Assignment 5

Task 1

In the dataflow model, in contrast to batch processing, data is processed in a **pipeline** of operators, this can cause a problem of *stream synchronisation*. If states are arbitrarily shared across the pipeline, a possible problem is that as soon as an upstream operator updates a shared state, a downstream operator will observe the new value immediately, even if the upstream operator is not meant to let the downstream operator consume the new value.

In our example, when the third `map` operator attempts to calculate the running average for all items until `x`, it is very likely that both `sum` and `count` have been updated by the other `map` operators which have received later items.

To enable inter-operator communication while respecting the shared-nothing principle, a possible approach is to explicitly put the data to be communicated into the output of an operator and serve as the input of another operator, so that every operator sees the values it is meant to see.

Task 2

The operator is implemented as:

```
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 = 0.0;
        let mut count = 0.0;

        let input = (0..10).to_stream(scope);
        input.unary(Pipeline, "Average", move |_, _| move |input, output|
{
                    input.for_each(|time, data| {
                        let mut vector = Vec::new();  // give back a new
vector to Timely
                        data.swap(&mut vector);
                        let mut session = output.session(&time);
                        for datum in vector.drain(..) {
                            sum += datum as f64;
                            count += 1.0;
                            session.give((datum, sum / count))
                        }
                    });
                }
            })
            .inspect(|x| println!("seen: {:?}", x));
```

```
    });
}
```

To formulate it as a reusable operator:

```
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)> {
       let mut sum = 0.0;
       let mut count = 0.0;
       self.unary(Pipeline, "Average", move |_, _| move |input, output| {
           input.for_each(|time, data| {
               let mut vector = Vec::new();
               data.swap(&mut vector);
               for datum in vector.drain(..) {
                   sum += datum as f64;
                   count += 1.0;
                   output.session(&time).give((datum, sum / count));
               }
           });
       })
    }
}
```

We need to add `use timely::dataflow::{Stream, Scope};` in the head, and the main function becomes:

```
fn main() {
    timely::example(|scope| {
        let input = (0..10).to_stream(scope);
        input.average()
             .inspect(|x| println!("seen: {:?}", x));
    });
}
```

Further, to generics, the following need to be added:

```
extern crate num_traits;

use timely::Data;
use num_traits::Num;
use num_traits::cast::ToPrimitive;
use timely::dataflow::{Stream, Scope};

trait NumData: Num + ToPrimitive + Copy + Data {}
```

```
impl<T: Num + ToPrimitive + Copy + Data> NumData for T {}

trait Average<G: Scope, D: NumData> {
    fn average(&self) -> Stream<G, (D, f64)>;
}
impl<G: Scope, D: NumData> Average<G, D> for Stream<G, D> {
    fn average(&self) -> Stream<G, (D, f64)> {
       let mut sum = 0.0;
       let mut count = 0.0;
       self.unary(Pipeline, "Average", move |_, _| move |input, output| {
           input.for_each(|time, data| {
               let mut vector = Vec::new();
               data.swap(&mut vector);
               for datum in vector.drain(..) {
                   sum += datum.to_u64().unwrap() as f64;
                   count += 1.0;
                   output.session(&time).give((datum, sum / count));
               }
           });
       })
    }
}
```

Task 3

First, in each epoch, the tuples should be uniformly distributed among all workers so that the advantage of worker parallism is taken at the best, and each worker handles a part of the tuples. The partitioning strategy should be as equal among workers as possible, for example, in a round-robin manner. Secondly, the operator should maintain two worker-local aggregate states `sum` and `count` for the tuples it received within one epoch. After one epoch is finished, there should be another stage to accumulate `sum` and `count` from all workers to one "master" worker, and this worker finally outputs the global average of this epoch.

Task 4

A HashMap is used to stash data. The data is not emitted until the notification of the completion of its epoch.

```
extern crate timely;

use std::collections::HashMap;
use timely::dataflow::operators::{Operator, UnorderedInput, Inspect};
use timely::dataflow::channels::pact::Pipeline;
use timely::Data;
use timely::dataflow::{Stream, Scope};

trait Reorder<G: Scope, D: Data> {
    fn reorder(&self) -> Stream<G, D>;
}
impl<G: Scope, D: Data> Reorder<G, D> for Stream<G, D> {
    fn reorder(&self) -> Stream<G, D> {
        let mut stash = HashMap::new();
```

```
        self.unary_notify(Pipeline, "Reorder", vec![], move |input,
output, barrier| {
            while let Some((time, data)) = input.next() {
                stash.entry(time.time().clone())
                        .or_insert(Vec::new())
                        .push(data.replace(Vec::new()));
                barrier.notify_at(time.retain());
            }
            // when notified
            while let Some((time, count)) = barrier.next() {
                println!("time {:?} complete with count {:?}!", time,
count);
                let mut session = output.session(&time);
                if let Some(list) = stash.remove(time.time()) {
                    for mut vector in list.into_iter() {
                        session.give_vec(&mut vector);
                    }
                }
            }
        })
    }
}

fn main() {
    timely::execute_from_args(std::env::args(), |worker| {
        let (mut input, cap) = worker.dataflow::<usize, _, _>(|scope| {
            let (input, stream) = scope.new_unordered_input();
            stream
                .reorder()
                .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');
    }).unwrap();
}
```

The output is:

```
time Capability { time: 1, internal: ... } complete with count 1!
time Capability { time: 2, internal: ... } complete with count 1!
time Capability { time: 3, internal: ... } complete with count 1!
```

```
@t=1 | seen: 'A'
@t=1 | seen: 'a'
@t=2 | seen: 'B'
@t=2 | seen: 'b'
@t=3 | seen: 'C'
@t=3 | seen: 'c'
```

Task 5

The operator assumes that data arrives in order (or at least, no data arrive for a window after the window has expired). Three states are maintained:

- `stash: HashMap<char, Vec<(usize, Action)>>`: maps from session id to actions with timestamps.
- `closing_time_to_session_ids: HashMap<_, Vec<char>>`: if a key `time` exists, it maps to all session_ids that are supposed to time out at that time. This HashMap is used to look up which sessions to output on notification.
- `session_id_to_closing_time: HashMap<char, usize>`: maps from session_id to the time it is supposed to time out. It helps us update `closing_time_to_session_ids` easily when the closing time of a session needs to be updated.

Every time a new datum comes in, it is pushed into `stash`, the other two hash maps are updated, and a delayed notification is registered. But the expiry of the corresponding session can be further postponed by other actions, so when a previously registered notification is fired, we first check if there still exists some sessions in `closing_time_to_session_ids` that are supposed to close at this time. If it really does, we drain all data of this session whose timestamp is within the timeout bound.

```
extern crate timely;

use std::collections::HashMap;
use timely::dataflow::{ProbeHandle, Stream, Scope};
use timely::dataflow::operators::{Inspect, Probe, UnorderedInput,
FrontierNotificator};
use timely::dataflow::operators::generic::operator::Operator;
use timely::dataflow::channels::pact::Pipeline;

#[derive(Clone, Debug)]
enum Action {
    Start,
    PageLoad,
    Click,
    KeyPress(char),
}

trait SessionWindow<G: Scope> {
    fn sessionize(&self, epoch_timeout: usize) -> Stream<G, (char,
Vec<Action>)>;
}

impl<G: Scope<Timestamp=usize>> SessionWindow<G> for Stream<G, (char,
```

```
Action)> {
    fn sessionize(&self, epoch_timeout: usize) -> Stream<G, (char,
Vec<Action>)> {

        self.unary_frontier(Pipeline, "SessionWindow", |_cap, _info| {
            let mut vector = Vec::new();
            let mut notificator = FrontierNotificator::new();
            let mut stash: HashMap<char, Vec<(usize, Action)>> =
HashMap::new();  // session_id -> list of (timestamp, action)
            let mut closing_time_to_session_ids: HashMap<_, Vec<char>> =
HashMap::new();  // session closure time -> list of closing session id(s)
            let mut session_id_to_closing_time: HashMap<char, usize> =
HashMap::new();  // session id -> session closure time

            move |input, output| {
                while let Some((time, data)) = input.next() {
                    data.swap(&mut vector);
                    for (session_id, user_interaction) in vector.drain(..)
{
                        stash.entry(session_id.clone())
                             .or_insert(Vec::new())
                             .push((*time.time(), user_interaction));
                        // remove old closing time if it's within
`epoch_timeout`
                        if let Some(closing_time) =
session_id_to_closing_time.get(&session_id) {
                            if *time.time() < *closing_time {
                                // this session_id is no longer supposed
to close at this closing_time
                                if let Some(session_ids) =
closing_time_to_session_ids.get_mut(closing_time) {
                                    session_ids.retain(|&x| x !=
session_id); // remove session_id from session_ids
                                }
                            }
                        }
                        // update closing time
                        let notification_time = *time.time() +
epoch_timeout;
                        let entry =
session_id_to_closing_time.entry(session_id).or_insert(notification_time);
                        *entry = notification_time;

closing_time_to_session_ids.entry(notification_time)
                                                   .or_insert(Vec::new())
                                                   .push(session_id);

notificator.notify_at(time.delayed(&notification_time));
                        // println!("session {:?} received {:?}, to be
notified at {:?} (closing time {:?})", session_id, user_interaction,
notification_time, session_id_to_closing_time.get(&session_id));
                    }
                };
```

```
                notificator.for_each(&[input.frontier()], |time,
_notificator| {
                    // println!("time {:?} complete! sessions timeout
{:?}", time, closing_time_to_session_ids.get(time.time()));
                    // get all session ids to be closed at this timestamp
                    if let Some(session_ids) =
closing_time_to_session_ids.remove(time.time()) {
                        for session_id in session_ids.into_iter() {
                            // get actions of this session
                            if let Some(actions) =
stash.get_mut(&session_id) {
                                let mut i = 0;
                                let mut list = Vec::new();
                                while i != actions.len() {
                                    if *time.time() >= actions[i].0 +
epoch_timeout {
                                        let val = actions.remove(i);
                                        // println!("push {:?} from time
{:?}", val.1, val.0);
                                        list.push(val.1);
                                    } else {
                                        i += 1;
                                    }
                                }
                                if actions.len() == 0 {
                                    // println!("session {:?} emptied",
session_id);
                                    stash.remove(&session_id);
                                }
                                // println!("Giving {:?}: {:?}",
session_id, list);
                                output.session(&time).give((session_id,
list));
                            }
                        }
                    }
                });
            }
        })

    }
}

fn main() {
    timely::execute_from_args(std::env::args(), |worker| {
        // Schema: (epoch, (session_id: char, user_interaction: Action))
        let input_records = vec![
            // Case 1: burst of activity with no inactivity gap
            (0, ('A', Action::Start)),
            (10, ('A', Action::PageLoad)),
            (50, ('A', Action::Click)),
            (100, ('A', Action::KeyPress('h'))),
            (150, ('A', Action::KeyPress('o'))),
            (200, ('A', Action::Click)),
```

```
            (250, ('A', Action::KeyPress('w'))),
            (250, ('A', Action::KeyPress('d'))),
            (300, ('A', Action::KeyPress('y'))),
            (350, ('A', Action::Click)),

            // Case 2: user took a break so their session is fragmented
into several pieces
            (0, ('B', Action::Start)),
            (20, ('B', Action::PageLoad)),
            (25, ('B', Action::KeyPress('f'))),
            (26, ('B', Action::KeyPress('u'))),
            (27, ('B', Action::KeyPress('n'))),

            (1000, ('B', Action::Click)),
            (1050, ('B', Action::Click)),
            (1100, ('B', Action::KeyPress(':'))),
            (1110, ('B', Action::KeyPress('-'))),
            (1120, ('B', Action::KeyPress(')'))),

            (2000, ('B', Action::Click)),
            (2001, ('B', Action::Click)),
            (2002, ('B', Action::Click)),
            (2002, ('B', Action::Click)),
            (2003, ('B', Action::Click)),
        ];

        // Construct the dataflow graph
        let mut probe = ProbeHandle::new();
        let (mut input, cap0) = worker.dataflow::<usize, _, _>(|scope| {
            let (input, stream) = scope.new_unordered_input();
            stream
                .sessionize(500)
                .inspect_batch(move |epoch, data| {
                    for d in data {
                        println!("@t={} | seen: {:?}", epoch, d);
                    }
                })
                .probe_with(&mut probe);
            input
        });

        // Feed the inputs and let the computation run
        let max_time = *input_records.iter().map(|(t, _)|
t).max().unwrap() + 1;
        for (time, data) in input_records {
            input.session(cap0.delayed(&time)).give(data);
        }
        drop(input);
        drop(cap0);  // (Here we promise not to produce any outputs for
time 0 or later)

        while probe.less_than(&max_time) {
            worker.step();
        }
```

```
    }).unwrap();
}
```

The output is:

```
@t=527 | seen: ('B', [Start, PageLoad, KeyPress('f'), KeyPress('u'),
KeyPress('n')])
@t=850 | seen: ('A', [Start, PageLoad, Click, KeyPress('h'),
KeyPress('o'), Click, KeyPress('w'), KeyPress('d'), KeyPress('y'), Click])
@t=1620 | seen: ('B', [Click, Click, KeyPress(':'), KeyPress('-'),
KeyPress(')')])
@t=2503 | seen: ('B', [Click, Click, Click, Click, Click])
```