Processing Streams with Apache Flink

Lorenzo Affetti (lorenzo.affetti@gmail.com)

Stream Processing

Processing Data

Essentially it can be broken in 3 simple phases:

- Take input from sources
- Transform it
- Output to sink systems

Examples:

- Events from sensors -> mean temperature in the last 5 milliseconds.
- Clicks on web pages -> least visited page.
- Database of facts about product sales -> top requested products in the last month

Input Data

- **Unbounded Datasets**:
  - Infinite datasets appended continuously (e.g. machine log data, measurements from a sensor, push notifications);
- ***Bounded Datasets***:
  - Finite and unchanging datasets (e.g. dump of a database, .csv file containing blood measurements for patients)

Expressing the Transformation

- The transformation step can be done with different techniques, but, in its essence it carries out some computation.
- The computation can be expressed as a procedure, a graph of computation, or a SQL query.

Sinks

- Sinks could be any of:
  - Database (Postgres, MongoDB, Cassandra);
  - Files or S3;
  - Sockets/messaging systems (Kafka);
  - *Another stage of the processing pipeline*.

Why Is SP a Thing?

Breaking the ice:
Let’s define *stream processing* with our words and see why it is something different from “normal” data processing.

Why Is SP a Thing?

That’s right:

- **streams and records**/events as first class citizens;
- dealing with **unbounded data.**

What does it mean?

- Produce output **as new records are ingested** and **consider the evolution of data (time)**.
- What is the mean of **infinite integers**?
- What if we **fail**? Can we replay?

Let’s compare stream processing with “old” style processing.

How to process data (1/4)

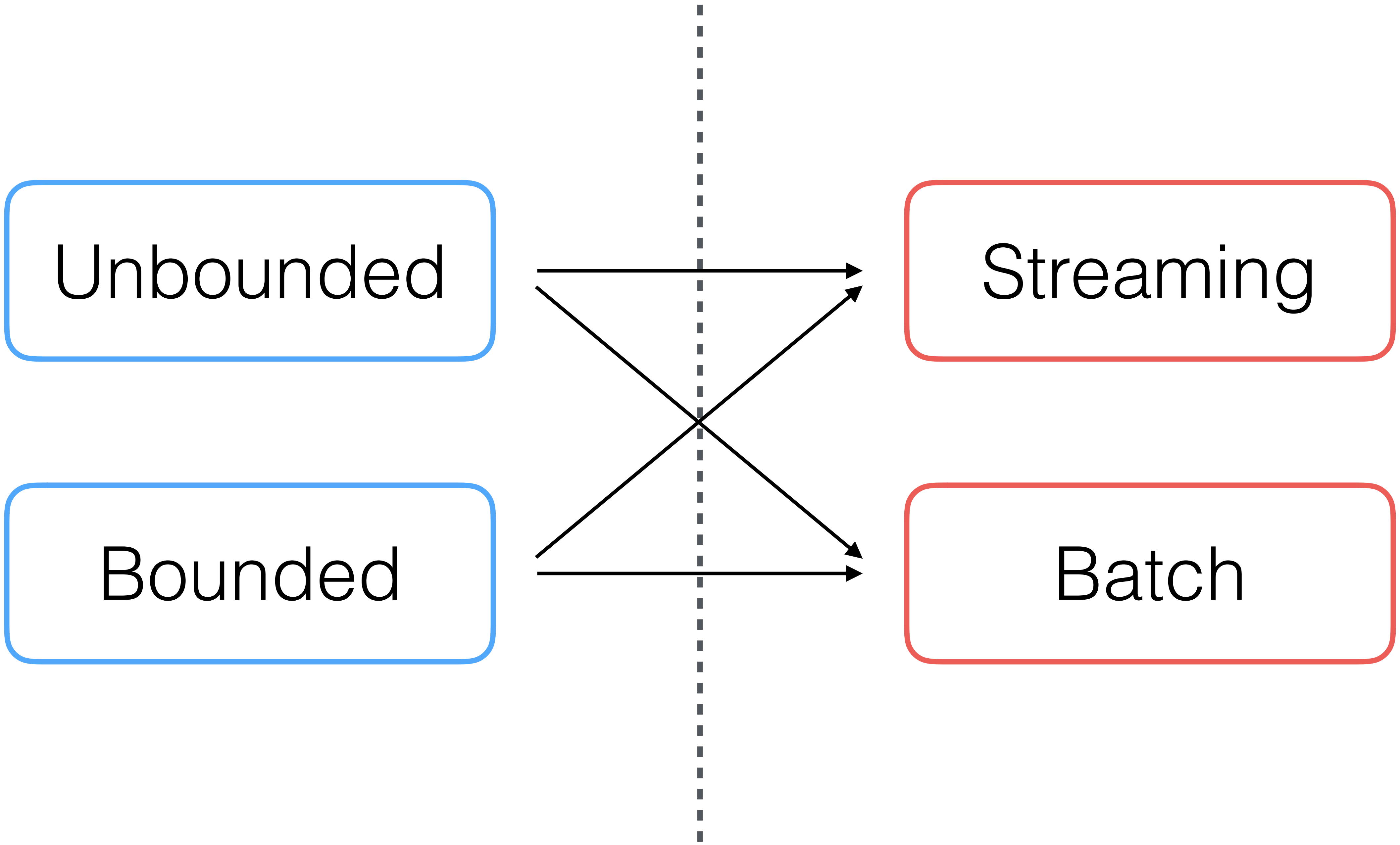
The different techniques are (historically):

- **Batch**: processing that runs to completeness in a finite amount of time.
- **Streaming**: possibly infinite processing that executes continuously as long as data is produced.

How to process data (2/4)

- The differences between batch and stream processing are not in their model of computation, but they are intimately connected to the inherent nature of the sources.
- Moreover, batch processing is not designed to transform records (react to events) as they come.
- Batch processing is not interested with the *evolution of the input* (e.g. computing a rolling or windowed average), but it uses the input to compute some final result.
- However, **a Stream Processor (SP) can always mimic a batch processor by slicing streams into time windows** (more on this later) and perform the computation using the windowed records as a finite input.
- Indeed, **batch processing is a specialised case of stream processing**.
- Obviously, some sources fit best some processing method and can thus take leverage of their nature, you can figure out which one.

How to process data (3/4)

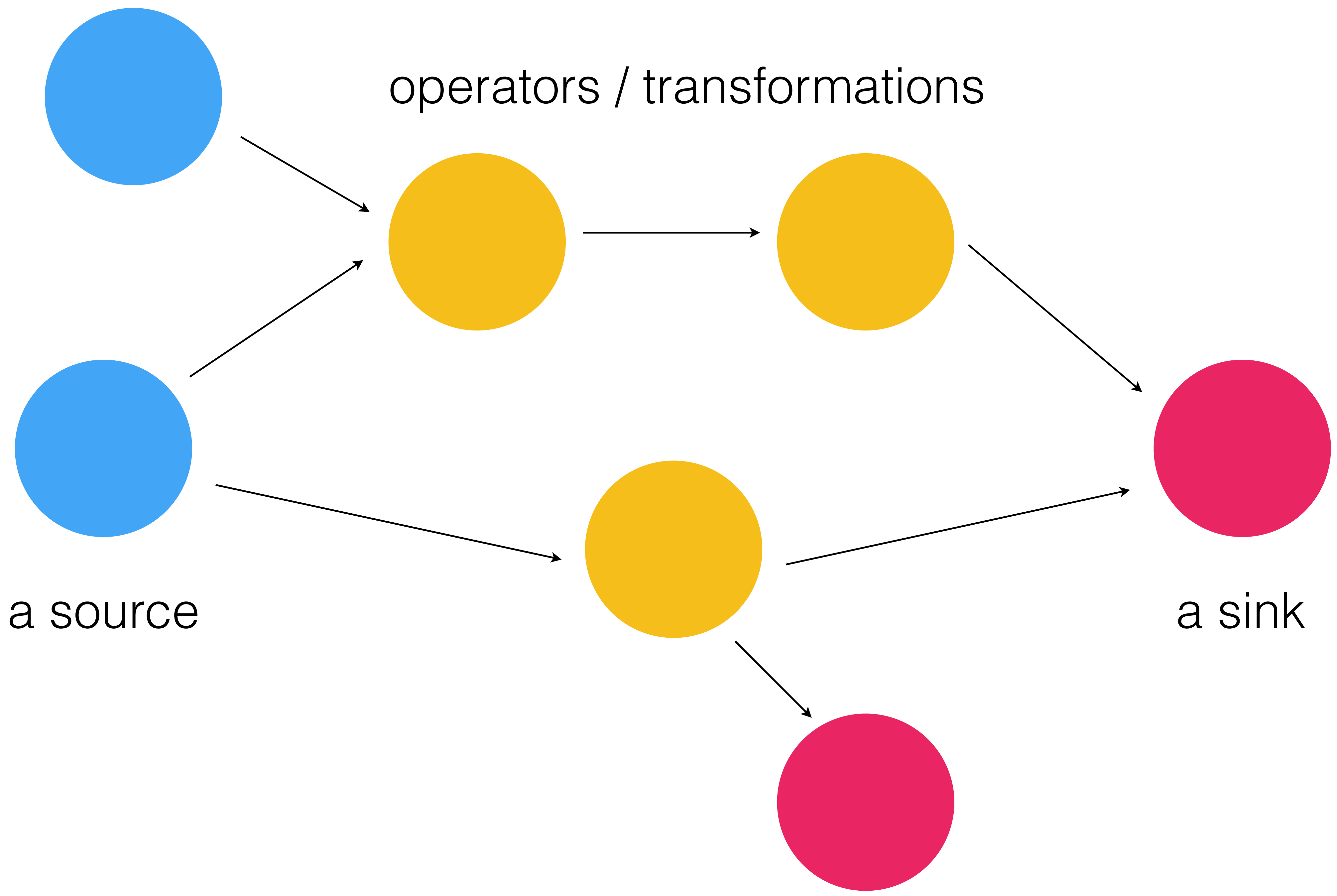


How to process data (4/4)

The difference in knowing that the input is finite impacts:

- Managing time (out-of-order events)
- Fault-tolerance

Describing Computation: the Graph of Computation (aka topology)



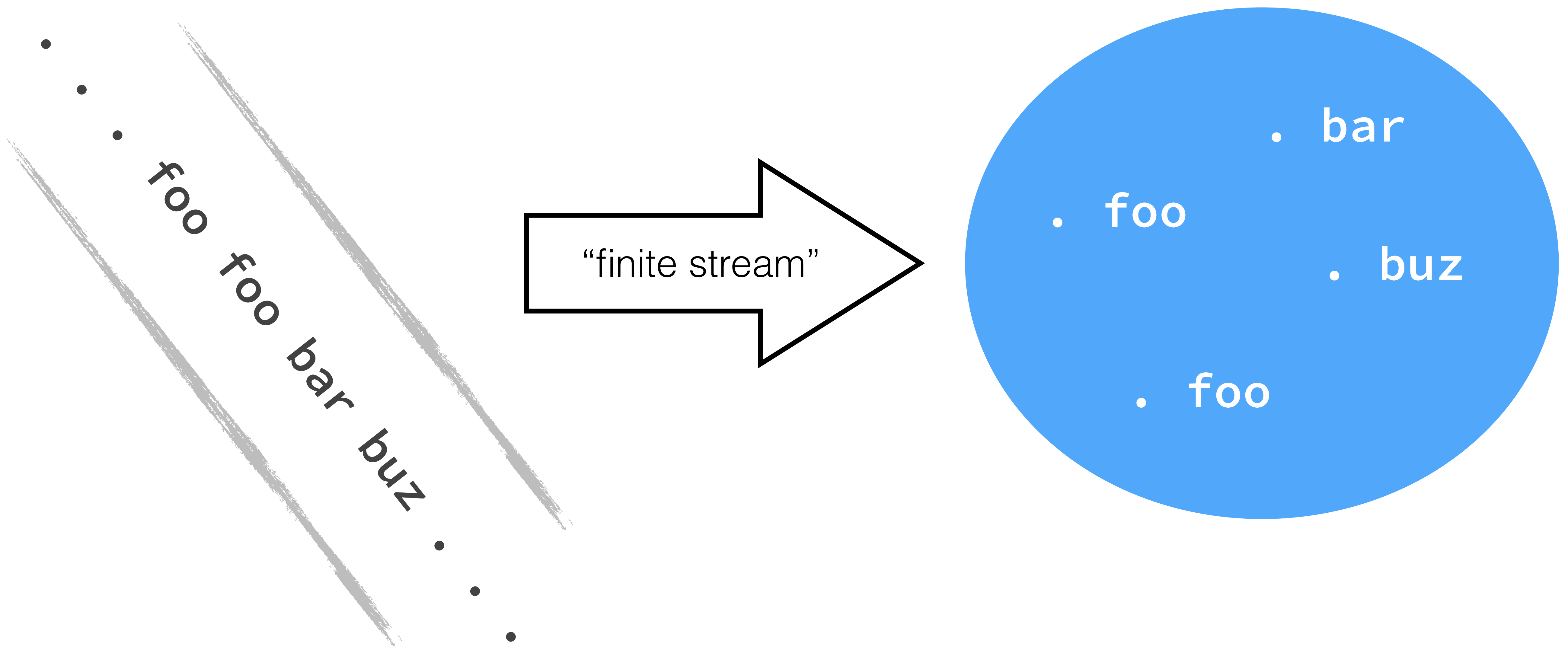
Stream Processing —> Flink

Apache Flink

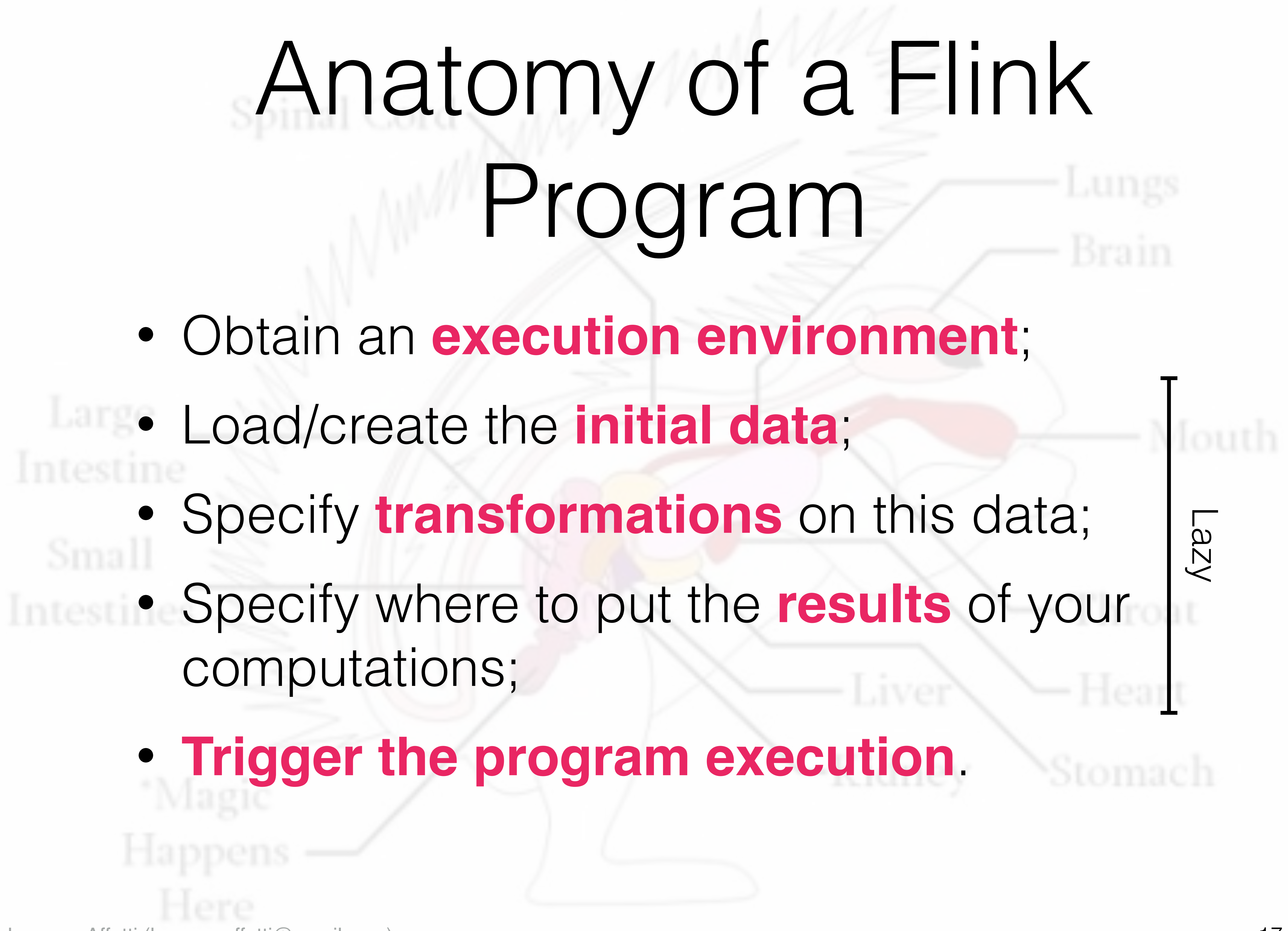
- An **open-source** Stream Processor (SP)
- Leverages every concept we said before: it offers a unified approach to batch and stream processing
- It provides exactly-once processing guarantee in case of failure
- Provides first class state management
- Distributed system management

DataStream & DataSet

Link offers abstractions for unbounded and bounded data



Anatomy of a Flink Program



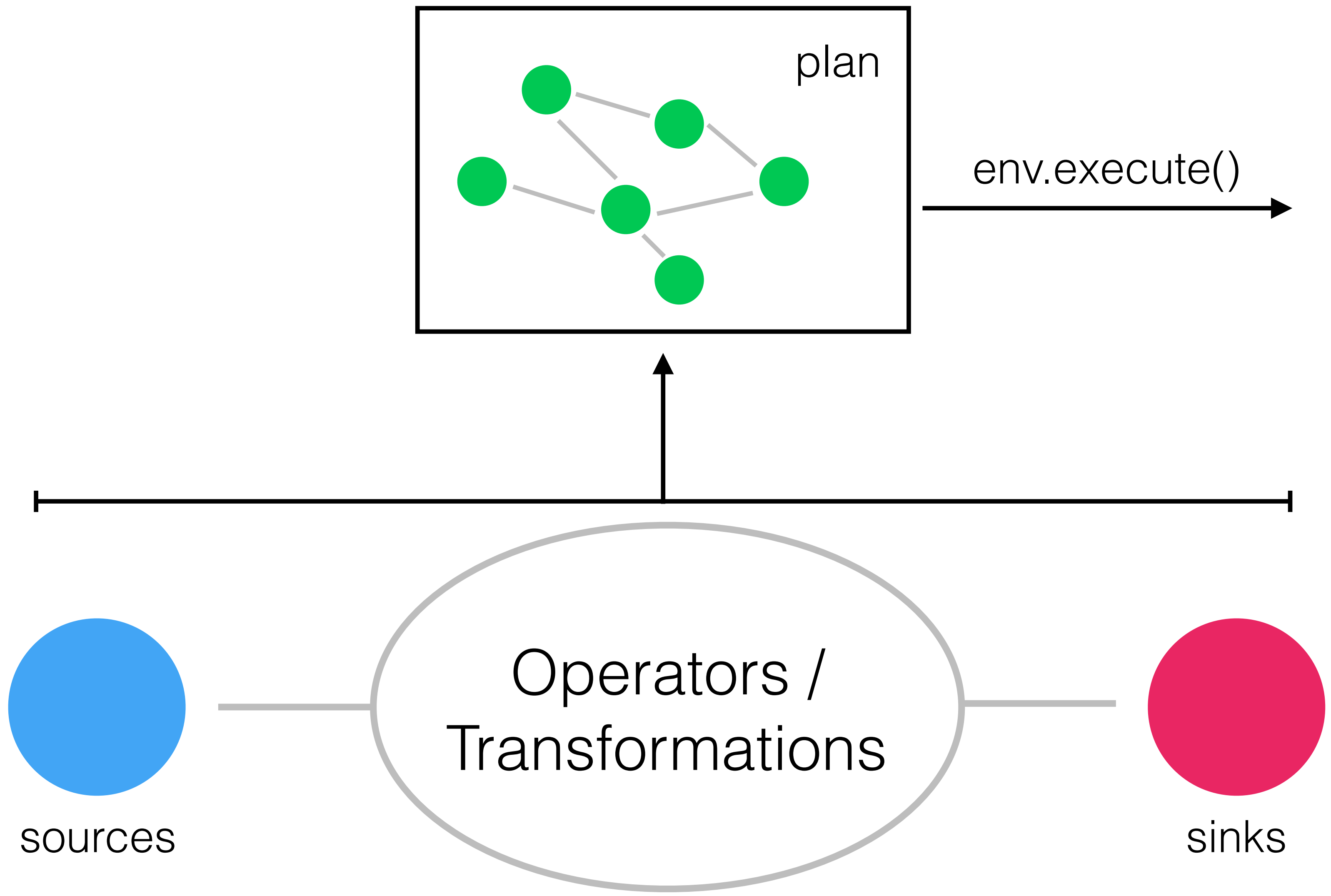
- Obtain an **execution environment**;
- Load/create the **initial data**;
- Specify **transformations** on this data;
- Specify where to put the **results** of your computations;

Lazy

- **Trigger the program execution**.

Lazy Evaluation

- Flink optimizes the graph of computation
- Only when computation is triggered the plan is sent to physical executors



Environment Setup

…

Transformations

Exercise

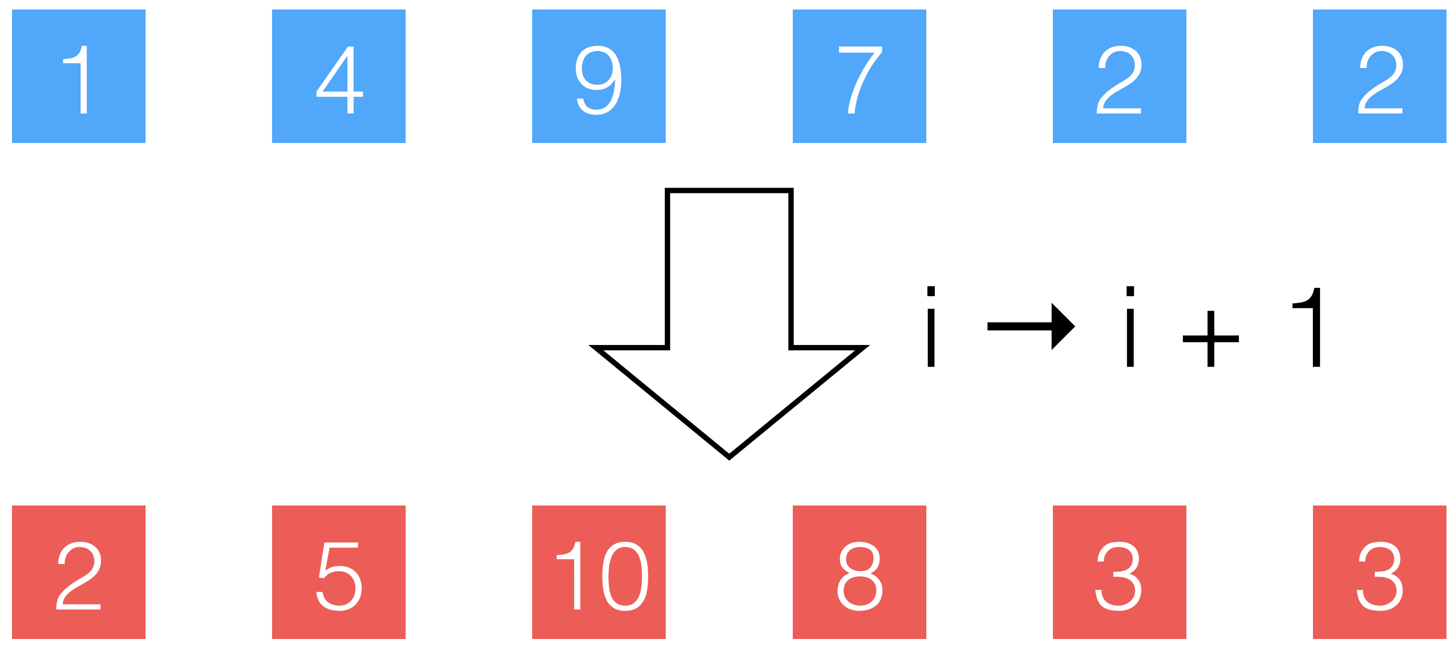
- Create a stream of a thousand random integers and print it.

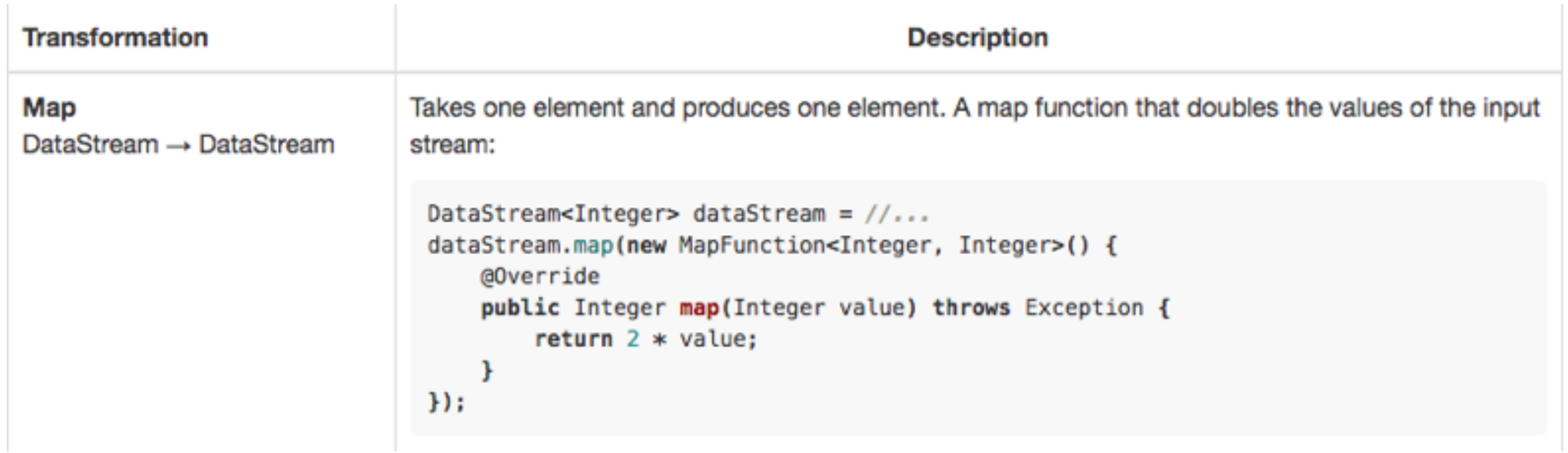
Exercise

- Create a stream from a socket source and print it.

Transformations

Map





Transformation	Description
Map DataStream → DataStream	Takes one element and produces one element. A map function that doubles the values of the input stream:

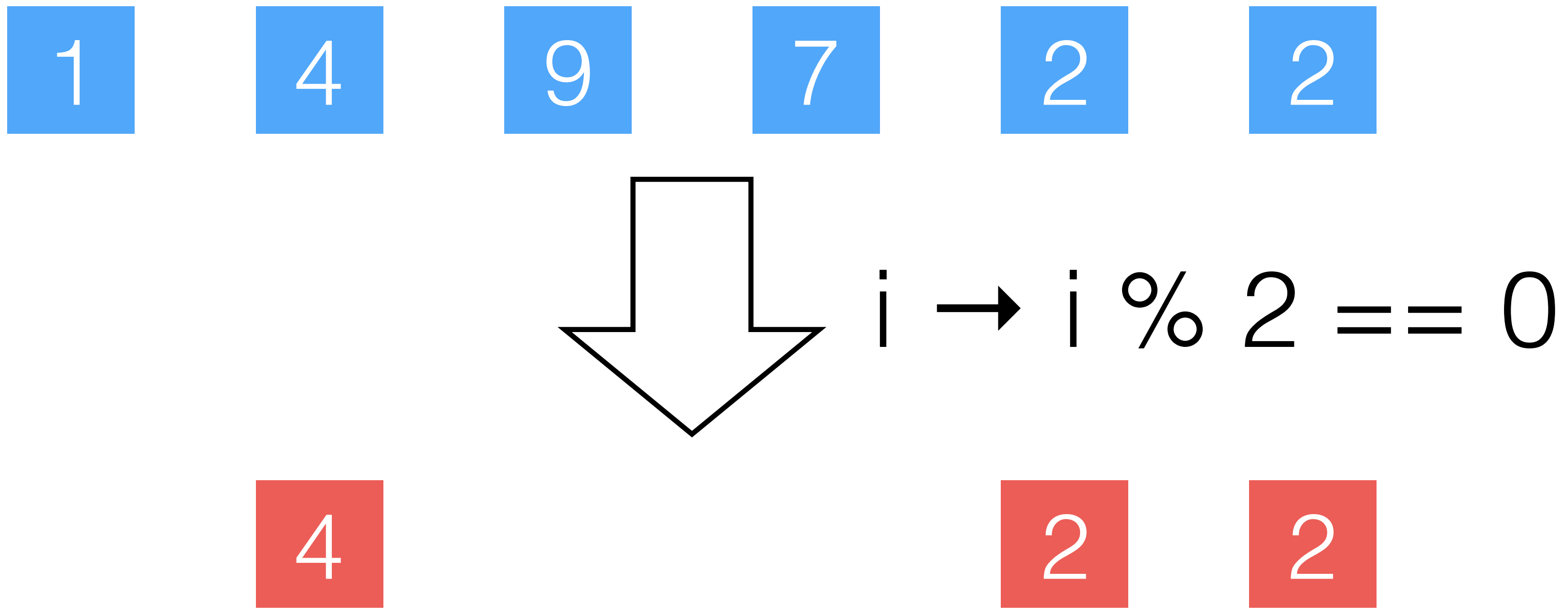
```
DataStream<Integer> dataStream = //...
dataStream.map(new MapFunction<Integer, Integer>() {
    @Override
    public Integer map(Integer value) throws Exception {
        return 2 * value;
    }
});
```

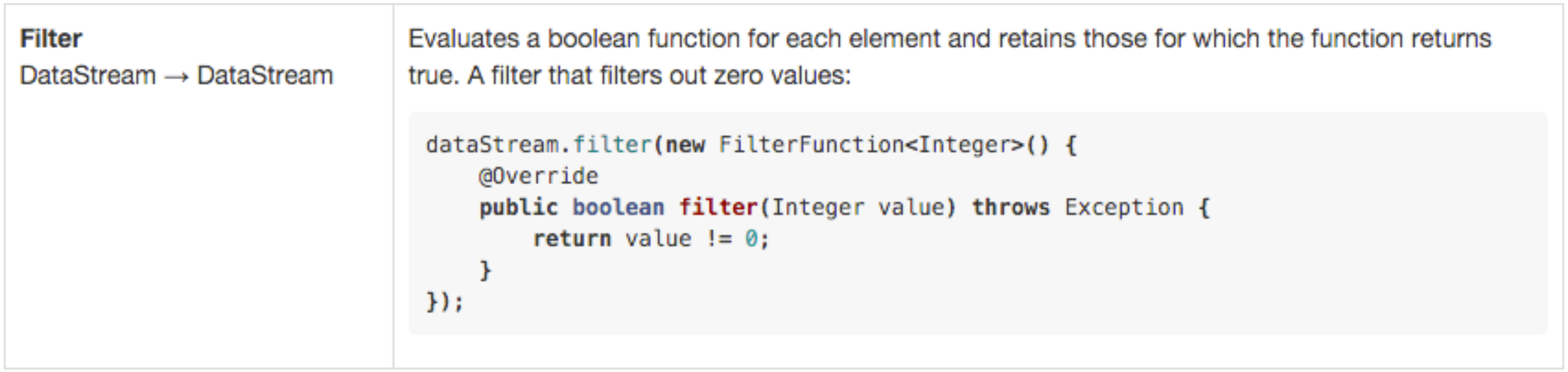
Exercise

- Take a stream of a thousand random integers;
- Generate a new stream where the *i-th* element is the square of the *i-th* element in the initial stream;
- Print the resulting stream.

Transformations

Filter





Filter DataStream → DataStream	Evaluates a boolean function for each element and retains those for which the function returns true. A filter that filters out zero values:

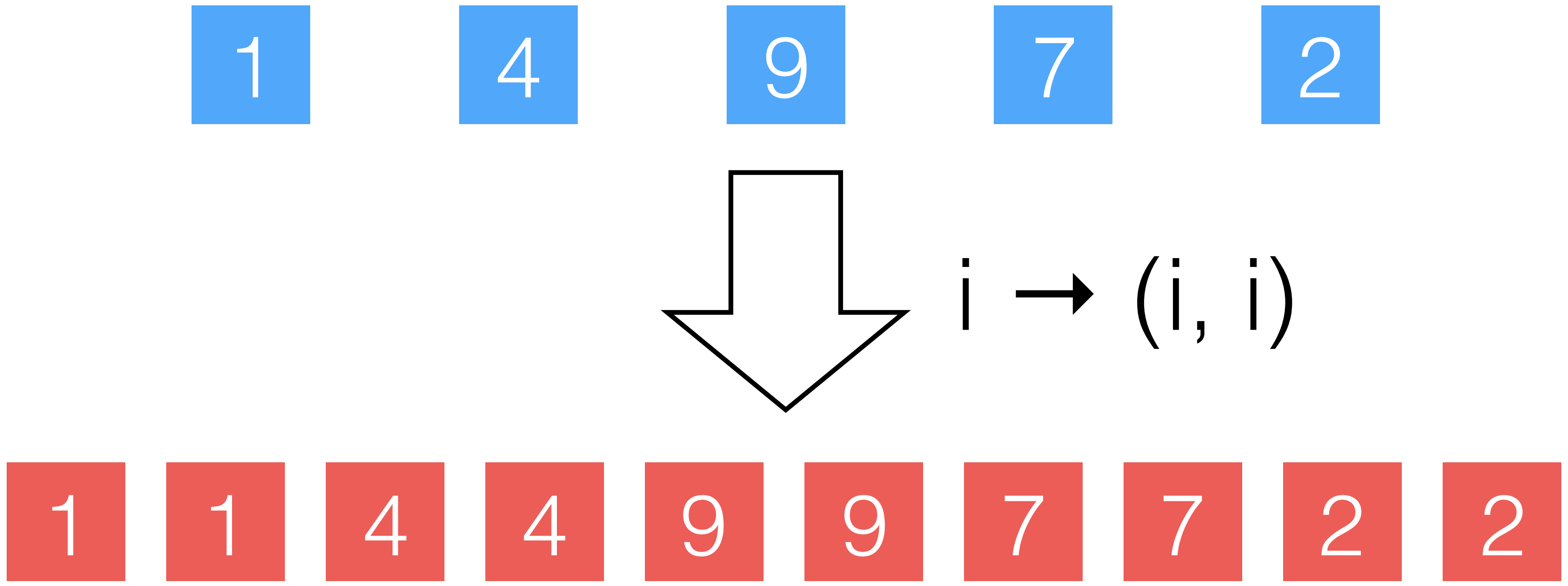
```
dataStream.filter(new FilterFunction<Integer>() {
    @Override
    public boolean filter(Integer value) throws Exception {
        return value != 0;
    }
});
```

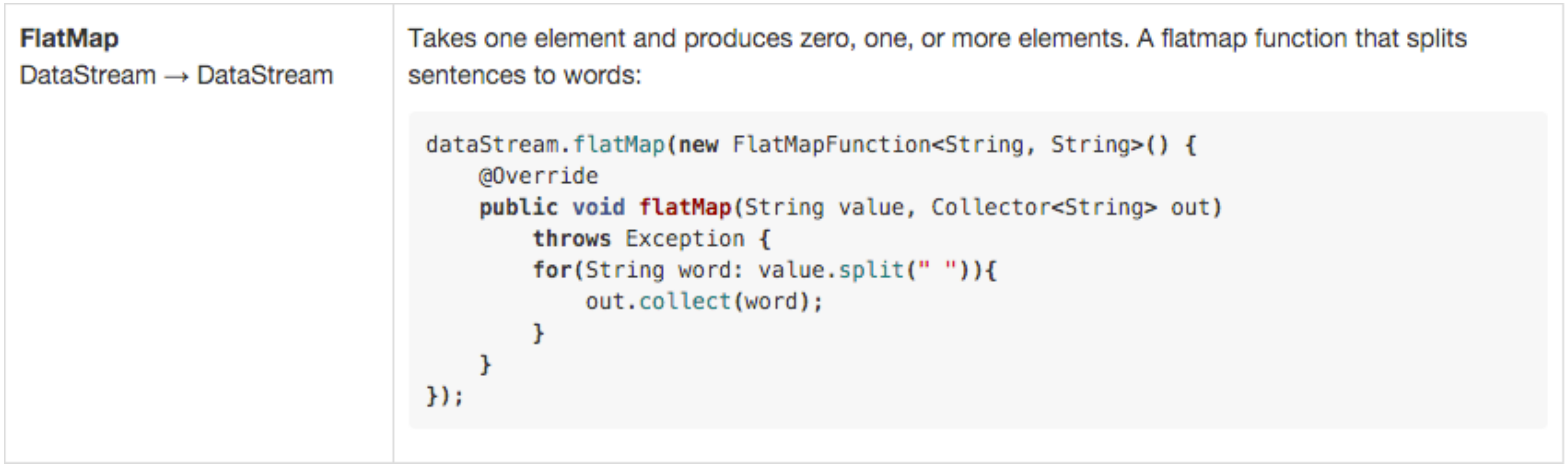
Exercise

- Take a stream of a thousand random integers;
- Generate a new stream where the *i-th* element is the string representation of the *i-th* element in the initial stream;
- Generate a new stream containing only the elements that contain the digit “4”;
- Print the resulting stream.

Transformations

FlatMap





FlatMap DataStream → DataStream	Takes one element and produces zero, one, or more elements. A flatmap function that splits sentences to words:

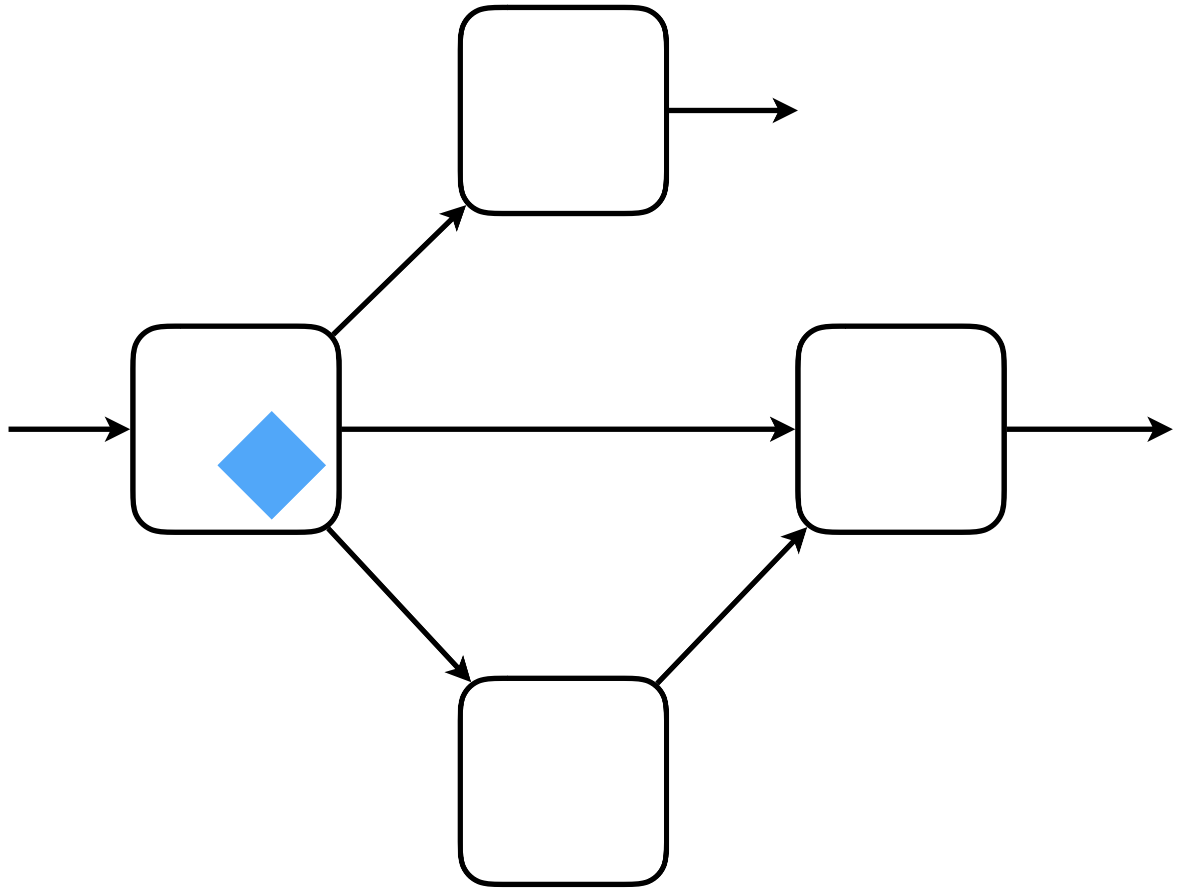
```
dataStream.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public void flatMap(String value, Collector<String> out)
        throws Exception {
        for(String word: value.split(" ")){
            out.collect(word);
        }
    }
});
```

Exercise

- Take a stream of a thousand random integers;
- Generate a new stream of digits (e.g. 1 42 9 —> 1 4 2 9);
- Print the resulting stream.

Stateful Operators

- Some operators need to retain **state**
- Think of examples



Exercise

- Implement a word counter.
- Try to increase the parallelism, what happens (we'll solve it in a minute)?

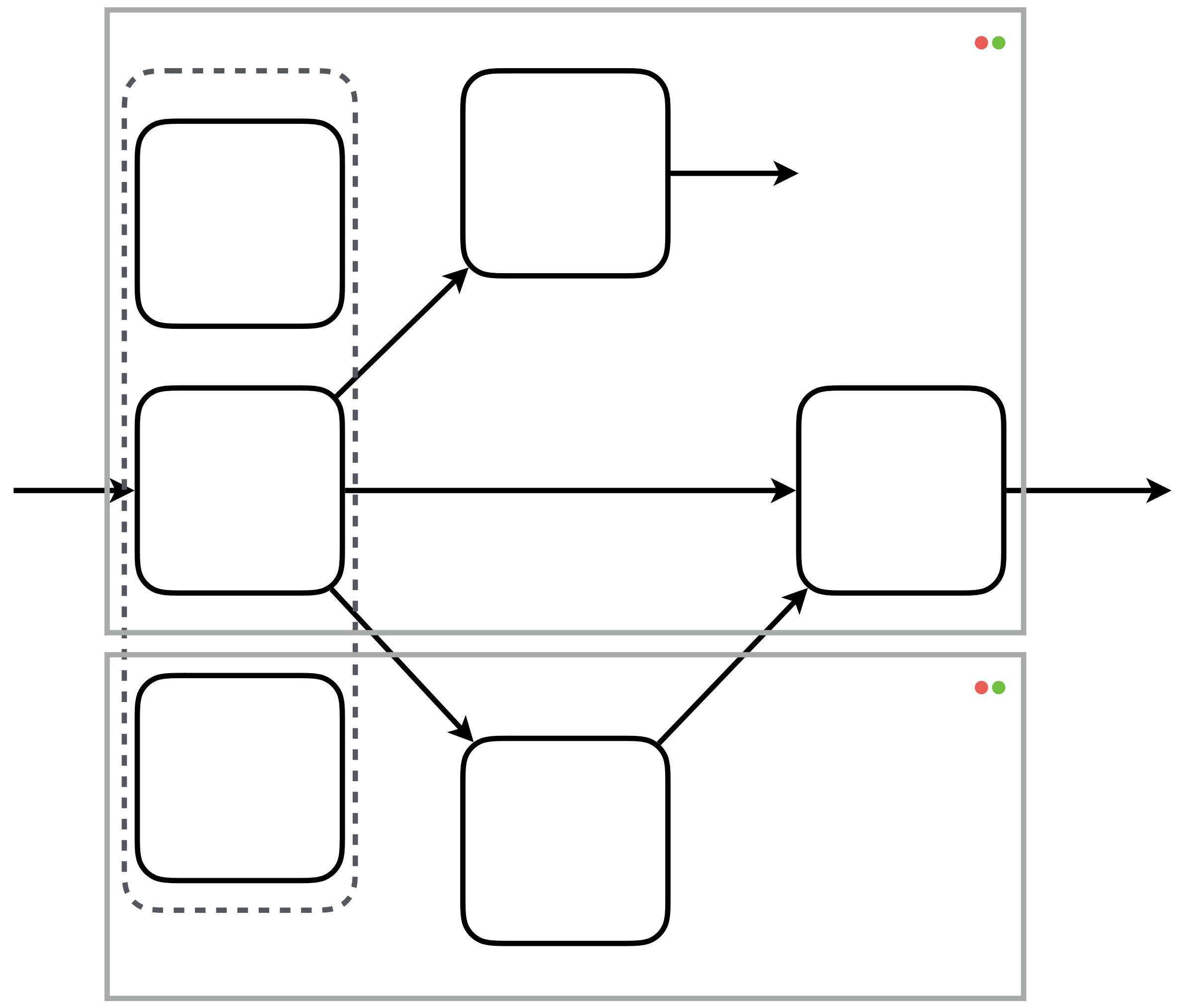
* sorry, no guarantees

Dataflow programming and Parallelism

a short* digression

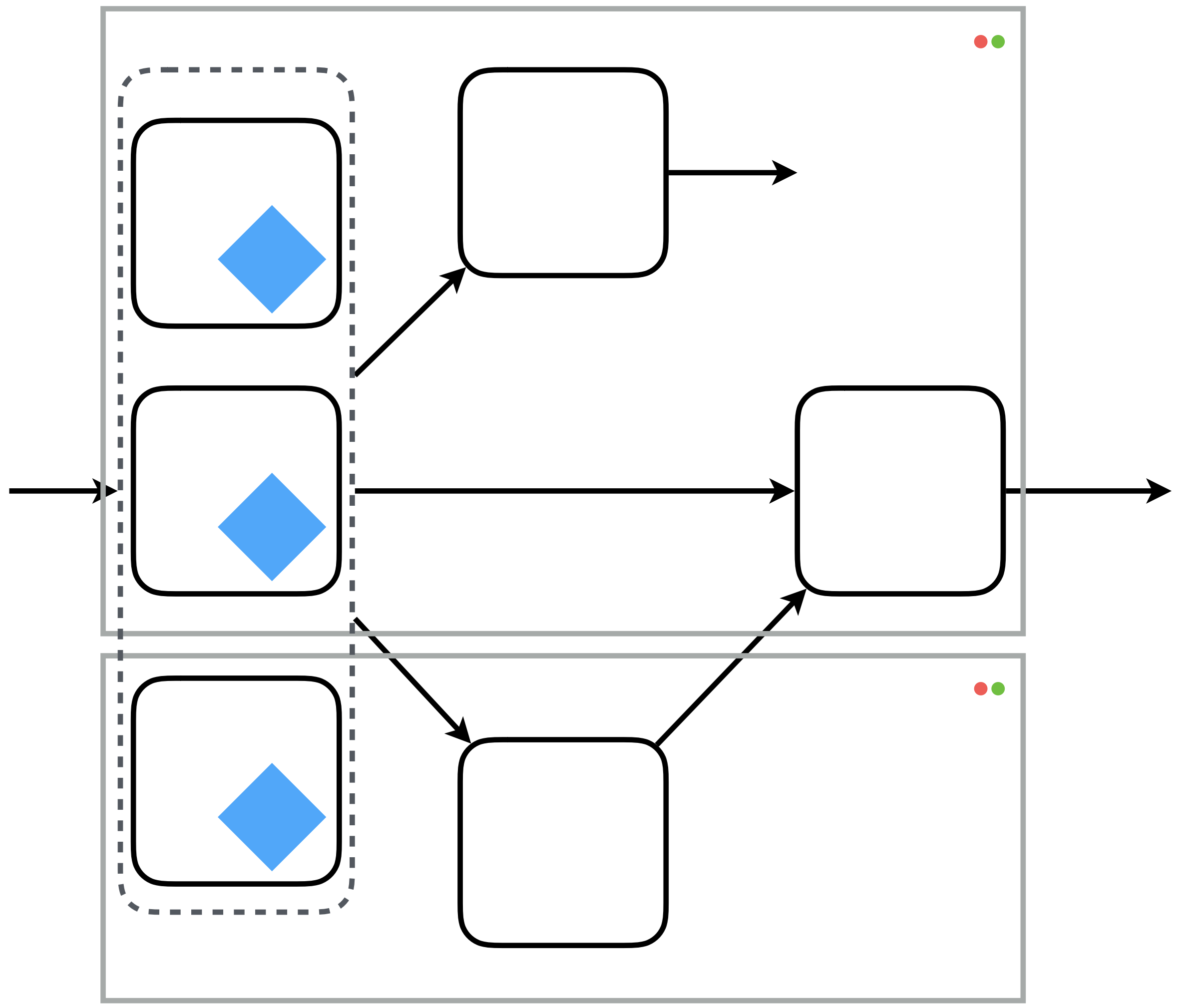
Task and Data Parallelism

- Every operator can run in parallel with the others (*task-parallelism*)
- Every operator can be replicated and process an independent partition of the stream (*data-parallelism*)



Stateful Operators

- We know that some operators need to store an internal state to perform computation (e.g. a rolling count, average, or maximum)
- If the **stateful operator** is replicated, every replica owns its local shard of the state.
- How to manage distributed state?

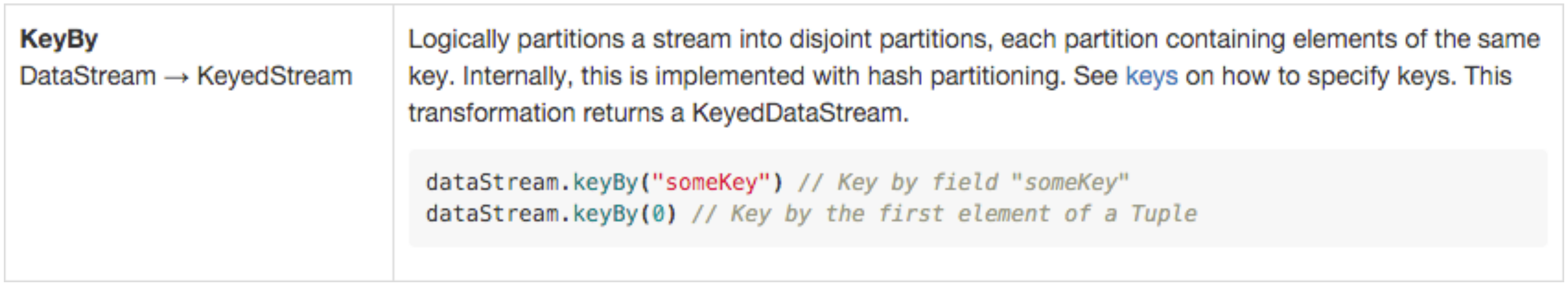


Transformations

KeyBy

With the keyBy transformation, you can specify how to **extract a key** from every element of the stream.

Two elements with the same key will be in the same partition and processed by the same replica.



KeyBy DataStream → KeyedStream	Logically partitions a stream into disjoint partitions, each partition containing elements of the same key. Internally, this is implemented with hash partitioning. See keys on how to specify keys. This transformation returns a KeyedDataStream.

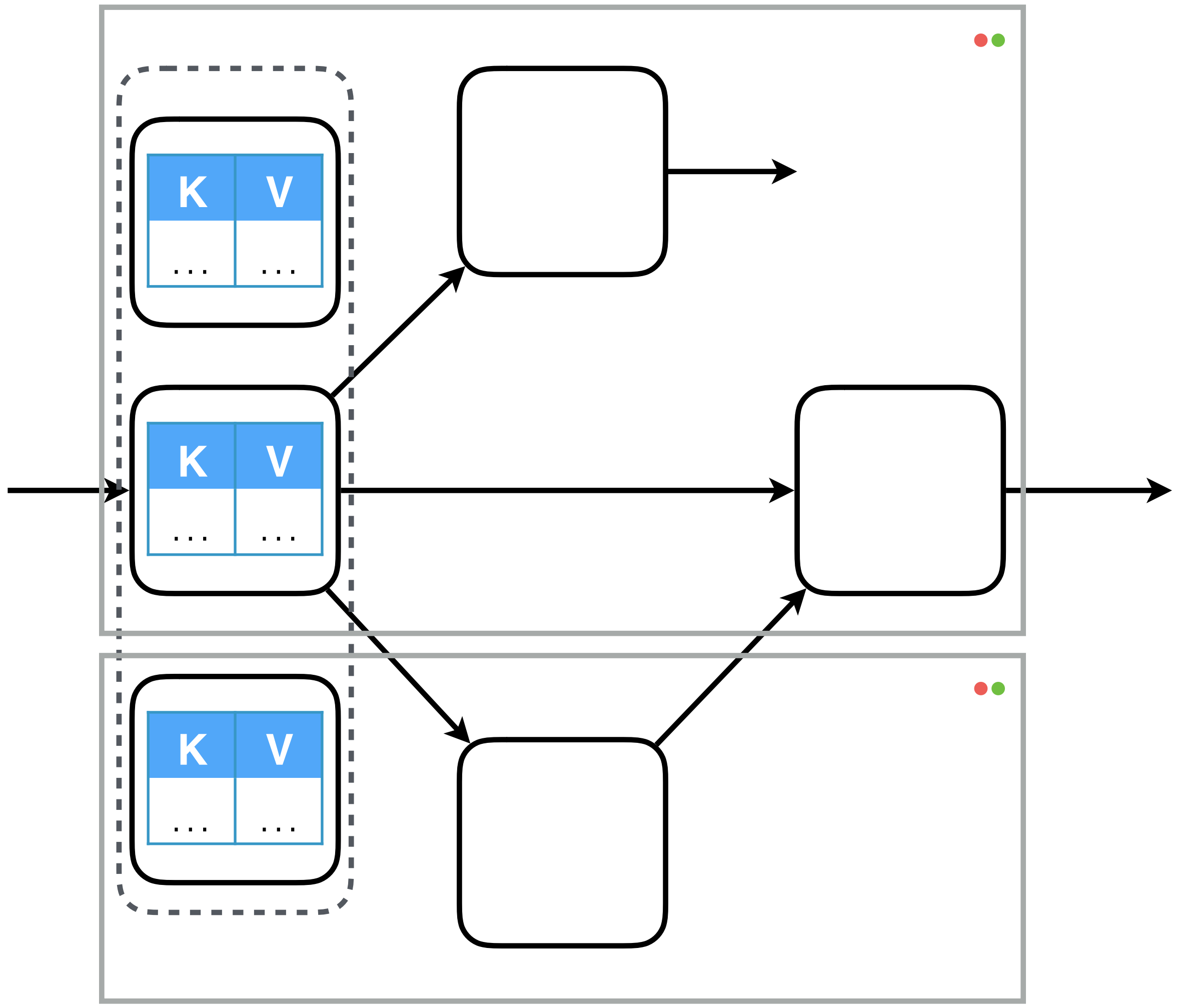
```
dataStream.keyBy("someKey") // Key by field "someKey"
dataStream.keyBy(0) // Key by the first element of a Tuple
```

Exercise

- Implement a *parallel* word counter.

Keyed State

- The stored state is often a key-value mapping in which the keys are determined by the input KeyedStream.
- Flink provides an abstraction called Keyed State that has nice APIs for accessing the state and automatic state management[1].



[1] State management will be clarified in the part dedicated to fault-tolerance.

Exercise

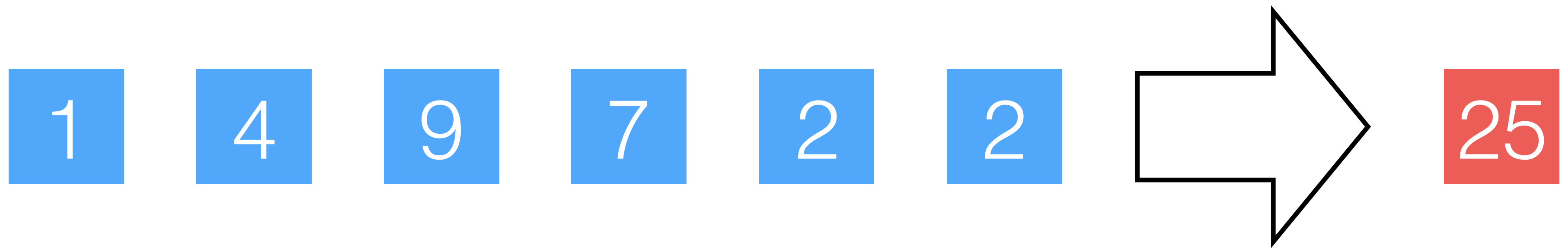
- Implement a word counter using Flink’s Keyed State.

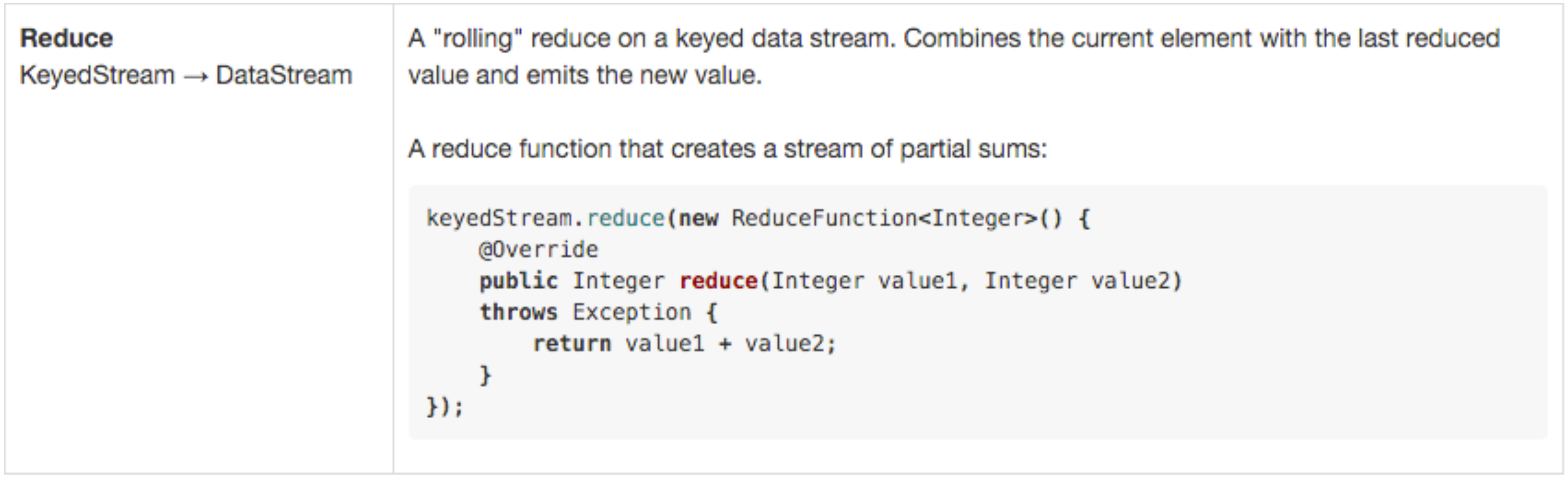
Back to Transformations

Transformations

Reduce

$(acc, i) \rightarrow acc + i$





Reduce KeyedStream → DataStream	A "rolling" reduce on a keyed data stream. Combines the current element with the last reduced value and emits the new value. A reduce function that creates a stream of partial sums:

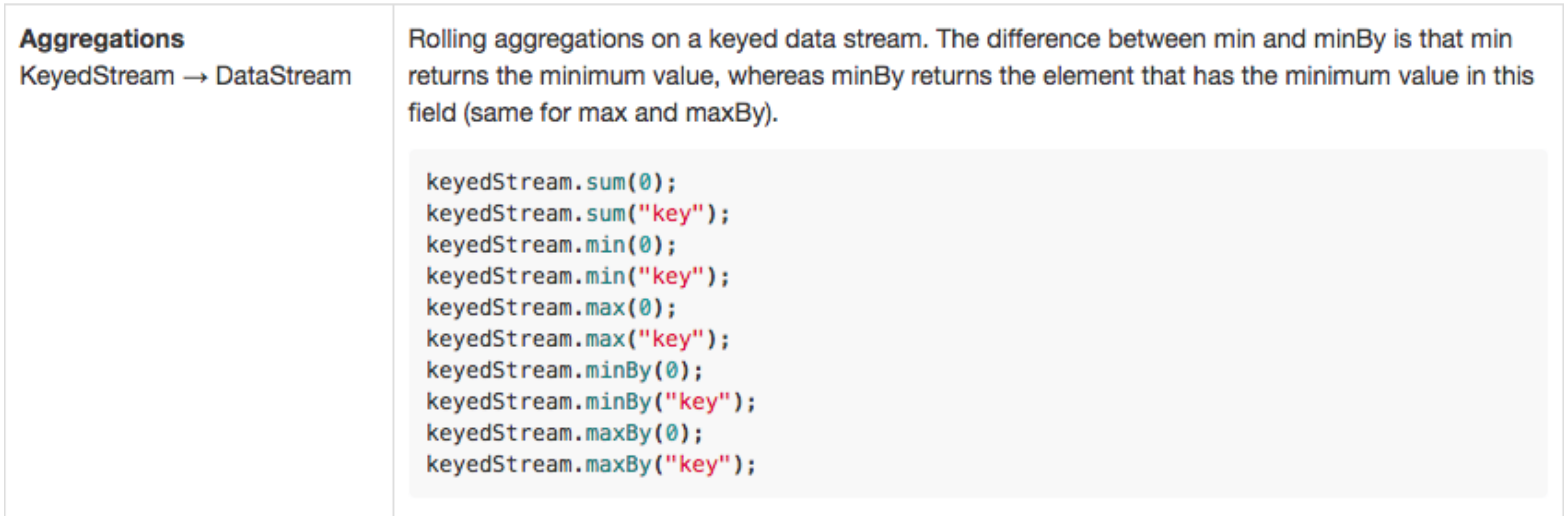
```
keyedStream.reduce(new ReduceFunction<Integer>() {
    @Override
    public Integer reduce(Integer value1, Integer value2)
    throws Exception {
        return value1 + value2;
    }
});
```

Exercise

- Take a stream of a thousand random integers;
- Generate a new stream where the *i-th* element is the square of the *i-th* element in the initial stream;
- Generate a stream where the *i-th* element is the sum of every element of the original stream from 0 to position *i*;
- Print the resulting stream.

Transformations
Aggregations

- Flink bakes some standard aggregations on KeyedStreams for you.
- For custom aggregations you should use a stateful map or flatMap[1]



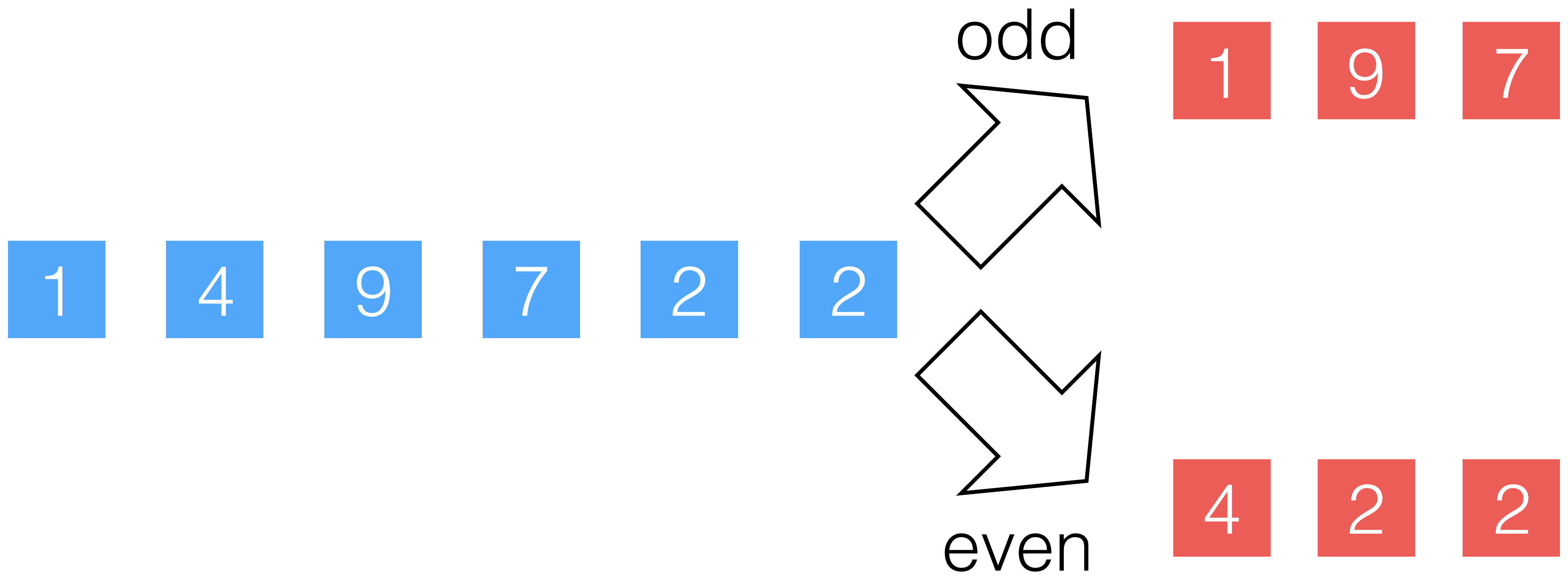
Aggregations KeyedStream → DataStream	Rolling aggregations on a keyed data stream. The difference between min and minBy is that min returns the minimum value, whereas minBy returns the element that has the minimum value in this field (same for max and maxBy).

```
keyedStream.sum(0);
keyedStream.sum("key");
keyedStream.min(0);
keyedStream.min("key");
keyedStream.max(0);
keyedStream.max("key");
keyedStream.minBy(0);
keyedStream.minBy("key");
keyedStream.maxBy(0);
keyedStream.maxBy("key");
```

[1] https://github.com/apache/flink/pull/3816

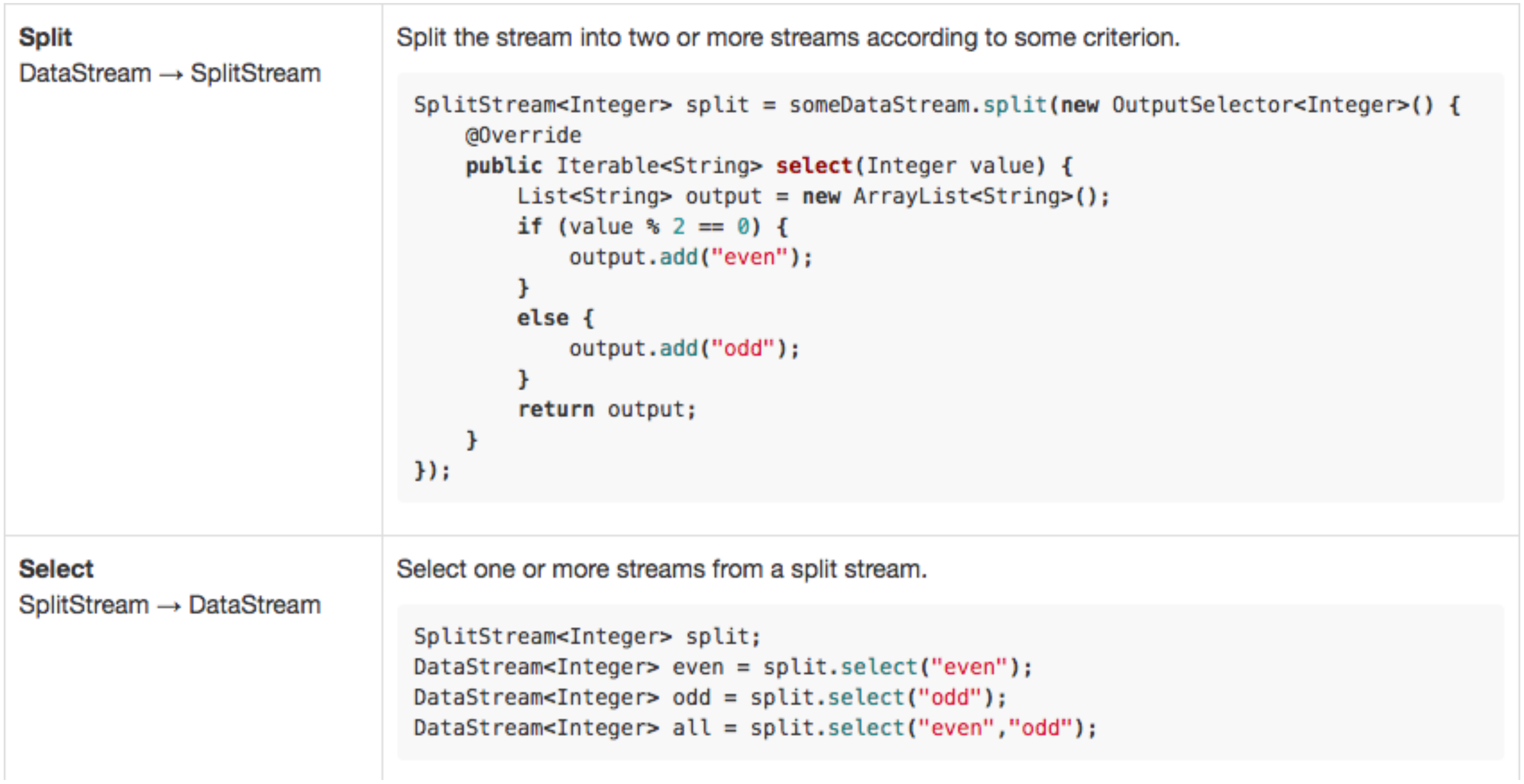
Transformations

Split & Select



Transformations

Split & Select



Split
DataStream → SplitStream

Split the stream into two or more streams according to some criterion.

```
SplitStream<Integer> split = someDataStream.split(new OutputSelector<Integer>() {
    @Override
    public Iterable<String> select(Integer value) {
        List<String> output = new ArrayList<String>();
        if (value % 2 == 0) {
            output.add("even");
        }
        else {
            output.add("odd");
        }
        return output;
    }
});
```

Select
SplitStream → DataStream

Select one or more streams from a split stream.

```
SplitStream<Integer> split;
DataStream<Integer> even = split.select("even");
DataStream<Integer> odd = split.select("odd");
DataStream<Integer> all = split.select("even","odd");
```

Exercise

- Take a stream of a thousand random integers;
- Generate a new stream *A* containing the elements divisible by 5 and the ones by 3;
- Generate a new stream *B* containing the elements divisible by 2;
- Map a square function to *A* and a cube function to *B*;
- Print the resulting streams.

* sorry, no guarantees

Windowing & Time

a short* digression

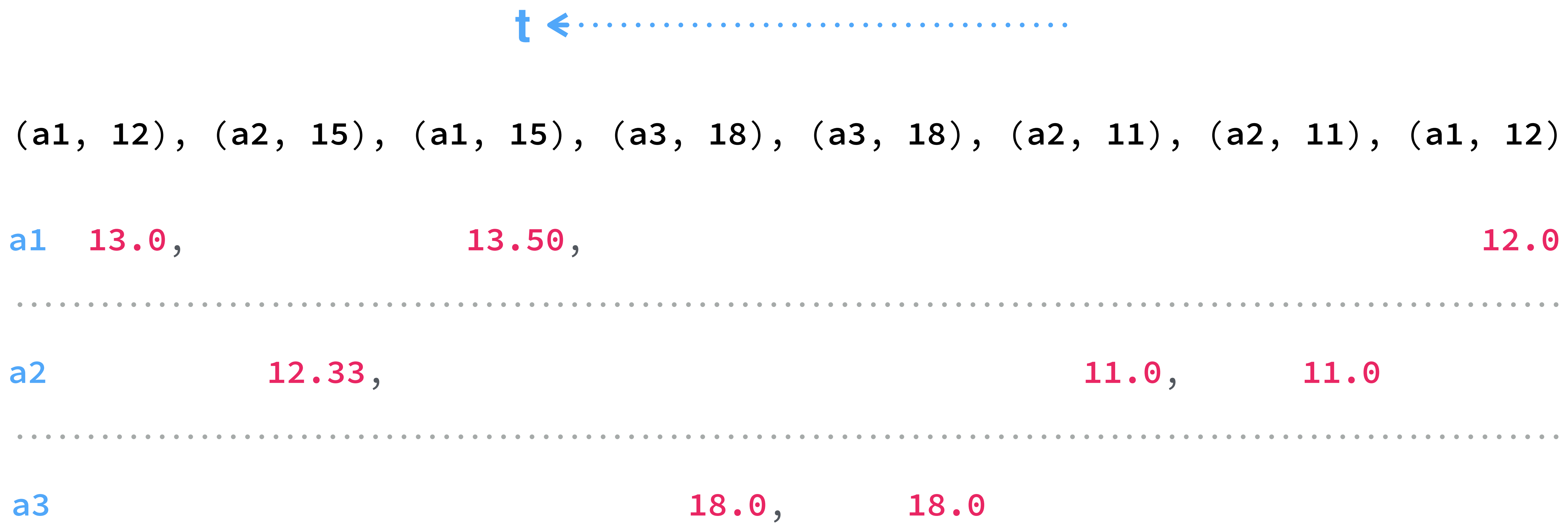
Rolling Average

A simple rolling operation outputs a new result as new records are ingested and processed…

```
(a1, 12), (a2, 15), (a1, 15), (a3, 18), (a3, 18), (a2, 11), (a2, 11), (a1, 12)

    14.0,     14.28,     14.16,      14.0,      13.0,     11.33,      11.5,      12.0
```

Keyed Average

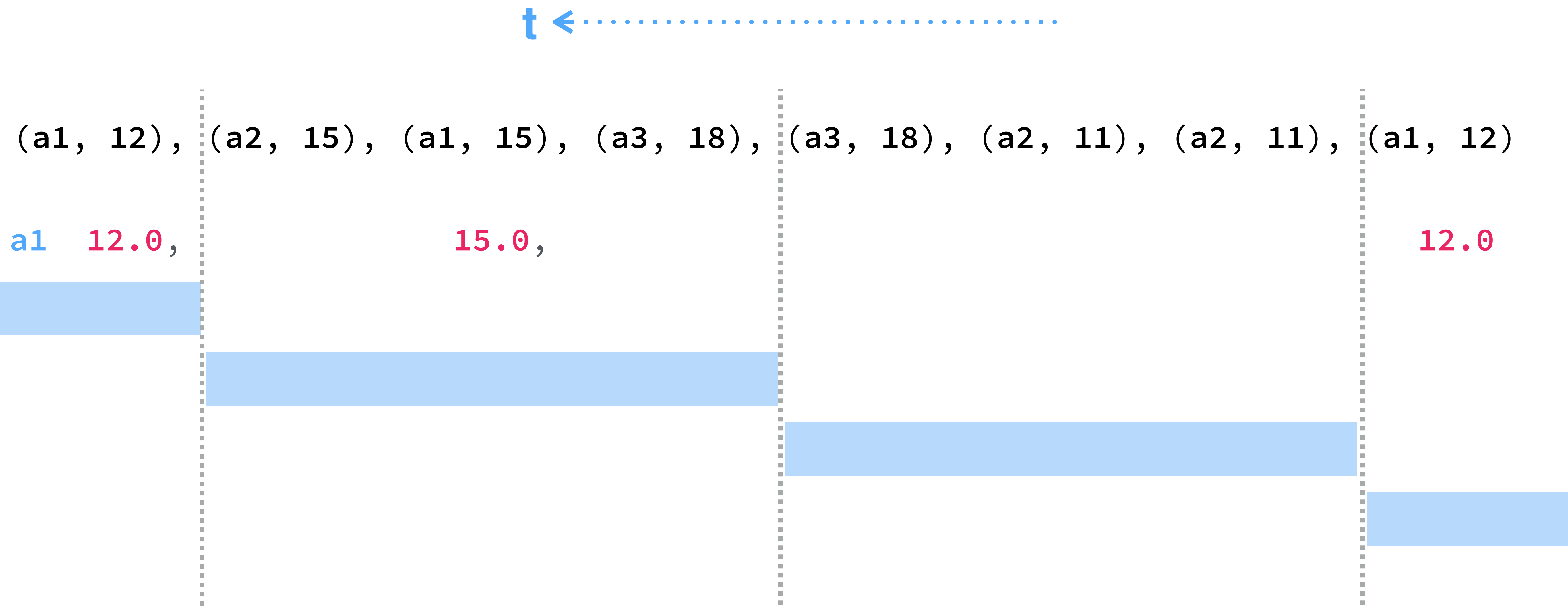
A keyBy operation partitions the stream and let's you compute a partitioned-by-key average.



```
(a1, 12), (a2, 15), (a1, 15), (a3, 18), (a3, 18), (a2, 11), (a2, 11), (a1, 12)

a1  13.0,            13.50,                                            12.0

a2        12.33,                                11.0,      11.0

a3                              18.0,      18.0
```

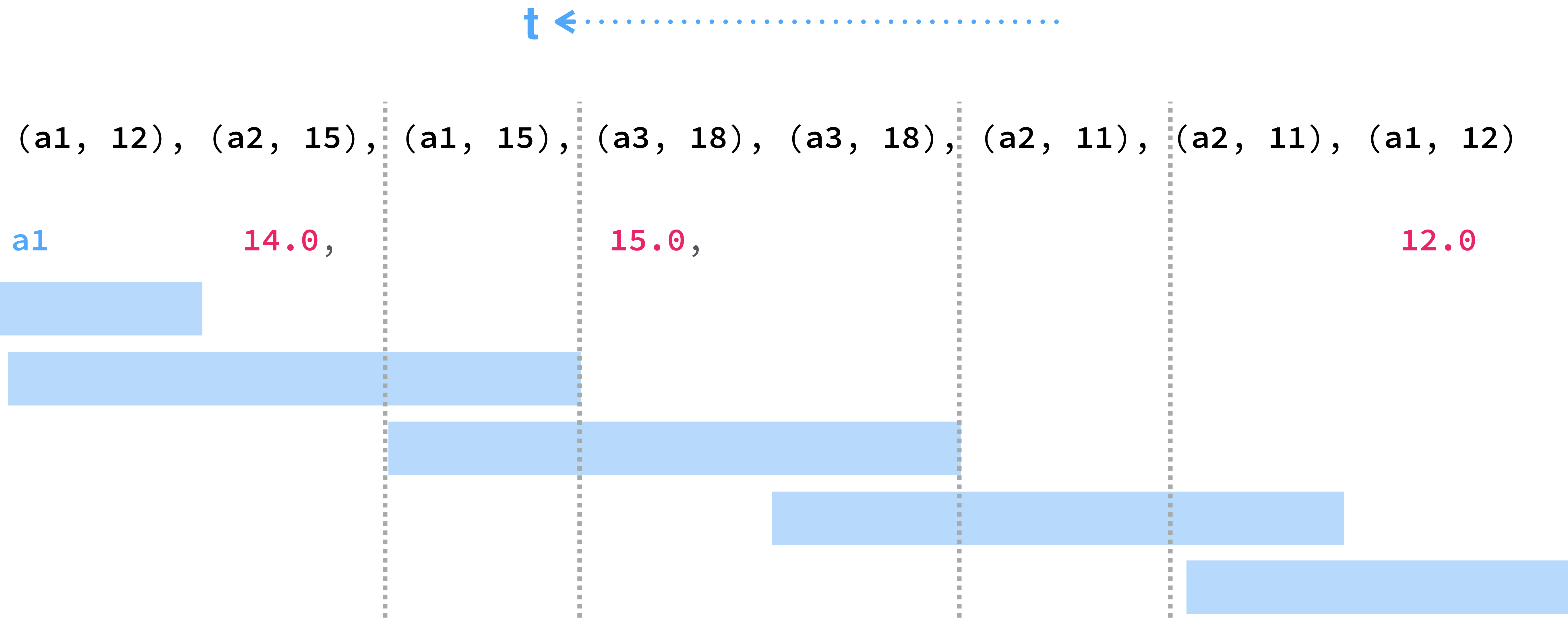
Tumbling Window

Take a single partition as example, you can slice time in fixed and non-overlapping slices


t
(a1, 12), (a2, 15), (a1, 15), (a3, 18), (a3, 18), (a2, 11), (a2, 11), (a1, 12)
a1 12.0, 15.0, 12.0

Sliding Window

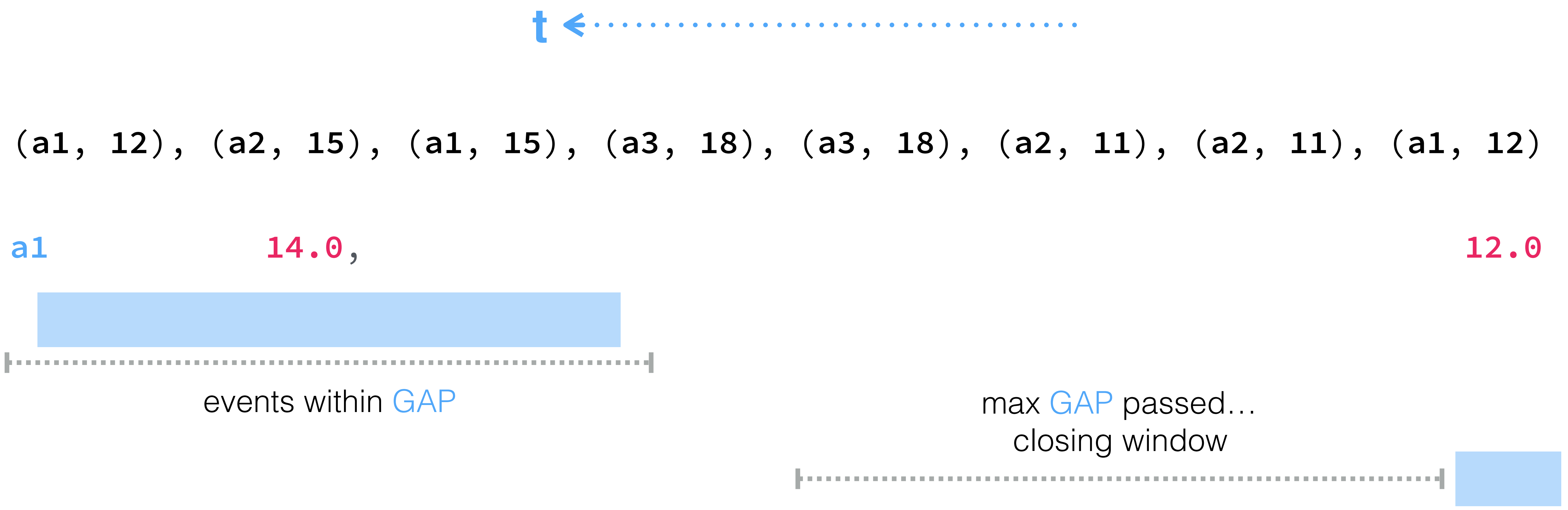
Take a single partition as example, you can slice time in fixed and ~~non-~~overlapping slices



t
(a1, 12), (a2, 15), (a1, 15), (a3, 18), (a3, 18), (a2, 11), (a2, 11), (a1, 12)
a1
14.0,
15.0,
12.0

Session Window

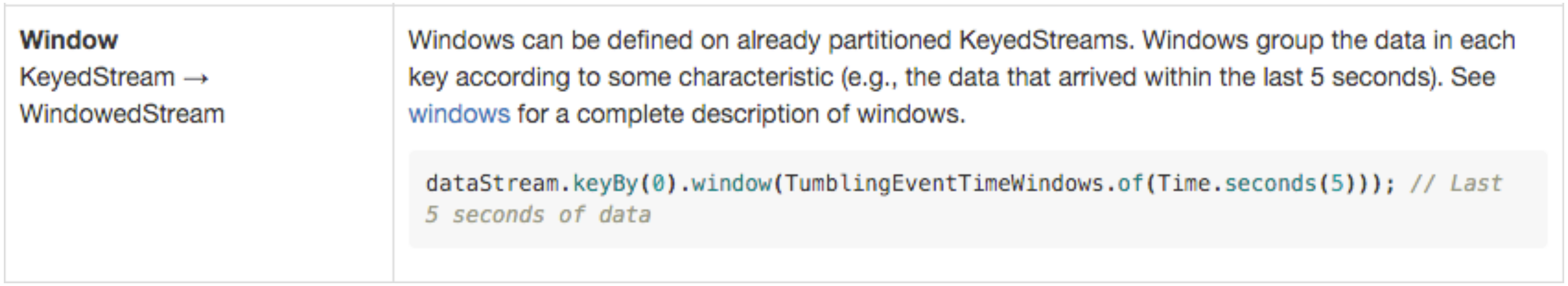
You can also slice time dynamically, by closing windows if no event happens within a certain GAP.



```
(a1, 12), (a2, 15), (a1, 15), (a3, 18), (a3, 18), (a2, 11), (a2, 11), (a1, 12)
```

```
a1              14.0,                                                        12.0
```

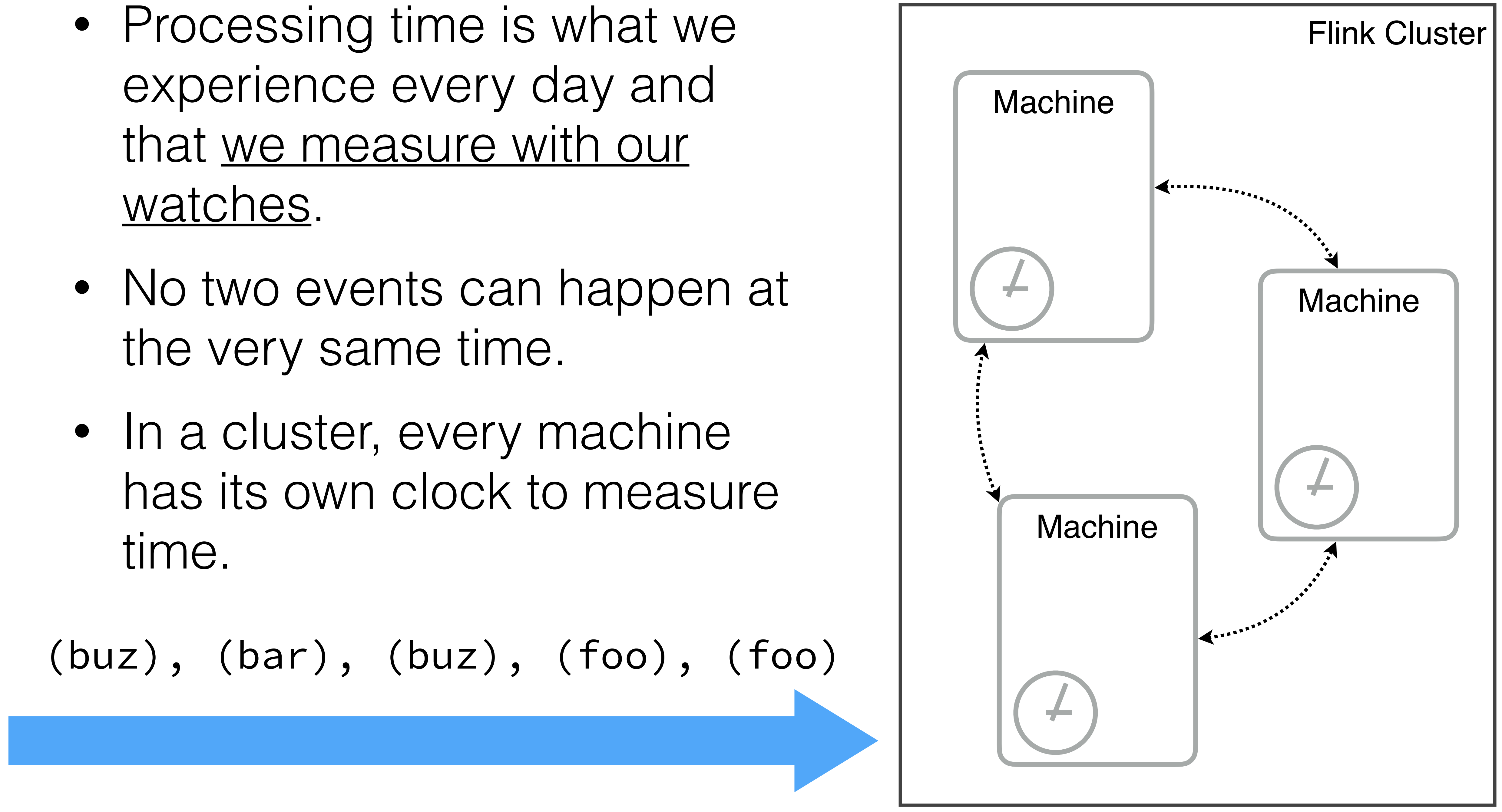
Window



Window KeyedStream → WindowedStream	Windows can be defined on already partitioned KeyedStreams. Windows group the data in each key according to some characteristic (e.g., the data that arrived within the last 5 seconds). See windows for a complete description of windows. `dataStream.keyBy(0).window(TumblingEventTimeWindows.of(Time.seconds(5))); // Last 5 seconds of data`

Two Times
Processing Time

- Processing time is what we experience every day and that we measure with our watches.
- No two events can happen at the very same time.
- In a cluster, every machine has its own clock to measure time.

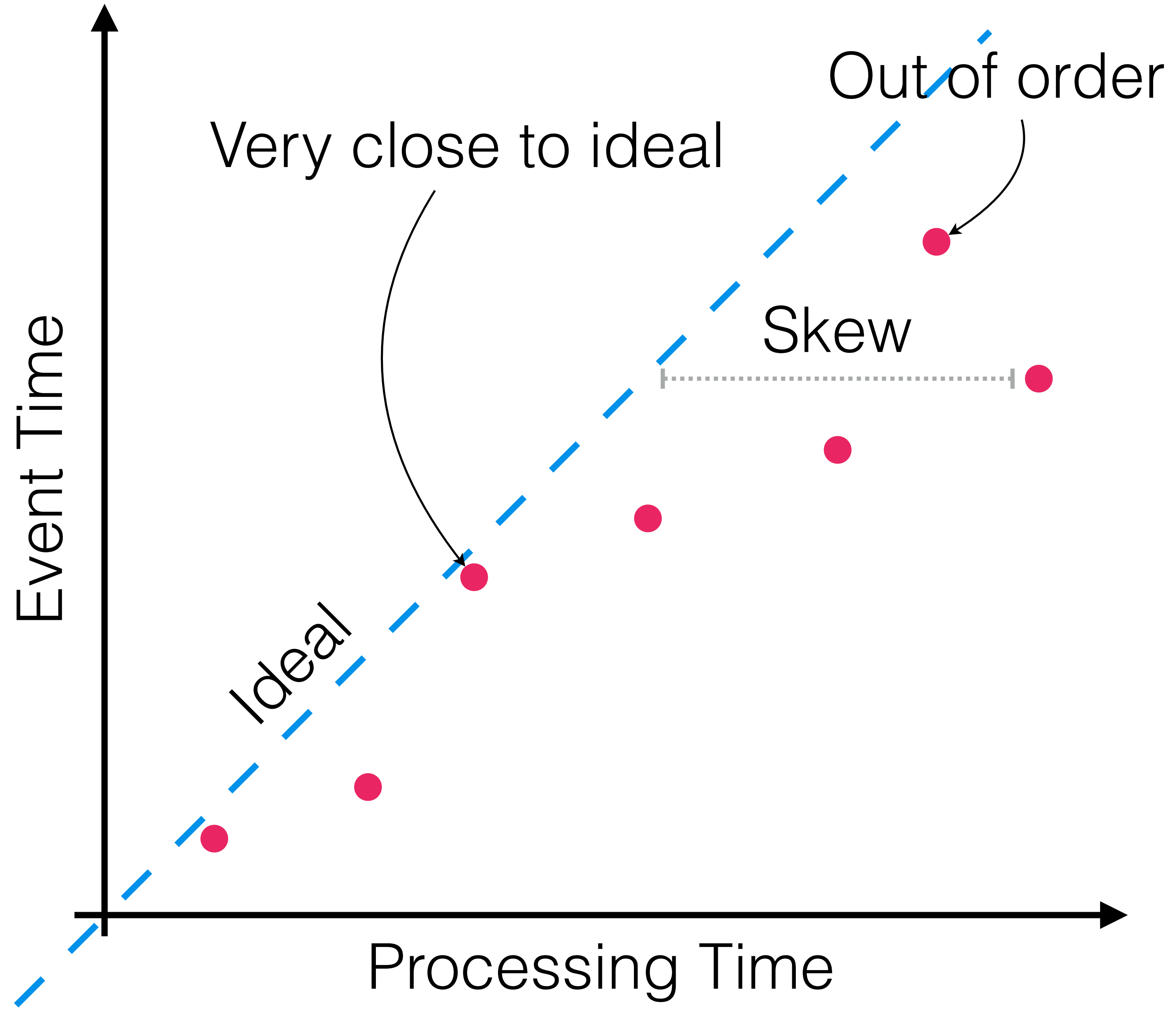


The Problem with Processing Time

Processing is affected by:

- Network congestion
- CPU contention & thread scheduling

That's why the **results (that involve time) obtained with processing time are not deterministic and not reproducible**.



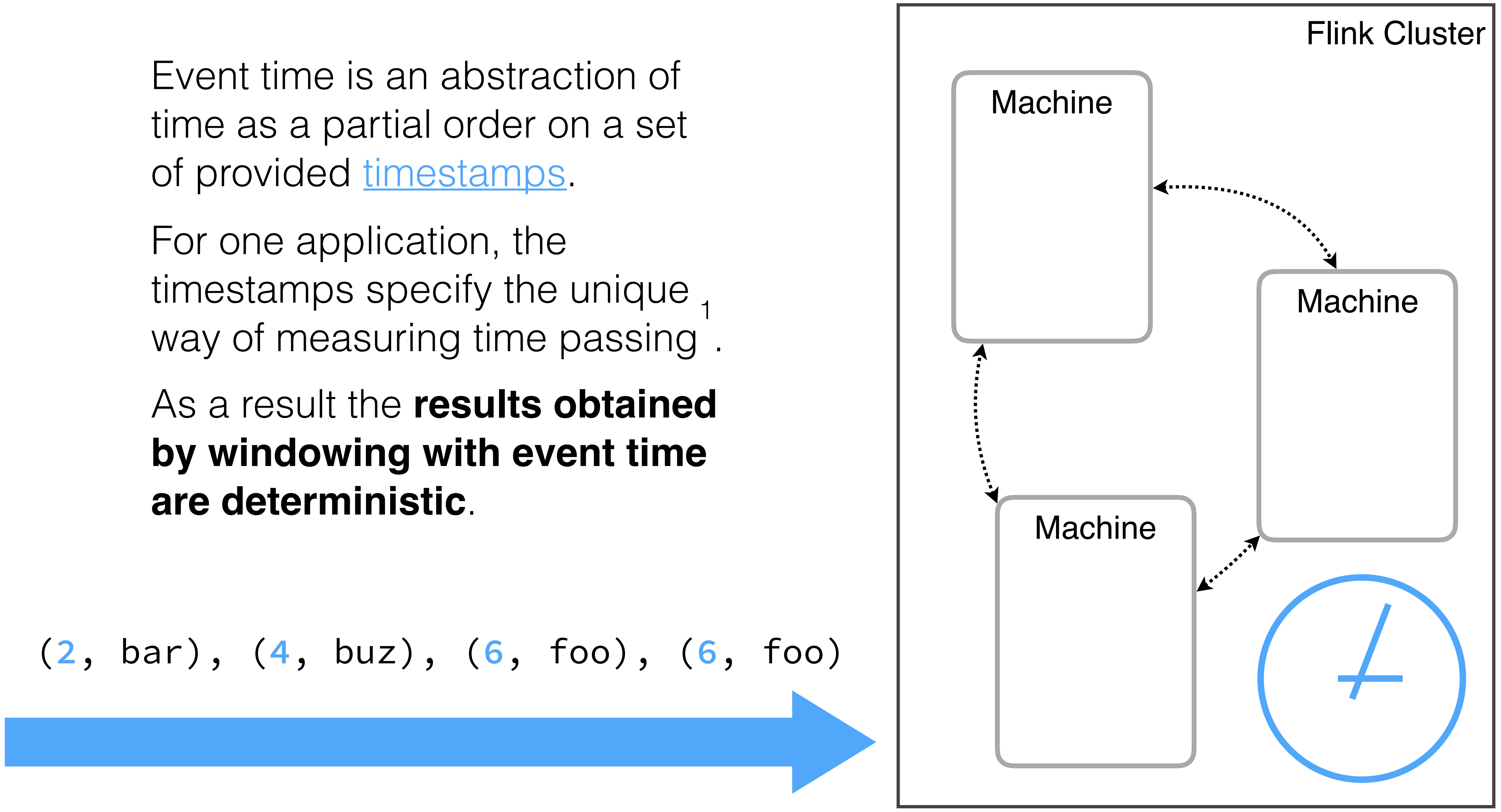
Two Times
Event Time

Event time is an abstraction of time as a partial order on a set of provided timestamps.

For one application, the timestamps specify the unique way of measuring time passing[1].

As a result the **results obtained by windowing with event time are deterministic**.

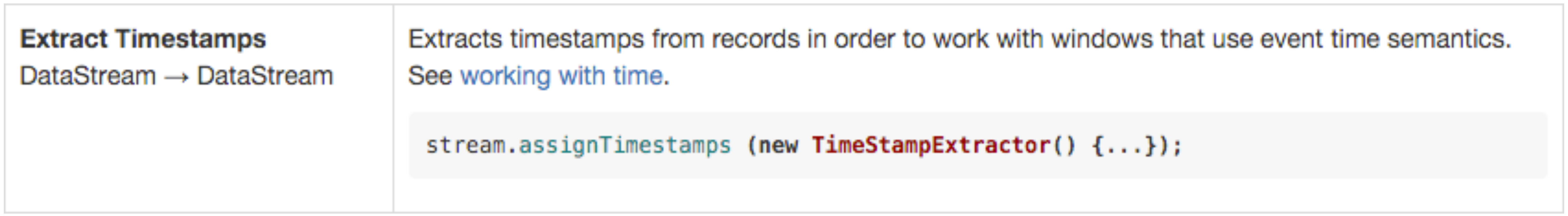
```
(2, bar), (4, buz), (6, foo), (6, foo)
```



[1] How to make time slide forward in a cluster is treated in appendix.

Extract Timestamp

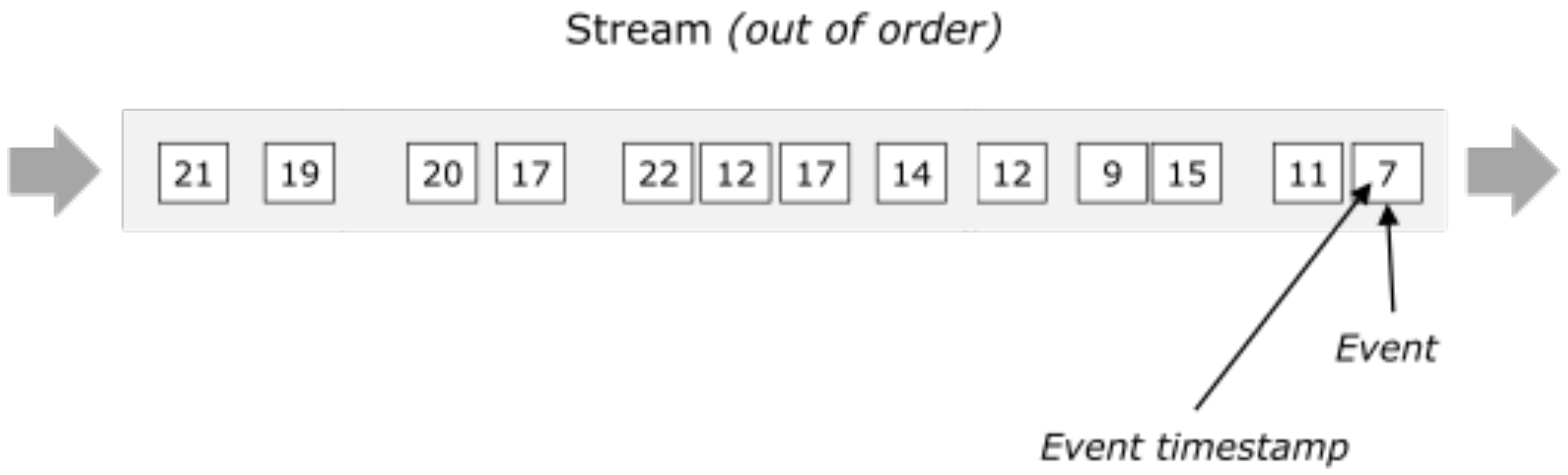
The user has to specify how the timestamps can be extracted from the ingested data.



Extract Timestamps DataStream → DataStream	Extracts timestamps from records in order to work with windows that use event time semantics. See working with time. `stream.assignTimestamps (new TimeStampExtractor() {...});`

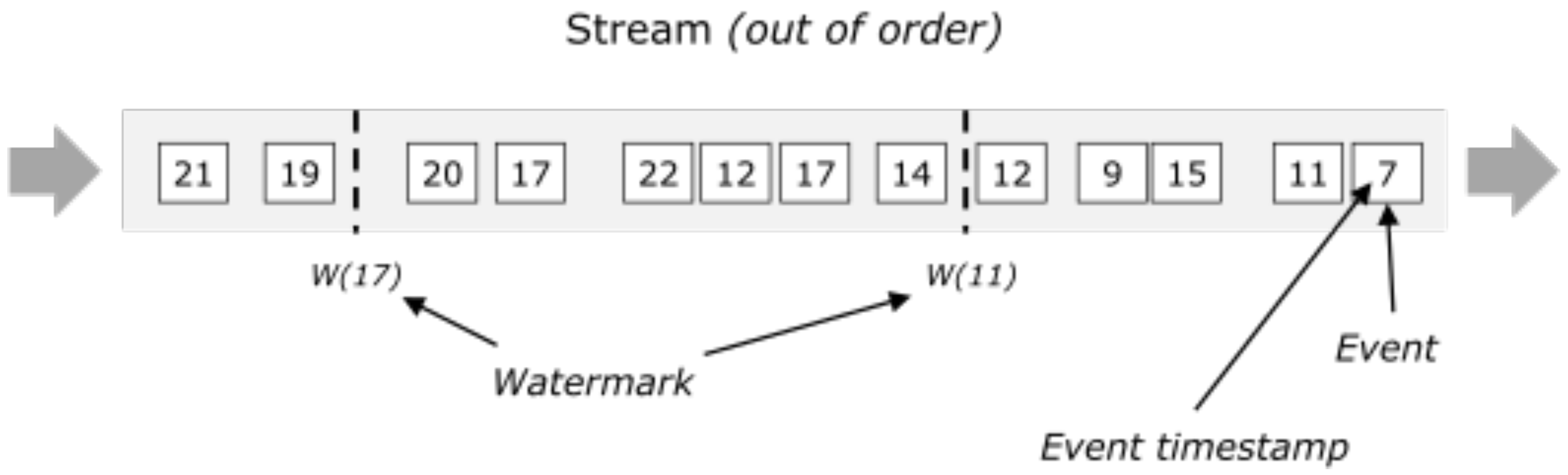
Time Progress

How to keep the progress of event time?



Out-of-order Records and Watermarks

Watermarks



Watermarks

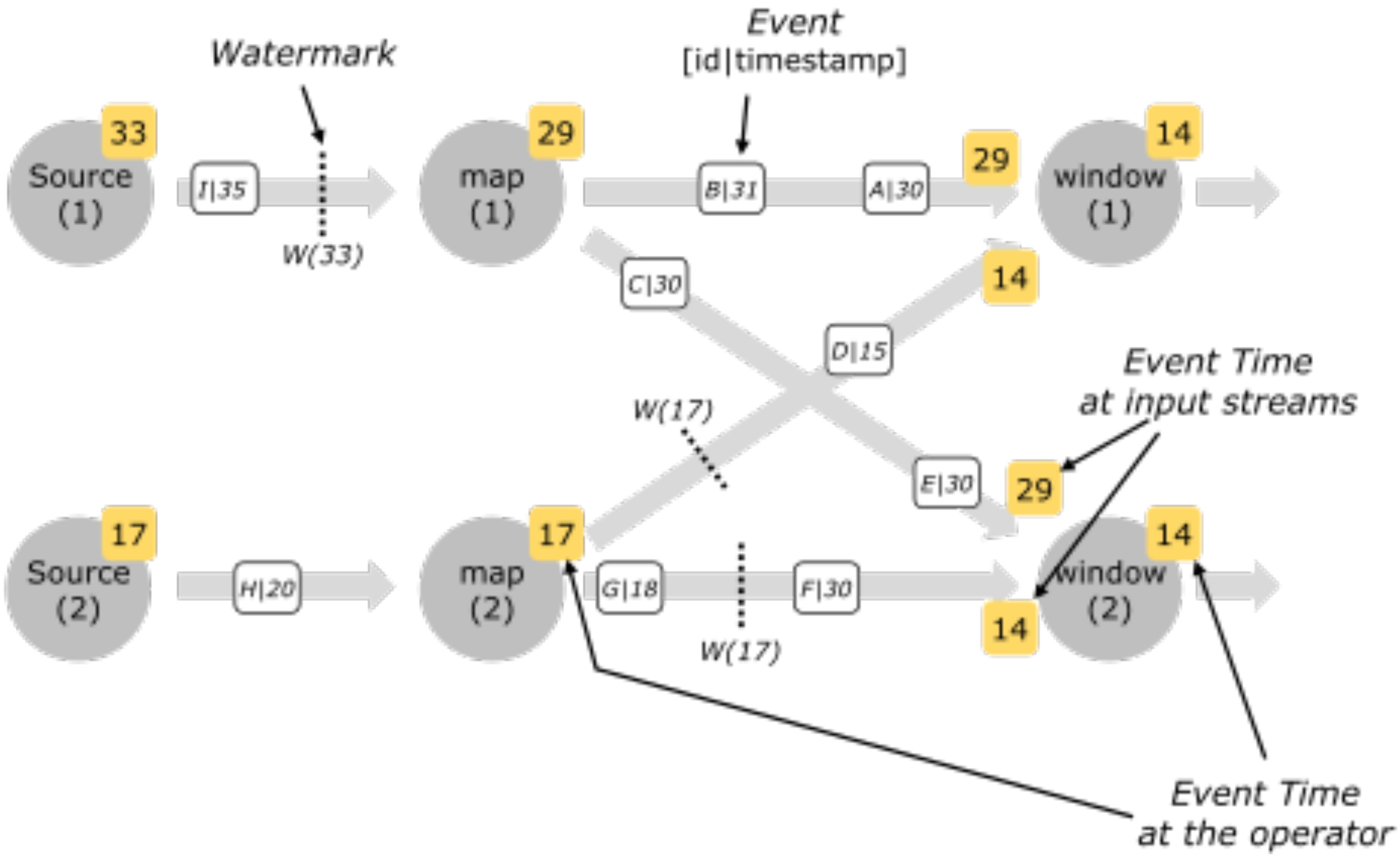
- Watermarks can be:
  - directly **injected** in the streams by the sources;
  - **extracted** from a timestamp field with various techniques[1];
- WM = *t* means that event time has reached *t*, thus that no record with a timestamp lower or equal to *t* will ever come;
- They become the clock of the system.

[1] more info in Flink's documentation

Watermarks in Parallel

- Watermarks are generated at, or directly after, source functions. Each parallel subtask of a source function usually generates its watermarks *independently*. These watermarks define the event time at that particular parallel source.
- 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.
- **Some operators consume multiple input streams**; a union, for example, or operators following a keyBy(…) or partition(…) function. **Such an operator's current event time is the minimum of its input streams' event times**. As its input streams update their event times, so does the operator.

Watermarks in Parallel



Aggregating Windows

Function	ACC	IN	OUT	MERGE?*
reduce	T	T	T	Y
aggregate	A	I	O	Y
fold	A	I	A	N

* works with mergeable windows (e.g. session windows)?

Demo

- In a building, sensors provides continuous measurements of the temperature in some rooms.
- The demo computes the average temperature by room every 5 milliseconds.
- Record generation is controlled by an external socket in order to show how the system behaves every time a new element is generated.
- The demo shows how the system handles out-of-order records generated by a parallel source (parallelism = 2).

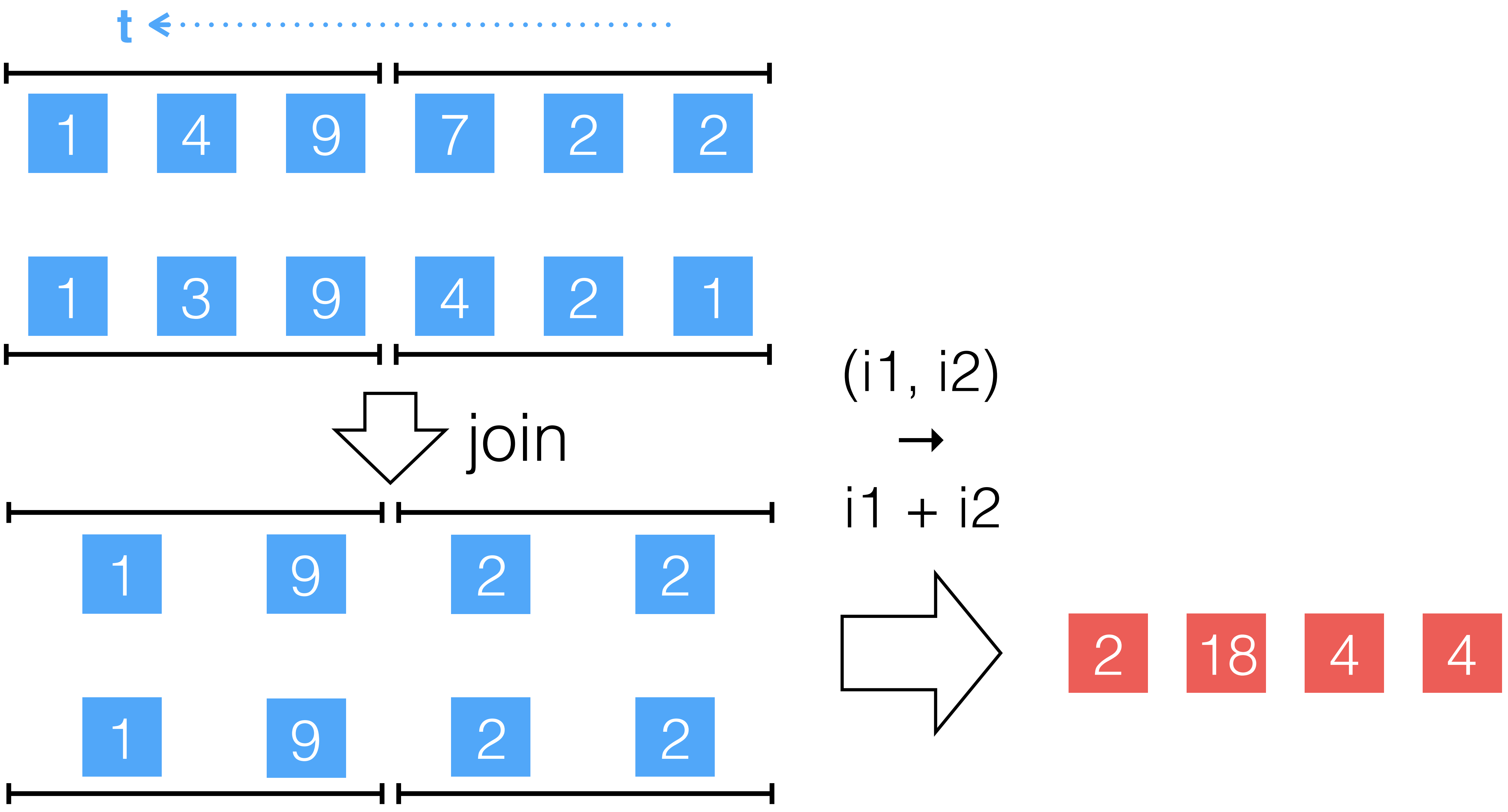
Exercise

- In a building, sensors provide continuous measurements of the temperature in some rooms.
- Calculate the average temperature in every room for the last (event-time) 15 millisecond every (event-time) 5 milliseconds

Back to Transformations

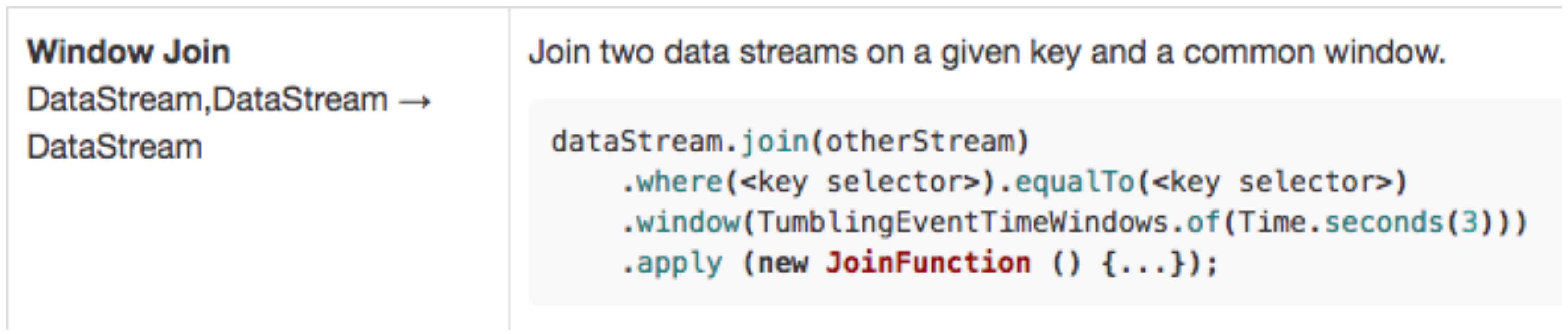
Transformations

(Window) Join



Transformations

(Window) Join



Window Join DataStream,DataStream → DataStream	Join two data streams on a given key and a common window.

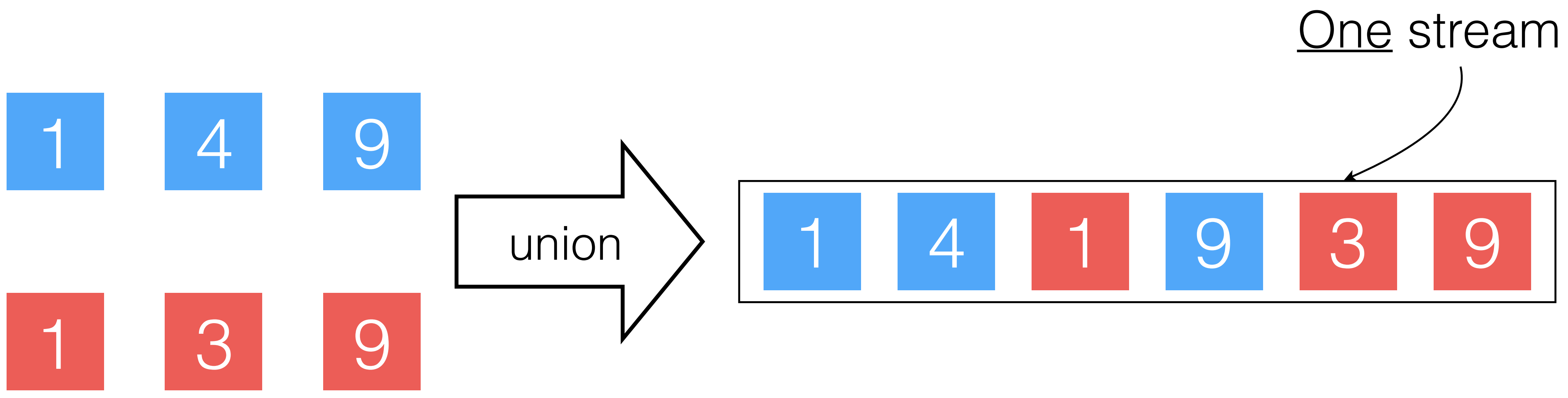
```
dataStream.join(otherStream)
    .where(<key selector>).equalTo(<key selector>)
    .window(TumblingEventTimeWindows.of(Time.seconds(3)))
    .apply (new JoinFunction () {...});
```

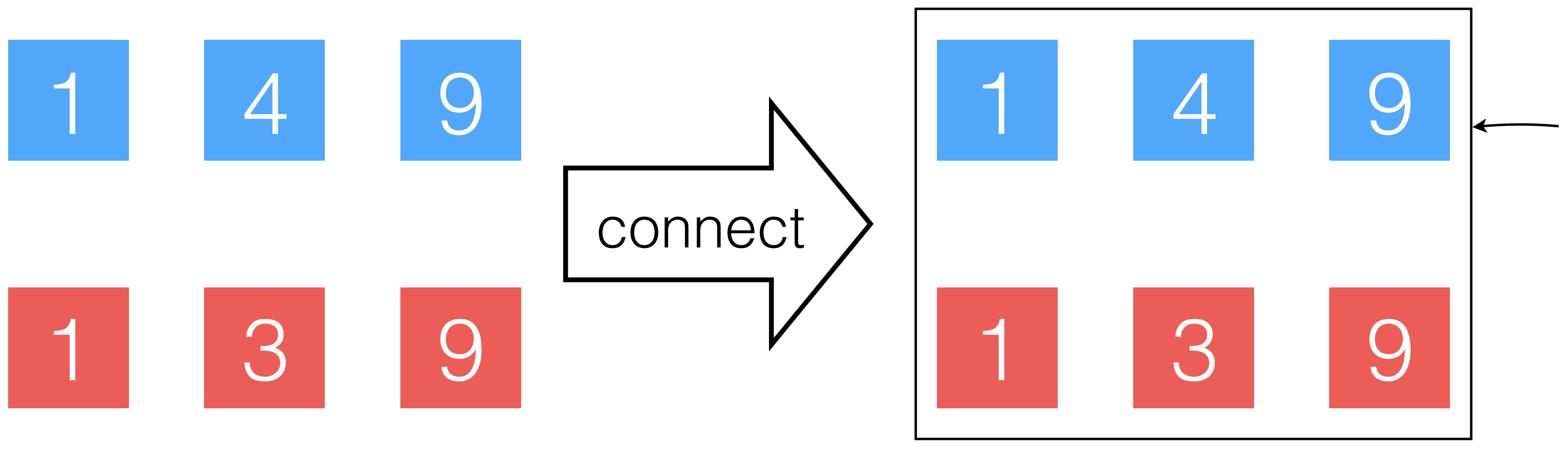
Exercise

- Generate two streams as in slide 30 (use env.fromElements(T…) and the NumberGenerator provided);
- Reproduce the example in slide 30 (you can use Tuples to simplify equality constraints).

Transformations

Union VS Connect



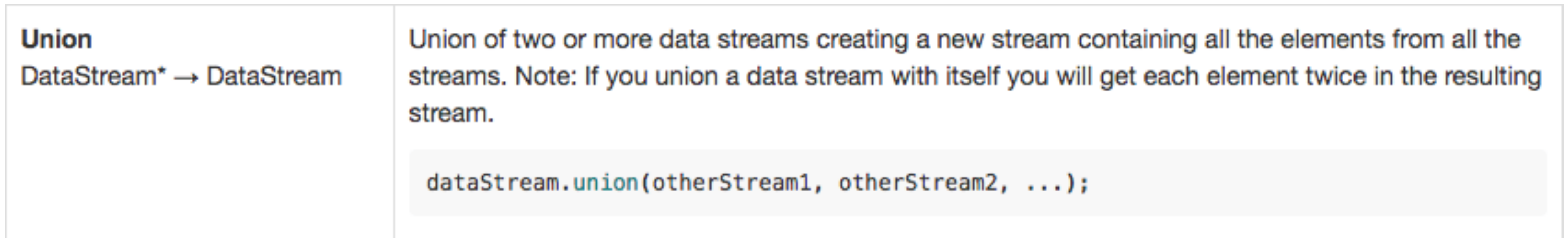


One connected stream.

There is no defined order between the elements.

You can share state upon processing (coMap, coFlatMap)

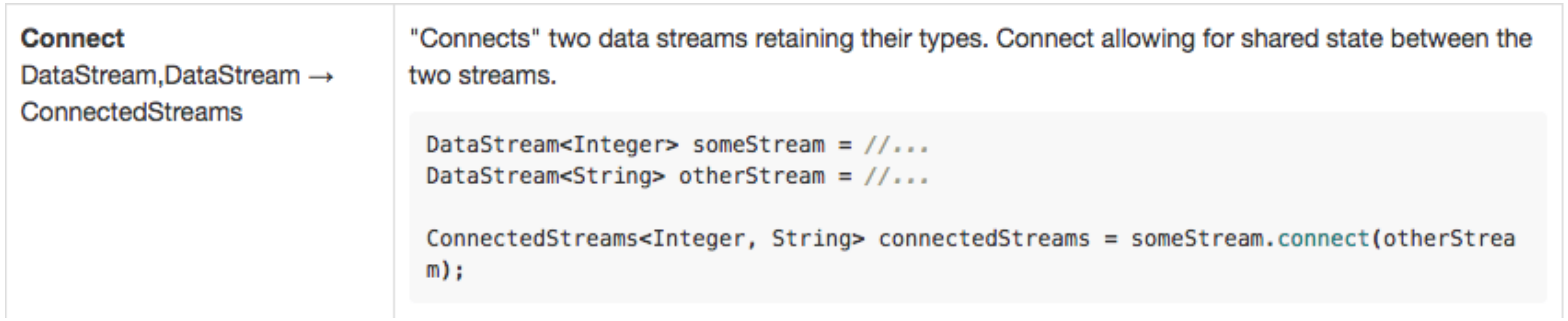
Transformations
Union VS Connect



Union
DataStream* → DataStream

Union of two or more data streams creating a new stream containing all the elements from all the streams. Note: If you union a data stream with itself you will get each element twice in the resulting stream.

```
dataStream.union(otherStream1, otherStream2, ...);
```



Connect
DataStream,DataStream → ConnectedStreams

"Connects" two data streams retaining their types. Connect allowing for shared state between the two streams.

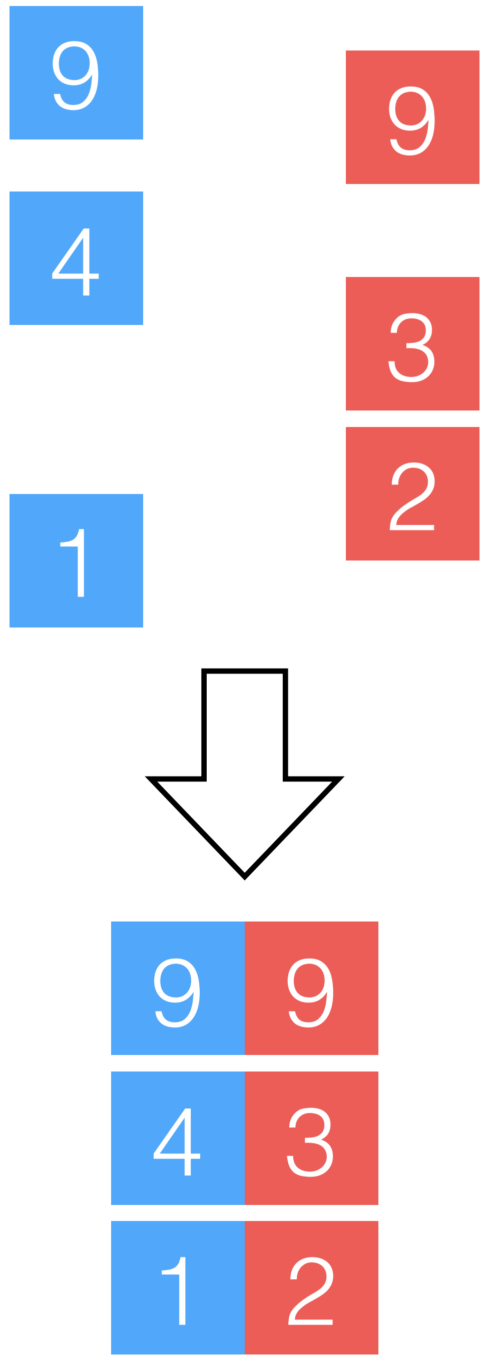
```
DataStream<Integer> someStream = //...
DataStream<String> otherStream = //...

ConnectedStreams<Integer, String> connectedStreams = someStream.connect(otherStrea
m);
```

Exercise

- Take two stream of integers;
- Implement a rolling count both with union and connect.

Exercise

- Take two stream of integers
- *Align* them.



Transformations

Process Function

- They are like low-level FlatMap transformations with access to keyed state and event (or processing) time *timers*.
- The *TimerService* can be used to register callbacks for future time instants.
- When a timer's particular time is reached, the *onTimer(...)* method is called;
- During that call, all states are again scoped to the key with which the timer was created, allowing timers to manipulate keyed state.

Exercise

- Retrieve the temperature data from sensors;
- Output the rooms that did not receive any update for the temperature in the last 5 milliseconds.