

---

## Differential Dataflow

An implementation of differential dataflow over timely dataflow on Rust.

### Background

Differential dataflow is a data-parallel programming framework designed to efficiently process large volumes of data and to quickly respond to arbitrary changes in input collections. You can read more in the differential dataflow mdbuf and in the differential dataflow documentation.

Differential dataflow programs are written as functional transformations of collections of data, using familiar operators like `map`, `filter`, `join`, and `reduce`. Differential dataflow also includes more exotic operators such as `iterate`, which repeatedly applies a differential dataflow fragment to a collection. The programs are compiled down to timely dataflow computations.

For example, here is a differential dataflow fragment to compute the out-degree distribution of a directed graph (for each degree, the number of nodes with that many outgoing edges):

```
1 let out_degr_dist =
2 edges.map(|(src, _dst)| src)      // extract source
3     .count()                     // count occurrences of source
4     .map(|(_src, deg)| deg)      // extract degree
5     .count();                    // count occurrences of degree
```

Alternately, here is a fragment that computes the set of nodes reachable from a set `roots` of starting nodes:

```
1 let reachable =
2 roots.iterate(|reach|
3     edges.enter(&reach.scope())
4     .semijoin(reach)
5     .map(|(src, dst)| dst)
6     .concat(reach)
7     .distinct()
8 )
```

Once written, a differential dataflow responds to arbitrary changes to its initially empty input collections, reporting the corresponding changes to each of its output collections. Differential dataflow can react quickly because it only acts where changes in collections occur, and does no work elsewhere.

In the examples above, we can add to and remove from `edges`, dynamically altering the graph, and get immediate feedback on how the results change: if the degree distribution shifts we'll see the changes, and if nodes are now (or no longer) reachable we'll hear about that too. We could also add to and remove from `roots`, more fundamentally altering the reachability query itself.

---

Be sure to check out the differential dataflow documentation, which is continually improving.

### An example: counting degrees in a graph.

Let's check out that out-degree distribution computation, to get a sense for how differential dataflow actually works. This example is `examples/hello.rs` in this repository, if you'd like to follow along.

A graph is a collection of pairs (`Node`, `Node`), and one standard analysis is to determine the number of times each `Node` occurs in the first position, its "degree". The number of nodes with each degree is a helpful graph statistic.

To determine the out-degree distribution, we create a new timely dataflow scope in which we describe our computation and how we plan to interact with it.

```
1 // create a degree counting differential dataflow
2 let (mut input, probe) = worker.dataflow(|scope| {
3
4     // create edge input, count a few ways.
5     let (input, edges) = scope.new_collection();
6
7     let out_degr_distr =
8     edges.map(|(src, _dst)| src)    // extract source
9         .count()                  // count occurrences of source
10        .map(|(_src, deg)| deg)    // extract degree
11        .count();                 // count occurrences of degree
12
13    // show us something about the collection, notice when done.
14    let probe =
15    out_degr_distr
16        .inspect(|x| println!("observed: {:?}", x))
17        .probe();
18
19    (input, probe)
20 });
```

The `input` and `probe` we return are how we get data into the dataflow, and how we notice when some amount of computation is complete. These are timely dataflow idioms, and we won't get in to them in more detail here (check out the timely dataflow repository).

If we feed this computation with some random graph data, say fifty random edges among ten nodes, we get output like

```
1 Echidnatron% cargo run --release --example hello -- 10 50 1 inspect
2 Finished release [optimized + debuginfo] target(s) in 0.05s
3 Running `target/release/examples/hello 10 50 1 inspect`
4 observed: ((3, 1), 0, 1)
5 observed: ((4, 2), 0, 1)
```

---

```
6 observed: ((5, 4), 0, 1)
7 observed: ((6, 2), 0, 1)
8 observed: ((7, 1), 0, 1)
9 round 0 finished after 772.464µs (loading)
```

This shows us the records that passed the `inspect` operator, revealing the contents of the collection: there are five distinct degrees, three through seven. The records have the form `((degree, count), time, delta)` where the `time` field says this is the first round of data, and the `delta` field tells us that each record is coming into existence. If the corresponding record were departing the collection, it would be a negative number.

Let's update the input by removing one edge and adding a new random edge:

```
1 observed: ((2, 1), 1, 1)
2 observed: ((3, 1), 1, -1)
3 observed: ((7, 1), 1, -1)
4 observed: ((8, 1), 1, 1)
5 round 1 finished after 149.701µs
```

We see here some changes! Those degree three and seven nodes have been replaced by degree two and eight nodes; looks like one node lost an edge and gave it to the other!

How about a few more changes?

```
1 round 2 finished after 127.444µs
2 round 3 finished after 100.628µs
3 round 4 finished after 130.609µs
4 observed: ((5, 3), 5, 1)
5 observed: ((5, 4), 5, -1)
6 observed: ((6, 2), 5, -1)
7 observed: ((6, 3), 5, 1)
8 observed: ((7, 1), 5, 1)
9 observed: ((8, 1), 5, -1)
10 round 5 finished after 161.82µs
```

Well a few weird things happen here. First, rounds 2, 3, and 4 don't print anything. Seriously? It turns out that the random changes we made didn't affect any of the degree counts, we moved edges between nodes, preserving degrees. It can happen.

The second weird thing is that in round 5, with only two edge changes we have six changes in the output! It turns out we can have up to eight. The degree eight gets turned back into a seven, and a five gets turned into a six. But: going from five to six *changes* the count for each, and each change requires two record differences. Eight and seven were more concise because their counts were only one, meaning just arrival and departure of records rather than changes.

---

## Scaling up

The appealing thing about differential dataflow is that it only does work where changes occur, so even if there is a lot of data, if not much changes it can still go quite fast. Let's scale our 10 nodes and 50 edges up by a factor of one million:

```
1 Echidnatron% cargo run --release --example hello -- 100000000 500000000 1
  inspect
2   Finished release [optimized + debuginfo] target(s) in 0.04s
3   Running `target/release/examples/hello 100000000 500000000 1 inspect`
4 observed: ((1, 336908), 0, 1)
5 observed: ((2, 843854), 0, 1)
6 observed: ((3, 1404462), 0, 1)
7 observed: ((4, 1751921), 0, 1)
8 observed: ((5, 1757099), 0, 1)
9 observed: ((6, 1459805), 0, 1)
10 observed: ((7, 1042894), 0, 1)
11 observed: ((8, 653178), 0, 1)
12 observed: ((9, 363983), 0, 1)
13 observed: ((10, 181423), 0, 1)
14 observed: ((11, 82478), 0, 1)
15 observed: ((12, 34407), 0, 1)
16 observed: ((13, 13216), 0, 1)
17 observed: ((14, 4842), 0, 1)
18 observed: ((15, 1561), 0, 1)
19 observed: ((16, 483), 0, 1)
20 observed: ((17, 143), 0, 1)
21 observed: ((18, 38), 0, 1)
22 observed: ((19, 8), 0, 1)
23 observed: ((20, 3), 0, 1)
24 observed: ((22, 1), 0, 1)
25 round 0 finished after 15.470465014s (loading)
```

There are a lot more distinct degrees here. I sorted them because it was too painful to look at the unsorted data. You would normally get to see the output unsorted, because they are just changes to values in a collection.

Let's perform a single change again.

```
1 observed: ((5, 1757098), 1, 1)
2 observed: ((5, 1757099), 1, -1)
3 observed: ((6, 1459805), 1, -1)
4 observed: ((6, 1459807), 1, 1)
5 observed: ((7, 1042893), 1, 1)
6 observed: ((7, 1042894), 1, -1)
7 round 1 finished after 228.451µs
```

Although the initial computation took about fifteen seconds, we get our changes in about 230 microseconds; that's about one hundred thousand times faster than re-running the computation. That's

---

pretty nice. Actually, it is small enough that the time to print things to the screen is a bit expensive, so let's stop doing that.

Now we can just watch as changes roll past and look at the times.

```
1 Echidnatron% cargo run --release --example hello -- 10000000 50000000 1
    no_inspect
2 Finished release [optimized + debuginfo] target(s) in 0.04s
3 Running `target/release/examples/hello 10000000 50000000 1
    no_inspect`
4 round 0 finished after 15.586969662s (loading)
5 round 1 finished after 1.070239ms
6 round 2 finished after 2.303187ms
7 round 3 finished after 208.45µs
8 round 4 finished after 163.224µs
9 round 5 finished after 118.792µs
10 ...
```

Nice. This is some hundreds of microseconds per update, which means maybe ten thousand updates per second. It's not a horrible number for my laptop, but it isn't the right answer yet.

### Scaling .. “along”?

Differential dataflow is designed for throughput in addition to latency. We can increase the number of rounds of updates it works on concurrently, which can increase its effective throughput. This does not change the output of the computation, except that we see larger batches of output changes at once.

Notice that those times above are a few hundred microseconds for each single update. If we work on ten rounds of updates at once, we get times that look like this:

```
1 Echidnatron% cargo run --release --example hello -- 10000000 50000000
    10 no_inspect
2 Finished release [optimized + debuginfo] target(s) in 0.04s
3 Running `target/release/examples/hello 10000000 50000000 10
    no_inspect`
4 round 0 finished after 15.556475008s (loading)
5 round 10 finished after 421.219µs
6 round 20 finished after 1.56369ms
7 round 30 finished after 338.54µs
8 round 40 finished after 351.843µs
9 round 50 finished after 339.608µs
10 ...
```

This is appealing in that rounds of ten aren't much more expensive than single updates, and we finish the first ten rounds in much less time than it takes to perform the first ten updates one at a time. Every round after that is just bonus time.

---

As we turn up the batching, performance improves. Here we work on one hundred rounds of updates at once:

```
1 Echidnatron% cargo run --release --example hello -- 10000000 50000000
    100 no_inspect
2     Finished release [optimized + debuginfo] target(s) in 0.04s
3     Running `target/release/examples/hello 10000000 50000000 100
        no_inspect`
4 round 0 finished after 15.528724145s (loading)
5 round 100 finished after 2.567577ms
6 round 200 finished after 1.861168ms
7 round 300 finished after 1.753794ms
8 round 400 finished after 1.528285ms
9 round 500 finished after 1.416605ms
10 ...
```

We are still improving, and continue to do so as we increase the batch sizes. When processing 100,000 updates at a time we take about half a second for each batch. This is less “interactive” but a higher throughput.

```
1 Echidnatron% cargo run --release --example hello -- 10000000 50000000
    100000 no_inspect
2     Finished release [optimized + debuginfo] target(s) in 0.04s
3     Running `target/release/examples/hello 10000000 50000000 100000
        no_inspect`
4 round 0 finished after 15.65053789s (loading)
5 round 100000 finished after 505.210924ms
6 round 200000 finished after 524.069497ms
7 round 300000 finished after 470.77752ms
8 round 400000 finished after 621.325393ms
9 round 500000 finished after 472.791742ms
10 ...
```

This averages to about five microseconds on average; a fair bit faster than the hundred microseconds for individual updates! And now that I think about it each update was actually two changes, wasn't it. Good for you, differential dataflow!

## Scaling out

Differential dataflow is built on top of timely dataflow, a distributed data-parallel runtime. Timely dataflow scales out to multiple independent workers, increasing the capacity of the system (at the cost of some coordination that cuts into latency).

If we bring two workers to bear, our 10 million node, 50 million edge computation drops down from fifteen seconds to just over eight seconds.

---

```
1 Echidnatron% cargo run --release --example hello -- 10000000 50000000 1
    no_inspect -w2
2 Finished release [optimized + debuginfo] target(s) in 0.04s
3 Running `target/release/examples/hello 10000000 50000000 1
    no_inspect -w2`
4 round 0 finished after 8.065386177s (loading)
5 round 1 finished after 275.373µs
6 round 2 finished after 759.632µs
7 round 3 finished after 171.671µs
8 round 4 finished after 745.078µs
9 round 5 finished after 213.146µs
10 ...
```

That is a so-so reduction. You might notice that the times *increased* for the subsequent rounds. It turns out that multiple workers just get in each other's way when there isn't much work to do.

Fortunately, as we work on more and more rounds of updates at the same time, the benefit of multiple workers increases. Here are the numbers for ten rounds at a time:

```
1 Echidnatron% cargo run --release --example hello -- 10000000 50000000
    10 no_inspect -w2
2 Finished release [optimized + debuginfo] target(s) in 0.04s
3 Running `target/release/examples/hello 10000000 50000000 10
    no_inspect -w2`
4 round 0 finished after 8.083000954s (loading)
5 round 10 finished after 1.901946ms
6 round 20 finished after 3.092976ms
7 round 30 finished after 889.63µs
8 round 40 finished after 409.001µs
9 round 50 finished after 320.248µs
10 ...
```

One hundred rounds at a time:

```
1 Echidnatron% cargo run --release --example hello -- 10000000 50000000
    100 no_inspect -w2
2 Finished release [optimized + debuginfo] target(s) in 0.04s
3 Running `target/release/examples/hello 10000000 50000000 100
    no_inspect -w2`
4 round 0 finished after 8.121800831s (loading)
5 round 100 finished after 2.52821ms
6 round 200 finished after 3.119036ms
7 round 300 finished after 1.63147ms
8 round 400 finished after 1.008668ms
9 round 500 finished after 941.426µs
10 ...
```

One hundred thousand rounds at a time:

```
1 Echidnatron% cargo run --release --example hello -- 10000000 50000000
```

---

```

1000000 no_inspect -w2
2 Finished release [optimized + debuginfo] target(s) in 0.04s
3 Running `target/release/examples/hello 100000000 500000000 1000000
  no_inspect -w2`
4 round 0 finished after 8.200755198s (loading)
5 round 100000 finished after 275.262419ms
6 round 200000 finished after 279.291957ms
7 round 300000 finished after 259.137138ms
8 round 400000 finished after 340.624124ms
9 round 500000 finished after 259.870938ms
10 ...

```

These last numbers were about half a second with one worker, and are decently improved with the second worker.

### Going even faster

There are several performance optimizations in differential dataflow designed to make the underlying operators as close to what you would expect to write, when possible. Additionally, by building on timely dataflow, you can drop in your own implementations a la carte where you know best.

For example, we also know in this case that the underlying collections go through a *sequence* of changes, meaning their timestamps are totally ordered. In this case we can use a much simpler implementation, `count_total`. This reduces the update times substantially, for each batch size:

```

1 Echidnatron% cargo run --release --example hello -- 100000000 500000000
  10 no_inspect -w2
2 Finished release [optimized + debuginfo] target(s) in 0.04s
3 Running `target/release/examples/hello 100000000 500000000 10
  no_inspect -w2`
4 round 0 finished after 5.985084002s (loading)
5 round 10 finished after 1.802729ms
6 round 20 finished after 2.202838ms
7 round 30 finished after 192.902µs
8 round 40 finished after 198.342µs
9 round 50 finished after 187.725µs
10 ...
11
12 Echidnatron% cargo run --release --example hello -- 100000000 500000000
  100 no_inspect -w2
13 Finished release [optimized + debuginfo] target(s) in 0.04s
14 Running `target/release/examples/hello 100000000 500000000 100
  no_inspect -w2`
15 round 0 finished after 5.588270073s (loading)
16 round 100 finished after 3.114716ms
17 round 200 finished after 2.657691ms
18 round 300 finished after 890.972µs
19 round 400 finished after 448.537µs

```



---

```

20 round 500 finished after 384.565µs
21 ...
22
23 Echidnatron% cargo run --release --example hello -- 10000000 50000000
      100000 no_inspect -w2
24     Finished release [optimized + debuginfo] target(s) in 0.04s
25     Running `target/release/examples/hello 10000000 50000000 100000
      no_inspect -w2`
26 round 0 finished after 6.486550581s (loading)
27 round 100000 finished after 89.096615ms
28 round 200000 finished after 79.469464ms
29 round 300000 finished after 72.568018ms
30 round 400000 finished after 93.456272ms
31 round 500000 finished after 73.954886ms
32 ...

```

These times have now dropped quite a bit from where we started; we now absorb over one million rounds of updates per second, and produce correct (not just consistent) answers even while distributed across multiple workers.

## A second example: k-core computation

The k-core of a graph is the largest subset of its edges so that all vertices with any incident edges have degree at least k. One way to find the k-core is to repeatedly delete all edges incident on vertices with degree less than k. Those edges going away might lower the degrees of other vertices, so we need to *iteratively* throwing away edges on vertices with degree less than k until we stop. Maybe we throw away all the edges, maybe we stop with some left over.

Here is a direct implementation, in which we repeatedly take determine the set of active nodes (those with at least k edges point to or from them), and restrict the set `edges` to those with both `src` and `dst` present in `active`.

```

1 let k = 5;
2
3 // iteratively thin edges.
4 edges.iterate(|inner| {
5
6     // determine the active vertices      /-- this is a lie --\
7     let active = inner.flat_map(|(src,dst)| [src,dst].into_iter())
8                           .map(|node| (node, ()))
9                           .group(|_node, s, t| if s[0].1 > k { t.push(((,
10                                1)); })
11                           .map(|(node,_)| node);
12
13     // keep edges between active vertices
14     edges.enter(&inner.scope())
15         .semijoin(active)

```

---

```
15         .map(|(src,dst)| (dst,src))
16         .semijoin(active)
17         .map(|(dst,src)| (src,dst))
18     });
```

To be totally clear, the syntax with `into_iter()` doesn't work, because Rust, and instead there is a more horrible syntax needed to get a non-heap allocated iterator over two elements. But, it works, and

```
1 Running `target/release/examples/degrees 100000000 500000000 1 5 kcore1`
2 Loading finished after 72204416910
```

Well that is a thing. Who knows if 72 seconds is any good? (*ed*: it is worse than the numbers in the previous version of this readme).

The amazing thing, though is what happens next:

```
1 worker 0, round 1 finished after Duration { secs: 0, nanos: 567171 }
2 worker 0, round 2 finished after Duration { secs: 0, nanos: 449687 }
3 worker 0, round 3 finished after Duration { secs: 0, nanos: 467143 }
4 worker 0, round 4 finished after Duration { secs: 0, nanos: 480019 }
5 worker 0, round 5 finished after Duration { secs: 0, nanos: 404831 }
```

We are taking about half a millisecond to *update* the k-core computation. Each edge addition and deletion could cause other edges to drop out of or more confusingly *return* to the k-core, and differential dataflow is correctly updating all of that for you. And it is doing it in sub-millisecond timescales.

If we crank the batching up by one thousand, we improve the throughput a fair bit:

```
1 Running `target/release/examples/degrees 100000000 500000000 1000 5
   kcore1`
2 Loading finished after Duration { secs: 73, nanos: 507094824 }
3 worker 0, round 1000 finished after Duration { secs: 0, nanos: 55649900 }
4 worker 0, round 2000 finished after Duration { secs: 0, nanos: 51793416 }
5 worker 0, round 3000 finished after Duration { secs: 0, nanos: 57733231 }
6 worker 0, round 4000 finished after Duration { secs: 0, nanos: 50438934 }
7 worker 0, round 5000 finished after Duration { secs: 0, nanos: 55020469 }
```

Each batch is doing one thousand rounds of updates in just over 50 milliseconds, averaging out to about 50 microseconds for each update, and corresponding to roughly 20,000 distinct updates per second.

I think this is all great, both that it works at all and that it even seems to work pretty well.

---

## **Roadmap**

The issue tracker has several open issues relating to current performance defects or missing features. If you are interested in contributing, that would be great! If you have other questions, don't hesitate to get in touch.

## **Acknowledgements**

In addition to contributions to this repository, differential dataflow is based on work at the now defunct Microsoft Research lab in Silicon Valley, and continued at the Systems Group of ETH Zürich. Numerous collaborators at each institution (among others) have contributed both ideas and implementations.