Modern Dataflow

Frank McSherry
@frankmcsherry
http://github.com/frankmcsherry

Acks: Derek Murray, Rebecca Isaacs, Michael Isard, Paul Barham, Martin Abadi, Gordon Plotkin, Andrea Lattuada, Zaheer Chothia, John Liagouris,

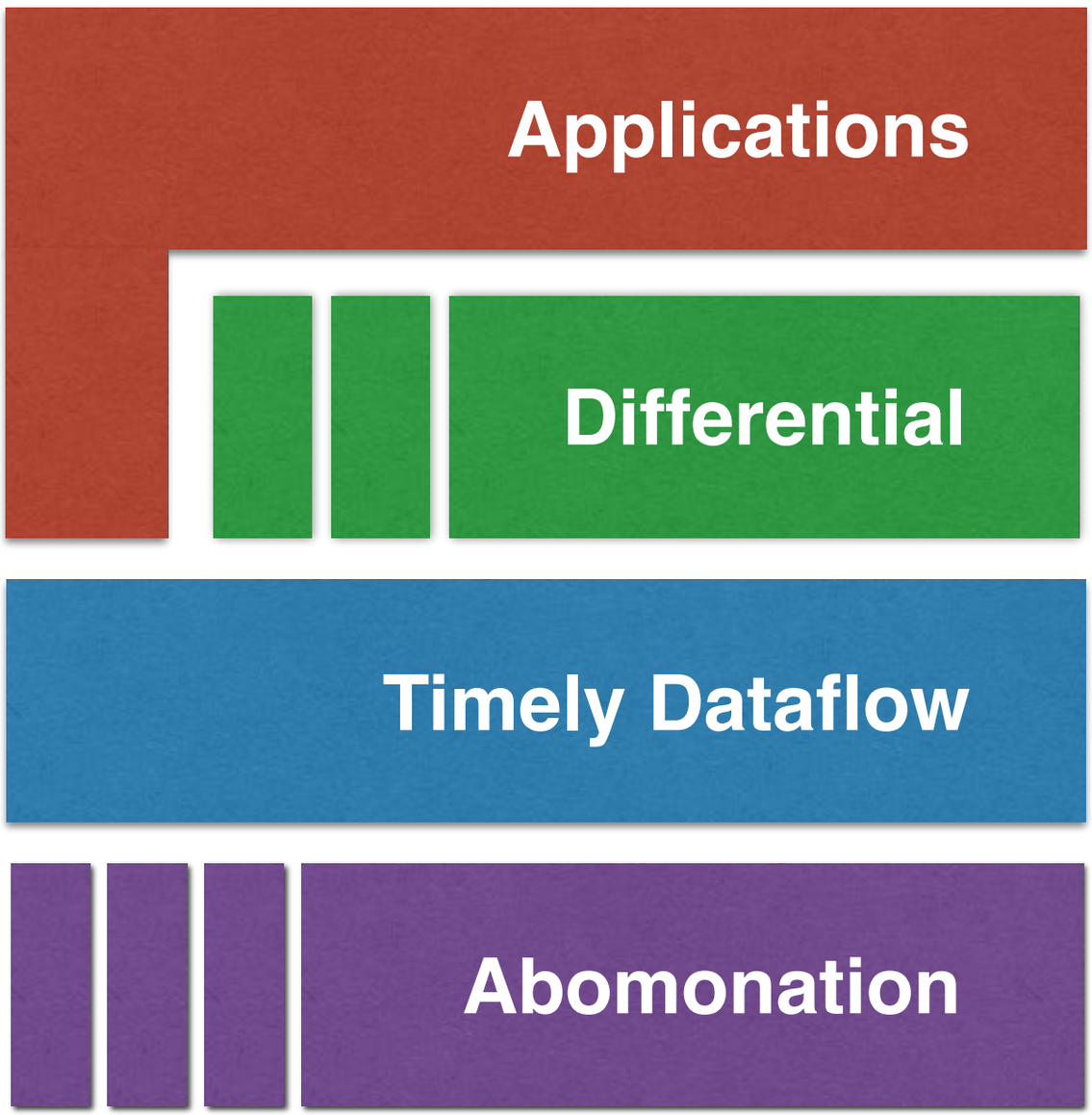
20xPR	cores	twitter_rv	uk_2007_05
Spark	128	857s	1759s
Giraph	128	596s	1235s
GraphLab	128	249s	833s
GraphX	128	419s	462s

[GraphX, OSDI 2014]

20xPR	cores	twitter_rv	uk_2007_05
Spark	128	857s	1759s
Giraph	128	596s	1235s
GraphLab	128	249s	833s
GraphX	128	419s	462s
Laptop	1	**110s**	**256s**

[COST, HotOS 2015]

20xPR	cores	twitter_rv	uk_2007_05
Spark	128	857s	1759s
Giraph	128	596s	1235s
GraphLab	128	249s	833s
GraphX	128	419s	462s
Laptop	1	**110s**	**256s**
Timely	128	**15s**	**19s**



Cool stuff like real-time graph analytics, etc.

Language for scalable incremental computation

A bit like an OS for data-parallel compute

Data serialization at memory bandwidth

Differential Dataflow

"Functional Reactive Programming at Scale"

http://github.com/frankmcsherry/differential-dataflow

[based off of Differential Dataflow, CIDR 2013]

People are good at programming with collections.

(at least, better than with streams)

Gist: collection-oriented programming language,
the system manages changes to collections.

```
fn your_prog: [D] -> [R] = /* .. */;

for t in times {
    let output[t] = your_prog(input[t]);
}
```

d_output: Stream<(Data, Time, isize)>

input streams of changes

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;
```

“program” : dataflow assembly

```
nodes.join(edges)     // one hop neighbors
     .concat(nodes)   // plus original nodes
     .distinct()      // extended neighborhood
```

dataflow execution

```
for t in times {
    nodes.insert(..);
    edges.insert(..);
}
```

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;
```

```
nodes.join(edges)    // one hop neighbors
     .concat(nodes)  // plus original nodes
     .distinct()     // extended neighborhood
```

```
for t in times {
    nodes.insert(..);
    edges.insert(..);
}
```

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;
```

```
nodes.join(edges)    // one hop neighbors
     .concat(nodes)  // plus original nodes
     .distinct()     // extended neighborhood
```

```
for t in times {
    nodes.insert(..); nodes.remove(..);
    edges.insert(..); edges.remove(..);
}
```

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;
```

```
        nodes.join(edges)    // one hop neighbors
             .concat(nodes)  // plus original nodes
             .distinct()     // extended neighborhood
```

```
for t in times {
    nodes.insert(..); nodes.remove(..);
    edges.insert(..); edges.remove(..);
}
```

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;

nodes.iterate(|reach| {

    nodes.join(edges)    // one hop neighbors
         .concat(nodes)  // plus original nodes
         .distinct()     // extended neighborhood

});

for t in times {
    nodes.insert(..); nodes.remove(..);
    edges.insert(..); edges.remove(..);
}
```

```
let nodes = /* pairs (node, bool) */;
let edges = /* pairs (node, node) */;

nodes.iterate(|reach| {

    reach.join(edges)     // one hop neighbors
         .concat(nodes)   // plus original nodes
         .distinct()      // extended neighborhood

});

for t in times {
    nodes.insert(..); nodes.remove(..);
    edges.insert(..); edges.remove(..);
}
```

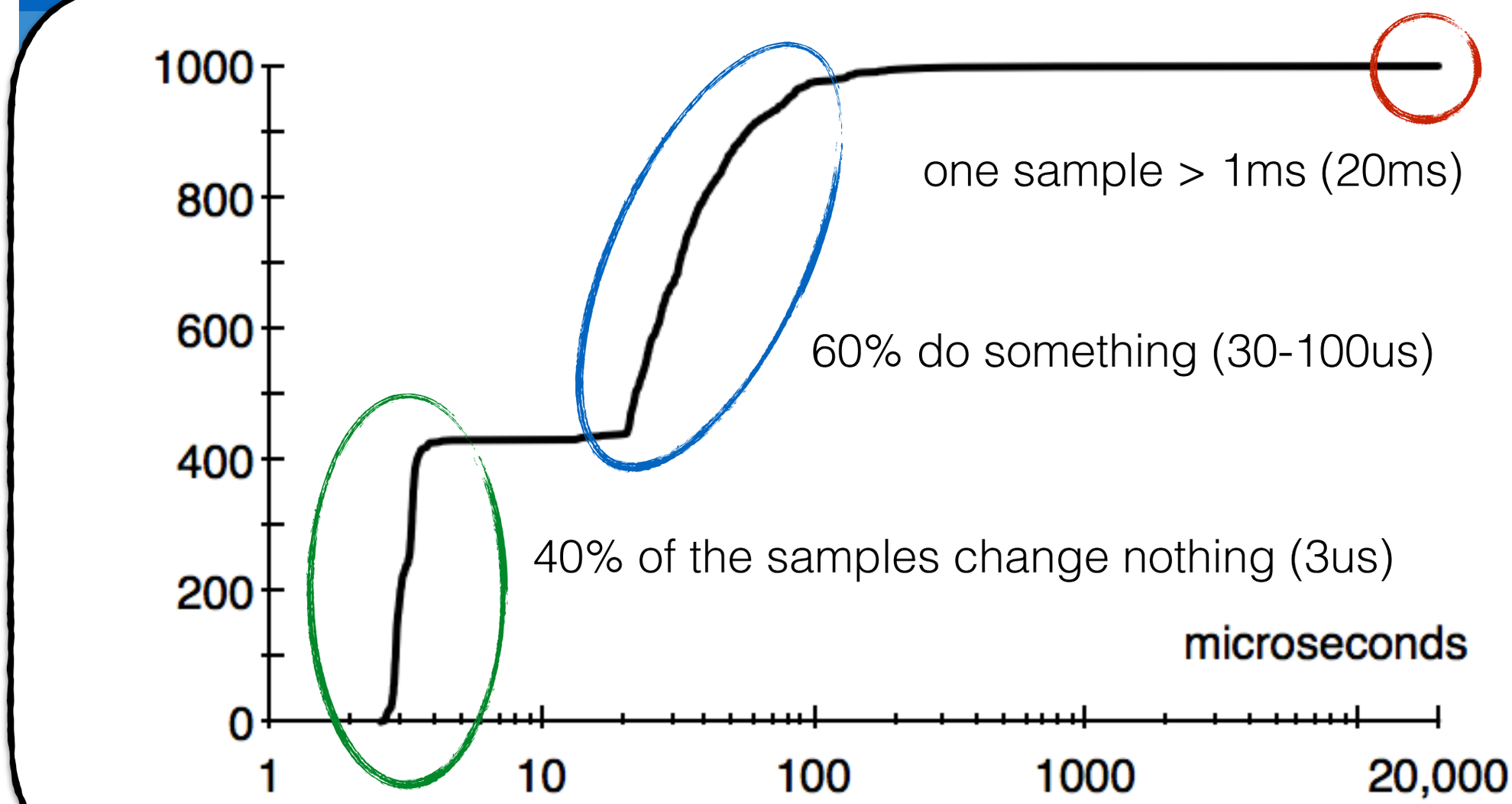
Stream<((node, bool), (Time, u64), isize)>

Reach	cores	livejournal	orkut
GraphX	128	36s	48s
SociaLite	128	52s	67s
Myria	128	5s	6s
BigDatalog	128	17s	20s

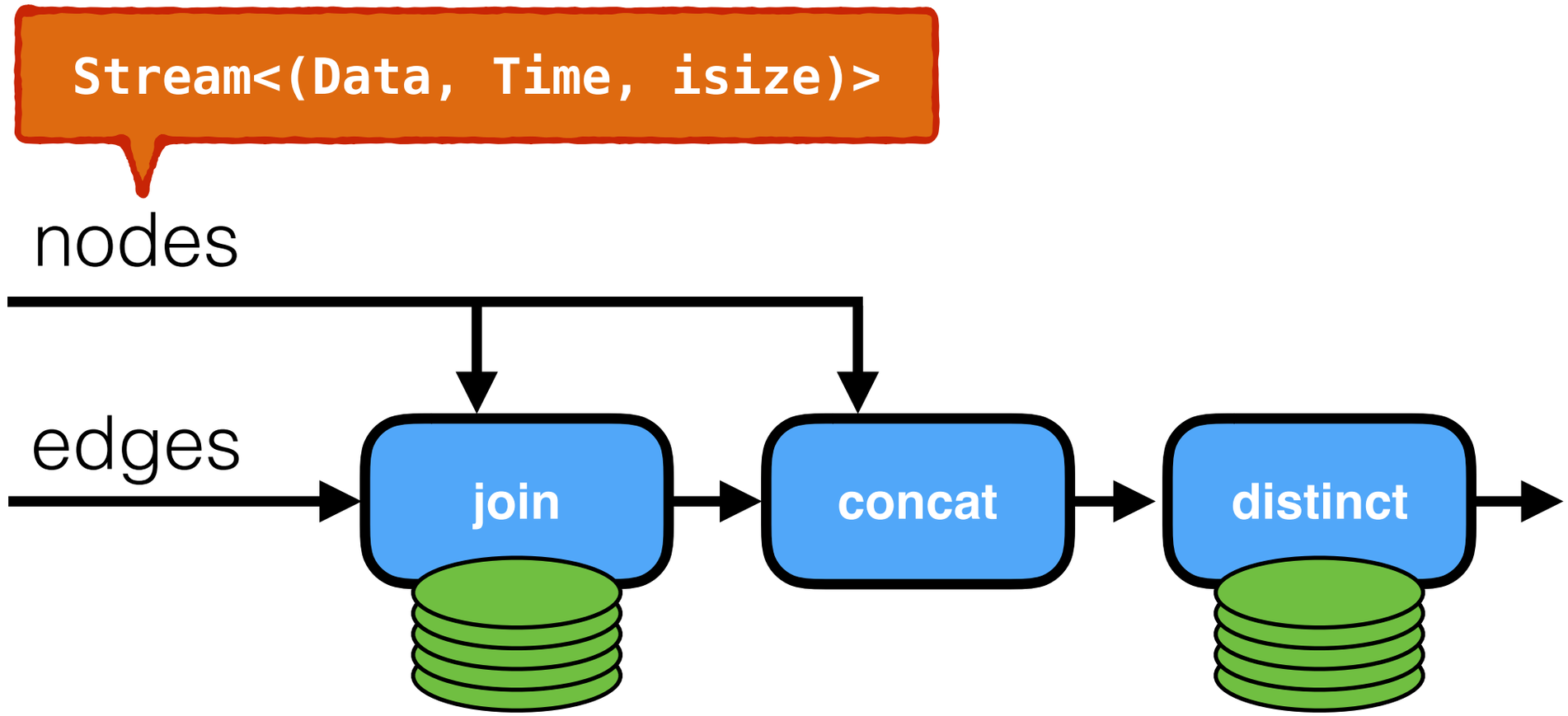
[BigDatalog, SIGMOD 2016]

Reach	cores	livejournal	orkut
GraphX	128	36s	48s
SociaLite	128	52s	67s
Myria	128	5s	6s
BigDatalog	128	17s	20s
Differential	1	**7s**	**15s**

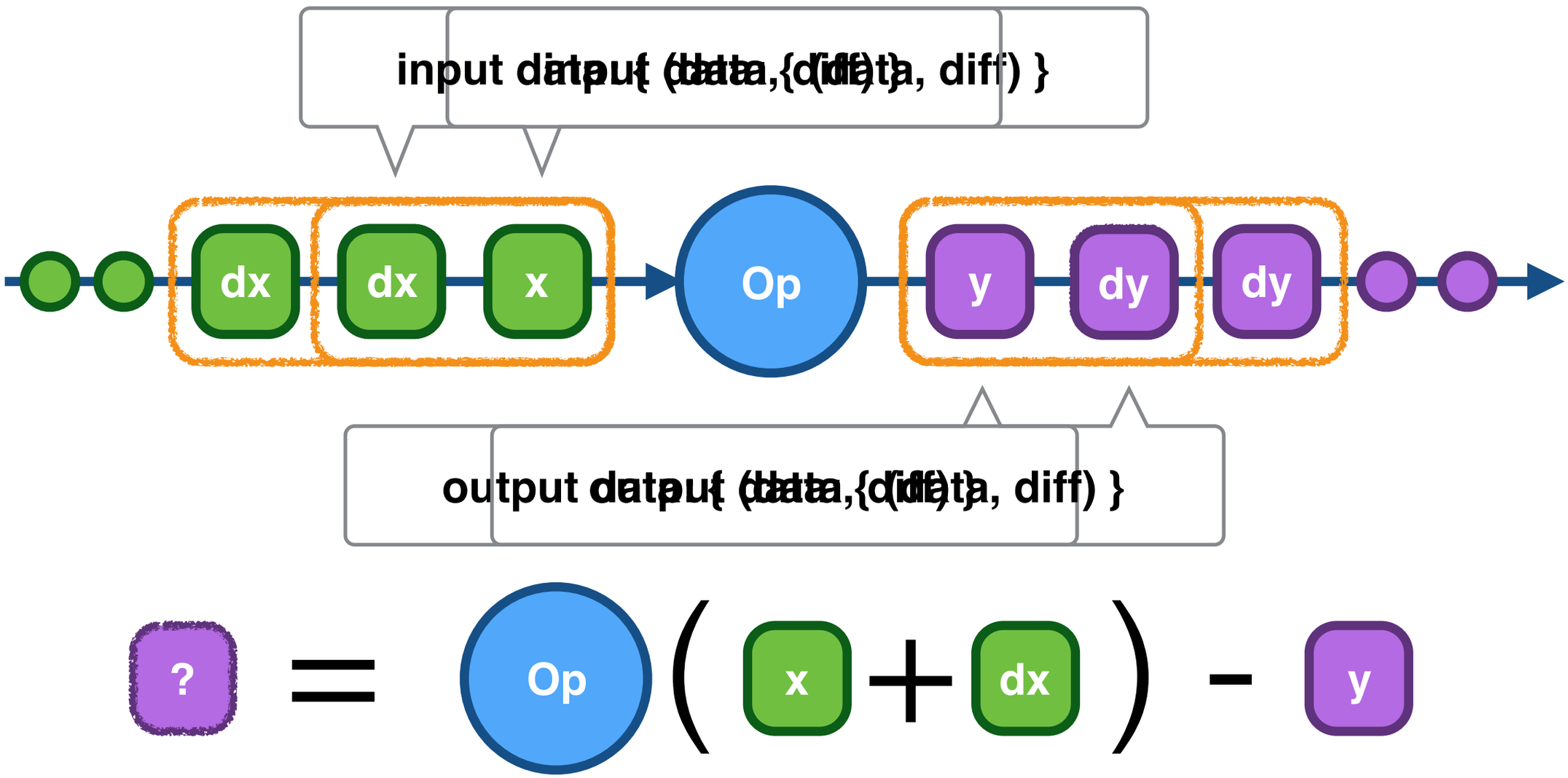
Reach	cores	livejournal	orkut
update	1	30us	62us



Reach	cores	livejournal	orkut
GraphX	128	36s	48s
SociaLite	128	52s	67s
Myria	128	5s	6s
BigDatalog	128	17s	20s
Differential	1	**7s**	**15s**
update	1	**50us**	**62us**

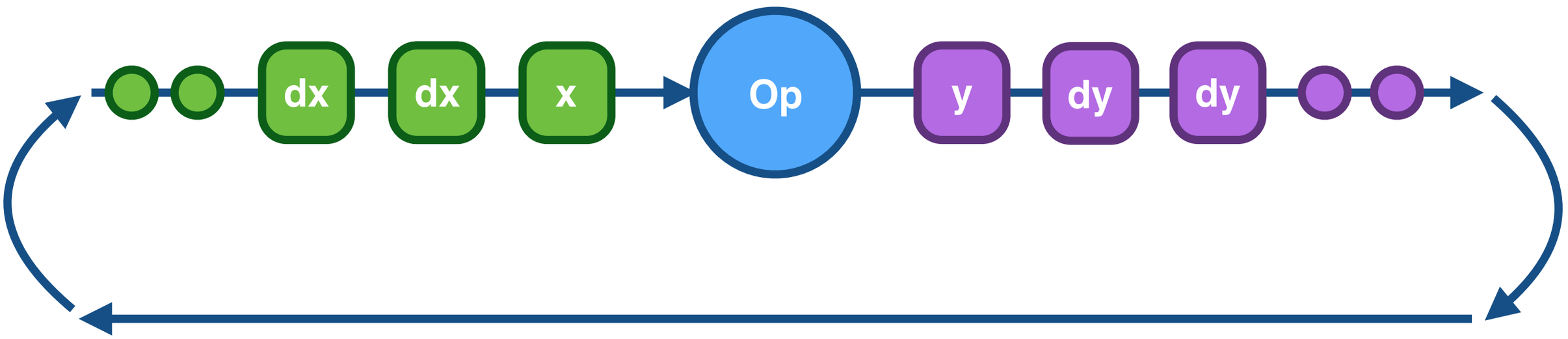

Stream<(Data, Time, isize)>
nodes
edges
join
concat
distinct

Incremental Dataflow



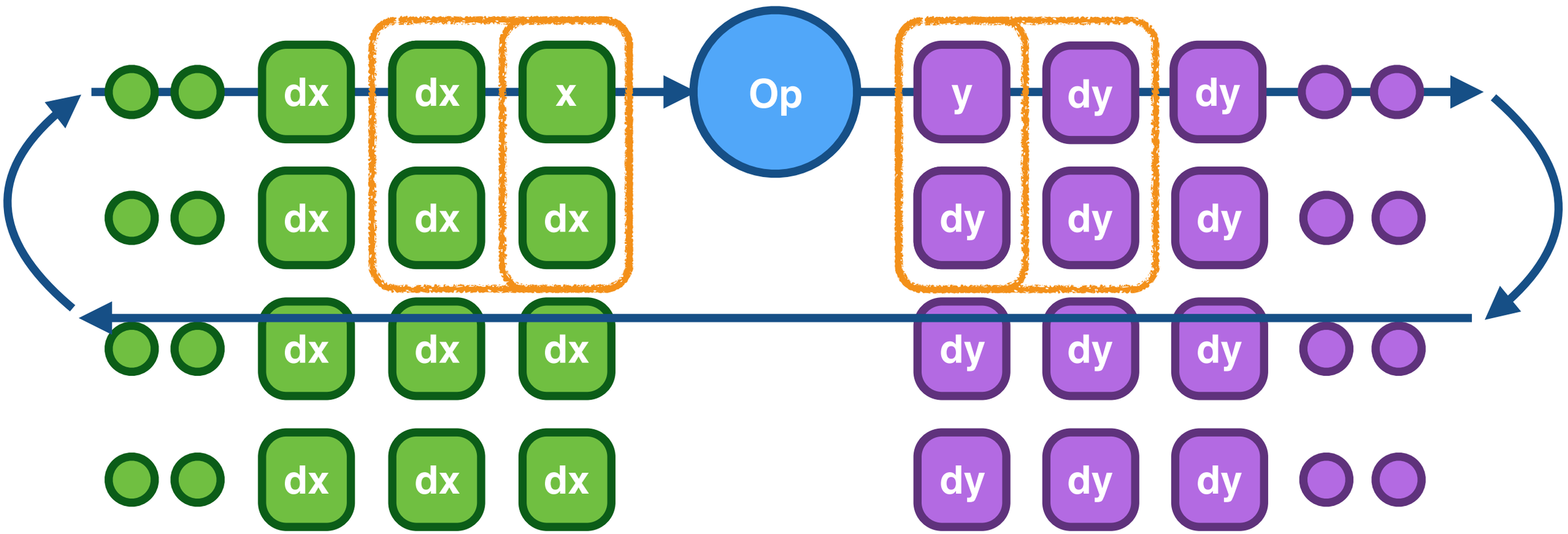
Iterative Dataflow

e.g. semi-naive bottom-up datalog

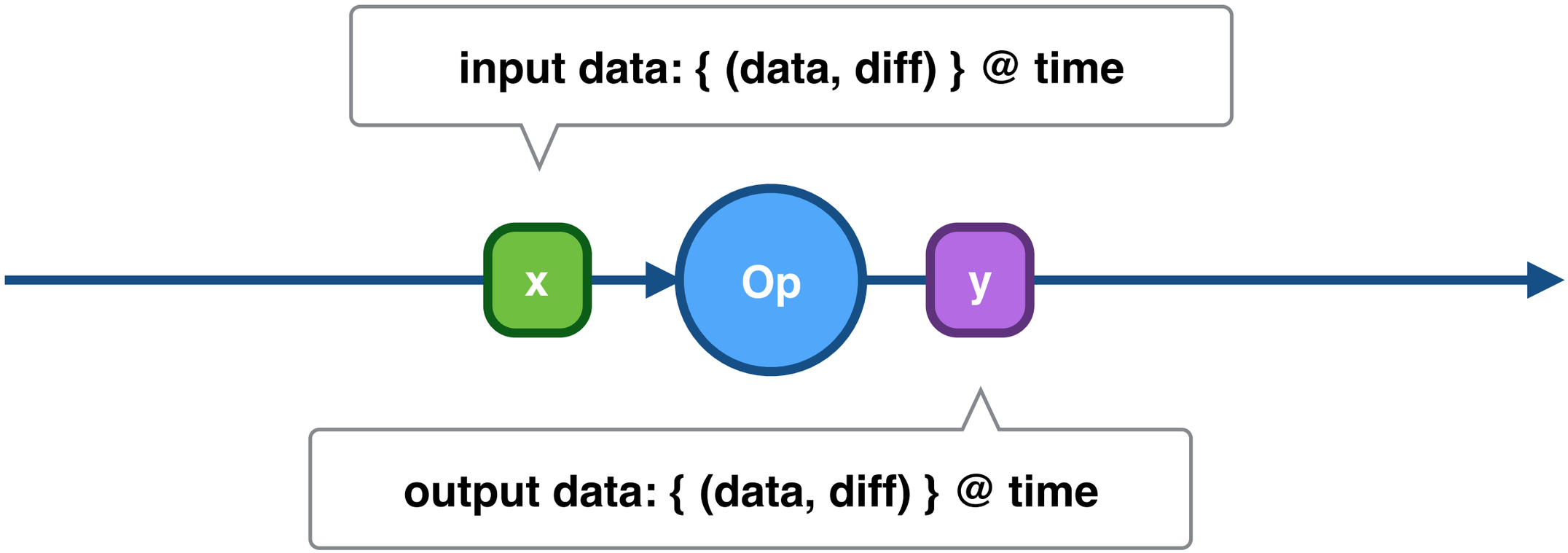


Differential Dataflow

(finally)



Differential Dataflow

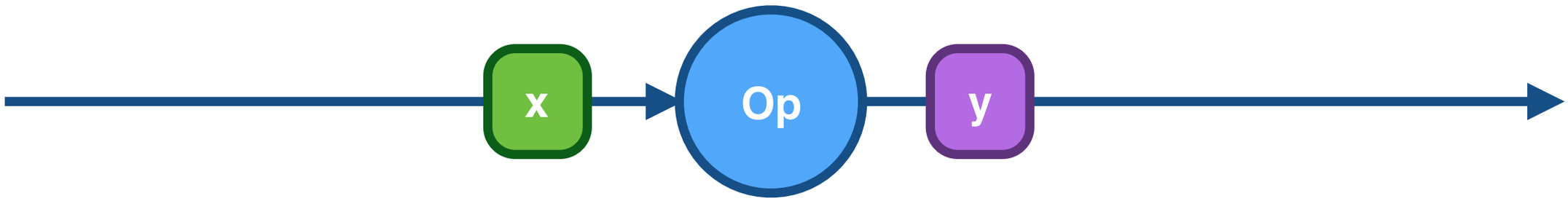


Important: times are only **partially** ordered

Differentiation on a discrete partial order

Differential Dataflow

with data-parallel operators

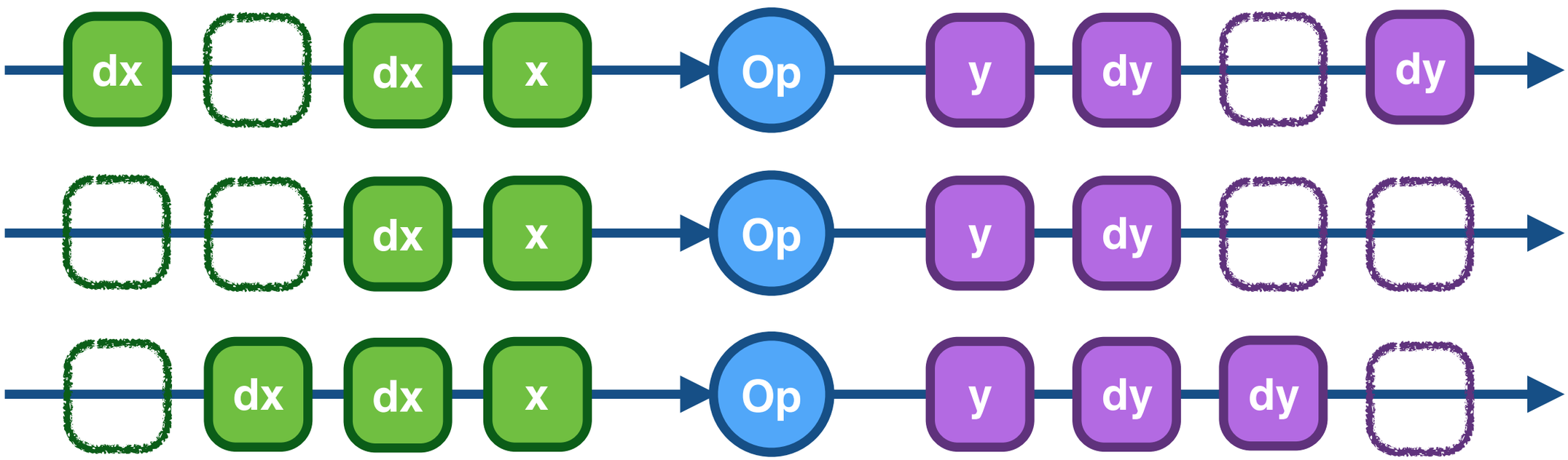


Differential Dataflow

with data-parallel operators

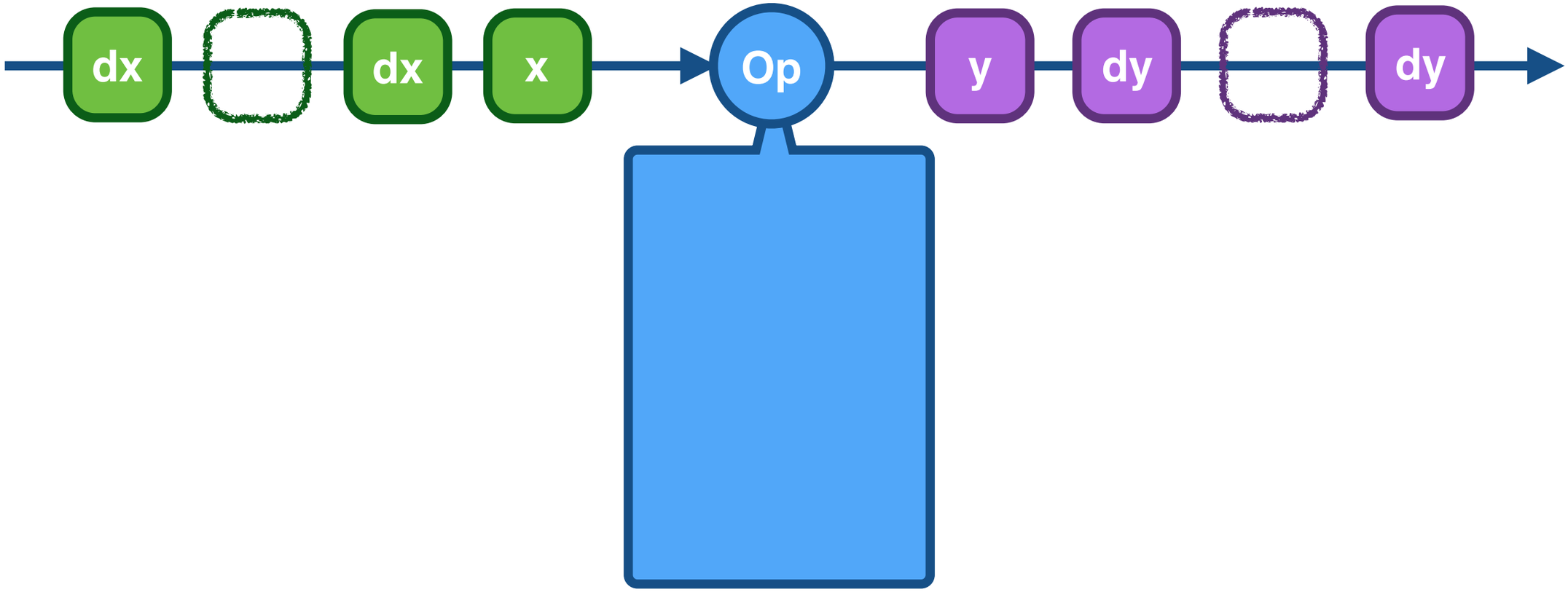
Differential Dataflow

with data-parallel operators



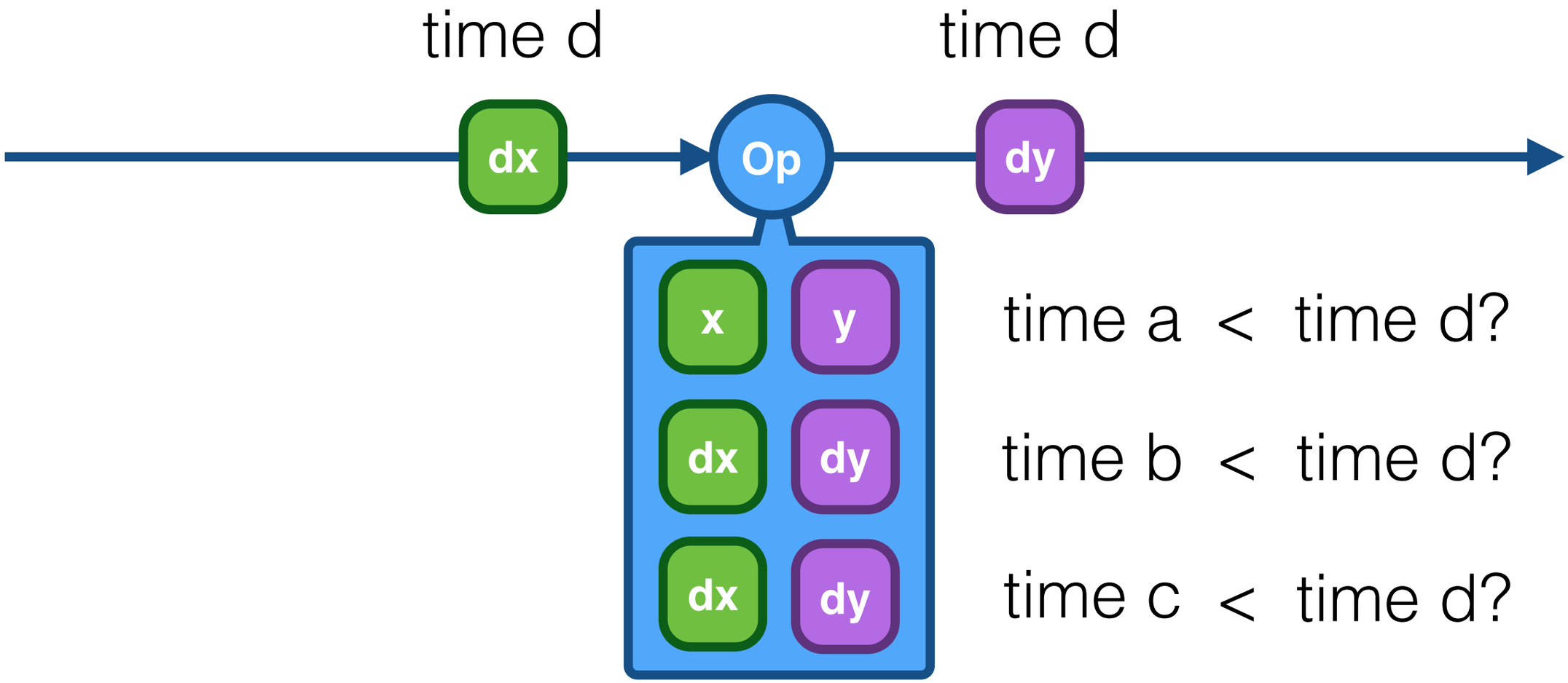
Differential Dataflow

with data-parallel operators



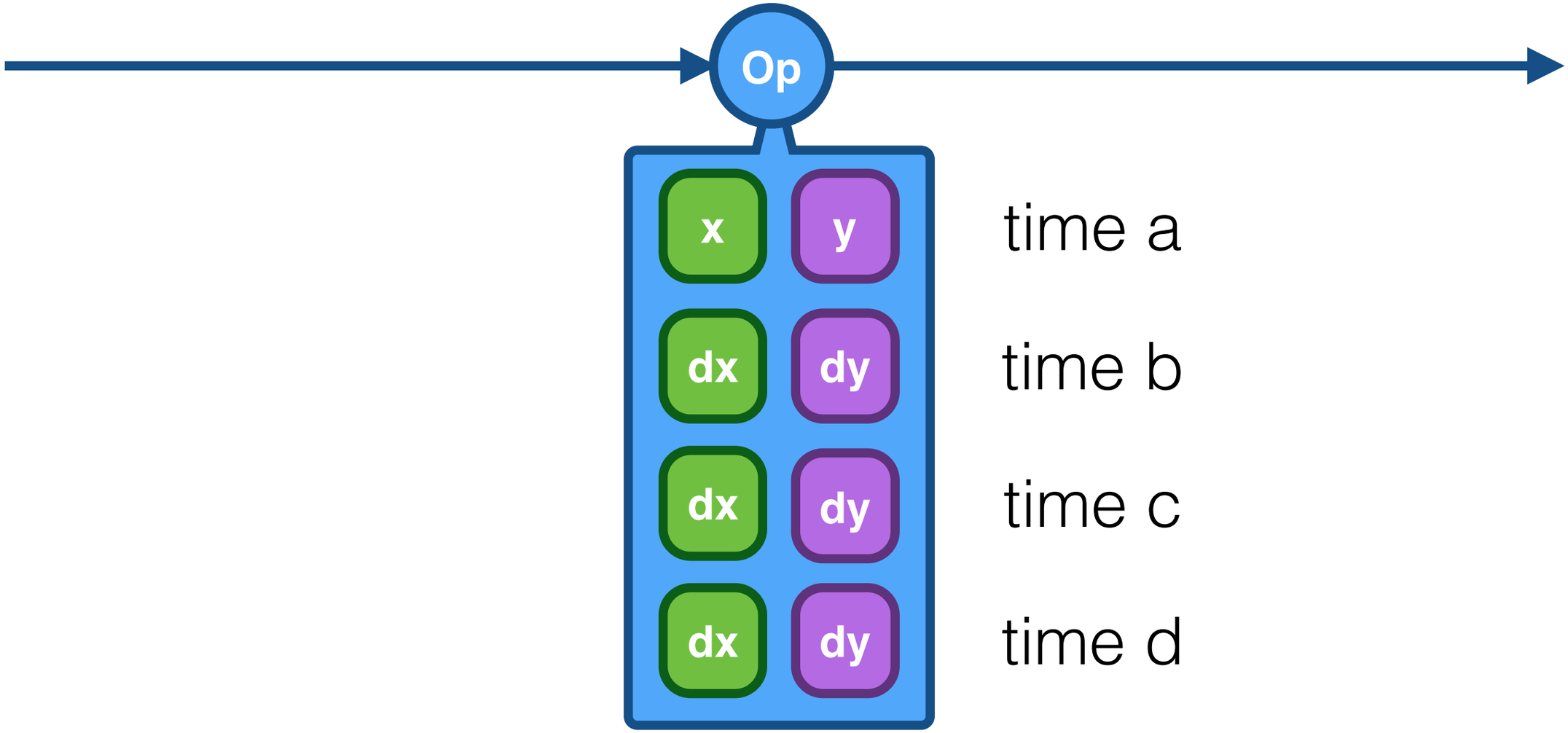
Differential Dataflow

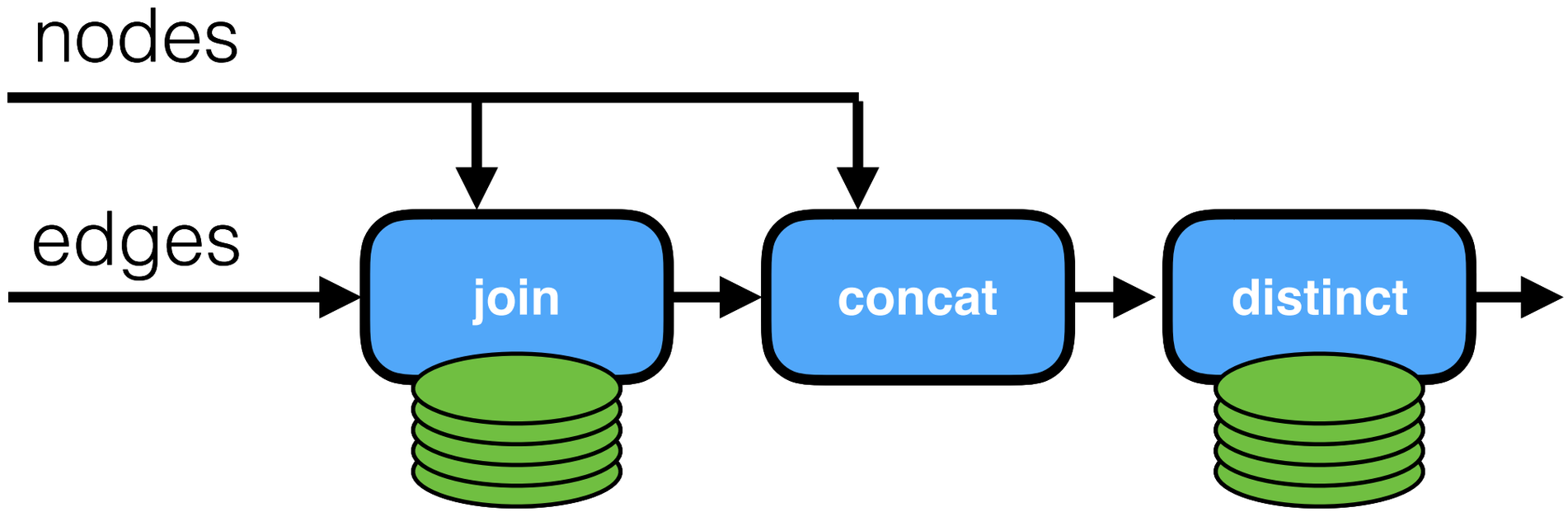
with data-parallel operators

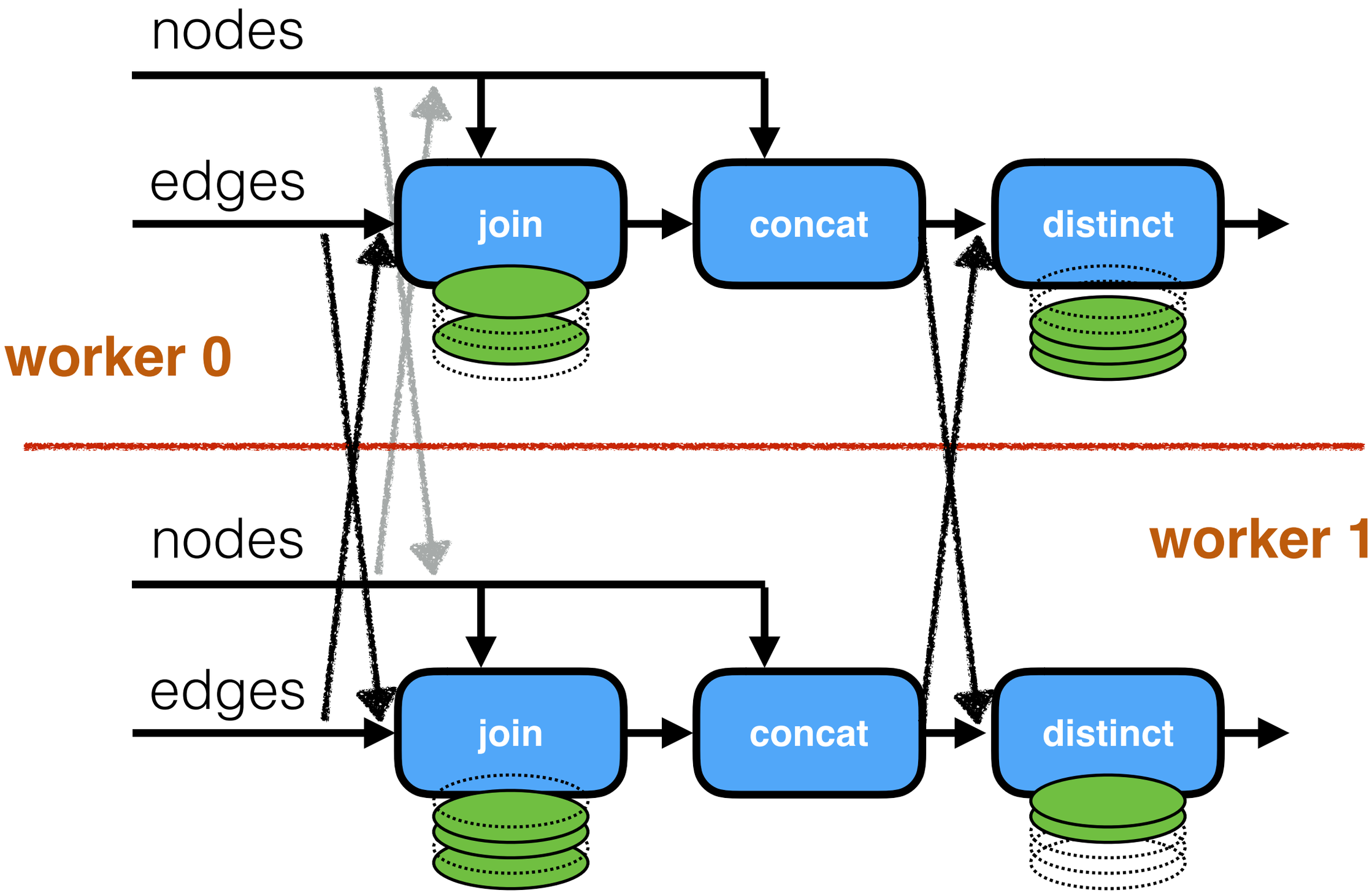


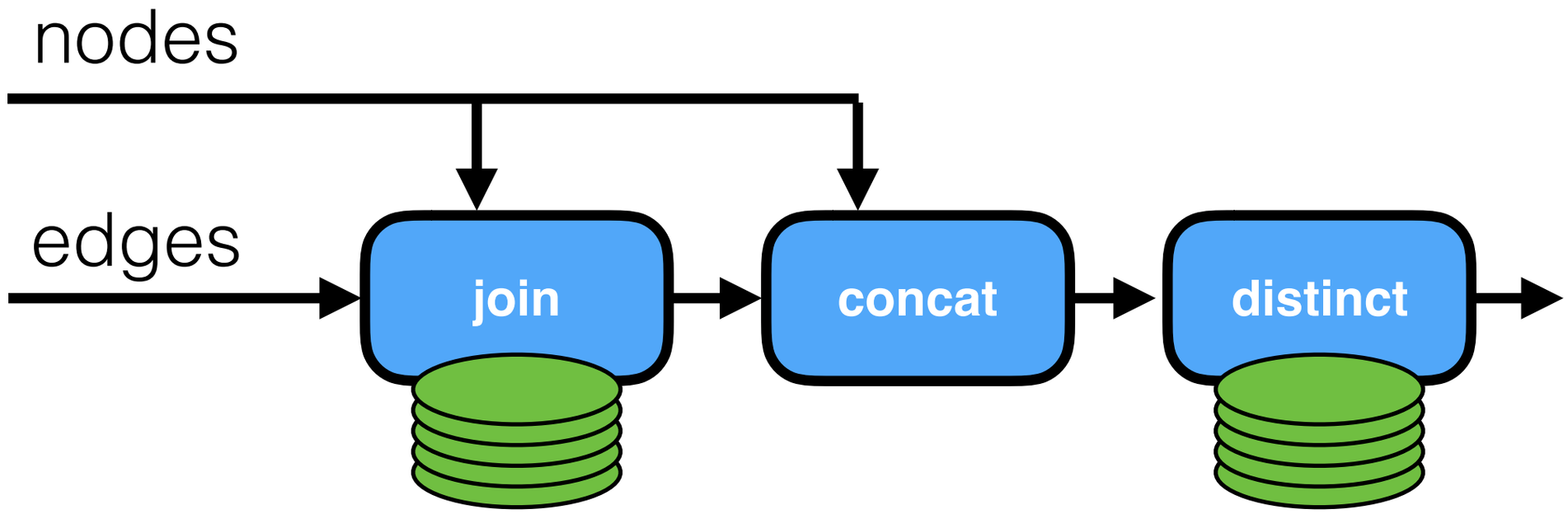
Differential Dataflow

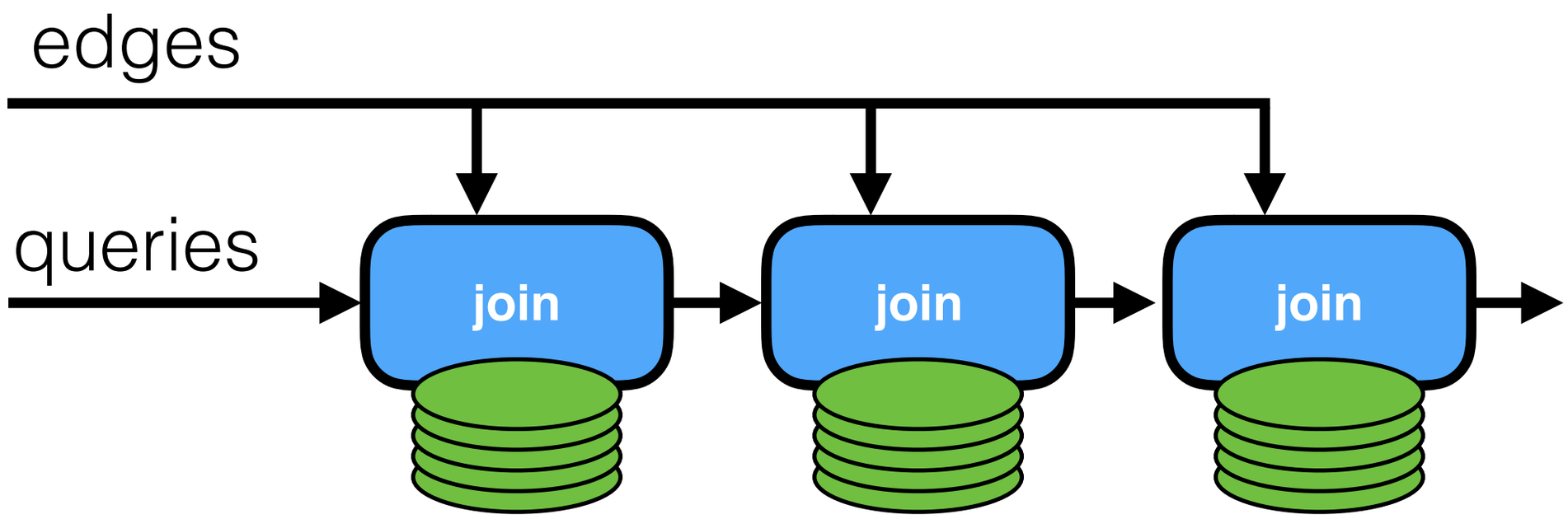
with data-parallel operators

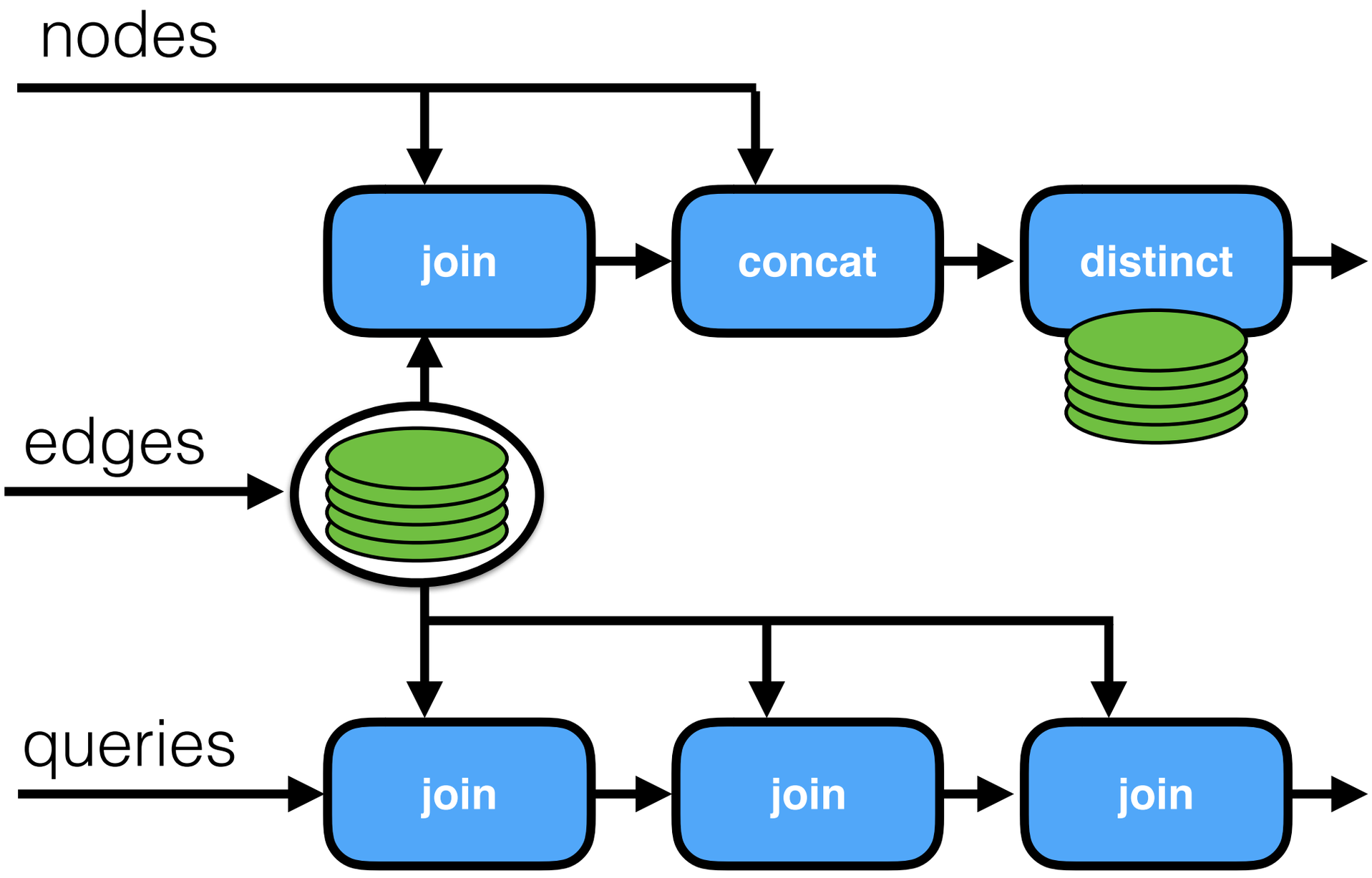


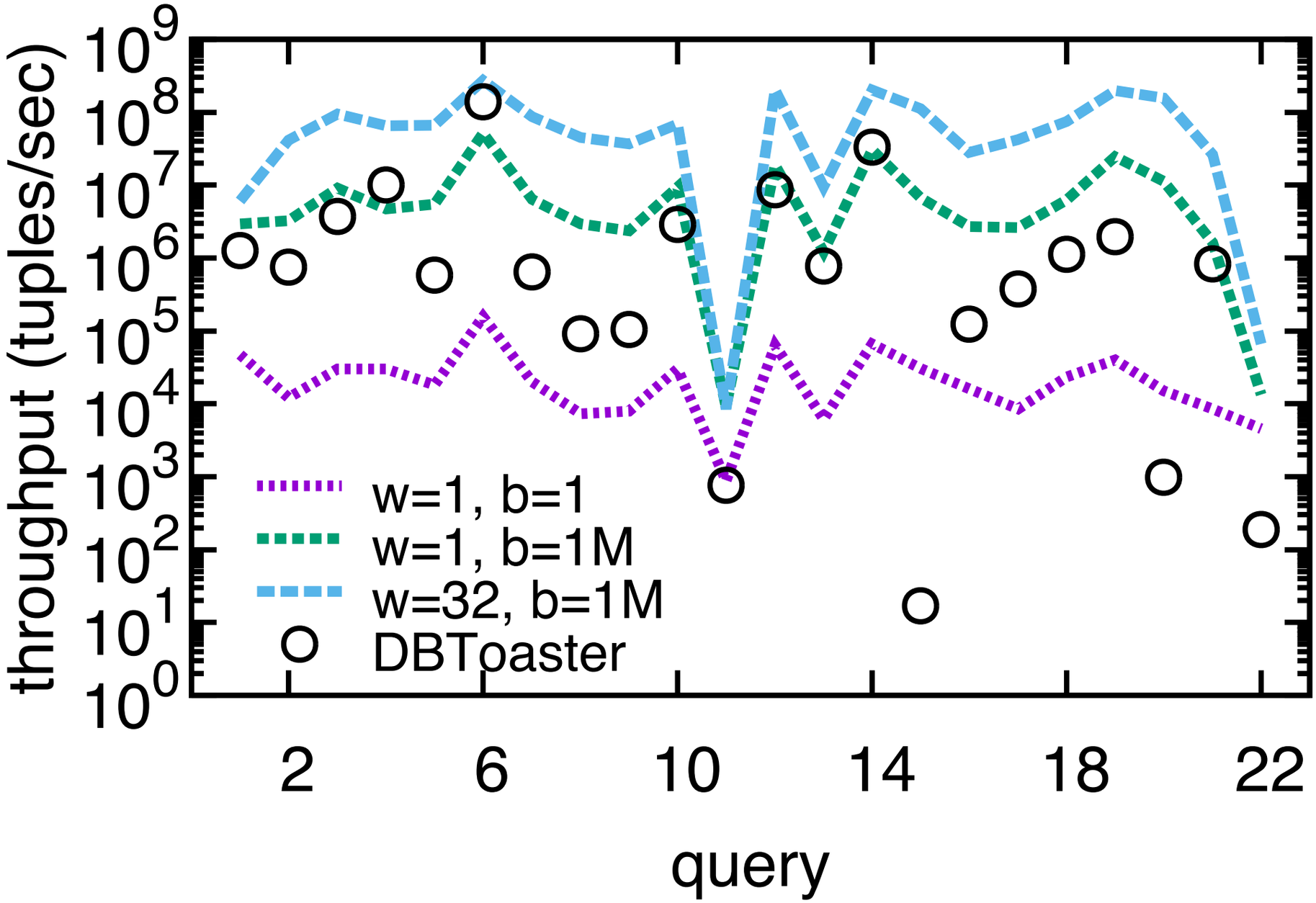

nodes
edges
join
concat
distinct

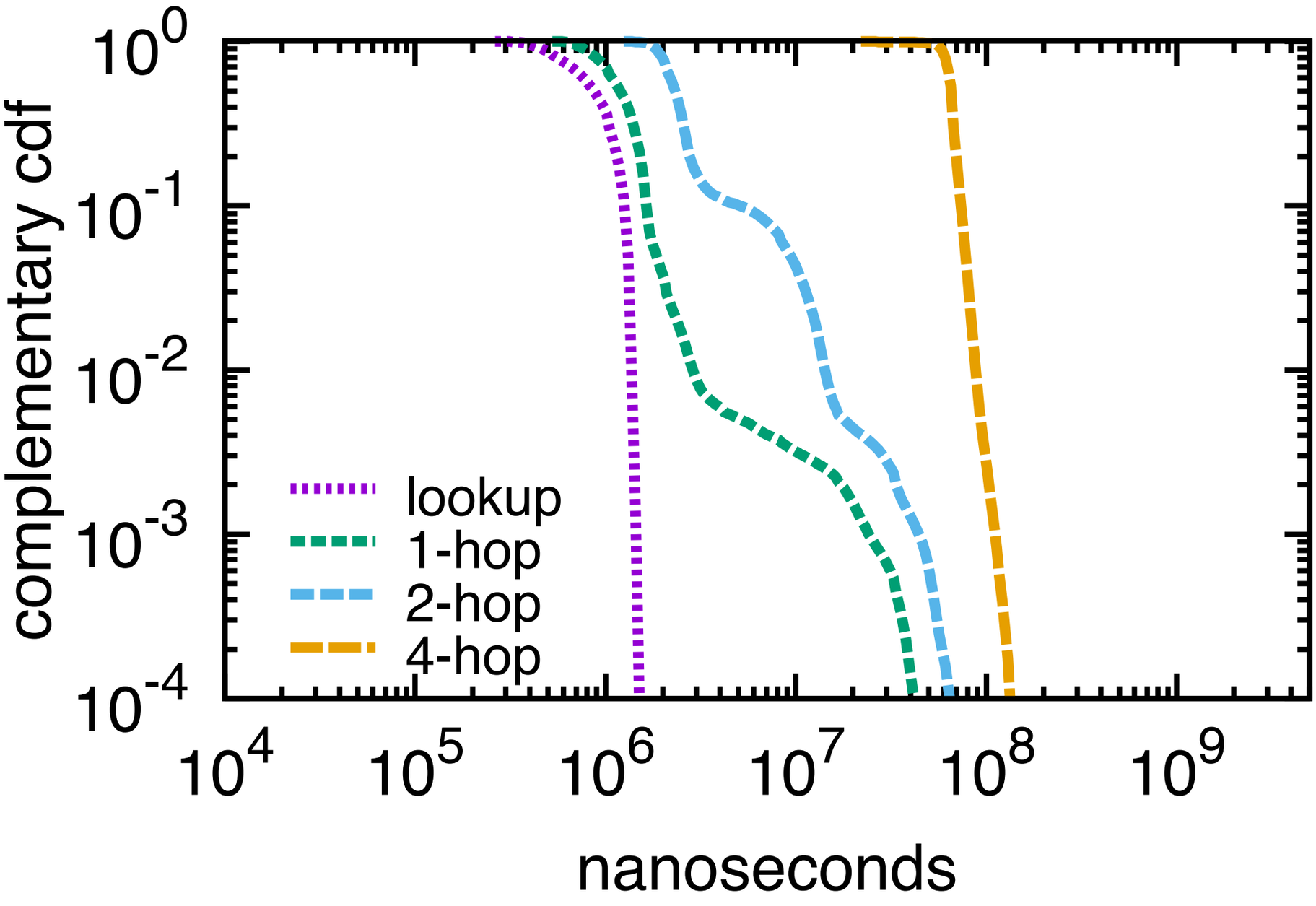

nodes
edges
join
concat
distinct
worker 0
nodes
edges
join
concat
distinct
worker 1


nodes
edges
join
concat
distinct


edges
queries
join
join
join


nodes
join
concat
distinct
edges
queries
join
join
join


throughput (tuples/sec)
10^9
10^8
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^0
w=1, b=1
w=1, b=1M
w=32, b=1M
DBToaster
2
6
10
14
18
22
query


10^0
10^{-1}
10^{-2}
10^{-3}
10^{-4}
complementary cdf
lookup
1-hop
2-hop
4-hop
10^4
10^5
10^6
10^7
10^8
10^9
nanoseconds

Dataflow	cores	httpd	psql	linux
SociaLite	4	4h	OOM	OOM
Graspan	4	11.3m	143.8m	713.8m

[Graspan, ASPLOS 2017]

Dataflow	cores	httpd	psql	linux
SociaLite	4	4h	OOM	OOM
Graspan	4	11.3m	143.8m	713.8m
Differential	1	10.9s	37.0s	76.8s
change (med)	1	22.0ms	185ms	1.11ms
change (max)	1	218ms	1.48s	8.13ms

PointsTo	cores	httpd	psql	linux
SociaLite	4	>24h	OOM	OOM
Graspan	4	479.9m	353.1m	99.7m

[Graspan, ASPLOS 2017]

PointsTo	cores	httpd	psql	linux
SociaLite	4	>24h	OOM	OOM
Graspan	4	479.9m	353.1m	99.7m
Differential	1	536.3s	362.0s	423.1s
Optimized	1	77.4s	75.9s	191.3s
-Sharing	1	91.9s	94.3s	401.7s

A quick taste:
Explaining outputs in modern data analytics
[VLDB2016]

Declarative Datalog debugging for mere mortals
[S. Köhler et al.]

"Describe provenance/lineage as Datalog queries"

Replace "Datalog" with Differential Dataflow:

1. More expressive (e.g. Map/Reduce/Aggr),
2. Incrementally maintained.

A quick taste:
Explaining outputs in modern data analytics
[VLDB2016]

Replace “Datalog” with Differential Dataflow:

1. More expressive (e.g. Map/Reduce/Aggr),
2. Incrementally maintained.

Explanations:

For a target subset of the observed output,
identify a subset of the input such that:
compute yields output containing the subset.

Differential Dataflow

current conclusions (ongoing)

Functional queries + timestamping = hugely powerful:

1. Automatic incrementalization (even w/loops).
2. Throughput scale-out w/o concurrency. (*c.f.* Calvin)
3. Shared state between multiple queries. (*c.f.* RDBMS)

Recovering RBDMS features w/o sacrificing usability.
We *are* sacrificing e.g. transactions. This is not OLTP.

Timely Dataflow

A data-parallel dataflow runtime

http://github.com/frankmcsherry/timely-dataflow

[based off of Naiad, SOSP 2013]

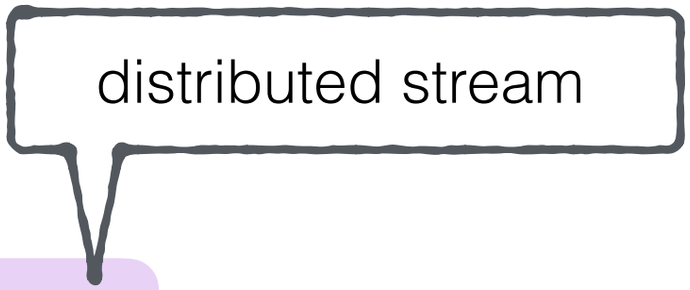
```
extern crate timely;

use timely::dataflow::operators::*;

fn main() {
    timely::example(|scope| {
        (0..10).to_stream(scope)
               .exchange(|&x| x)
               .inspect(|x| println!("{}", x));
    });
}
```

Your SPMD program.
You write this once.

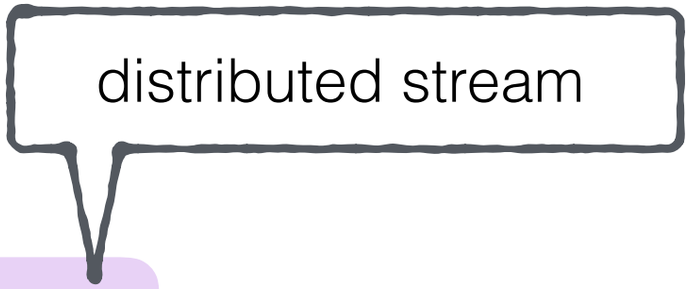
```
extern crate timely;

use timely::dataflow::operators::*;

fn main() {

    timely::example(|scope| {

        (0..10).to_stream(scope)
               .exchange(|&x| x)
               .inspect(|x| println!("{}", x));

    });

}
```

Rust iterator

```
extern crate timely;

use timely::dataflow::operators::*;

fn main() {

    timely::example(|scope| {

        (0..10).to_stream(scope)
                .exchange(|&x| x)
                .inspect(|x| println!(“{}”, x));

    });

}
```



```
extern crate timely;

use timely::dataflow::operators::*;

fn main() {

    timely::example(|scope| {
        (0..10).to_stream(scope)
               .exchange(|&x| x)
               .inspect(|x| println!(“{}”, x));
    });

}
```



```
extern crate timely;

use timely::dataflow::operators::*;

fn main() {

    timely::example(|scope| {
        (0..10).to_stream(scope)
               .exchange(|&x| x)
               .inspect(|x| println!("{}", x));
    });

}
```

distributed stream

```rust
fn main() {
    timely::execute_from_args(std::env::args(), |worker| {

    });
}
```

```
fn main() {
    timely::execute_from_args(std::env::args(), |worker| {

        let input = worker.new_input();
        let probe = worker.dataflow(|scope| {
            input.to_stream(scope)
                 .exchange(|&x| x)
                 .inspect(|x| println!("{}", x))
                 .probe()
        });

    });
}
```

Adds a timestamp to each record

Reports on possibility of timestamps

aph using a
new input and ending in a probe.

```rust
fn main() {
    timely::execute_from_args(std::env::args(), |worker| {

        let input = worker.new_input();
        let probe = worker.dataflow(|scope| {
            input.to_stream(scope)
                 .exchange(|&x| x)
                 .inspect(|x| println!("{}", x))
                 .probe()
        });

        for round in 0..10 {
            input.send(round * round);
            input.advance_to(round + 1);
            while probe.less_than(input.time()) {
                worker.step();
            }
        }
    });
}
```

Drive the dataflow by supplying inputs and running until cleared.

"What times might **probe** still see?"
Information about distributed state.

```rust
fn main() {
    timely::execute_from_args(std::env::args(), |worker| {

        let input = worker.new_input();
        let probe = worker.dataflow(|scope| {
            input.to_stream(scope)
                 .exchange(|&x| x)
                 .inspect(|x| println!("{}", x))
                 .probe()
        });

        for round in 0..10 {
            input.send(round * round);
            input.advance_to(round + 1);
            while probe.less_than(input.time()) {
                worker.step();
            }
        }
    });
}
```

```rust
.iterate(|cycle| {
    cycle.filter(|&x| x > 0)
         .map(|x| x - 1)
         .exchange(|&x| x)
})
```

exchange benchmark

one thread:	1us
two threads:	2us
two processes:	40us

```rust
fn main() {
    timely::execute_from_args(std::env::args(), |worker| {

        let input = worker.new_input();
        let probe = worker.dataflow(|scope| {
            input.to_stream(scope)
                                    <------
                .probe()
        });

        for round in 0..10 {
            input.send(round * round);
            input.advance_to(round + 1);
            while probe.less_than(input.time() - 2) {
                worker.step();
            }
        }
    });
}
```

```rust
.iterate(|cycle| {
    cycle.filter(|&x| x > 0)
         .map(|x| x - 1)
         .exchange(|&x| x)
})
```

concurrent iterations

better utilizations!

```
// convenient!
stream.map(logic);
```

Data movement requirements.
Could have been **Pipeline**.

behavior when scheduled.

```rust
// convenient!
stream.map(logic);

// equival
stream.una
    while let Some((time, data)) = input.next() {
        let
        for
            session.give(logi
        }
    }

});
```

Clarity on resource management: Ownership drives collection/reuse.

Destructuring ensures valid data

ained, reused

It’s basically just your code running.

Required to send

drain(..)

Memory / control safety at compile time. Fewer errors, more predictable behavior.

Abomonation

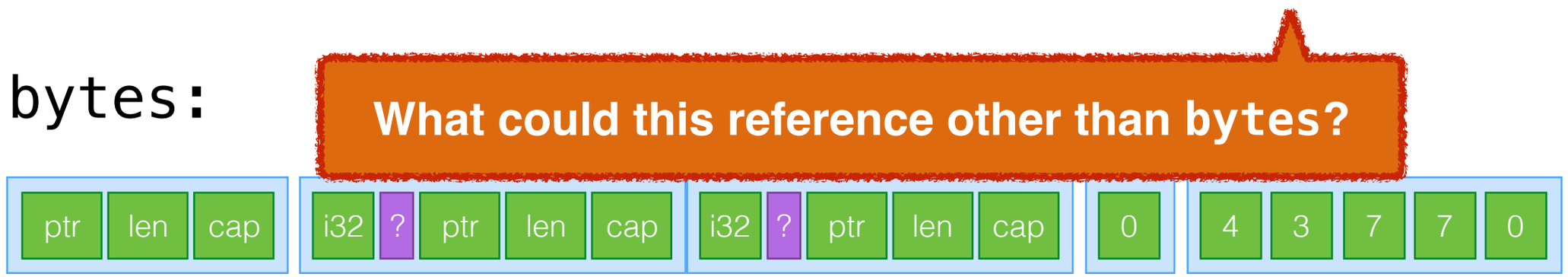
Serialization at memory bandwidth

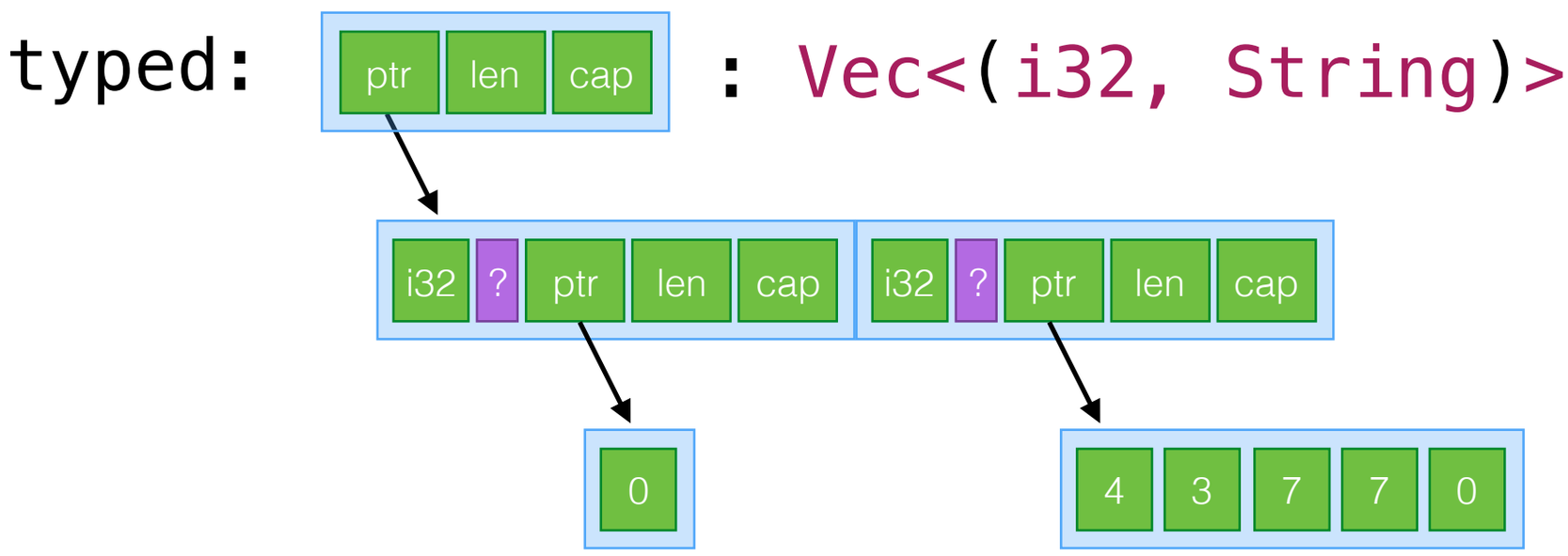
https://github.com/frankmcsherry/abomonation

The name is from the first reaction to the approach.

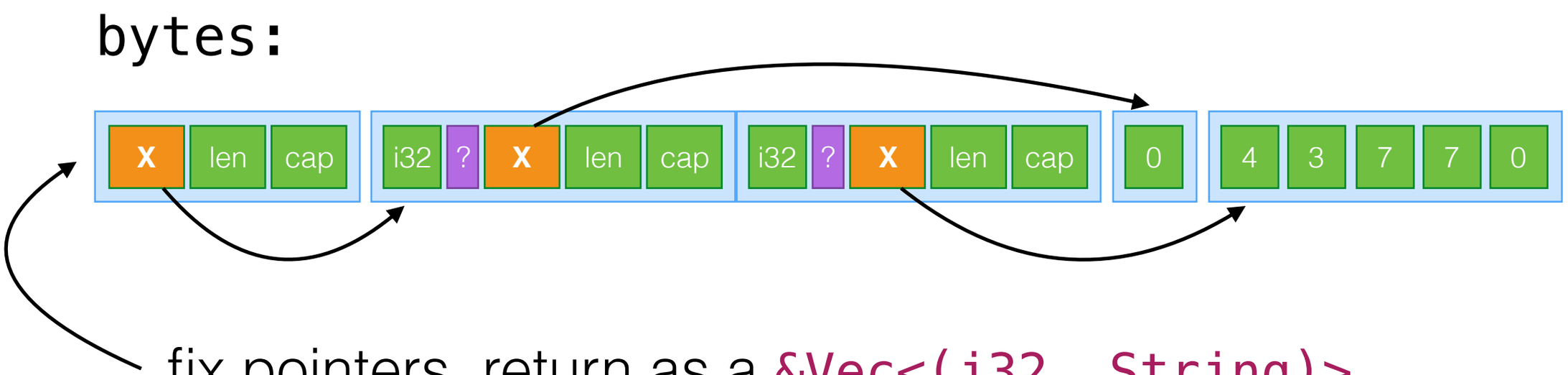
Be warned.

```rust
// serializes referenced typed data into bytes.
fn encode<T: Abomonation>(typed: &T, bytes: &mut Vec<u8>);

// presents a typed view of serialized data.
fn decode<T: Abomonation>(bytes: &mut [u8]) -> &T;
```





```rust
// serializes referenced typed data into bytes.
fn encode<T: Abomonation>(typed: &T, bytes: &mut Vec<u8>);

// presents a typed view of serialized data.
fn decode<T: Abomonation>(bytes: &mut [u8]) -> &T;
```



1. totally ignores alignment (x64: ok, arm: bad).
2. memcpy of padding bytes is apparently UB.
3. assumes same architecture, binary.
4. blah blah blah.

```rust
impl Log {
    pub fn new() -> Log {
        Log {
            timestamp: 2837513946597,
            zone_id: 123456,
            zone_plan: ZonePlan::FREE,
            http: Http {
                protocol: HttpProtocol::HTTP11,
                status: 200,
                request_uri: "/cdn-cgi/trace".to_owned(),
            },
            origin: Origin {
                ip: "1.2.3.4".to_owned(),
                port: 8000,
                hostname: "www.example.com".to_owned(),
                protocol: OriginProtocol::HTTPS,
            },
            country: Country::US,
            cache_status: CacheStatus::Hit,
            server_ip: "192.168.1.1".to_owned(),
            server_name: "metal.cloudflare.com".to_owned(),
            remote_ip: "10.1.2.3".to_owned(),
            bytes_dlv: 123456,
            ray_id: "10c73629cce30078-LAX".to_owned(),
        }
    }
}
```

```rust
unsafe_abomonate!(Log : http, origin, server_ip, server_name, remote_ip, ray_id);
unsafe_abomonate!(Http : content_type, user_agent, referer, request_uri);
unsafe_abomonate!(Origin : ip, hostname);
```

```rust
#![feature(test)]
#[macro_use]
extern crate abomonation;
extern crate test;

use test::Bencher;
use abomonation::{Abomonation, encode, decode};

#[bench]
fn bench_populate(b: &mut Bencher) {
    b.iter(|| {
        Log::new()
    });
}
```

```
bench_populate:          267 ns/iter (+/- 124)
```

```rust
#[bench]
fn bench_populate(b: &mut Bencher) {
    b.iter(|| {
        Log::new()
    });
}

#[bench]
fn bench_serialize(b: &mut Bencher) {
    let log = Log::new();
    let mut bytes = vec![];
    unsafe { encode(&log, &mut bytes); }
    b.bytes = bytes.len() as u64;
    b.iter(|| {
        bytes.clear();
        unsafe { encode(&log, &mut bytes); }
        test::black_box(&bytes);
    });
}
```

```
bench_populate:          267 ns/iter (+/- 124)
```

```
 9  #[bench]
10  fn bench_populate(b: &mut Bencher) {
11      b.iter(|| {
12          Log::new()
13      });
14  }
15
16  #[bench]
17  fn bench_serialize(b: &mut Bencher) {
18      let log = Log::new();
19      let mut bytes = vec![];
20      unsafe { encode(&log, &mut bytes); }
21      b.bytes = bytes.len() as u64;
22      b.iter(|| {
23          bytes.clear();
24          unsafe { encode(&log, &mut bytes); }
25          test::black_box(&bytes);
26      });
27  }
```

```
bench_populate:          267 ns/iter (+/- 124)
bench_serialize:          54 ns/iter (+/- 4)  = 10148 MB/s
```

```
    b.iter(|| {
        bytes.clear();
        unsafe { encode(&log, &mut bytes); }
        test::black_box(&bytes);
    });
}

#[bench]
fn bench_deserialize(b: &mut Bencher) {
    let log = Log::new();
    let mut bytes = vec![];
    unsafe { encode(&log, &mut bytes); }
    b.bytes = bytes.len() as u64;
    b.iter(|| {
        test::black_box(unsafe { decode::<Log>(&mut bytes) });
    });
}
```

```
bench_populate:            267 ns/iter (+/- 124)
bench_serialize:            54 ns/iter (+/- 4)  = 10148 MB/s
```

```
    b.iter(|| {
        bytes.clear();
        unsafe { encode(&log, &mut bytes); }
        test::black_box(&bytes);
    });
}

#[bench]
fn bench_deserialize(b: &mut Bencher) {
    let log = Log::new();
    let mut bytes = vec![];
    unsafe { encode(&log, &mut bytes); }
    b.bytes = bytes.len() as u64;
    b.iter(|| {
        test::black_box(unsafe { decode::<Log>(&mut bytes) });
    });
}
```

```
bench_populate:          267 ns/iter (+/- 124)
bench_serialize:          54 ns/iter (+/- 4)  = 10148 MB/s
bench_deserialize:         8 ns/iter (+/- 1)  = 68500 MB/s
```

```
22      b.iter(|| {
23          bytes.clear();
24          unsafe { encode(&log, &mut bytes); }
25          test::black_box(&bytes);
26      });
27  }
28
29  #[bench]
30  fn bench_deserialize(b: &mut Bencher) {
31      let log = Log::new();
32      let mut bytes = vec![];
33      unsafe { encode(&log, &mut bytes); }
34      b.bytes = bytes.len() as u64;
35      b.iter(|| {
36          test::black_box(unsafe { decode::<Log>(&mut bytes) });
37      });
38  }
```

It's even faster than this

This is really fast

```
bench_populate:           267 ns/iter (+/- 124)
bench_serialize:           54 ns/iter (+/- 4)   = 10148 MB/s
bench_deserialize:          8 ns/iter (+/- 1)   = 68500 MB/s
bench_deserialize_test: 112 ns/iter (+/- 25) =  4892 MB/s
```

Projects

http://github.com/frankmcsherry/timely-dataflow

http://github.com/frankmcsherry/differential-dataflow

http://github.com/frankmcsherry/abomonation

Projects

http://github.com/frankmcsherry/timely-dataflow
http://github.com/frankmcsherry/differential-dataflow
http://github.com/frankmcsherry/abomonation

Moar projects

http://github.com/frankmcsherry/recycler
http://github.com/frankmcsherry/columnar

Projects

http://github.com/frankmcsherry/timely-dataflow
http://github.com/frankmcsherry/differential-dataflow
http://github.com/frankmcsherry/abomonation

Moar projects

http://github.com/frankmcsherry/recycler
http://github.com/frankmcsherry/columnar

Scandalous blog:

http://github.com/frankmcsherry/blog

Projects

http://github.com/frankmcsherry/timely-dataflow
http://github.com/frankmcsherry/differential-dataflow
http://github.com/frankmcsherry/abomonation

Moar projects

http://github.com/frankmcsherry/recycler
http://github.com/frankmcsherry/columnar

Scandalous blog:

http://github.com/frankmcsherry/blog

Excited to have more eyeballs, help, criticism. stars…