Data Stream Processing with Apache Flink

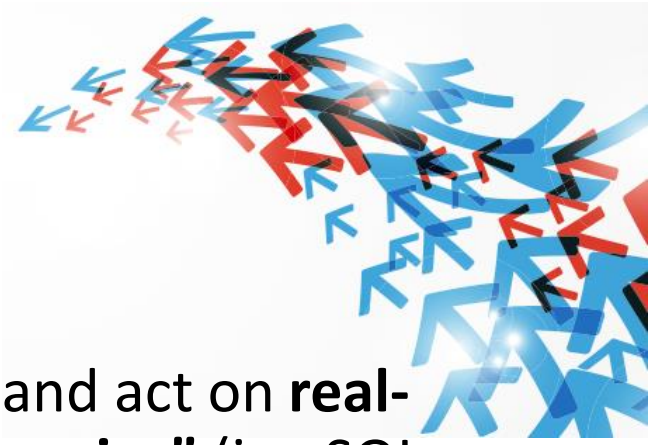
Sina Nourian

Contents

Introduction to Stream Processing and Apache Flink

Stream Processing

- Stream processing is a computer programming paradigm, equivalent to:
  - Dataflow Programming
  - Event Stream Programming
  - Reactive Programming



- Stream processing is designed to analyze and act on **real-time** streaming data, using **"continuous queries"** (i.e. SQL-type queries that operate over time and buffer windows).

- Stream processing solutions are designed to handle **high volume** in **real time** with a **scalable**, **highly available** and **fault tolerant** architecture.

Use Cases for Real Time Stream Processing Systems

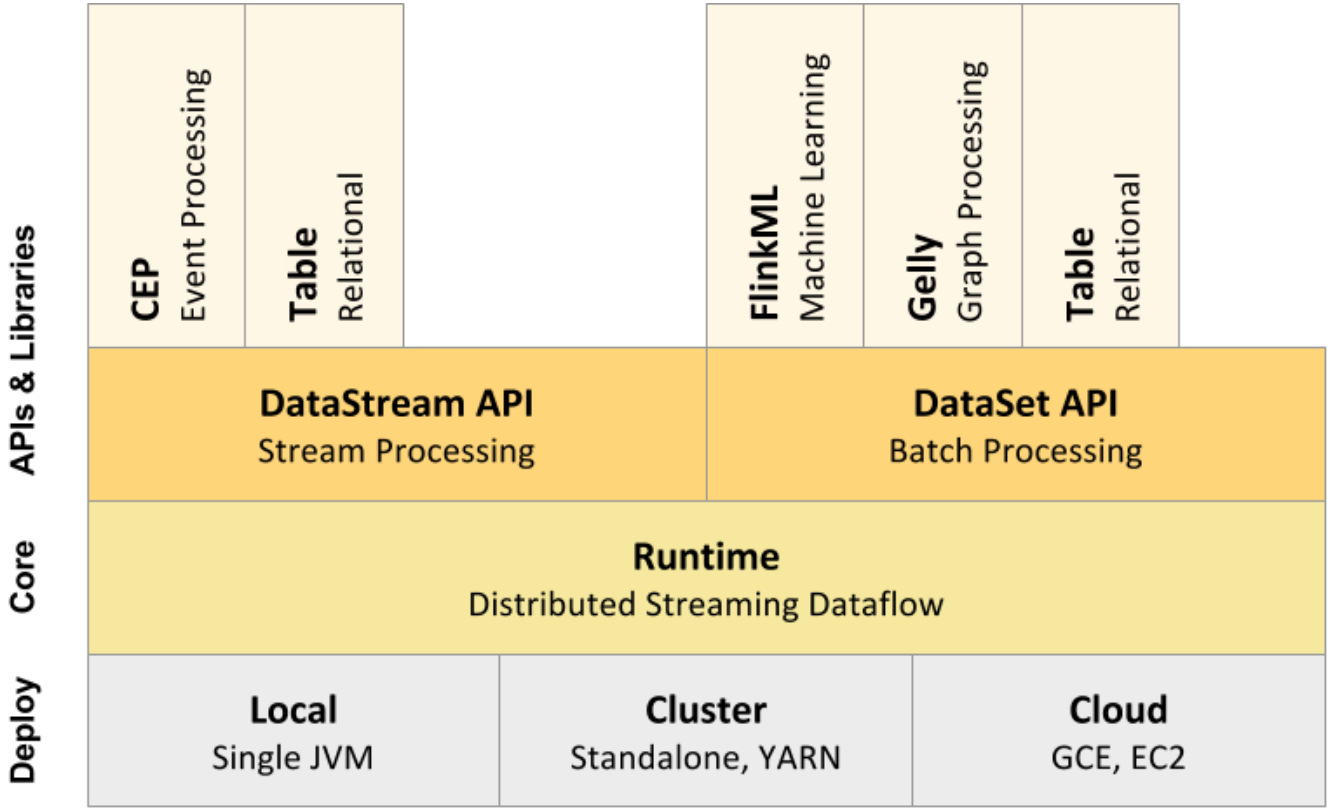
- **Financial Services**
  - Real-time fraud detection.
  - Real-time mobile notifications.
- **Healthcare**
  - Smart hospitals - collect data and readings from hospital devices (vitals, IVs, MRI, etc.) analyze and alert in real time.
  - Biometrics - collect and analyze data from patient devices that collect vitals while outside of care facilities.
- **Ad Tech**
  - Real-time user targeting based on segment and preferences.
- **Oil & Gas**
  - Real-time monitoring of pumps/rigs.
- **Telecommunications**
  - Real-time antenna optimization based on user location data.
  - Real-time charging and billing based on customer usage, ability to populate up-to-date usage dashboards for users.

What are the Requirements?

- **Low latency**
  - Results in millisecond
- **High throughput**
  - Millions of events per second
- **Exactly-once consistency**
  - Correct results in case of failures
- **Out-of-order events**
  - Process events based on their associated time

What is Apache Flink?

- Apache Flink is an open source platform for scalable stream and batch processing.
- The core of Flink is a distributed streaming dataflow engine.
  - Executes dataflows in parallel on clusters
  - Provides a reliable backend for various workloads



Comparison

- One of the strengths of Apache Flink is the way it combines many desirable capabilities that have previously required a trade-off in other projects.

					
Streaming Model	Native	Micro-batching	Micro-batching	Native	Native
API	Compositional		Declarative	Compositional	Declarative
Guarantees	At-least-once	Exactly-once	Exactly-once	At-least-once	Exactly-once
Fault Tolerance	Record ACKs		RDD based Checkpointing	Log-based	Checkpointing
State Management	Not build-in	Dedicated Operators	Dedicated DStream	Stateful Operators	Stateful Operators
Latency	Very Low	Medium	Medium	Low	Low
Throughput	Low	Medium	High	High	High
Maturity	High		High	Medium	Low

Who uses Flink?

ResearchGate

Alibaba.com

ERICSSON

King

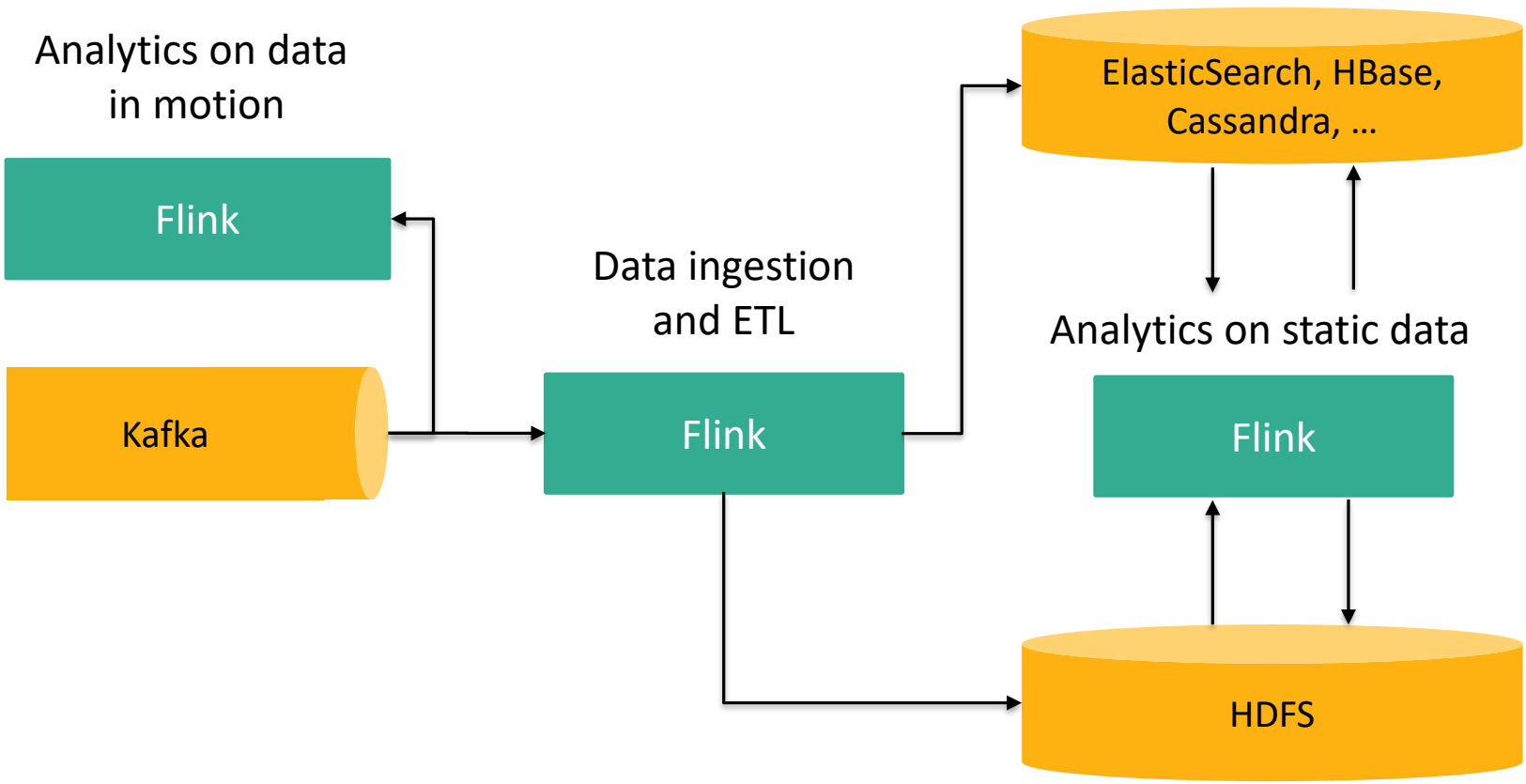
bouygues TELECOM

CapitalOne

otto group

zalando

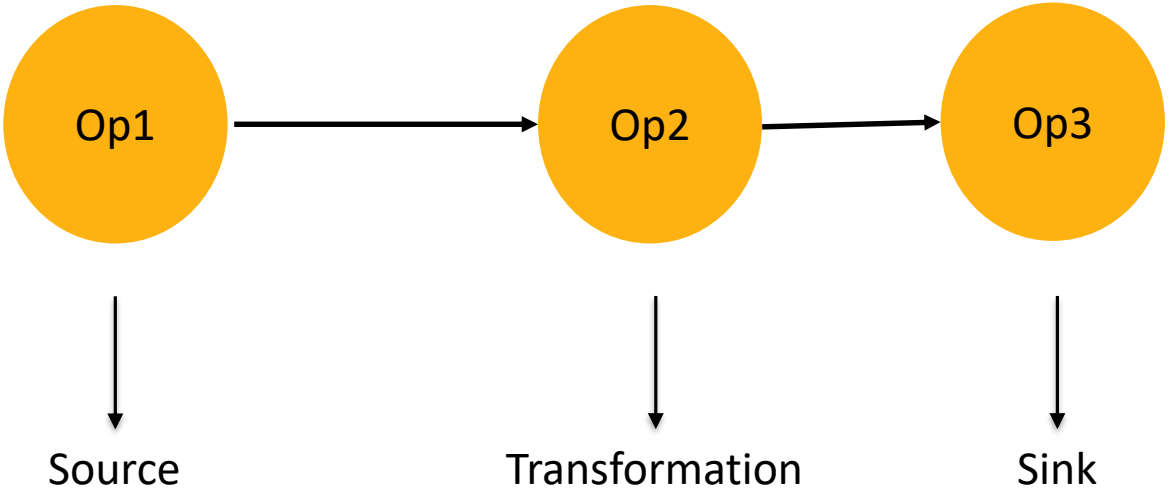
Flink in Streaming Architectures



Auto-Parallelizing Stateful Distributed Streaming Applications

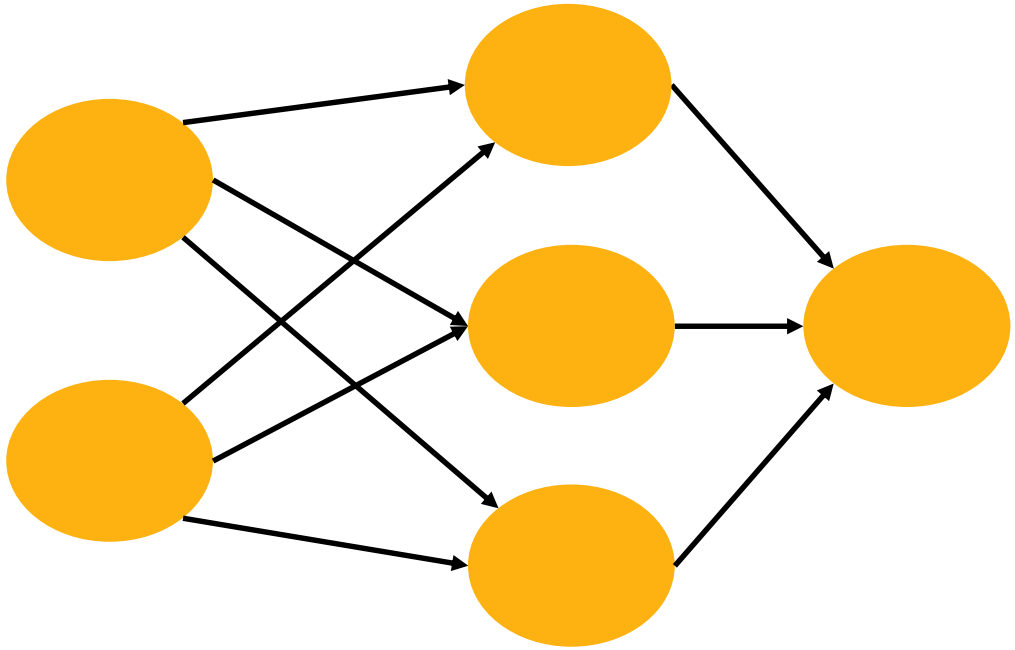
Stream Graph

- Streaming applications are directed graphs
  - Vertices are operators
  - Edges are data streams.



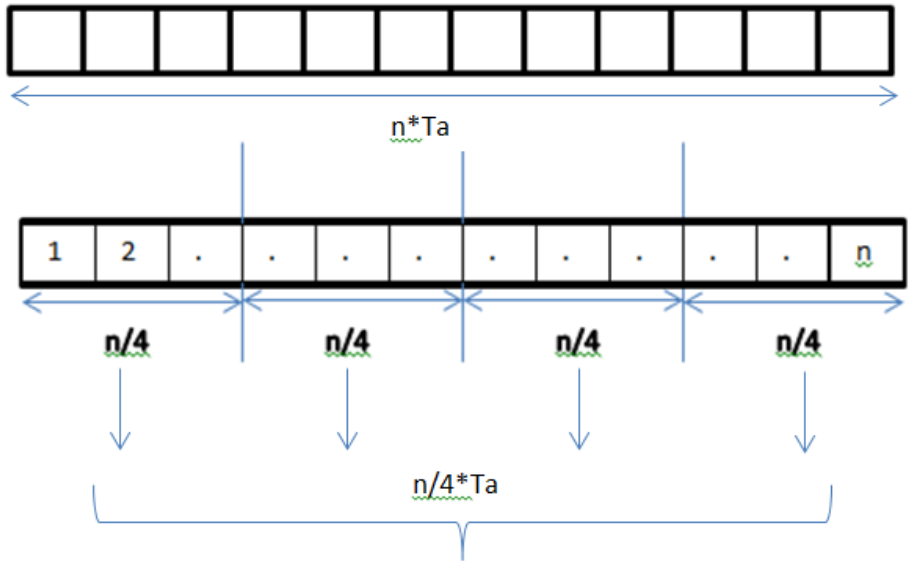
Stream Graph

- When operators are connected in chains, they expose inherent pipeline parallelism.
- When the same streams are fed to multiple operators that perform distinct tasks, they expose inherent task parallelism



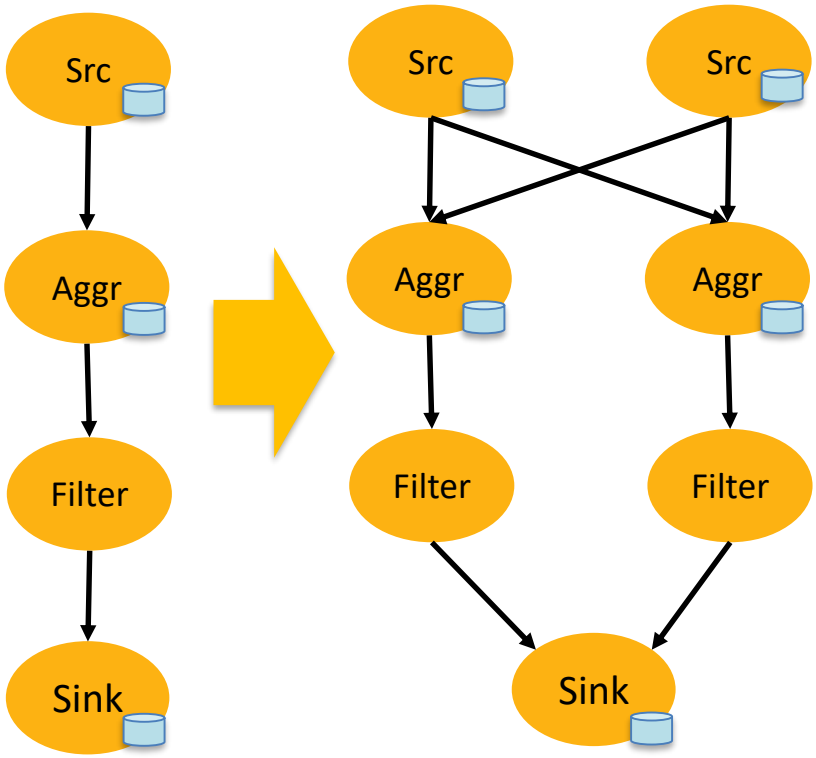
Data Parallelism

- Data parallelism involves splitting data streams and replicating operators.
- In a streaming context, replication of operators is data parallelism because each replica of an operator performs the same task on a different portion of the data.
- The parallelism obtained through replication can be more well-balanced than the inherent parallelism in a particular stream graph, and is easier to scale to the resources at hand.
- Data parallelism has the advantage that it is not limited by the number of operators in the original stream graph.



Data Parallelism

```
composite Main {
  type
    Entry = tuple<uint32 uid, rstring server, rstring msg>;
    Summary = tuple<uint32 uid, int32 total>;
  graph
    stream<Entry> Messages = ParSrc() {
      param    servers:      "logs.*.com";
               partitionBy:  server;
    }
    stream<Summary> Summaries = Aggregate(Messages) {
      window   Messages:   tumbling, time(5), partitioned;
      param    partitionBy: uid;
      output   Summaries: uid = Any(uid), total = Count();
    }
    stream<Summary> Suspects = Filter(Summaries) {
      param    filter:       total > 100;
    }
    () as Sink = FileSink(Suspects) {
      param    file:         "suspects.csv";
               format:       csv;
    }
}
```



Routing

- When parallel regions only have **stateless operators**, the splitters route tuples in **round-robin** fashion, regardless of the ordering strategy.
- When parallel regions have **partitioned state**, the splitter uses all of the attributes that define the partition key to compute a **hash value**. That hash value is then used to route the tuple, ensuring that the same attribute values are always routed to the same operators.
- **Stateful Operators:**
  - Join
  - Aggregate
  - KeyBy (PartitionBy)
  - Windows
  - …
- **Stateless Operators:**
  - Map
  - Filter
  - …

Flink's Dataflow Programming Model

Programs and Dataflows

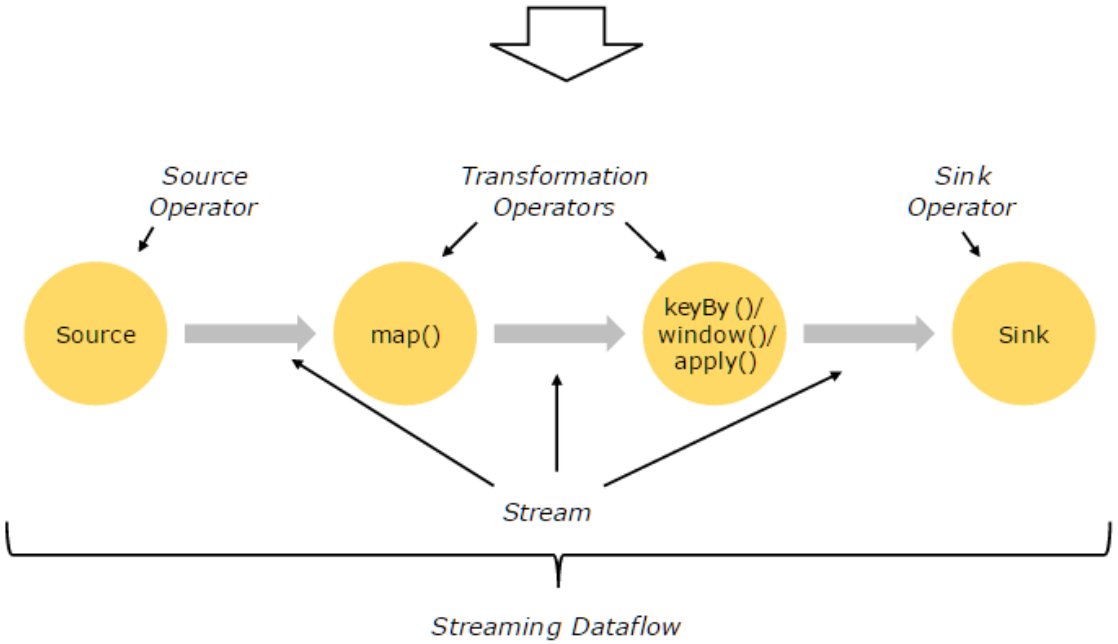
```
DataStream<String> lines = env.addSource(
                           new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new RollingSink(path));
```

Source

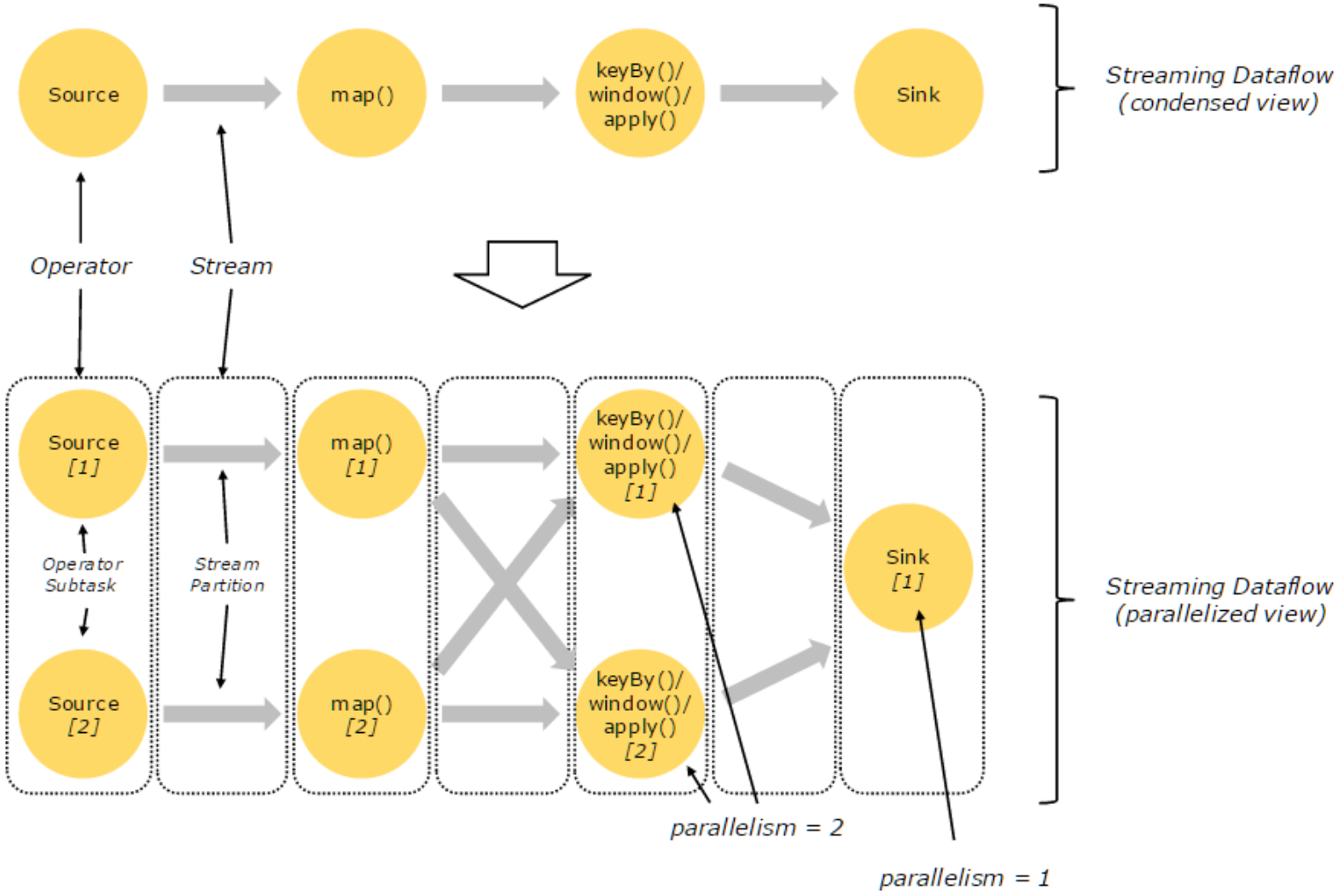
Transformation

Transformation

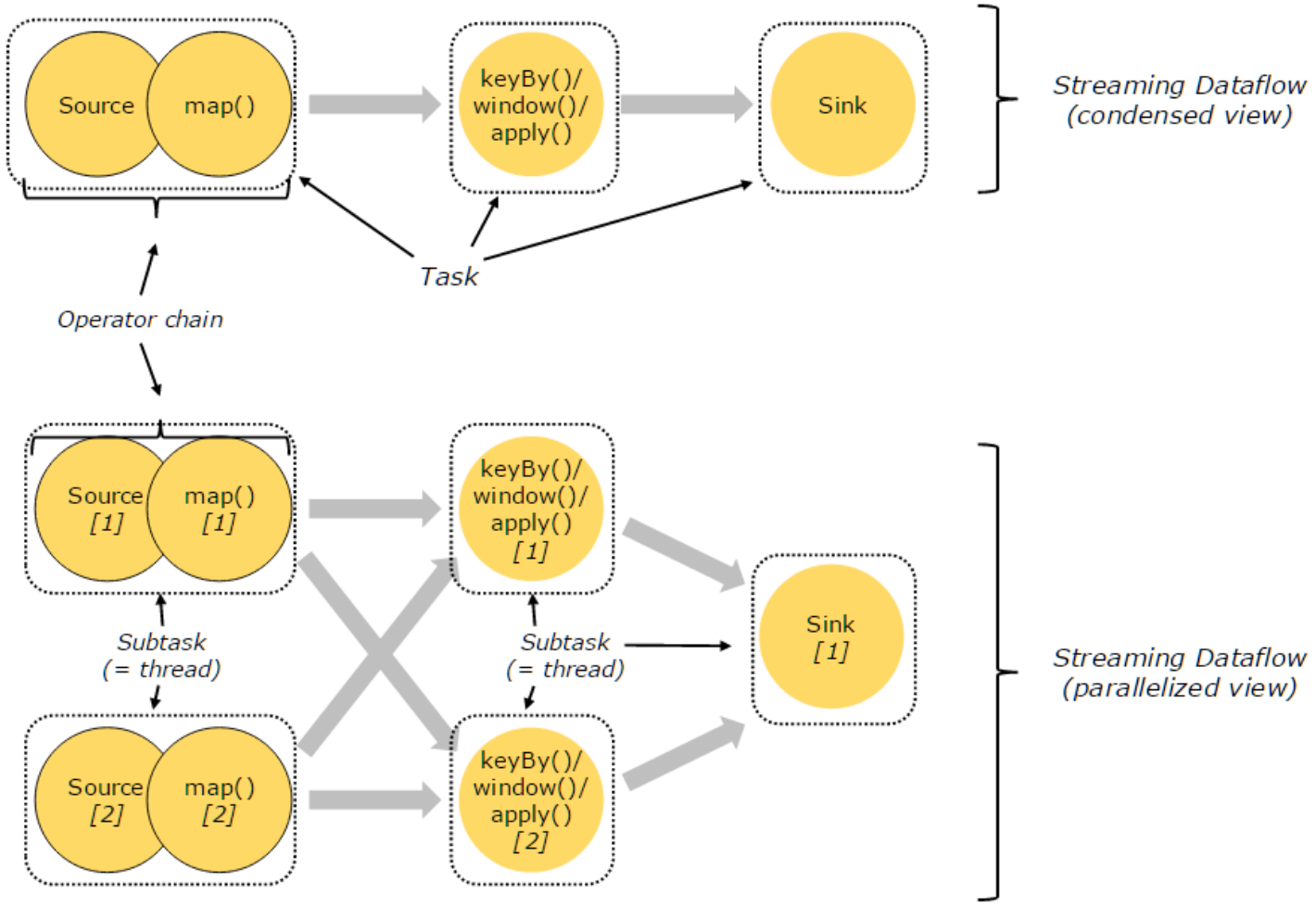
Sink



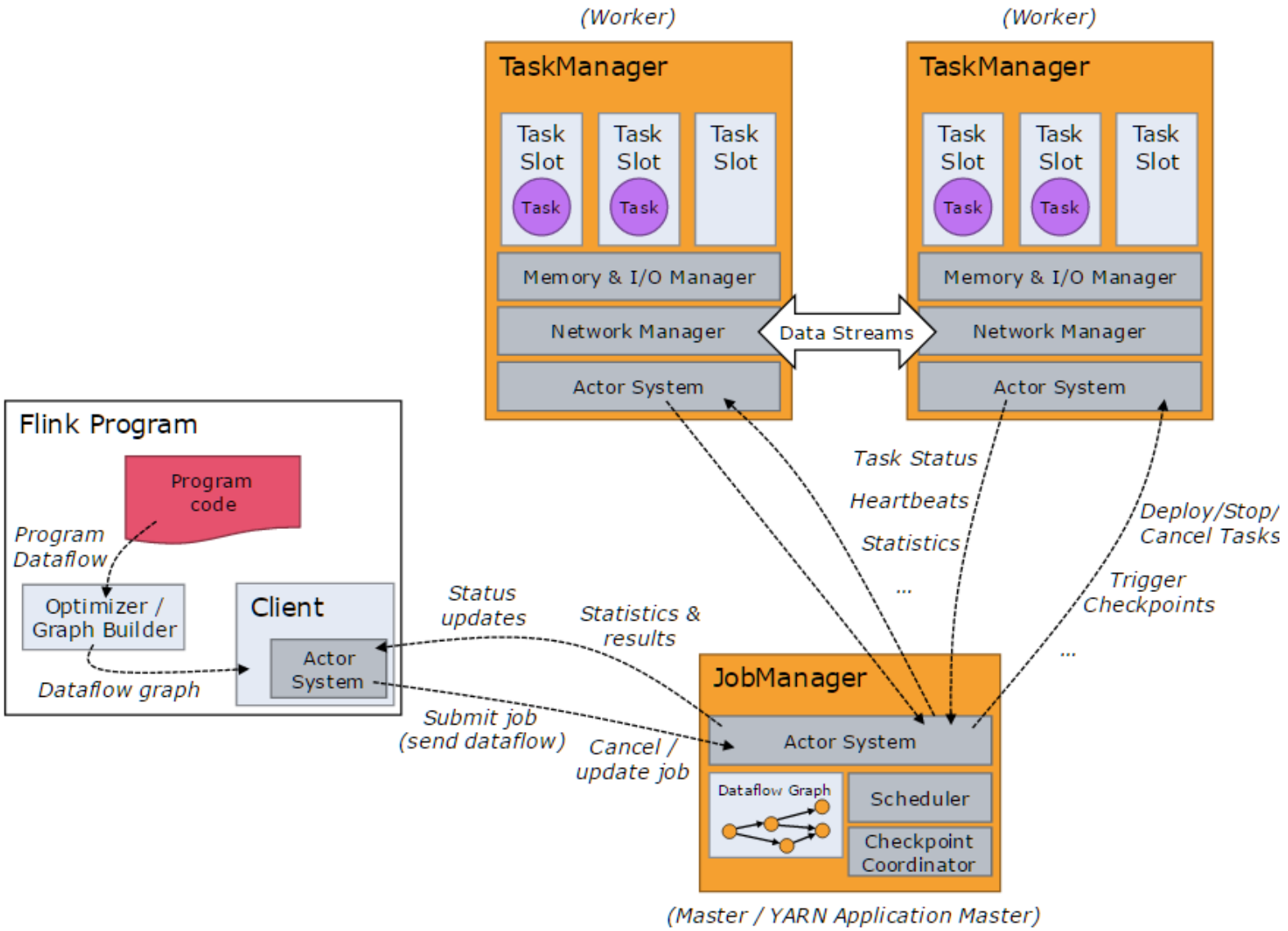
Parallel Dataflows



Tasks and Operator Chains

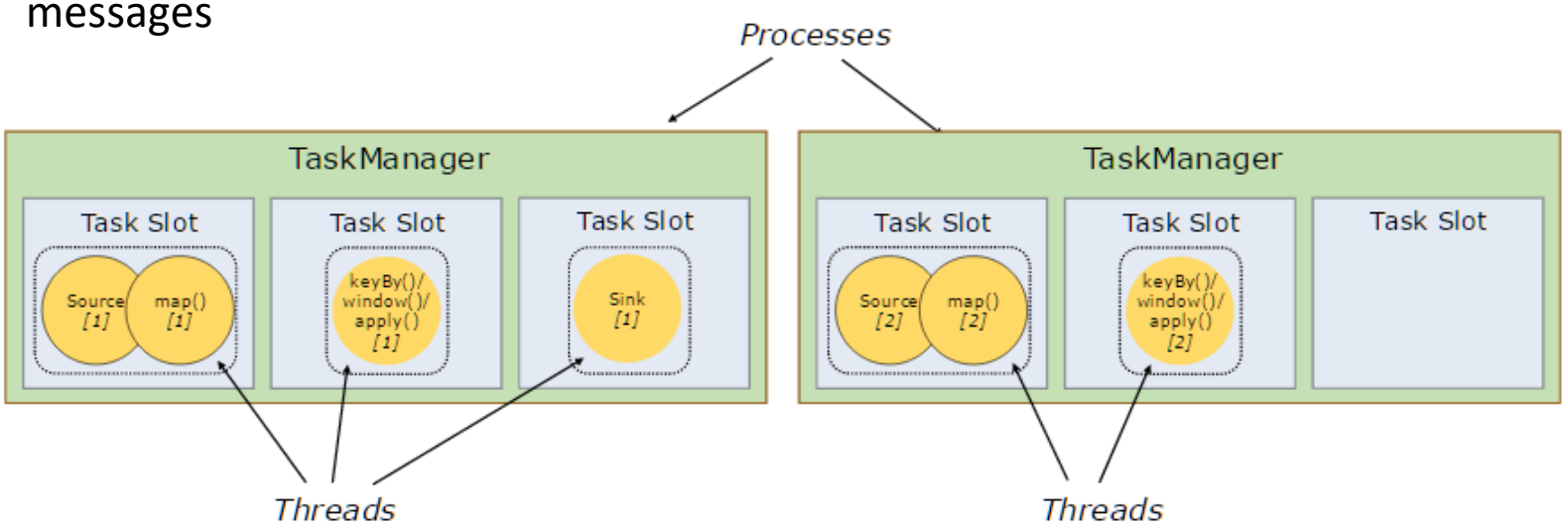


Distributed Execution



Task Slots and Resources

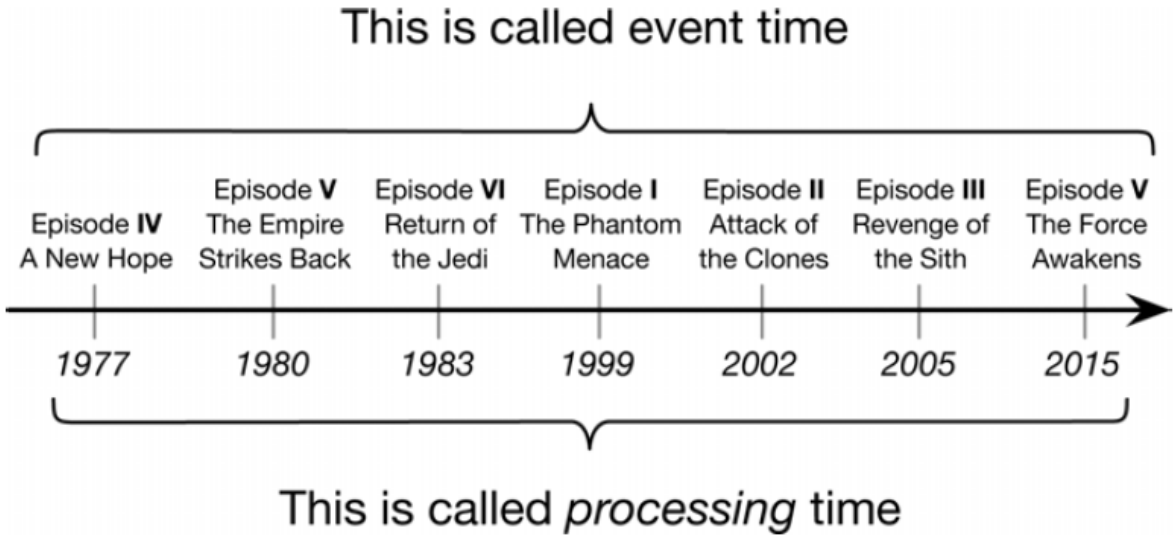
- Each worker (TaskManager) is a *JVM process*, and may execute one or more subtasks in separate threads.
- Each *task slot* represents a fixed subset of resources of the TaskManager. A TaskManager with three slots, for example, will dedicate 1/3 of its managed memory to each slot.
- By adjusting the number of task slots, users can define how subtasks are isolated from each other. Having one slot per TaskManager means each task group runs in a separate JVM
- Tasks in the same JVM share TCP connections (via multiplexing) and heartbeat messages



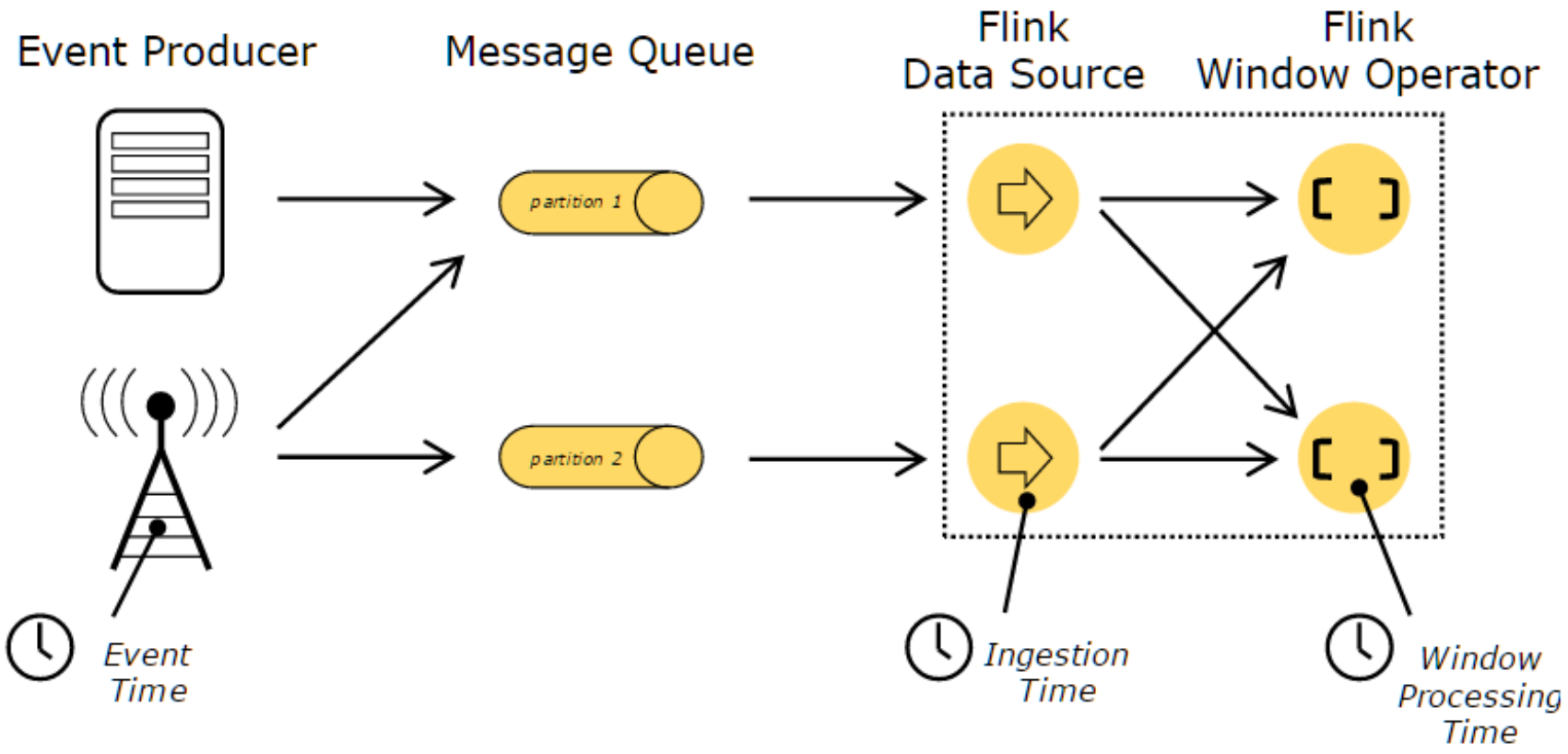
Handling Time

Notions of Time

- Processing Time
  - The time that the event is observed by the machine that is processing it.
  - Best performance and the lowest latency.
- Event Time
  - The time that an event actually happened in the real world.
- Ingestion Time
  - The time that the event enters the stream processing framework.

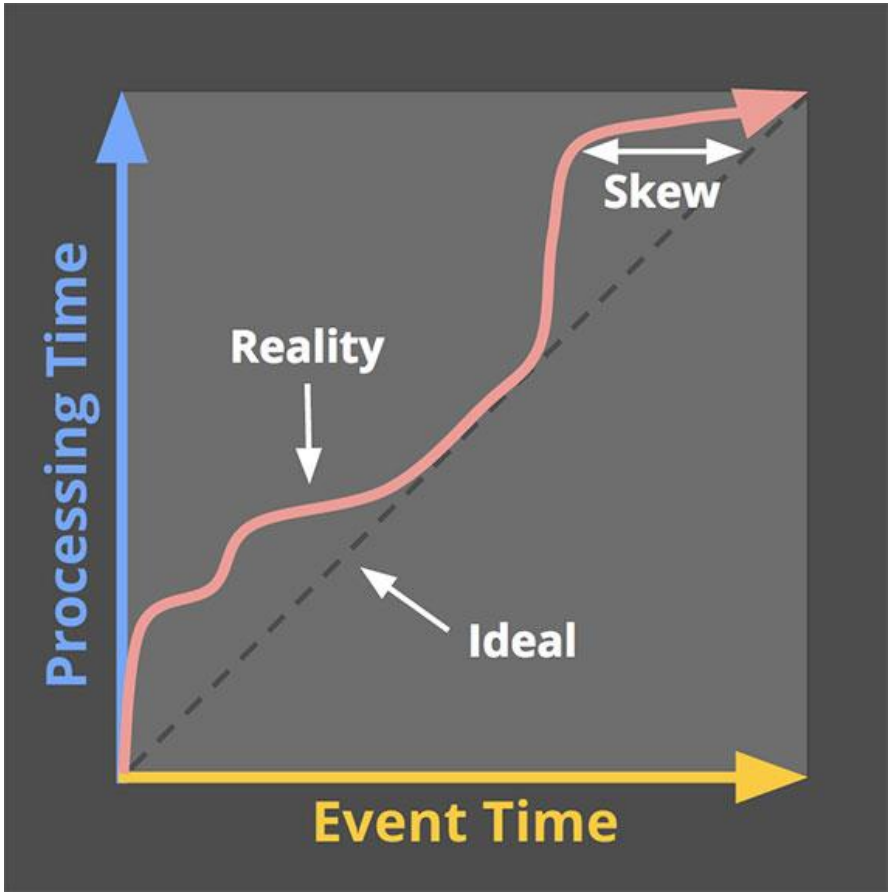


Notions of Time



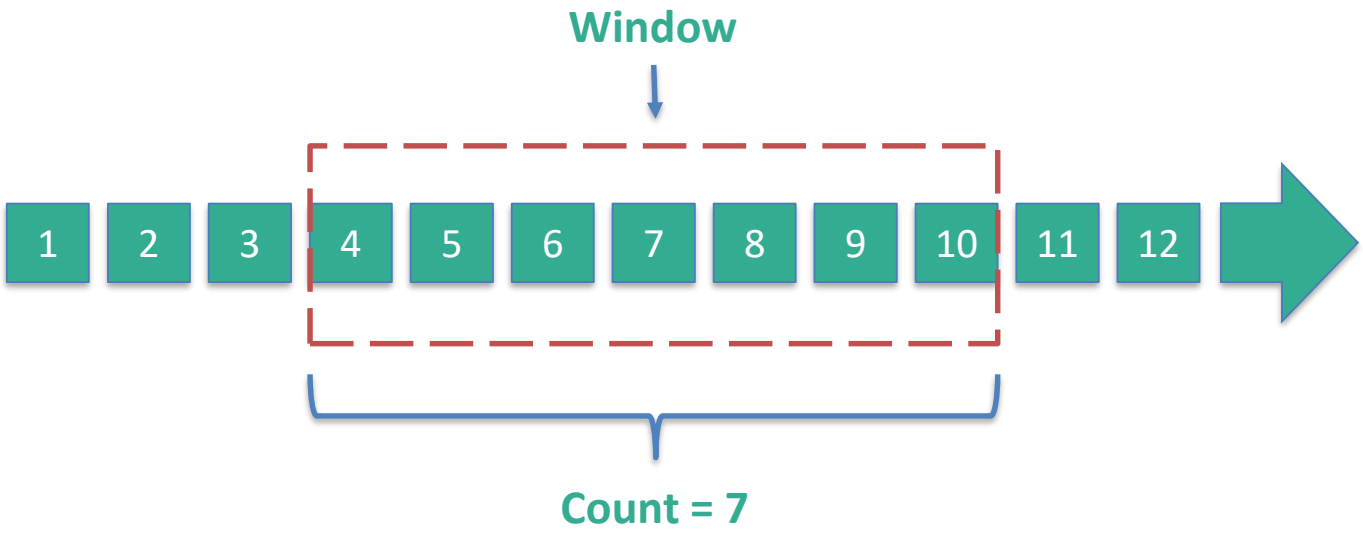
Notions of Time

- Event time and processing time always have a time-varying lag (called event time skew).



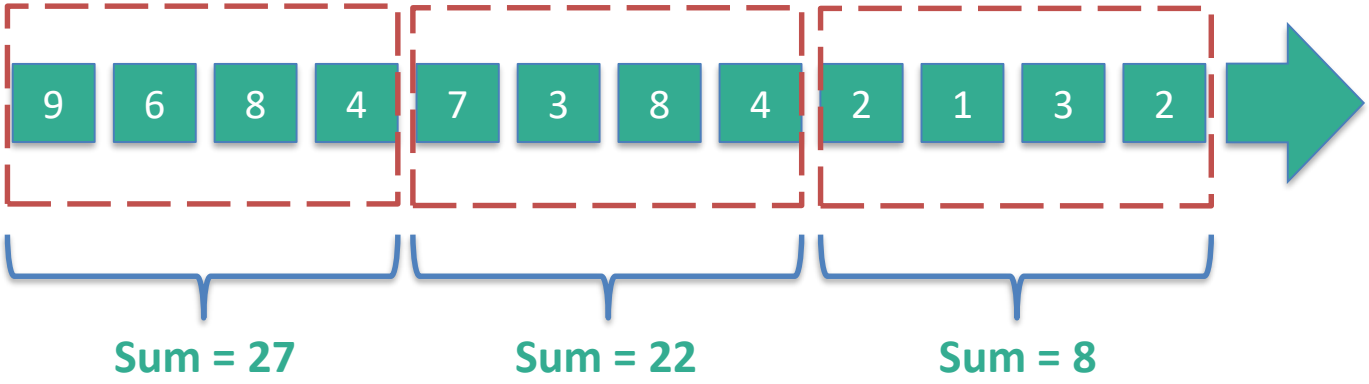
Window

- Windows are the mechanism to group and collect a bunch of events by time or some other characteristic in order to do some analysis on these events as a whole (e.g., to sum them up).



Time Window

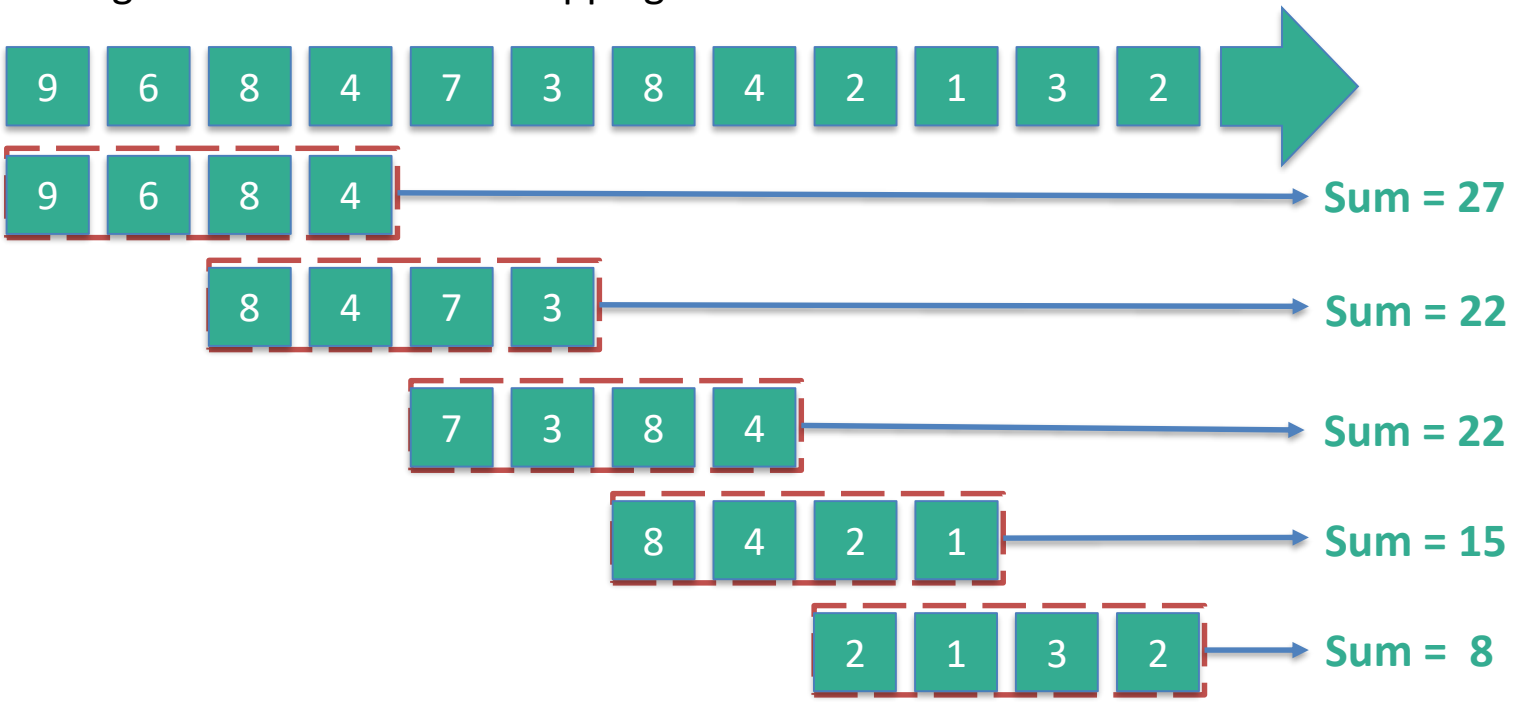
- Tumbling Windows
  - A *tumbling windows* assigner assigns each element to a window of a specified *window size*.
  - Tumbling windows have a fixed size and do not overlap.



A tumbling time window of 1 minute that sums the last minute's worth of values.

Time Window

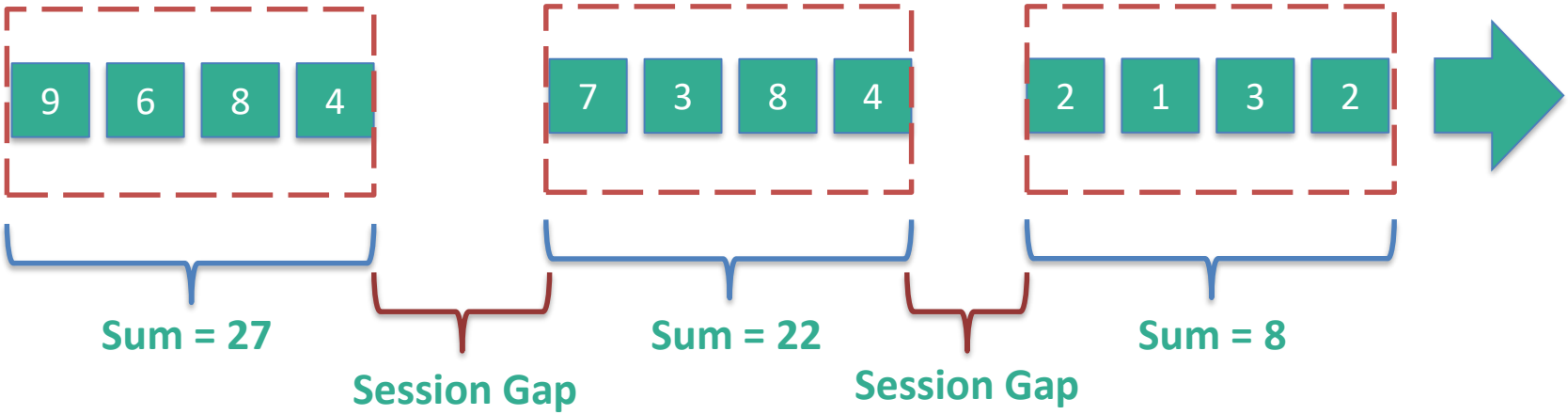
- Sliding Windows
  - The sliding windows assigner assigns elements to windows of fixed length.
  - Sliding windows can be overlapping if the slide is smaller than the window size.



A sliding time window that computes the sum of the last minute's values every half minute.

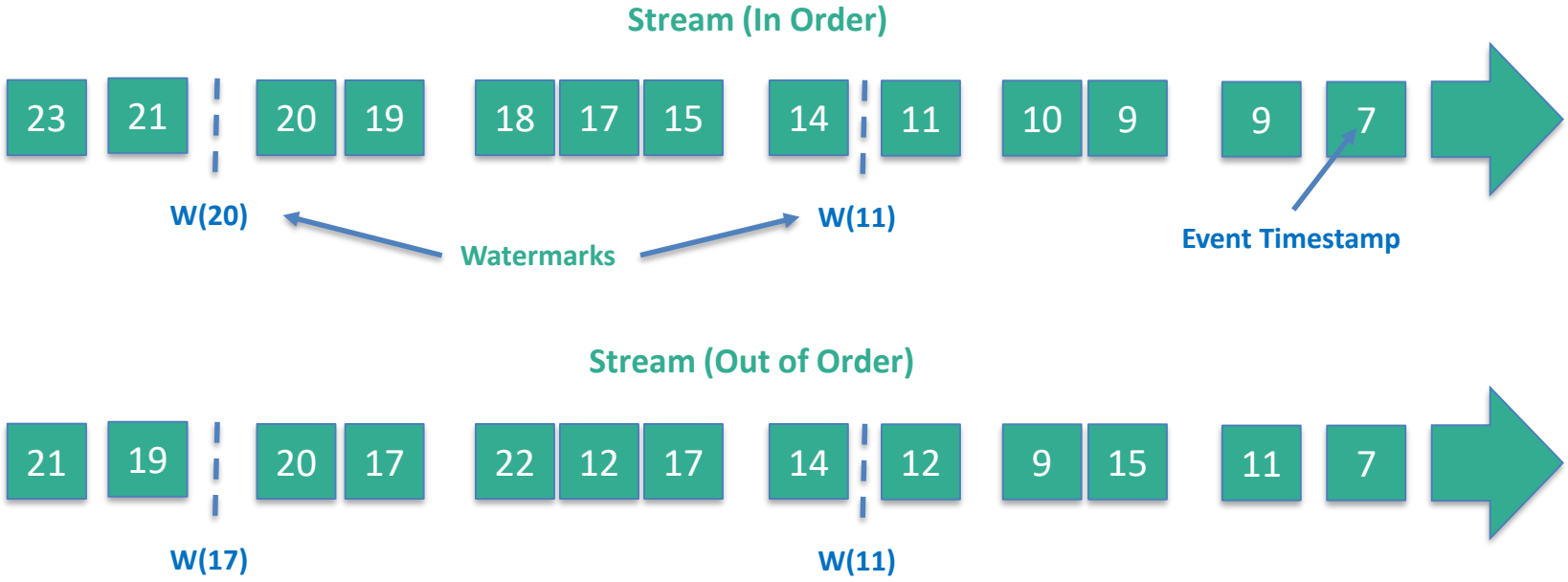
Session Windows

- Session Windows
  - The *session windows* assigner groups elements by sessions of activity.
  - A session window closes when it does not receive elements for a certain period of time.



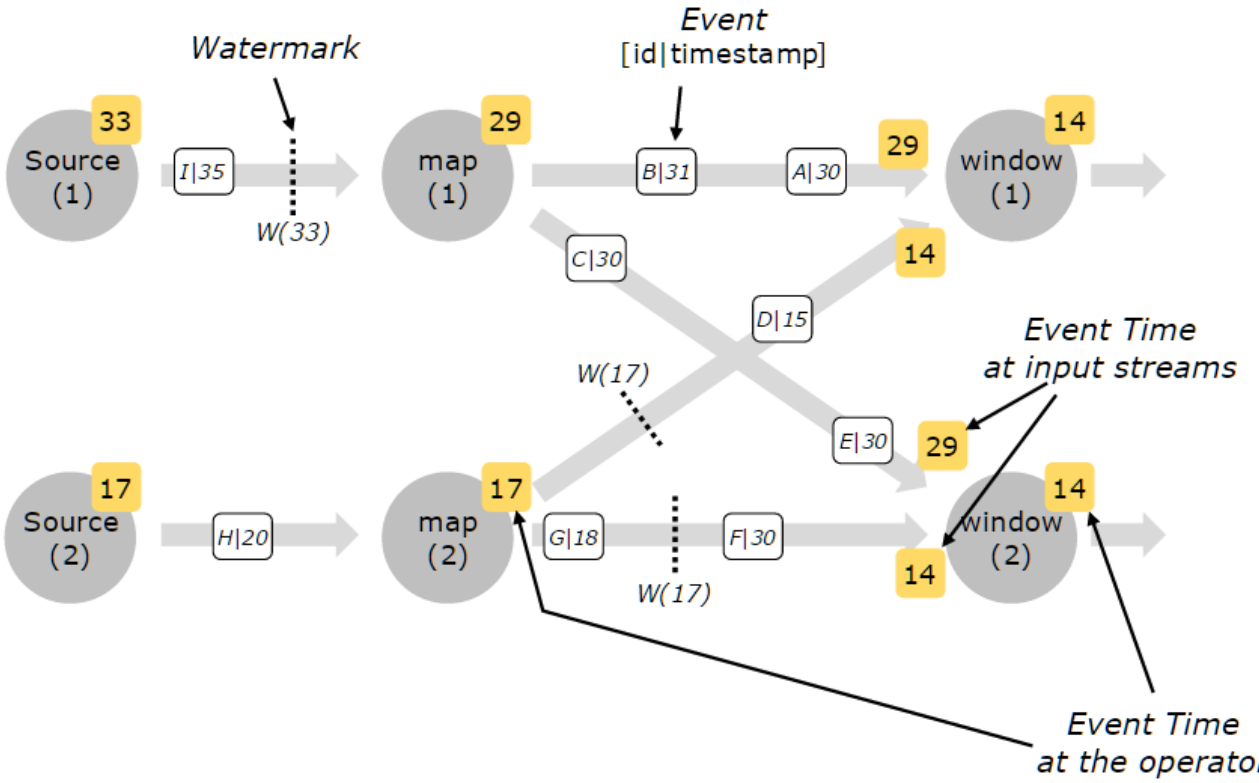
Watermarks

- The mechanism in Flink to measure progress in event time is **Watermarks**.
- Watermarks flow as part of the data stream and carry a timestamp *t*
- A *Watermark(t)* declares that event time has reached time *t* in that stream, meaning that there should be no more elements from the stream with a timestamp *t'* <= *t*



Watermarks in Parallel Streams

- As the watermarks flow through the streaming program, they advance the event time at the operators where they arrive.
- Whenever an operator advances its event time, it generates a new watermark downstream for its successor operators.
- The operator's current event time is the minimum of the input streams' event time. As the input streams update their event time, so does the operator.



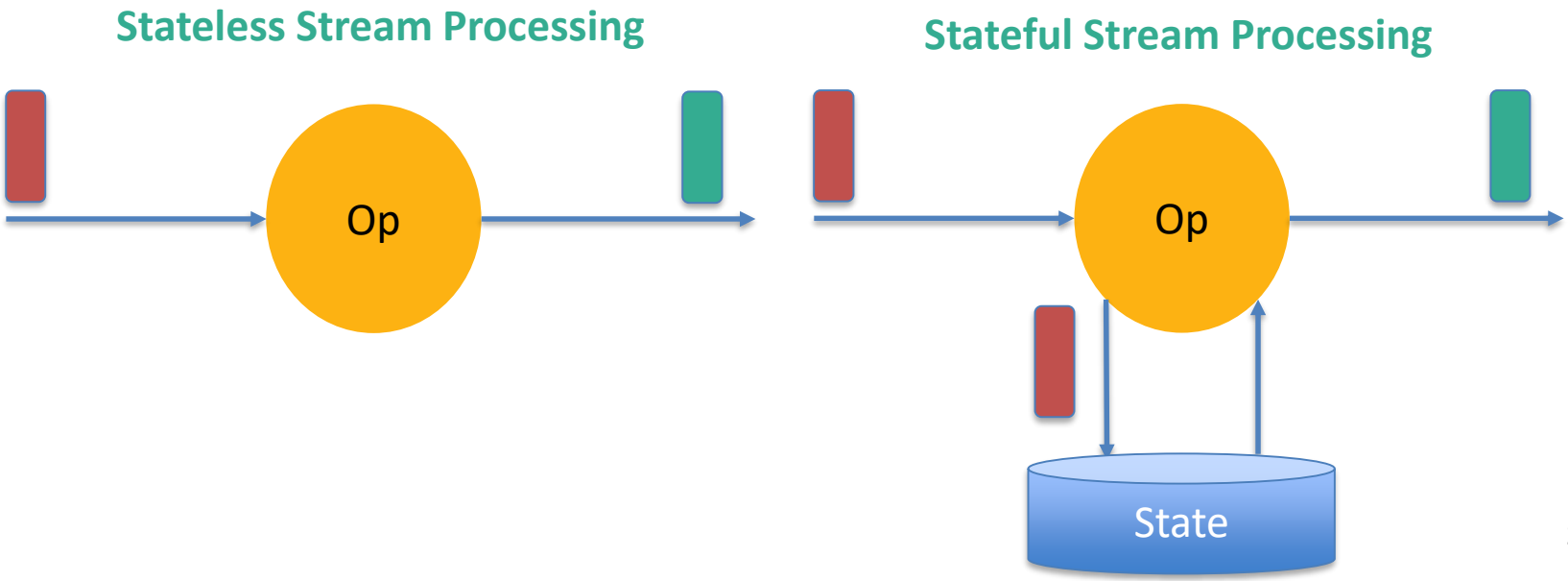
Late Elements

- If a certain elements violate the watermark condition, delaying the watermarks by too much is often not desirable, because it delays the evaluation of the event time windows by too much.
- Some streaming programs will explicitly expect a number of late elements.
- Late elements are elements that arrive after the system's event time clock (as signaled by the watermarks) has already passed the time of the late element's timestamp.

Stateful Computation

Stateful Computation

- A stateful program creates output based on multiple events taken together
  - All types of windows
  - All kinds of state machines used for complex event processing (CEP).
  - All kinds of joins between streams as well as joins between streams, and static or slowly changing tables.



Notions of Consistency

- Consistency is, really, a different word for **“level of correctness”**
  - How correct are my results after a failure and a successful recovery compared to what I would have gotten without any failures?
- Assume that we are simply counting user logins within the last hour.
  - What is the count (the state) if the system experiences a failure?

Notions of Consistency

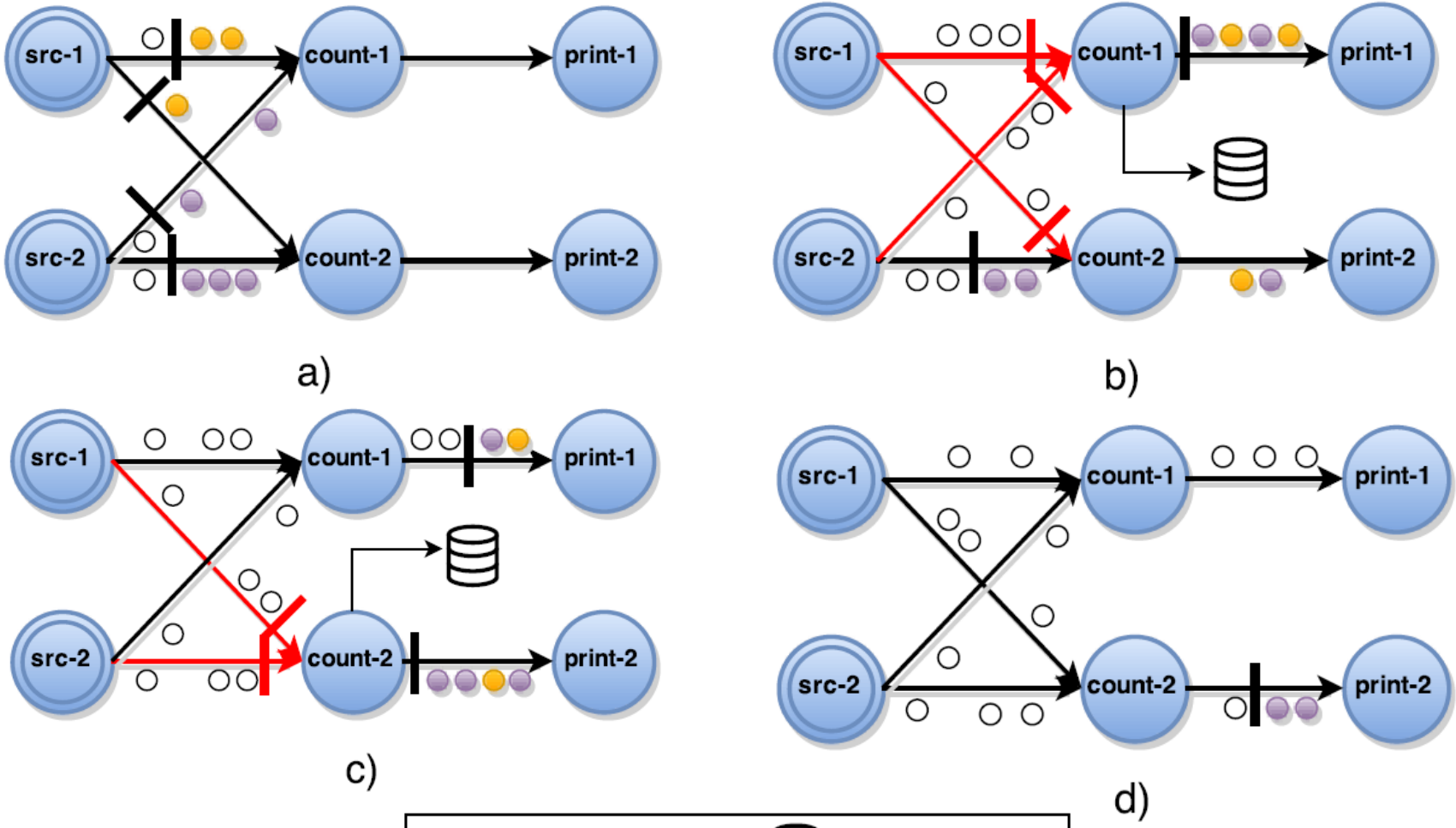
- People distinguish between three different levels of consistency:
  - **At most once:** At most once is really a euphemism for no correctness guarantees whatsoever—the count may be lost after a failure.
  - **At least once:** At least once, in our setting, means that the counter value may be bigger than but never smaller than the correct count. So, our program may over-count (in a failure scenario) but guarantees that it will never under-count.
  - **Exactly once:** Exactly once means that the system guarantees that the count will be exactly the same as it would be in the failure-free scenario.

Checkpoints: Guaranteeing Exactly Once

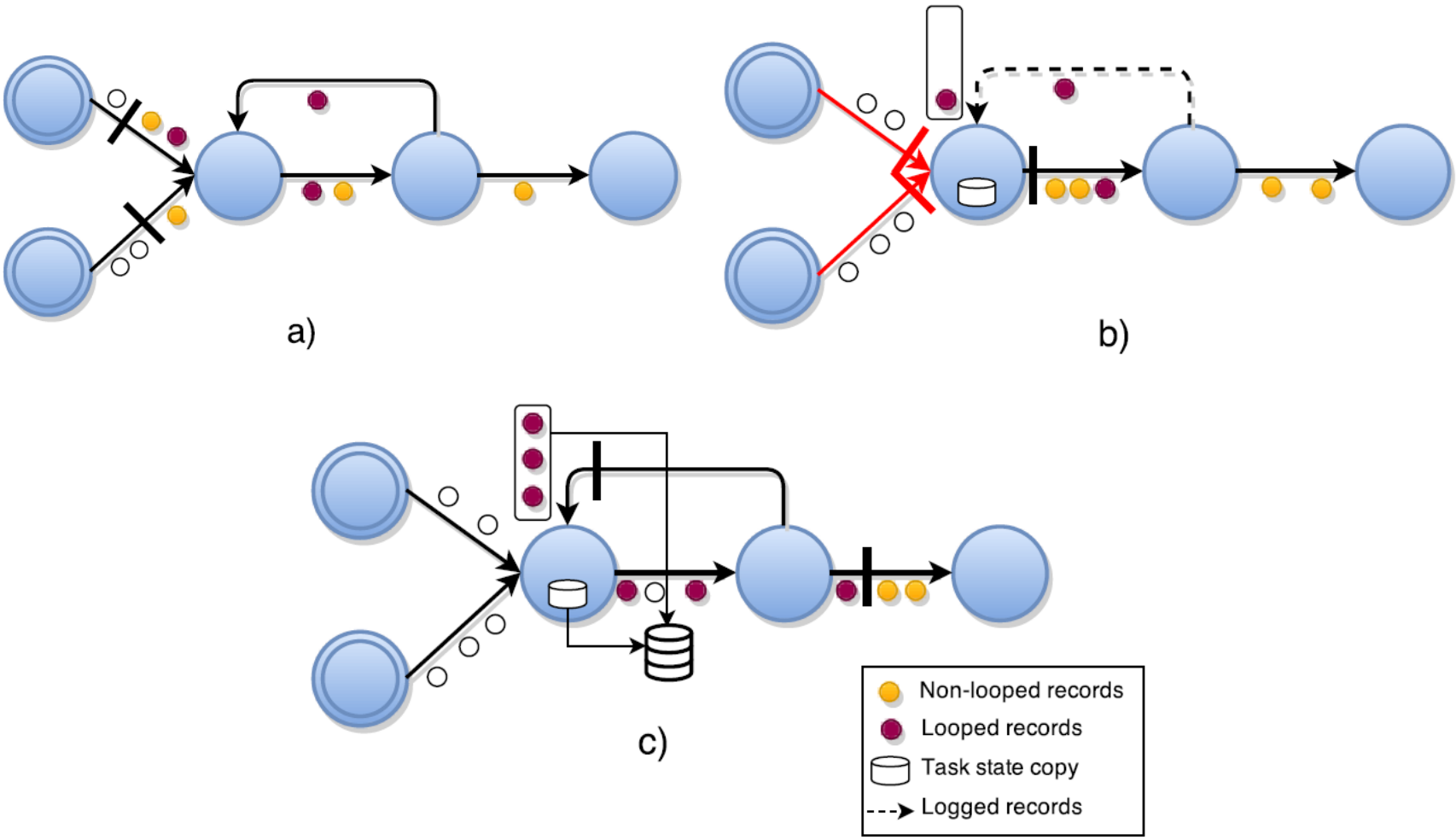
- Checkpoints allow Flink to recover state and positions in the streams to give the application the same semantics as a failure-free execution.

- Asynchronous Barrier Snapshotting (ABS) Algorithm
  - Low impact on performance
  - Low space costs
  - Linear scalability
  - Performing well with frequent snapshots.
  - Inspired by the standard **Chandy-Lamport** algorithm

ABS for Acyclic Dataflows

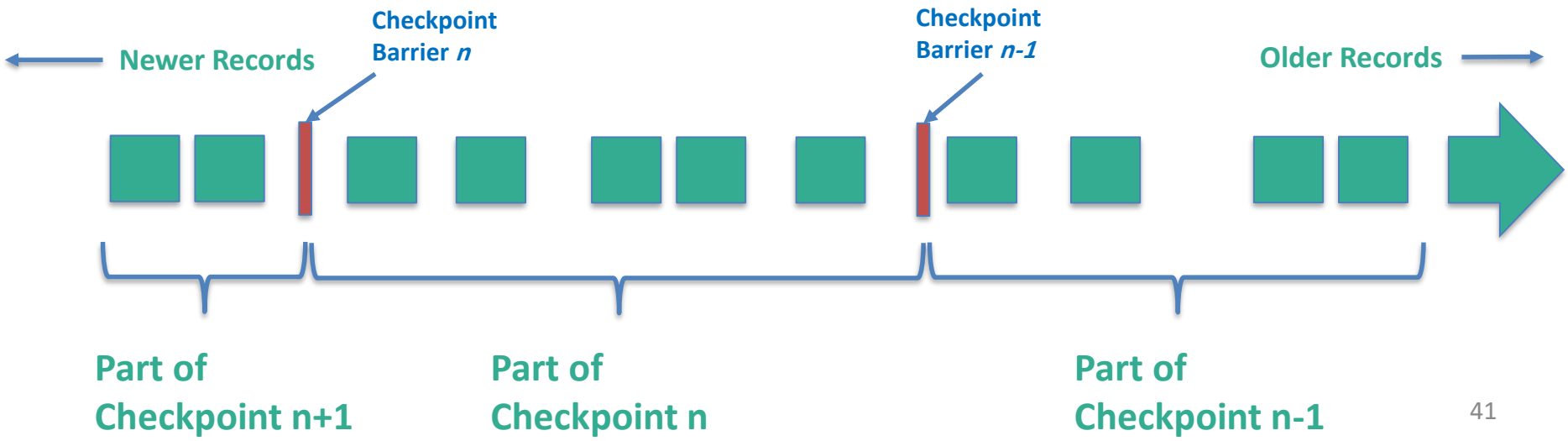


ABS for Cyclic Dataflows



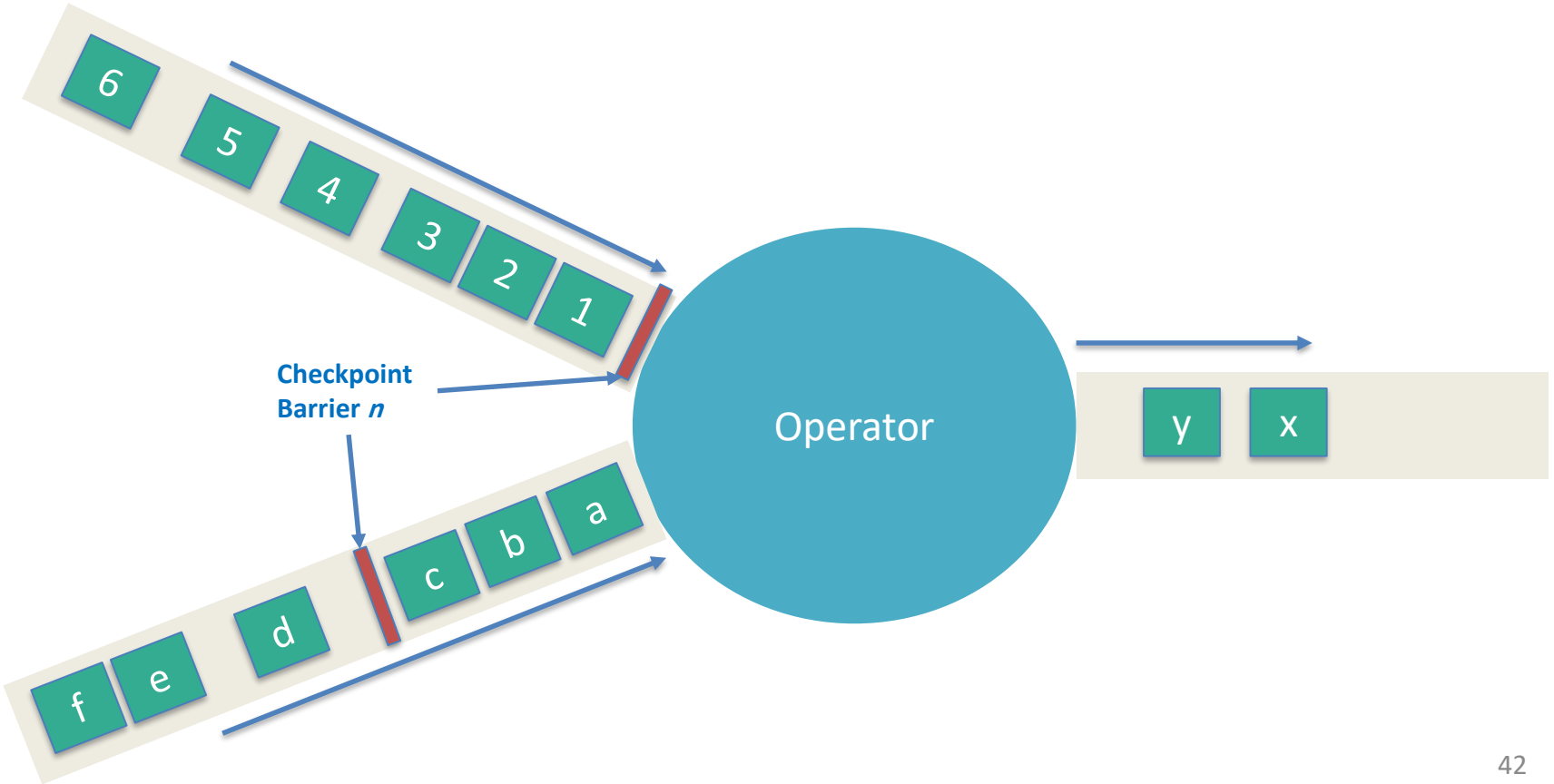
Checkpointing in Flink

- Barriers
  - Injected into the data stream and flow with the records as part of the data stream.
  - A barrier separates the records in the data stream into the set of records that goes into the current snapshot, and the records that go into the next snapshot.
  - Barriers do not interrupt the flow of the stream and are hence very lightweight.



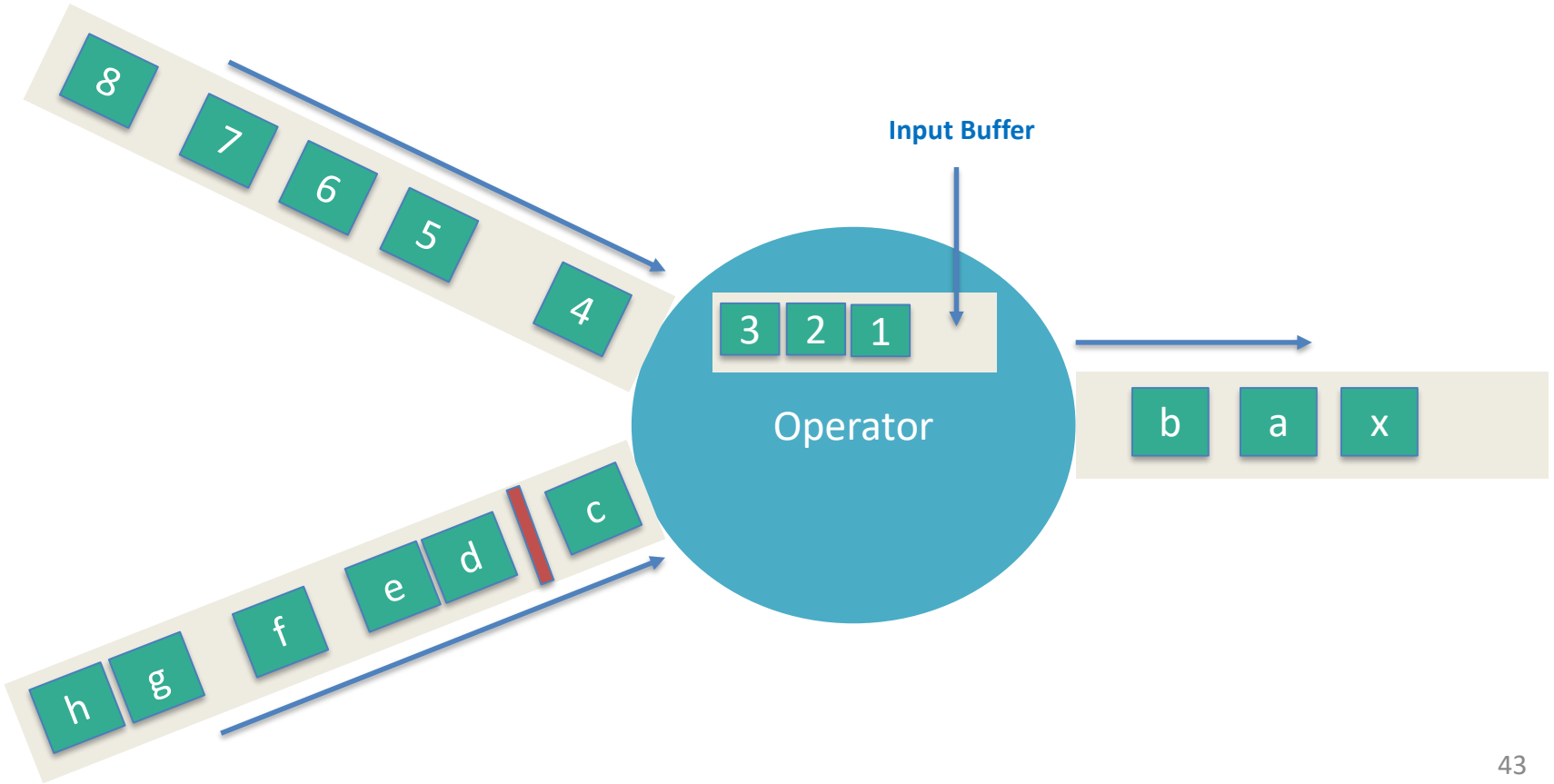
Checkpointing in Flink

- As soon as the operator received snapshot barrier *n* from an incoming stream, it cannot process any further records from that stream until it has received the barrier *n* from the other inputs as well. Otherwise, it would have mixed records that belong to snapshot *n* and with records that belong to snapshot *n+1*.



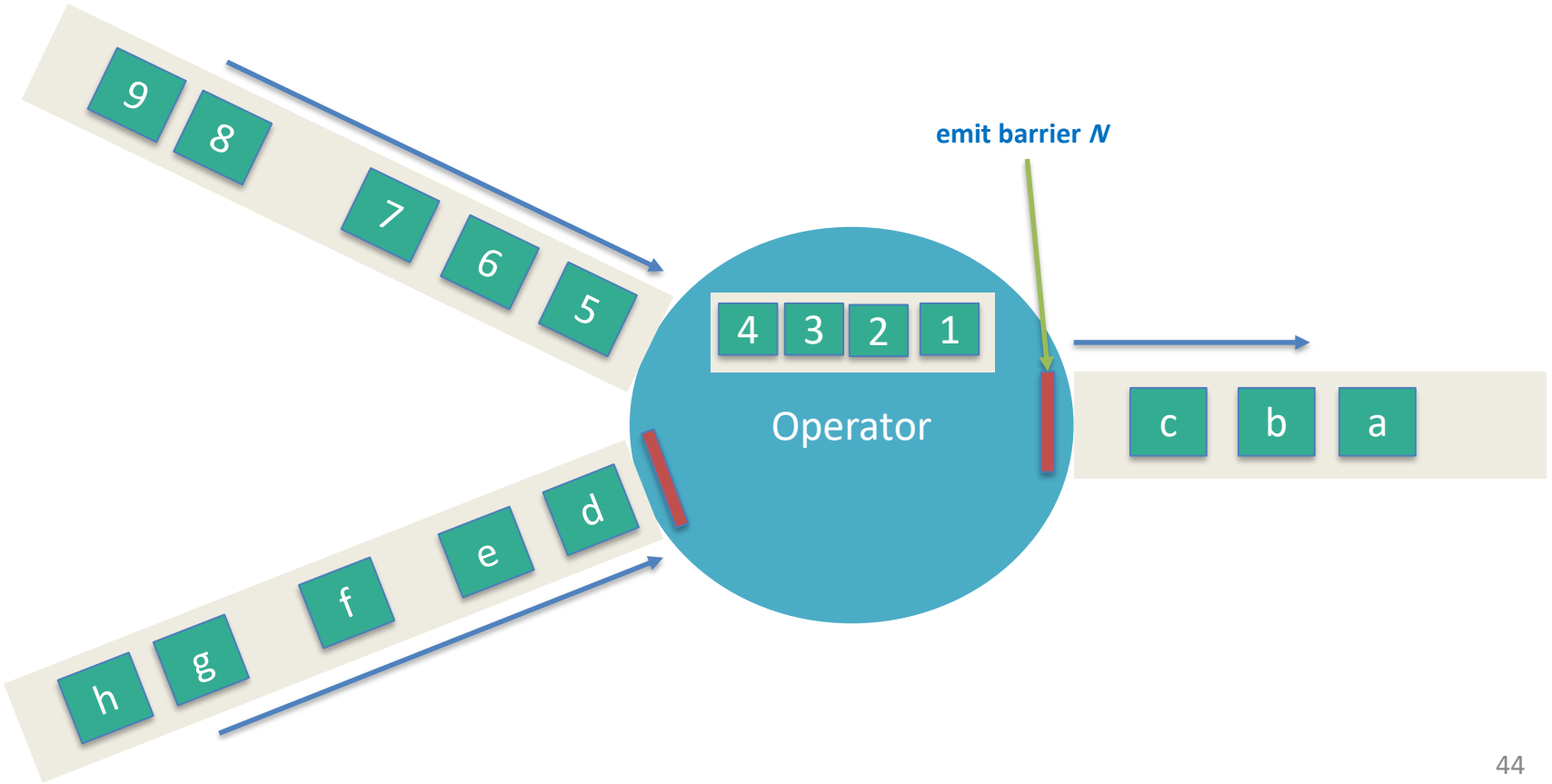
Checkpointing in Flink

- Streams that report barrier *n* are temporarily set aside. Records that are received from these streams are not processed, but put into an input buffer.



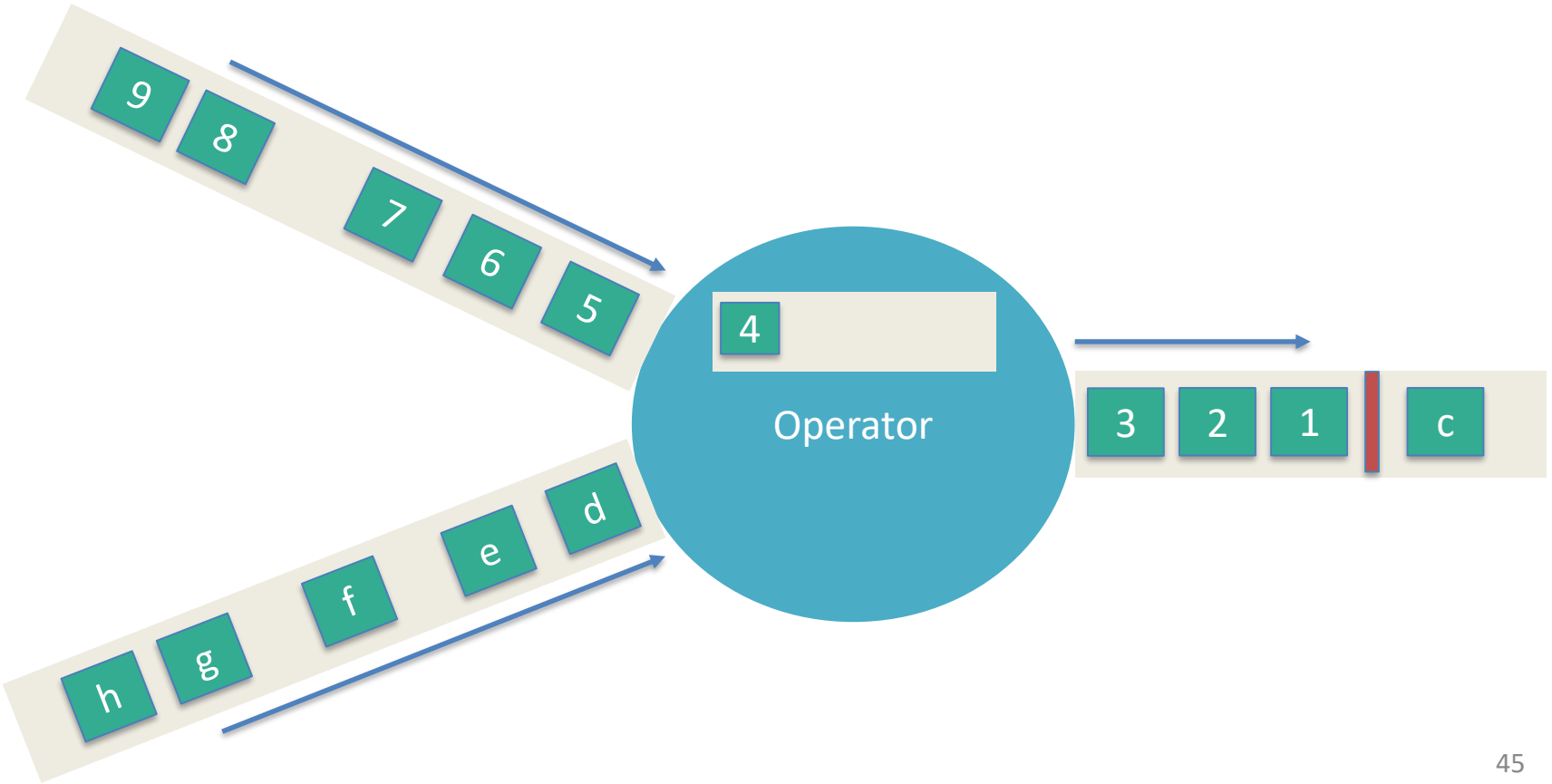
Checkpointing in Flink

- Once the last stream has received barrier *n*, the operator emits all pending outgoing records, and then emits snapshot *n* barriers itself.



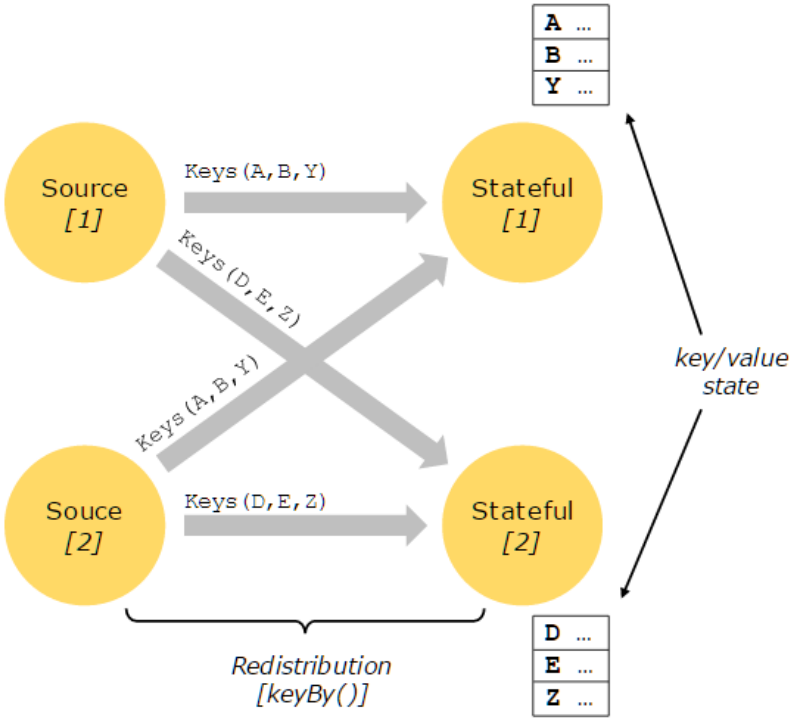
Checkpointing in Flink

- After that, it resumes processing records from all input streams, processing records from the input buffers before processing the records from the streams.



Stateful Operations

- When operators contain any form of *state*, this state must be part of the snapshots as well. Operator state comes in different forms
  - **User-defined state:** This is state that is created and modified directly by the transformation functions (like map() or filter()).
  - **System state:** This state refers to data buffers that are part of the operator's computation. A typical example for this state are the window buffers

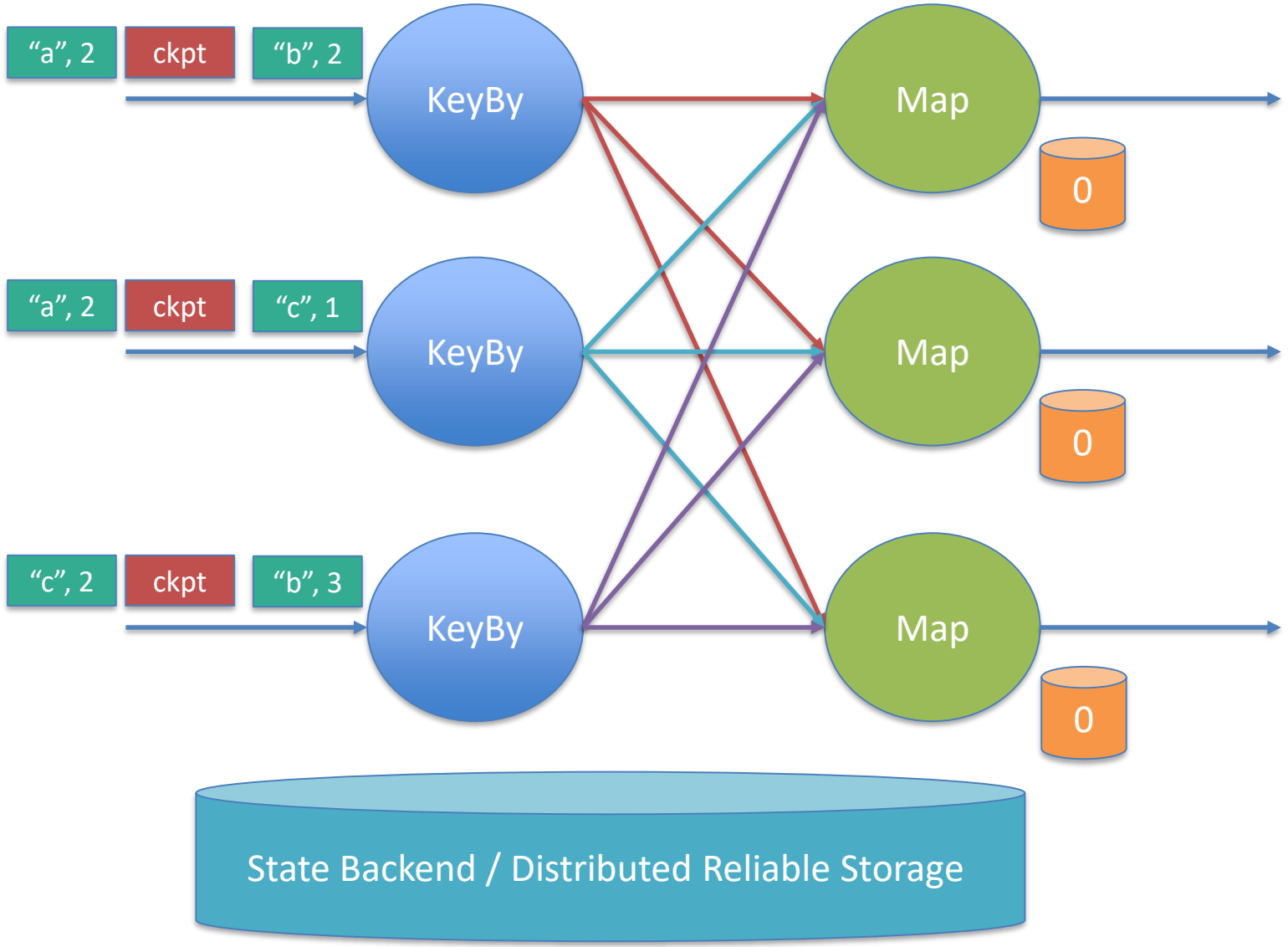


Stateful Operations

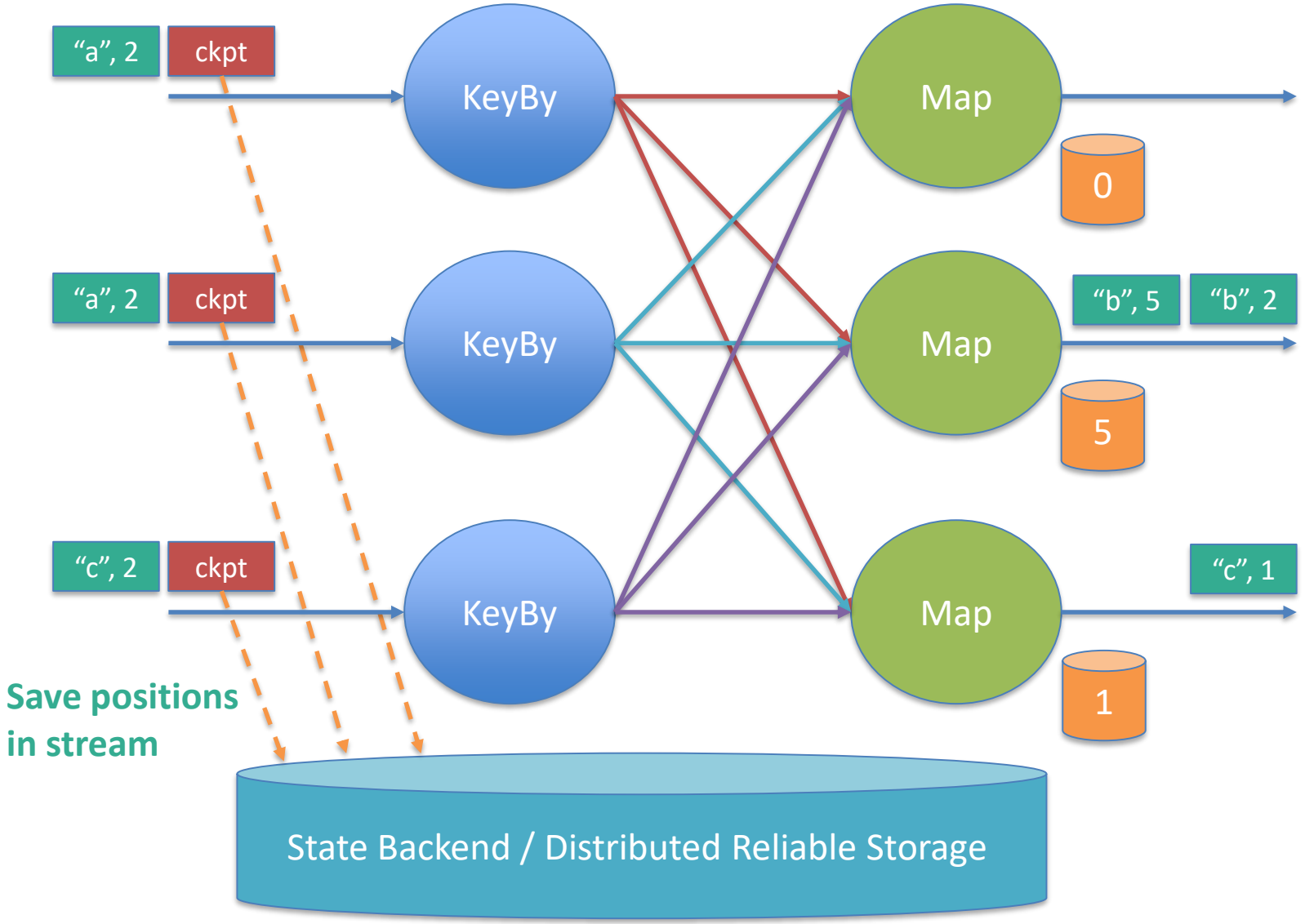
- Flink offers the user facilities to define state.
- An example:

```
val stream: DataStream[(String, Int)] = ...

val counts: DataStream[(String, Int)] = stream
  .keyBy(record => record._1)
  .mapWithState((in: (String, Int), count: Option[Int]) =>
    count match {
      case Some(c) => ( (in._1, c + in._2), Some(c + in._2) )
      case None => ( (in._1, in._2), Some(in._2) )
    })
```

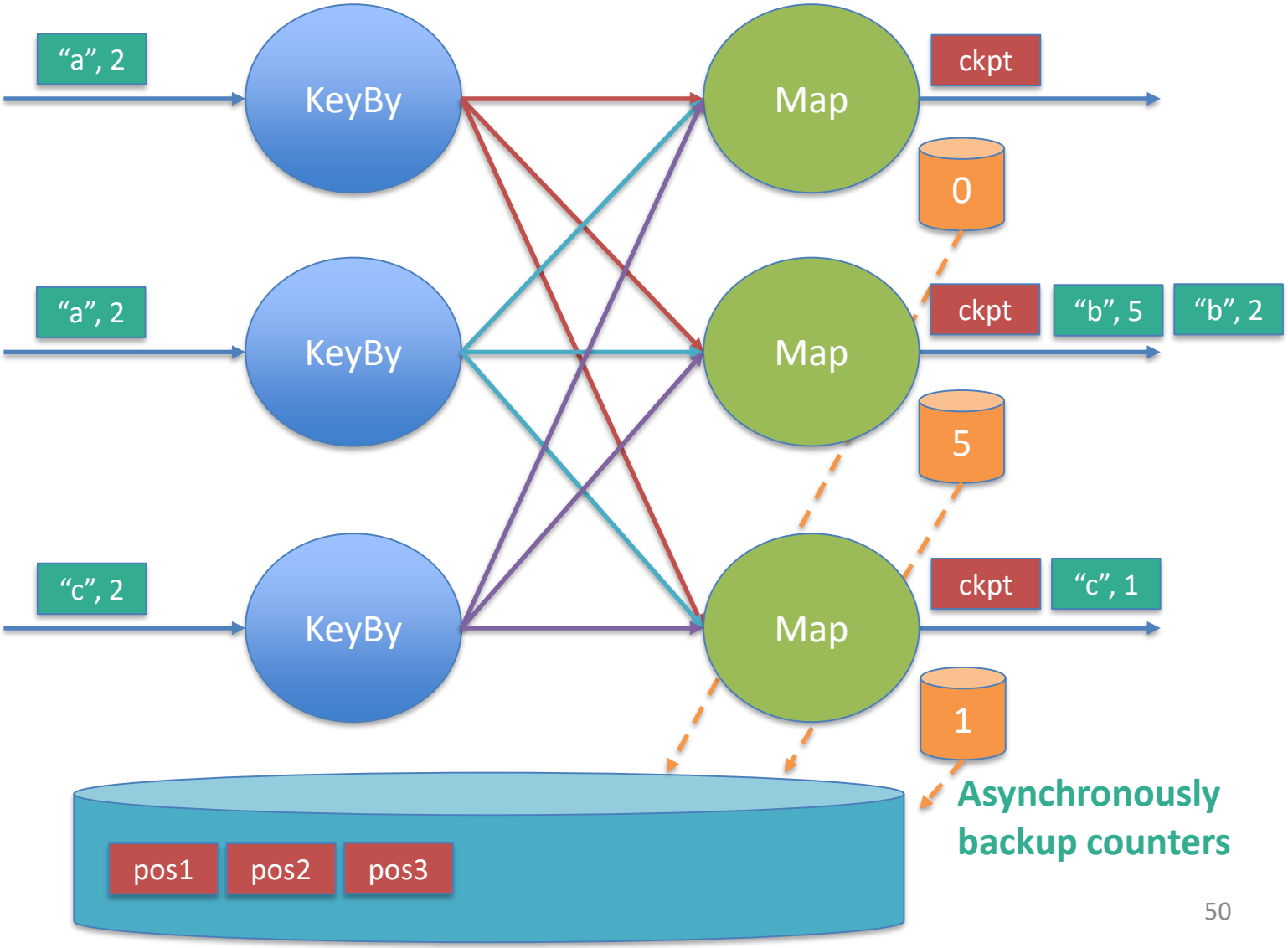
Stateful Operations



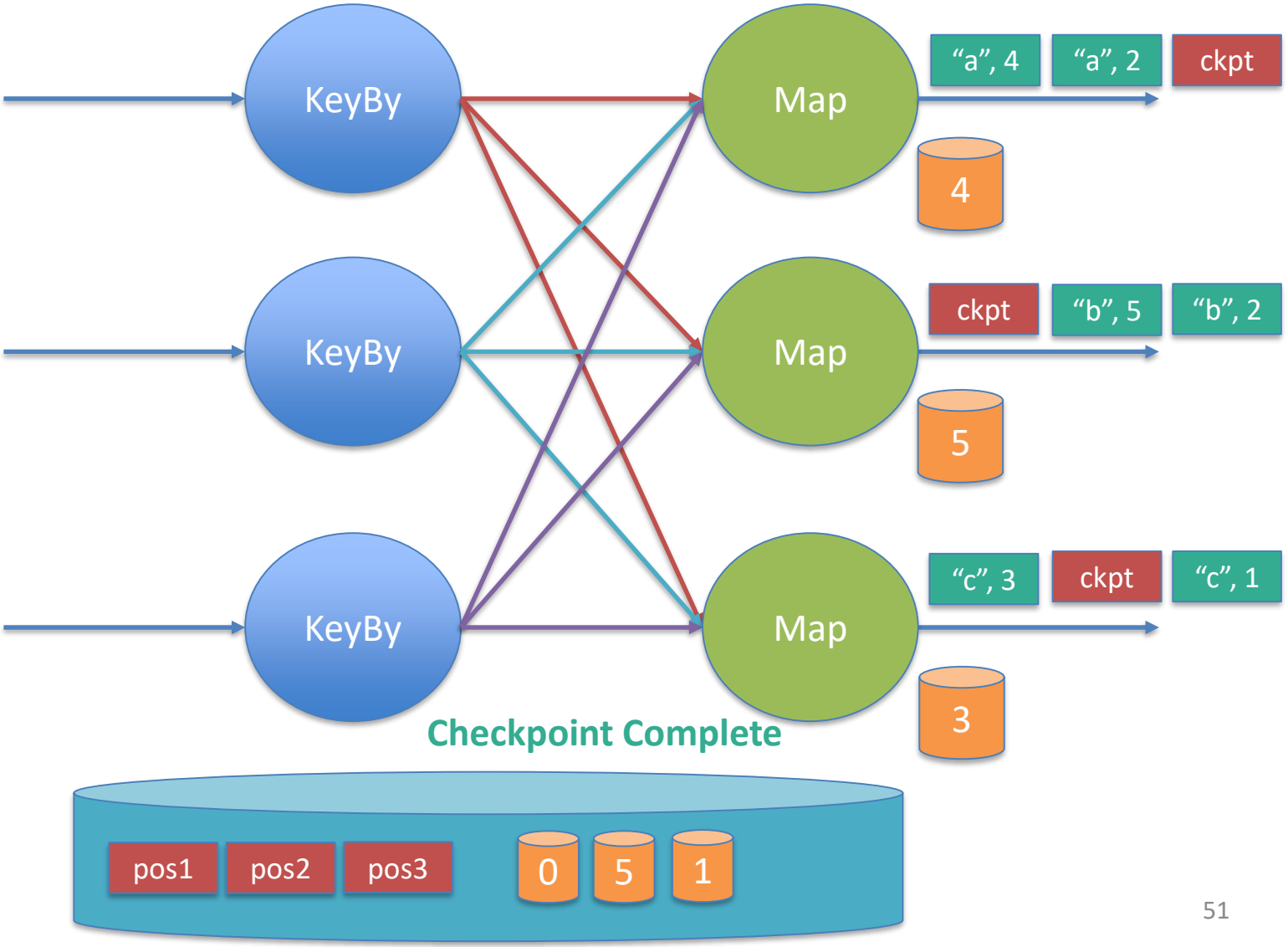
Stateful Operations



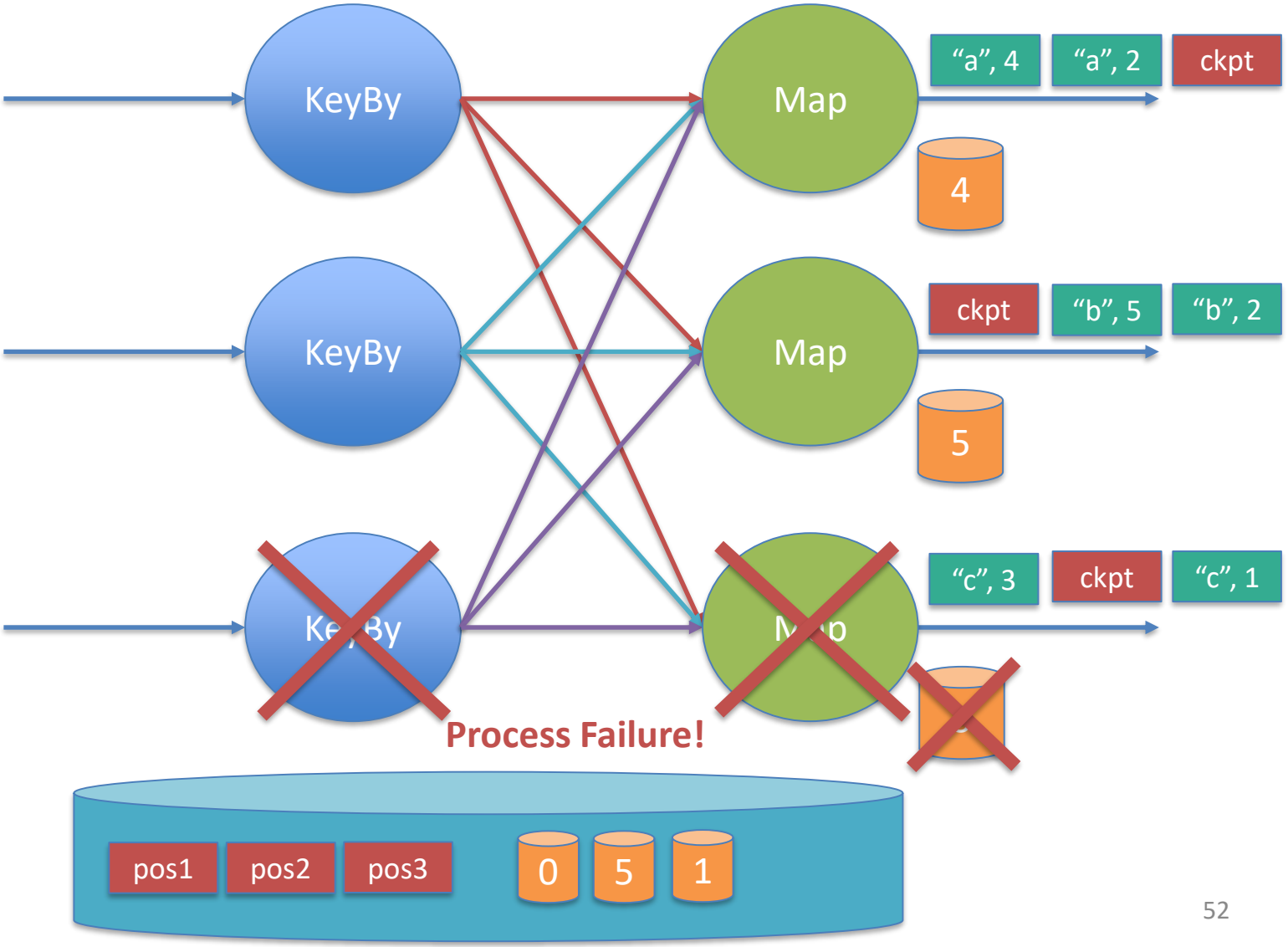
Stateful Operations



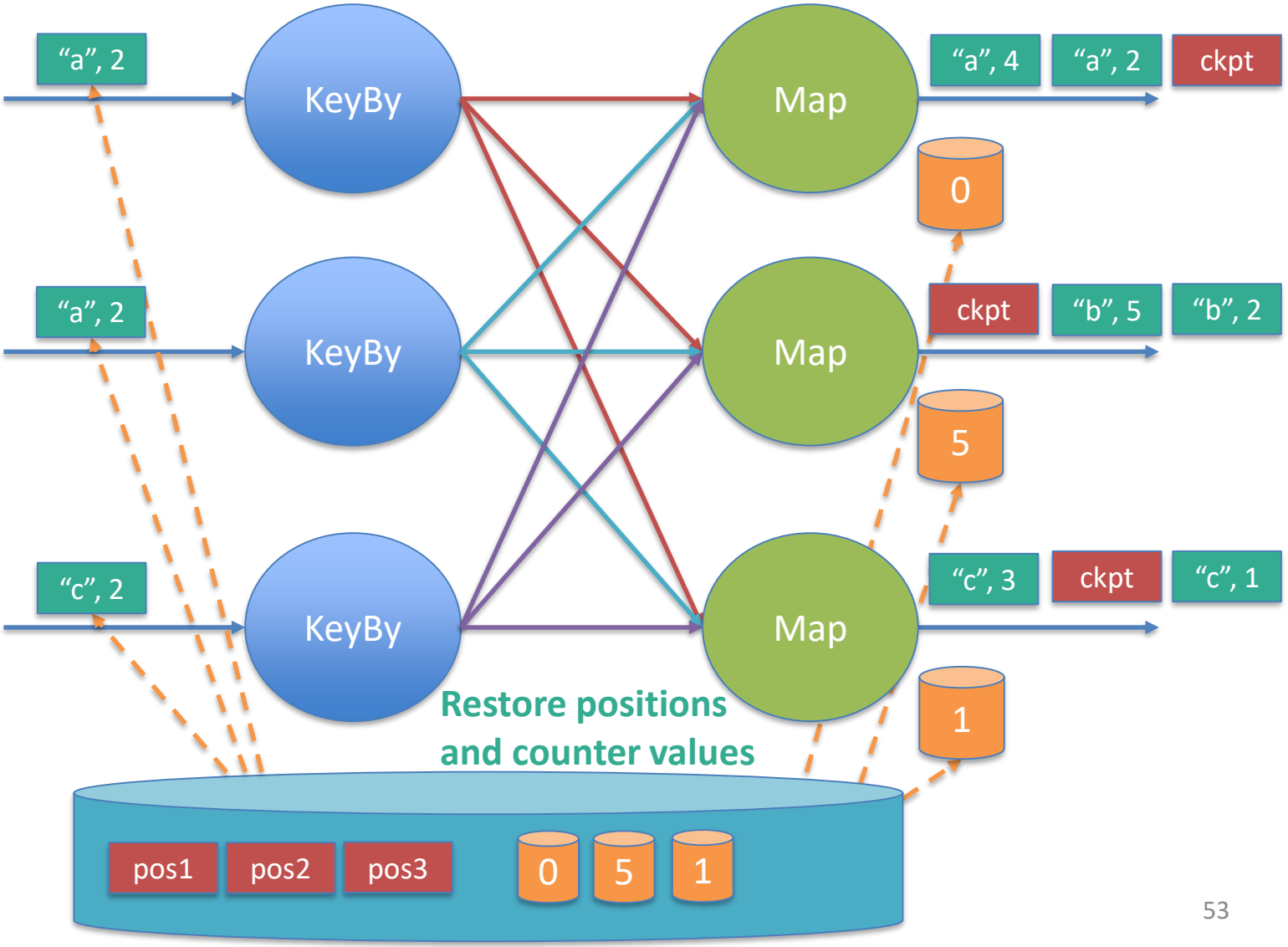
Stateful Operations



Stateful Operations



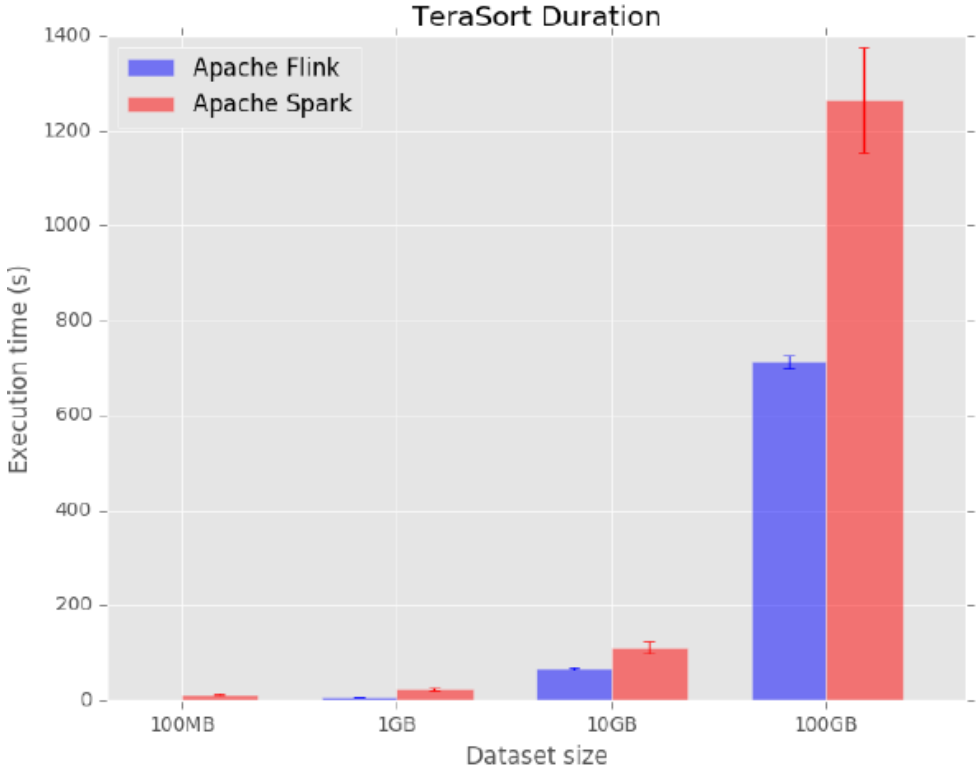
Stateful Operations



Flink Performance

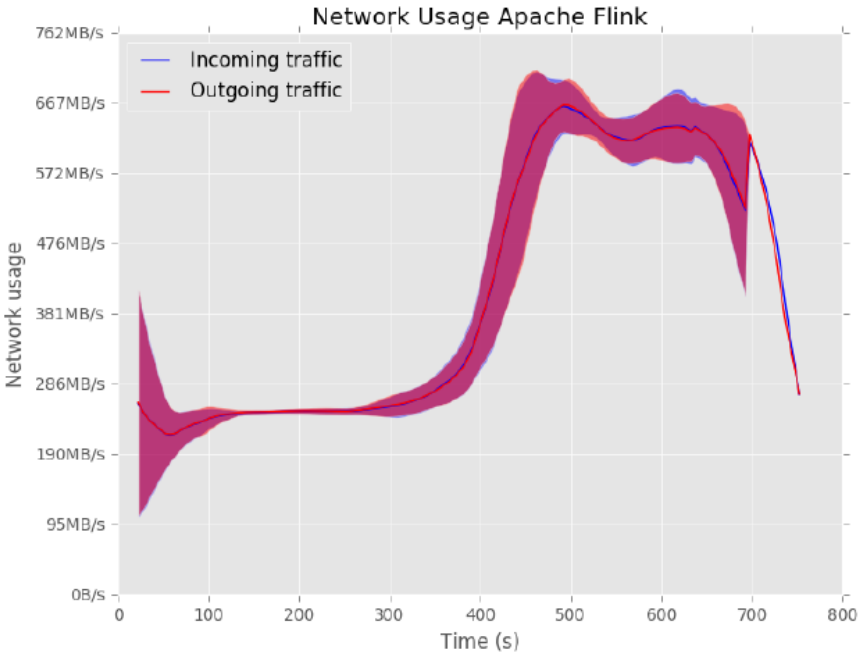
Batch API

- Apache Flink does the TeraSort job in about half of the time of Apache Spark. For very small cases, Apache Flink almost has no execution time while Apache Spark needs a significant amount of execution time to complete the job.

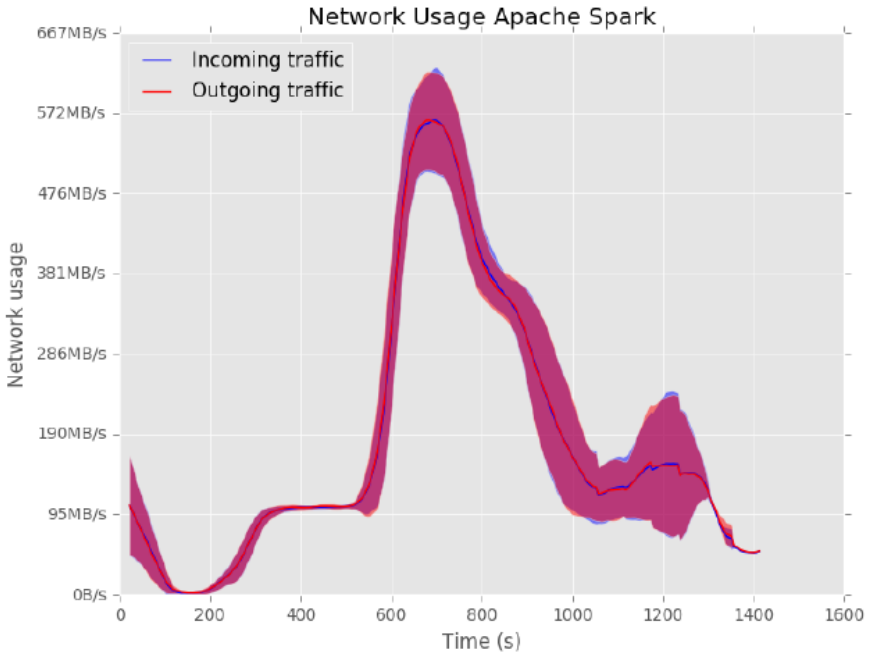


Batch API

- Apache Flink has about a constant rate of incoming and outgoing network traffic and Apache Spark does not have this constant rate of incoming and outgoing network traffic.



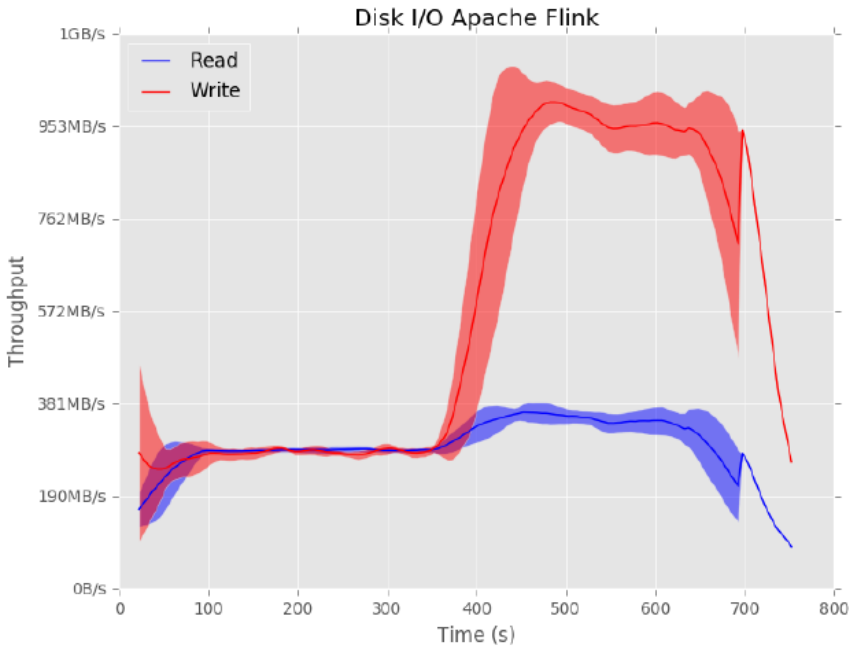
(a) Apache Flink.



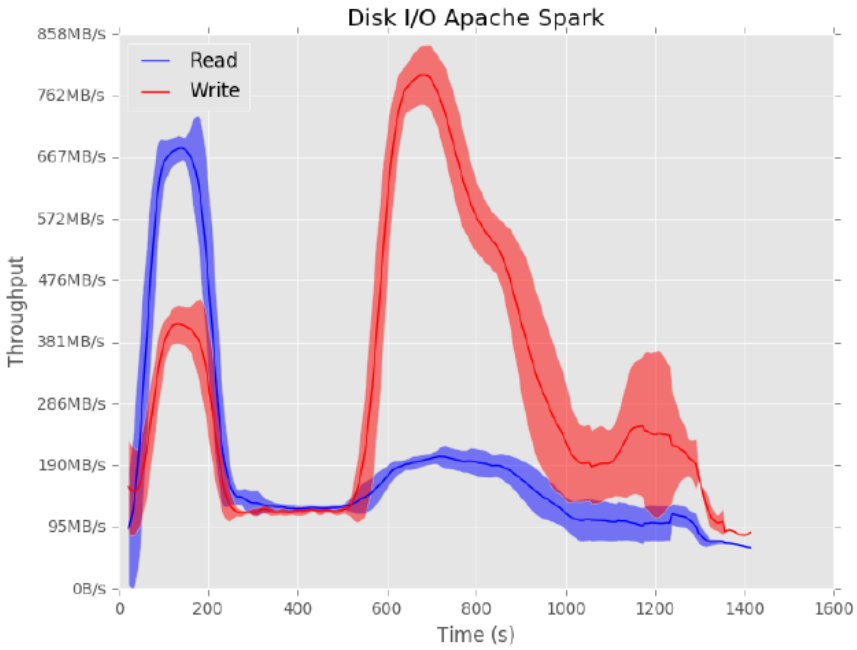
(b) Apache Spark.

Batch API

- The behaviour of the disk also reflects to behaviour of the network.



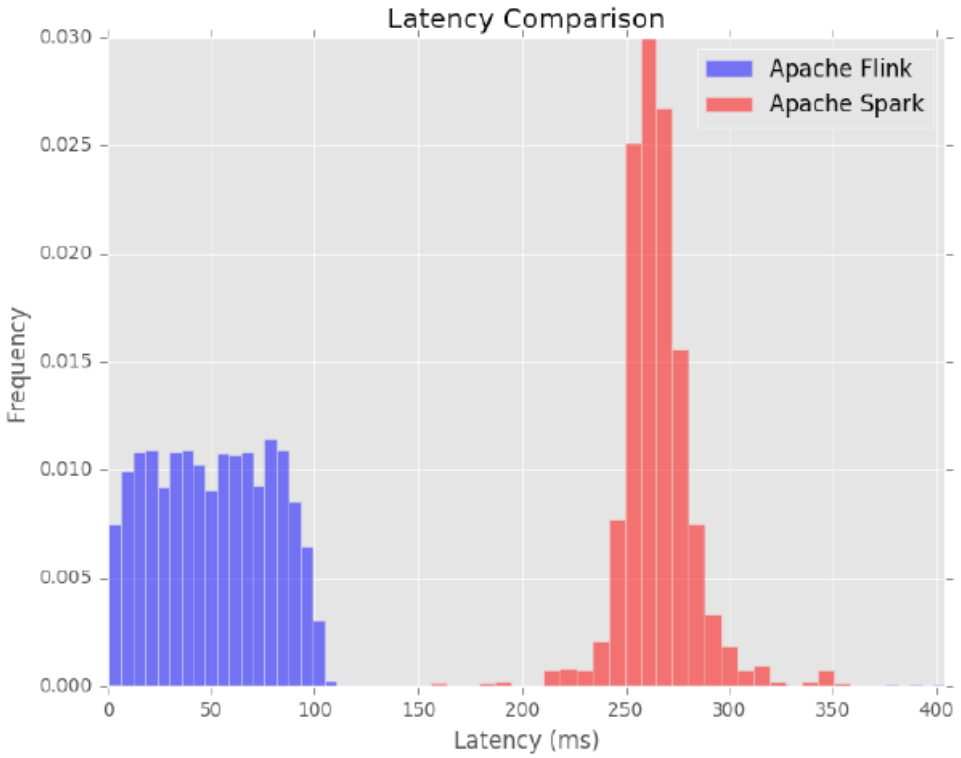
(a) Apache Flink.



(b) Apache Spark.

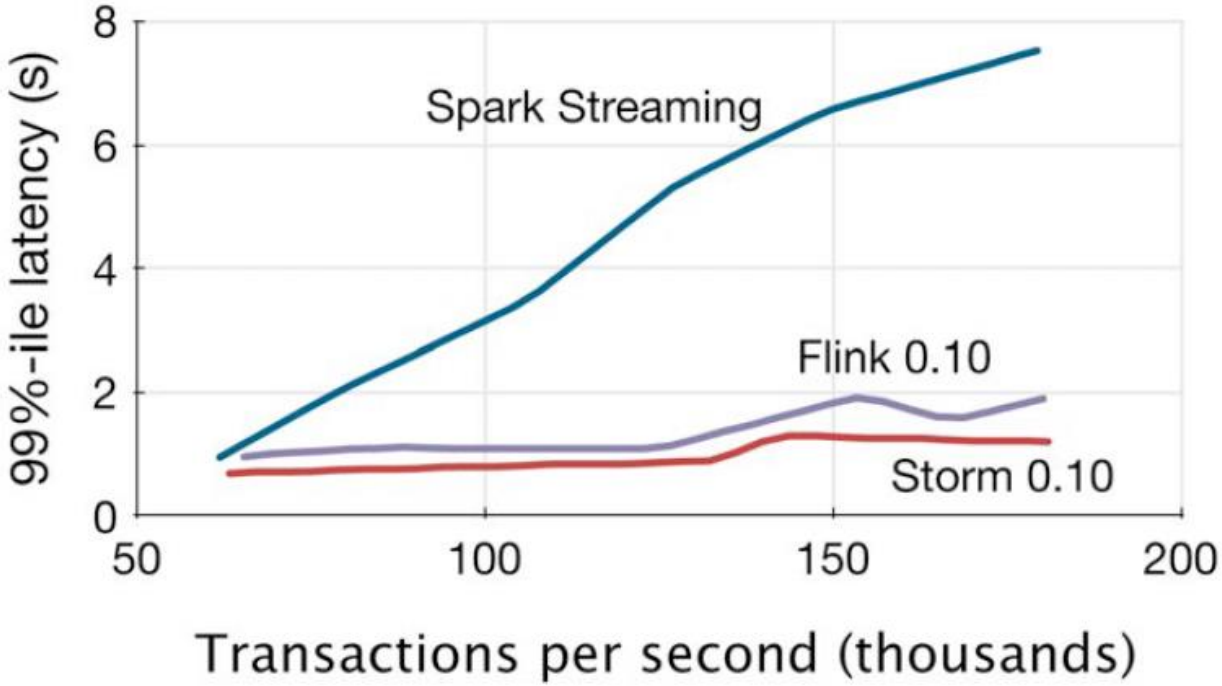
Stream API

- The fact that Apache Flink is fundamentally based on data streams is clearly reflected. The mean latency of Apache Flink is 54ms (with a standard deviation of 50ms) while the latency of Apache Spark is centered around 274ms (with a standard deviation of 65ms).



Stream API

- Spark Streaming suffered from a throughput-latency tradeoff. As batches increase in size, latency also increases. If batches are kept small to improve latency, throughput decreases. Storm and Flink can both sustain low latency as throughput increases.



Conclusion

- Flink is an open-source framework for distributed stream processing that:
  - Provides results that are **accurate**, even in the case of out-of-order or late-arriving data.
  - Is **stateful and fault-tolerant** and can seamlessly recover from failures while maintaining exactly-once application state.
  - Performs at **large scale**, running on thousands of nodes with very good throughput and latency characteristics.
  - Guarantees **exactly-once semantics for stateful computations**.
  - Supports stream processing and windowing with **event time semantics**.
  - Supports **flexible windowing** based on time, count, or sessions in addition to data-driven windows.
  - Flink's **fault tolerance is lightweight** and allows the system to maintain high throughput rates and provide exactly-once consistency guarantees at the same time.
  - Is capable of **high throughput and low latency** (processing lots of data quickly).

References

- Apache Flink Documentation
  - https://ci.apache.org/projects/flink/flink-docs-release-1.2/
- Introduction to Apache Flink
  - https://www.mapr.com/introduction-to-apache-flink
- Auto-Parallelizing Stateful Distributed Streaming Applications
  - http://dl.acm.org/citation.cfm?id=2370826
- The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing
  - https://research.google.com/pubs/pub43864.html
- Lightweight Asynchronous Snapshots for Distributed Dataflows
  - http://arxiv.org/abs/1506.08603
- Apache Flink: Distributed Stream Data Processing
  - https://cds.cern.ch/record/2208322/files/report.pdf