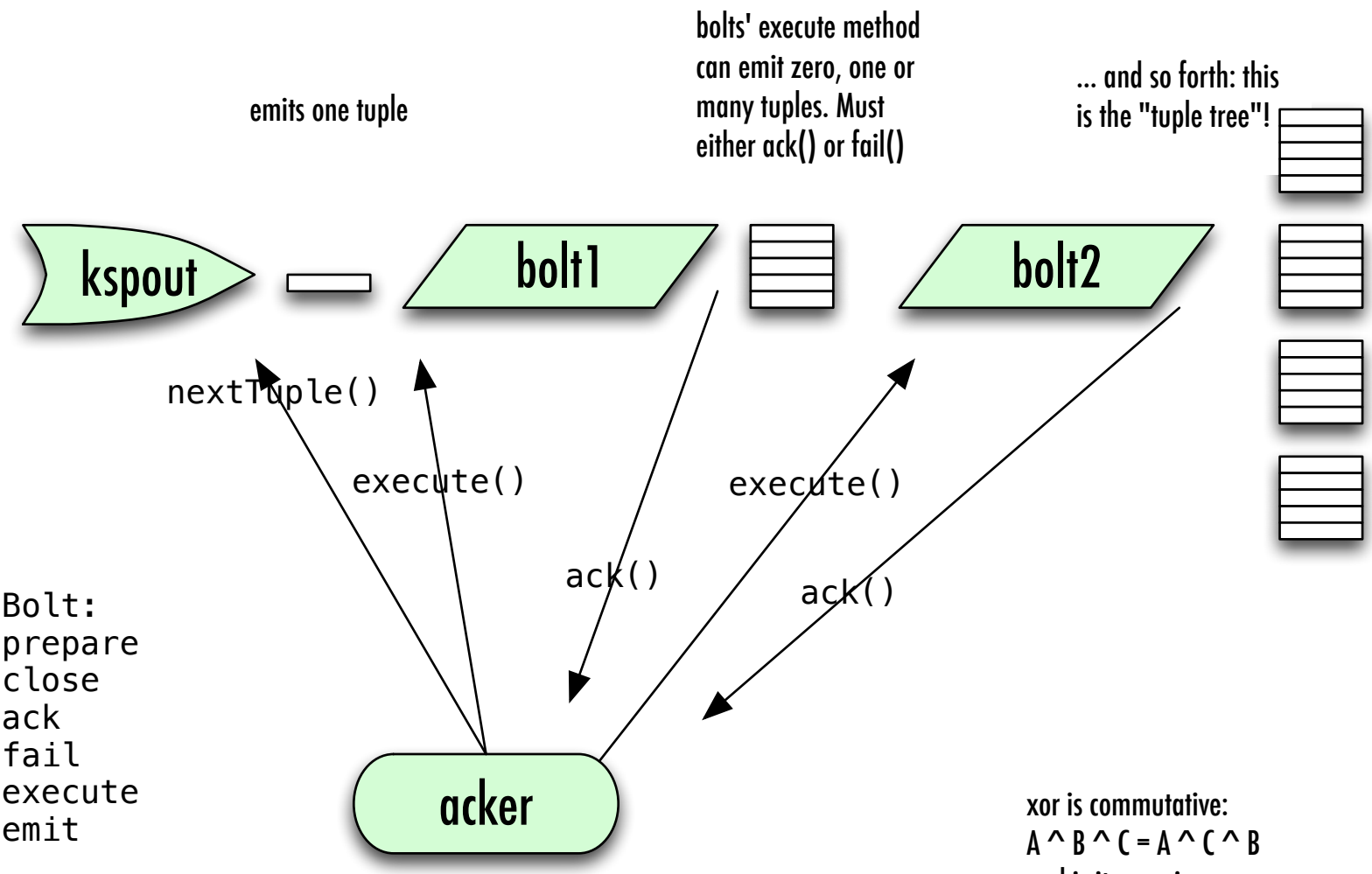
Storm

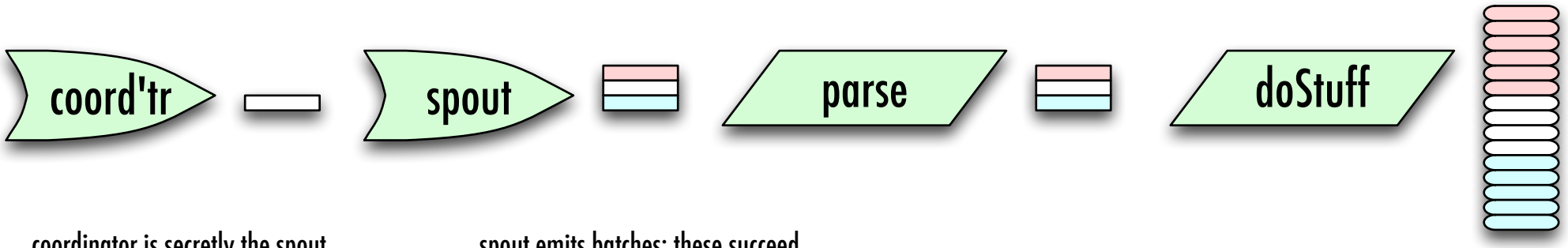



```
Spout:     Bolt:
prepare    prepare
close      close
ack        ack
fail       fail
nextTuple  execute
           emit
```

Every tuple tree is as good as (and processed independently and concurrently) with every other tuple tree.
No ordering, no blocking, no awareness of other tuples, and a tuple might be processed many times

acker tracks acks. Each [bolt,tuple] pair has a unique-enough 64-bit global ID, which is XOR'ed when tuple is dispensed, and XOR'ed when it is acked.
https://github.com/nathanmarz/storm/wiki/Guaranteeing-message-processing

xor is commutative:
A ^ B ^ C = A ^ C ^ B
and is its own inverse:
A ^ A = 0
So if every execute has an ack, the combined XORs will be zero, always.
In principle, you can XOR a bunch of numbers to be zero as well, but it is exceptionally unlikely.
QUESTION: does also check count?



coordinator is secretly the spout
at trident batch delay period,
will emit a transaction tuple
it has a serially incrementing
transaction ID, kept forever
even across restarts.
(We're only going to talk about
Opaque Transactional topologies btw)

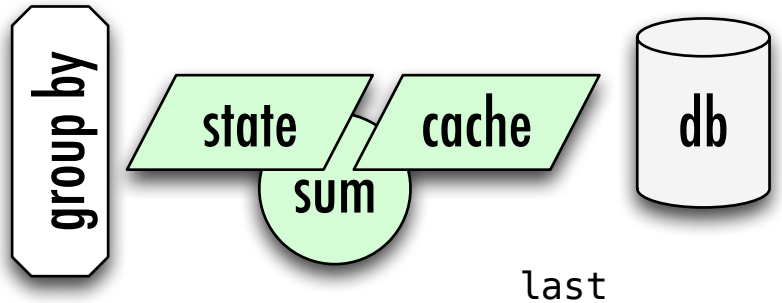
spout emits batches; these succeed
or fail as a whole. (That's because
they're part of the Storm tuple
tree of the coordinator's seed
tuple).

tuples in batches are freely processed in parallel and asynchronously unless there are barriers (eg the state after the group by)

TODO: where is the blocking behavior for state code

Trident:
* exactly once processing
* transactional db support
* layer on the flow DSL
* some primitive aggregators

non-transactional: batching behavior only
transactional: exactly once; batches are always
processed in whole
opaque transactional: all records are processed, but
might not be in same batches



`last acking`

the state doesn't ask the cache to fetc until it has a whole batches' worth of records to hand over. This is trident logic, not storm.

Those "aggregables" are reduced into rolled-up aggregates. So you might have 2500 inbound records that resul in 900 distinct aggregates. (If you ha eight aggregables [A, A, C, A, B, D, B, A] you would get four partial aggregates {A: 4, B: 2, C: 1, D: 1}.

It's clever about doing partial aggregates ("algebraic" reducers).

It looks in the cache for the old total count. Anything that isn't there it fetches from the database. This lets you do efficient batch requests, a huge scalability boon.
Once the cache is fresh, it determines the next aggregated value and writes it to the cache and to the DB, then ack()s the batch (all the tuples in the batch, really).
If a batch had 900 aggregates, and it had prior counts for 250 of them, then it will _read_ 650 records and _write_ 900. It always does a put for every new observed count.

¡Note!: The database writes do *not* have to be transactional. It's the whole thing – the whole batch, end-to-end – that has to have transactional integrity, not just the DB code.

Let's say for transaction ID 69 the old aggregated values were {A:20, B: 10, C: 1, D: 0}, and new aggregated values were {A: 24, B: 12, C: 2, D: 1}.

It stores {A: [24, 20, 69], B: [12, 10, 69], C: [2, 1, 69], D: [1, 0, 69]}

TODO: verify order of that list

If I am processing batch

Since this is a _State_, you have contractual obligation from Trident that batch 69 will *not* be processed until and unless batch 68 has succeeded.

So when I go to read from the DB, I will usually see something like

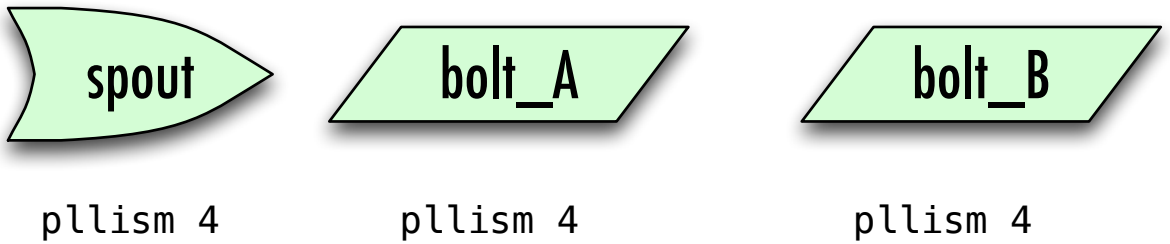
{A: [20, ??, 68], B: [10, ??, 68], C: [1, ??, 68]}

I might instead however see

{A: [??, 20, 69], B: [??, 10, 69], C: [??, 1, 69], D: [??, 0, 69]}

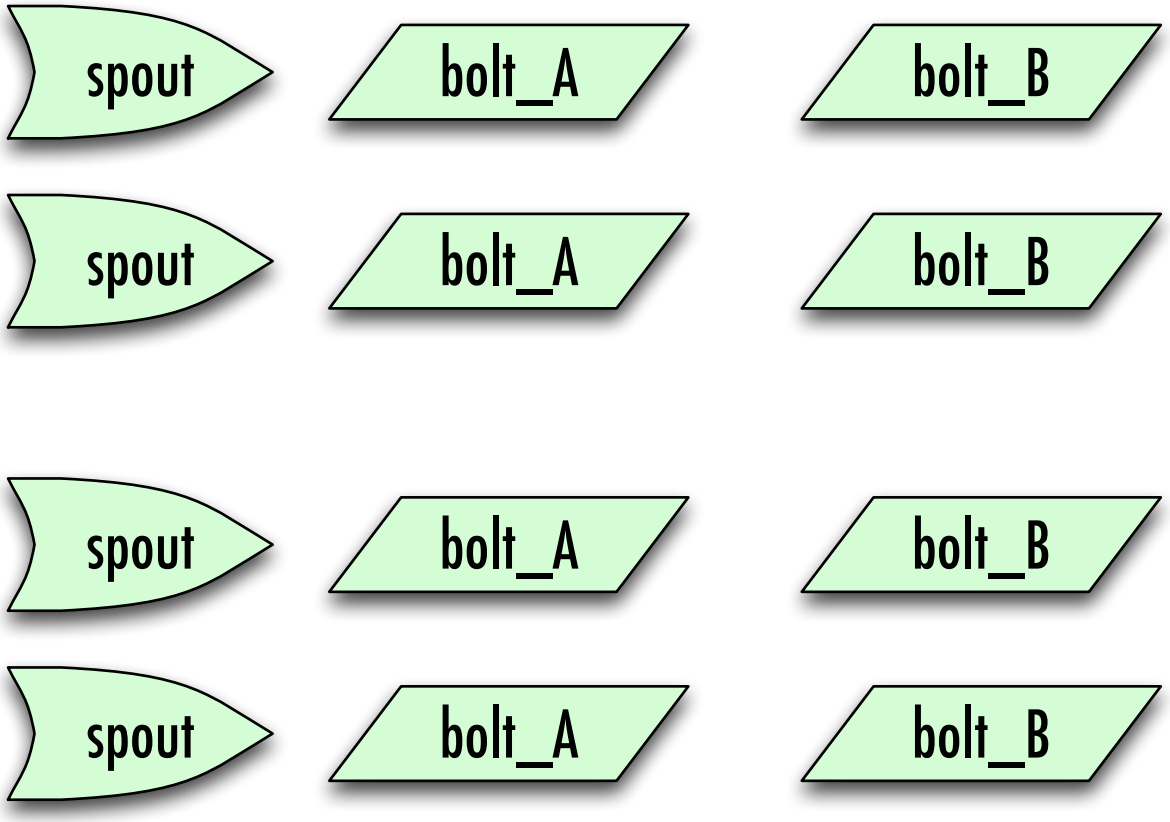
This means another attempt has been here: maybe it succeeded but was slow; maybe it failed; maybe _I_ am the one who is succeeding but slow. In any case, I don't know whether to trust the _new_ (first slot) values for this state, but I do know that I can trust the prior (second slot) values saved from batch 68. I just use those, and clobber the existing values with my new, correct counts.

Topology

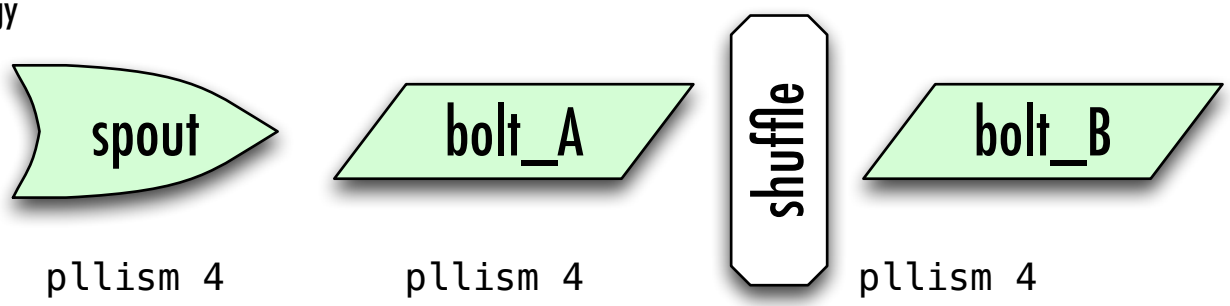


Physical

These will send directly from spouts to A's to B's

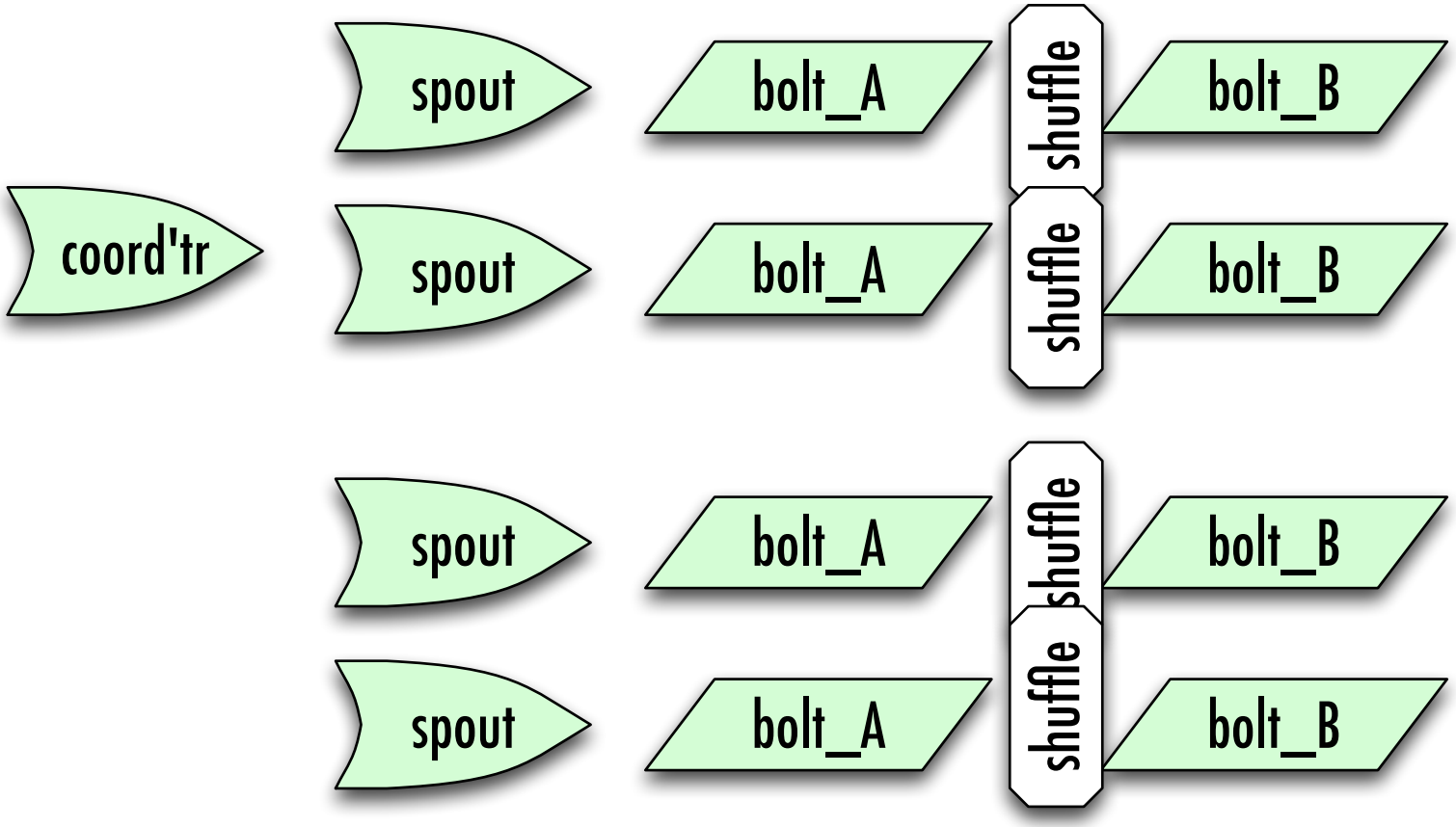


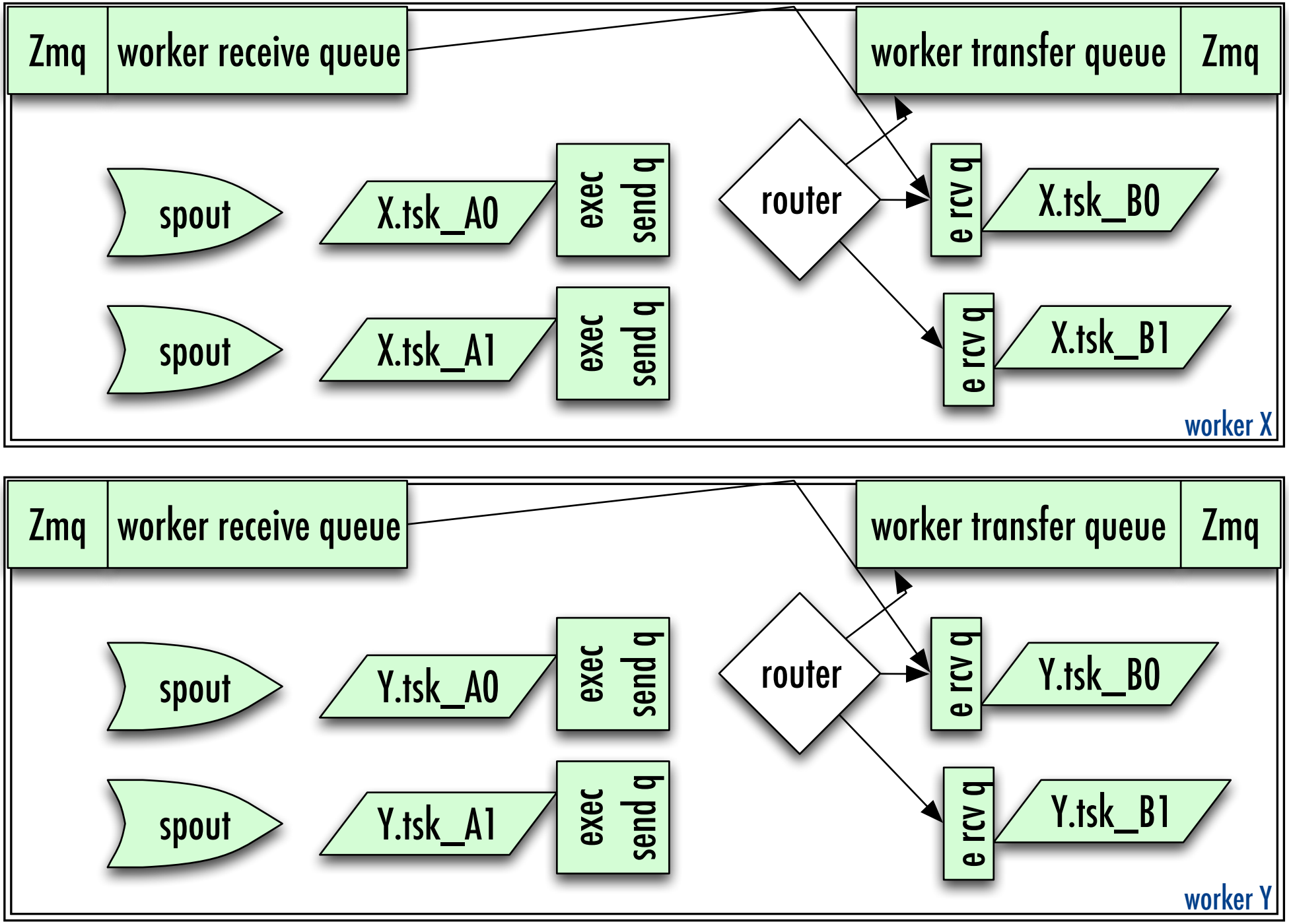
Topology

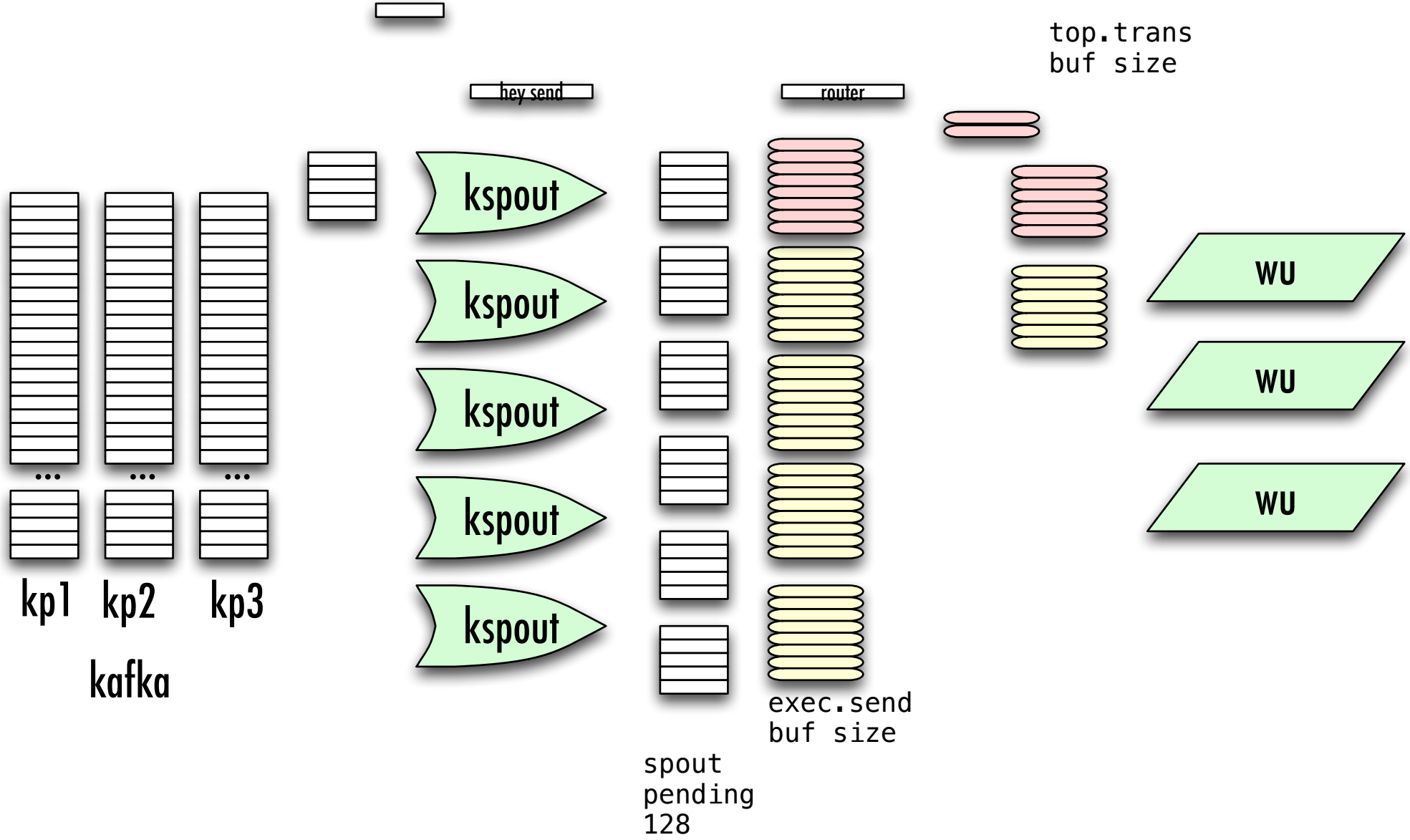


Physical

These will need the send end receive buffers

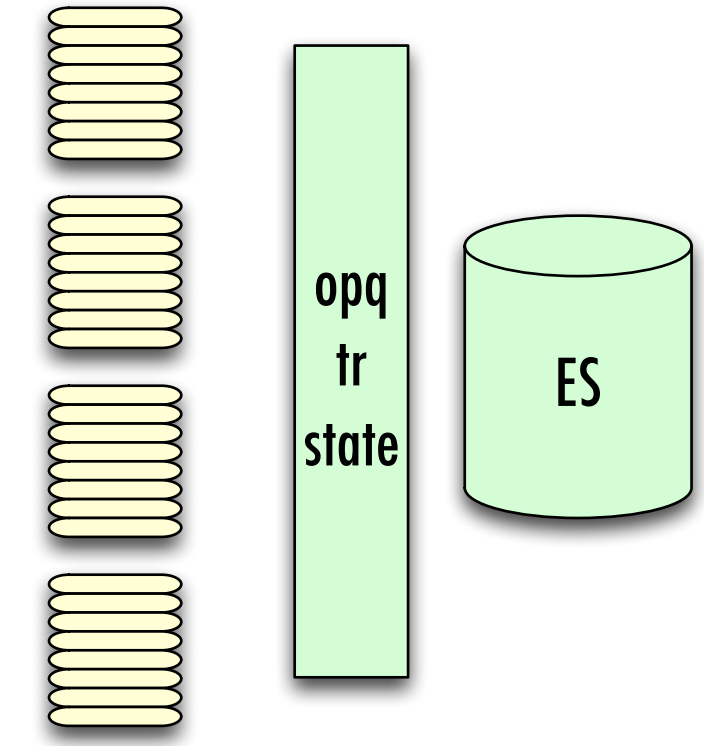



Zmq
worker receive queue
worker transfer queue
Zmq
spout
X.tsk_A0
exec
send q
router
e rcv q
X.tsk_B0
spout
X.tsk_A1
exec
send q
e rcv q
X.tsk_B1
worker X
Zmq
worker receive queue
worker transfer queue
Zmq
spout
Y.tsk_A0
exec
send q
router
e rcv q
Y.tsk_B0
spout
Y.tsk_A1
exec
send q
e rcv q
Y.tsk_B1
worker Y

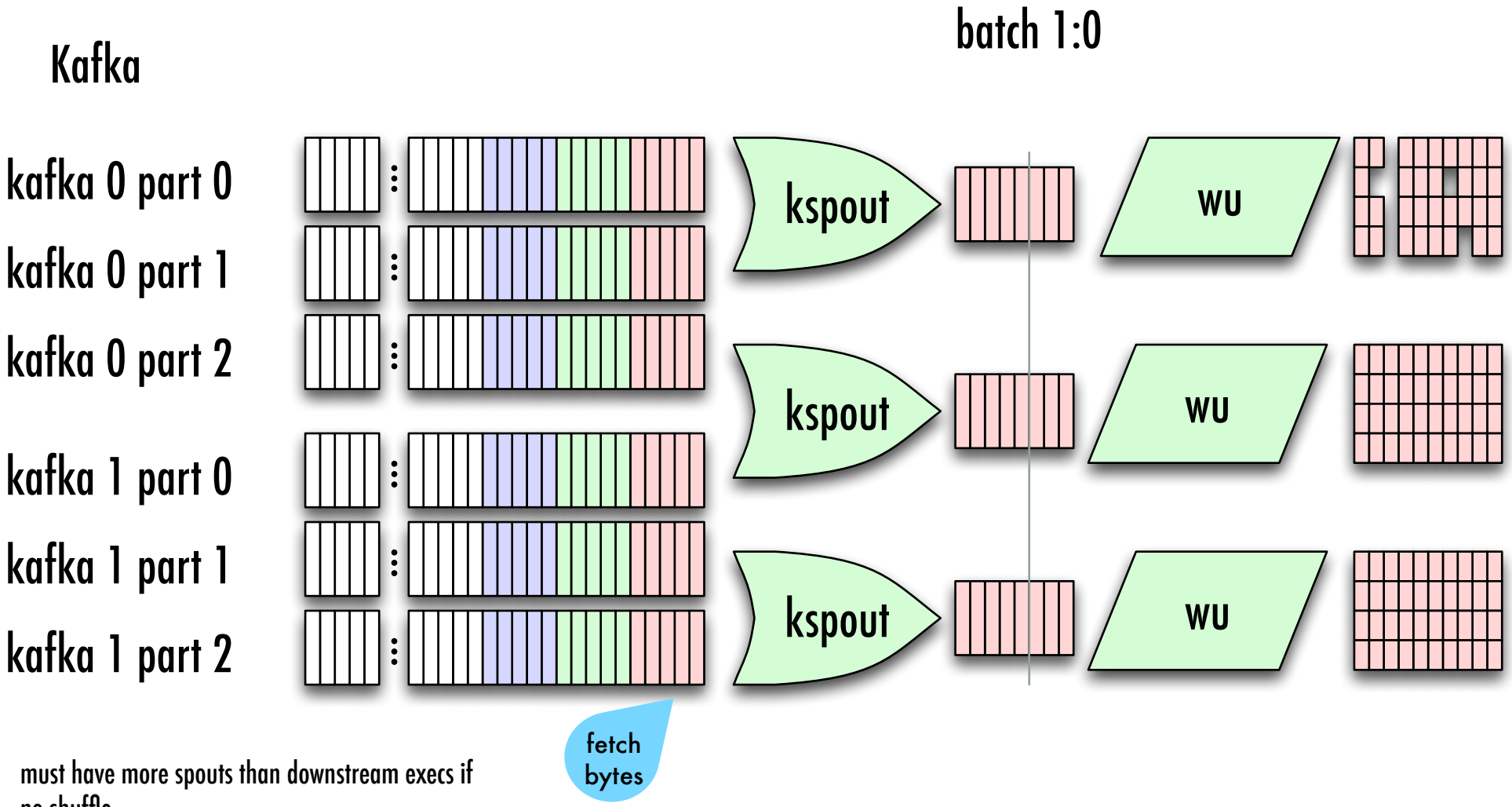


max_fetch_size_bytes: 125k
(about 100 records)

set spout||ism = kafka partitions



exec.send
buf size
top.trans
buf size



must have more spouts than downstream execs if no shuffle

* must have as many kfk partitions as spouts to keep them all fed

* tasks an even multiple of executors; executors an even multiple of workers; partitions an even multiple of spouts; workers an even multiple of machines