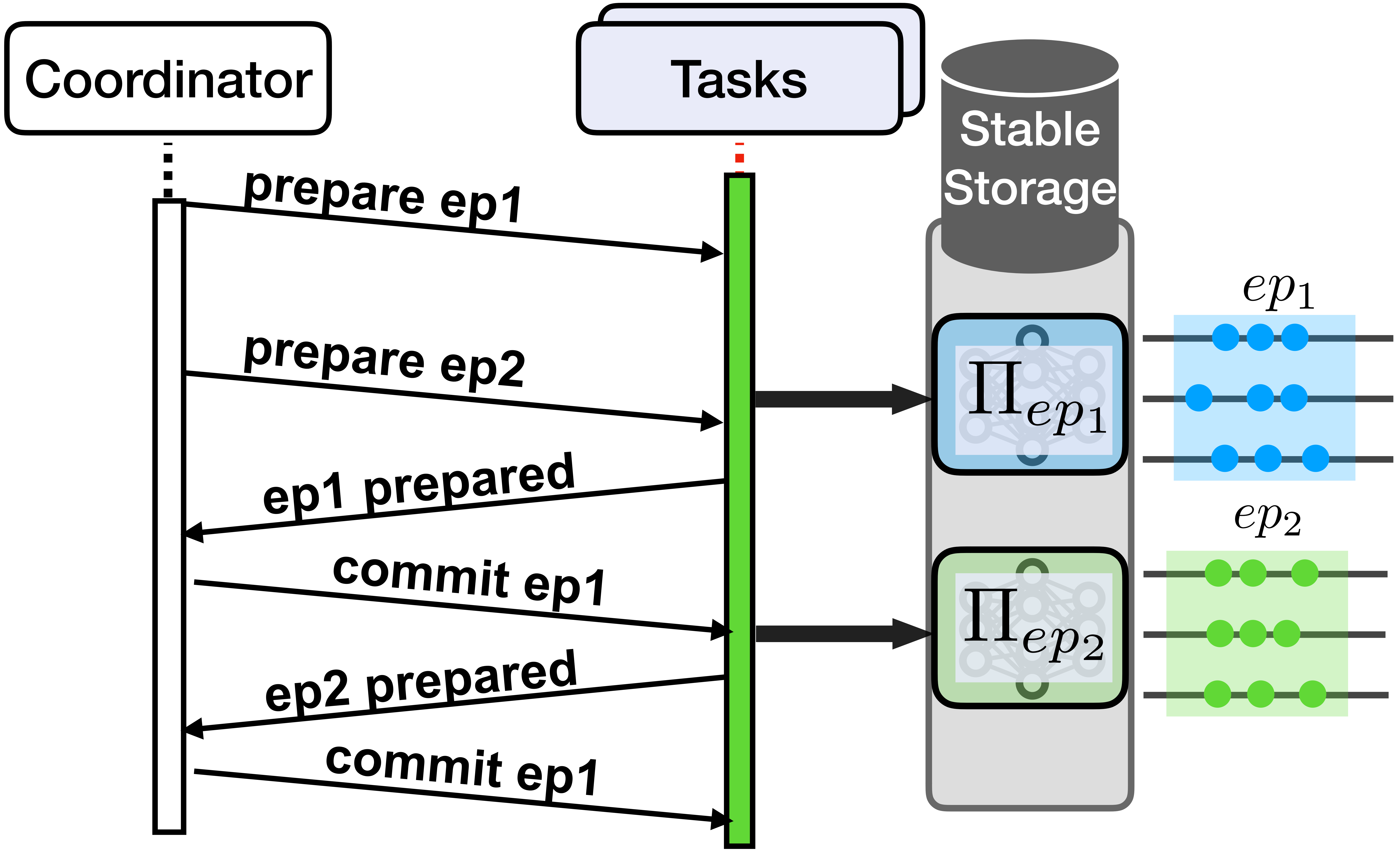


Synchronous Epoch Commits



Not Practical for Streaming (~Spark Streaming)

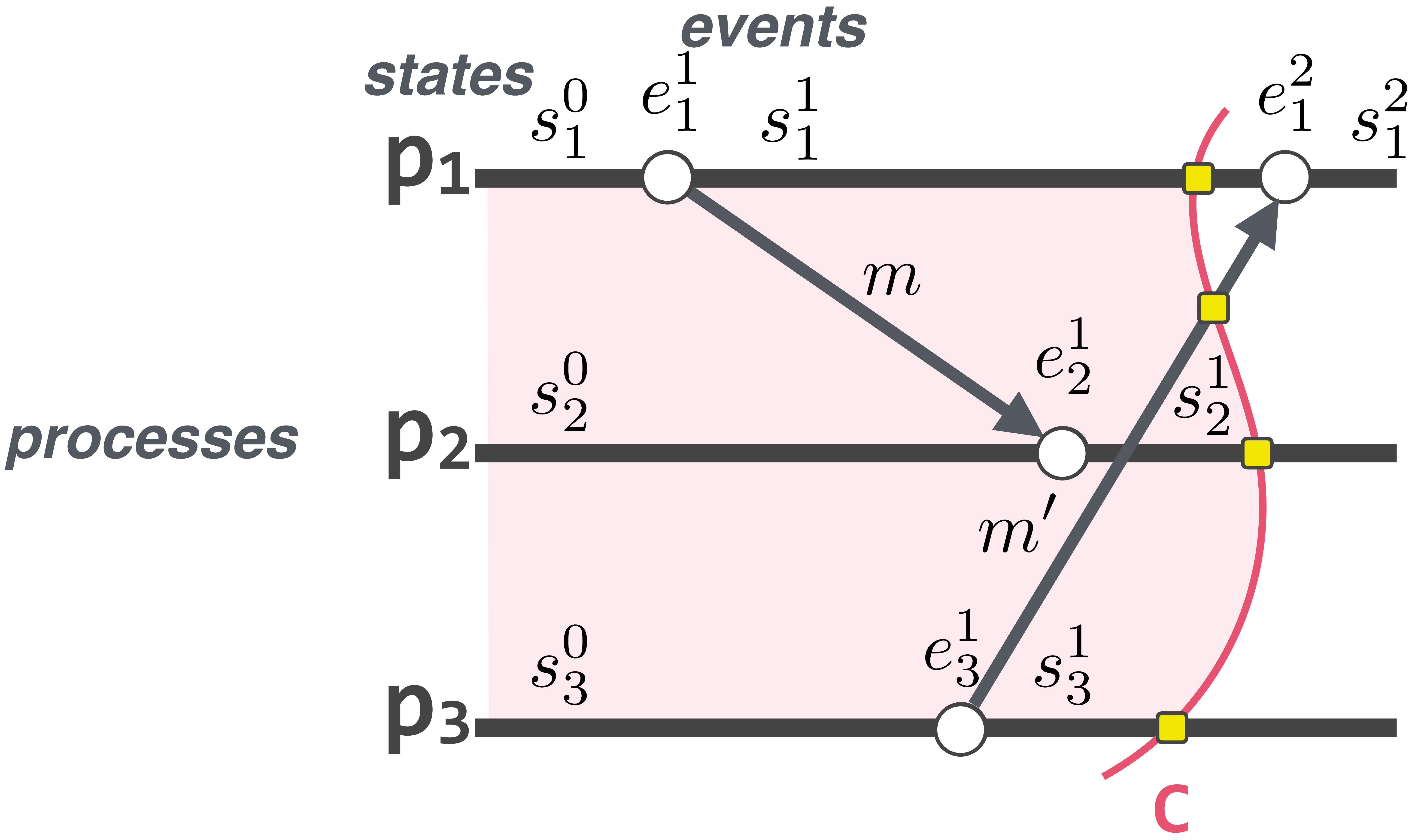
Asynchronous Epoch Commits



How? Using Distributed Snapshotting

Recap: Snapshotting Protocols

Traditional Snapshotting Protocols: Distributed Algorithms that capture system states that form a **distributed cuts** in a system execution

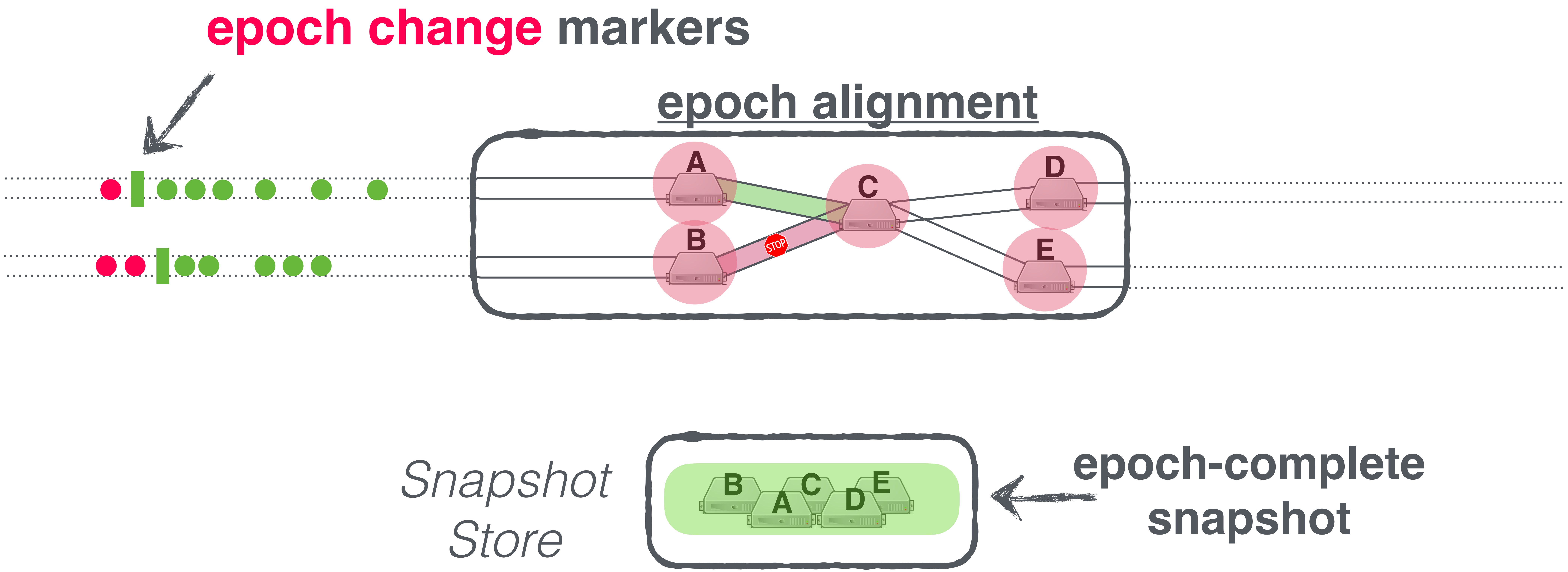


Snapshot of C

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

But we need to complete *in-progress* computation in Stream Processing

Epoch Snapshotting Algorithm



State Management in Practice

1. End-to-End Guarantees

2. Reconfiguration

Snapshots

3. Version Control

4. Isolation

The Epoch Commit Protocol
1a Prepare (insert markers)
1b Pre-Commit (snapshot)
1c Prepared/Aborted
2a Commit
2b Mark Committed
Snapshot Coordinator
1a
1c
2a
2b
1b
Output Logs
ep_3
ep_2
ep_1
ep_3
ep_2
ep_1
External State Backend
Input Logs
(already committed)
pre-committed
committed
1b
Local State Backend
pending snapshot
External File System
Π_{ep_3}
Π_{ep_2}
Π_{ep_1}

State Management in Practice

1. End-to-End Guarantees

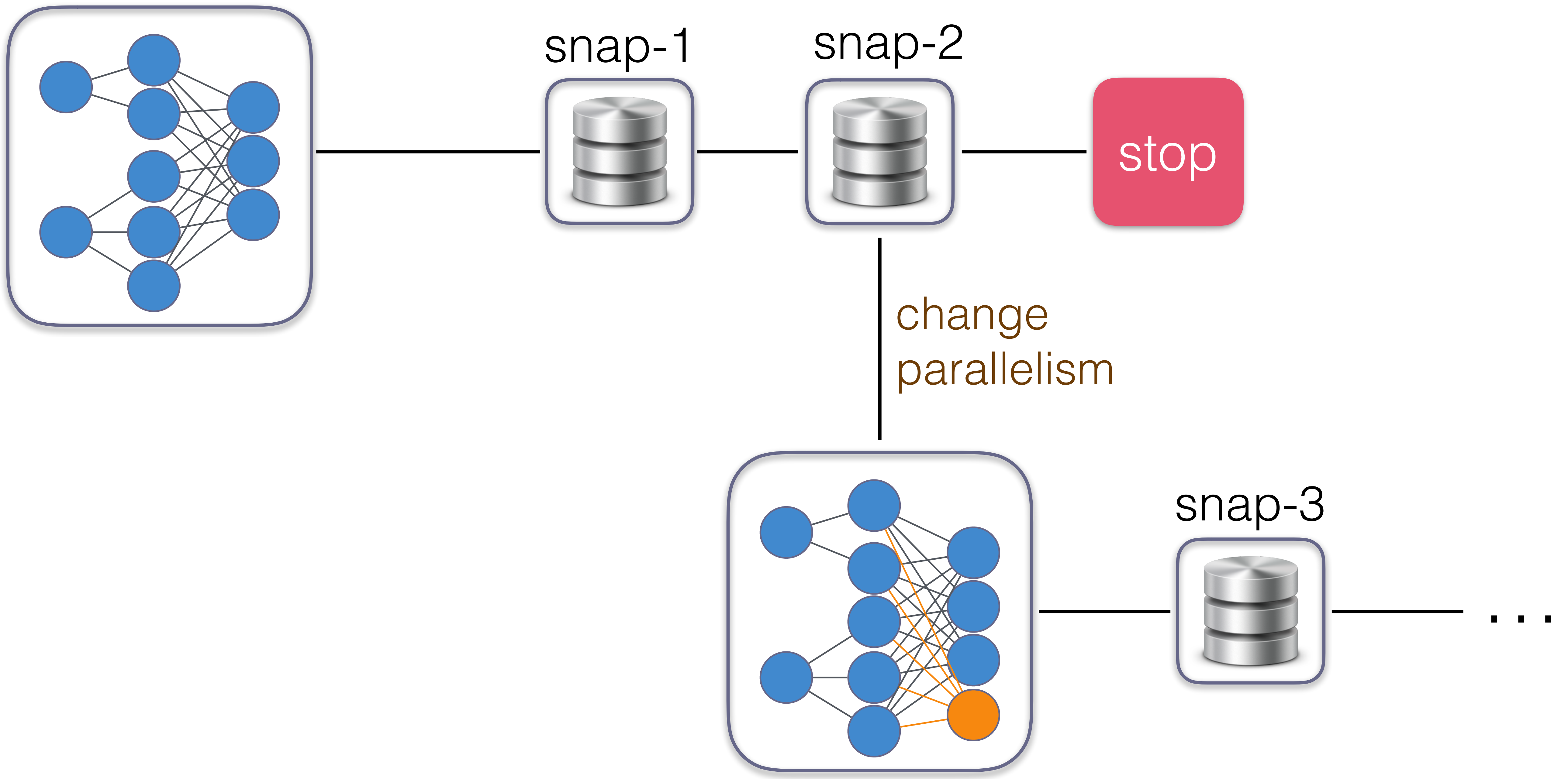
2. Reconfiguration

Snapshots

3. Version Control

4. Isolation

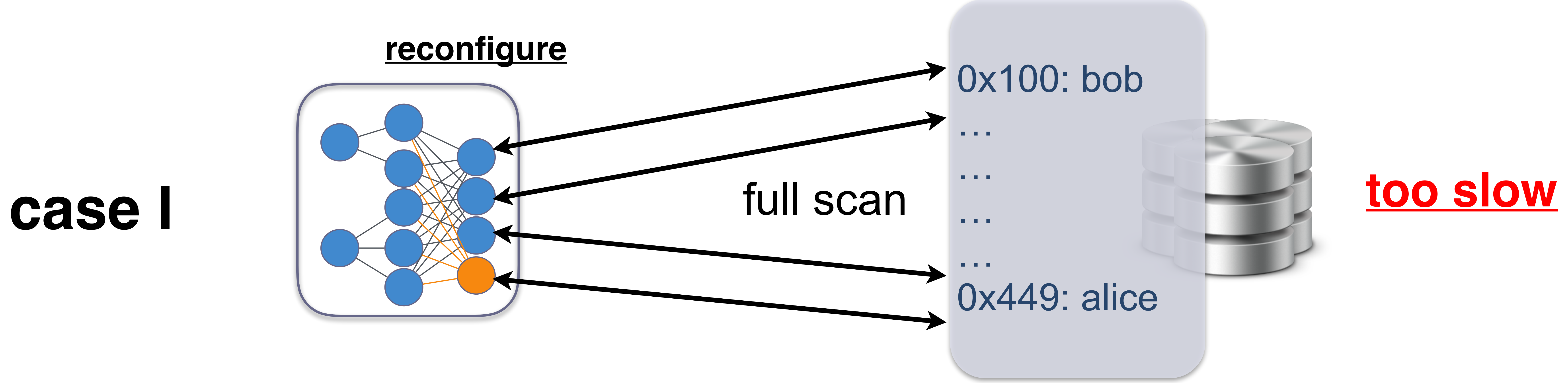
Dataflow Reconfiguration



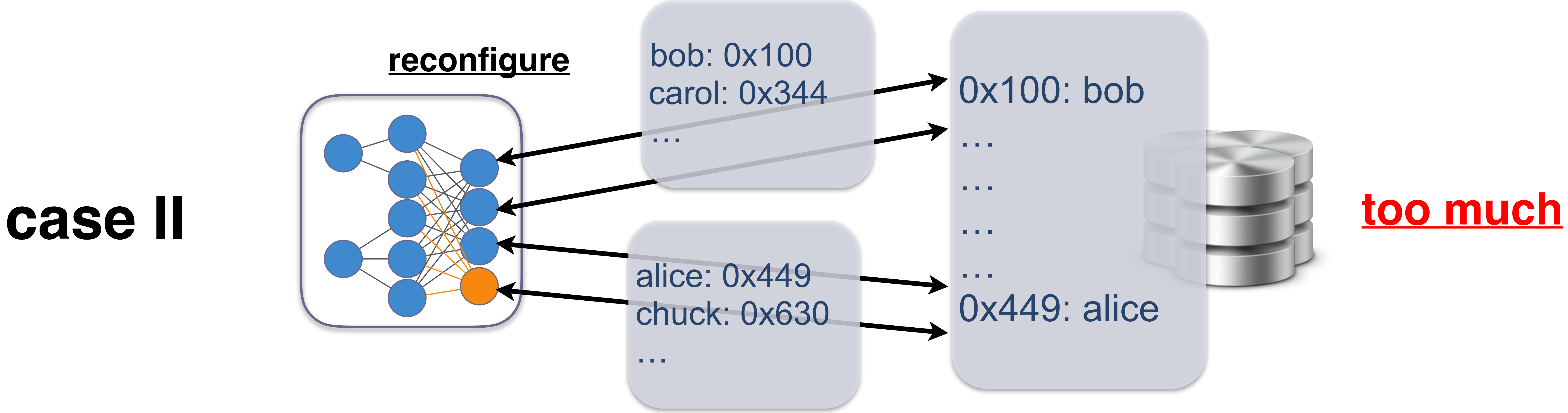
Problem: *How is state* ***repartitioned*** *from a snapshot?*

Reconfiguration: The Issue

Scan Remote Storage for Responsible Keys

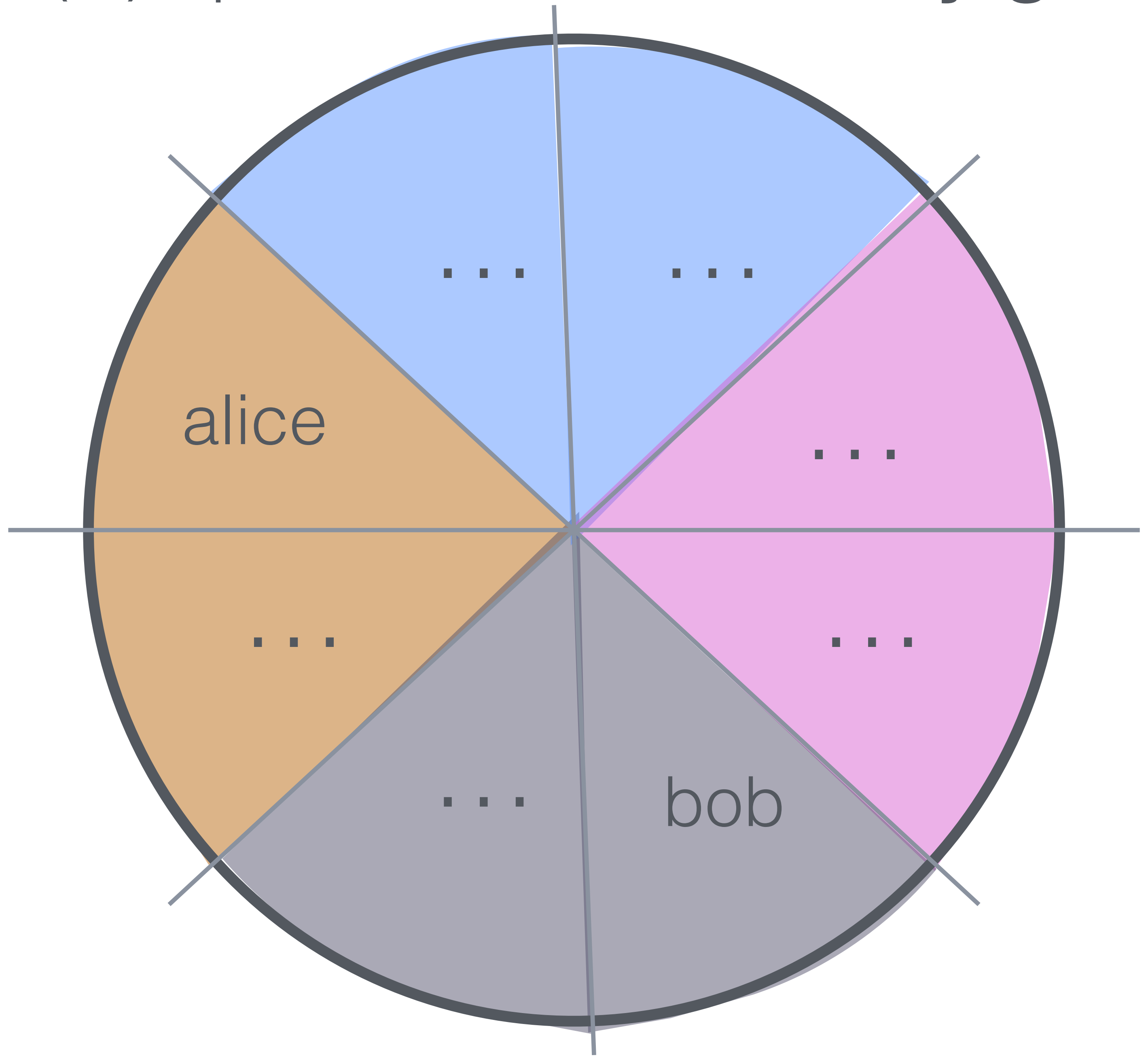


Include Key Locations in Snapshot Metadata



State Partitioning

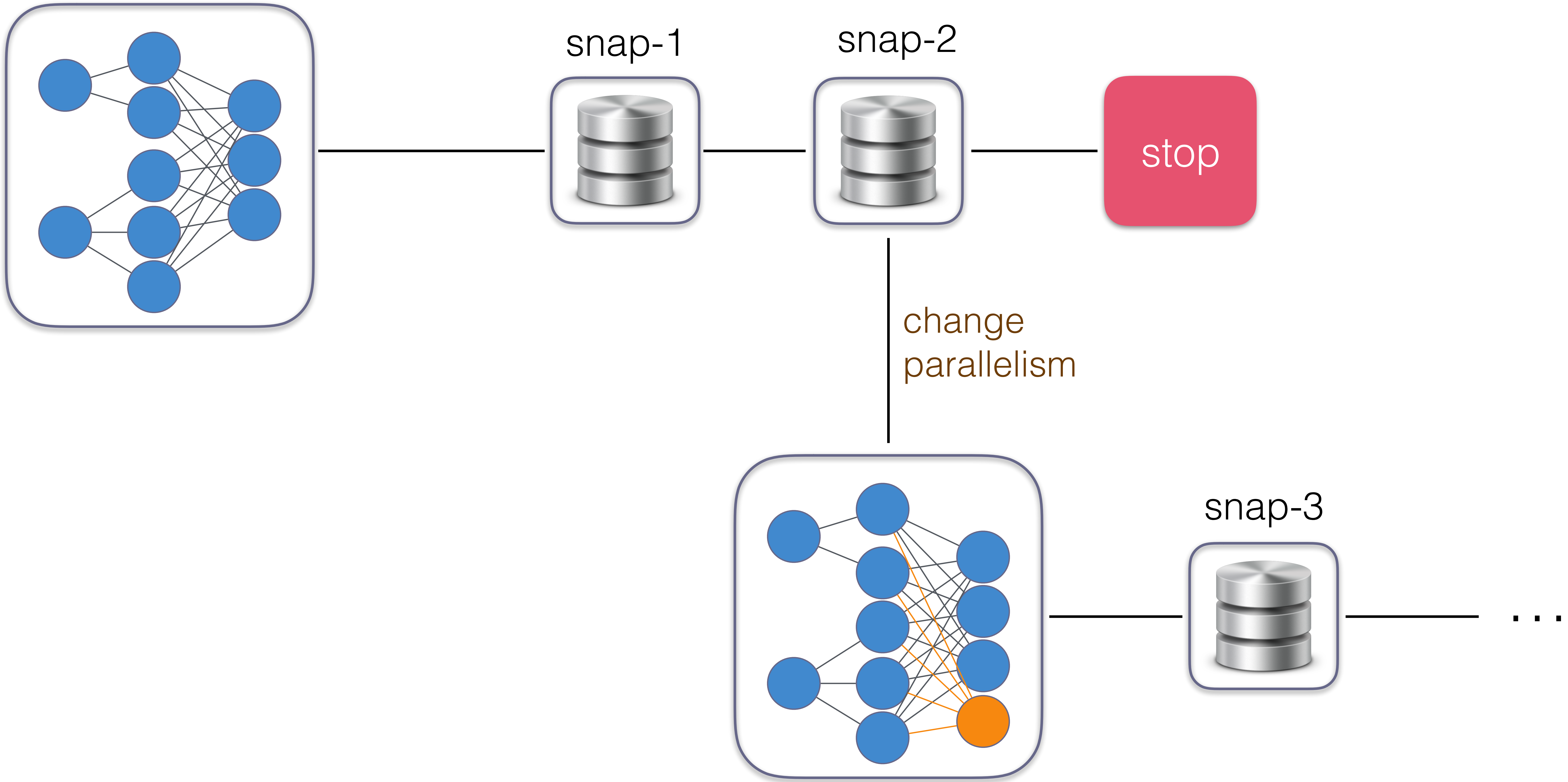
Pre-partition state in
hash(K) space, into **fixed n key-groups**



- **Snapshot Metadata:** *Contains a reference per stored Key-Group (less metadata)*
- **Reconfiguration:** ***Contiguous*** *key-group allocation to available tasks (less IO)*

Note: number of key groups controls trade-off between metadata to keep and reconfiguration speed

Usages:Reconfiguration



State Management in Practice

1. End-to-End Guarantees

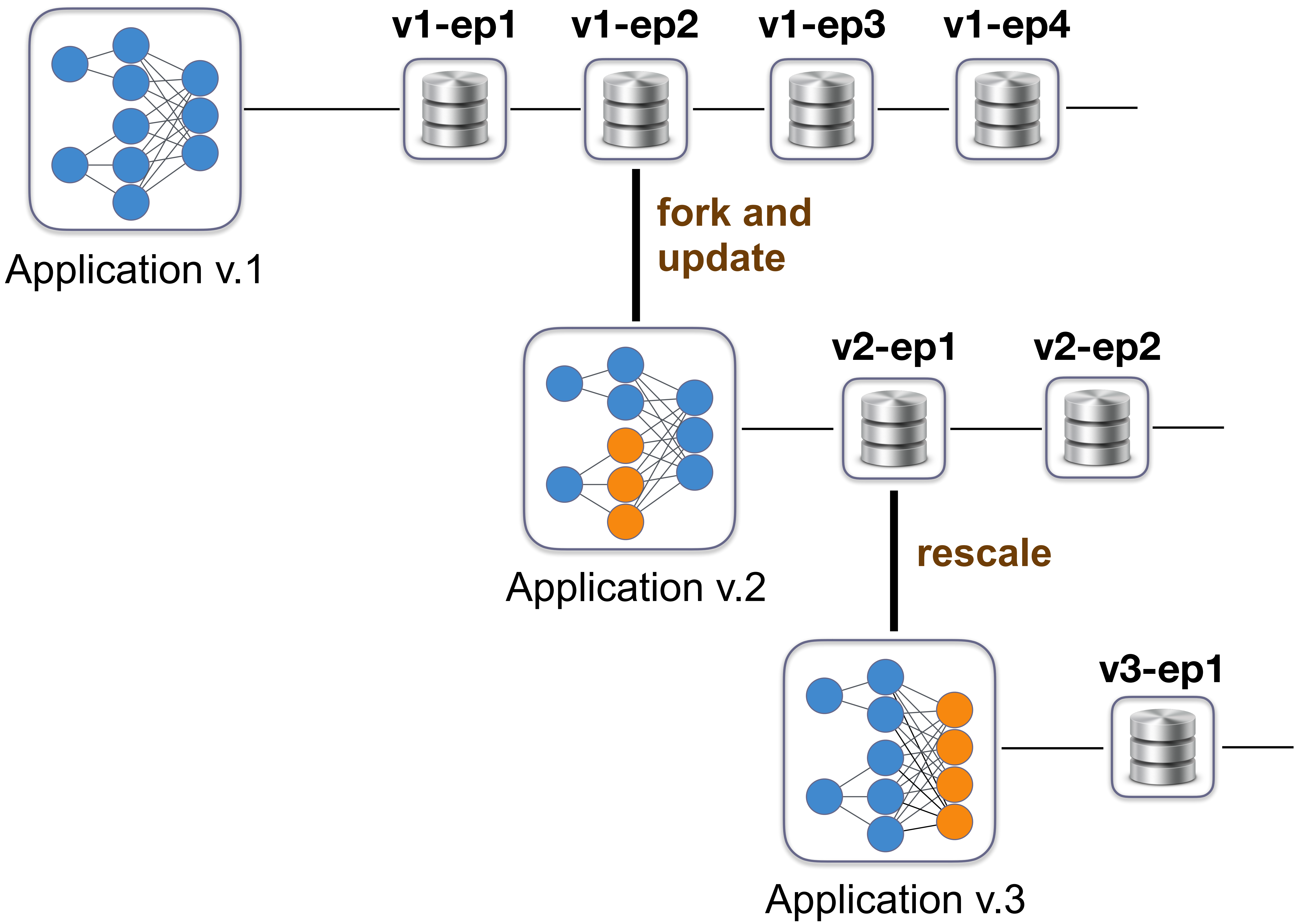
2. Reconfiguration

Snapshots

3. Version Control

4. Isolation

Usages: App Provenance



State Management in Practice

1. End-to-End Guarantees

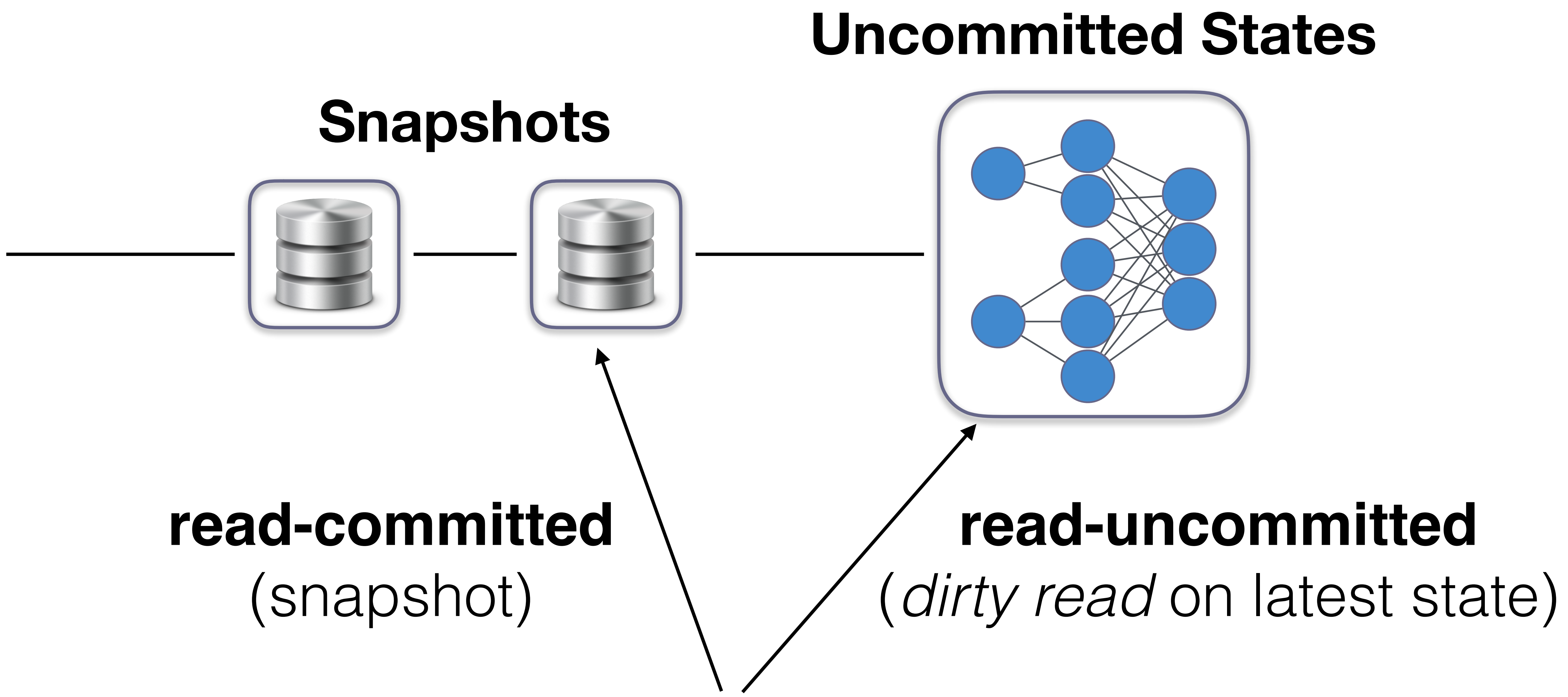
2. Reconfiguration

Snapshots

3. Version Control

4. Isolation

Usages: External Access Isolation



external query

```
select from facebook.userID, clients.name …
inner join clients on …
```

Further Optimisations

- **<u>Asynchronous</u> Snapshots**
  - make triggering snapshots <u>cost-free.</u>
- **<u>Incremental</u> Snapshots**
  - avoid full state copy and commit only **deltas**
  - make overhead of snapshots nearly **constant**
- **Both** are provided by Log-Structure-Merge backends, i.e. **Rocksdb.**

RocksDB

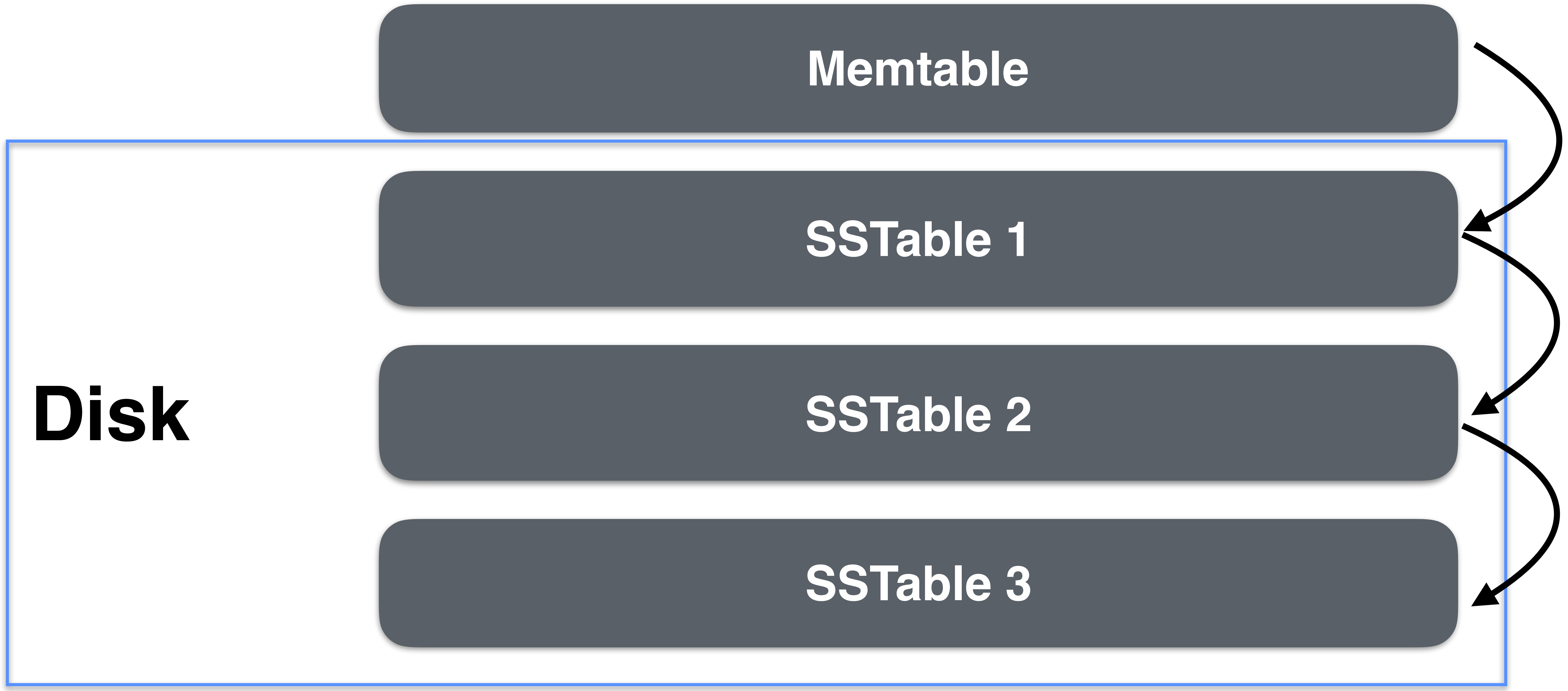
- Embedded (local-only) key-value store used by Flink, Spark, Kafka etc.
- Main Idea: Sequential disk seeks/writes (log) are way faster than random writes (database).

The Memtable

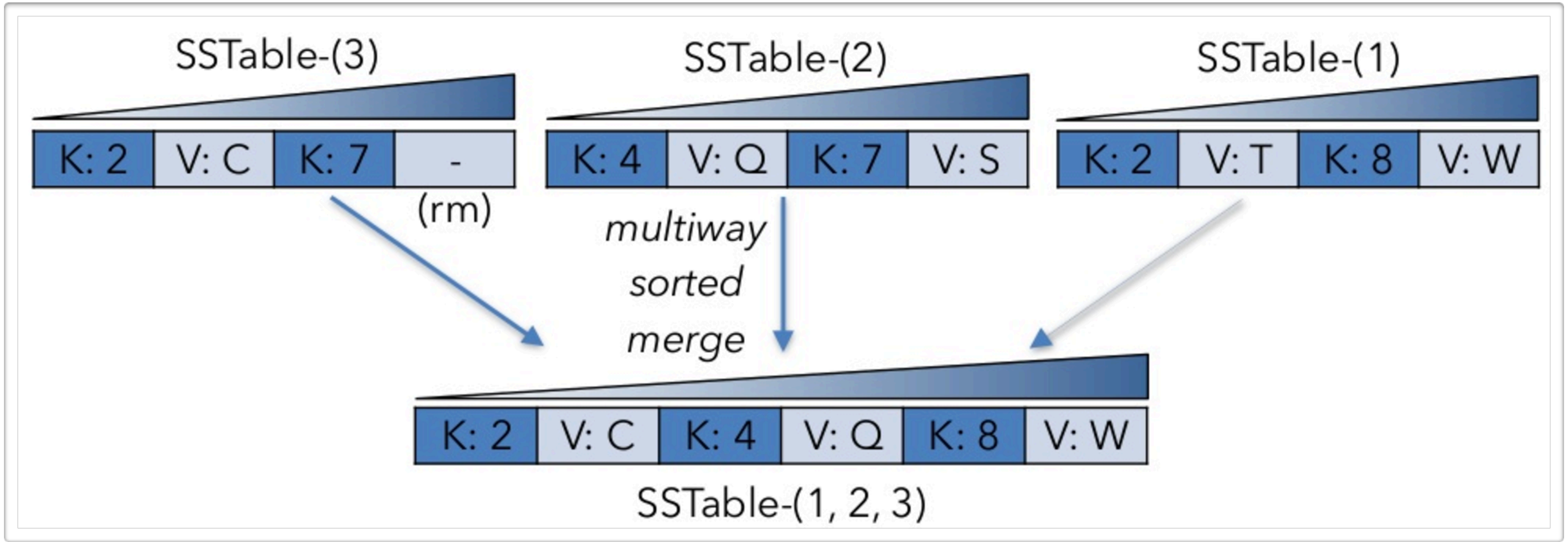
- Mutable in-memory buffer for KV Pairs
- Reads and writes are executed here first
- Is **asycnhronously flashed to disk** and turn into an **SSTable** (on demand or on size limit)

SSTable

- Persisted memtables that have become **immutable.**
- Sorted by **Key**.
- Key Reads start from **memtable** and go down over committed sstables for every miss.
- **Optimisations**: Index/bloomfilter



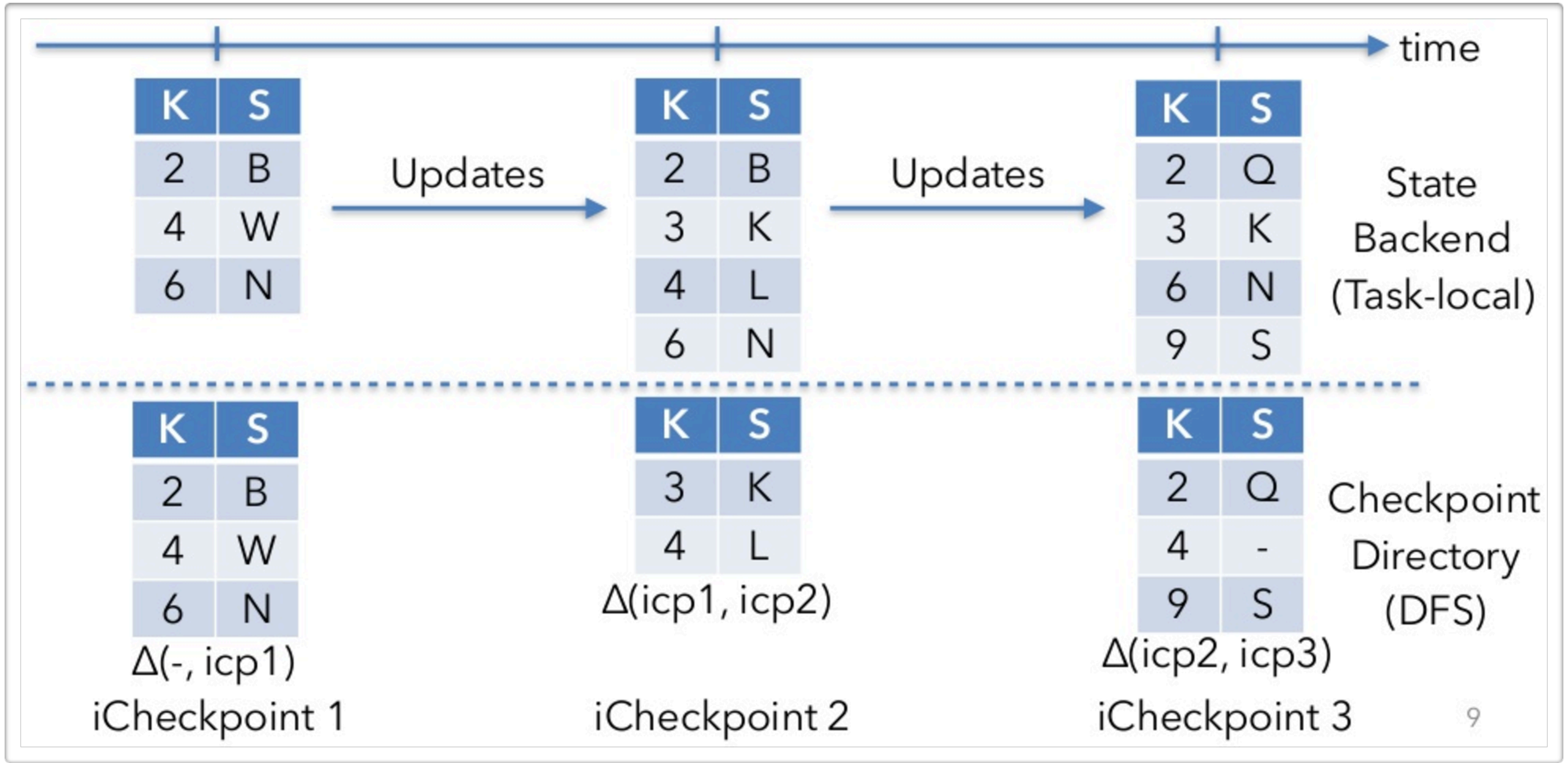
Compaction


SSTable-(3)
K: 2
V: C
K: 7
-
(rm)
SSTable-(2)
K: 4
V: Q
K: 7
V: S
SSTable-(1)
K: 2
V: T
K: 8
V: W
multiway
sorted
merge
K: 2
V: C
K: 4
V: Q
K: 8
V: W
SSTable-(1, 2, 3)

Asynchronous Snapshots

- Triggering (on marker) **flushes memtable**
- Iterator restricts access only on current **SSTables.** (used to copy snapshot to hdfs)
- Further changes go to memtable (simple).

Incremental Snapshots

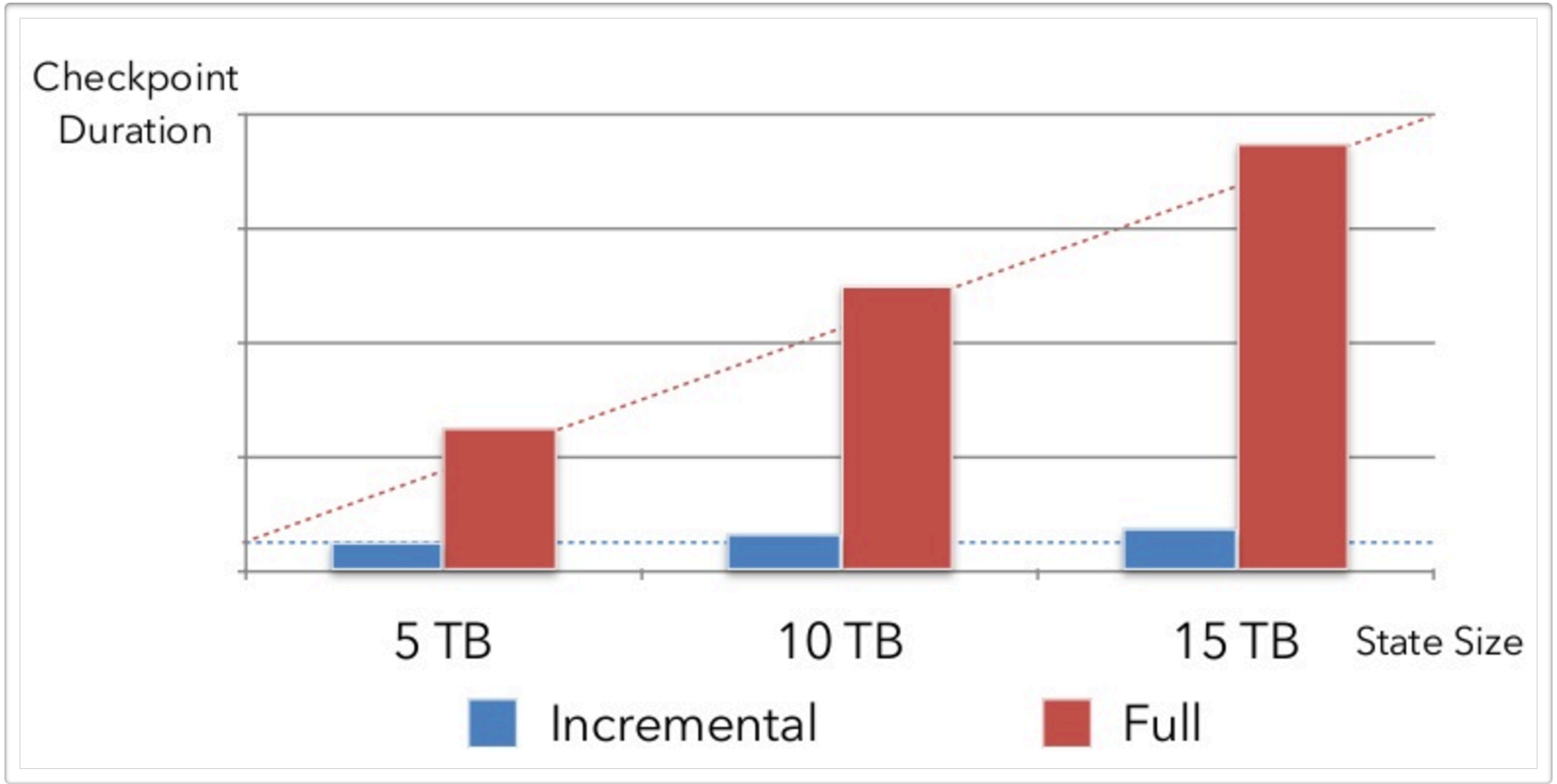


Incremental Snapshots

- Memtable == deltas, by definition.
- Triggering (on marker) **flushes memtable.**
- Copy **only new** (sstable) files to **hdfs.**
- Add **reference counting** for sstable files.

Asynchronously combine incremental snapshots to derive full snapshot (faster for reconfiguration)

Incremental Snapshots



Observation

- Triggering snapshots apparently takes **no execution time.**
- The **time** to **complete** an **epoch** depends on **asynchronous copying.**
- Local Snapshotting happens **asynchronously.**
- **Incremental Snapshots** can decrease background copying significantly.

Part III

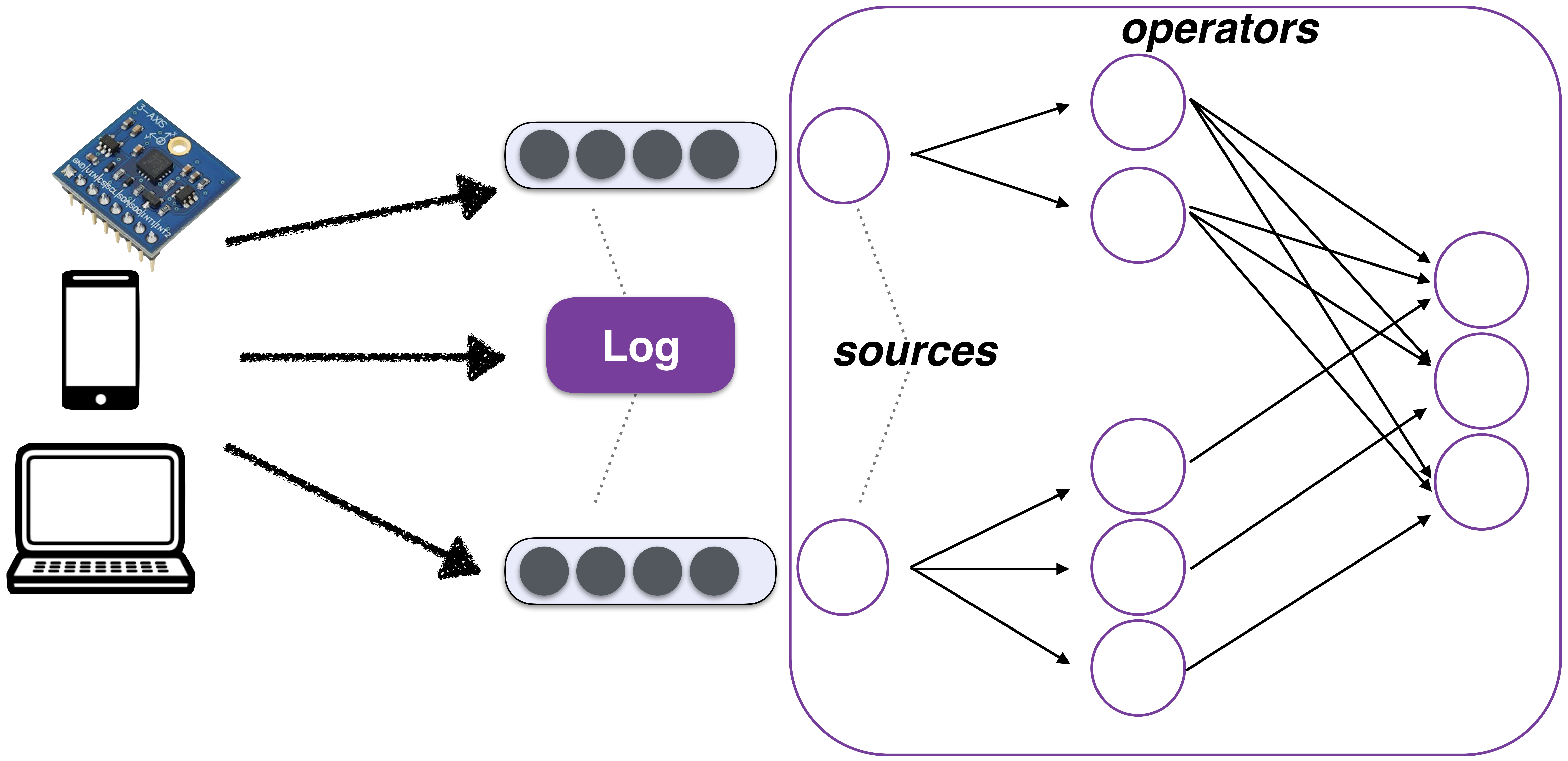
Time and Out-Of-Order

Wow does it work?

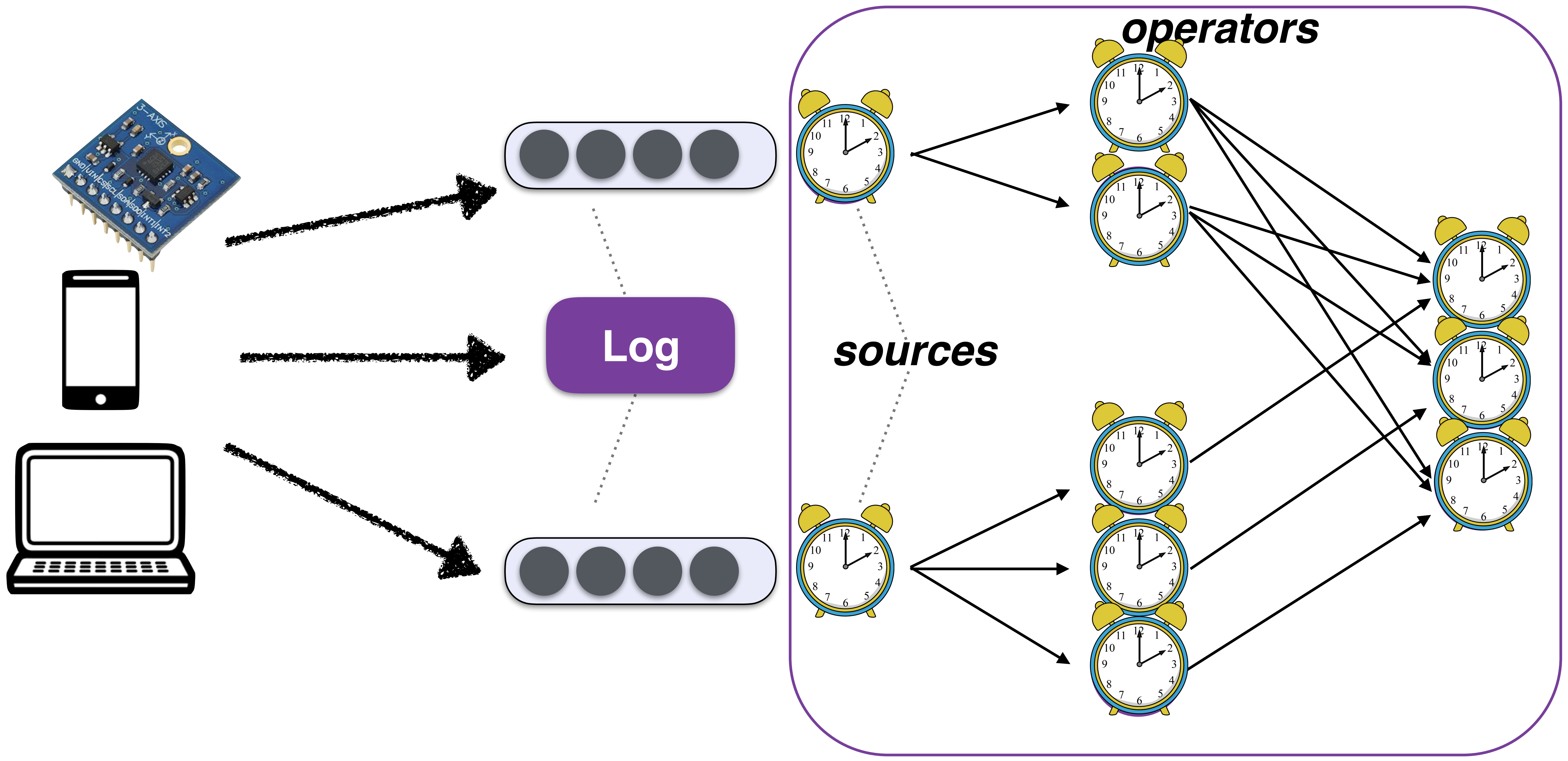
Window
Word Count
(Apache Flink)

```
val windowCounts = text.flatMap { w => w.split("\\s") }
  .map { w => WordWithCount(w, 1) }
  .keyBy("word")
  .timeWindow(Time.seconds(5))
  .sum("count")
```

Reasoning about Time

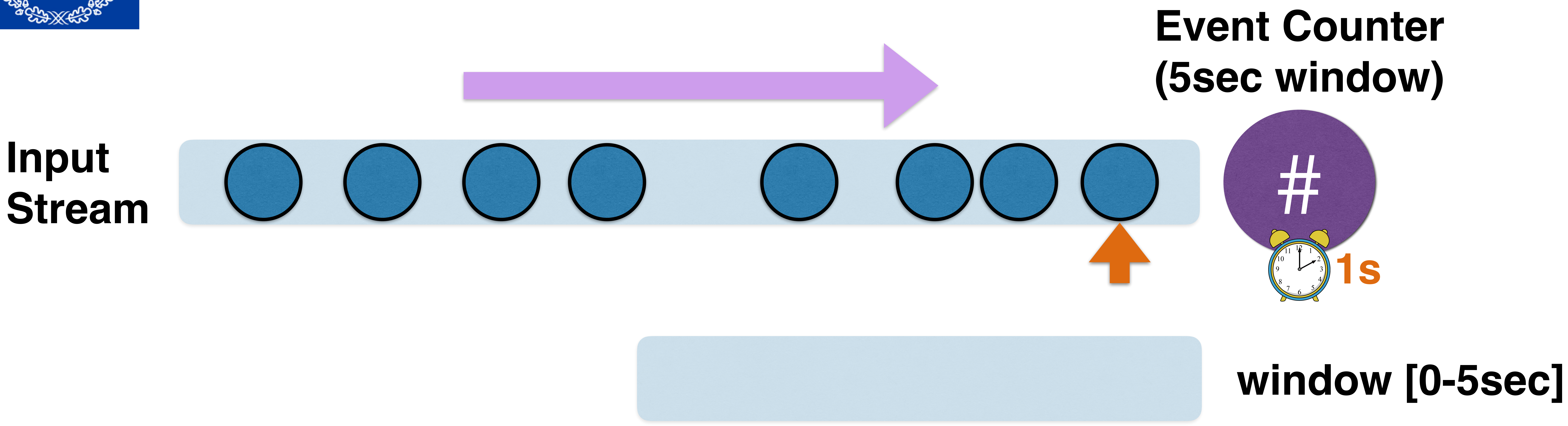


Reasoning about Time

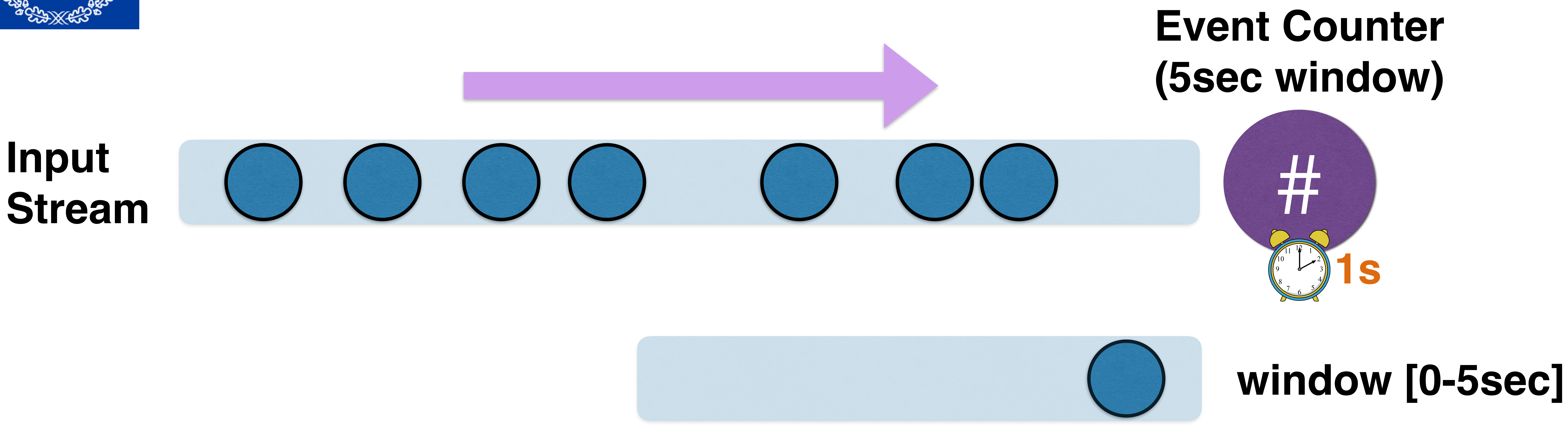


Every Task has a clock

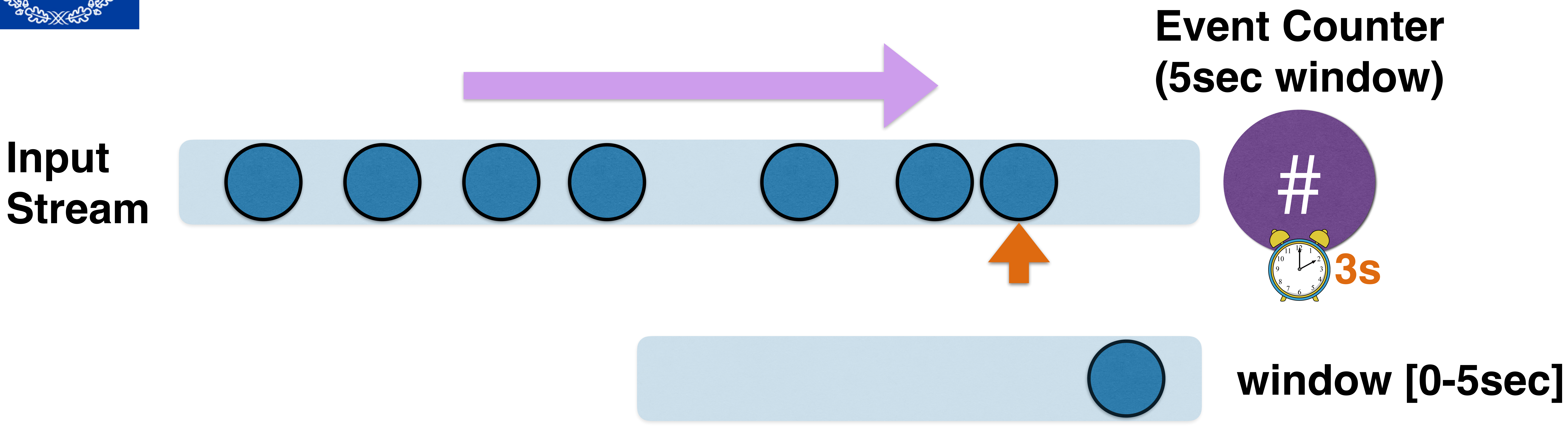
Processing Time Example



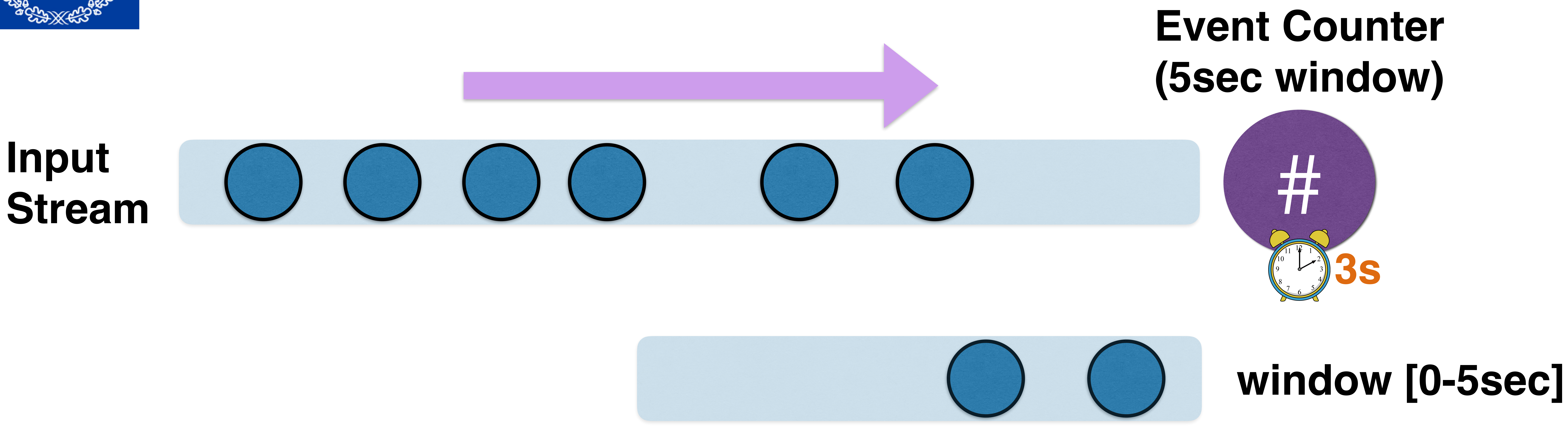
Processing Time Example



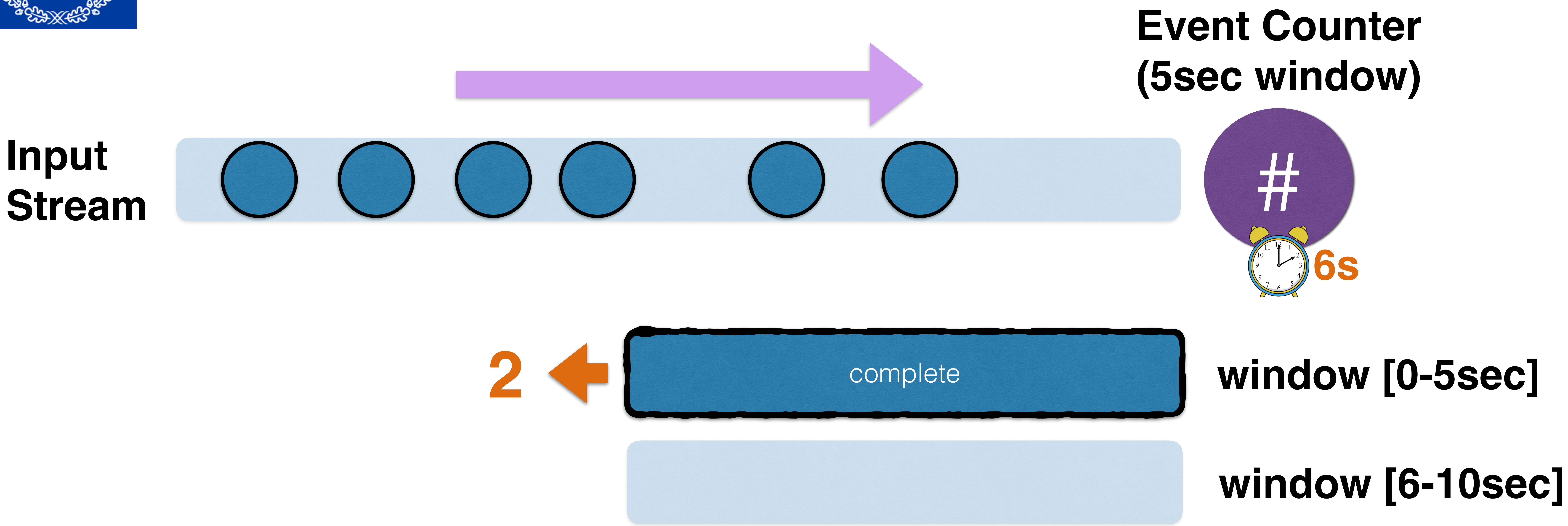
Processing Time Example



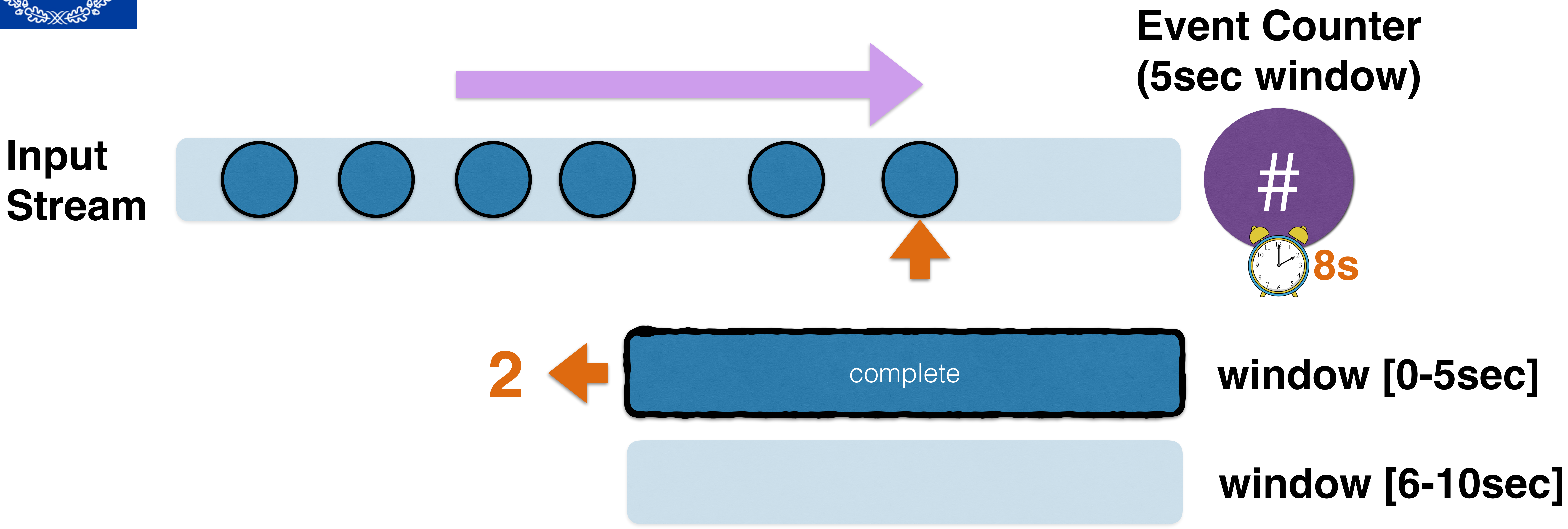
Processing Time Example



Processing Time Example

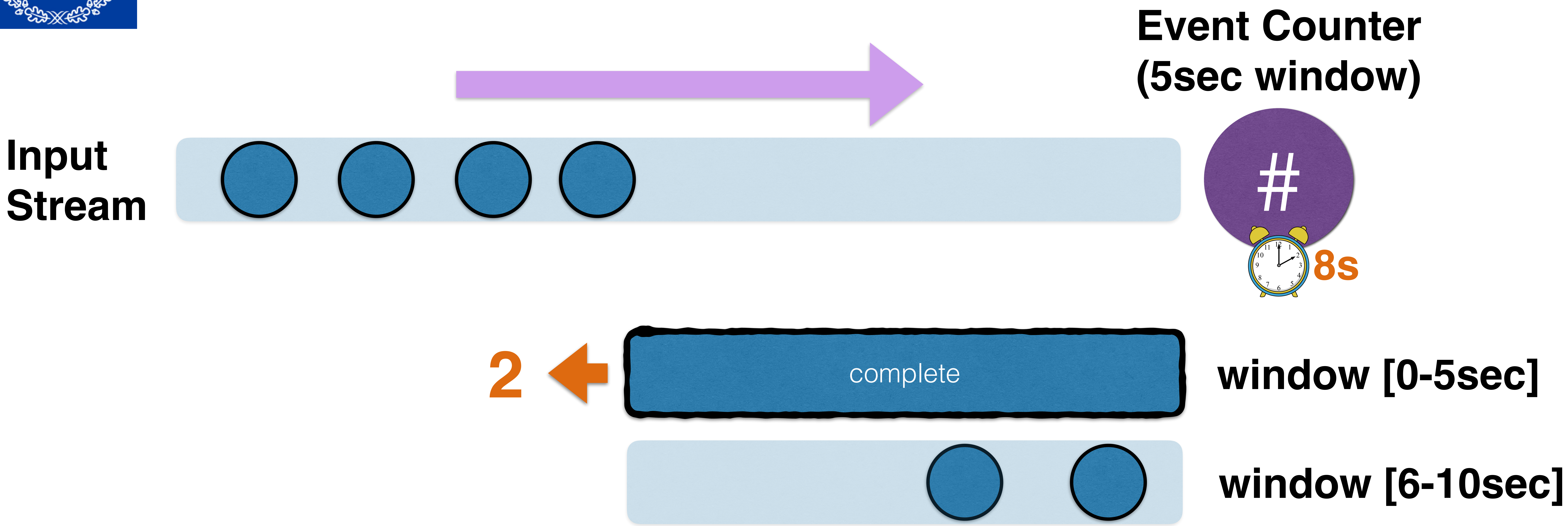


Processing Time Example

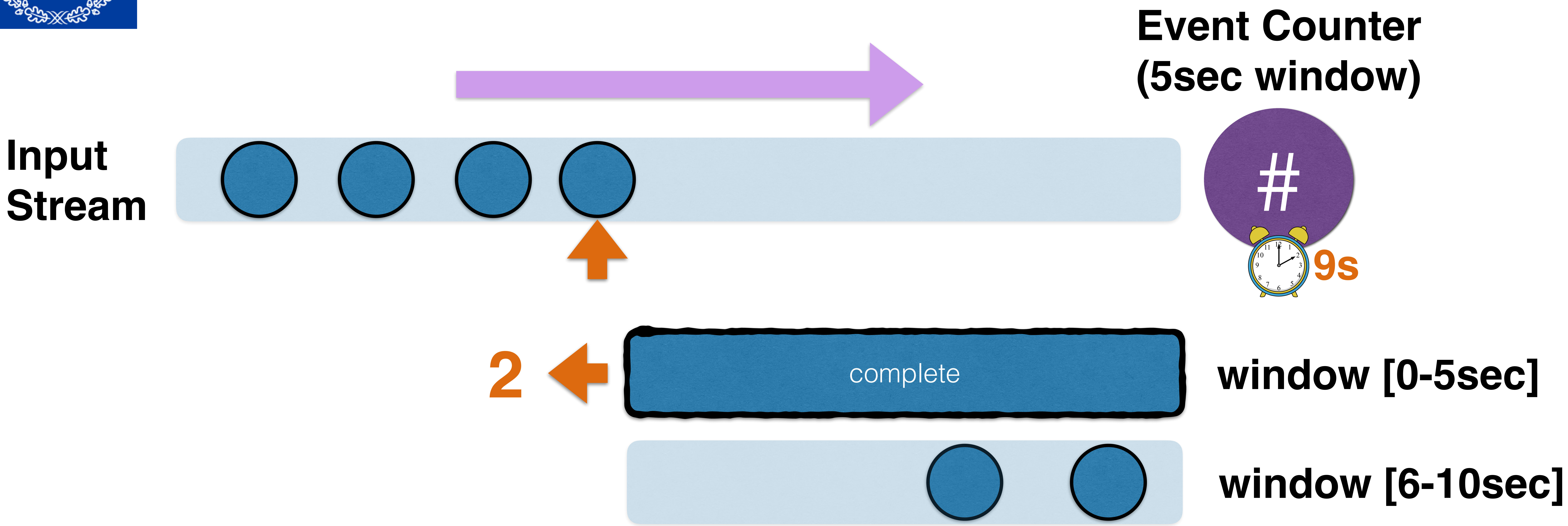


Processing Time Example

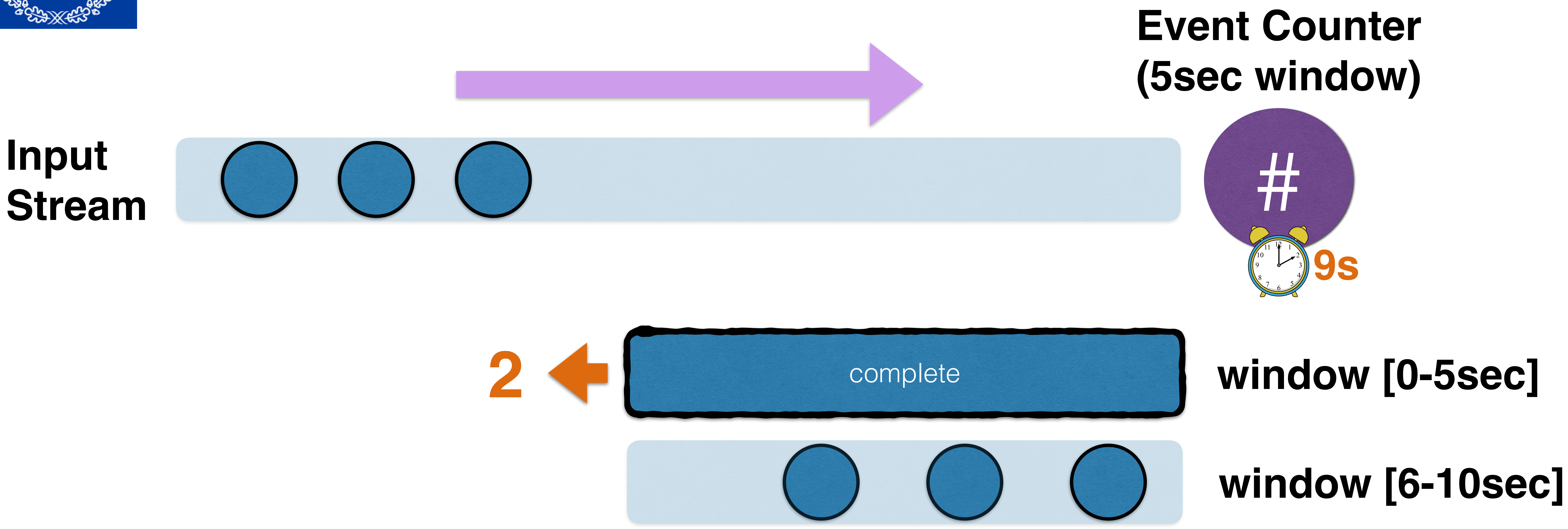
Processing Time Example



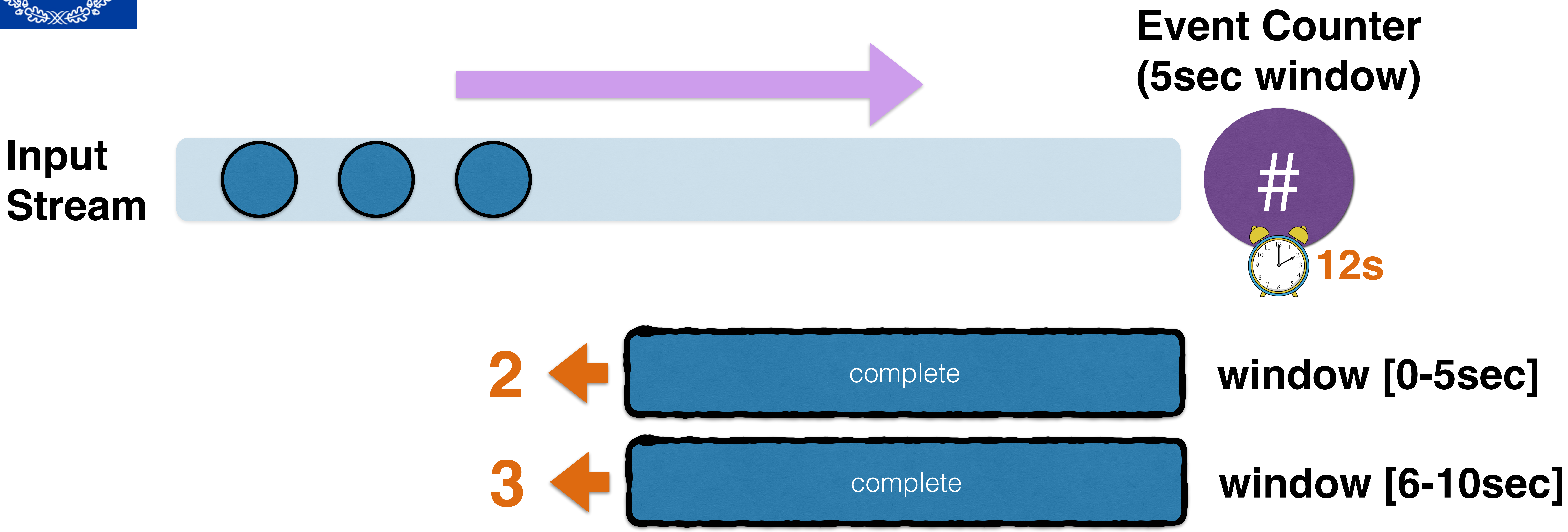
Processing Time Example



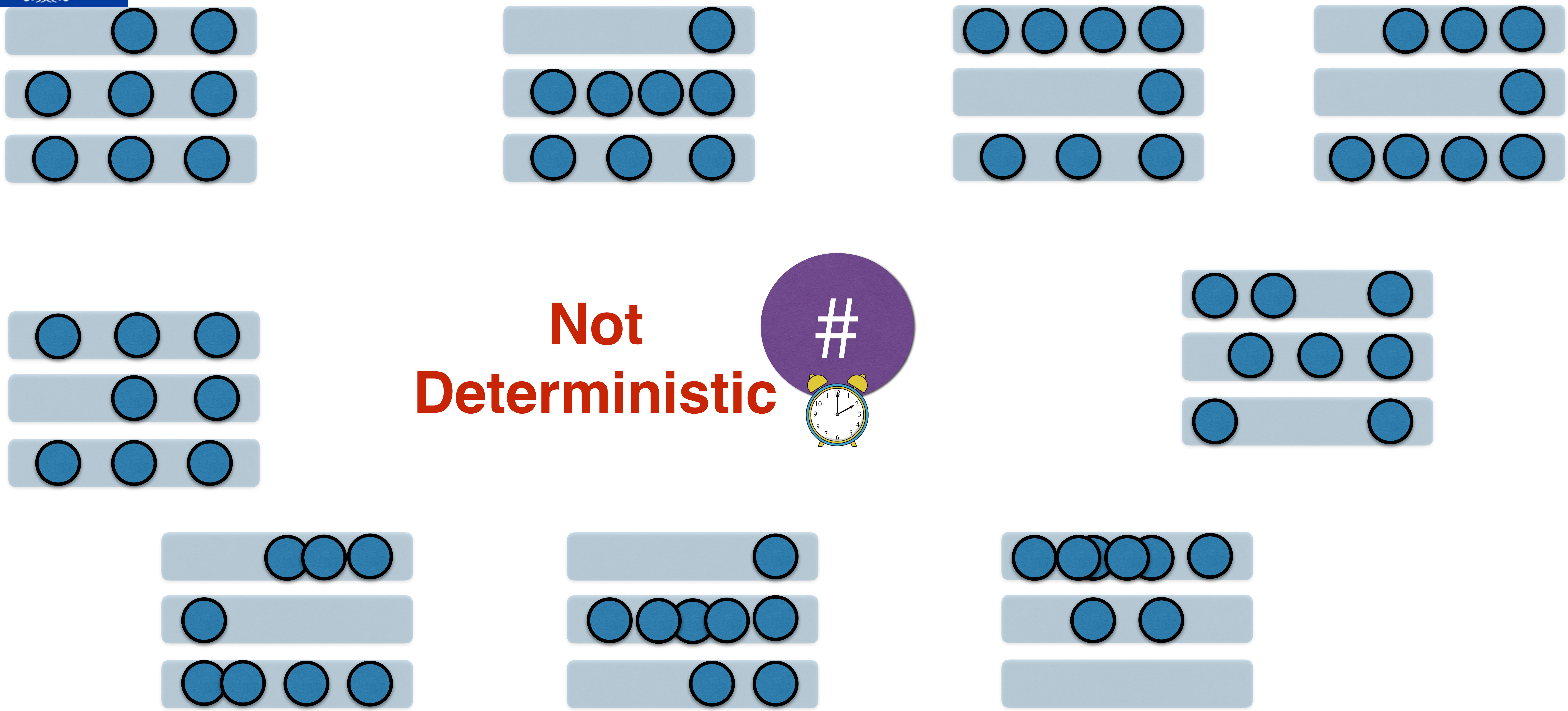
Processing Time Example



Processing Time Example

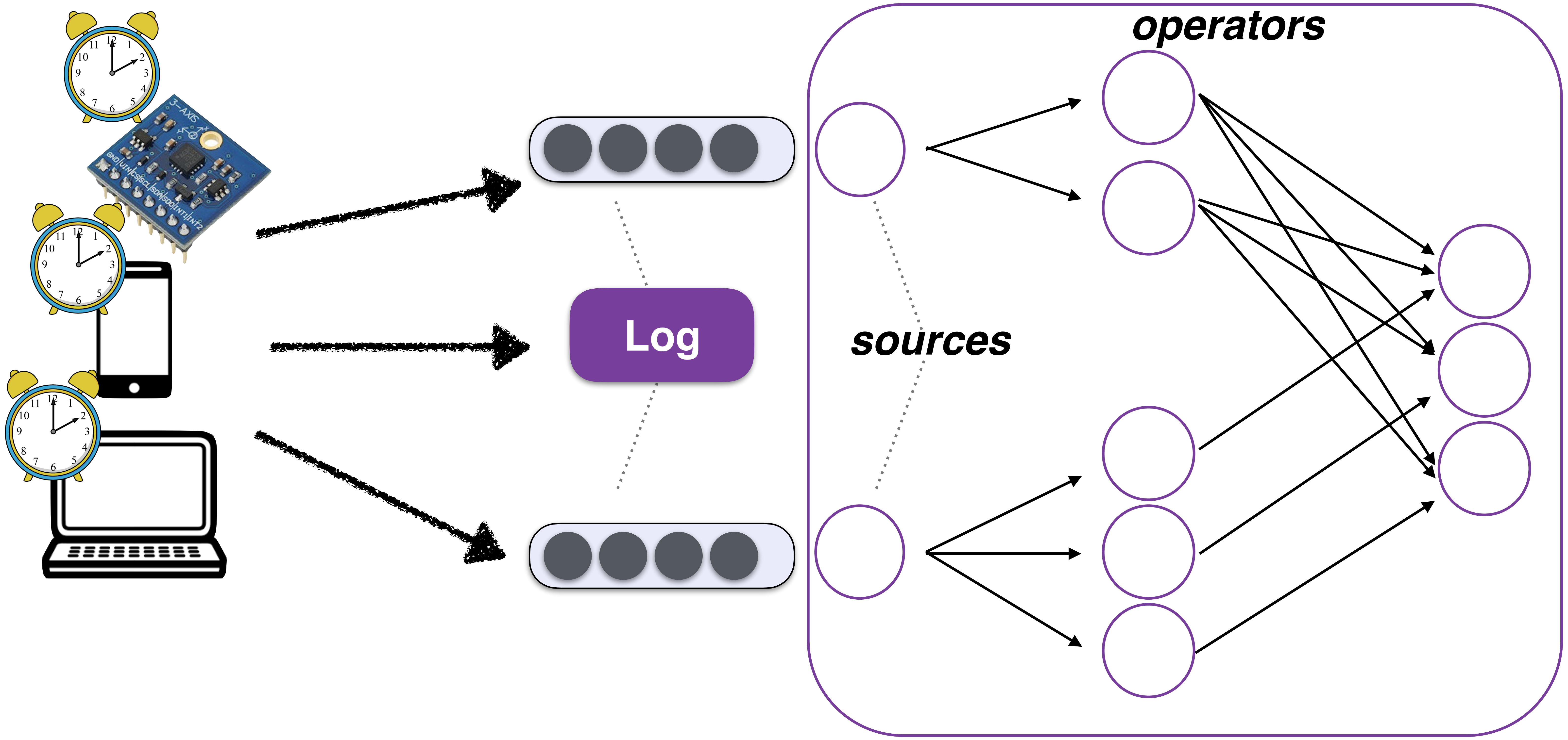


Processing Time : The Issue

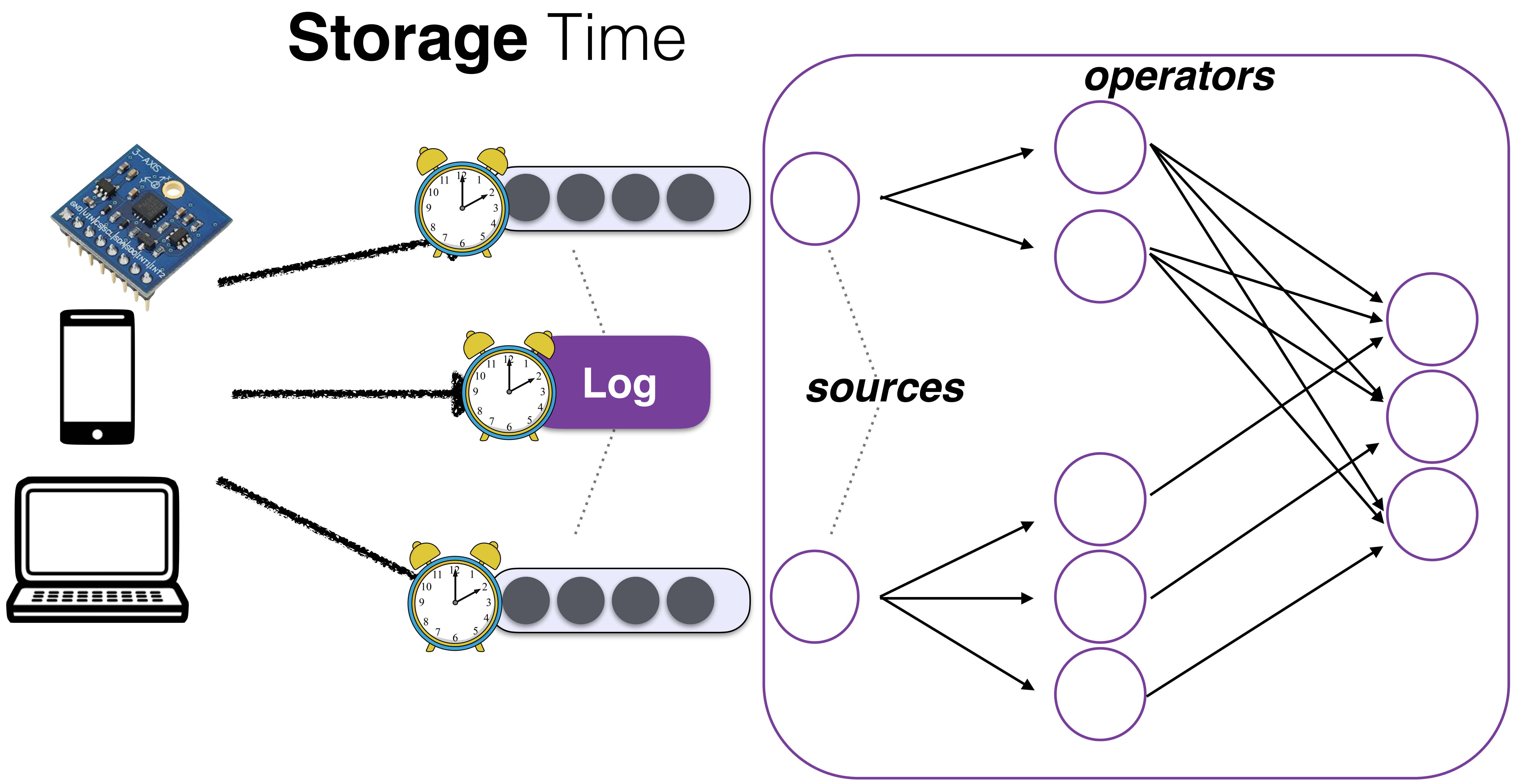


Event Time

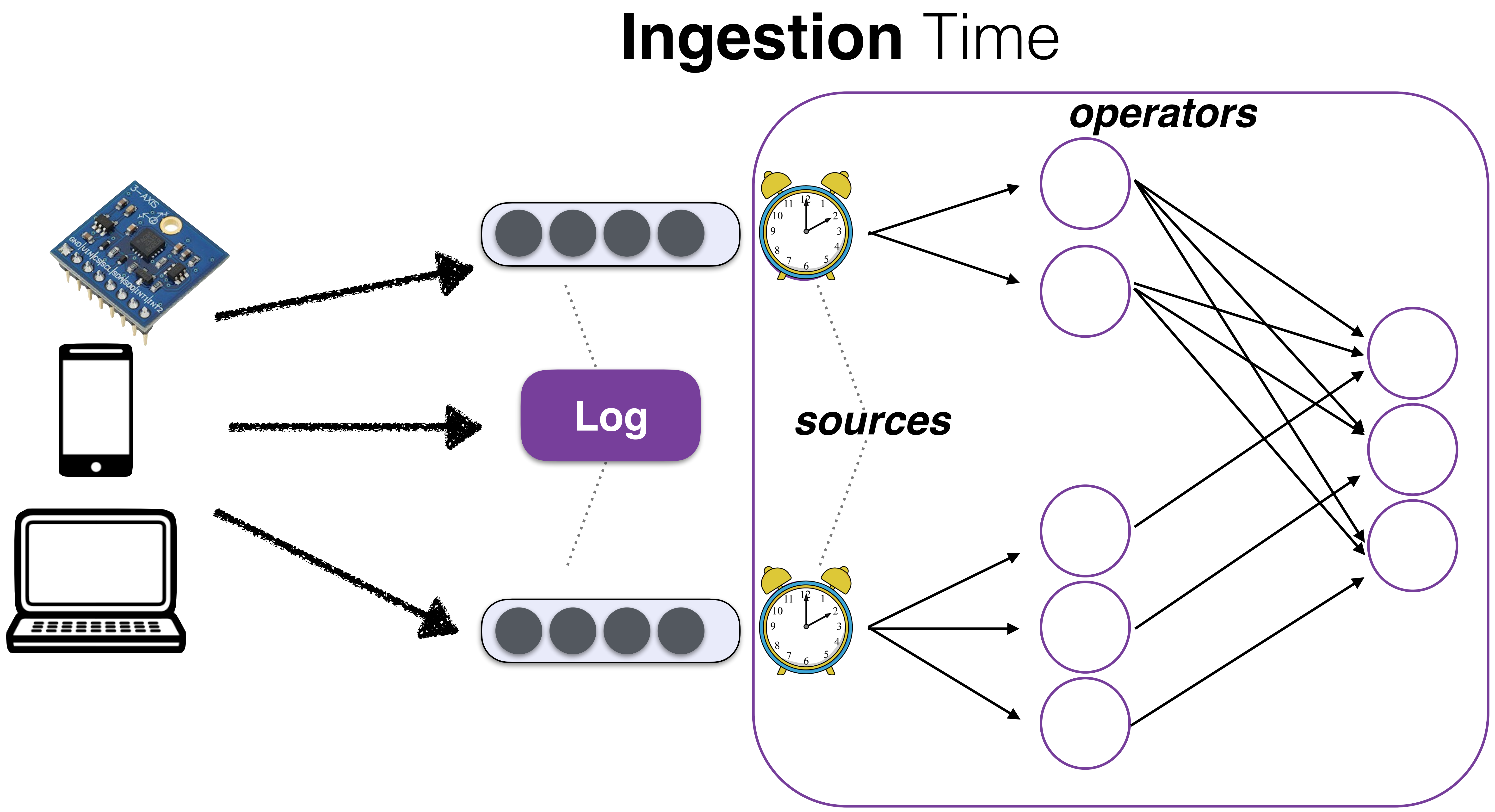
Origin Time



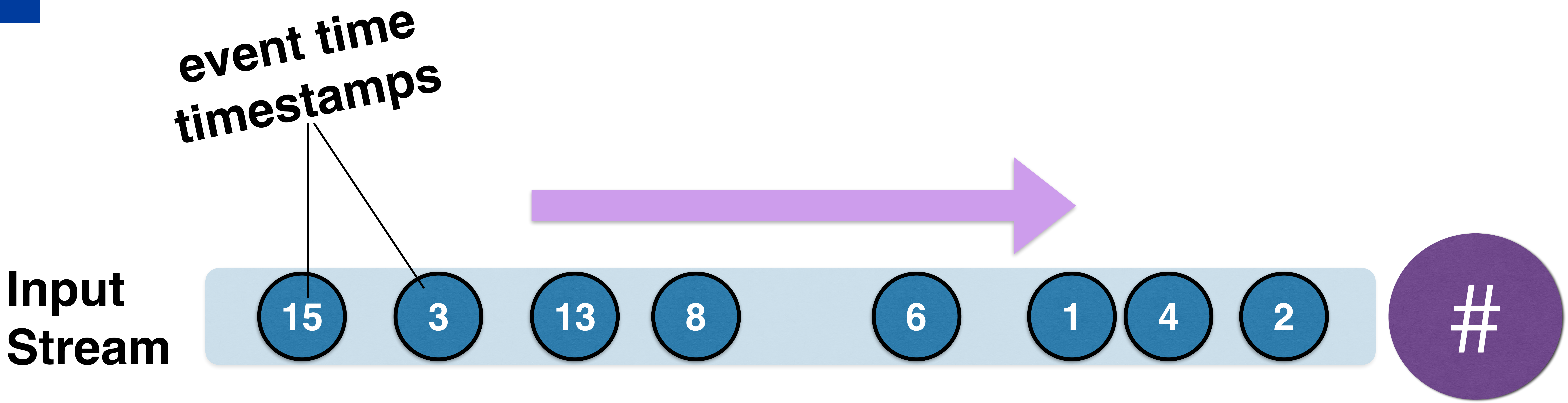
Event Time



Event Time



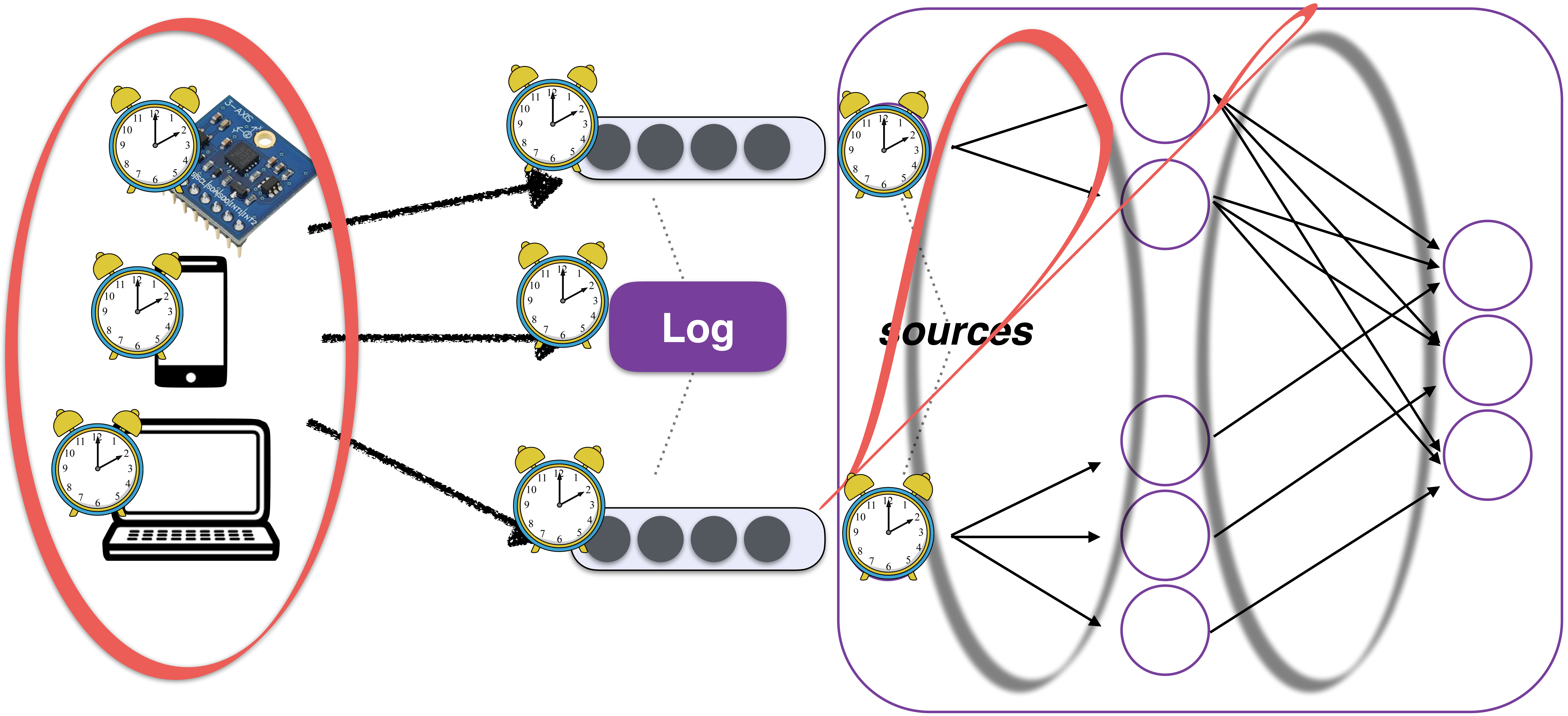
Event Time

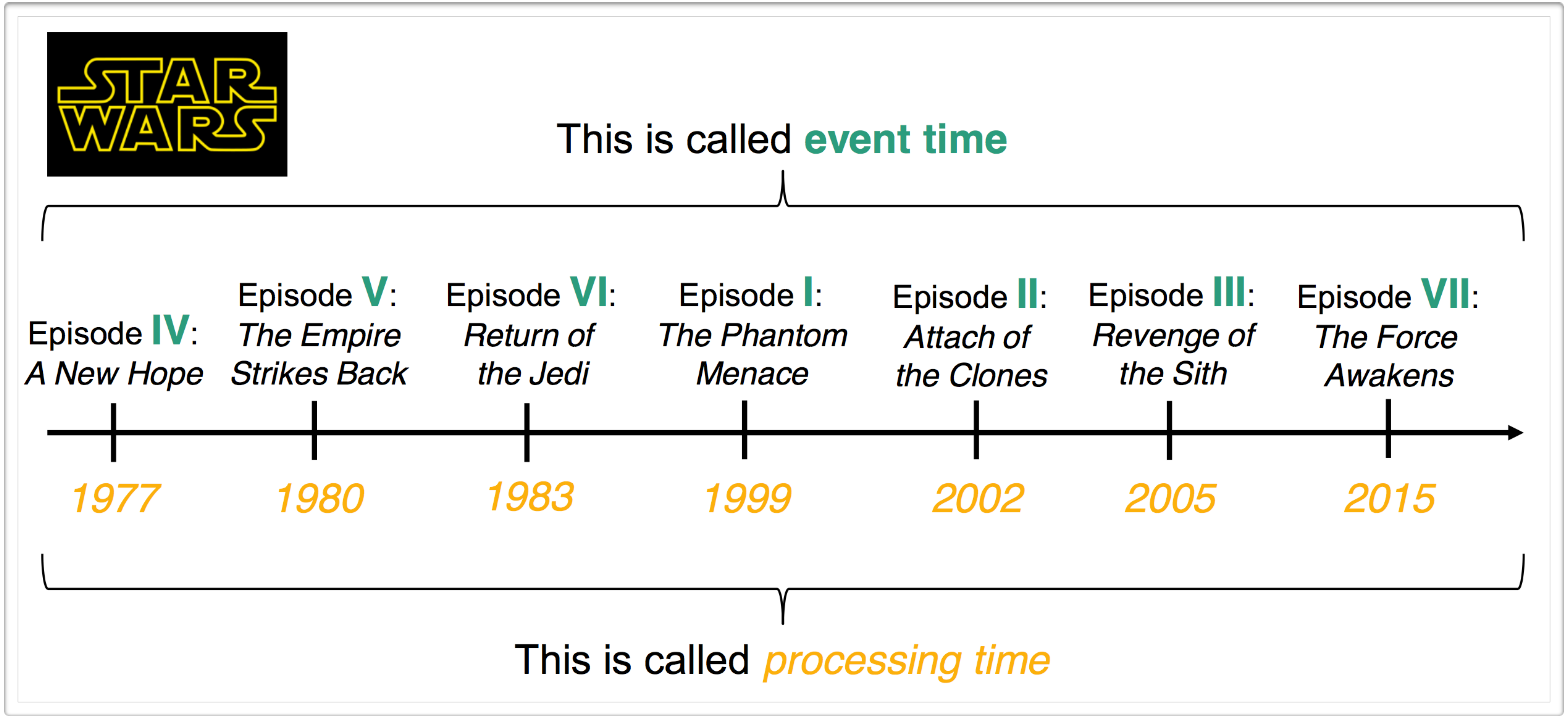


- **Problem**: Distributed events arrive **out of order**

Out-of-Orderness is unavoidable

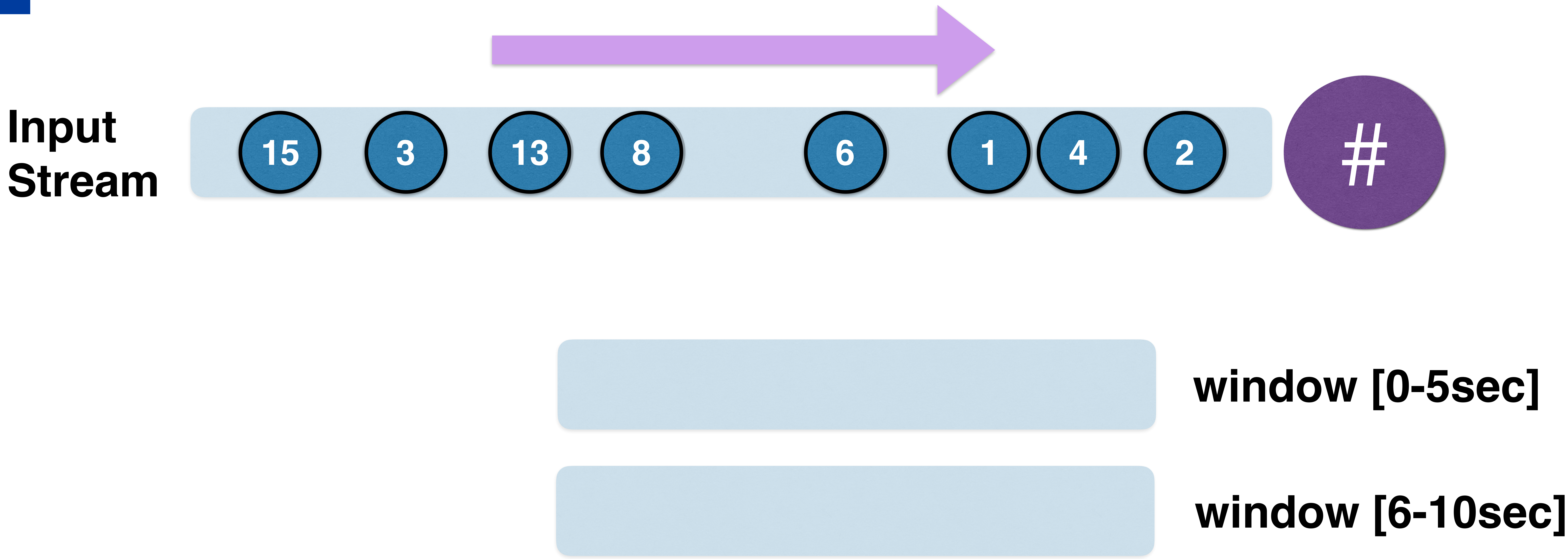
- Origin Devices can **disconnect** temporarily (e.g., train tunnels).
- There is **interleaving** both in message logs (kafka) and on **shuffles** between PEs.



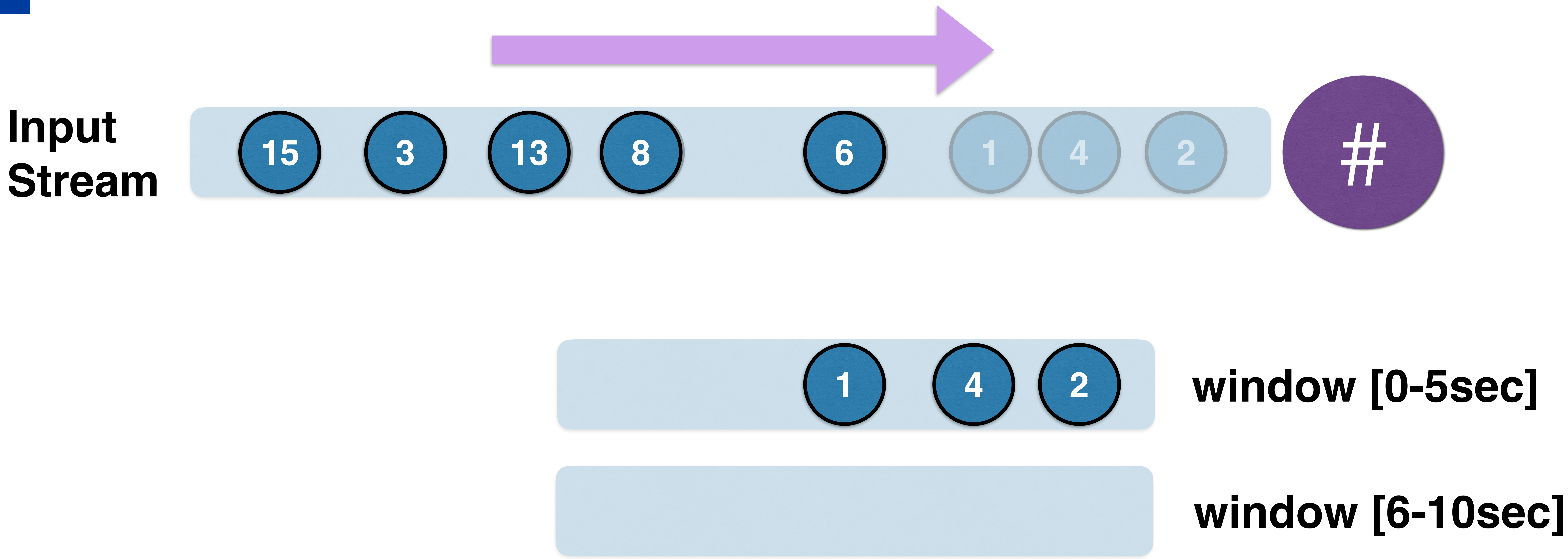


slide by Stephan Ewen & Kostas Tzoumas

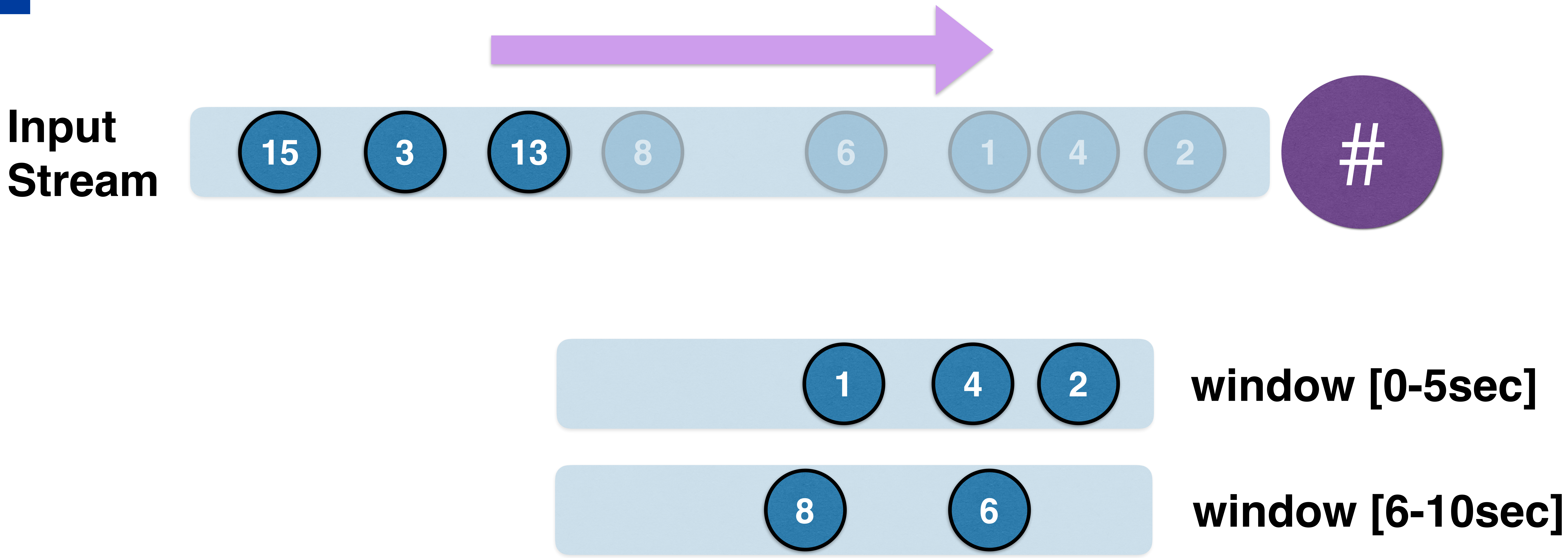
Event Time



Event Time



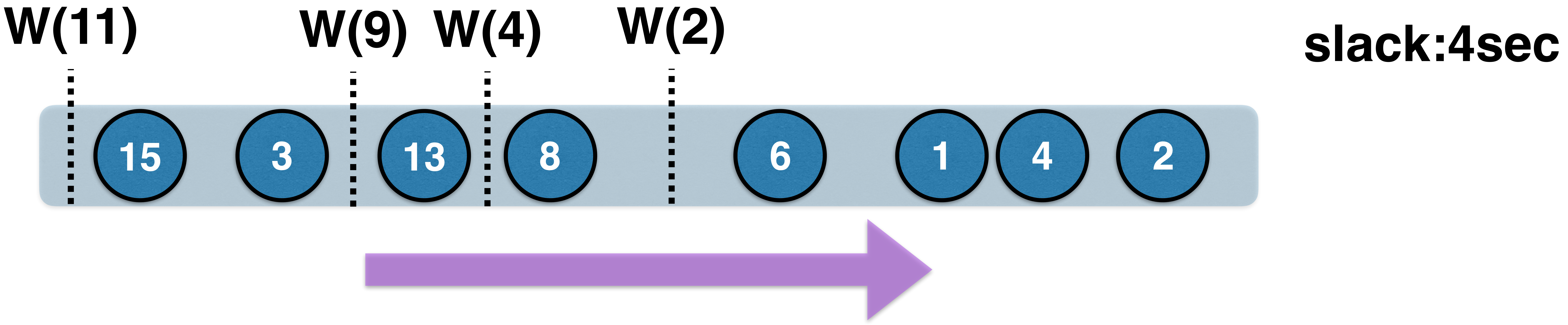
Event Time



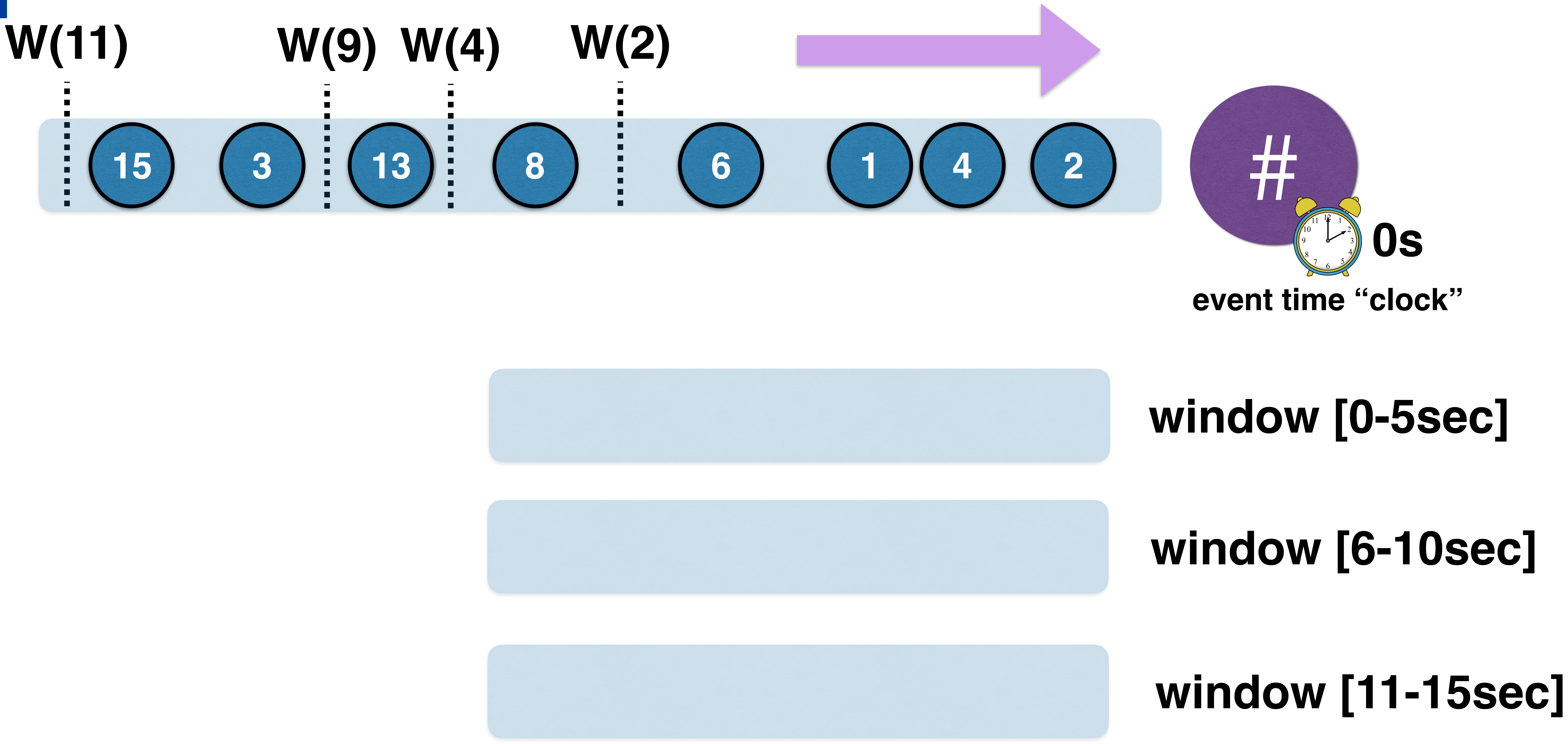
- **Problem**: How do we know when a window is complete?

Solution

- We define a **slack: bound how long** to wait for late events .
- **Low Watermarks:** system-generated events that indicate lowest expected timestamp (using the slack).

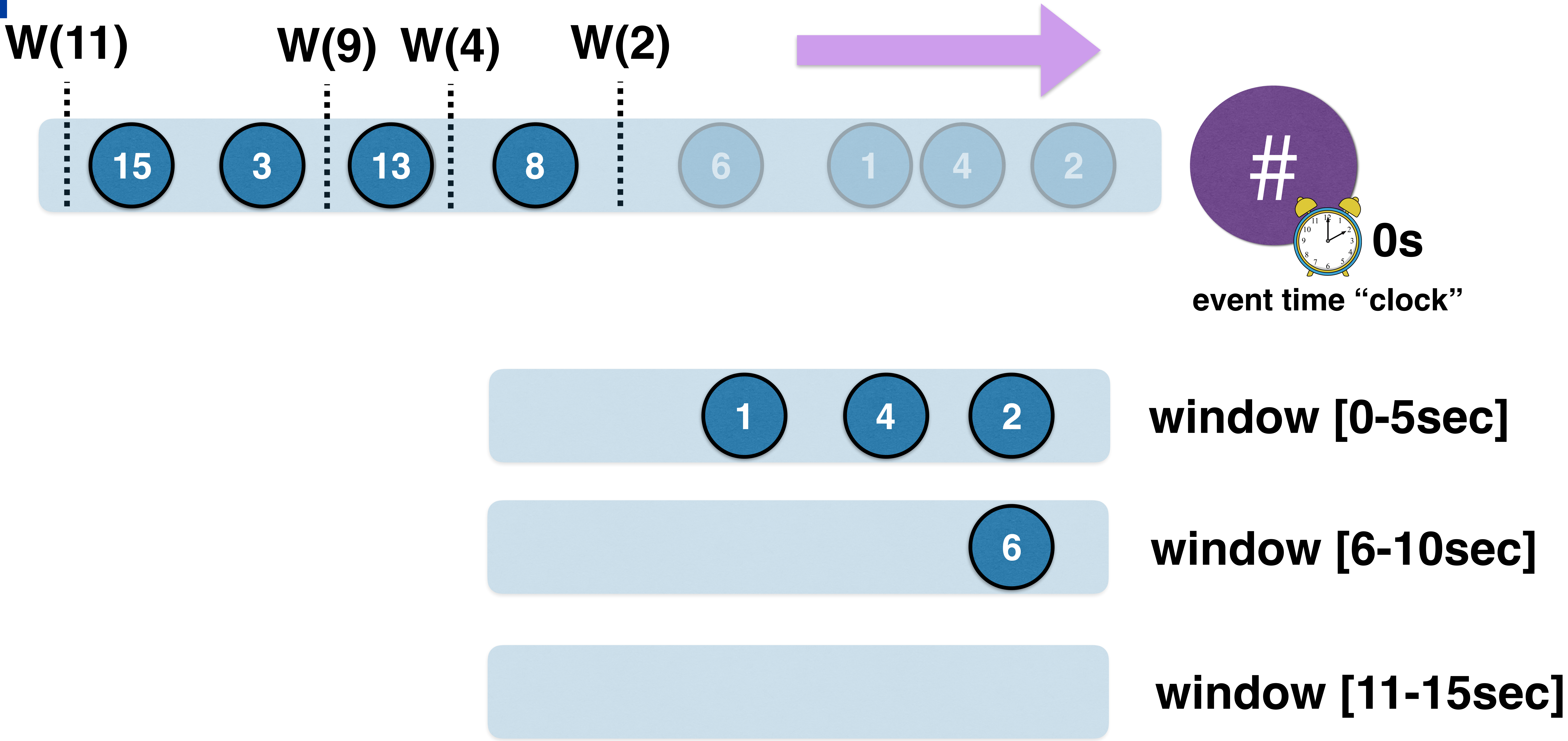


Example

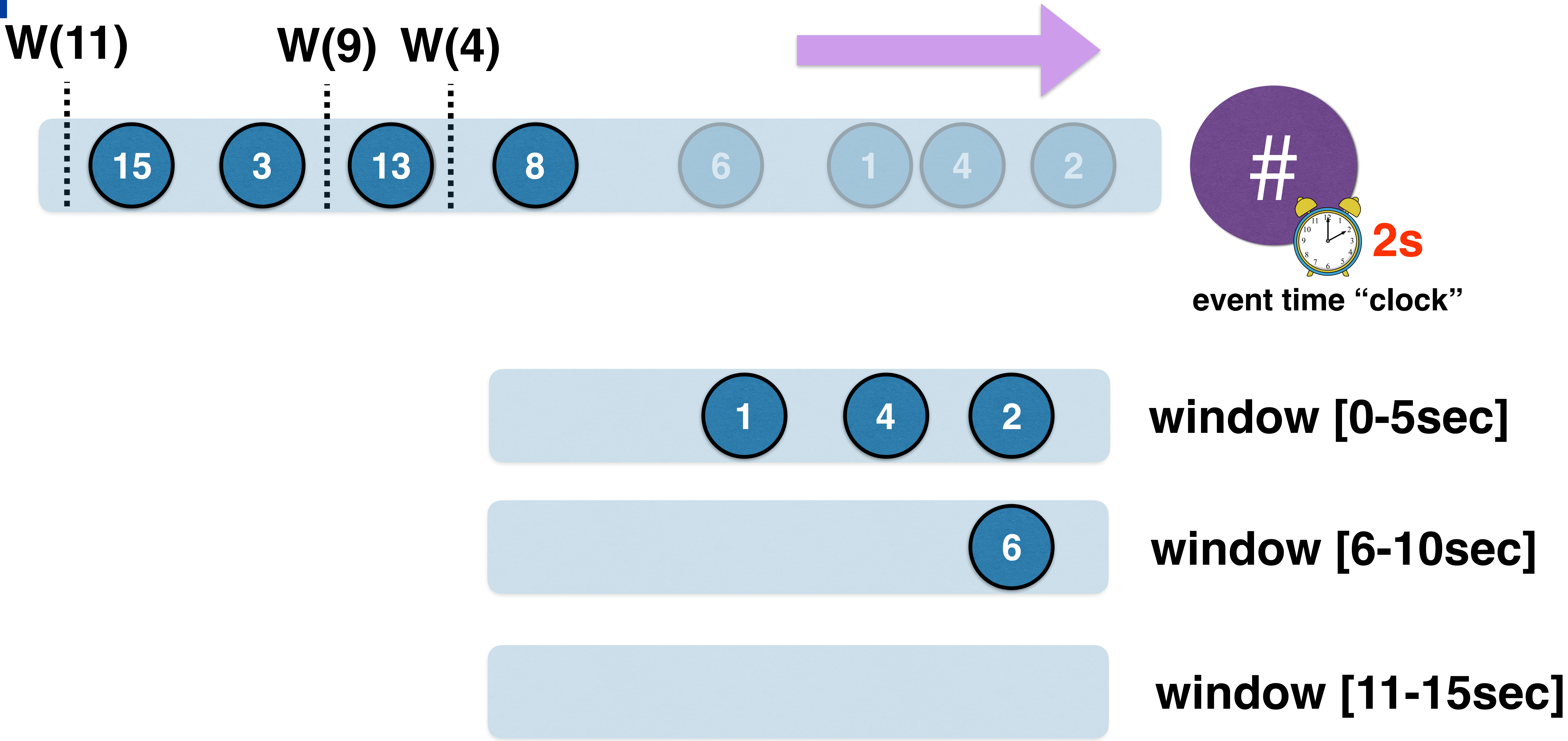


W(11)
W(9)
W(4)
W(2)
15
3
13
8
6
1
4
2
#
0s
event time “clock”
window [0-5sec]
window [6-10sec]
window [11-15sec]

Example

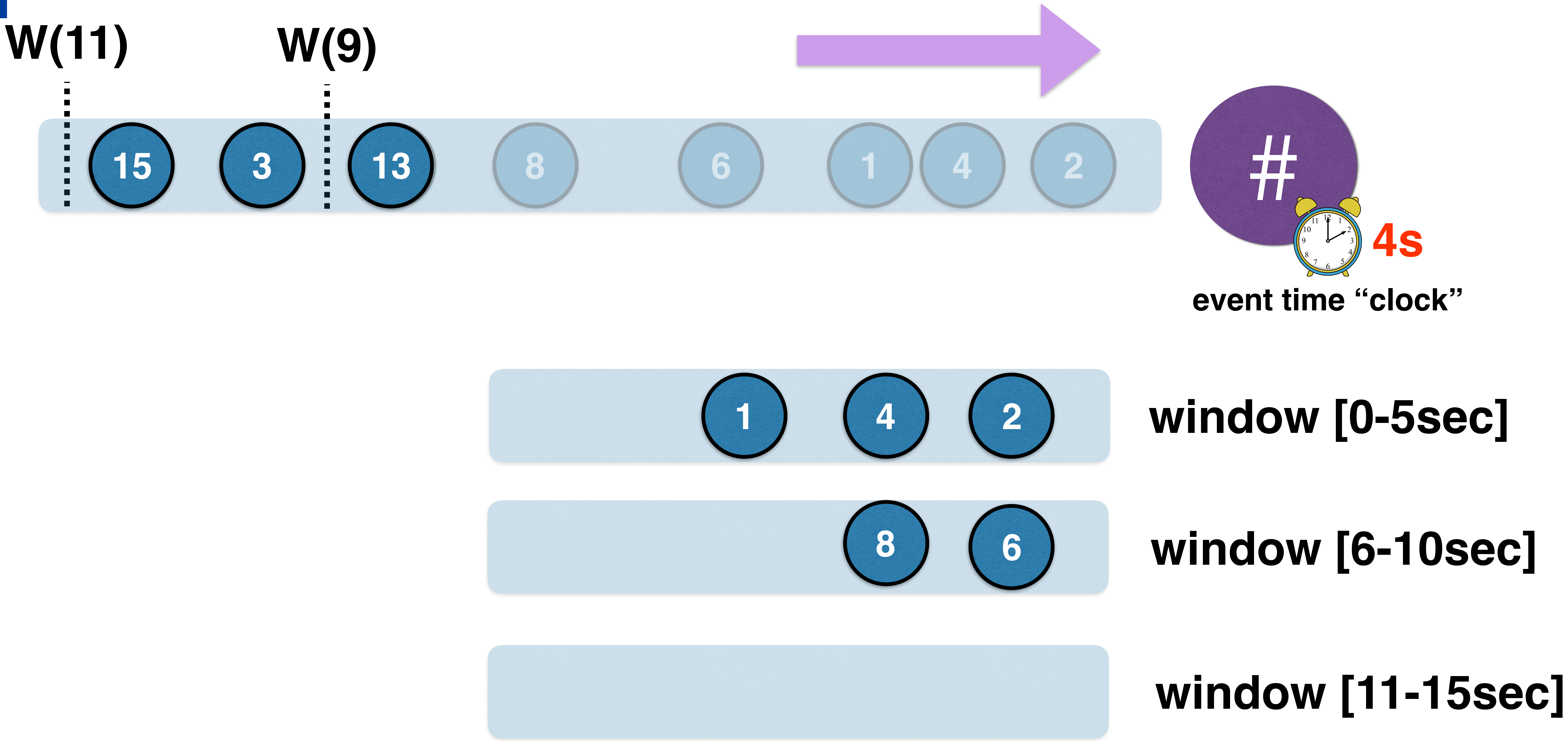

W(11)
W(9)
W(4)
W(2)
15
3
13
8
6
1
4
2
#
0s
event time “clock”
1
4
2
window [0-5sec]
6
window [6-10sec]
window [11-15sec]

Example



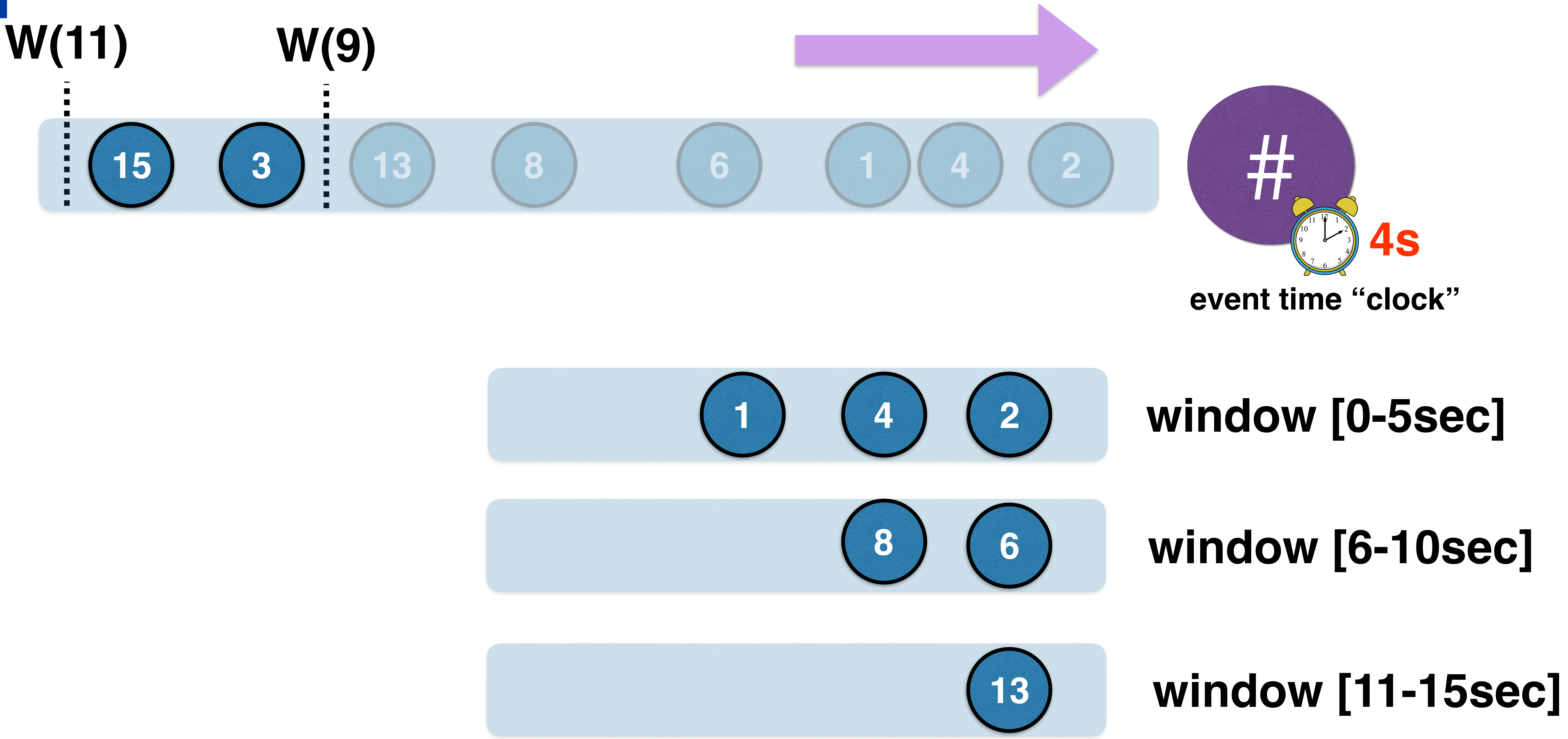
W(11)
W(9)
W(4)
15
3
13
8
6
1
4
2
#
2s
event time “clock”
1
4
2
window [0-5sec]
6
window [6-10sec]
window [11-15sec]

Example

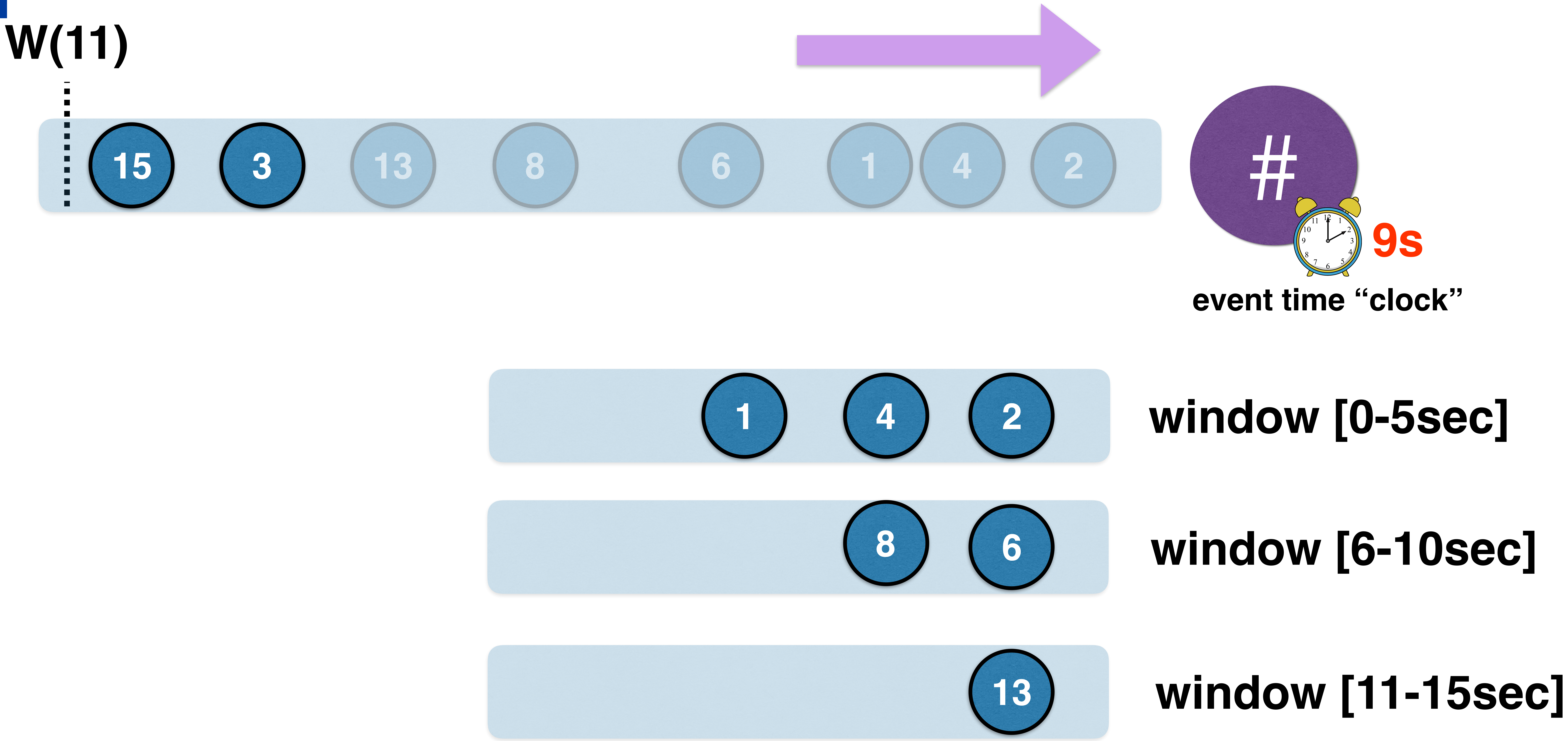


W(11)
W(9)
15
3
13
8
6
1
4
2
#
4s
event time "clock"
1
4
2
window [0-5sec]
8
6
window [6-10sec]
window [11-15sec]

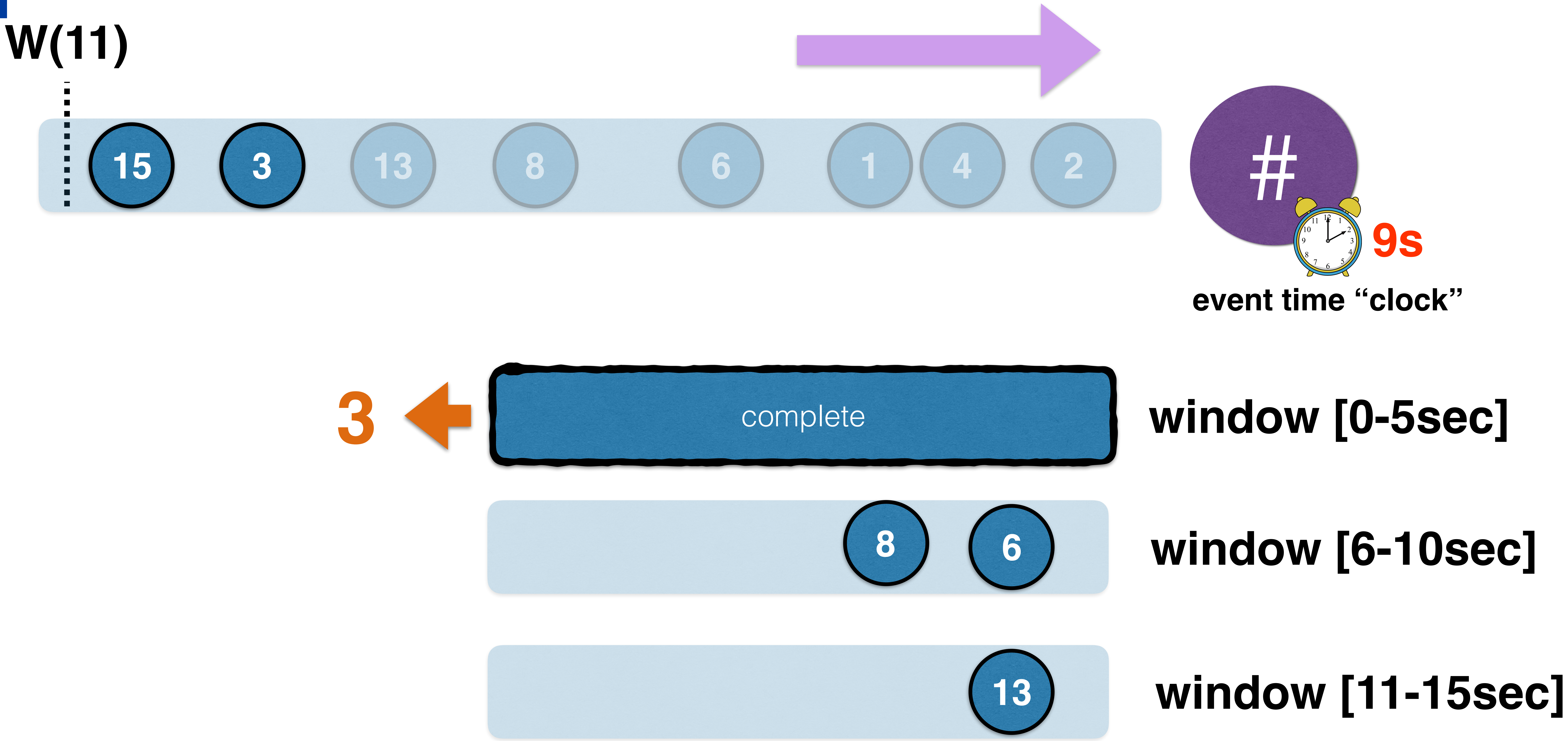
Example


W(11)
W(9)
15
3
13
8
6
1
4
2
#
4s
event time "clock"
1
4
2
window [0-5sec]
8
6
window [6-10sec]
13
window [11-15sec]

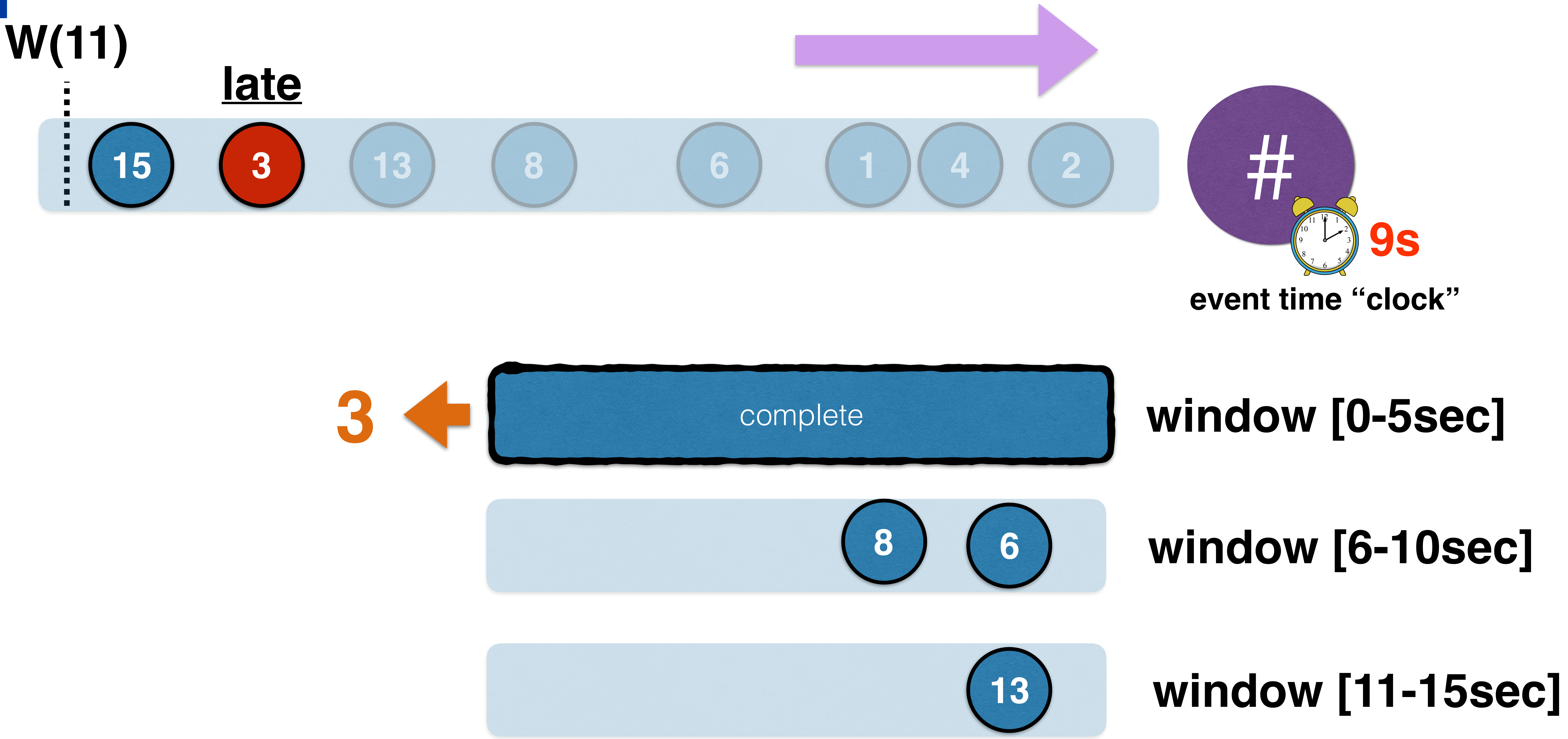
Example


W(11)
15
3
13
8
6
1
4
2
#
9s
event time "clock"
1
4
2
window [0-5sec]
8
6
window [6-10sec]
13
window [11-15sec]

Example


W(11)
15
3
13
8
6
1
4
2
#
9s
event time "clock"
3
complete
window [0-5sec]
8
6
window [6-10sec]
13
window [11-15sec]

Example

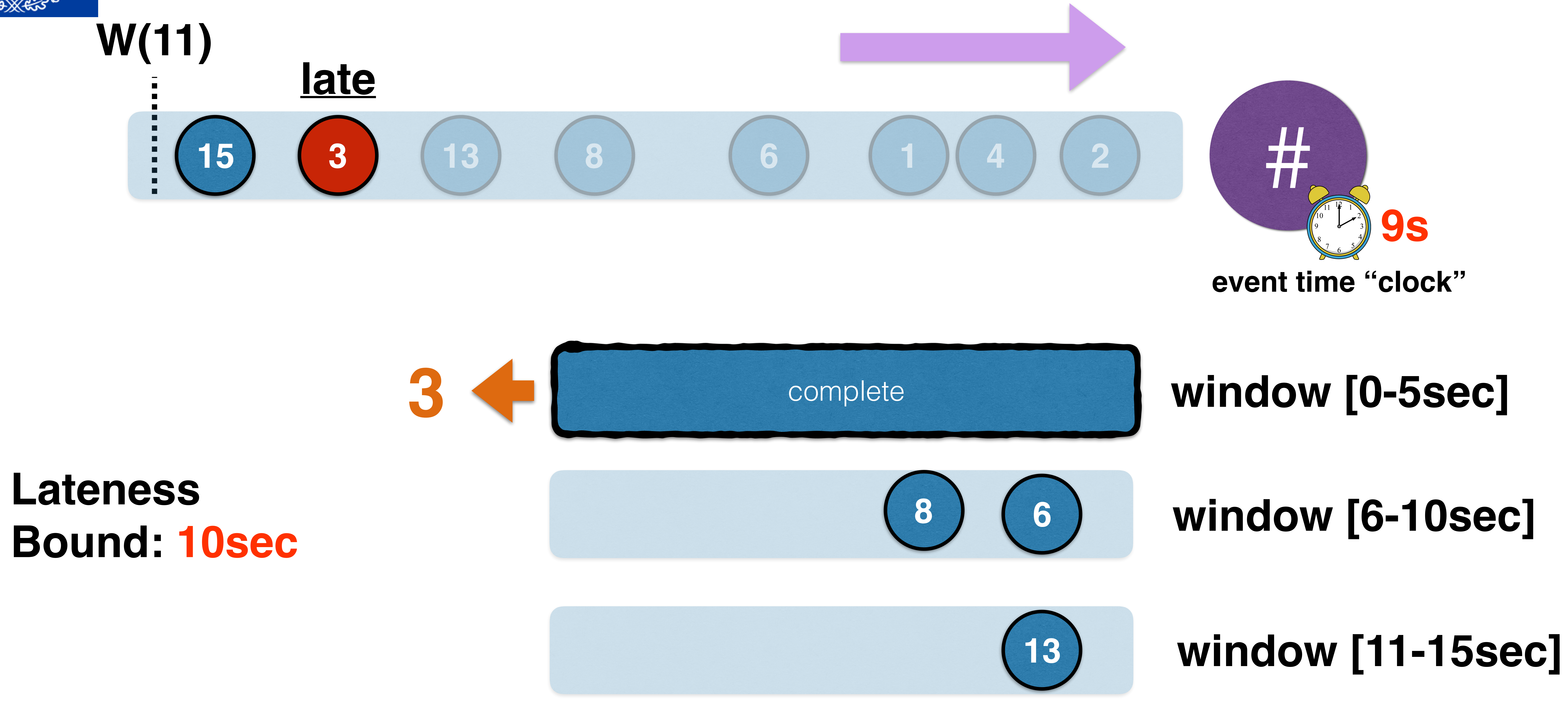


W(11)
late
15
3
13
8
6
1
4
2
#
9s
event time "clock"
3
complete
window [0-5sec]
8
6
window [6-10sec]
13
window [11-15sec]

Late Events?

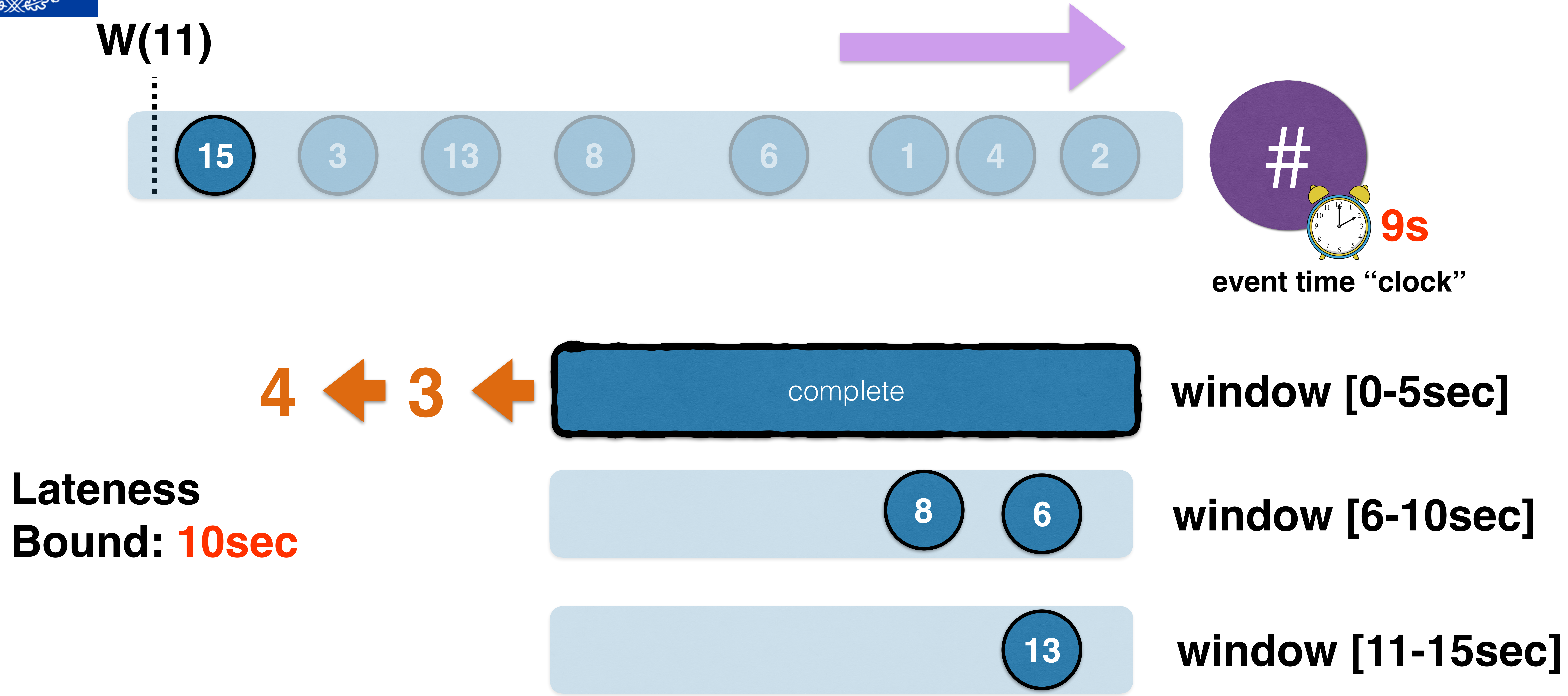
- Allow applications to choose how to handle late events:
  - Drop them
  - **Bound** Lateness and update or.. drop

Example



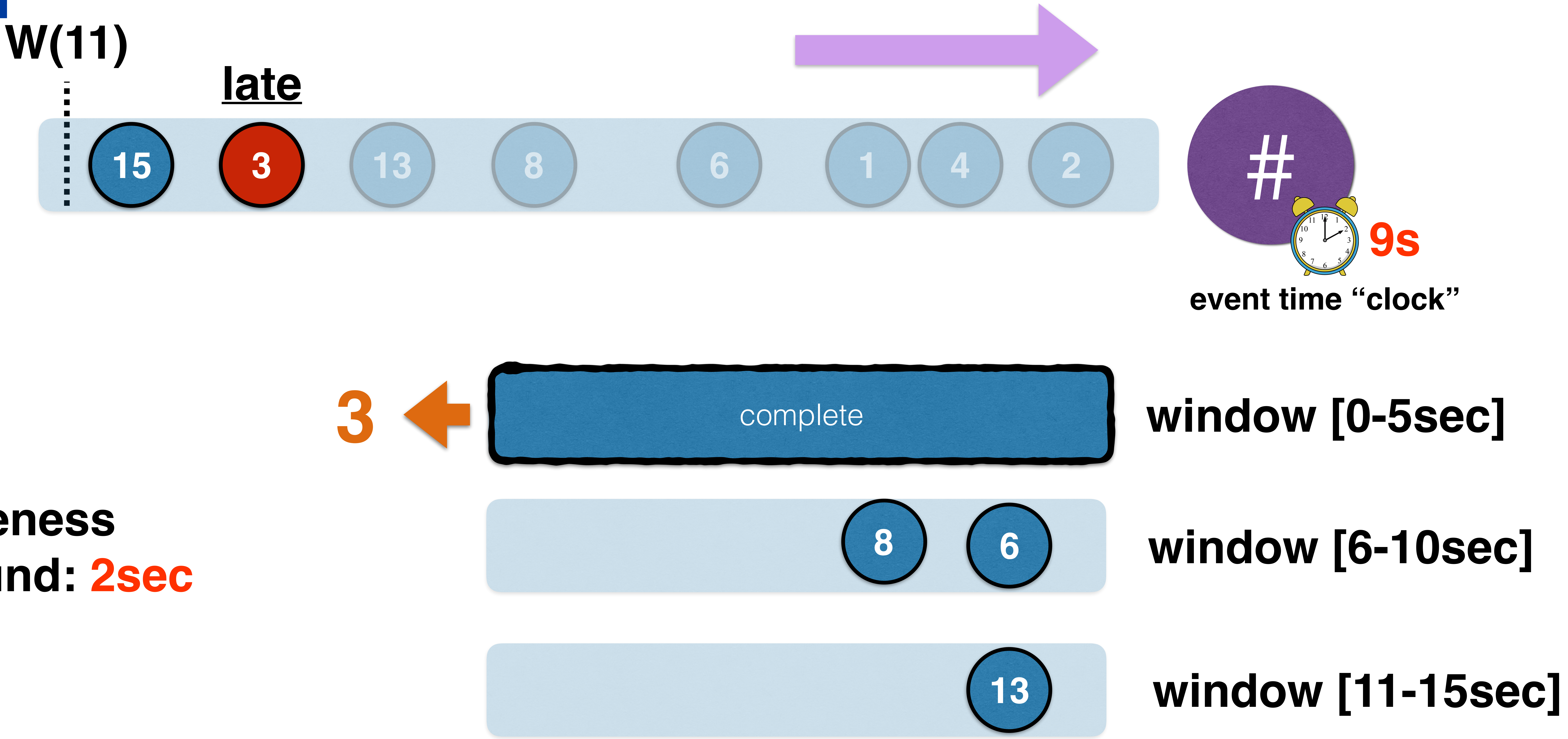
W(11)
late
15
3
13
8
6
1
4
2
#
9s
event time "clock"
3
complete
window [0-5sec]
8
6
window [6-10sec]
13
window [11-15sec]
Lateness
Bound: 10sec

Example



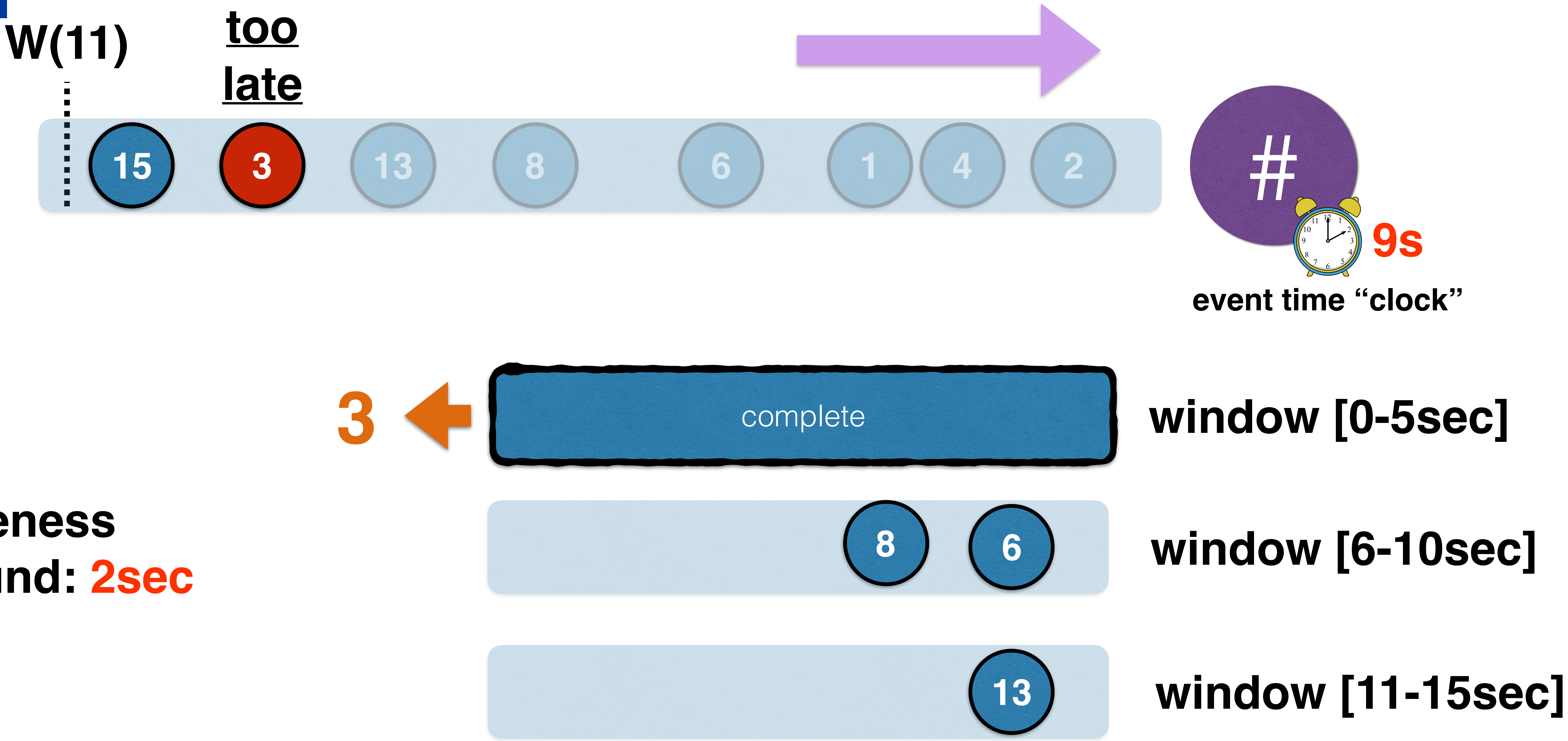
W(11)
15
3
13
8
6
1
4
2
#
9s
event time "clock"
4
3
complete
window [0-5sec]
8
6
window [6-10sec]
13
window [11-15sec]
Lateness
Bound: 10sec

Example



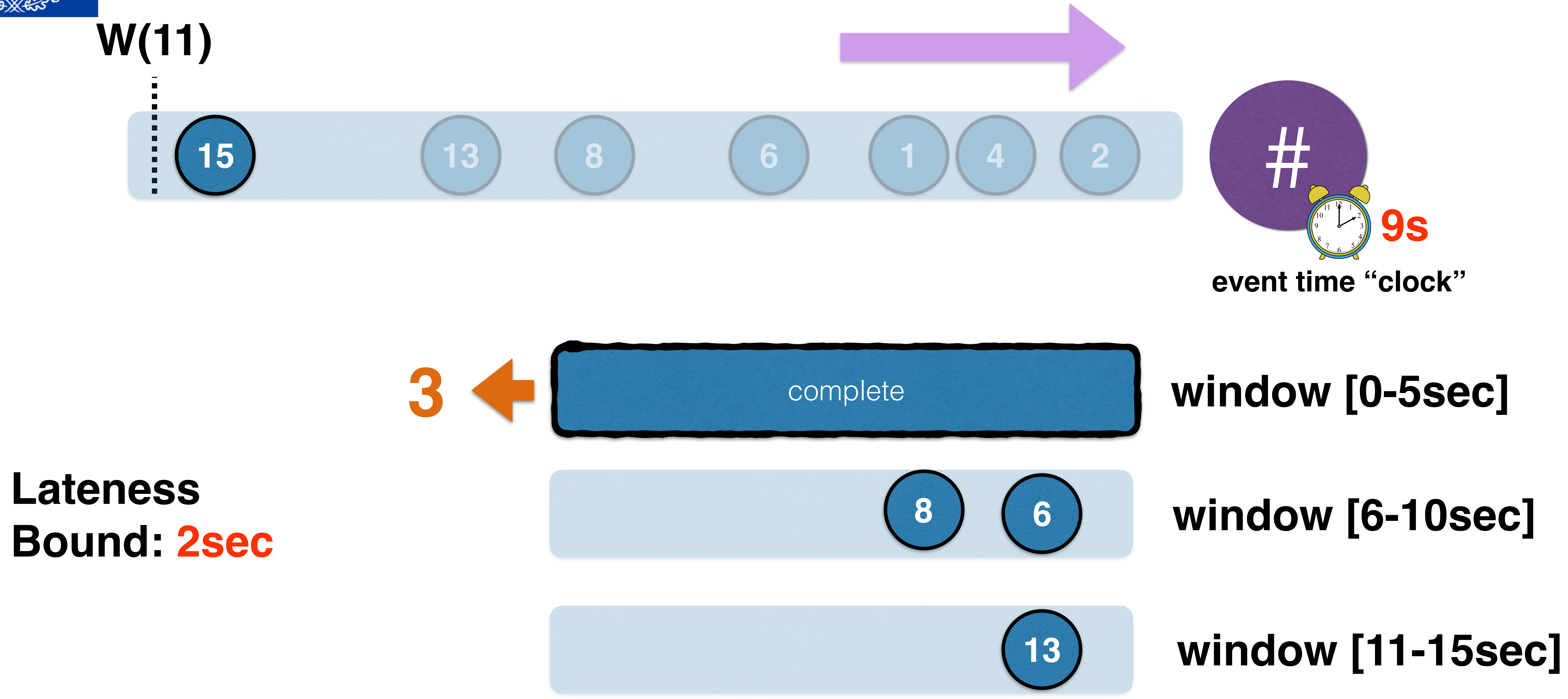
Lateness Bound: 2sec

Example



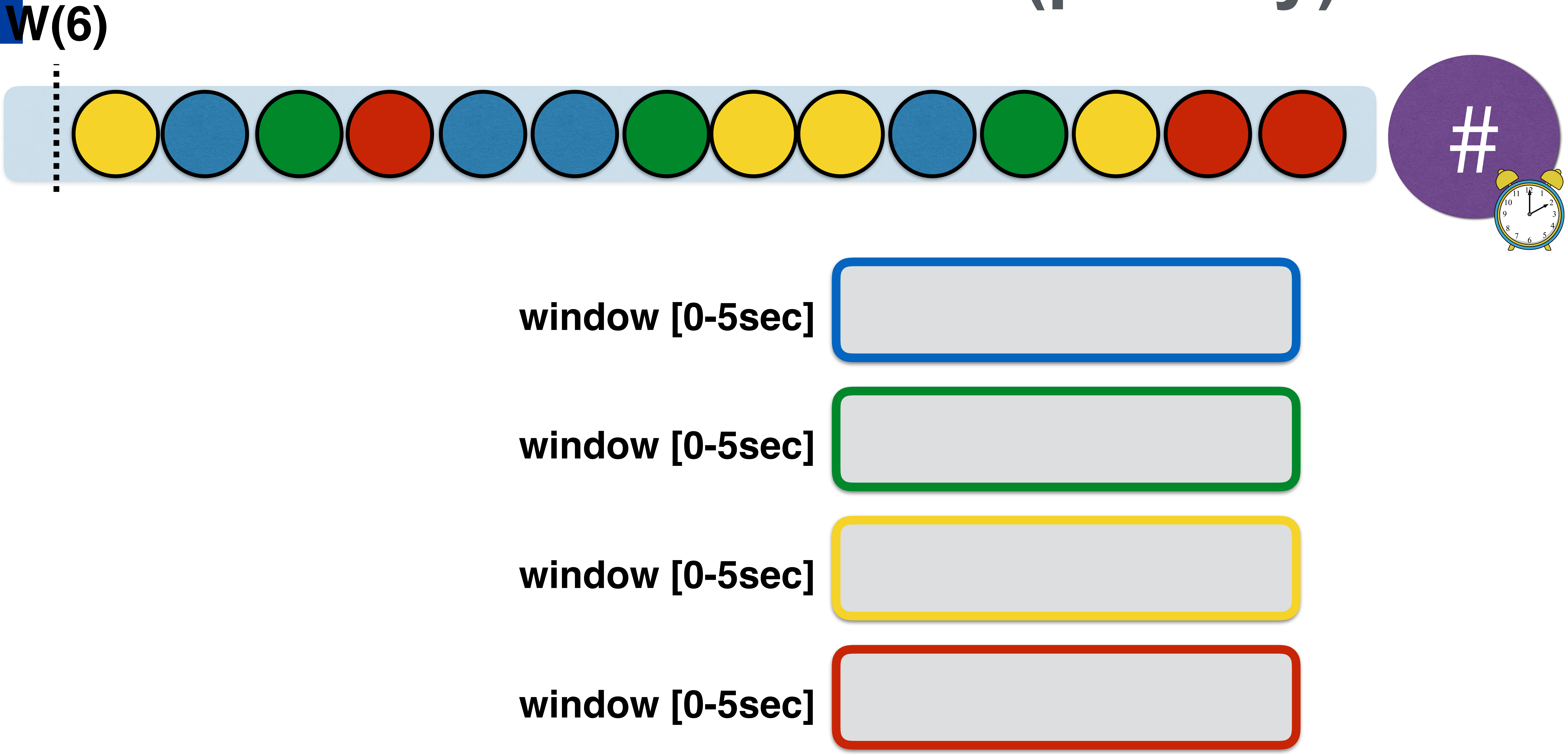
Lateness
Bound: 2sec

Example

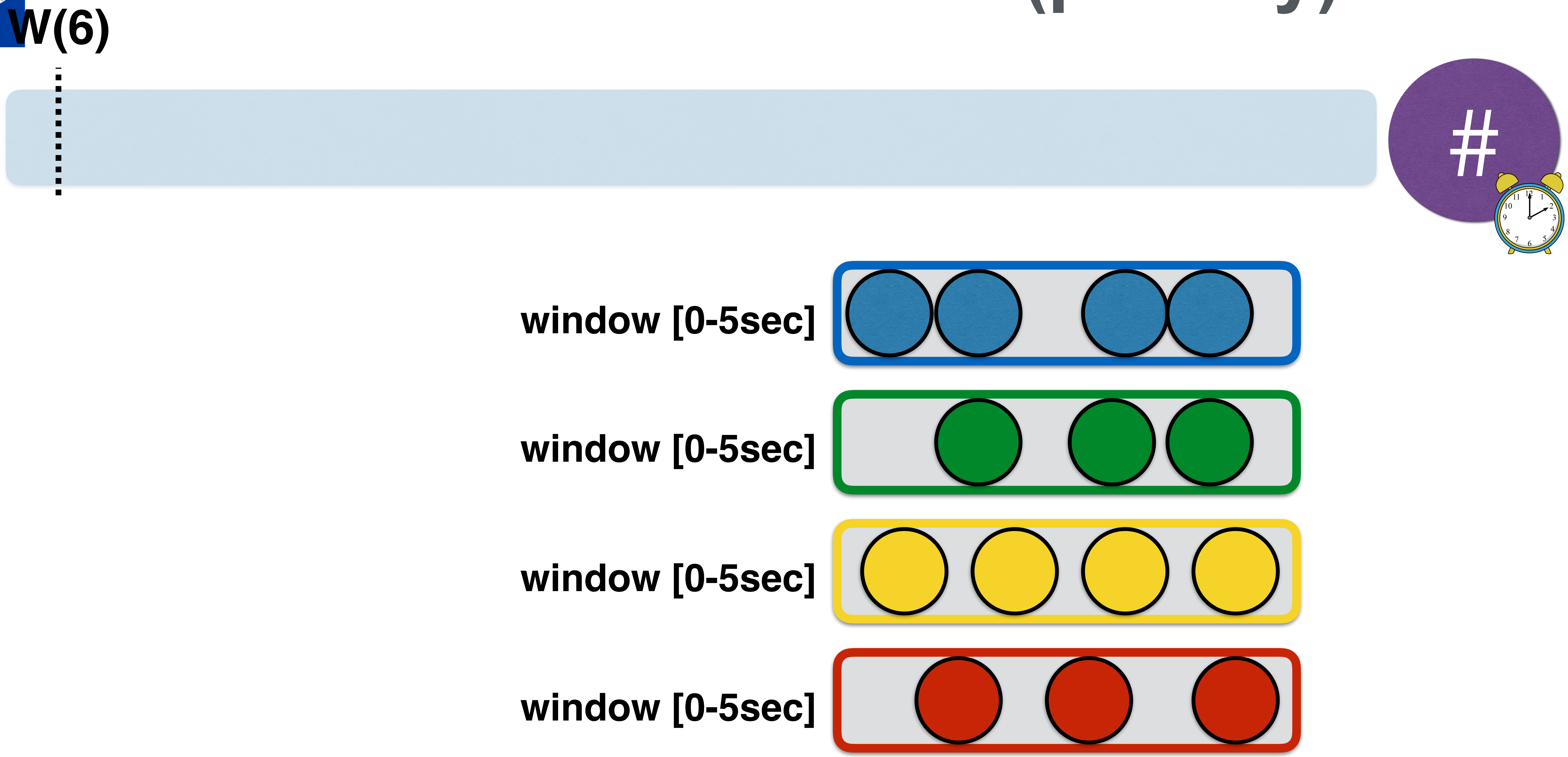


W(11)
15
13
8
6
1
4
2
#
9s
event time "clock"
3
complete
window [0-5sec]
Lateness
Bound: 2sec
8
6
window [6-10sec]
13
window [11-15sec]

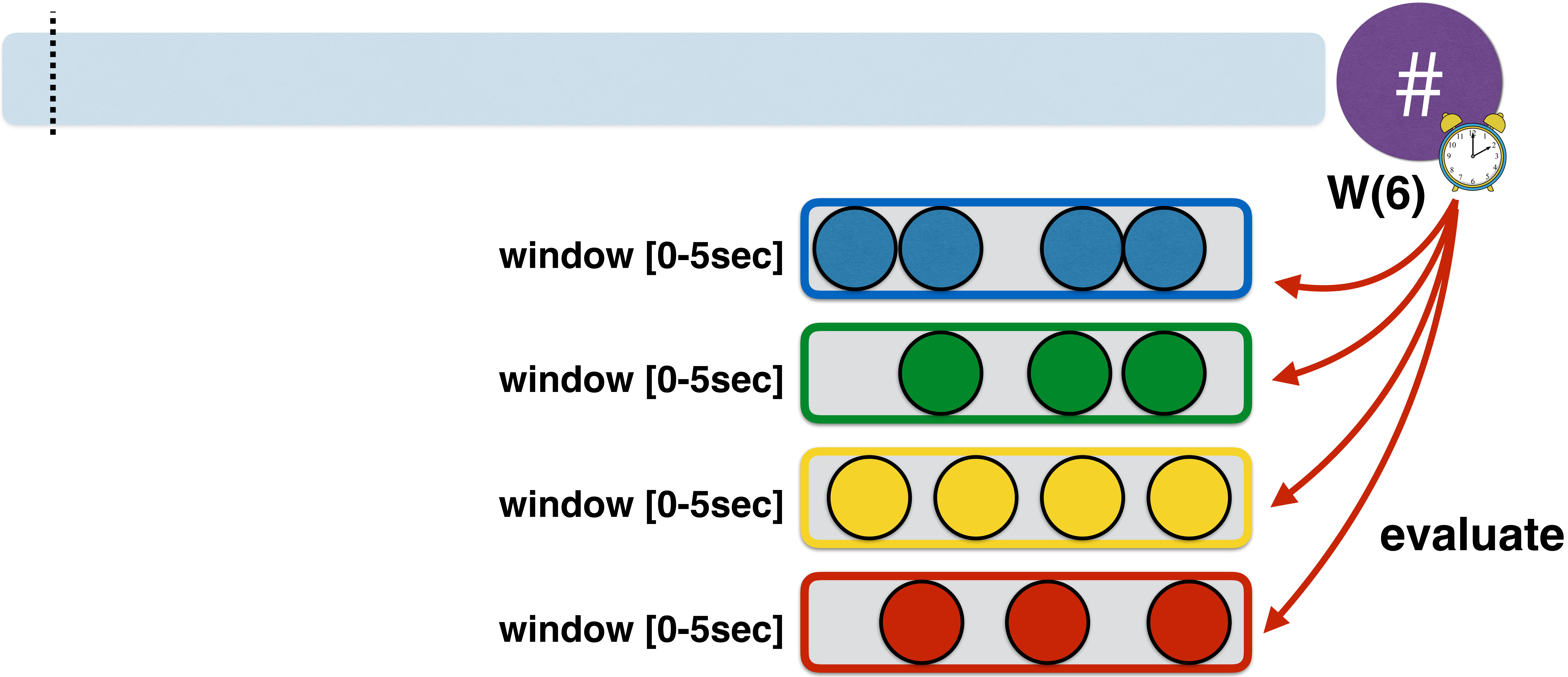
Data Parallel Windows (per key)



Data Parallel Windows (per key)



Data Parallel Windows (per key)



Getting Hands Dirty

http://training.ververica.com

https://github.com/ververica/sql-training

DOCS : https://ci.apache.org/projects/flink/flink-docs-release-1.9/

Further Readings

- [Paper] State Management In Apache Flink
- [Thesis] Scalable and Reliable Data Stream Processing
- [Paper] The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing.
- [Paper] One SQL to Rule Them All: An Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables
- [Paper] Out-of-order processing: a new architecture for high-performance stream systems.
- [Blog] The world beyond batch: Streaming 101 by Tyler Akidau https://www.oreilly.com/ideas/the-world-beyond-batch-streaming-101

Announcements

- **Flink Forward 2019, the premier conference in Apache Flink needs volunteers (attendance free of charge). https://europe-2019.flink-forward.org/register**

- **For MSc Thesis / Summer Job in Data Processing Systems Research mail me!**

Next-Gen Continuous Analytics

teaser