Data Structure Adventures

Erlang Factory 2015

Joe Blomstedt

Basho Technologies

Erlang is highly productive language

Scalable

Distributed

Fault Tolerant

Performance envy

Solution

Write a NIF!

Warning

Use this functionality with extreme care!

A native function is executed as a direct extension of the native code of the VM. Ex
can **not** provide the same services as provided when executing Erlang code, such a
the native function doesn't behave well, the whole VM will misbehave.

- A native function that crash will crash the whole VM.
- An erroneously implemented native function might cause a VM internal state i
  miscellaneous misbehaviors of the VM at any point after the call to the native
- A native function that do **`lengthy work`** before returning will degrade respons
  strange behaviors. Such strange behaviors include, but are not limited to, ext
  between schedulers. Strange behaviors that might occur due to lengthy work

Data structures
Dispatch/protection
Statistics

Data Structures

orddict

dict

gb_trees

	1000	10k	100k	1mm	10mm
orddict	25	2770	--	--	--
dict	2	21	315	16485	--
gb_trees	3	44	577	8095	--

	1000	10k	100k	1mm	10mm
bt	4	59	708	8894	--
dict	2	21	315	16485	--
gb_trees	3	44	577	8095	--

Immutable

Shared structure

```
D1 = dict:new(),

D2 = dict:store(1, 10, D1),
D3 = dict:store(1, 15, D2),

10 = dict:fetch(1, D2),
15 = dict:fetch(1, D3).
```

	1000	10k	100k	1mm	10mm
ets	8	8	49	497	5296
dict	2	21	315	16485	--
gb_trees	3	44	577	8095	--

Concurrent
Fast
Off Heap

Immutable

Shared Structure

Concurrent

Fast

Off Heap

CoW B-Tree

basho

riak

root

(1,10),(5,50),(9,90)

root (1,A),(5,B)

A (1,10),(2,20)

B (5,50),(9,90)

basho

riak

root (1,A),(5,B)

A (1,10),(2,20),(3,30)

B (5,50),(9,90)

root (1,A),(5,B)

A (1,10),(2,20),(3,30)

B (5,50),(6,60),(9,90)

root (1,A),(3,C),(5,B)

A (1,10),(2,20)

C (3,30),(4,40)

B (5,50),(6,60),(9,90)

basho

riak

root (1,E),(5,F)

E (1,A),(3,C)

F (5,B),(7,D)

A (1,10),(2,20)

C (3,30),(4,40)

B (5,50),(6,60)

D (7,70),(9,90)

basho

riak

```
[{1,
  [{1,[{1,10},{2,20}]},
   {3,[{3,30},{4,40}]}]},
 {5,
  [{5,[{5,50},{6,60}]},
   {7,[{7,70},{9,90}]}]}]
```

```
find(Key, #tree{height=Height, root=Root}) ->
    find(1, Height, Key, Root).

find(Depth, Height, Key, Node)
  when Depth == Height ->
    %% leaf node
    orddict:find(Key, Node);

find(Depth, Height, Key, Node) ->
    %% inner node
    {_, Child} = search(Node, Key),
    find(Depth + 1, Height, Key, Child).
```

	1000	10k	100k	1mm	10mm
bt	4	59	708	8894	--
dict	2	21	315	16485	--
gb_trees	3	44	577	8095	--

Immutable

Shared Structure

Concurrent

Fast

Off Heap

basho

riak

Rewrite as a NIF

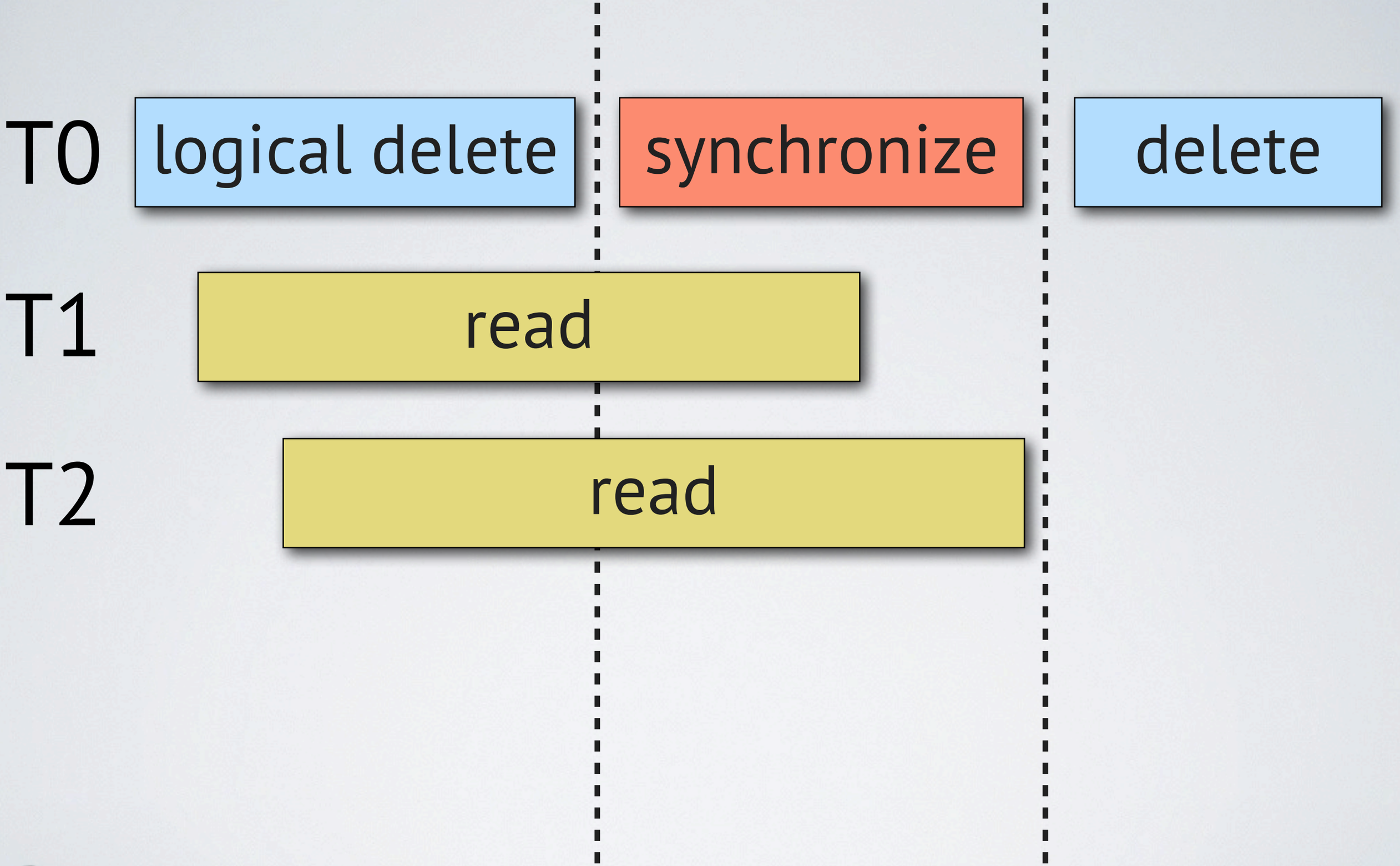
Allocation Snapshots

Reclamation (SMR)

Atomics / Ordering

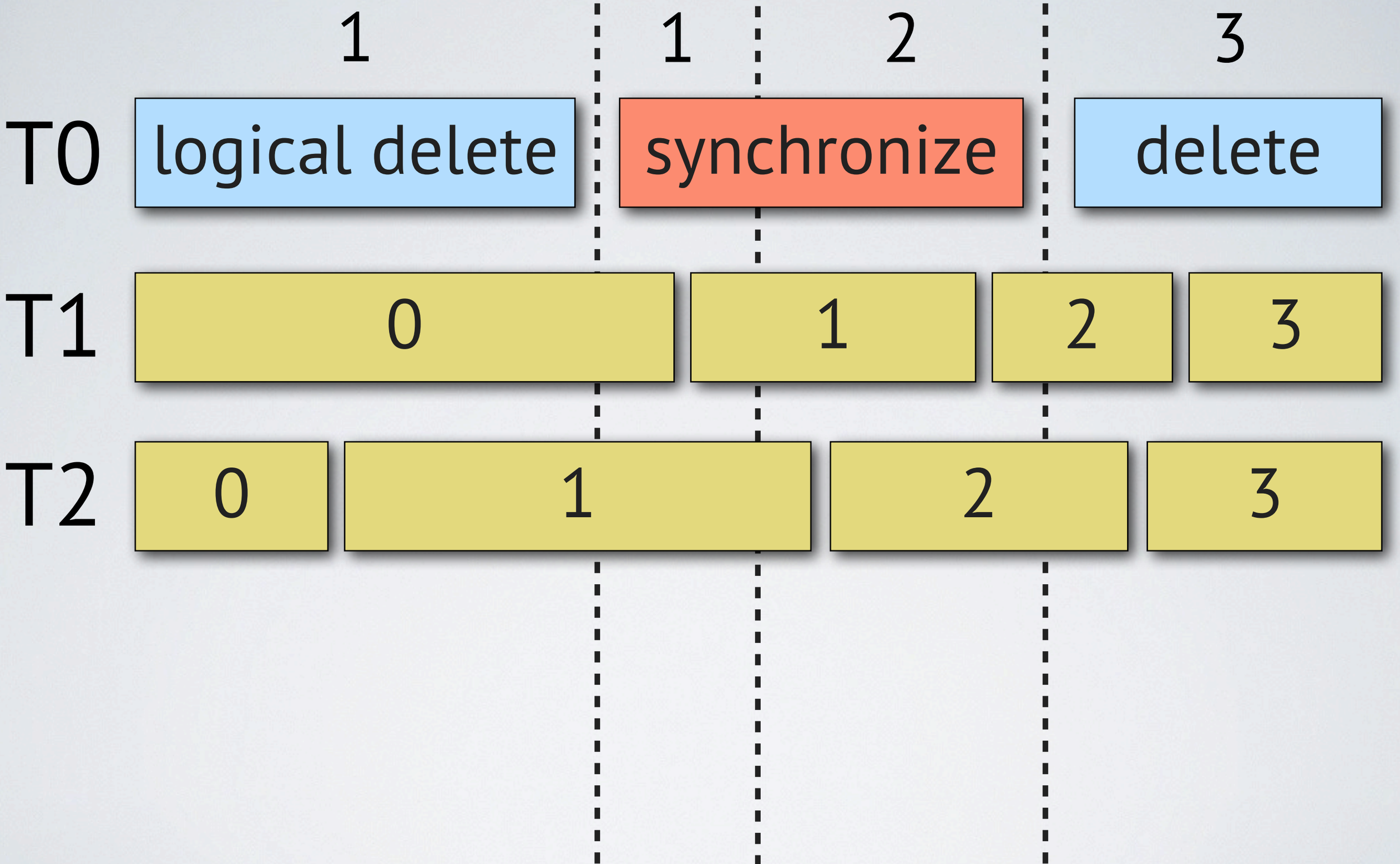
Epoch Reclamation

Grace Period Detection



basho

riak


1
1
2
3
T0
logical delete
synchronize
delete
T1
0
1
2
3
T2
0
1
2
3

```
std::atomic<Root*> root;

void reader() {
  while(true) {
    epoch_begin();
    Root *r = root;
    do_something(r);
    epoch_end();
  }
}
```

```
void writer() {
  while(true) {
    Root *old_root = root;
    Root *new_root = update(root);
    root = new_root;
    epoch_synchronize();
    delete root;
  }
}
```

```
std::atomic<Root*> root;
uint64_t epoch;

void reader() {
  while(true) {
    epoch_begin();
    Root *r = root;
    snapshot(r.epoch);
    epoch_end();
    do_something(r);
  }
}
```

```
void writer() {
  while(true) {
    epoch++;
    Root *old_root = root;
    Root *new_root = update(root);
    new_root->birth = epoch;
    old_root->death = epoch;
    root = new_root;
    garbage.push_back(old_root);
    collect();
  }
}
```

```
void collect() {
  epoch_synchronize();
  for(auto *item : garbage) {
    bool live = false;
    for(auto epoch : snapshots) {
      if((epoch >= item->birth) &&
         (epoch  < item->death)) {
        live = true;
        break;
      }
    }
    if(live)
      keep.push_back(item);
    else
      delete item;
  }
  garbage.swap(keep);
}
```

Flat Combining

(request)

ready?

(request)

ready?

(request)

ready?

locked?

requests[]

(request)

ready?

(request)

ready?

(request)

ready?

locked?

requests[]

(request)

ready?

locked?

requests[]

(request)

ready?

(request)

ready?

(request)

ready?

locked?

requests[]

(request)

ready?

(request)

ready?

(request)

ready?

locked?

requests[]

(request)

ready?

(request)

ready?

(request)

ready?

locked?

requests[]

(request)

ready?

(request)

ready?

(request)

ready?

(request)

ready?

(request)

ready?

locked?

requests[]

```
B1 = btn:new(),

B2 = btn:store(1, 10, B1),
B3 = btn:store(1, 15, B2),
```

```
10 = btn:fetch(1, B1),
15 = btn:fetch(1, B2).
```

	1000	10k	100k	1mm	10mm
bt	4	59	708	8894	--
ets	8	8	49	497	5296
bt_nif	11	23	267	2393	20908

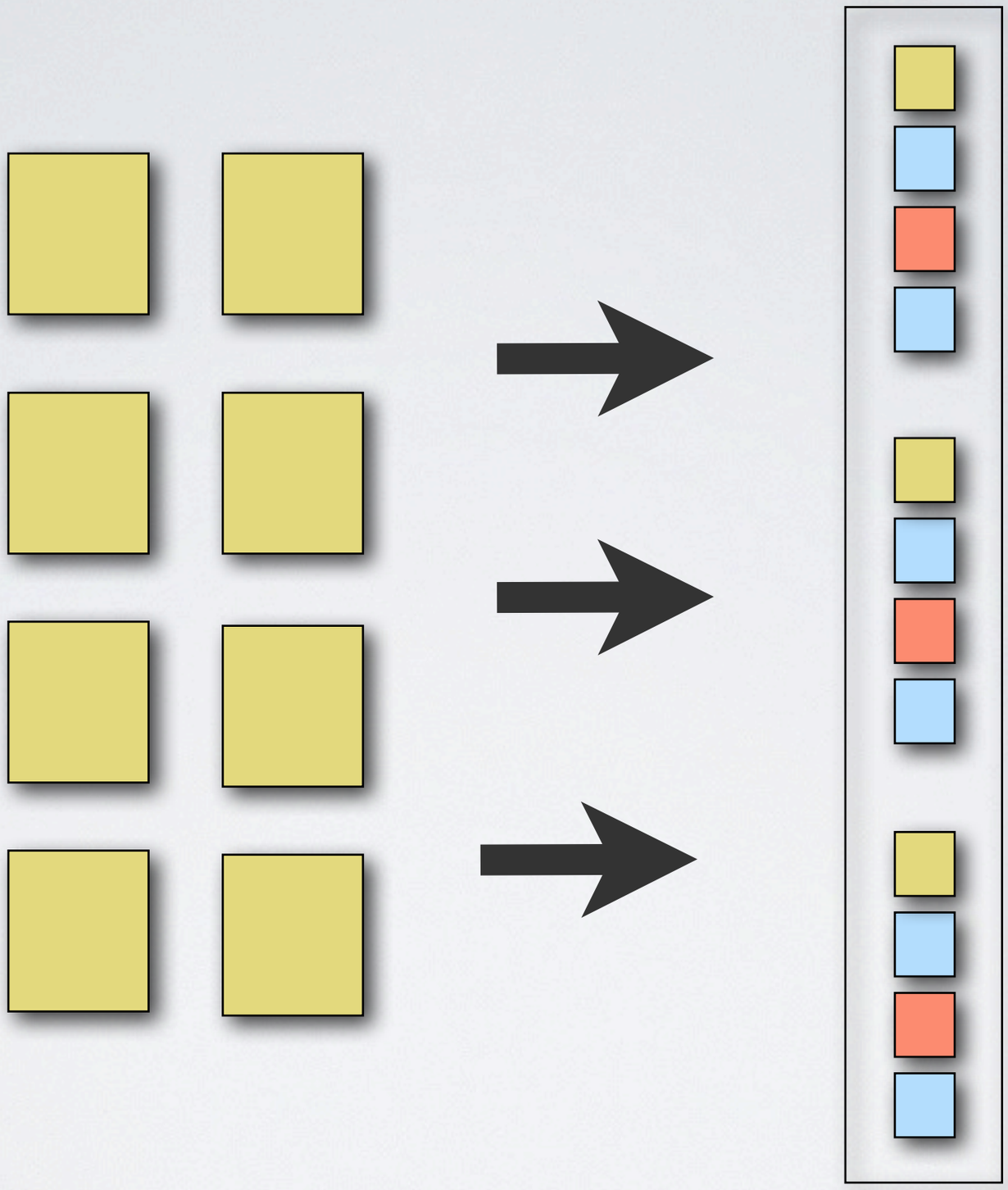
basho

riak

```
btn:m_new(test),
btn:m_store(test, 1, 10).
btn:m_store(test, 1, 15),
spawn(fun() ->
                15 = btn:m_fetch(1)
      end).
```

	1000	10k	100k	1mm	10mm
bt_nif2	3	6	68	744	8513
ets	8	8	49	497	5296
bt_nif	11	23	267	2393	20908

Worker Dispatch

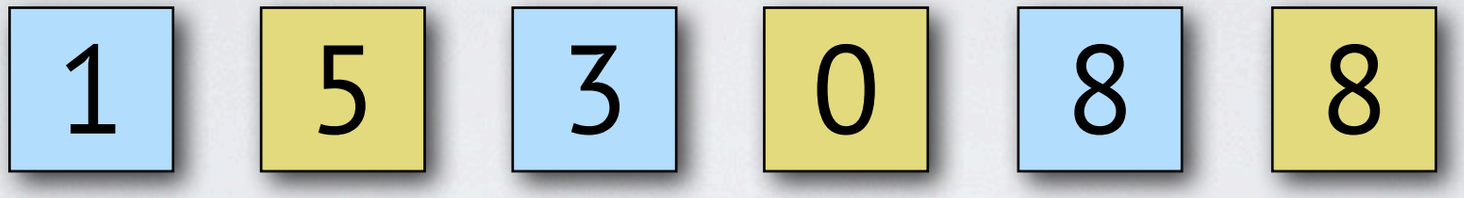


Load balancing

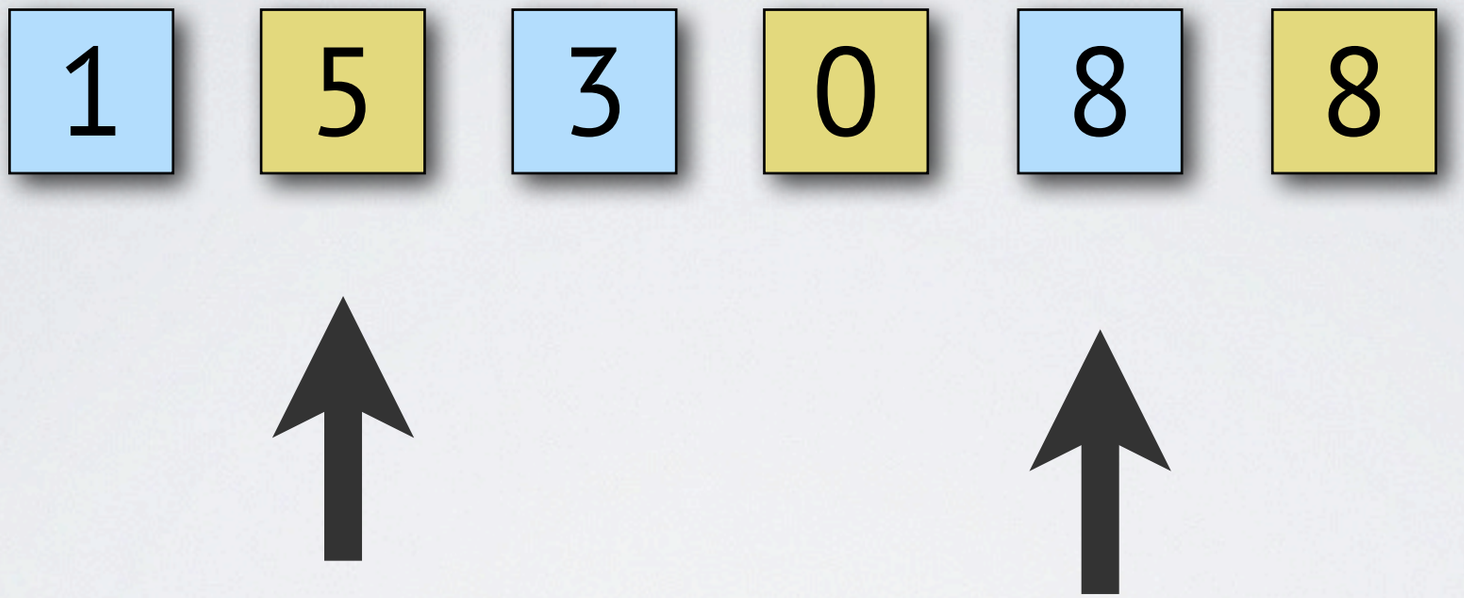
Overload protection

sidejob

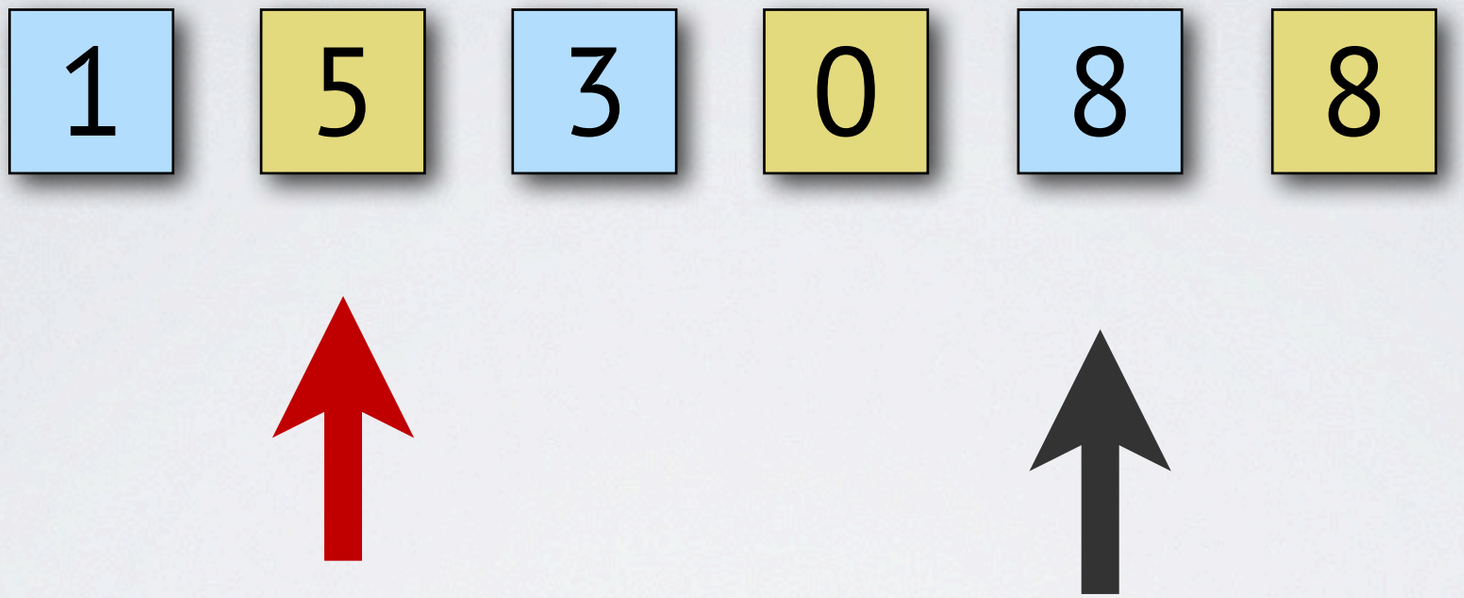
message counters (ETS)



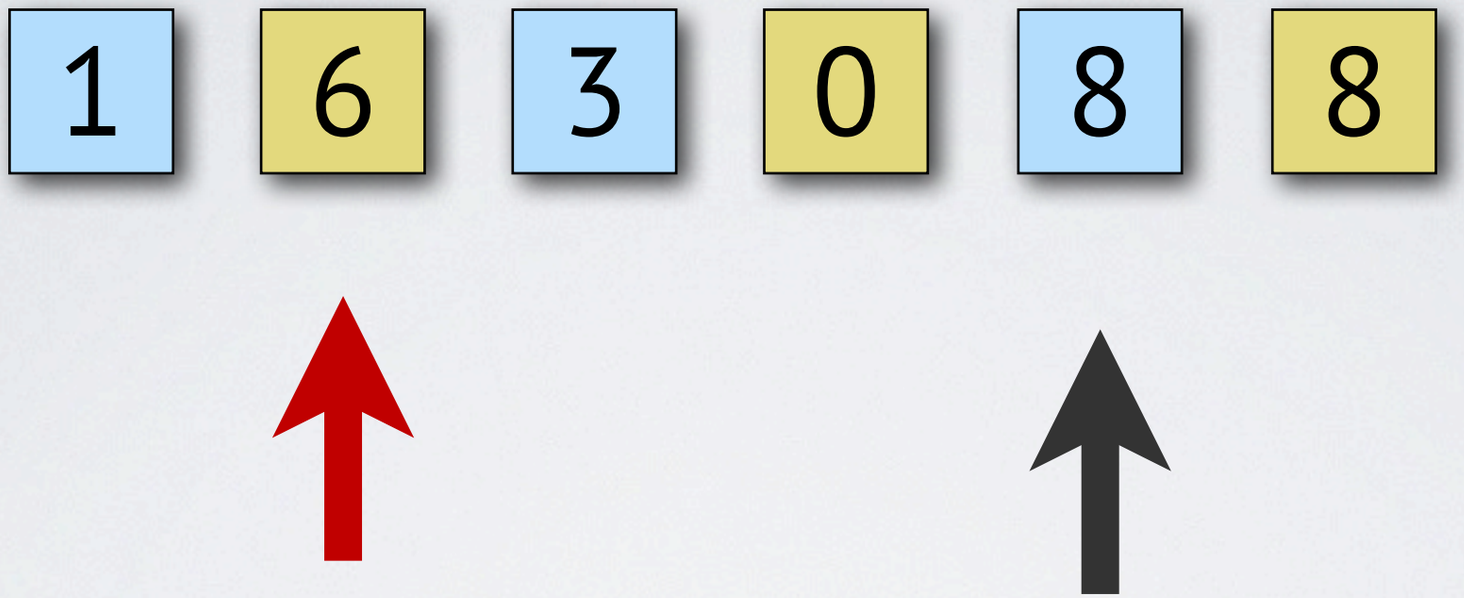
message counters (ETS)



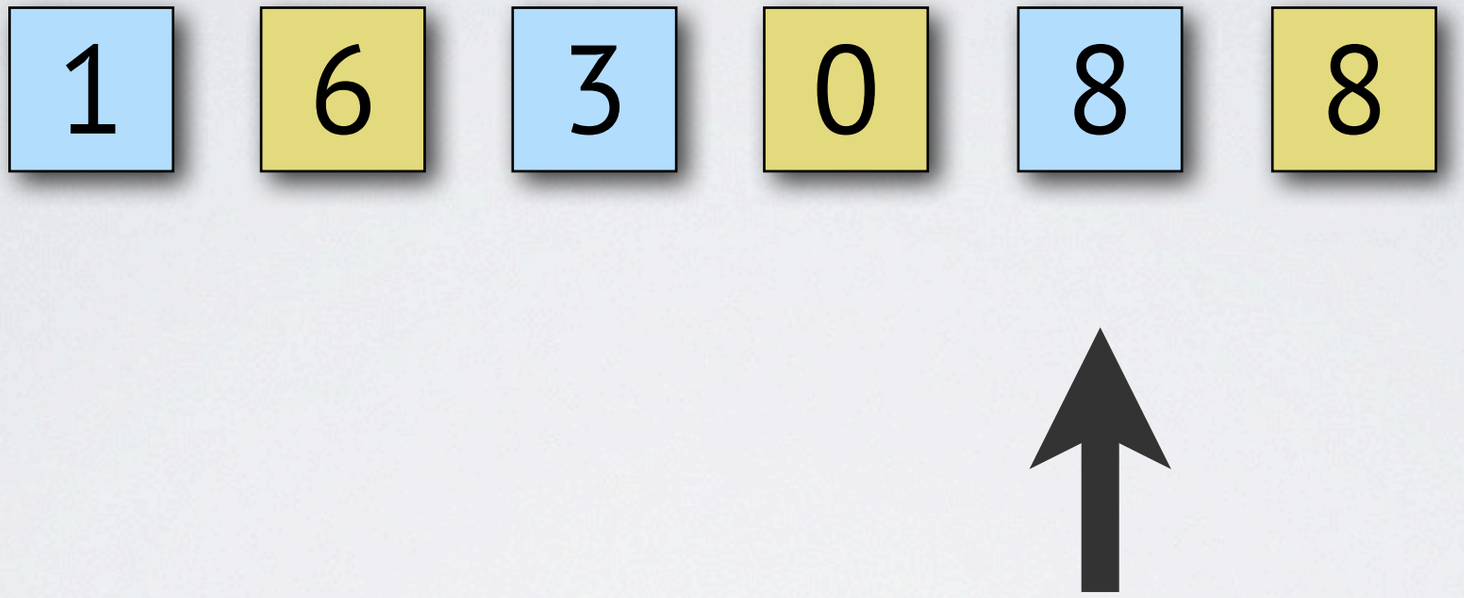
message counters (ETS)



message counters (ETS)



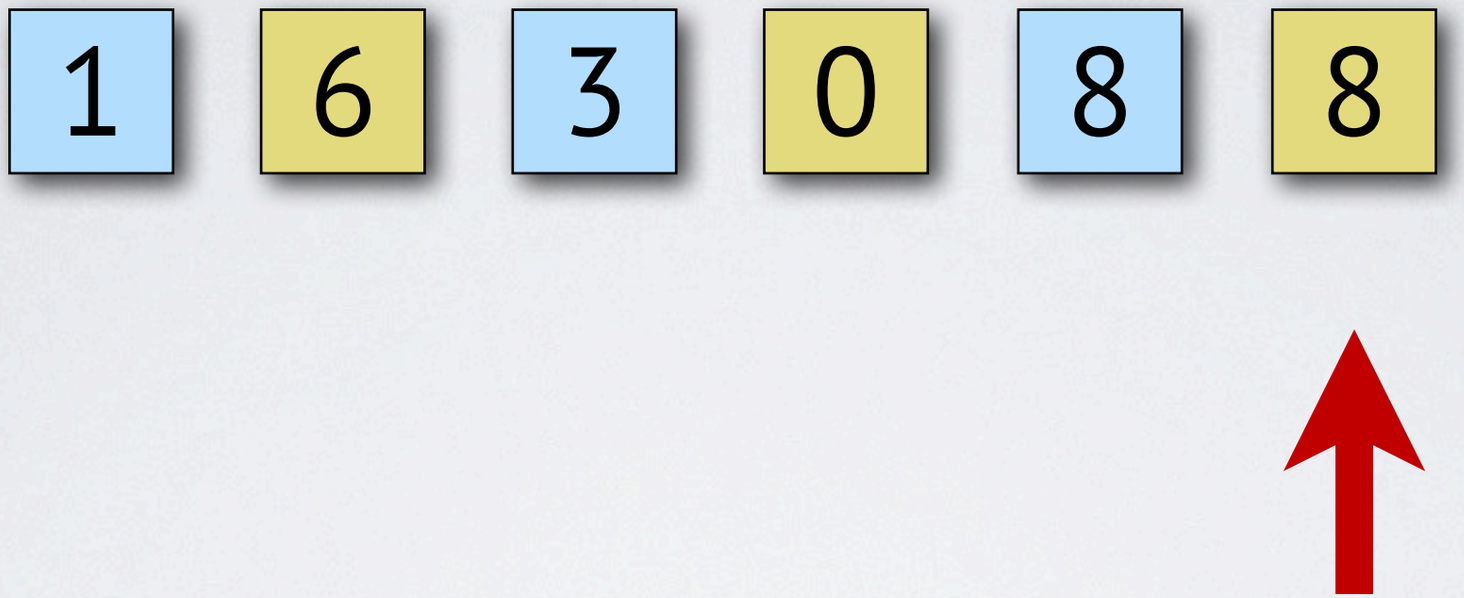
message counters (ETS)



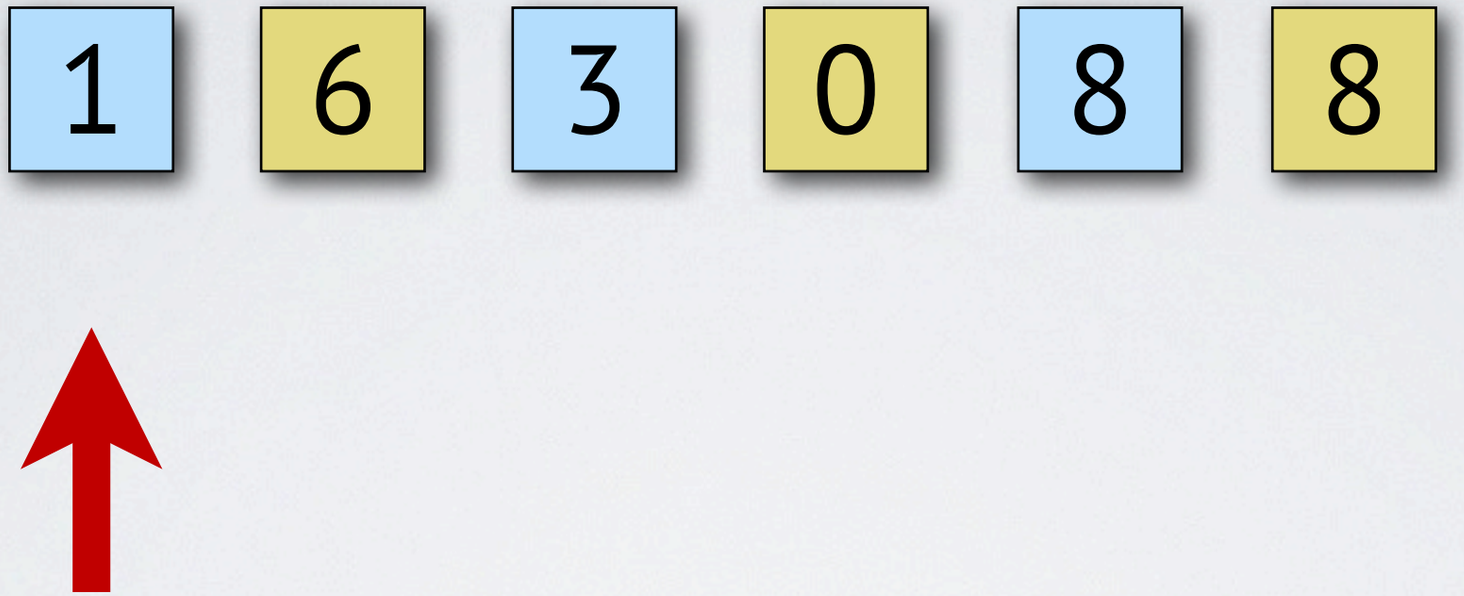
message counters (ETS)

1 6 3 0 8 8

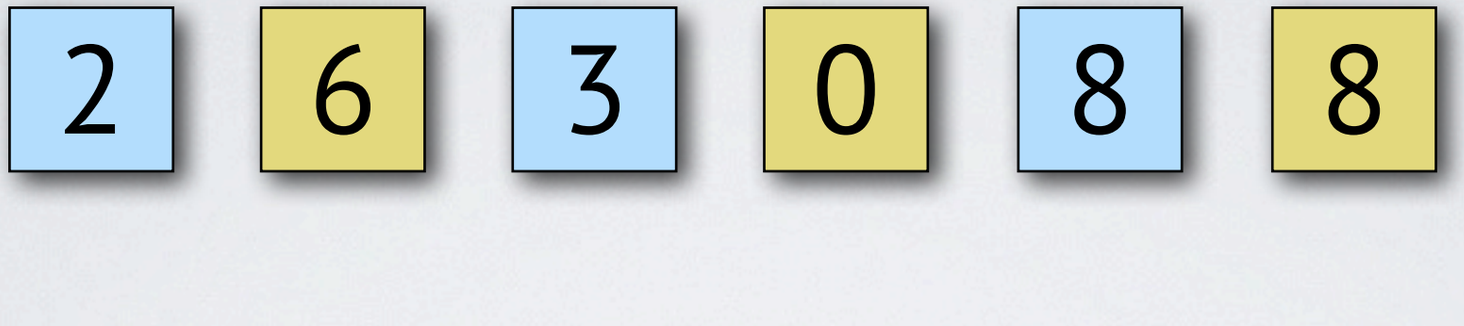
message counters (ETS)



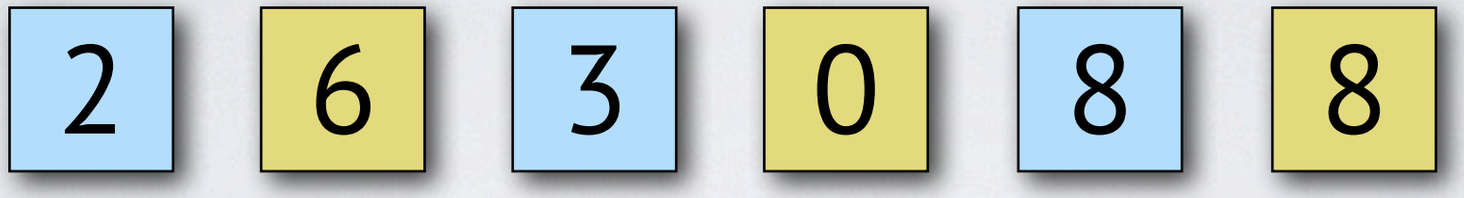
message counters (ETS)



message counters (ETS)



message counters (ETS)



dispatch NIF

```
dispatch:new(test).

sender() ->
    Pid = dispatch:find(test),
    Pid ! Msg,
    ok.

worker() ->
    Name = dispatch:listen(test, self()),
    worker(Name).

worker(Name) ->
    receive Msg ->
            do_something(Msg),
            dispatch:ack(test, Name),
            worker(Name)
    end.
```

Faster under contention

(About 1.5x)

Statistics

folsom

exometer

(others?)

Challenge

min/max

mean

latency percentiles

metrics++

Riak issue for years

Optimized counters in Riak 1.3

“Scheduler” Partitioned counters in ETS

Histograms still a challenge

Reservoir sampling (ETS)

Let's write a NIF!

min/max via atomics

```
template<class TA, class T>
void atomic_max(TA &atomic, T val) {
  T current = atomic.load(std::memory_order_relaxed);
  while(val > current) {
    current = val;
    val = atomic.exchange(val);
  }
}
```

Normal reservoir sampling with atomic increments

200 metrics

10 million events each

8-16 workers

30-40s runtime

50 million events/s

Future Work

Partitioned Reservoir sampling

http://gregable.com/2007/10/reservoir-sampling.html

Conclusion

NIFs can help

Optimized reusable components

NIFs are hard

Garbage collection/SMR

Allocation

Copying

Erlang + NIFs are hard

But, think it's work it

Much more work to do

github.com/jtuple/ef2015

Questions?