

Hello!

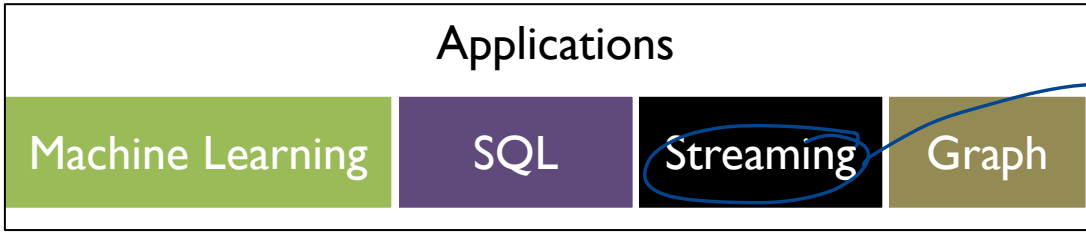
CS 744: SPARK STREAMING

Shivaram Venkataraman

Spring 2025

ADMINISTRIVIA

- Course Projects feedback → *Soon!*
- Midterm grades
- Shivaram travel week after Spring break ↘ *Jason Mchoney*
- Cloudlab reservations: Per-user from now
- Google credits! → *Piazza*
- Reading group interest?



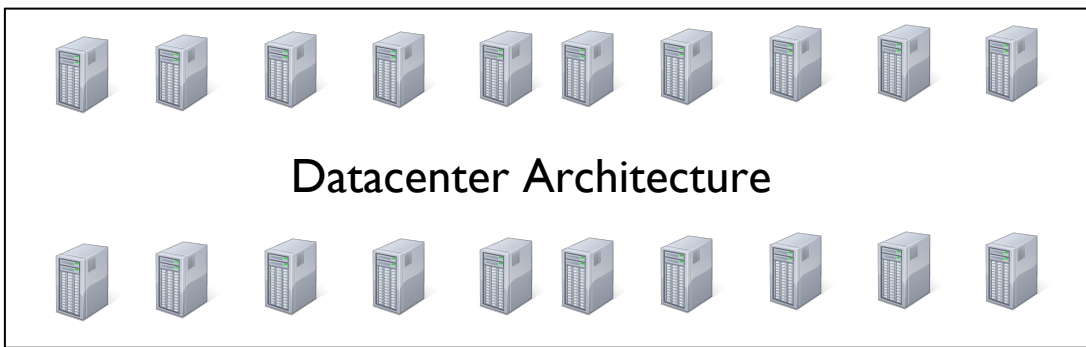
Dataflow
↳ API



Flink
↳ Continuous operator



Stream



spark
streaming

DASHBOARDS

Sales Dashboard

Total Sales
\$3,256.8M

Number of Deals
17,164

Avg Deal Size
\$189,545

Rev. per Salesperson
\$20.5M

Week of Date Closed

December 6, 200 - December 25, 20



Region

(All)

Country

(All)

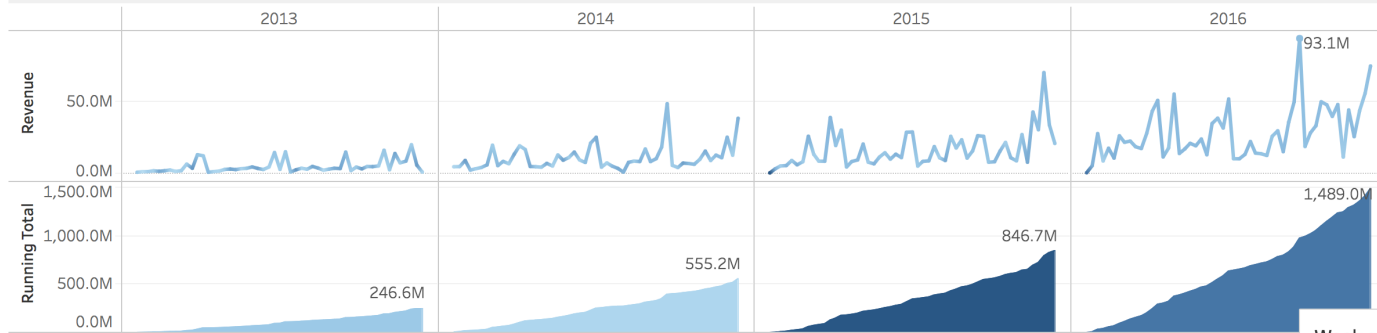
Sales Team

- (All)
- Small and Midmarket
- Enterprise

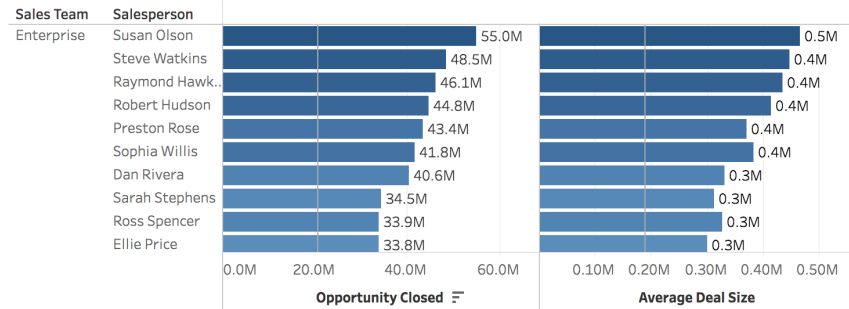
Avg Deal Size/Salesperson



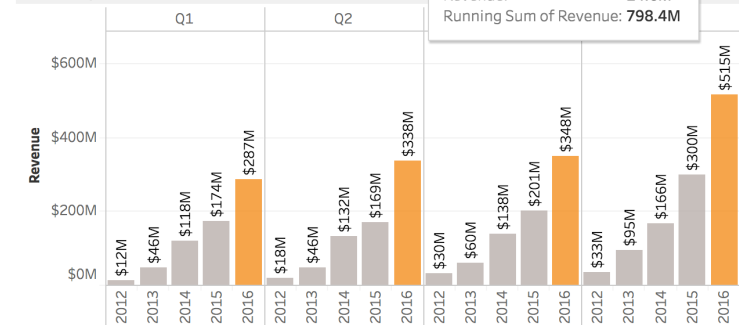
Revenue Over Time



Sales Team Performance

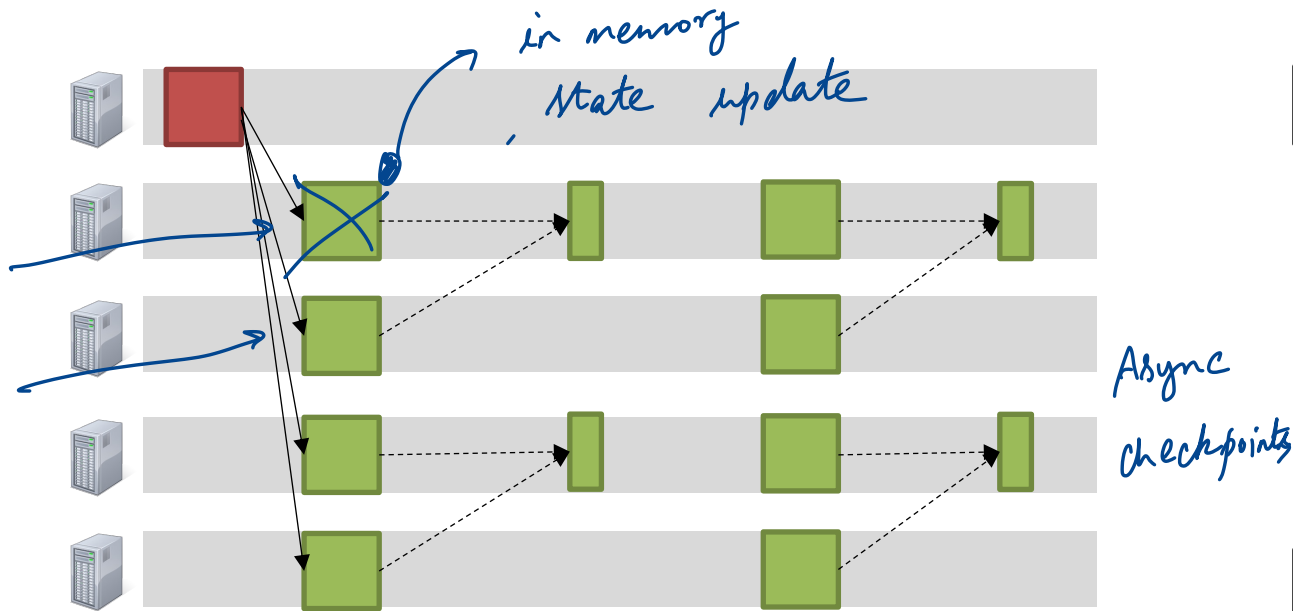


Revenue by Quarter



Week of September 4, 2016
Revenue: 14.6M
Running Sum of Revenue: 798.4M

CONTINUOUS OPERATOR MODEL

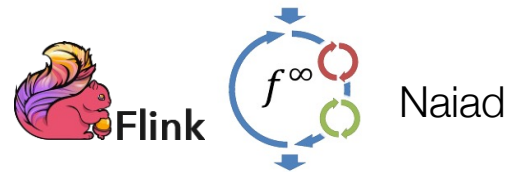
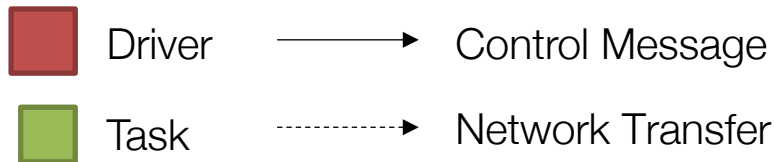


Long-lived operators

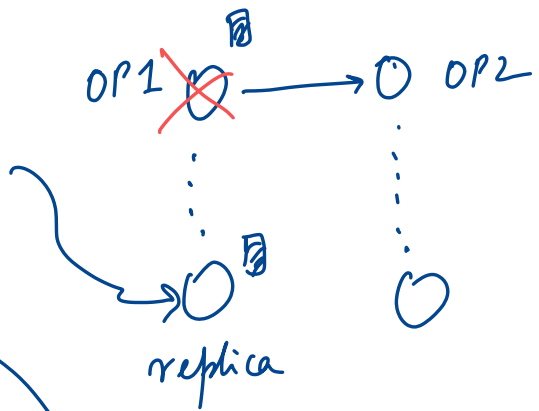
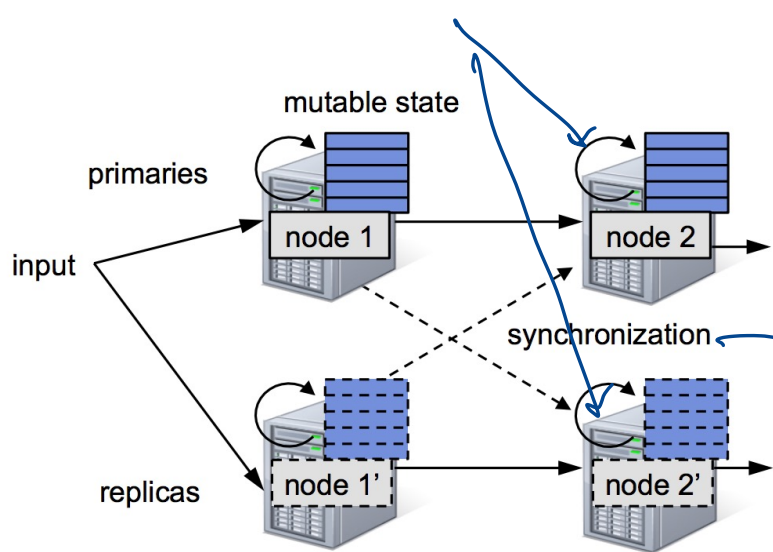
Mutable State

Distributed Checkpoints
for Fault Recovery

Stragglers ?



CONTINUOUS OPERATORS



2x more hardware required

complex

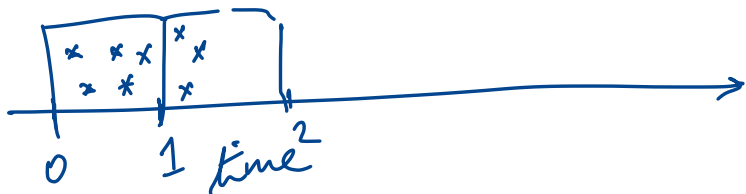
slow down the system

SPARK STREAMING: GOALS

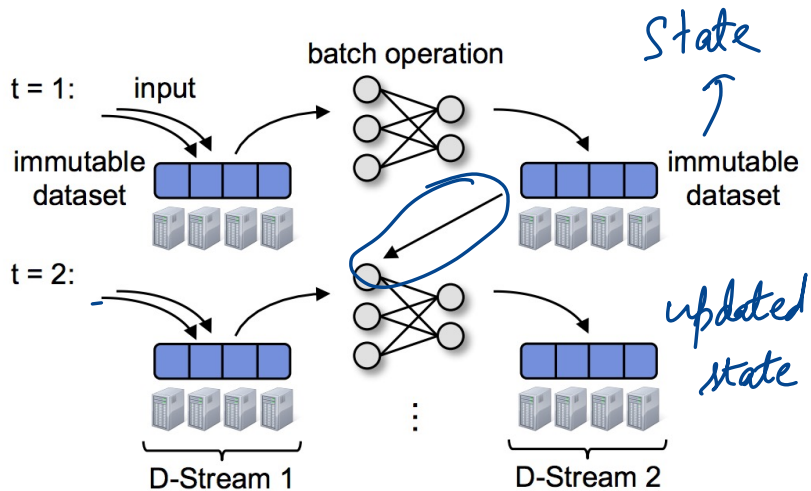
1. Scalability to hundreds of nodes *scale out is important*
2. Minimal cost beyond base processing (no replication)
3. Second-scale latency *seconds-scale*
length of time between when an event arrives to when it is reflected in the output
4. Second-scale recovery from faults and stragglers
pretty fast to recover from failures

DISCRETIZED STREAMS (DSTREAMS)

batch input records & periodically dispatch the batch for processing



→ reusing batch computation framework at every time step



EXAMPLE

```
url1, 10  
url2, 5  
⋮
```

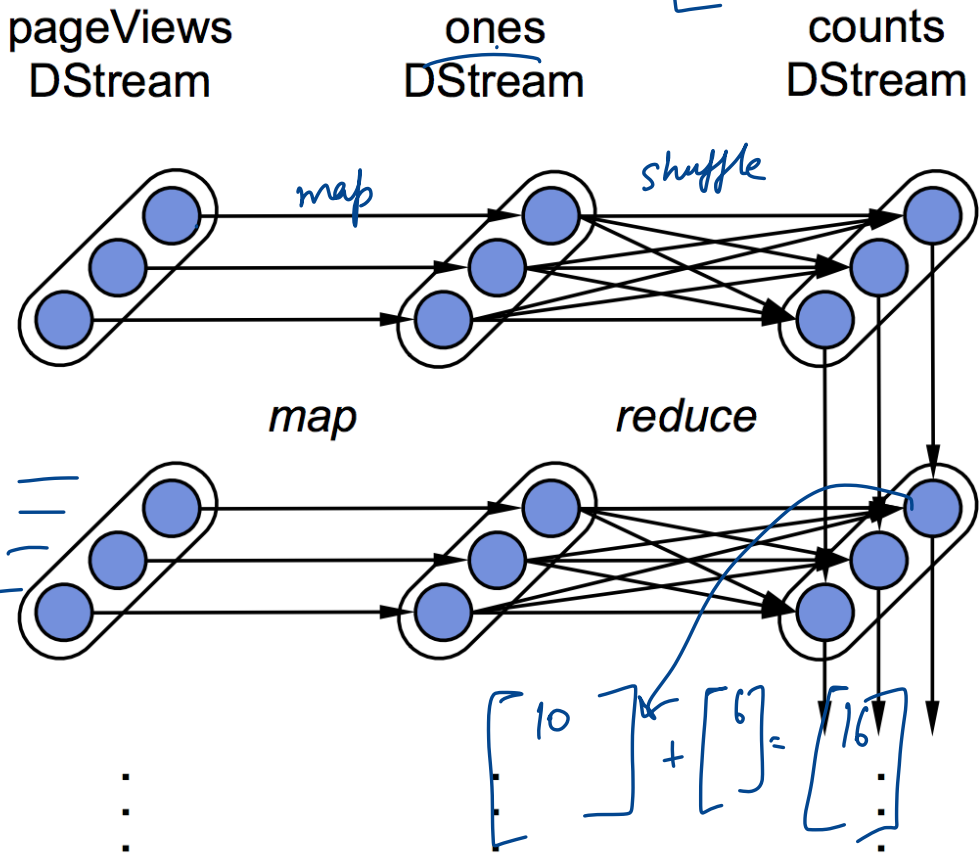
```
pageViews =  
readStream(http://...,  
"1s")
```

new operator
batch size

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

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

global sum of counts / hits for each URL



DSTREAM API

Transformations

→ Stateless: map, reduce, groupBy, join

*same transformations
as RDD API*

→ Stateful:

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

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

*across S batches → track & garbage
collect old state*

STATE MANAGEMENT



Motivation:
windowing on
event time

Tracking State: streams of (Key, Event) \rightarrow (Key, State)

```
events.track(  
  (key, ev) => 1,  $\rightarrow$  Initialize State for this window
```

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

```
  "30s")
```

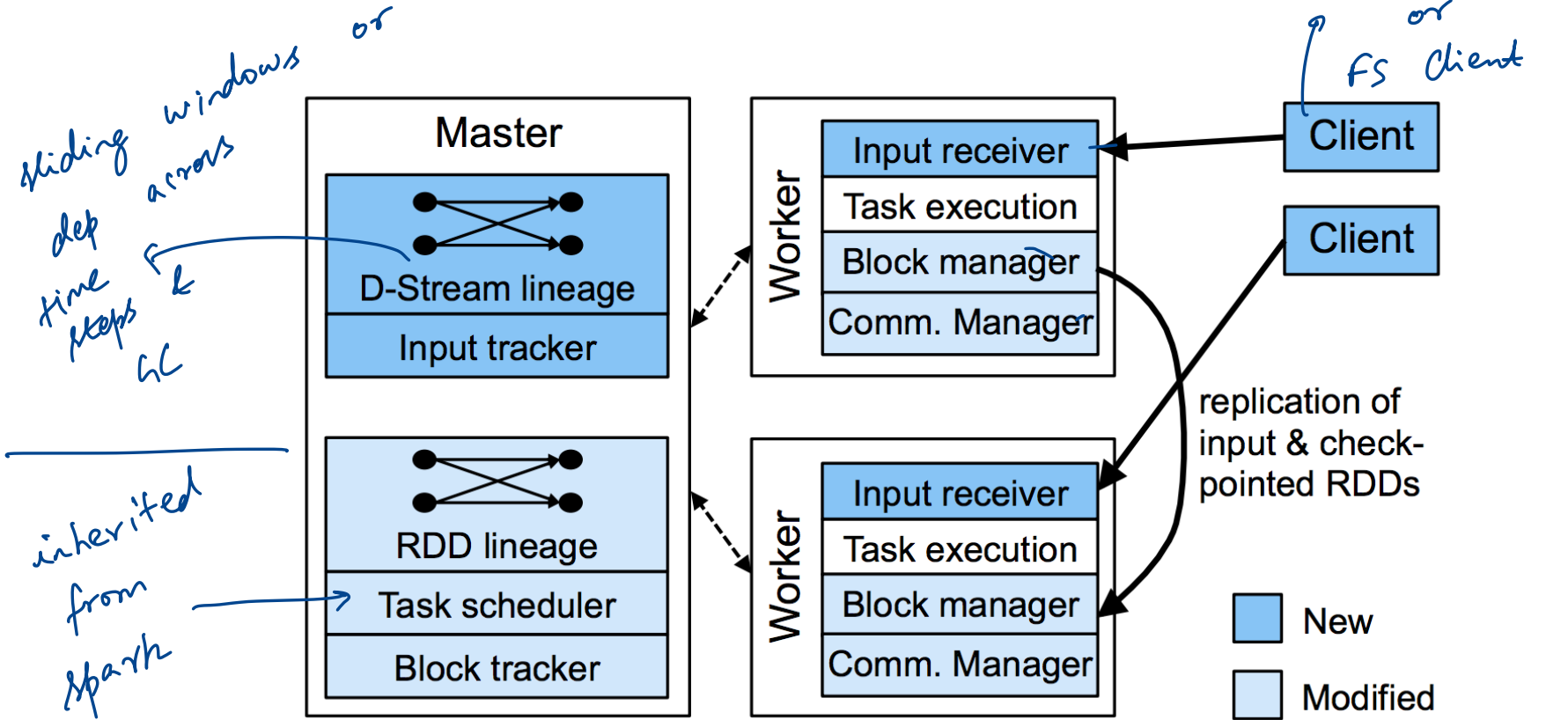
Tracked by
the system

Forget time

Update the State given
a new event & prev
state : return new
state

URL or
user id etc.

SYSTEM IMPLEMENTATION



OPTIMIZATIONS

Timestep Pipelining

No barrier across timesteps unless needed

Tasks from the next timestep scheduled before current finishes

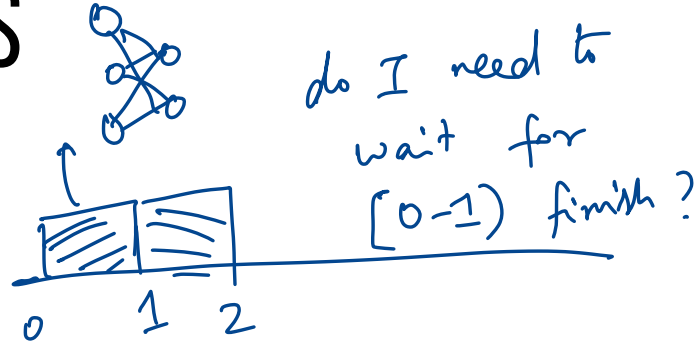
↳ wait for dependencies

Checkpointing → at a regular interval

Async I/O, as RDDs are immutable

Truncate lineage after checkpoint

→ similar to Spark as well



FAULT TOLERANCE: PARALLEL RECOVERY

Worker failure

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

re-run any tasks
in progress

Strategy

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

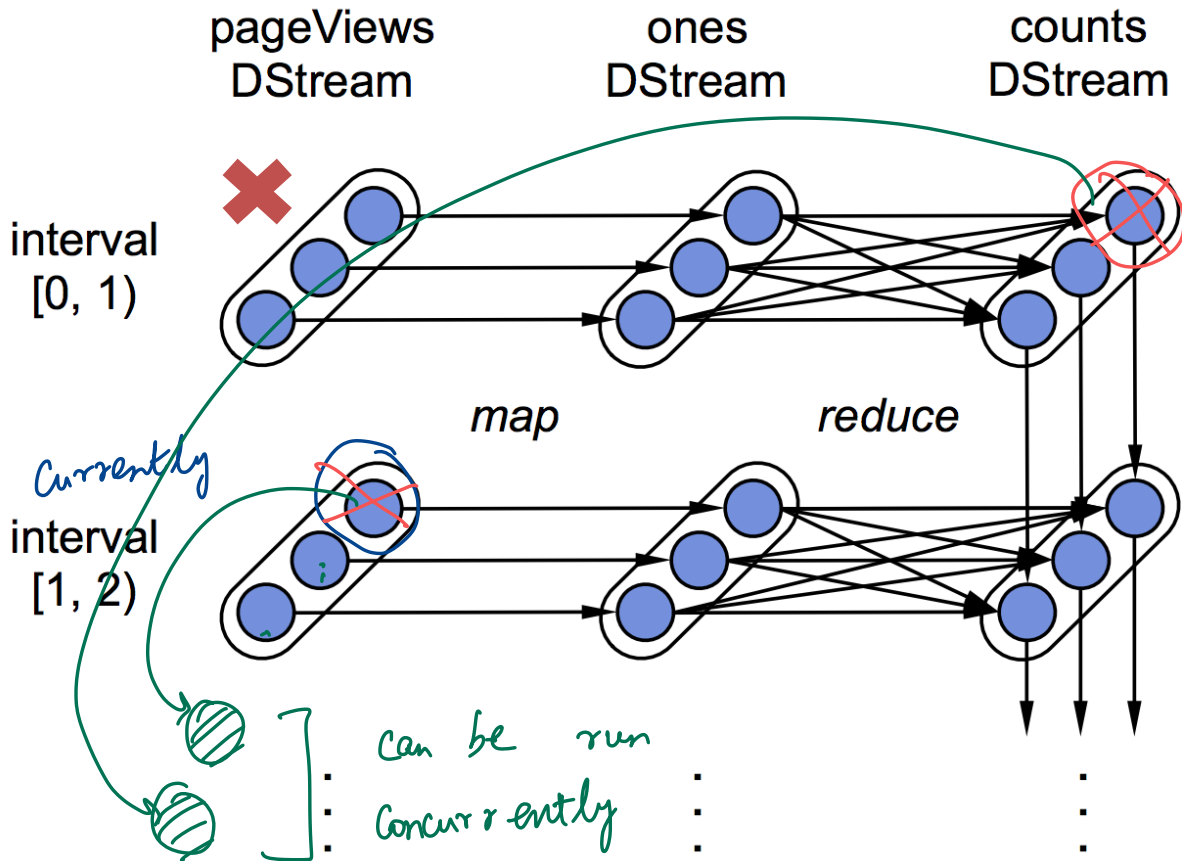
EXAMPLE

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

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

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

4 machines → 3 machines
8 partitions



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).

Spark driver had no fault tolerance (just restart)

save to disk

similar to GFS

driver can figure out what needs to be computed

SUMMARY

Micro-batches: New approach to stream processing

Simplifies fault tolerance, straggler mitigation

Unifying batch, streaming analytics



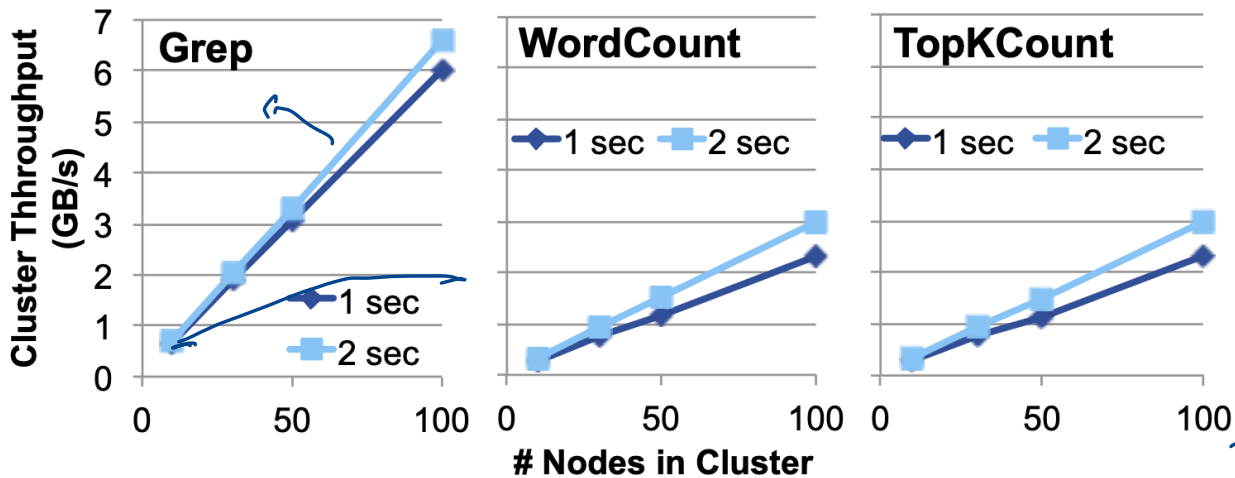
Share code



DISCUSSION

<https://forms.gle/6j3PCKV6AG7WVGqIw9>

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



latency bound ↑

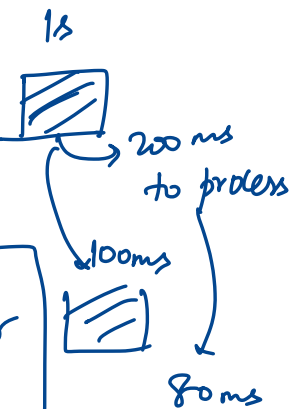
- fewer micro batches
- more records ??
- more tput ?

vectorization etc.

spark overhead per micro batch

100 ms → overhead is higher ??

input constrained "too small"



Discuss what part of queries considered by the Dataflow paper could use the "track" operation in Spark Streaming.

NEXT STEPS

Next week: Spring break!

Next class: Graph processing!

Midterm grades soon!