Extending Flink's Streaming APIs

Kostas Kloudas
@KLOUBEN_K
dataArtisans

Flink Forward San Francisco
April 11, 2017

dataArtisans

Original creators of **Apache Flink®**

dA PLATFORM

Providers of the **dA Platform**, a supported Flink distribution

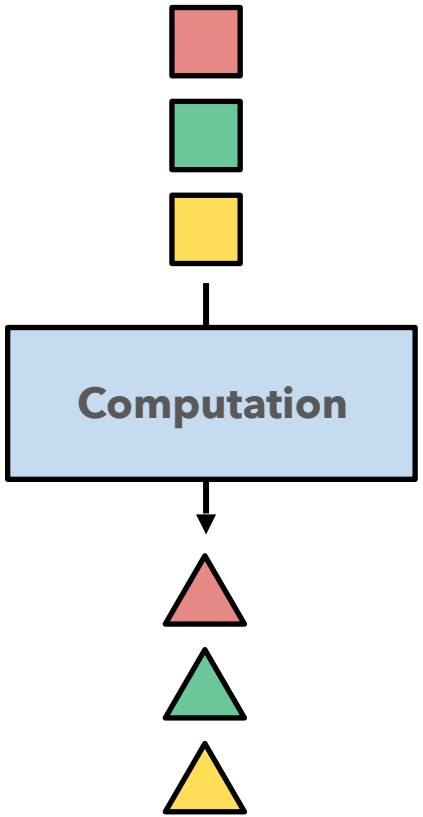
Extensions to the DataStream API

Extensions to the DataStream API

- ProcessFunction for Low-level Operations
- Support for Asynchronous I/O

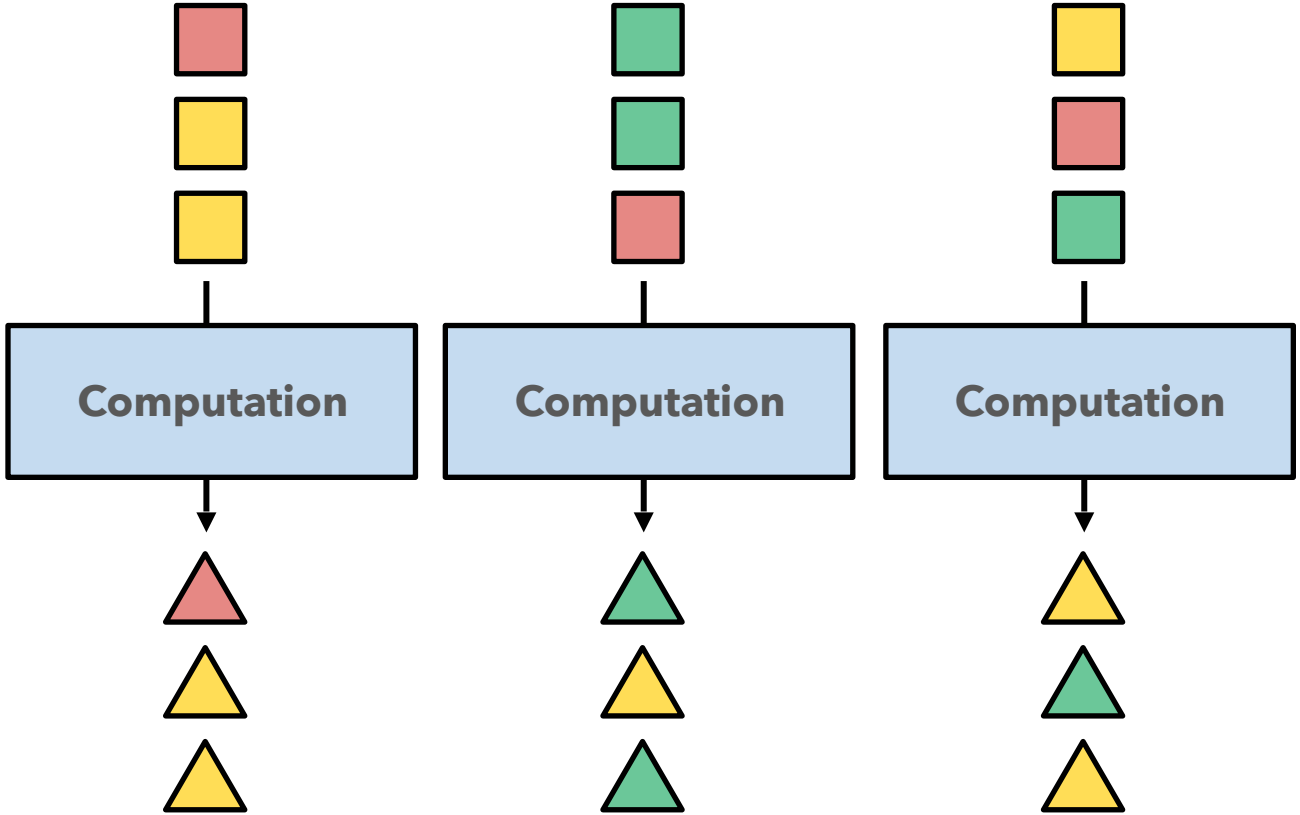
ProcessFunction

Stream Processing



Computations on never-ending "streams" of events

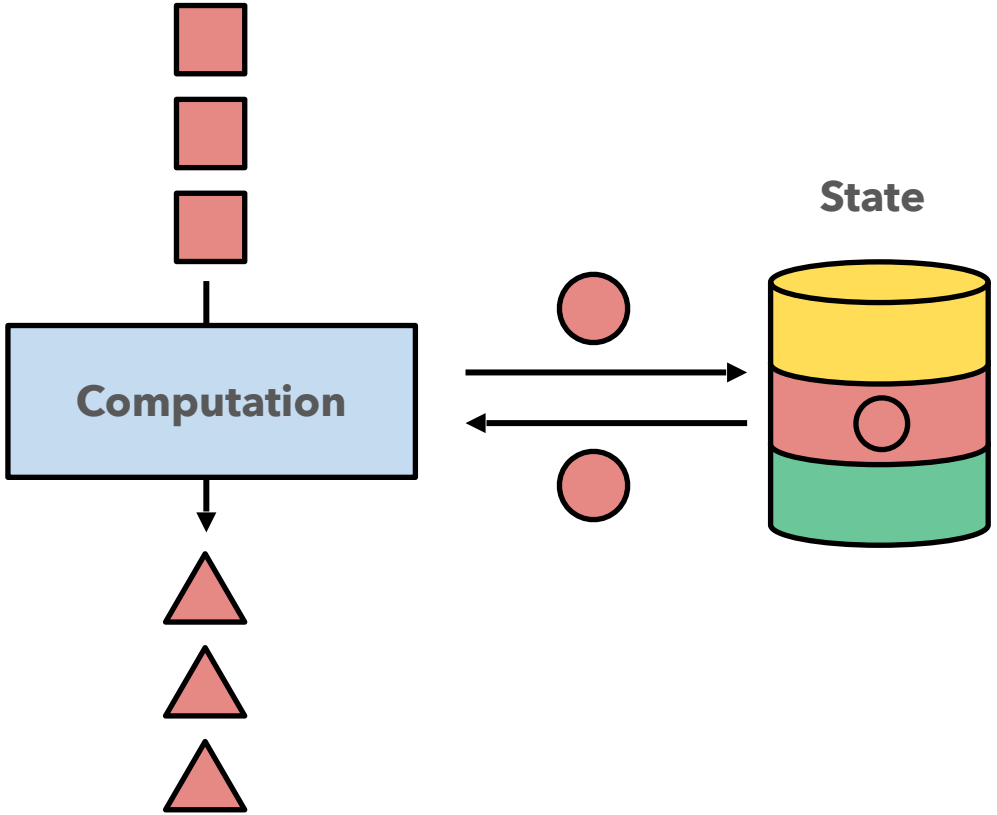
Distributed Stream Processing



Computation spread across many machines

Stateful Stream Processing

Result depends on history of stream



Stream Processing Engines

- Time:
  - handle infinite streams
  - with out-of-order events
- State:
  - guarantee fault-tolerance (distributed)
  - guarantee consistency (infinite streams)

ProcessFunction

- Gives access to all basic building blocks:
  - Events
  - Fault-tolerant, Consistent State
  - Timers (event- and processing-time)
  - Side Outputs

Common Usecase Skeleton A

- On each incoming element:
  - update some state
  - register a callback for a moment in the future
- When that moment comes:
  - Check a condition and perform a certain action, *e.g.* emit an element

Before the ProcessFunction

- Use built-in windowing:
  - +Expressive
  - +A lot of functionality out-of-the-box
  - - Not always intuitive
  - - An overkill for simple cases
- Write your own operator:
  - - Too many things to account for

ProcessFunction

- Simple yet powerful API:

```
/**
  * Process one element from the input stream.
  */
void processElement(I value, Context ctx, Collector<O> out) throws Exception;


/**
  * Called when a timer set using {@link TimerService} fires.
  */
void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out) throws
Exception;
```

ProcessFunction

- Simple yet powerful API:

```
/**
  * Process one element from the input stream.
  */
void processElement(I value, Context ctx, Collector<O> out) throws Exception;


/**
  * Called when a timer set using {@link TimerService} fires.
  */
void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out) throws
Exception;
```

A collector to emit result values

ProcessFunction

- Simple yet powerful API:

```
/**
  * Process one element from the input stream.
  */
void processElement(I value, Context ctx, Collector<O> out) throws Exception;


/**
  * Called when a timer set using {@link TimerService} fires.
  */
void onTimer(long timestamp, OnTimerContext ctx, Collector<O> out) throws
Exception;
```

1. Get the timestamp of the element
2. Register and use side outputs
3. Interact with the TimerService to:
   - query the current time
   - register timers

1. Do the above
2. Query if we are on Event or Processing time

ProcessFunction: example

- Requirements:
  - maintain counts per incoming key, and
  - emit the key/count pair if no element came for **the key** in the last 100 ms (in event time)

ProcessFunction: example

- Implementation sketch:
  - Store the **count**, **key** and **last mod timestamp** in a `ValueState` (scoped by key)
  - For each record:
    - update the counter and the last mod timestamp
    - register a timer 100ms from “now” (in event time)
  - When the timer fires:
    - check the timer’s timestamp against the last mod time for that key and
    - emit the key/count pair if they differ by 100ms

ProcessFunction: example

```
public class MyProcessFunction extends
        ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>> {

    // define your state descriptors

    @Override
    public void processElement(Tuple2<String, Long> value, Context ctx,
                Collector<Tuple2<String, Long>> out) throws Exception {
        // update our state and register a timer
    }

    @Override
    public void onTimer(long timestamp, OnTimerContext ctx,
                Collector<Tuple2<String, Long>> out) throws Exception {
        // check the state for the key and emit a result if needed
    }
}
```

ProcessFunction: example

```
public class MyProcessFunction extends
        ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>> {

    // define your state descriptors
    private final ValueStateDescriptor<CounterWithTS> stateDesc =
        new ValueStateDescriptor<>("myState", CounterWithTS.class);









}
```

ProcessFunction: example

```
public class MyProcessFunction extends
        ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>> {

    @Override
    public void processElement(Tuple2<String, String> value, Context ctx,
                       Collector<Tuple2<String, Long>> out) throws Exception {

        ValueState<MyStateClass> state =
getRuntimeContext().getState(stateDesc);
        CounterWithTS current = state.value();
        if (current == null) {
            current = new CounterWithTS();
            current.key = value.f0;
        }
        current.count++;
        current.lastModified = ctx.timestamp();
        state.update(current);
        ctx.timerService().registerEventTimeTimer(current.lastModified + 100);
    }
}
```

ProcessFunction: example

```
public class MyProcessFunction extends
        ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>> {

    @Override
    public void onTimer(long timestamp, OnTimerContext ctx,
                 Collector<Tuple2<String, Long>> out) throws Exception {

        CounterWithTS result = getRuntimeContext().getState(stateDesc).value();

        if (timestamp == result.lastModified + 100) {
            out.collect(new Tuple2<String, Long>(result.key, result.count));
        }
    }
}
```

ProcessFunction: example

```
stream.keyBy("key")
.process(new MyProcessFunction())
```

ProcessFunction: Side Outputs

- Additional (to the main) output streams

- No type limitations
  - each side output can have its own type

ProcessFunction: example+

- Requirements:
  - maintain counts per incoming key, and
  - emit the key/count pair if no element came for **the key** in the last 100 ms (in event time)
  - in other case, if the count > 10, send the key to a side-output named gt10

ProcessFunction: example+

```
final OutputTag<String> outputTag = new OutputTag<String>("gt10"){};

SingleOutputStreamOperator<Tuple2<String, Long>> mainStream = input.process(
    new ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>>() {

    @Override
    public void onTimer(long timestamp, OnTimerContext ctx,
                        Collector<Tuple2<String, Long>> out) throws Exception {

       CounterWithTS result =
getRuntimeContext().getState(adStateDesc).value();
       if (timestamp == result.lastModified + 100) {
           out.collect(new Tuple2<String, Long>(result.key, result.count));

       } else if (result.count > 10) {
           ctx.output(outputTag, result.key);
       }
}

DataStream<String> sideOutputStream = mainStream.getSideOutput(outputTag);
```

ProcessFunction: example+

```
final OutputTag<String> outputTag = new OutputTag<String>("gt10"){};

SingleOutputStreamOperator<Tuple2<String, Long>> mainStream = input.process(
    new ProcessFunction<Tuple2<String, String>, Tuple2<String, Long>>() {

    @Override
    public void onTimer(long timestamp, OnTimerContext ctx,
                        Collector<Tuple2<String, Long>> out) throws Exception {

      CounterWithTS result =
getRuntimeContext().getState(adStateDesc).value();
      if (timestamp == result.lastModified + 100) {
          out.collect(new Tuple2<String, Long>(result.key, result.count));

      } else if (result.count > 10) {
          ctx.output(outputTag, result.key);
      }
}

DataStream<String> sideOutputStream = mainStream.getSideOutput(outputTag);
```

ProcessFunction

- Applicable to Keyed streams
- For Non-Keyed streams:
  - group on a dummy key if you need the timers
    - BEWARE: parallelism of 1
  - Use it directly without the timers
- `CoProcessFunction` for low-level joins:
  - Applied on two input streams

Asynchronous I/O

Common Usecase Skeleton B

- On each incoming element:
  - extract some info from the element (e.g. key)
  - query an external storage system (DB or KV-store) for additional info
  - emit an enriched version of the input element

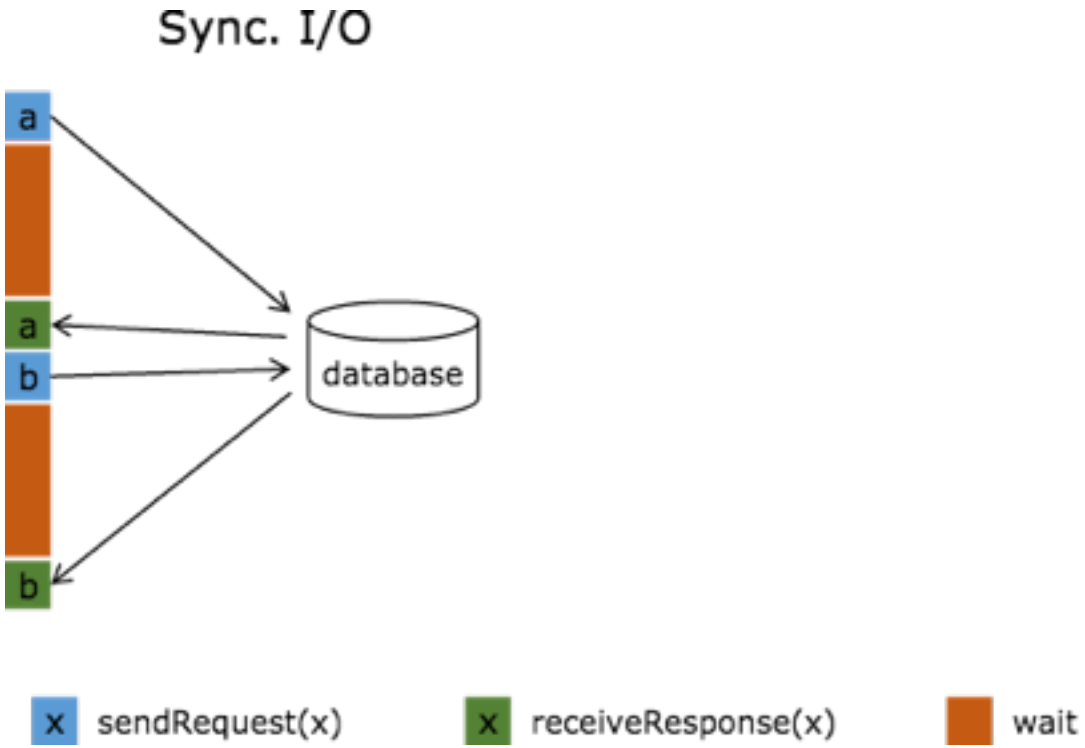
Before the AsuncIO support

- Write a `MapFunction` that queries the DB:
  - +Simple
  - - Slow (synchronous access) or/and
  - - Requires high parallelism (more tasks)
- Write your own operator:
  - - Too many things to account for

Before the AsyncIO support

- Write a `MapFunction` that queries the DB:
  - +Simple
  - - Slow (synchronous access) or/and
  - - Requires high parallelism (more tasks)
- Write your own operator:
  - - Too many things to account for

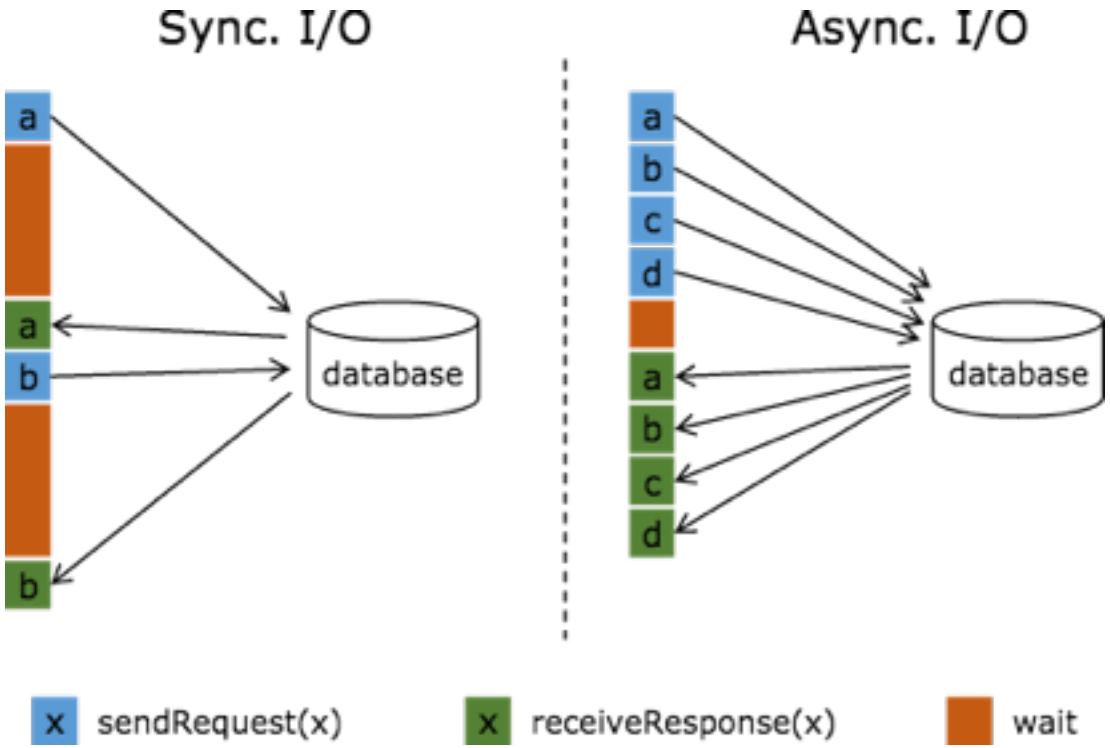
Synchronous Access



Synchronous Access

Communication delay can dominate application throughput and latency

Asynchronous Access



AsyncFunction

- Requirement:
  - a client that supports asynchronous requests
- Flink handles the rest:
  - integration of async IO with DataStream API
  - fault-tolerance
  - order of emitted elements
  - correct time semantics (event/processing time)

AsyncFunction

- Simple API:

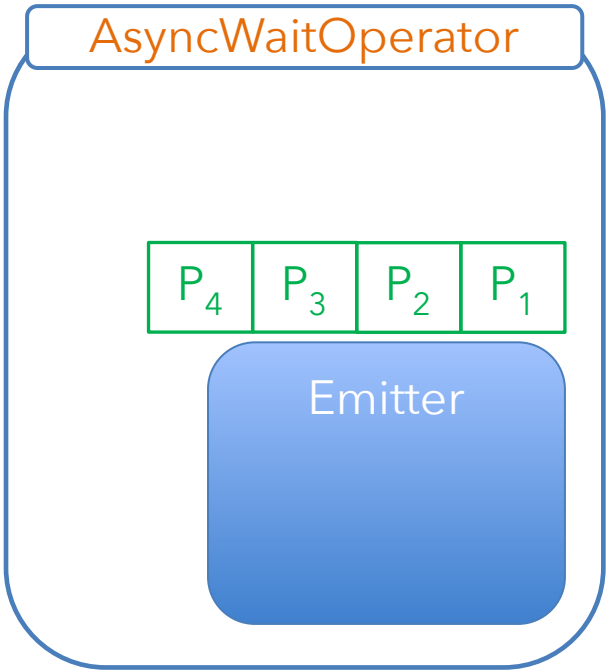
```
/**
  * Trigger async operation for each stream input.
  */
void asyncInvoke(IN input, AsyncCollector<OUT> collector) throws Exception;
```

- API call:

```
/**
  * Example async function call.
  */
DataStream<...> result = AsyncDataStream.(un)orderedWait(stream,
                    new MyAsyncFunction(), 1000, TimeUnit.MILLISECONDS, 100);
```

AsyncFunction

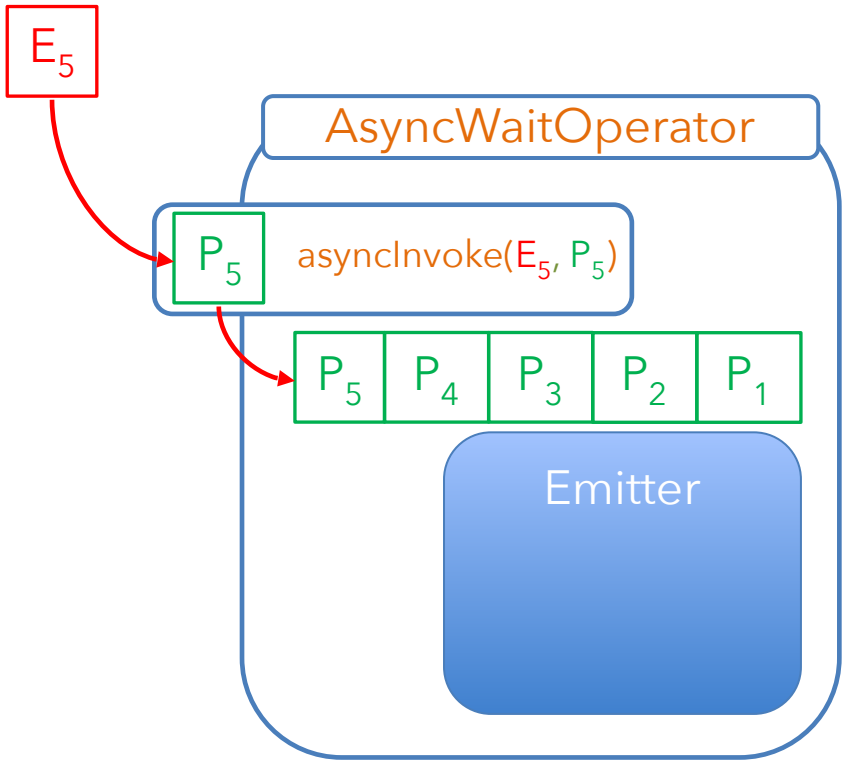
E_5



AsyncWaitOperator:

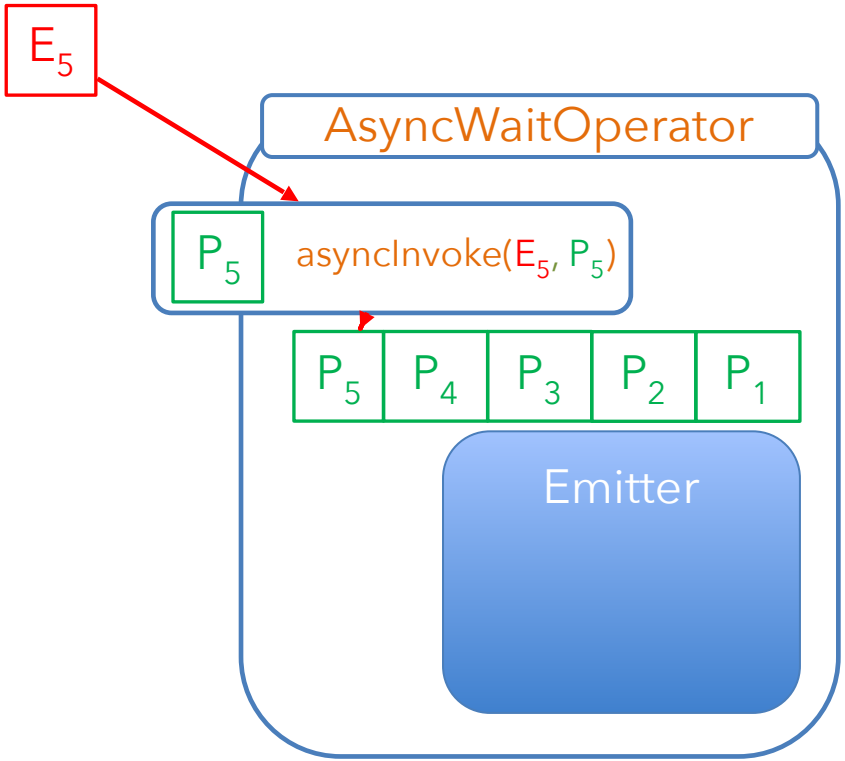
- a queue of “Promises”
- a separate thread (Emitter)

AsyncFunction



- Wrap E_5 in a "promise" P_5
- Put P_5 in the queue
- Call asyncInvoke(E_5, P_5)

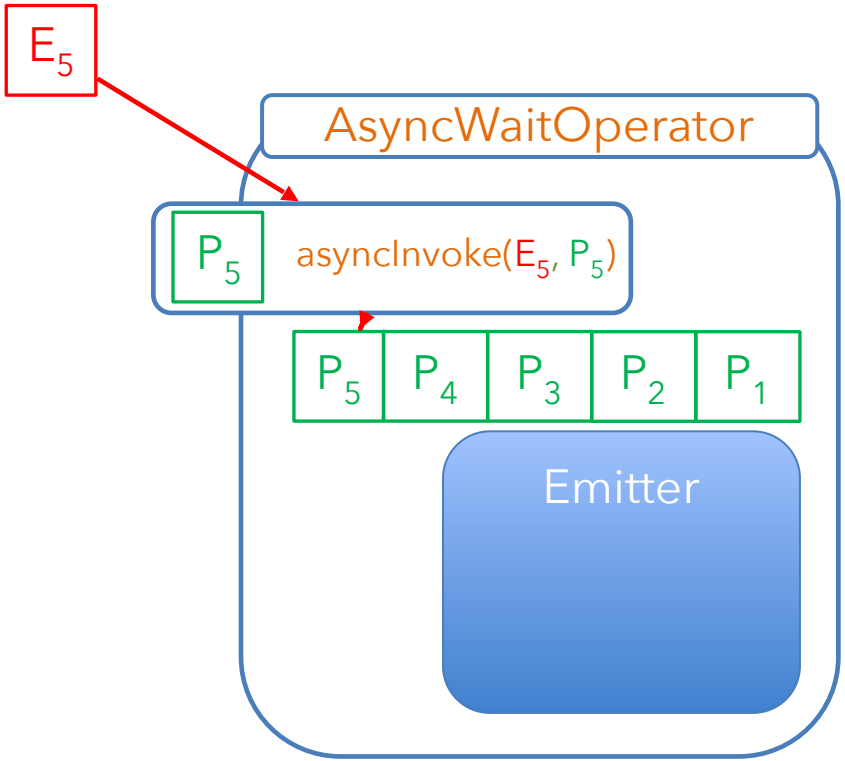
AsyncFunction



asyncInvoke(value, asyncCollector):

- a user-defined function
- value : the input element
- asyncCollector : the collector of the result (when the query returns)

AsyncFunction



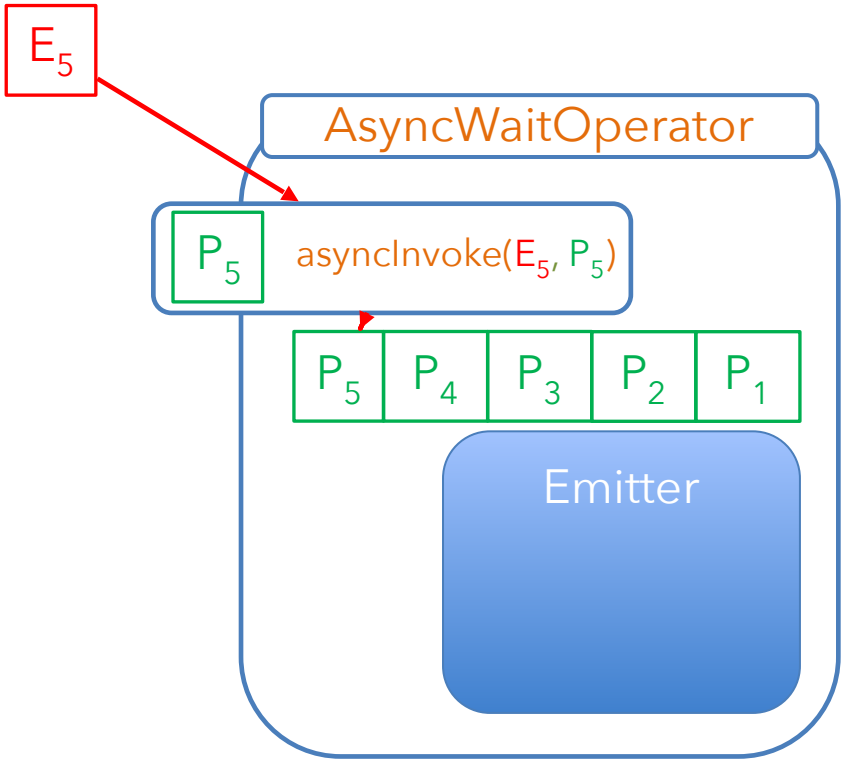
asyncInvoke(value, asyncCollector):

- a user-defined function
- value : the input element
- asyncCollector : the collector of the result (when the query returns)

```
Future<String> future = client.query(E5);

future.thenAccept((String result) -> {
P5.collect(
          Collections.singleton(
               new Tuple2<>(E5, result)));
});
```

AsyncFunction



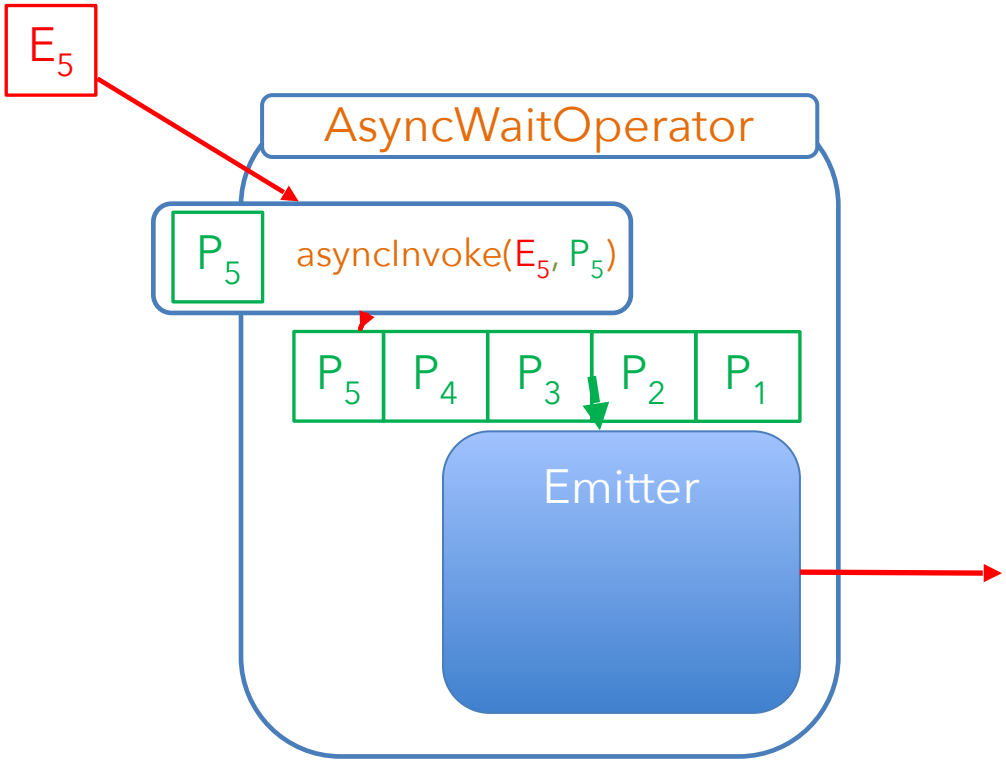
asyncInvoke(value, asyncCollector):

- a user-defined function
- value : the input element
- asyncCollector : the collector of the result (when the query returns)

```
Future<String> future = client.query(E5);

future.thenAccept((String result) -> {
P5.collect(
          Collections.singleton(
               new Tuple2<>(E5, result)));
});
```

AsyncFunction



Emitter:

- separate thread
- polls queue for *completed* promises (blocking)
- emits elements downstream

AsyncFunction

```
DataStream<Tuple2<String, String>> result =
      AsyncDataStream.(un)orderedWait(stream,
                      new MyAsyncFunction(),
                      1000, TimeUnit.MILLISECONDS,
                      100);
```

- our `asyncFunction`
- a timeout: max time until considered failed
- capacity: max number of in-flight requests

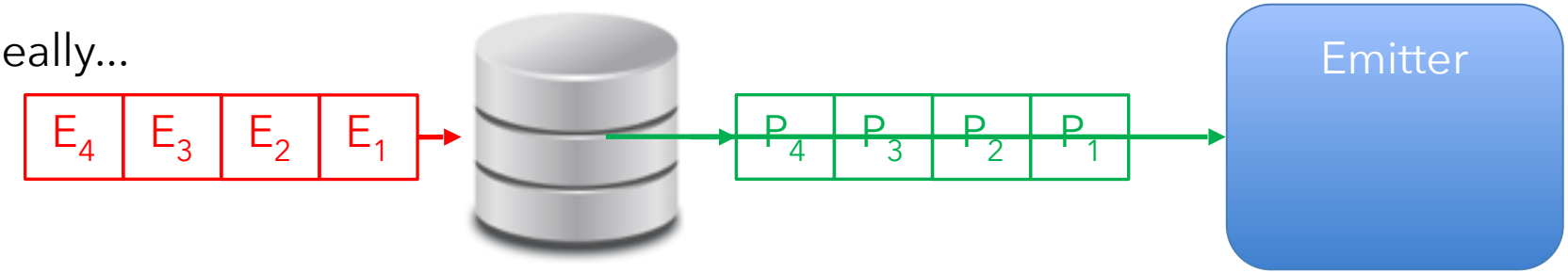
AsyncFunction

```
DataStream<Tuple2<String, String>> result =
      AsyncDataStream.(un)orderedWait(stream,
                    new MyAsyncFunction(),
                    1000, TimeUnit.MILLISECONDS,
                    100);
```

AsyncFunction

```
DataStream<Tuple2<String, String>> result =
      AsyncDataStream.(un)orderedWait(stream,
                  new MyAsyncFunction(),
                  1000, TimeUnit.MILLISECONDS,
                  100);
```

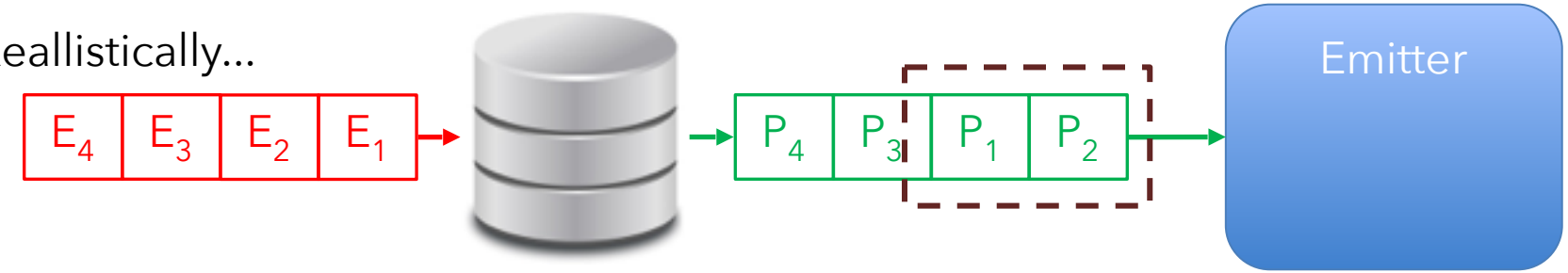
Ideally...



AsyncFunction

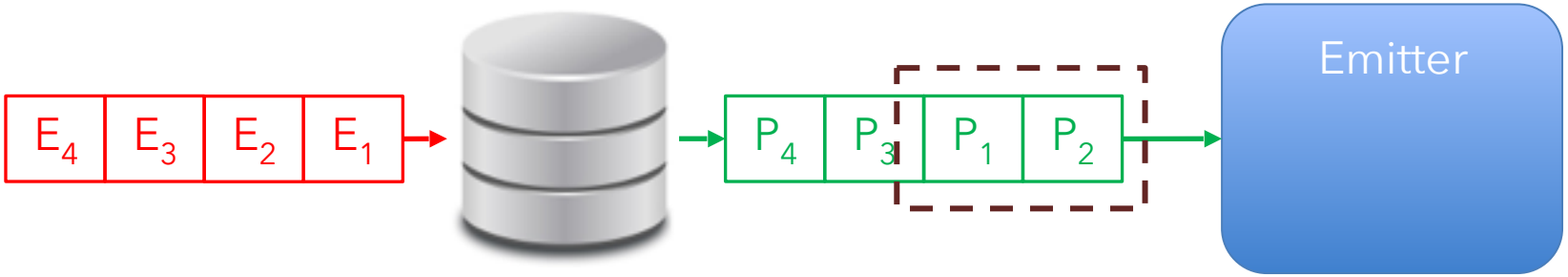
```
DataStream<Tuple2<String, String>> result =
      AsyncDataStream.unorderedWait(stream,
                  new MyAsyncFunction(),
                  1000, TimeUnit.MILLISECONDS,
                  100);
```

Reallistically...



...output ordered based on which request finished first

AsyncFunction



- `unorderedWait`: emit results in order of completion
- `orderedWait`: emit results in order of arrival

- **Always**: *watermarks never overpass elements and vice versa*

Documentation

- ProcessFunction:

https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/stream/process_function.html

https://ci.apache.org/projects/flink/flink-docs-release-1.3/dev/stream/process_function.html

- AsyncIO:

https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/stream/asyncio.html

Thank you!

@KLOUBEN_K

@ApacheFlink

@dataArtisans


FLINK
FORWARD
FLINK FORWARD IS COMING BACK TO BERLIN
SEPTEMBER 11-13, 2017
BERLIN.FLINK-FORWARD.ORG

We are hiring!

data-artisans.com/careers