Hello!

CS 744: SPARK STREAMING

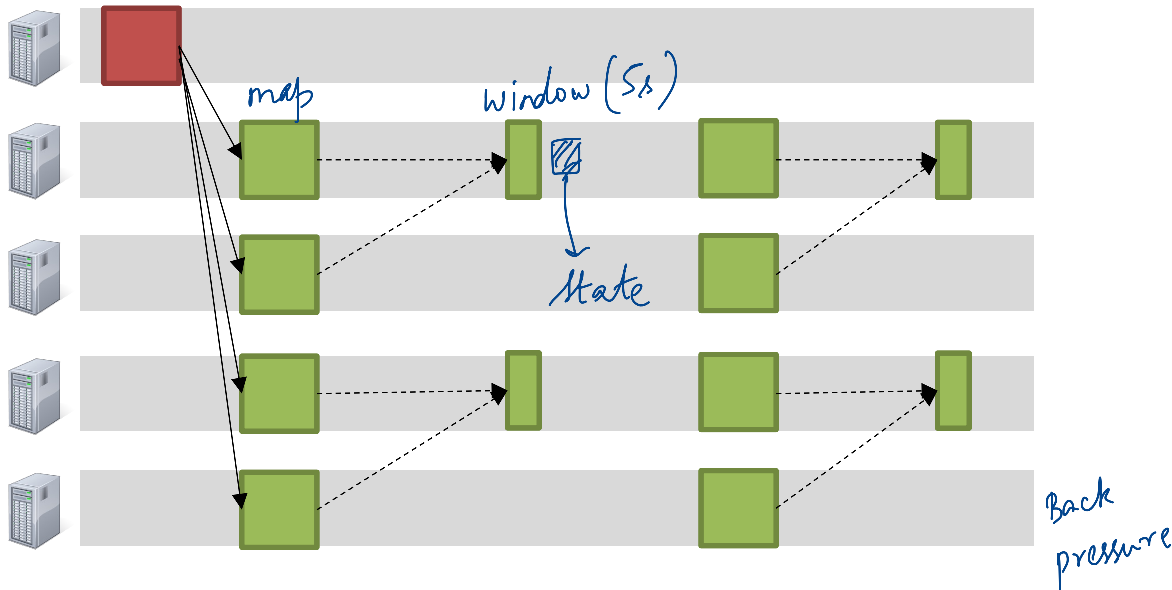
Shivaram Venkataraman

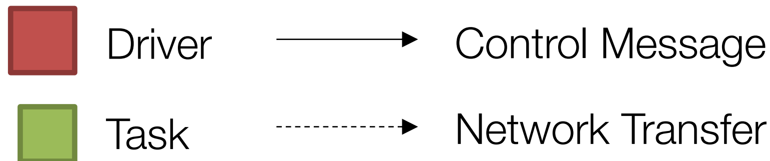
Fall 2022

ADMINISTRIVIA

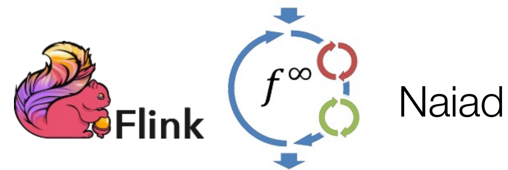
- Course Projects feedback → Mid-semester Check-In towards end of Nov
- Assignment2 grades → Roger for regrades
- Midterm grades – this week?

CONTINUOUS OPERATOR MODEL

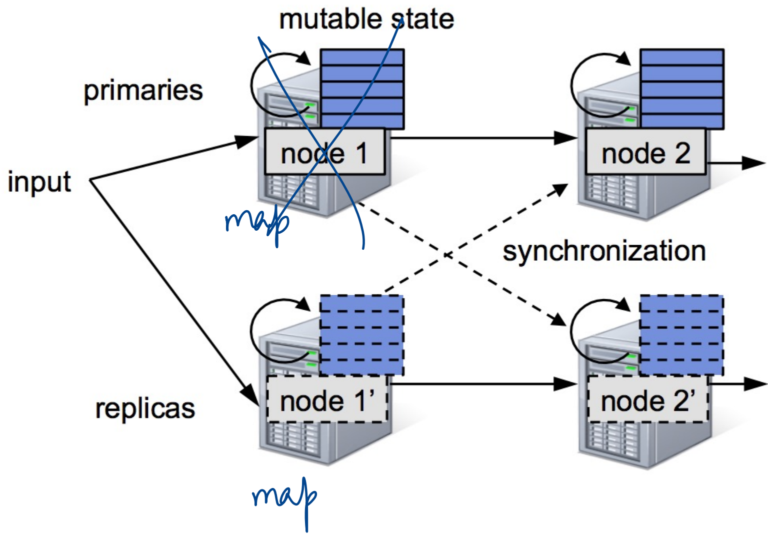




- Long-lived operators
- Mutable State
- Distributed Checkpoints for Fault Recovery
- Stragglers ?



CONTINUOUS OPERATORS



Replicate operators

↳ each operator is replicated to more than 1 machine

[Resource consumption is higher]

Replicas should be in-sync

- Fast recovery

SPARK STREAMING: GOALS

1. Scalability to hundreds of nodes → high throughput
2. Minimal cost beyond base processing (no replication) → esp. at large scale
3. Second-scale latency → Streaming context: time between input arriving and being reflected
4. Second-scale recovery from faults and stragglers

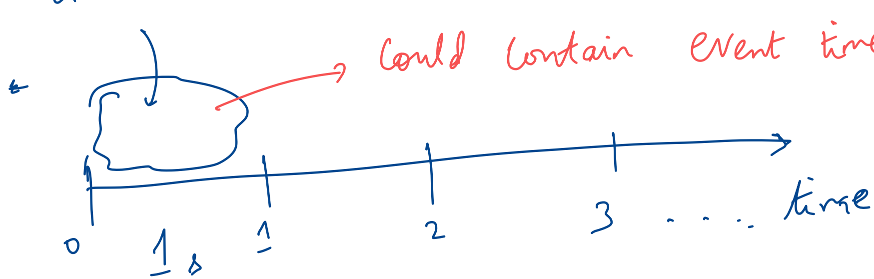
Normal operation

recover from failures fast

DISCRETIZED STREAMS (DSTREAMS)

Gather data

Could contain event times different



- Divided time into intervals
  - at every 1 s
- Query is run with stateless tasks
  - Output (and state) is saved
-

Similar to Spark / MapReduce

t = 1:
input
batch operation
output
immutable dataset
immutable dataset
t = 2:
D-Stream 1
D-Stream 2

read in state / output from prev. batch

API : easy to write queries

EXAMPLE

DStream: Discretized Stream

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

URL, Kafka, etc.

batch size

create DStream

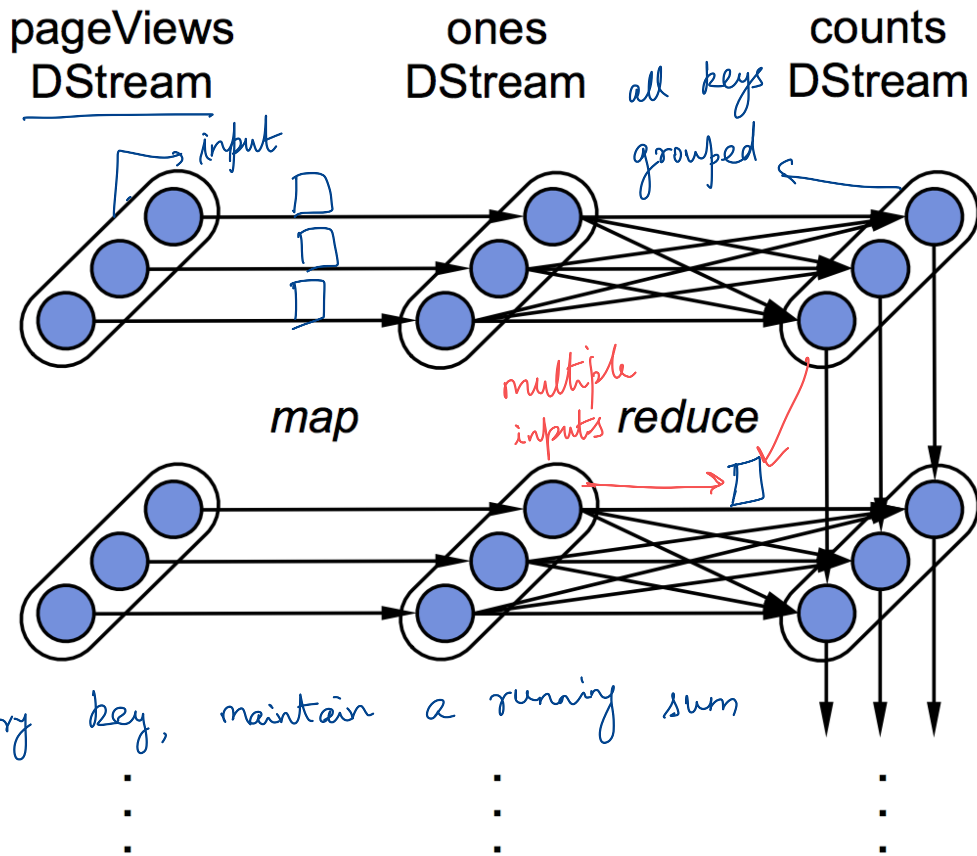
```
ones = pageViews.map(
   event =>(event.url, 1))
```

very similar map

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

for every key, maintain a running sum

Query



Running Reduce

URL Counts

t = [0, 1)

URL	Counts
a	2
b	3
⋮	

state : RDD or data structure stored in memory / disk

⋮

t = [1, 2)

new tuples

a	5
b	6

state . . .

⋮

t = 100

what state do you retain ?

DSTREAM API

Transformations

Similar to Flink

Stateless: map, reduce, groupBy, join

↳ Spark

Stateful:

Sliding window(“5s”) → RDDs with data in [0,5), [1,6), [2,7)

reduceByWindow(“5s”, (a, b) => a + b)

→ aggregates values for each key

↓ Creates a sliding window

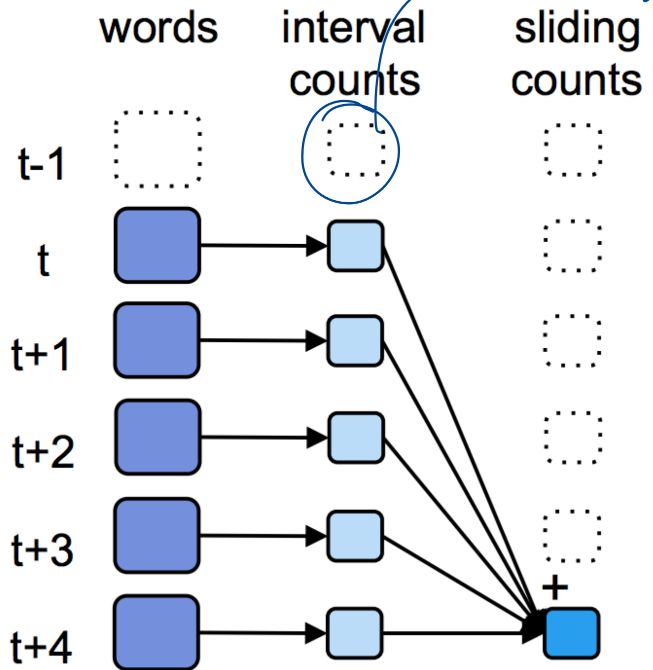
SLIDING WINDOW

batch size: 1s → on processing time

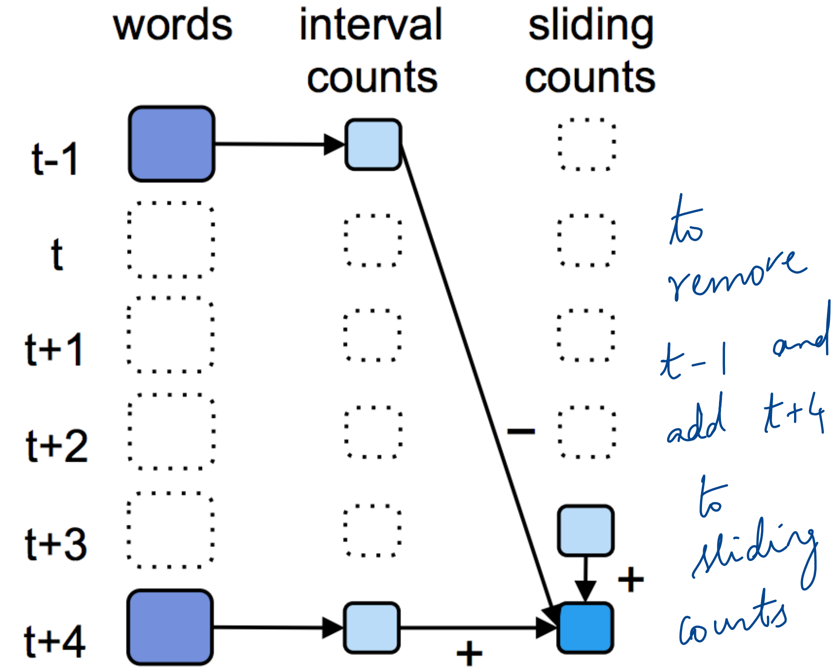
window size: 5s

no longer used → garbage collected

Add previous 5 each time



(a) Associative only



(b) Associative & invertible

to remove t-1 and add t+4 to sliding counts

STATE MANAGEMENT

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

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

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

  "30s")
```

User-defined class

event timestamps can be tracked inside users query

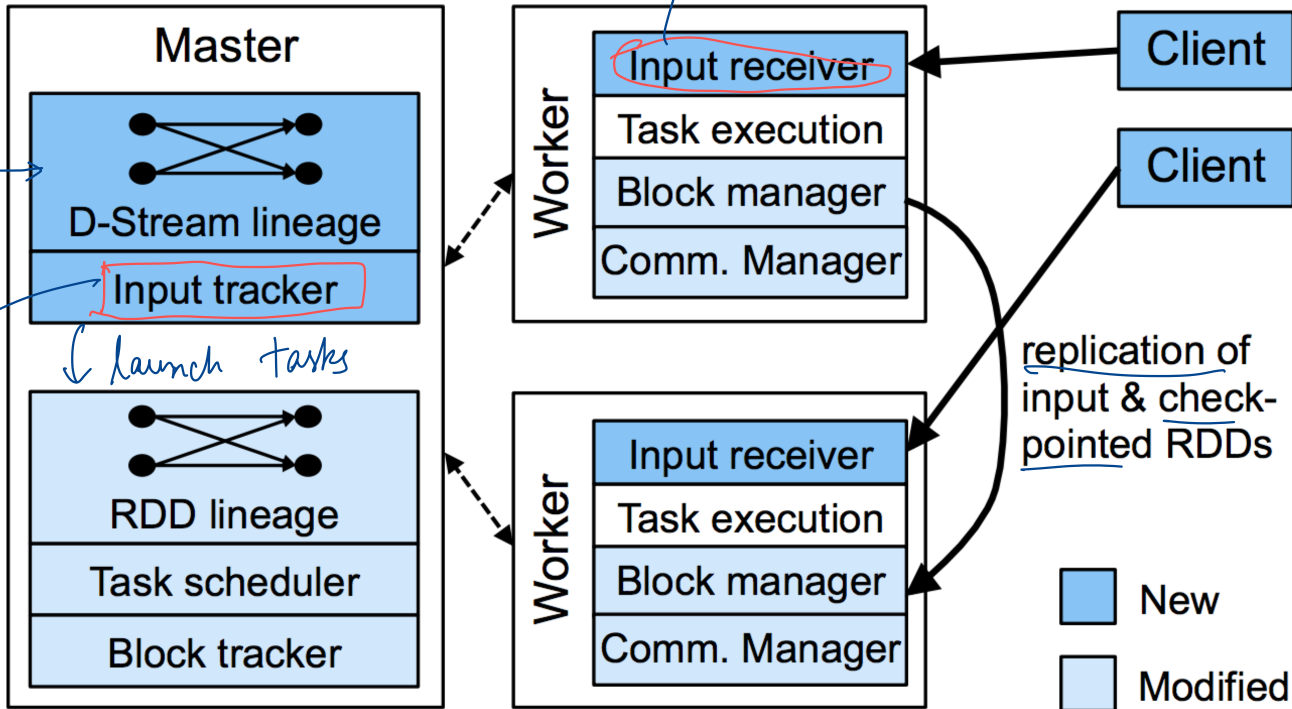
Initialize state

old state, event → new state

Time out to forget states

SYSTEM IMPLEMENTATION

Similar to Spark / GFS etc.



OPTIMIZATIONS

Tasks are stateless

→ launch for each timesteps

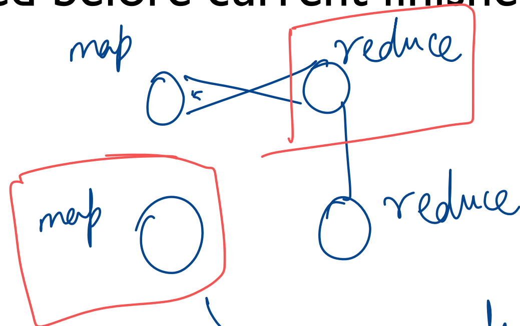
Timestep Pipelining

No barrier across timesteps unless needed

Tasks from the next timestep scheduled before current finishes

t=(0,1)

t=(1,2)



Can launch this while prev reduce is running

lineage is ∞

Checkpointing

Async I/O, as RDDs are immutable

Truncate lineage after checkpoint

RDD in memory t=5

background save to disk t=8

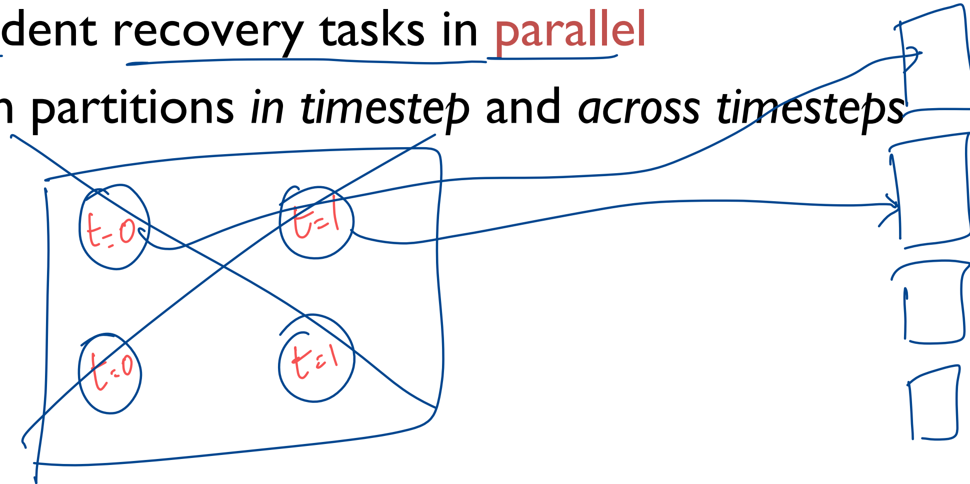
FAULT TOLERANCE: PARALLEL RECOVERY

Worker failure

- Need to recompute state RDDs stored on worker
- Re-execute tasks running on the worker → similar as Spark

Strategy

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

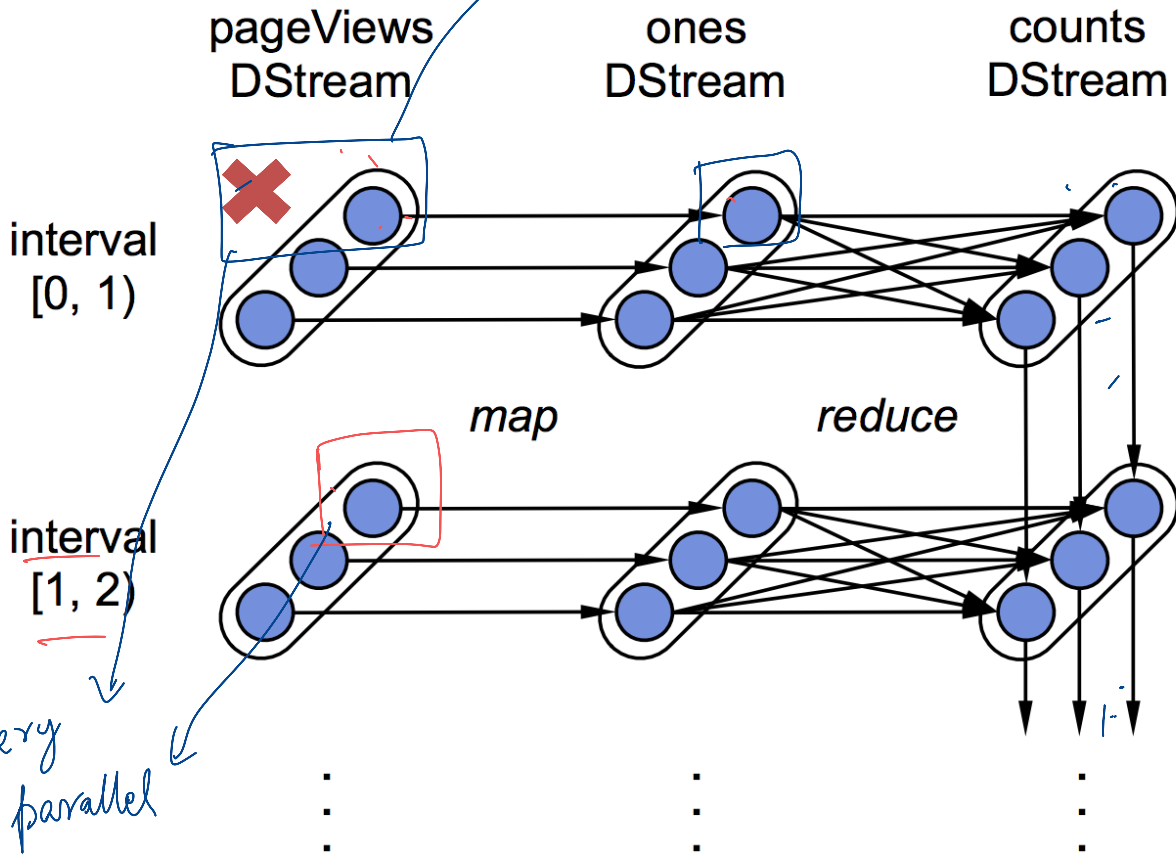


well defined semantics for each task

Async ckpt + lineage

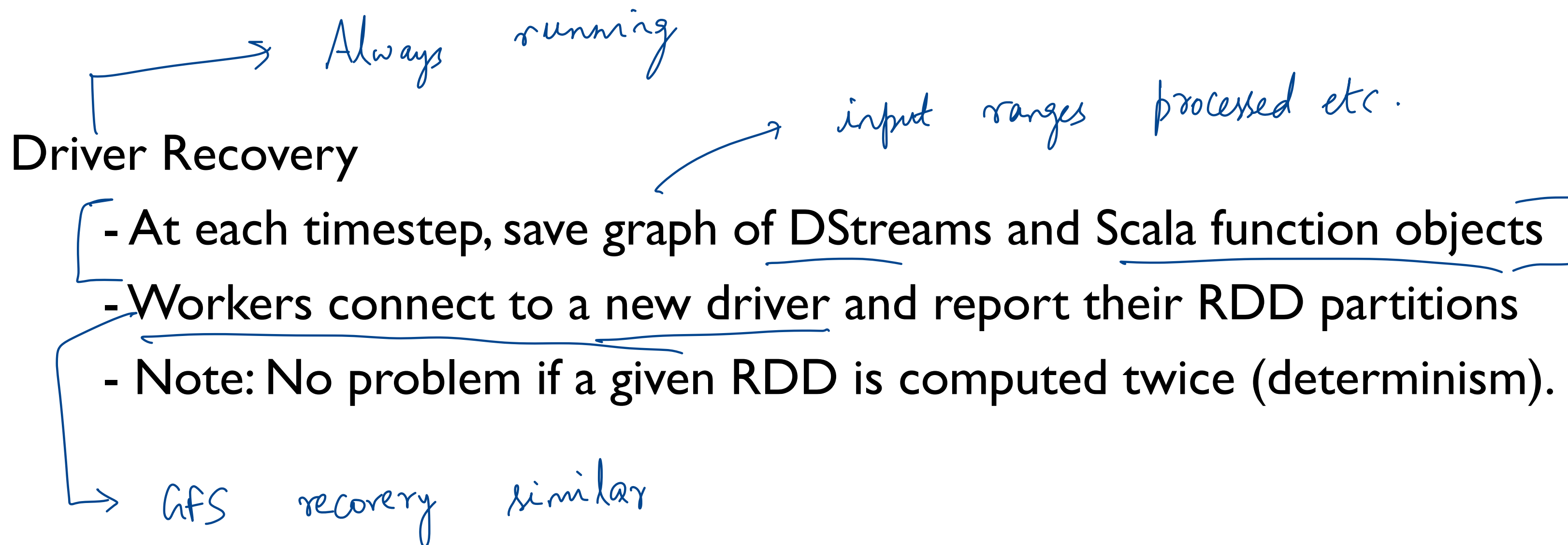
EXAMPLE

```
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



Driver Recovery

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

SUMMARY

Micro-batches: New approach to stream processing

Simplifies fault tolerance, straggler mitigation

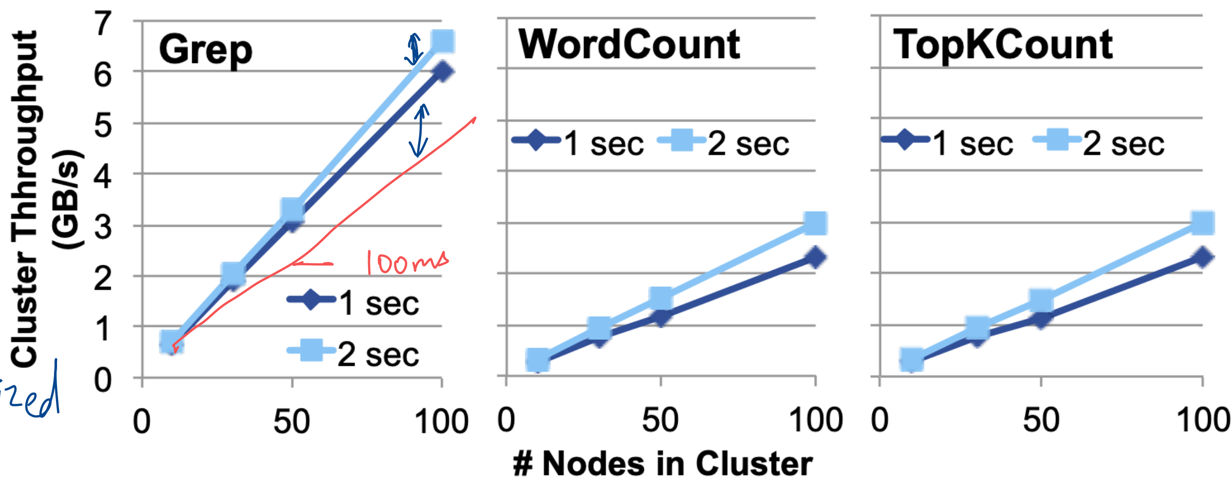
Unifying batch, streaming analytics

DISCUSSION

https://forms.gle/rkBykWeSgiQhPJf57

If the latency bound was made to 100ms, how do you think the above figure would change? What could be the reasons for it? What about 10s ?

Increase but not too much → fixed overheads already amortized



③ limited parallelism across timesteps

① Utilization of each task is lower

→ Time for task = fixed overhead + per-input time

100 MB

② Cross timestep dependencies → adds more metadata for master

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

NEXT STEPS

Next class: Graph processing!

Midterm grades soon!