Hi

CS 744: SPARK STREAMING

Shivaram Venkataraman

Fall 2021

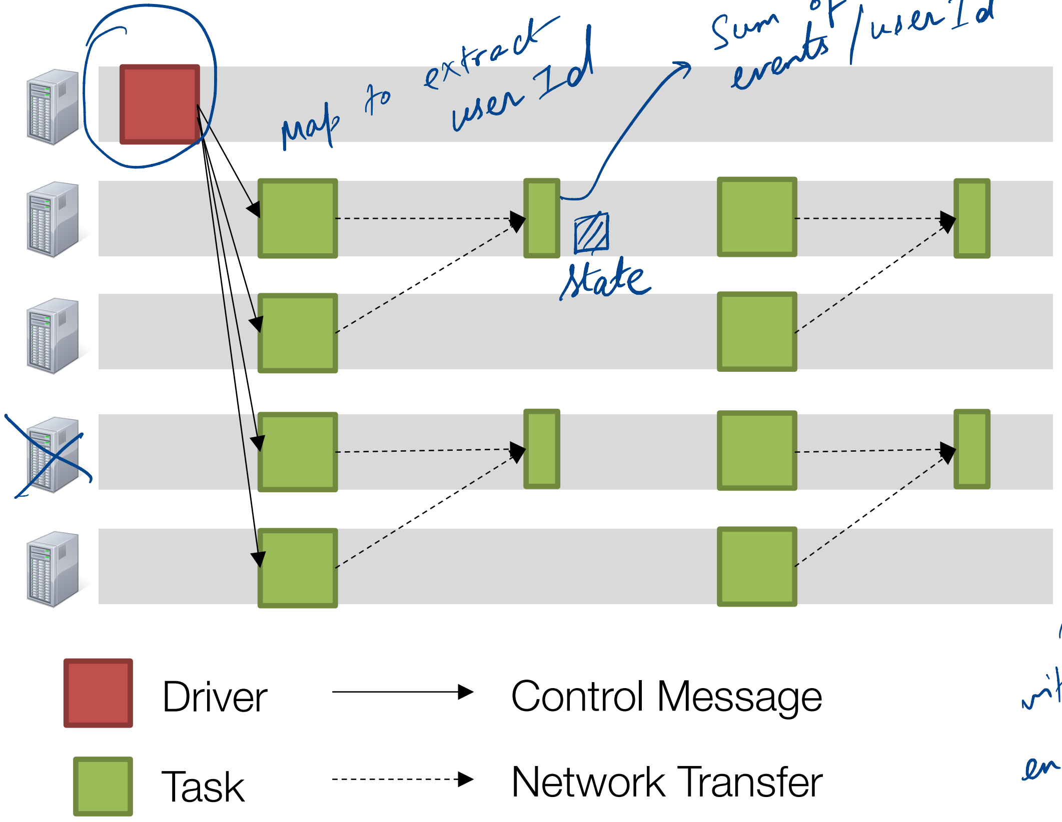
ADMINISTRIVIA

- Midterm grades this week → Friday
- Course Projects feedback → TODAY

- Google Cloud Credits
  ↳ $50 / student email address

Private Piazza / e-mail

CONTINUOUS OPERATOR MODEL



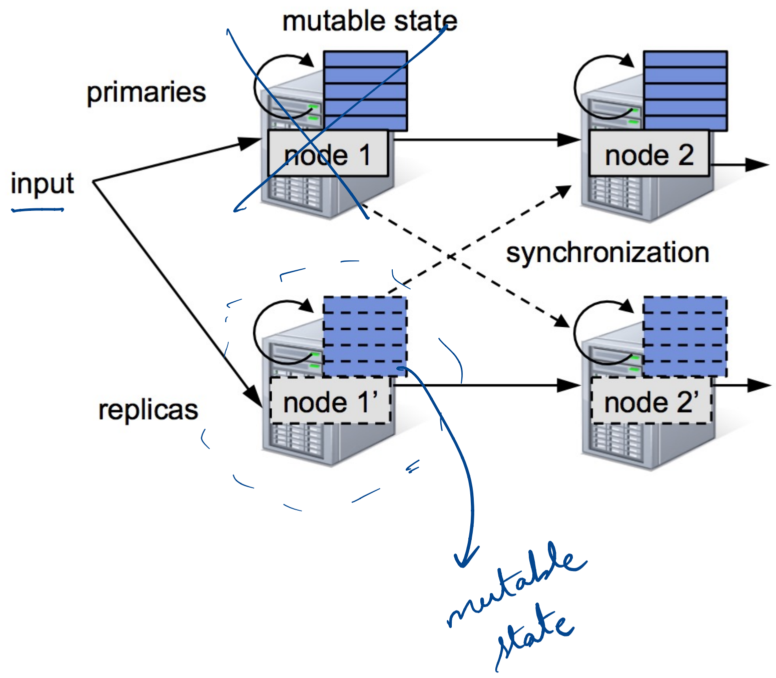
- Long-lived operators
- Mutable State
- Distributed Checkpoints for Fault Recovery
- Stragglers ?

Avoid stragglers with good engineering

Flink

Naiad

CONTINUOUS OPERATORS



- Replicate every operator to another machine
- Minimizes recovery time
- 2x the resources
- Replicas remain in-sync
    - replication protocol which also adds overhead

SPARK STREAMING: GOALS

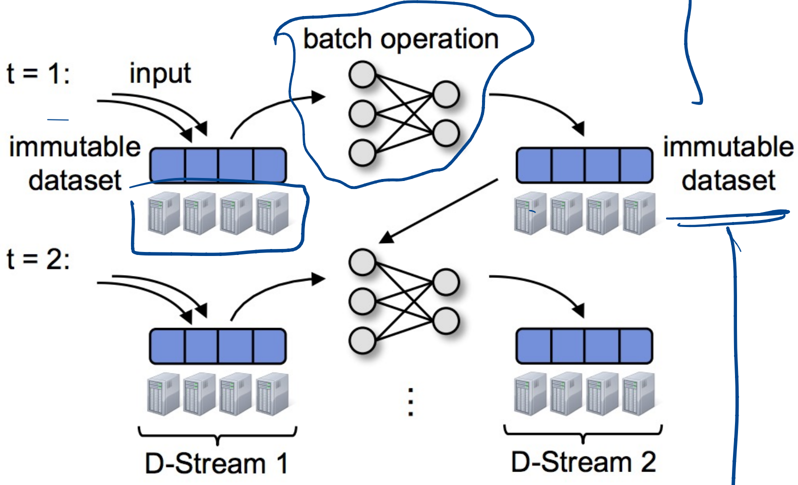
1. Scalability to hundreds of nodes → high throughput
2. Minimal cost beyond base processing (no replication) → resource efficiency
3. Second-scale latency = time between input arriving to when it is part of the output
4. Second-scale recovery from faults and stragglers

DISCRETIZED STREAMS (DSTREAMS)

- Divide time into a number of micro-batches
  - 1s as micro-batch
    t = 1s to 2s ← all events
- Run the batch operation on the input events, and save output (operator state) also as an RDD
- Next micro-batch, use the output from previous and compute

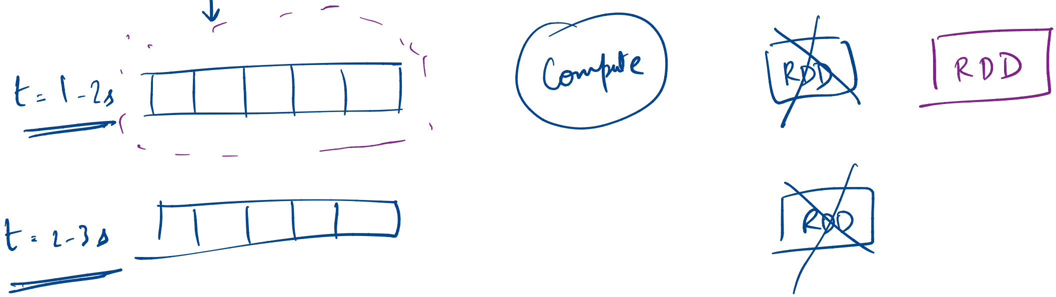
Memory / Disk

RDD



deterministic
→ recompute it, lineage

will this be the same



Processing time stamps / not event timestamps

EXAMPLE

DStream

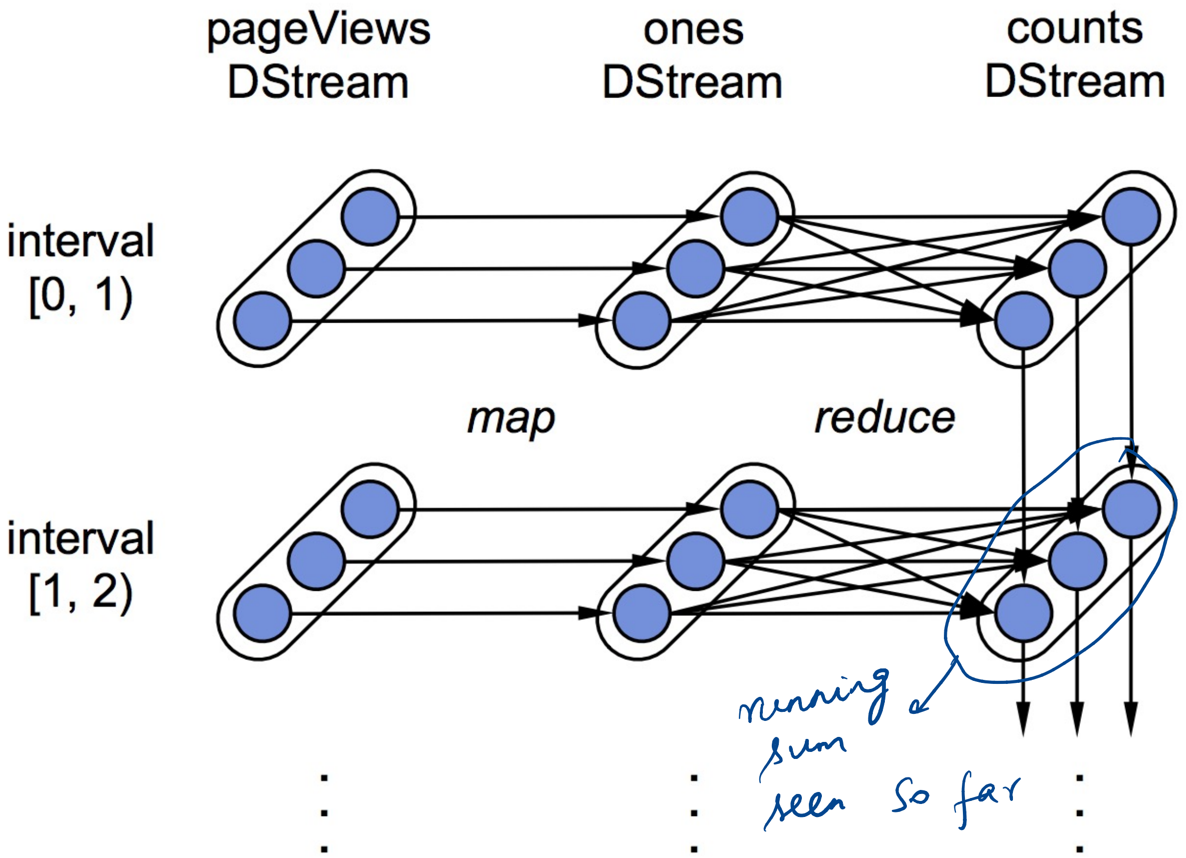
read a stream and not a file

```
pageViews =
  readStream(http://...,
             "1s")

ones = pageViews.map(
   event =>(event.url, 1))

counts =
    ones.runningReduce(
        (a, b) => a + b)
```

running sum of url events



DSTREAM API

Transformations

Stateless: map, reduce, groupBy, join → very similar to RDD API

Stateful:

sliding window("5s") → RDDs with data in [0,5), [1,6), [2,7)

reduceByWindow("5s", (a, b) => a + b)

↳ form a window of 5s and reduce using sum

SLIDING WINDOW

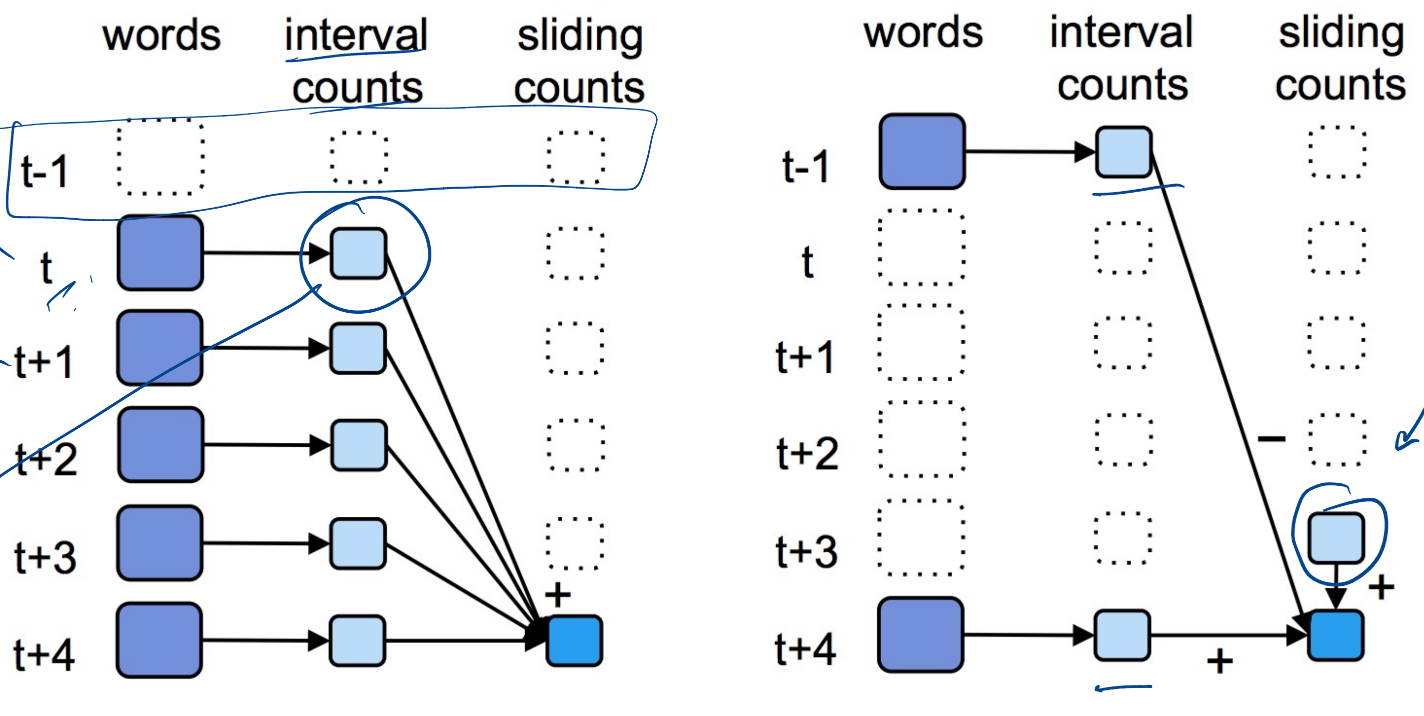
google.com, 102

google.com, 32

Computation is cheaper

Add previous 5 each time

hold on to this for 5s.



(a) Associative only (b) Associative & invertible

STATE MANAGEMENT

Tracking State: streams of (Key, Event) → (Key, State)

- User defined state object for every key

```
events.track(
  (key, ev) => 1,

  (key, st, ev) => ev == Exit ? null : 1,

  "30s")
```

Initialize state

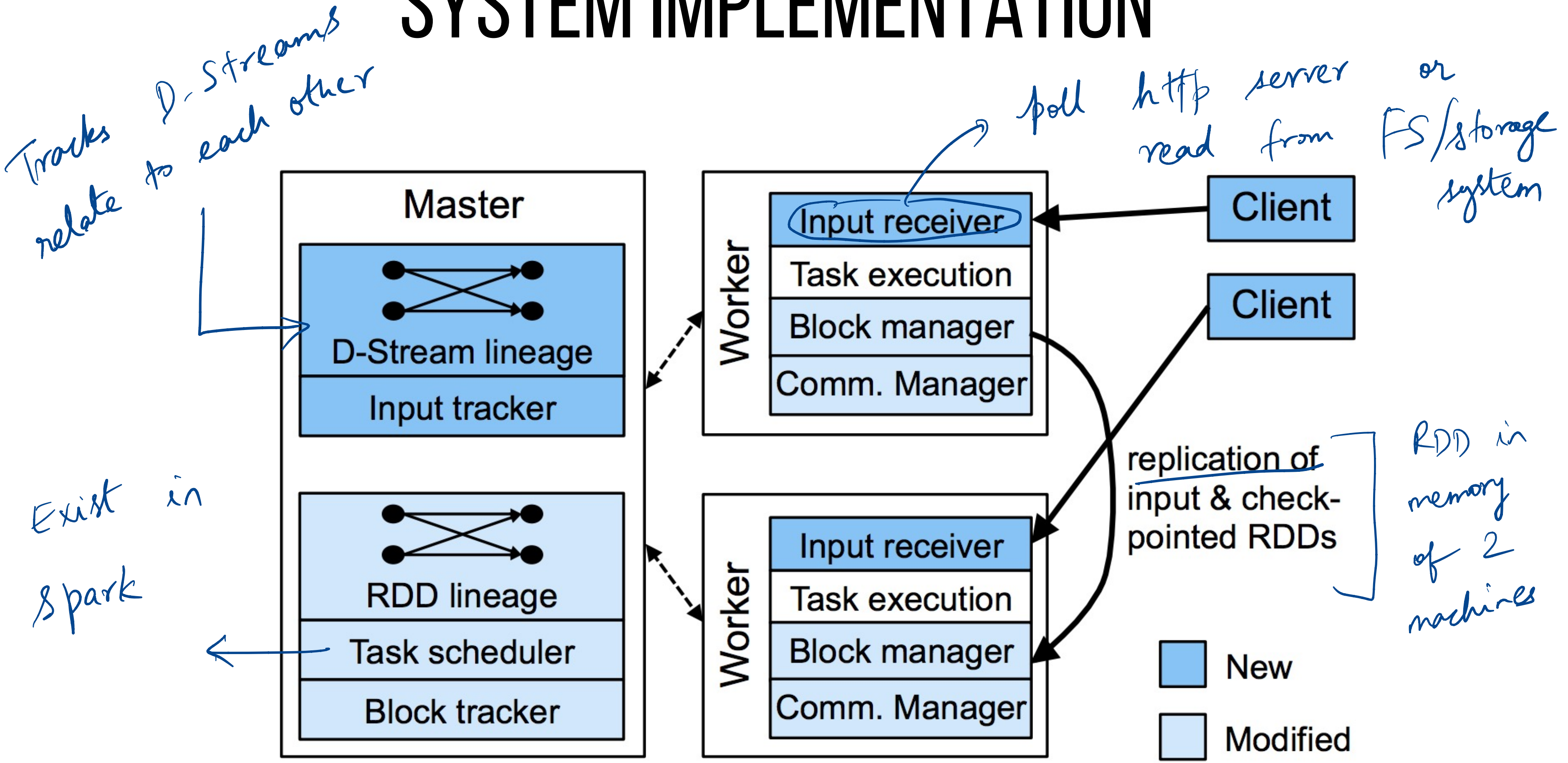
old state

new event

update my state based on event that arrived

Timeout to delete old states

SYSTEM IMPLEMENTATION



OPTIMIZATIONS

Timestep Pipelining

No barrier across timesteps unless needed

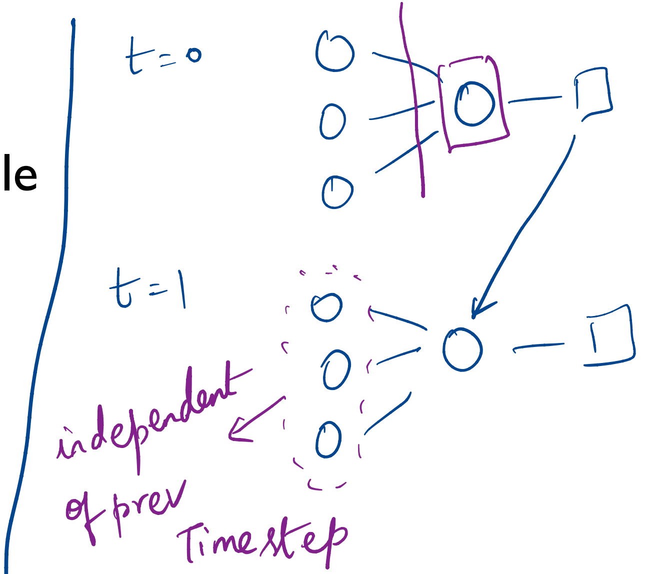
Tasks from the next timestep scheduled before current finishes

Checkpointing

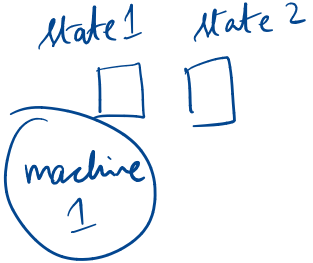
Async I/O, as RDDs are immutable

Forget lineage after checkpoint

stored in HDFS



FAULT TOLERANCE: PARALLEL RECOVERY



Worker failure

- Need to recompute state RDDs stored on worker
- Re-execute tasks running on the worker

Strategy

- Run all independent recovery tasks in parallel
- Parallelism from partitions *in timestep* and *across timesteps*

Fault recovery can run fast

Note
- Only recompute state / tasks which were lost

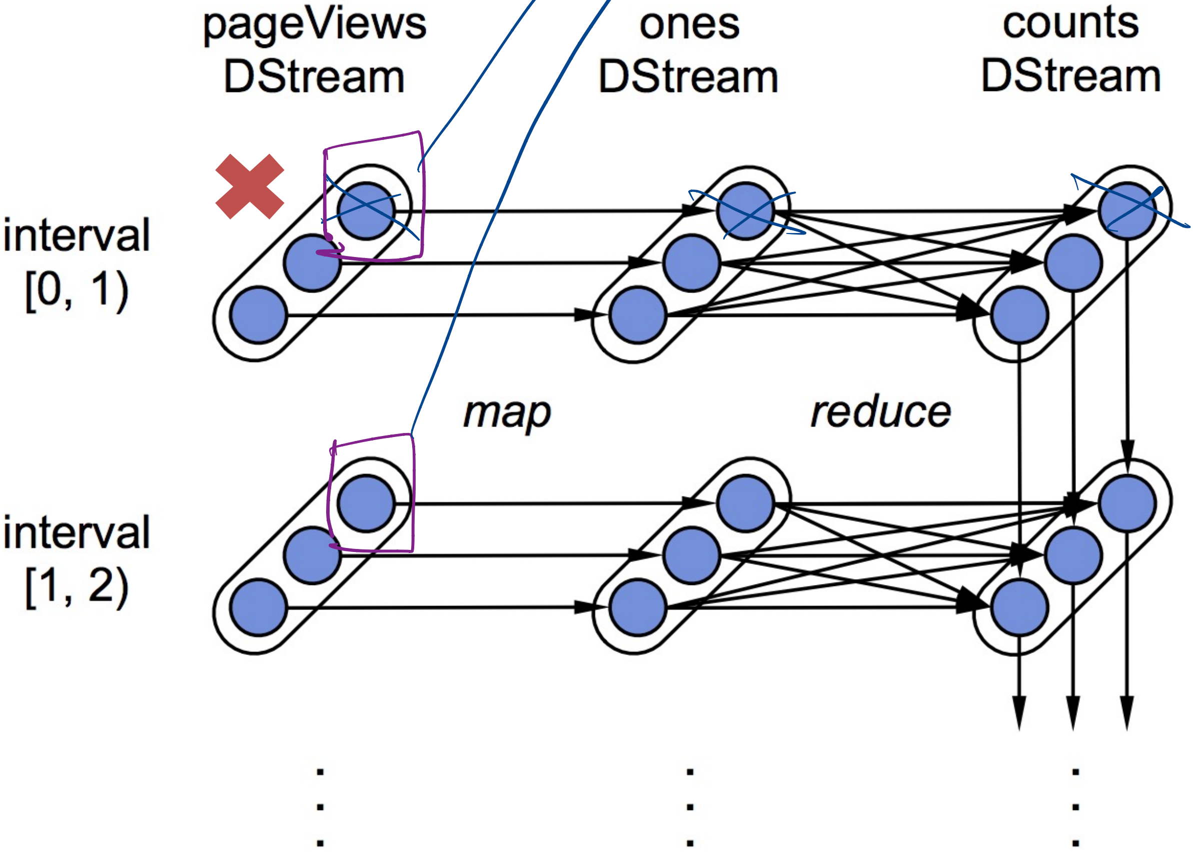
EXAMPLE

```
pageViews =
  readStream(http://...,
             "1s")

ones = pageViews.map(
   event =>(event.url, 1))

counts =
    ones.runningReduce(
        (a, b) => a + b)
```

can be run in parallel



FAULT TOLERANCE

Straggler Mitigation

Use speculative execution

Task runs more than 1.4x longer than median task → straggler

Master Recovery → computation runs 24 x 7

- At each timestep, save graph of DStreams and Scala function objects
- Workers connect to a new master and report their RDD partitions
- Note: No problem if a given RDD is computed twice (determinism).

→ similar in spirit to GFS recovery

SUMMARY

Micro-batches: New approach to stream processing

Simplifies fault tolerance, straggler mitigation

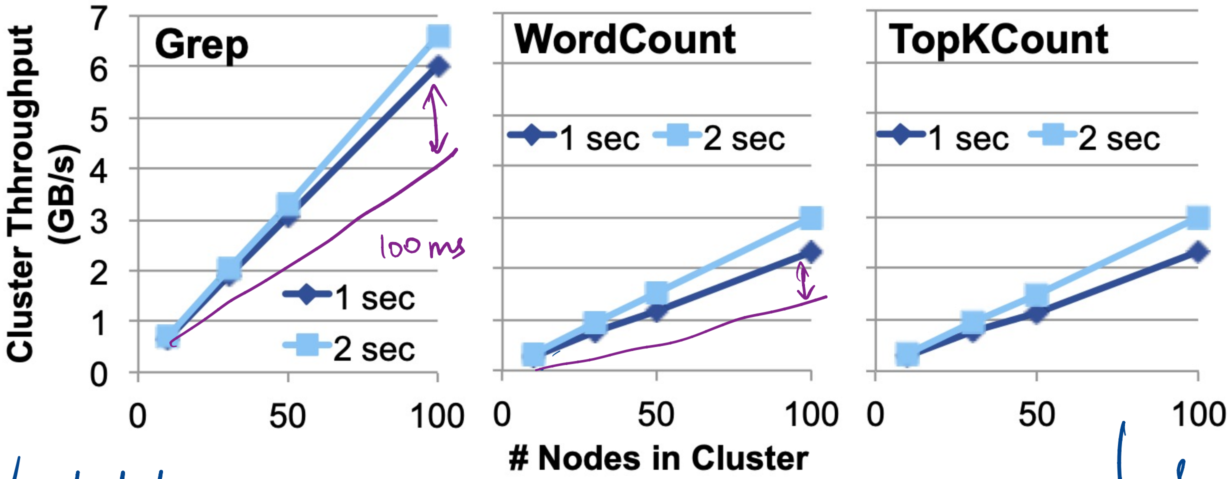
Unifying batch, streaming analytics

DISCUSSION

https://forms.gle/4Xbu9y9KTW5qph8H8

If the latency bound was made to 100ms, how do you think the above figure would change? What could be the reasons for it?

Overhead increases with cluster size



Every microbatch

= Fixed overhead + Time to do Grep / Word Count

2s ~

100ms Bigger factor

Small microbatch

⇒ lots of RDDs

↳ more metadata

↳ tracking / scheduling tasks

Consider the pros and cons of approaches in Naiad vs Spark Streaming. What application properties would you use to decide which system to choose?

Naiad / Flink

- low latency / quick processing

Spark Streaming

- fault recovery time

NEXT STEPS

Next class: Graph processing!

Midterm grades ASAP!