Good morning

CS 744: SPARK STREAMING

Shivaram Venkataraman

Fall 2020

ADMINISTRIVIA

- Midterm grades this week → ASAP
- Course Projects feedback → HotCRP

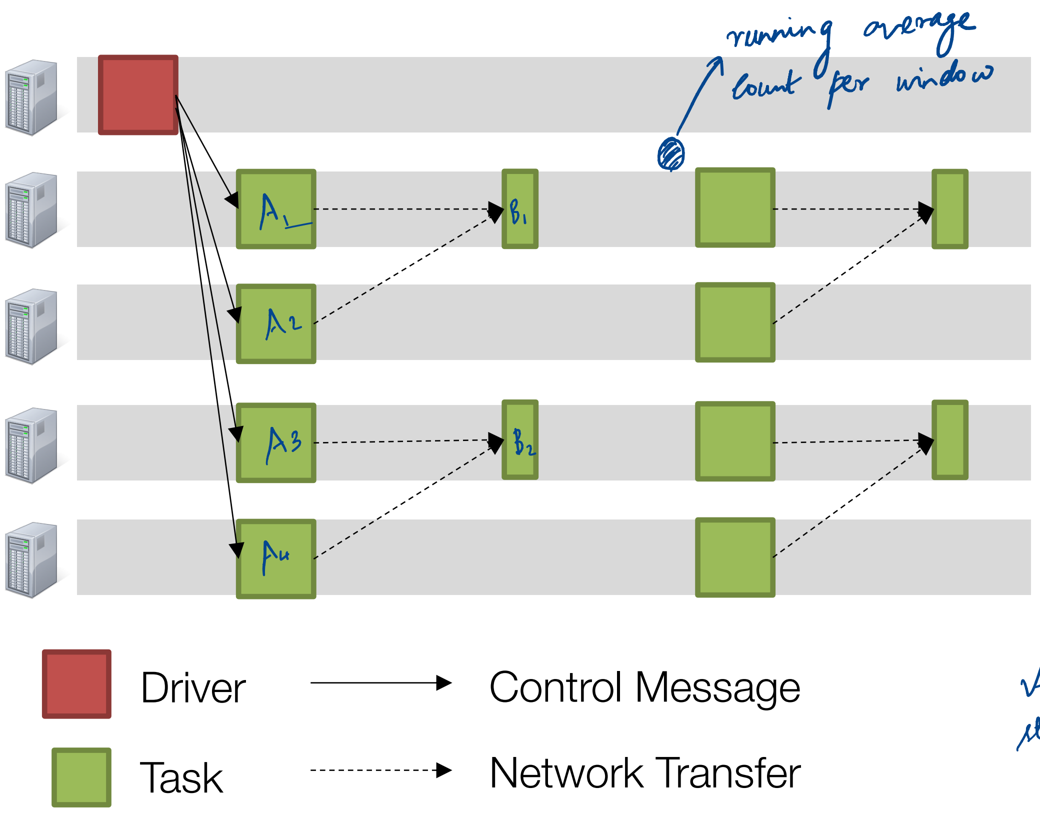
Hopefully you are working on this!

(1) Assign grades for project proposals

(2) mid semester update → Nov 20th

CONTINUOUS OPERATOR MODEL

A | High | B



Long-lived operators

Mutable State

Distributed Checkpoints for Fault Recovery

Roll back all operators to checkpoint

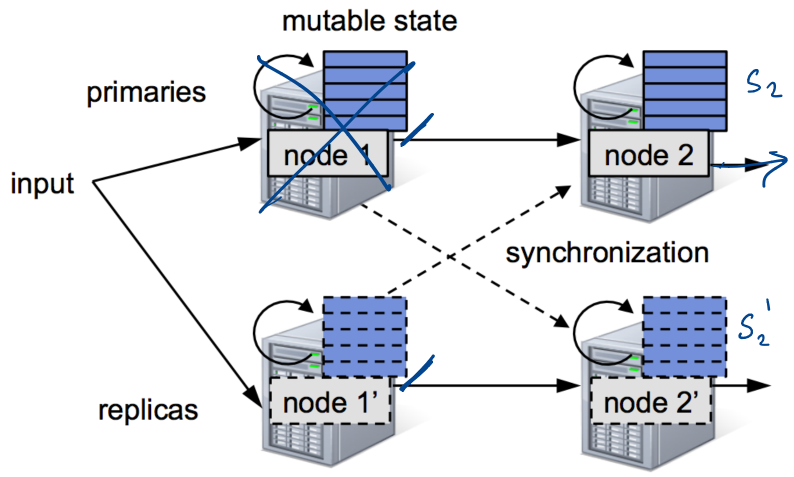
Stragglers ?

Avoid stragglers

Flink

Naiad

CONTINUOUS OPERATORS



Replication to provide fault tolerance

⇒ Multiple copies (say 2?) of each operator

(1) Overhead of 2x resources required

(2) Replicas need to be in sync

⇒ S_2 and S_2' should be the same

⇒ need to make sure replicas are synchronized during normal computation

↓

overhead!

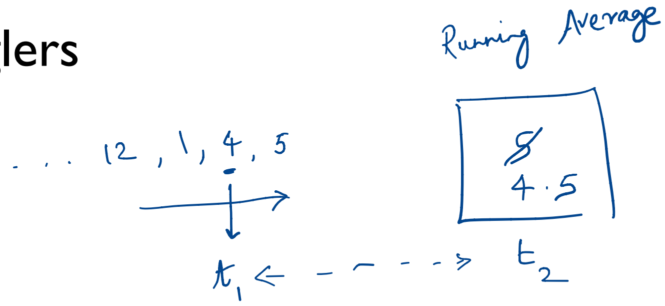
SPARK STREAMING: GOALS

1. Scalability to hundreds of nodes → To handle high tput streams

2. Minimal cost beyond base processing (no replication)

3. Second-scale latency → time between when event arrives to time when event is reflected in the output

4. Second-scale recovery from faults and stragglers



$t_2 - t_1$ = latency

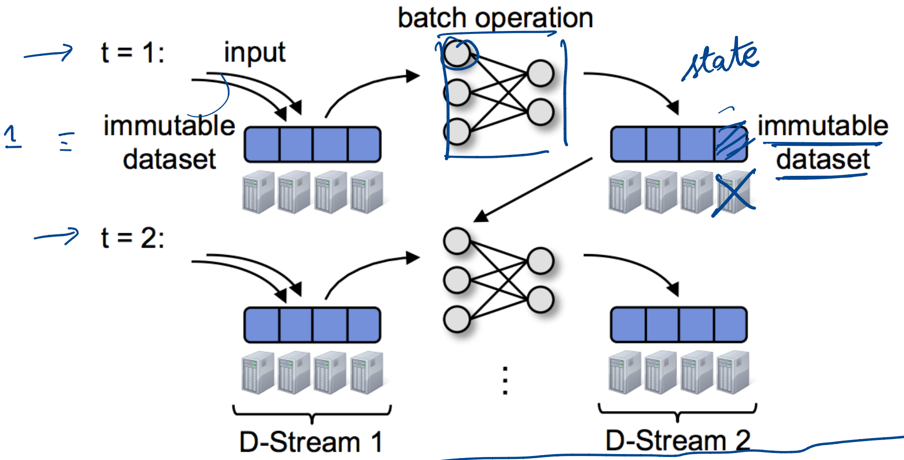
DISCRETIZED STREAMS (DSTREAMS)

- every micro batch
  run short, deterministic tasks
  to compute incremental output

micro batch duration = 1 s → overheads in batch computation

- every batch operation is stateless
  state is stored as immutable dataset

input 1 =



- each part of the state can be recovered independently
  → Use lineage to do recovery

non-deterministic is opposite
→ if you re-run output might be difficult

Task
random < 5 : output 0 else output 1
can be made deterministic

EXAMPLE

[google.com, 5
yahoo.com, 12
⋮]

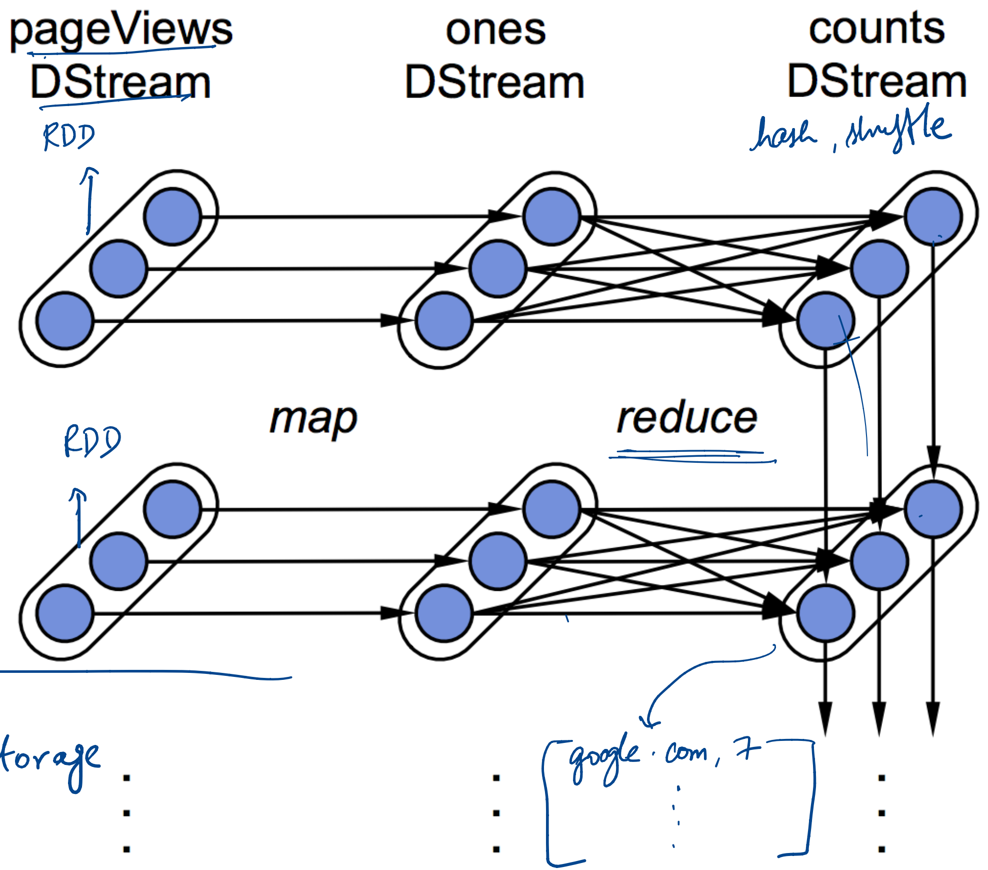
```
pageViews =
  readStream(http://...,
             "1s")

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

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

File system, or from HTTP etc.

micro batch duration



partitioned from input source
↳ read one chunk from storage.

DSTREAM API

Transformations

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

Stateful:

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

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

↓

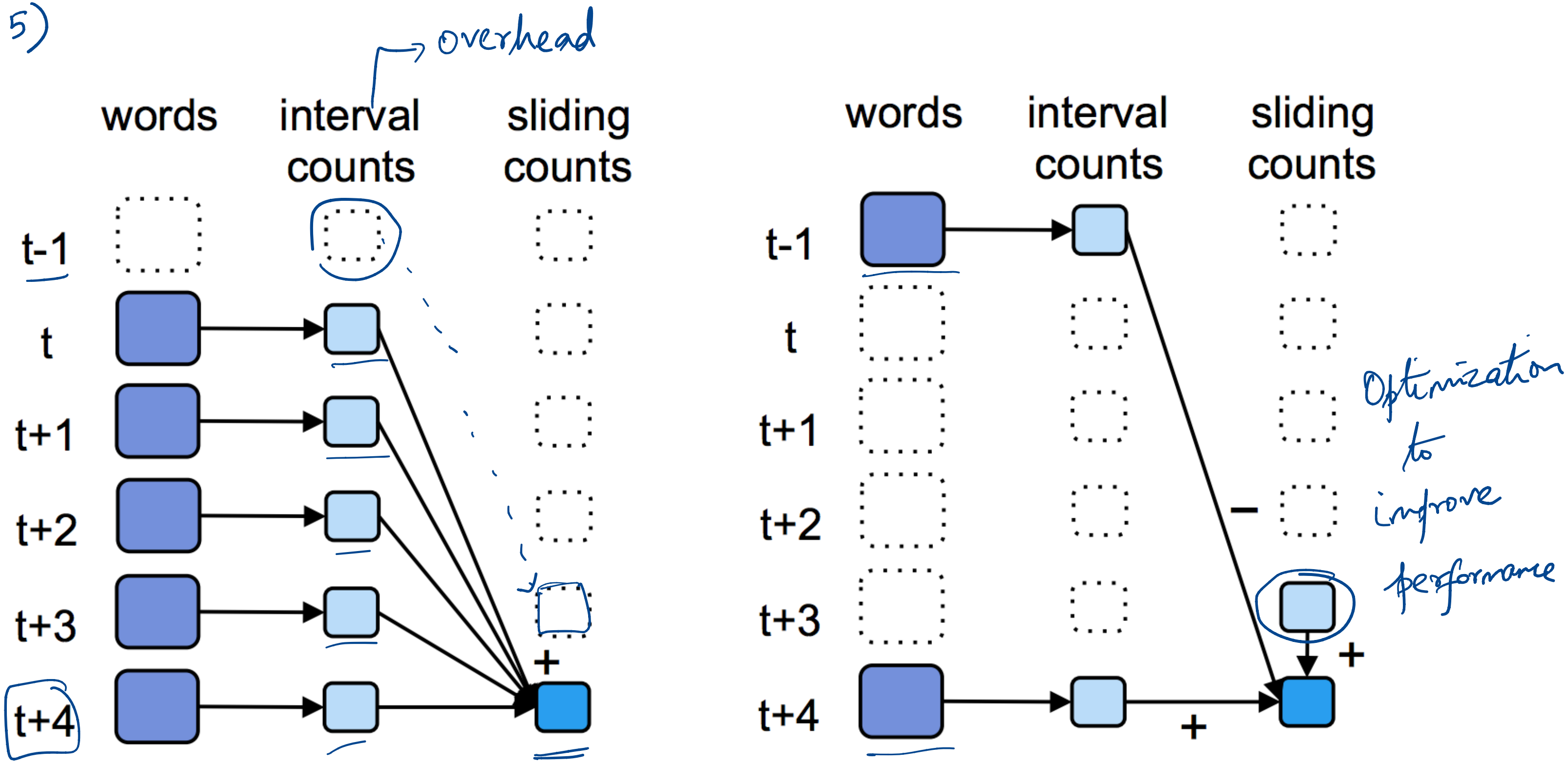
creates a sliding window and aggregates RDDs that belong to it.

micro batch duration = 1s
window duration = 5s
[t, t+5)

SLIDING WINDOW

Add previous 5 each time

reduce By Window



(a) Associative only (b) Associative & invertible

STATE MANAGEMENT

Tracking State: streams of (Key, Event) → (Key, State) → Session which has all events for a user satisfying some criteria [login → logout]

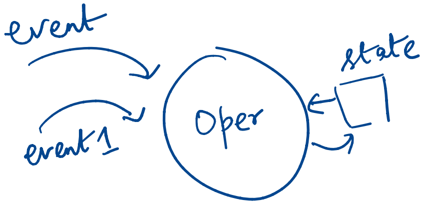
```
events.track(
  (key, ev) => 1,
  (key, st, ev) => ev == Exit ? null : 1,
  "30s")
```

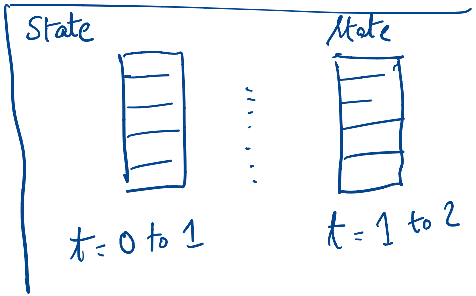
user-id

→ Initialize state

Update: given prev. state and a new event return new state

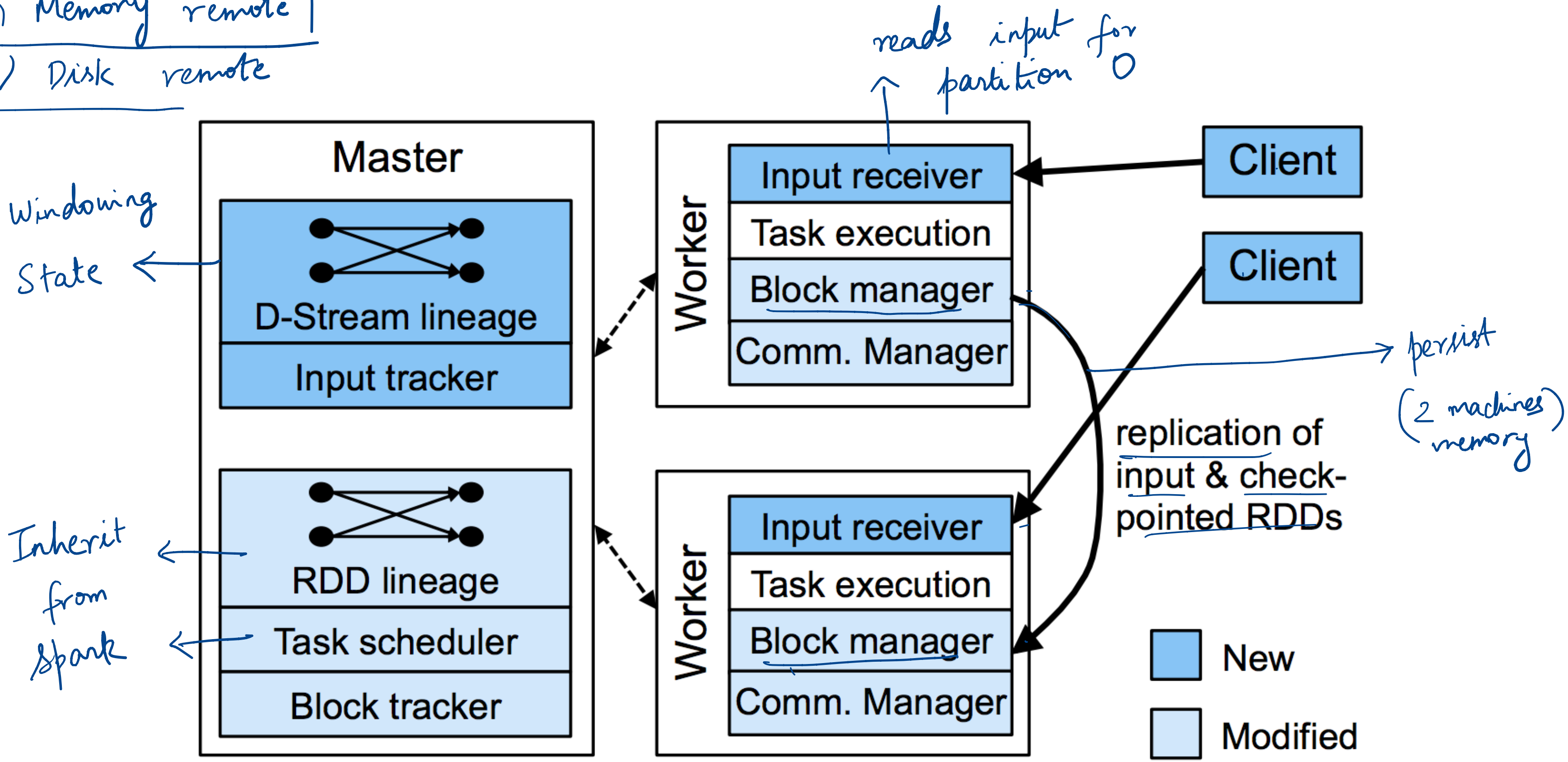
Timeout: forget old states





SYSTEM IMPLEMENTATION

Persist data safely
(1) Disk locally
(2) Memory remote
(3) Disk remote



OPTIMIZATIONS

Map

(0,1)

(1,2) schedule before prev finishes

Timestep Pipelining → fuse together map operations

- 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

can be done by storing to remote memory

FAULT TOLERANCE: PARALLEL RECOVERY

Worker failure

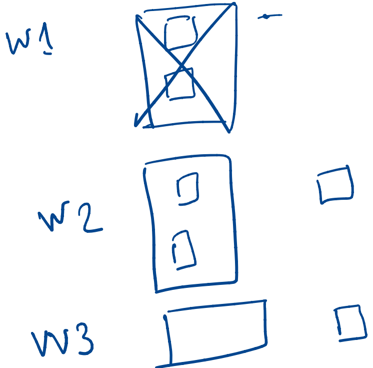
- Need to recompute state RDDs stored on worker

these might be used for future outputs

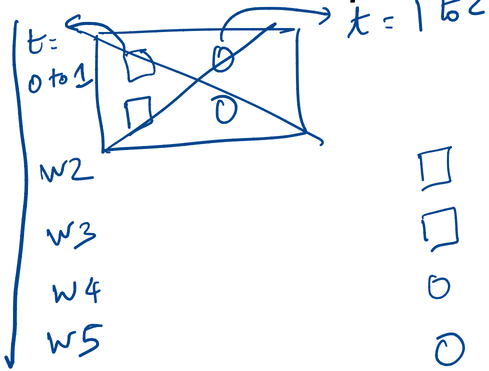
- Re-execute tasks running on the worker

Strategy

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



parallelism across partitions!



pending any dependencies!

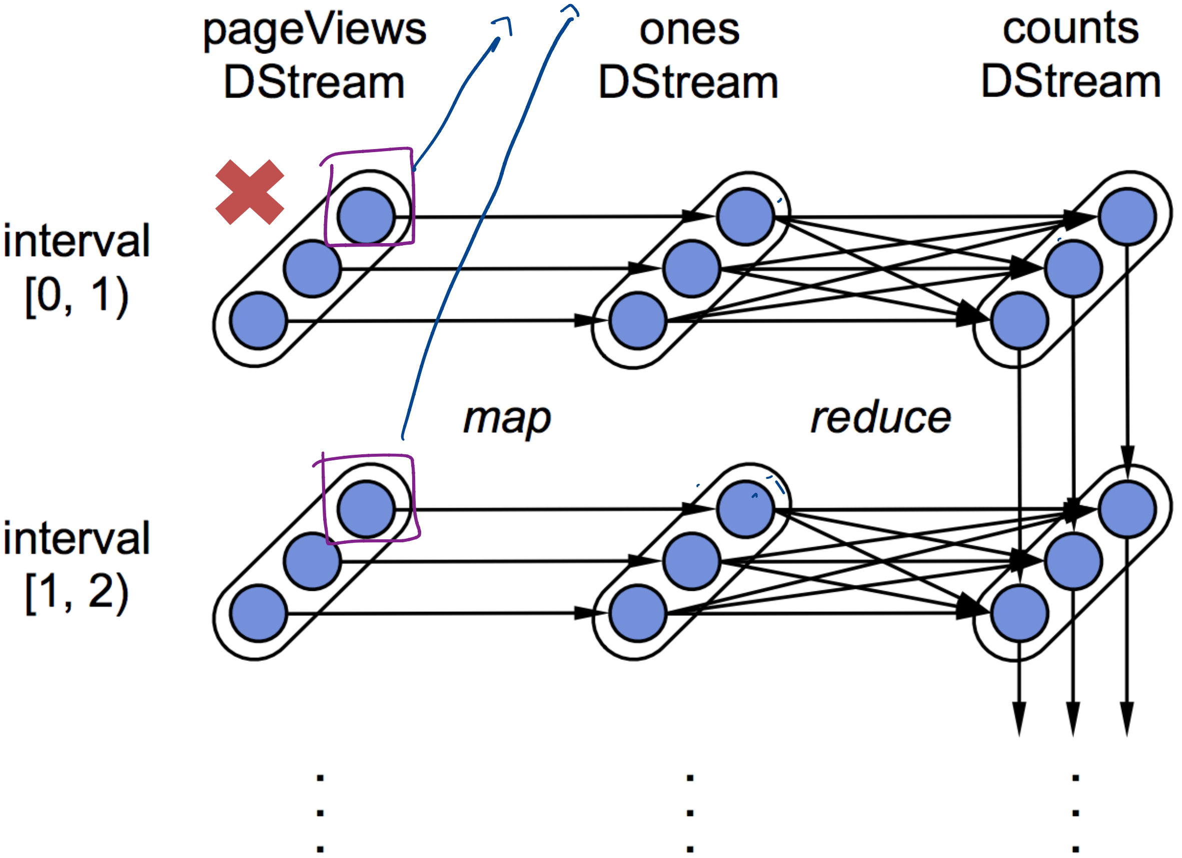
EXAMPLE

These are independent
Can be recovered in parallel.

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

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

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



FAULT TOLERANCE

Straggler Mitigation

Use speculative execution → fall back

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

Driver

~~Master~~ Recovery → Runs forever

MR Master → retry the job on failure

- 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).

→ GFS master recovery is similar!

SUMMARY

Micro-batches: New approach to stream processing

Simplifies fault tolerance, straggler mitigation

Unifying batch, streaming analytics

DISCUSSION

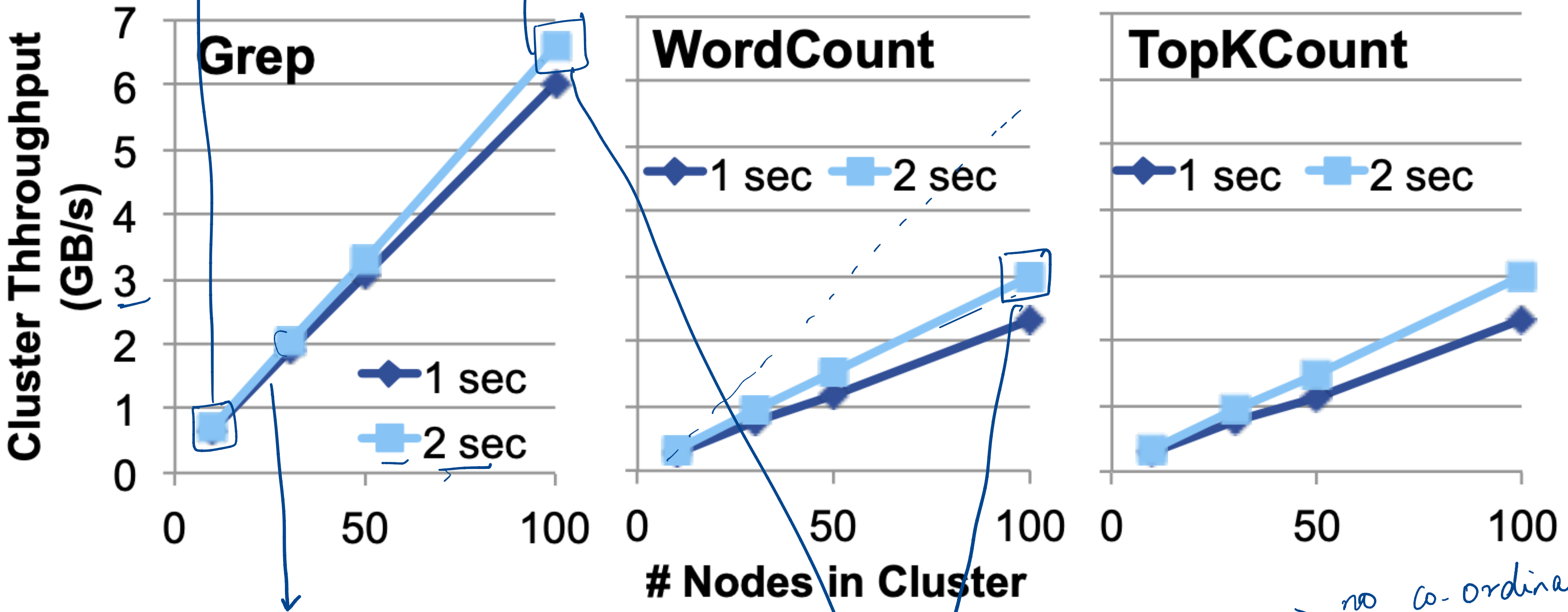
https://forms.gle/eiqbjJTU95bMQLtm9

$y = x$ vs. $y = 0.5x$

slope indicates overhead ⇒ more machines

higher throughput for larger mini batch size

overhead per mini-batch



linear growth with cluster size

more tput for grep!

→ no co-ordination for grep!

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

too low latency → low tput

overheads in task scheduling

tracking RDDs etc.

if we go to 1000 machines ⇒ overheads could be large!

↓

linear scaling might not last?

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

Naid

latency sensitive

iterative + streaming workflows

Spark Streaming

failures

stragglers

NEXT STEPS

Next class: Graph processing!

Midterm grades ASAP!

Lazy execution

↳ 1 sec (micro batch time)

trigger "forces computation"

Batching ?!

↳ Continuous operator

1 event := not optimal

$A \xrightarrow{<10 \text{ events}>} B$

send

↳ batching

Very low latency

→ MPI - based

→ C++ Actor model

↳ Erlang → Telephone companies