ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Spring Term 2019 Dr. Vasiliki Kalavri

Data Stream Processing and Analytics

Assignment 5

In this assignment you will learn:
- How to implement custom logic by writing your own Timely operator
- How to store and maintain state inside an operator
- How to correctly handle out-of-order arrivals

Until now we've tinkered with simple streaming tasks which could be expressed by stitching together built-in transformations and generic operators like *map* and *filter*. Some of these pipelines were also pure computations where the outputs could be derived solely as a function of the input tuples being fed in. While solving real analytic tasks you will quickly realize the need to perform arbitrary computation and be able to store and access intermediate data. The main theme underlying this assignment is "What if the logic you'd like to implement doesn't exist in the form of a pre-canned operator?"

Reflecting back, you'll notice how even the simple "word count" topology had to maintain and update counters which were sneakily concealed inside the aggregation operator. The ability to recall previously observed data is important in the broader sense, for instance with window operators – which organizes tuples by their timestamps – and similarly with queryable operators (e.g. distinct) – which index their data by key for efficient retrieval. Operators also vary by their 'shape' and how they connect to the outside world. Applications which consumes multiple streams or produces outputs, such as a join or broadcast require additional support from the streaming runtime so that they can express how tuples should be interleaved and re-routed. Most dataflow systems provide extension points to handle these cases where custom logic is required by allowing you to break out of the dataflow model and write classical imperative programs with the usual for-loops and collection types.

In Timely, the `Operator` trait contains the required primitives to construct your own operators. We'll be using the `unary` pattern which is named as such because it receives a single input stream and produces a single output stream. The interface is quite straightforward: you need to give your operator a name (used for debugging), choose a partitioning strategy and then implements whatever data processing actions you'd like to perform. For partitioning, you can choose between `Pipeline` (if it doesn't matter what data arrives at which worker) or `Exchange` to specify your desired hash function. The processing logic is written as a closure which receives handles to the input and output stream and, optionally also

notifications on completions of epochs. For a more thorough walkthrough of how this all fits together, consult the following sections of the reference documentation:

- Creating operators: https://timelydataflow.github.io/timely-dataflow/chapter_2/chapter_2_4.html
- Custom data types: https://timelydataflow.github.io/timely-dataflow/chapter_4/chapter_4_5.html

1. Baby steps: a custom operator with state

Let's start from the example of a supermarket invoice: imagine you're on a shoestring budget and would like to calculate a *running average* as you scan each item. In other words: given a stream of numbers as input, the desired output is to enrich the price of each article with the average of all prices seen thus far (i.e. cumulative sum divided by number of items). We could make use the `aggregate` operator but it isn't well-suited to this task since it only reports the total on completion of each timestamp (not alongside each tuple).

Task #1: A friend points out that you could use a map closure to access data elements as they flow past and that counters can be accumulated by capturing a state variable. They suggest the program below which, upon compiling, you notice is disallowed due to type errors.

```
extern crate timely;
use timely::dataflow::operators::{Inspect, Map, ToStream};

fn main() {
    timely::example(|scope| {
        let mut sum = 0.0;
        let mut count = 0.0;
        (0..10)
            .to_stream(scope)
            .map(move |x| { sum += x as f64; x })
            .map(move |x| { count += 1.0; x })
            .map(move |x| (x, (sum / count)))
            .inspect(|x| println!("seen: {:?}", x));
    });
}
```

Reflect about the dataflow model and discuss the problems that arise if state is arbitrarily shared across the pipeline. Suggest how operators can safely communicate with one another whilst respecting the shared-nothing principle.

Task #2: Use the skeleton below to implement a synopsis operator that augments a stream of numbers with its average of all observed numbers.

- To start with, implement your operator directly inline by filling in the gaps for partitioning function (A) and operator logic (B).

```
extern crate timely;

use timely::dataflow::channels::pact::Pipeline;
use timely::dataflow::operators::{Inspect, ToStream};
use timely::dataflow::operators::generic::operator::Operator;

fn main() {
    timely::example(|scope| {
```

```
        let mut sum = 0;

        let input = (0..10).to_stream(scope);
        input.unary(/* TODO(A) */, "Average", move |_, _| move |input, output| {
                input.for_each(|time, data| {
                    /* TODO(B): operator logic goes here */
                });
            })
            .inspect(|x| println!("seen: {:?}", x));
    });
}
```

- We'd now like to formulate this as a reusable operator by factoring out the operator logic and its state into a standalone trait. Start from the snippet below and explain what changes you need to make within your code.

```
trait Average<G: Scope> {
    fn average(&self) -> Stream<G, (u64, f64)>;
}

impl<G: Scope> Average<G> for Stream<G, u64> {
    fn average(&self) -> Stream<G, (u64, f64)> {
        // ... (to be filled in) …
    }
}
```

- [*Optional sub-task*: remove all hard-coded types and make use of generics so your operator is applicable to any data type which implements the `Num` trait. The `num-traits` crate contains utilities to create identity values (0 and 1) and perform basic numeric operations.]

Task #3: As you will notice, you had to use a sequential implementation because the per-tuple average strictly depended on all previous values. Assume we relax the requirements to only emit the average once per epoch. Describe how you could now adapt your design so it best takes advantage of worker parallelism.

- You don't need to provide code but should include a sketch of the operator logic, details about any state required and your choice of partitioning strategy.
- (Hint: a common pattern breaks the work into two stages – one stage to locally count in parallel followed by a sequential stage to accumulate globally.)

2. Using progress information to emit consistent state

Logically, there are two separate planes within a streaming runtime: the data plane is used to efficiently ship around tuples and the control plane carries metadata about worker progress. In the previous task, your operator for calculating the average only needed access to the former (data) but not the latter (timestamps). In this task, you will need to make use of completion notifications to transform potentially out-of-order inputs and correctly output them in-order according to their timestamps. Recall that, whilst data may be circulating for several different timestamps simultaneously, the frontier at an operator (i.e. lower bound on progress) always advances monotonically along the time axis.

Task #4: Write an operator '`reorder`' which receives inputs, stashes them away as operator-local state

and correctly emits all relevant records on the output once Timely notifies you about completion of a timestamp.

- As an example, consider streams with the following time and data pairs:
  - Input: `[(2, "Berry"), (1, "Apple"), (3, "Carrot"), (1, "Apricot")]`
  - Output: `[(1, "Apple"), (1, "Apricot"), (2, "Berry"), (3, "Carrot")]`
- You will need to switch from the `unary` pattern to `unary_notify` for delivery of completion notifications. In addition, the normal input handle only supports a single open timestamp and so, in order to simulate out-of-order data, you will need to substitute it with an `UnorderedInput`.
- You can assume that late arrivals are bounded by some constant or that the input operator will deliver an appropriate watermark. Until now you manually invoked `advance_to` to close one timestamp and open the next. As there will now be multiple open timestamps, you will use the `Capability` type which represents a session i.e. the ability to send data for a specific timestamp.
- Capabilities adopt the RAII pattern meaning that once they are dropped, the system knows that no further inputs will be sent for that timestamp and can accordingly deliver notifications to downstream operators. To retain the ability to send, a new capability at a later timestamp can be obtained first via the function `Capability::delayed`. The code snippet below illustrates how to use this interface:

```
let (mut input, mut cap) = worker.dataflow::<usize, _, _>(|scope| {
    let (input, stream) = scope.new_unordered_input();
    stream.inspect_batch(move |epoch, data| {
              for d in data {
                  println!("@t={} | seen: {:?}", epoch, d);
              }
          })
    input  // returns a pair (input::UnorderedHandle, Capability)
});

// Generate out-of-order inputs
input.session(cap.delayed(&2)).give('B');
input.session(cap.delayed(&1)).give('A');
input.session(cap.delayed(&2)).give('b');
input.session(cap.delayed(&3)).give('C');
input.session(cap.delayed(&3)).give('c');
input.session(cap.delayed(&1)).give('a');
```

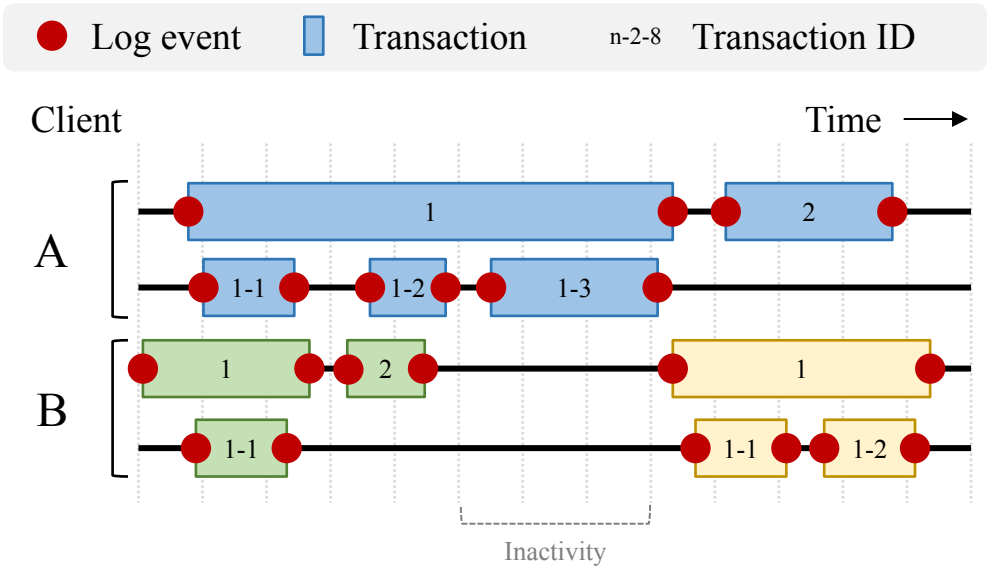
For this task you just need to submit a code snippet containing the `reorder` operator and an example dataflow that demonstrates outputs being correctly returned in timestamp order.

[*Remark*: the purpose of this task was just to familiarize you with the concept of notifications. In general, it is not recommended to delay data within the dataflow, as with the `reorder` operator, because it means idle workers cannot make use of slack time. By instead formulating operators to cope with out-of-order data, the system as a whole is not blocked and can peek ahead and process further into the stream which helps absorb dynamic changes in the workload.]

3. Implementing a session window using inactivity timeouts

Sliding and tumbling windows have fixed endpoints along the time axis. A *session* window captures a period of activity triggered by some initial action which is then closed once there is a gap of inactivity and so it has a dynamic size which is determined by the data itself. A typical example might be from web analytics such as "tell me all error codes and pages load by user 'Bob' within an hour of each other". Since users don't necessarily terminate their workflow (e.g. by logging out), the limits of a user session need to be determined by the absence of new data. A session window is therefore parameterized by a timeout and, as long as events keep occurring, the window duration will be continually extended.

This is shown pictorially in the following diagram which shows a stream of data elements grouped by originating user and how these are grouped into three distinct session windows (reflected by the colors).



Task #5: Write an operator which takes a stream of input data and returns elements grouped by originating session. The entire batch should be released once given timeout (in number of epochs) has elapsed and no more data has arrived for the given session. You can use the example above as inspiration or design any data layout corresponding to this interface:

```
trait SessionId {
    fn session_id(&self) -> &str;
}

trait SessionWindow<G: Scope, D: Data+SessionId> {
    fn sessionize(&self, epoch_timeout: usize) -> Stream<G, Vec<D>>;
}
```

Hint: as data arrives, you will need to continually extend the session closure by setting an 'alarm':

```
  notificator.notify_at(cap.delayed(&G::Timestamp::from_integer(t + timeout)));
```