POLITECNICO
MILANO 1863

Apache Flink

Alessandro Margara

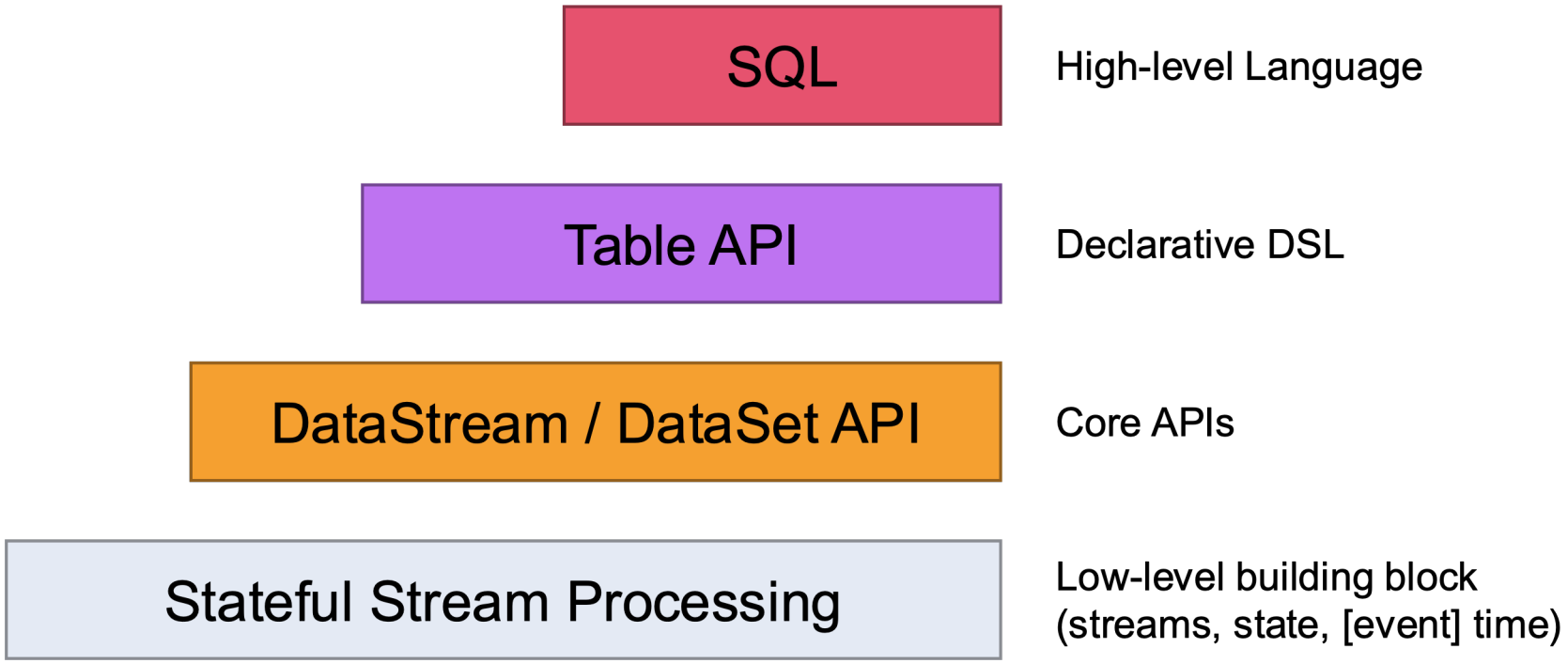
alessandro.margara@polimi.it

License

Big Data frameworks

- Big Data frameworks such as Apache Spark define a graph of computations
- Data “streams” from operator to operator
  - Duality between data streams and datasets
- A single engine can do both datasets (batch) and data stream (continuous) processing
- Different approaches
  - Apache Spark Streaming: building stream processing on top of (micro) batches
  - Apache Flink: building batch processing on top of a stream processing engine

Apache Flink



Apache Flink

- The basic building blocks of Flink are
  - Streams
  - Transformations (operators)

- When executed, Flink programs are transformed into streaming dataflows
  - Each dataflow is a directed acyclic graph (DAG)
  - Some form of cycles (iterations) are supported

Streaming Dataflow

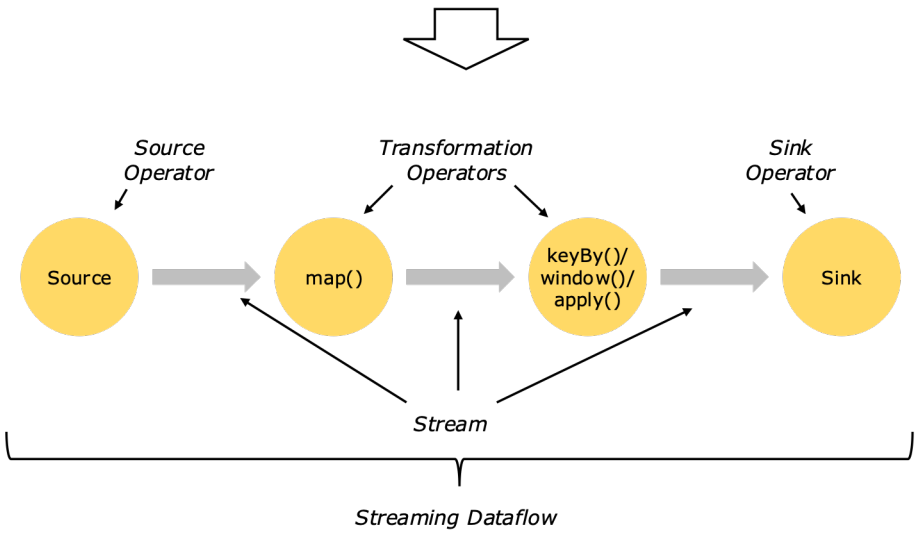
```
DataStream<String> lines = env.addSource(
                                new FlinkKafkaConsumer<>(…));

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

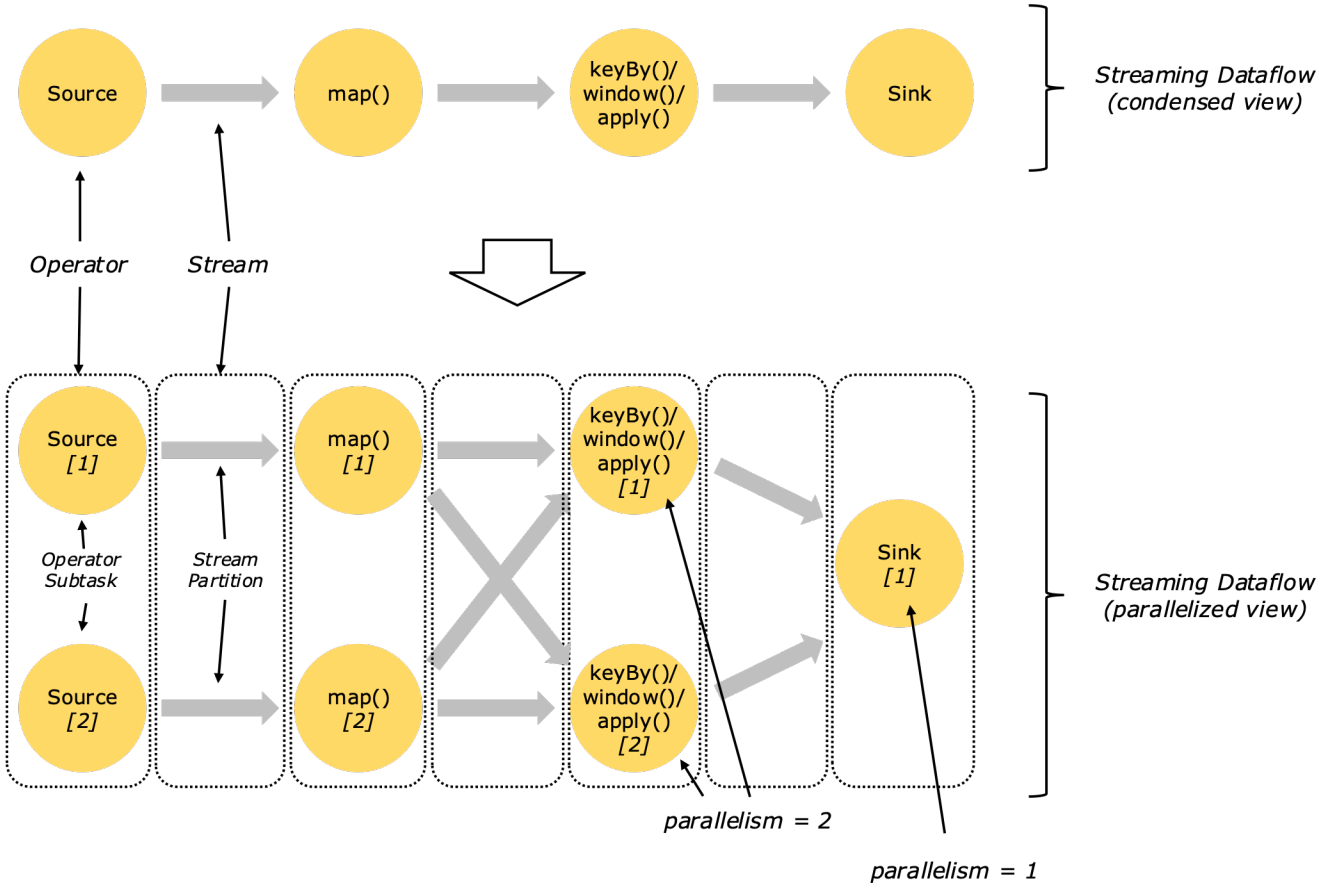
Sink



Parallel/Distributed Dataflow

- Programs in Flink are inherently parallel and distributed
  - A stream has one or more stream partitions
  - An operator has one or more operator subtasks
    - Operator subtasks are independent of one another …
    - … and execute in different threads / machines

Parallel/Distributed Dataflow



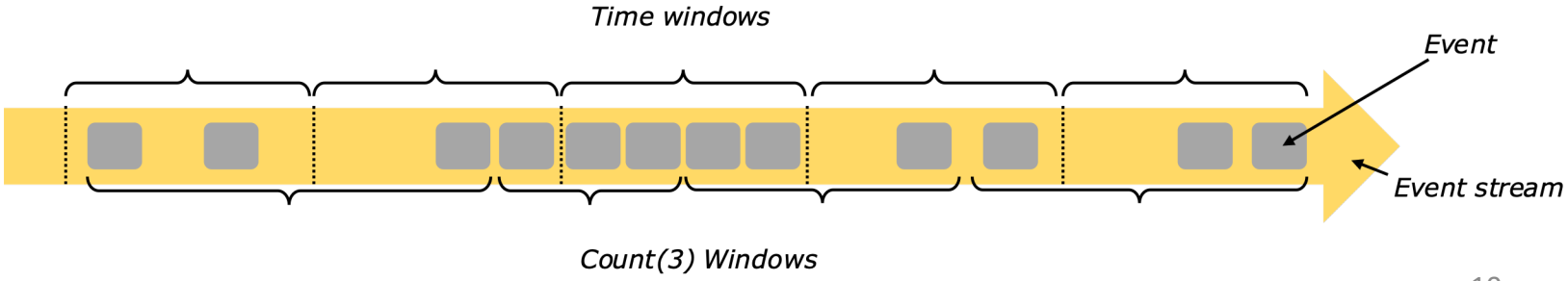
Parallel/Distributed Dataflow

- Streams can be classified according to the
  - One-to-one streams preserve the partitioning and ordering of the elements
    - Connect subtask[i] of one operator with subtask[i] of the downstream operator
  - Redistributing streams change the partitioning of the streams
    - Subtask[i] can produce results for any subtask of the downstream operator

Windows

- Blocking operators return an answer after consuming the entire input
  - E.g., count, avg

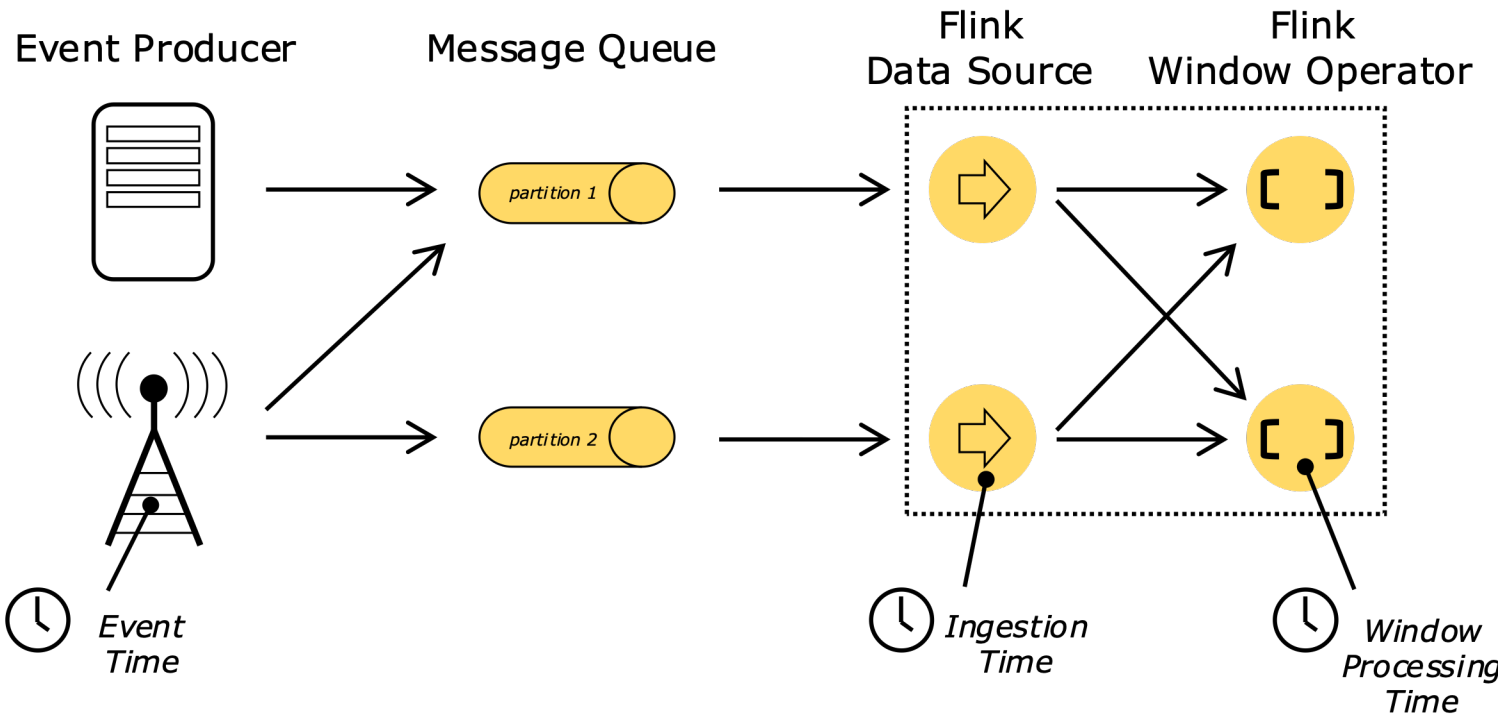
- This is clearly not possible in a streaming scenario
  - We need to select limited portion of the input stream to execute the operator on
  - These portions are defined using windows
    - Count-based (e.g., last 10 elements)
    - Time-based (e.g., last 5 minutes)
    - Session (based on elements)



Time

- In stream processing, there are three typical definitions of time
  - Event time: when an element was produced in a source
  - Ingestion time: when an element is received in the processing system
  - Processing time: the wall clock time in the operator that processes the element
- Flink considers by default event time, annotated by the sources

Time

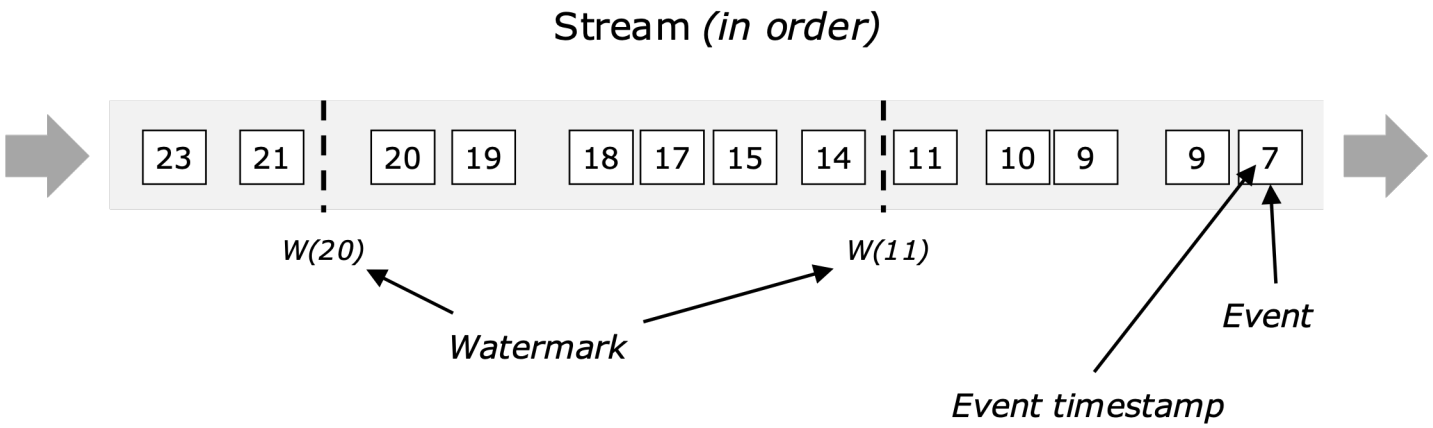

Event Producer
Message Queue
Flink
Data Source
Flink
Window Operator
partition 1
partition 2
Event
Time
Ingestion
Time
Window
Processing
Time

Event time and watermarks

- Event time can progress independently from processing time
  - For example, constant delay due to data transmission
  - In the worst case, data can even arrive out-of-order with respect to event time
- To ensure correctness (and repeatability) of results, Flink needs a way to measure the progress of event time
  - For example, consider a time-based window
- Flink measures progress in event time using watermarks
  - Watermarks flow as part of the data stream and carry a timestamp

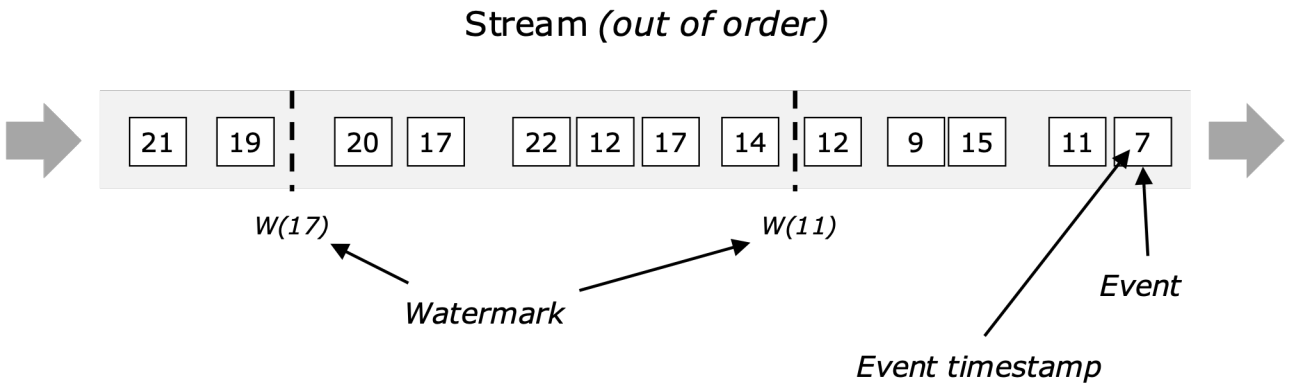
Event time and watermarks

- A watermark with timestamp t declares that event time has reached time t in that stream
  - It means that there will be no other elements in the stream with timestamp t’ <= t



Event time and watermarks

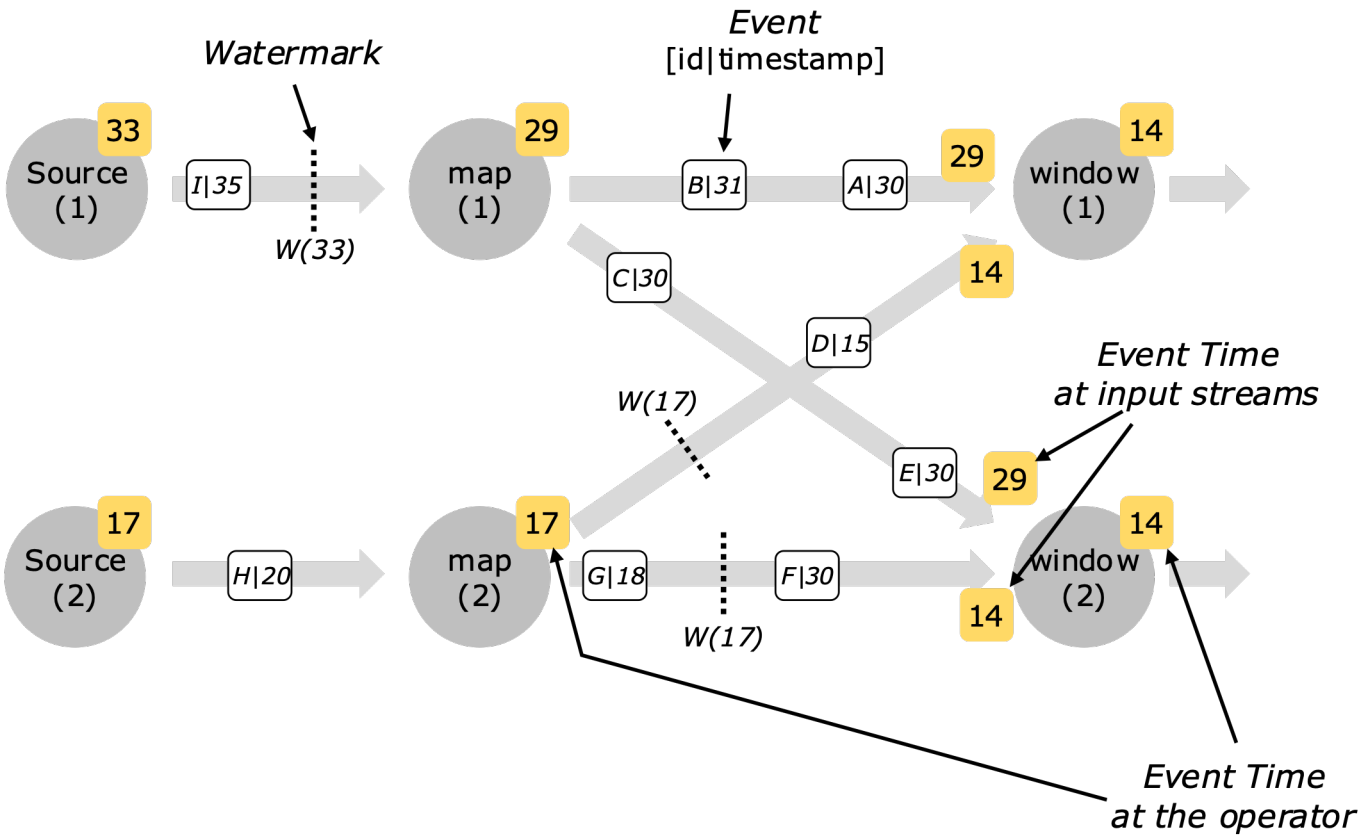
- Watermarks are crucial for out-of-order streams, where elements are not ordered by their event time
  - Once a watermark reaches an operator, the operator can advance its internal event time clock to the value of the watermark



Event time and watermarks

- Watermarks are generated at 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 (e.g., join) consume multiple input streams
  - Such an operator's current event time is the minimum of its input streams' event times

Event time and watermarks



Event time and watermarks

- What happens if some elements violate the watermark constraint?

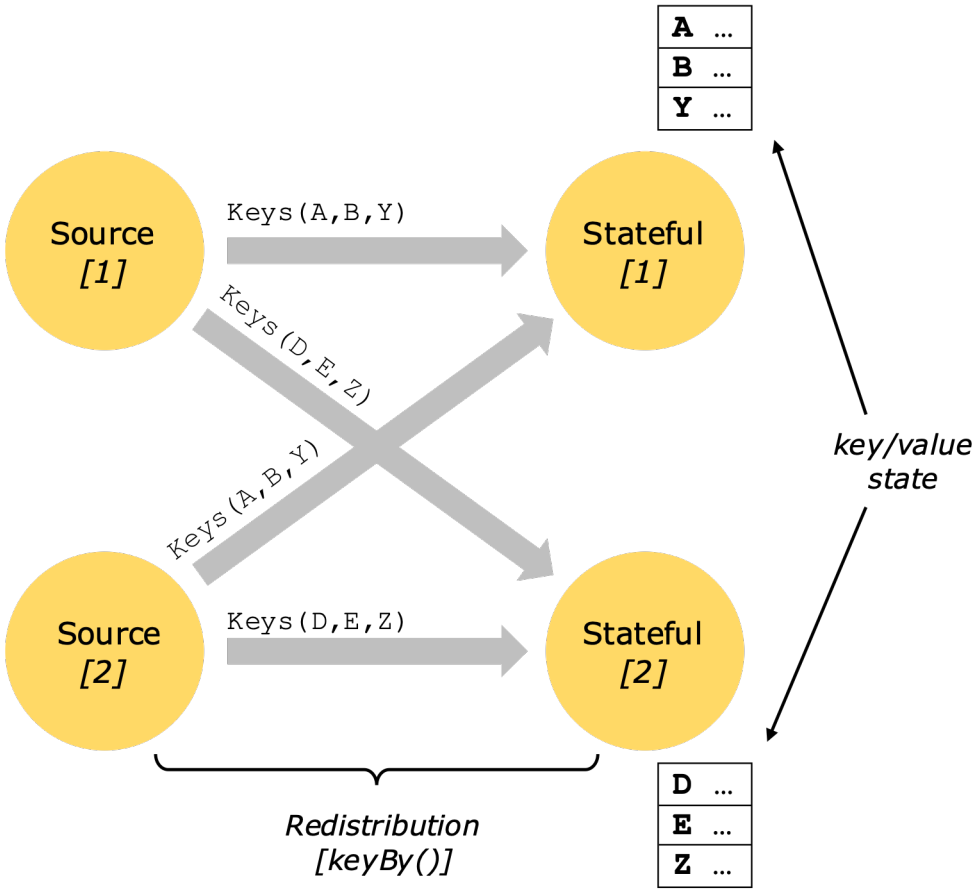
- Flink introduces the concept of allowed latency
  - By default the allowed latency is 0: if an element with event time t' arrives after a watermark with event time t > t', the element is discarded
  - If an element falls within its allowed latency, the results of the computation are updated accordingly and re-forwarded downstream

- In our examples, we will always use default sources that emit watermarks automatically
  - We will not change the watermarking policies

Stateful operators

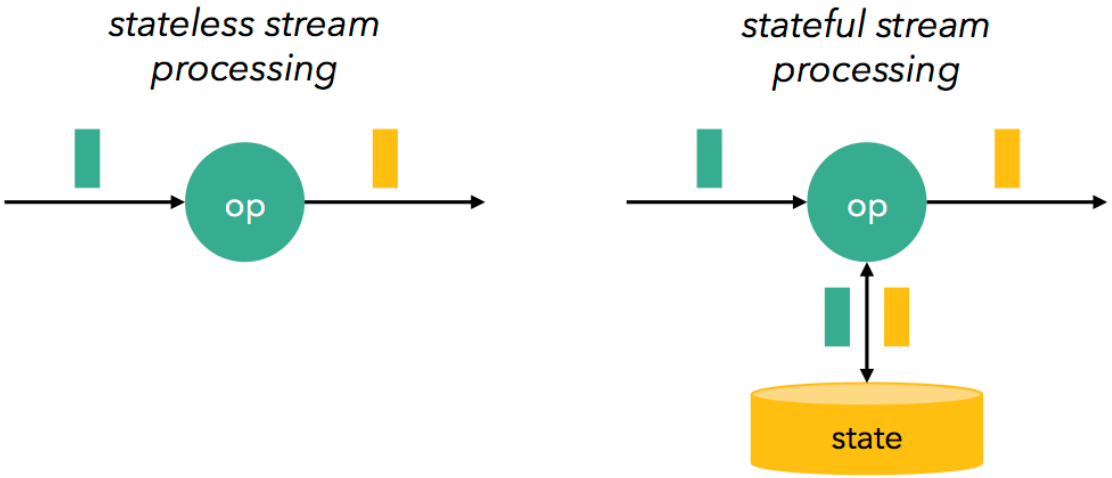
- Some operators look at an individual element at a time
  - Stateless operators
- Other operators record some state across multiple elements
  - Stateful operators
  - State stored partitioned by key
    - Think of it as an embedded key-value store

Stateful operators



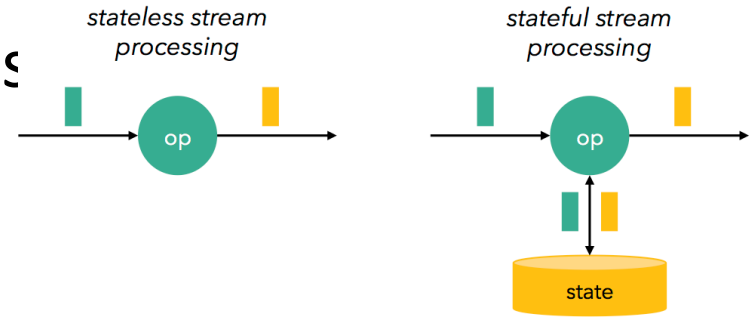
Fault tolerance

- Flink implements fault tolerance using
  - Checkpointing
  - Stream reply upon failure
    - Assumes that sources can record and reply past elements



Fault tolerance

- Checkpointing starts from a source
- Barriers forwarded to all downs operators
- Upon receiving a marker an operator saves its internal state
  - Forwards the marker downstream when it has received the marker from all incoming streams



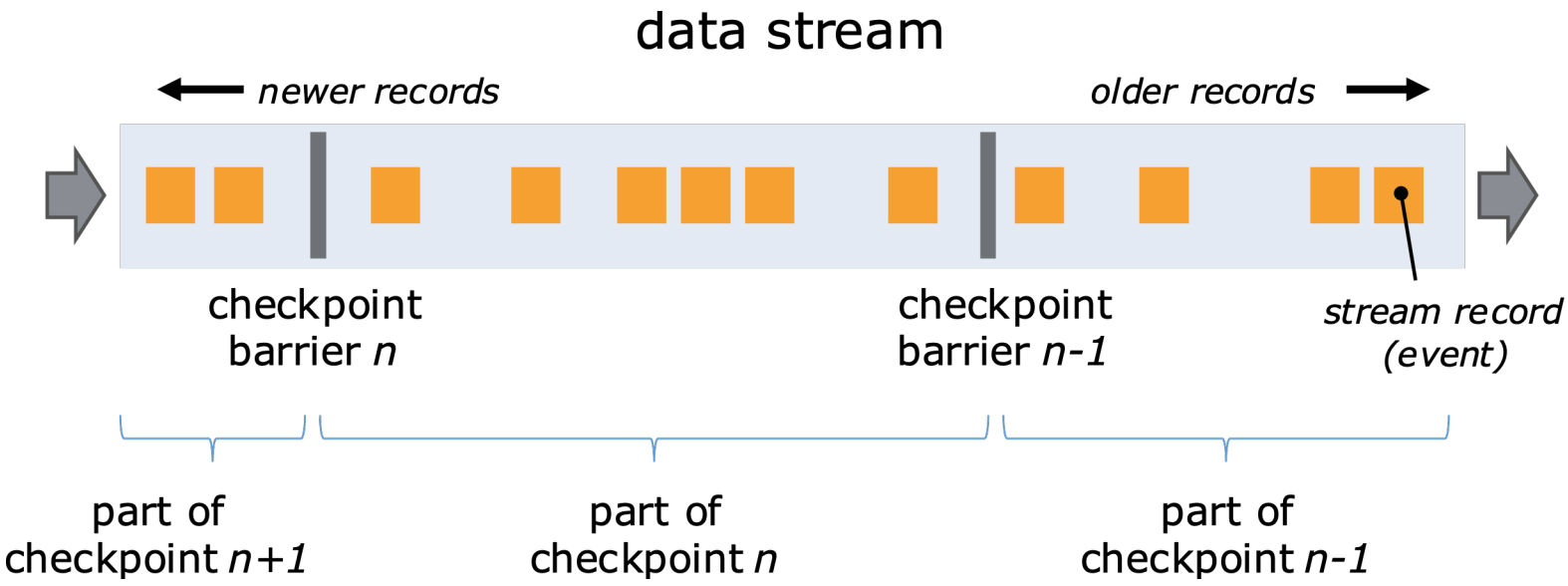
Fault tolerance

- In the case of failure …
- … Flink restores the state of each operator to the last checkpoint that reached the sinks …
- … and retransmits subsequent data

Fault tolerance in detail: barriers

- Flink injects barriers that flow with other elements as part of the data stream
- Barriers never overtake other elements
  - A barrier separates the streaming elements in the data stream into the set of records that goes into the current snapshot and the set of records that goes in the next snapshot
- Barriers do not interrupt the flow of the stream
- Multiple barriers from multiple snapshots can be in the stream at the same time
  - Meaning that various snapshots may happen concurrently

Fault tolerance in detail: barriers



Fault tolerance in detail: barriers

- Barriers are injected into the parallel data flow at the stream sources

- The point where barriers are injected is the position in the source stream up to which the snapshot covers the data
  - For example, in Apache Kafka, this position would be the last record's offset in the partition

- The position is reported to a coordinator (the JobManager)

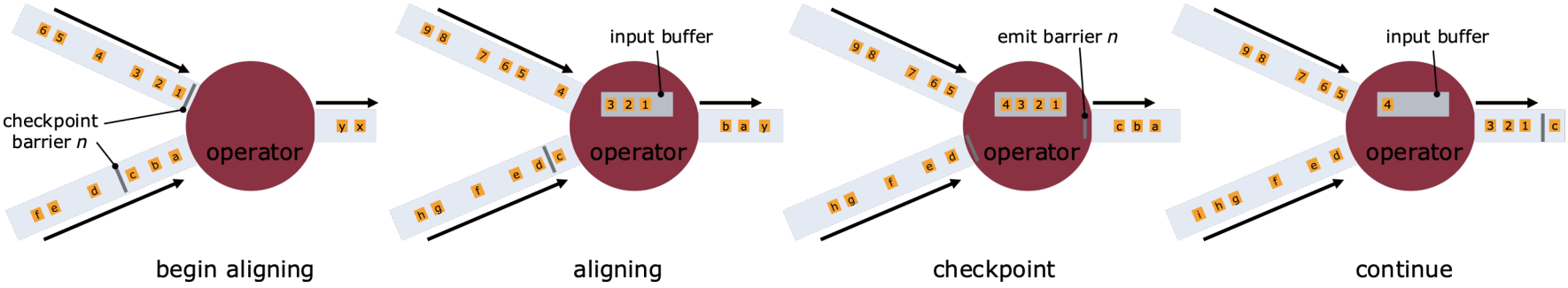
Fault tolerance in detail: barriers

- Barriers flow downstream
- When an intermediate operator has received a barrier for snapshot *n* from all of its input streams, it acknowledges snapshot *n* to the coordinator
- After all the sinks have acknowledged a snapshot, it is considered completed
- Once a snapshot n has been completed, the job will never ask the source again for elements that contributed to that snapshot

Fault tolerance in detail: aligning

- Operators that receive more than one input stream need to align the input streams on the snapshot barriers
  - As soon as the operator receives snapshot barrier n from an incoming stream, it cannot process any further elements from that stream until it has received the barrier n from the other input streams
  - Streams that report barrier n are temporarily set aside: records that are received from these streams are buffered
  - Once the last stream has received barrier n, the operator emits all pending outgoing elements, and then emits snapshot n barriers itself
  - After that, it resumes processing elements from all input streams, processing elements from the input buffers before processing elements from the streams

Fault tolerance in detail: aligning



Fault tolerance in detail: state

- When operators contain any form of state, this state must be part of the snapshots as well
  - User defined state: this is state that is created and modified directly by the transformation functions
  - System state: this state refers to data buffers that are part of the operator's computation
    - E.g., window buffers

Fault tolerance in detail: state

- Operators snapshot their state when they received all snapshot barriers from their input streams, before emitting the barriers downstream

- The state is saved in a state backend
  - Distributed filesystem (HDFS)
  - Database
  - …

Fault tolerance in detail: asynchronous snapshotting

- The above mechanism implies that operators stop processing input records while they are storing a snapshot of their state in the state backend
  - This synchronous mechanism can be expensive

- In practice, Flink adopts optimizations that enable asynchronous snapshotting
  - E.g., use of copy-on-write data structures

Fault tolerance in detail: recovery

- Upon failure, Flink selects the latest completed checkpoint *n*
- The system re-deploys the entire distributed dataflow and gives each operator the state that was snapshotted as part of checkpoint *n*
- The sources are set to restart reading the stream from the first element that was not part of checkpoint *n*

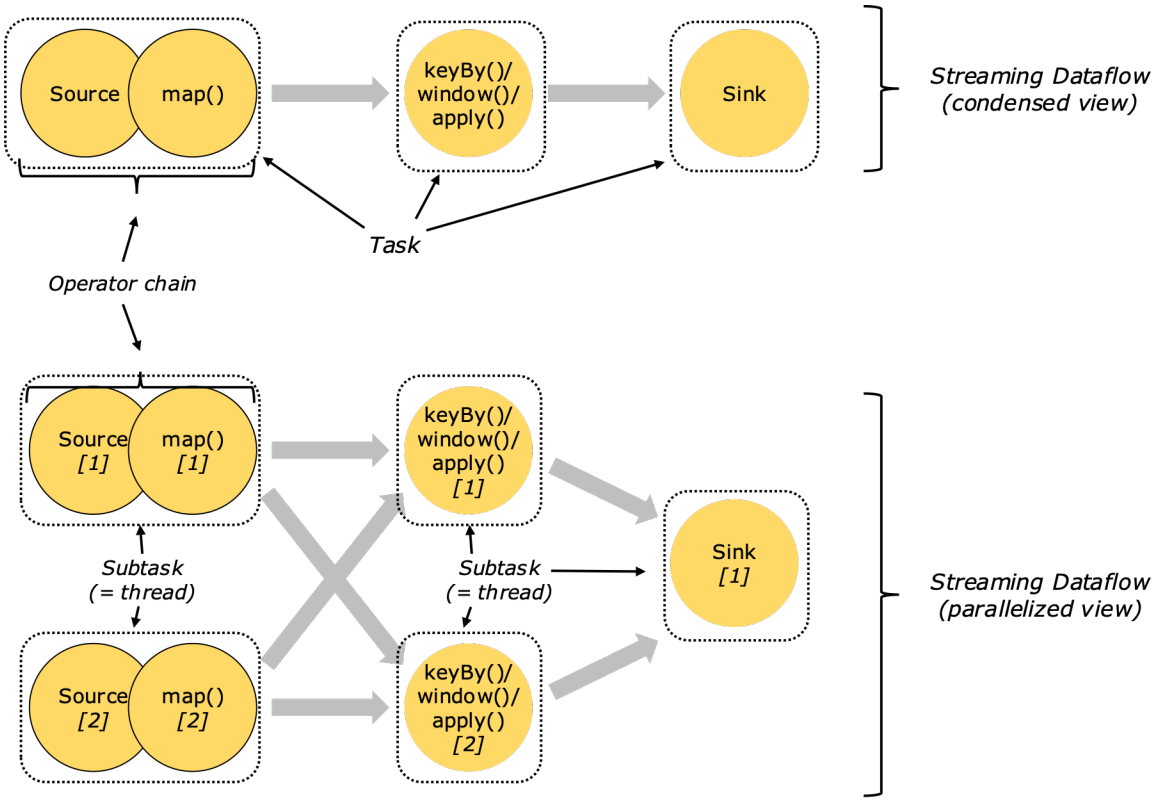
Dataset (batch) processing

- Flink executes batch programs as a special case of streaming ones
  - A dataset is treated internally as a (bounded) data stream
- Some minor differences
  - No checkpointing (complete re-execution)
  - Stateful operators store their state in memory

Distributed execution

- Flink chains operator subtasks together into tasks
- Each task is executed by one thread
- Optimization
  - Reduces inter-thread (or inter-machine) communication
- Chaining only possible with one-to-one streams

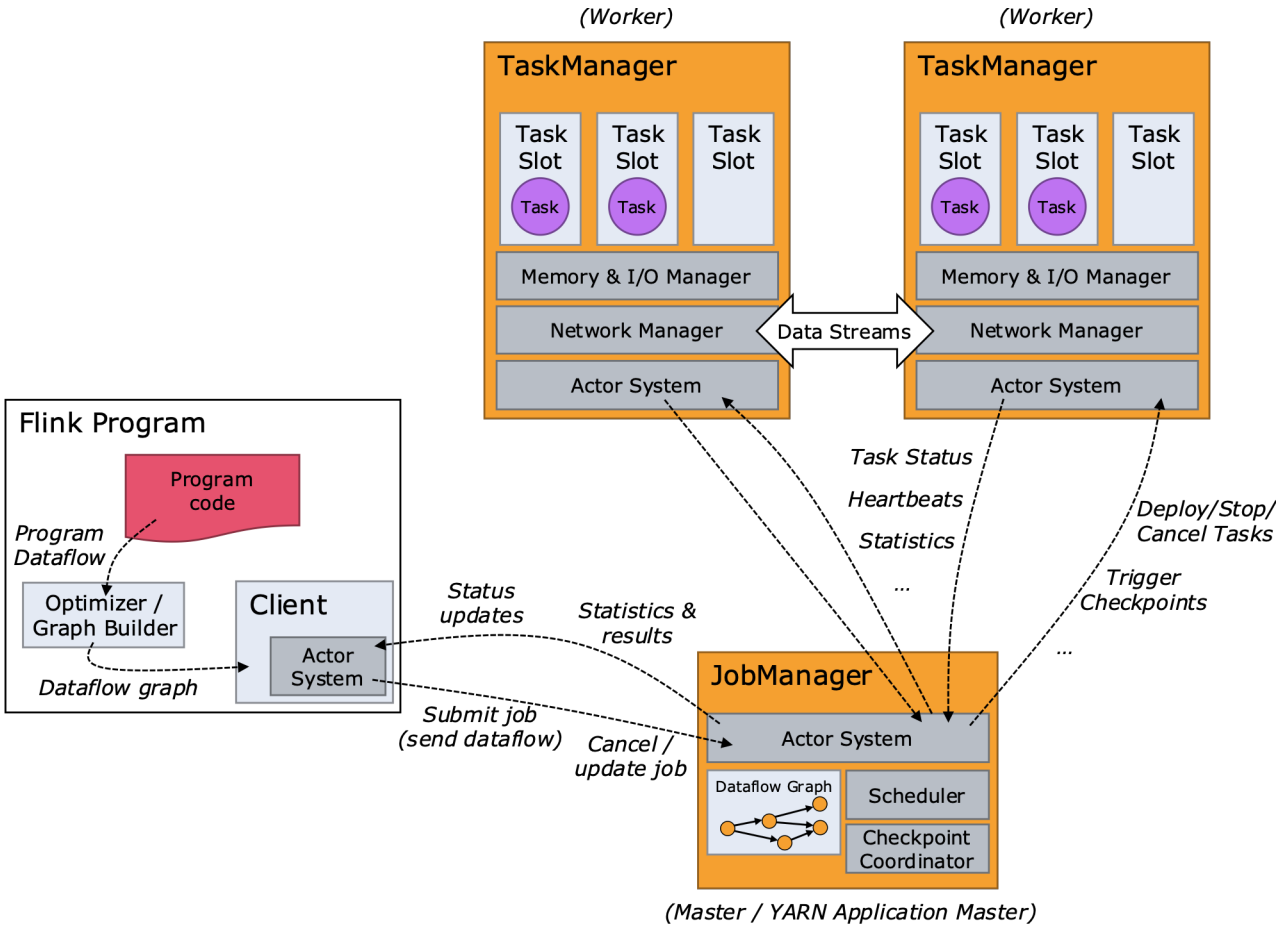
Distributed execution



Distributed execution

- The system includes
  - A job manager: the master that accepts jobs and distributes them to workers
    - Multiple job managers possible for high availability
  - Task managers: the workers that execute the tasks of a dataflow
- Clients submit jobs to the job manager as jar files

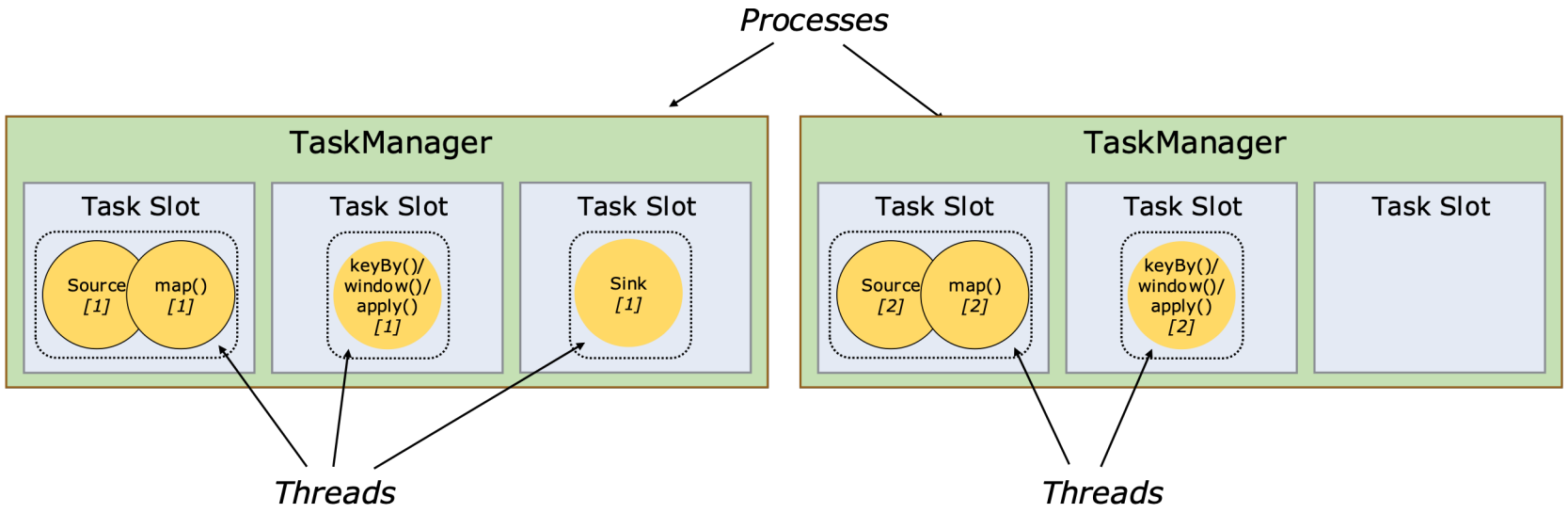
Distributed execution



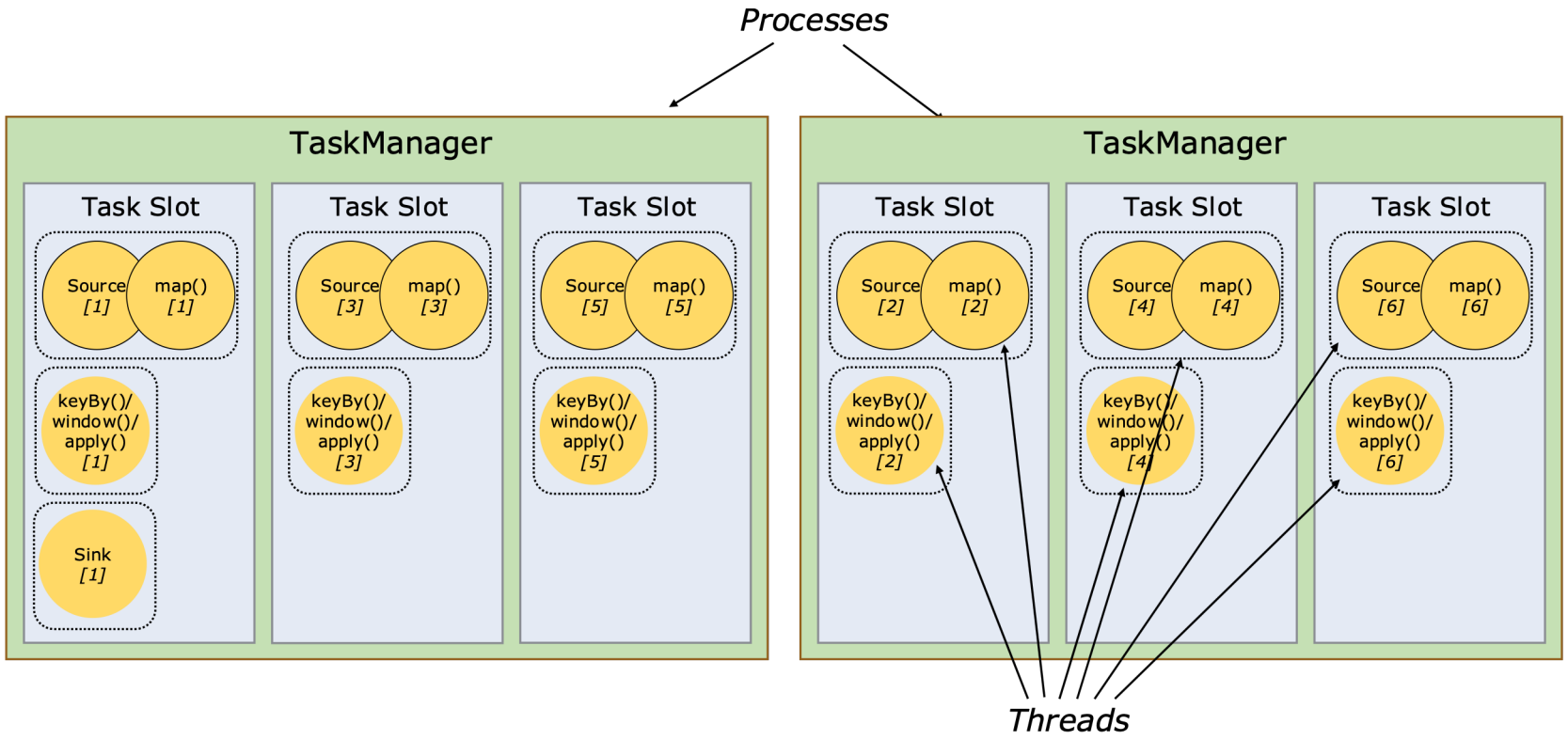
Resources

- Each TaskManager is a JVM process
  - It has n task slots to accept tasks
  - Task slots represent resources (memory and CPU cores)
  - Typical configuration: one slot per CPU core
- Tasks can share slots
  - Typical configuration: one parallel instance of each task on each slot

Resources


Processes
TaskManager
Task Slot
Source
[1]
map()
[1]
Task Slot
keyBy()/
window()/
apply()
[1]
Task Slot
Sink
[1]
TaskManager
Task Slot
Source
[2]
map()
[2]
Task Slot
keyBy()/
window()/
apply()
[2]
Task Slot
Threads
Threads

Resources


Processes
TaskManager
Task Slot
Source [1]
map() [1]
keyBy()/ window()/ apply() [1]
Sink [1]
Task Slot
Source [3]
map() [3]
keyBy()/ window()/ apply() [3]
Task Slot
Source [5]
map() [5]
keyBy()/ window()/ apply() [5]
TaskManager
Task Slot
Source [2]
map() [2]
keyBy()/ window()/ apply() [2]
Task Slot
Source [4]
map() [4]
keyBy()/ window()/ apply() [4]
Task Slot
Source [6]
map() [6]
keyBy()/ window()/ apply() [6]
Threads

Hands on

- Make sure you have sbt (>= 0.13.13)
- Download Flink (compiled release)
- Create a Flink project template for Scala
  - https://ci.apache.org/projects/flink/flink-docs-stable/dev/projectsetup/scala_api_quickstart.html

Hands on

- We will modify the template
  - sbt compile / clean
  - sbt run to run in local mode
  - sbt package to create the jar file to upload

Hands on

- To start Flink (from the flink-version dir)
  - Check the config in conf/flink-conf.yaml
    - The number of task slots
    - The address of the JobManager
    - The (heap) memory for the TaskManagers (JVM)
  - Start a JobManager and a TaskManager
    - bin/start-cluster.sh
  - You can also start/stop them separately
    - bin/jobmanager.sh start | stop
    - bin/taskmanager.sh start | stop
    - TaskManagers contact the JobManager at the specified address

Hands on

- Connect to the monitoring interface
  - http://localhost:8081
- Submit a job (jar)
  - Using the Web interface, or
  - bin/flink run my_job.jar

Hands on

- You can also run your code in a local execution environment
  - Single JVM
  - Very useful for initial testing and debugging

- When you use the factory method `Environment.getExecutionEnvironment` the concrete execution environment depends on the execution context
  - Local, cluster

Exercise

- Implement a word count application
  - Read input data from file
    - `ExecutionEnvironment` has a `readTextFile(String)` method that returns a `DataSource[String]`
  - For each line, produce a list of <word, count> tuples
  - Group the tuples by word
  - Perform the final reduction
  - Print on the screen

Exercise

- Now do the same in a streaming context
  - How can we “count” if the input stream is unbounded
  - Try to use windows!
    - Time based
    - Count based

Exercise

- Given two datasets (stored on two csv files)
  - Bank deposits (person, account, amount)
  - Bank withdrawal (person, account, amount)
- Print the set of accounts with a negative balance
- Print the person who withdrew the maximum amount of money, in total, even from multiple accounts

Fault tolerance

- Flink achieves fault tolerance through checkpointing
  - How does Flink know about the internal state of our operators?
  - In fact, it does not …
  - If we want to persist our own user-defined operator state, we need to save it in special data structures that are stored as part of checkpointing

Fault tolerance

- Two types of state
  - Operator state: one state element for each operator instance
  - Keyed state: one state element for each key
    - Only allowed on keyed streams
    - Common data structures
      - `ValueState[T]`
      - `ListState[T]`
      - `MapState[T]`
      - …

Fault tolerance hands on

- We create a simple example, where we implement our own counting window
  - Stream of input text lines
  - We partition the lines by their first letter
  - We sum the length of the lines in a tumbling window of size W
  - When the window is full, we output the sum

Fault tolerance hands on

- Our operator needs to keep track of the current count and sum

- We can use two `ValueState[Int]`
  - This creates a different value for each input key

Fault tolerance hands on

- We need to enable checkpointing on the streaming environment, specifying how frequently the checkpointing algorithm runs

```
environment.enableCheckpointing(1000)
```

- We can configure the state backend
  - E.g., save to file

```
environment.setStateBackend(…)
```

- We can configure the restart strategy
  - E.g., try to restart at most 10 times every 5 seconds

```
environment.setRestartStrategy(
RestartStrategies.fixedDelayRestart(10,
5000));
```

Fault tolerance hands on

- Now we can check that everything works
- We can stop a taskmanager
  - bin/taskmanager.sh stop
- We restart the taskmanager
  - bin/taskmanager.sh start
- We can verify that indeed the sum and the count are persisted!

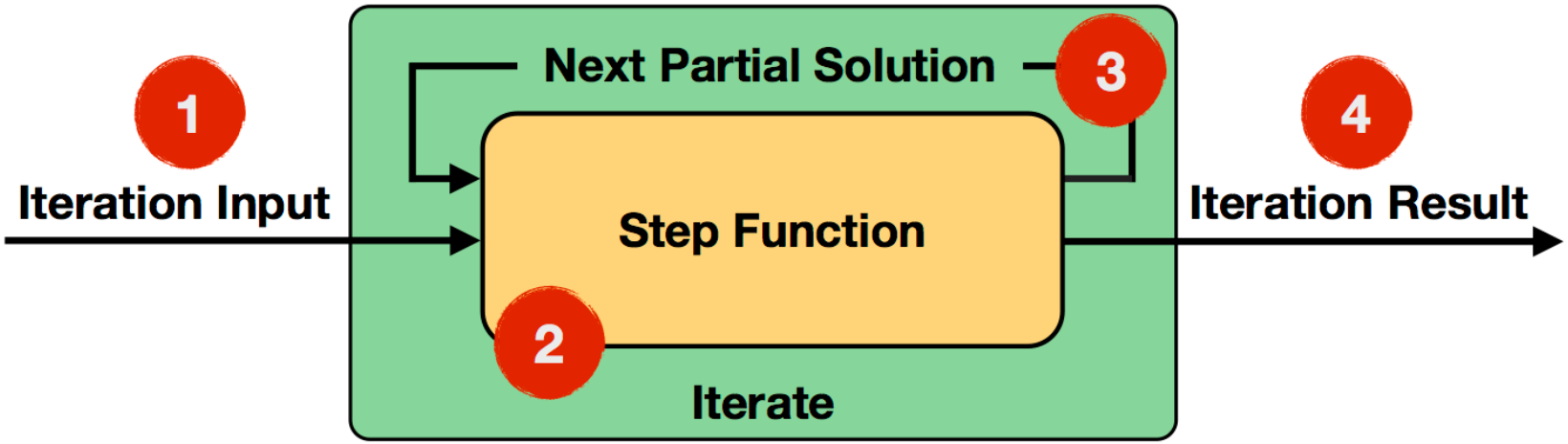
Iterative jobs hands on

- Iterative algorithms are common in many domains for data analysis
  - Graph-based data structures
  - Machine learning
- Flink implements iterative algorithms by defining a step function and embedding it into a special iteration operator
- Two variants
  - Iterate: in each iteration, the step function consumes the entire input
  - Iterate delta: deals with incremental iterations, which selectively modify elements of their solution rather than fully recompute it

Iterative jobs hands on

	Iterate	Iterate Delta
Iteration input	Partial solution	Workset and solution set
Step function	Arbitrary dataflow	
State update	Next partial solution	• Next workset • Changes to the solution set
Iteration result	Last partial solution	Solution set after the last iteration
Termination	• Maximum number of iterations • Custom aggregator convergence	• Maximum number of iterations • Empty workset • Custom aggregator convergence

Iterate operator



Iterate operator

1. Iteration input: initial input for the first iteration from a data source or previous operators

2. Step function: executed in each iteration
   - Arbitrary data flow operators

3. Next partial solution: in each iteration, the output of the step function will be fed back into the next iteration

4. Iteration result: output of the last iteration is used as input to the following operators (or sinks)

Example: sum words length

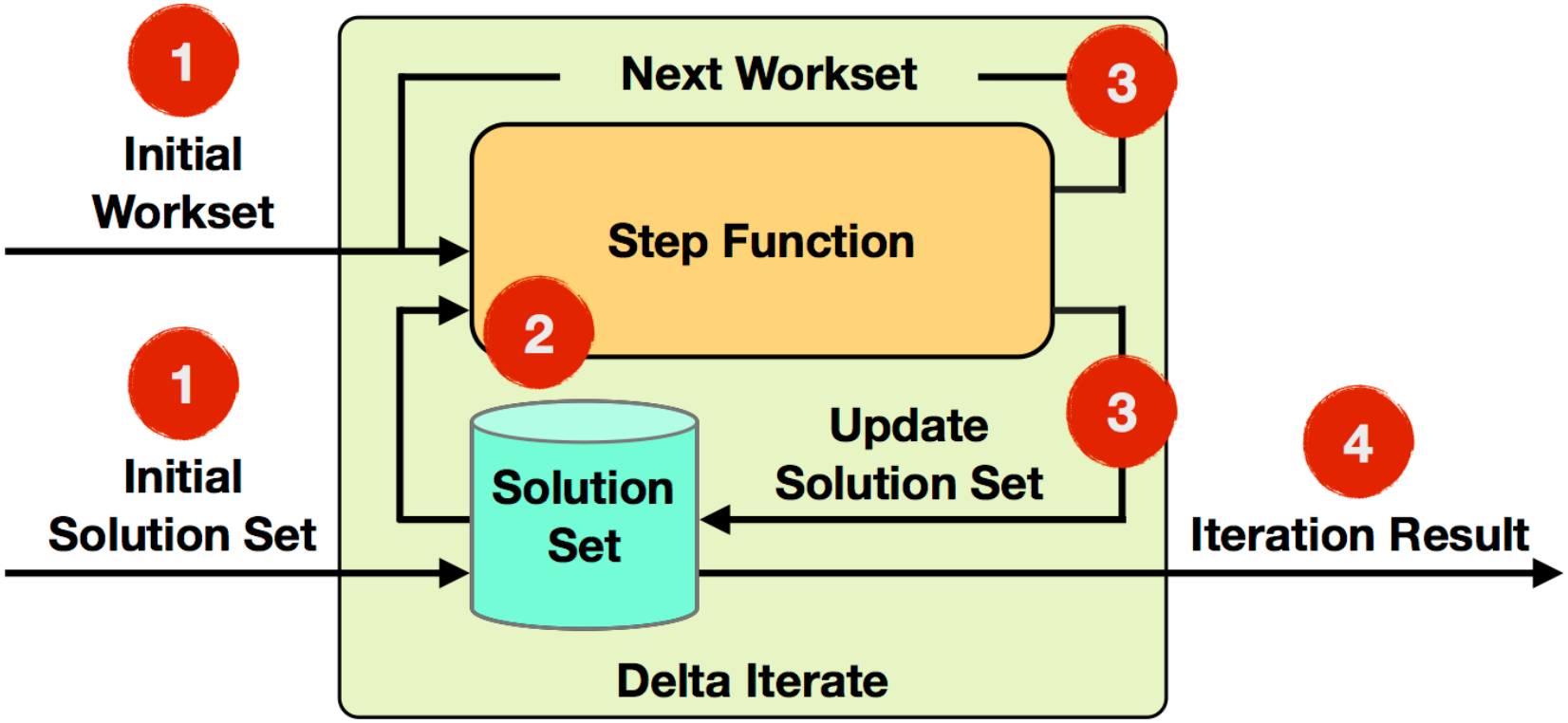
1. Iteration input: entire line, index=0, sum=0

2. Step function: if there is a word at the given index of the line, add its length to the sum

3. Next partial solution: the output of the step function

4. Iteration result: the partial solution when we reach the maximum number of iterations or when we processed all the words in all the lines

Delta iterate operator



Delta iterate operator

1. Iteration input: initial input for the first iteration from a data source or previous operators

2. Step function: executed in each iteration
   - Arbitrary data flow operators

3. Next workset/update solution set: the next workset will be fed back into the next iteration, while the solution set will be updated with the updates

4. Iteration result: solution set after the last iteration is used as input to the following operators (or sinks)

Example: reachable nodes

1. Iteration input: initial node

2. Step function: compute the nodes reachable from the last discovered nodes

3. Next workset/update solution set: newly discovered nodes

4. Iteration result: set of all reachable nodes

Iterative computations in DataStreams

- DataStreams also offer an iterate() method to perform iterative computations

- General pattern
  - Perform a set of transformations
  - Filter the events to send downstream
  - Filter the events to send back for further iterations
    - Pass them to the closeWith method

- Example: try to implement a streaming application that iteratively splits messages in two
  - Until the message length is below a given threshold
  - Or until the message is composed of a single word

TableAPI and SQL

- Table as central concept that serves as input and output of queries
  - Integrated with both DataSets and DataStreams
  - A table can be derived from a DataSet …
  - … or from a DataStream
    - The stream represents the set of changes that occur to the table
- Standard SQL queries to derive tables (views) from other tables
  - Derived tables reactively change with the original ones

Libraries

- Complex Event Processing
  - Detect patterns in input event streams
- Graphs
  - Implementation of common algorithms
  - Support to define iterative algorithms
- Machine learning
  - Supervised learning algorithms
  - Unsupervised learning algorithms

Resources

- Slides based on the official Flink documentation
- Visit
  - https://ci.apache.org/projects/flink/flink-docs-stable/
- More examples on the official GitHub repository
  - https://github.com/apache/flink/tree/master/flink-examples

Questions?