Welcome back!

CS 744: SPARK STREAMING

Shivaram Venkataraman

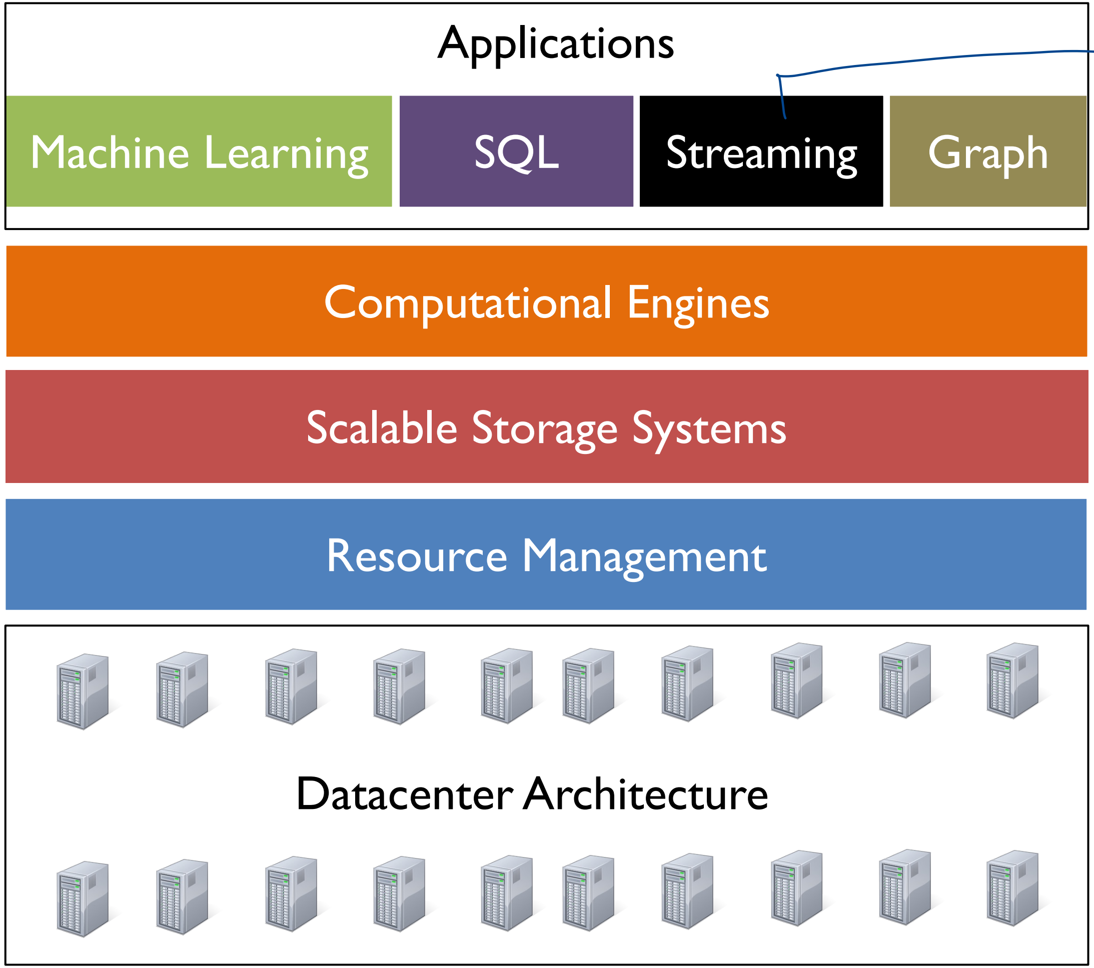
Spring 2024

ADMINISTRIVIA

- Course Projects feedback → Canvas
- Midterm grades – this week?

- Cloudlab reservations → Course Project
  - Per-user from now

↳ only you will be able to use that reservation



Dataflow model

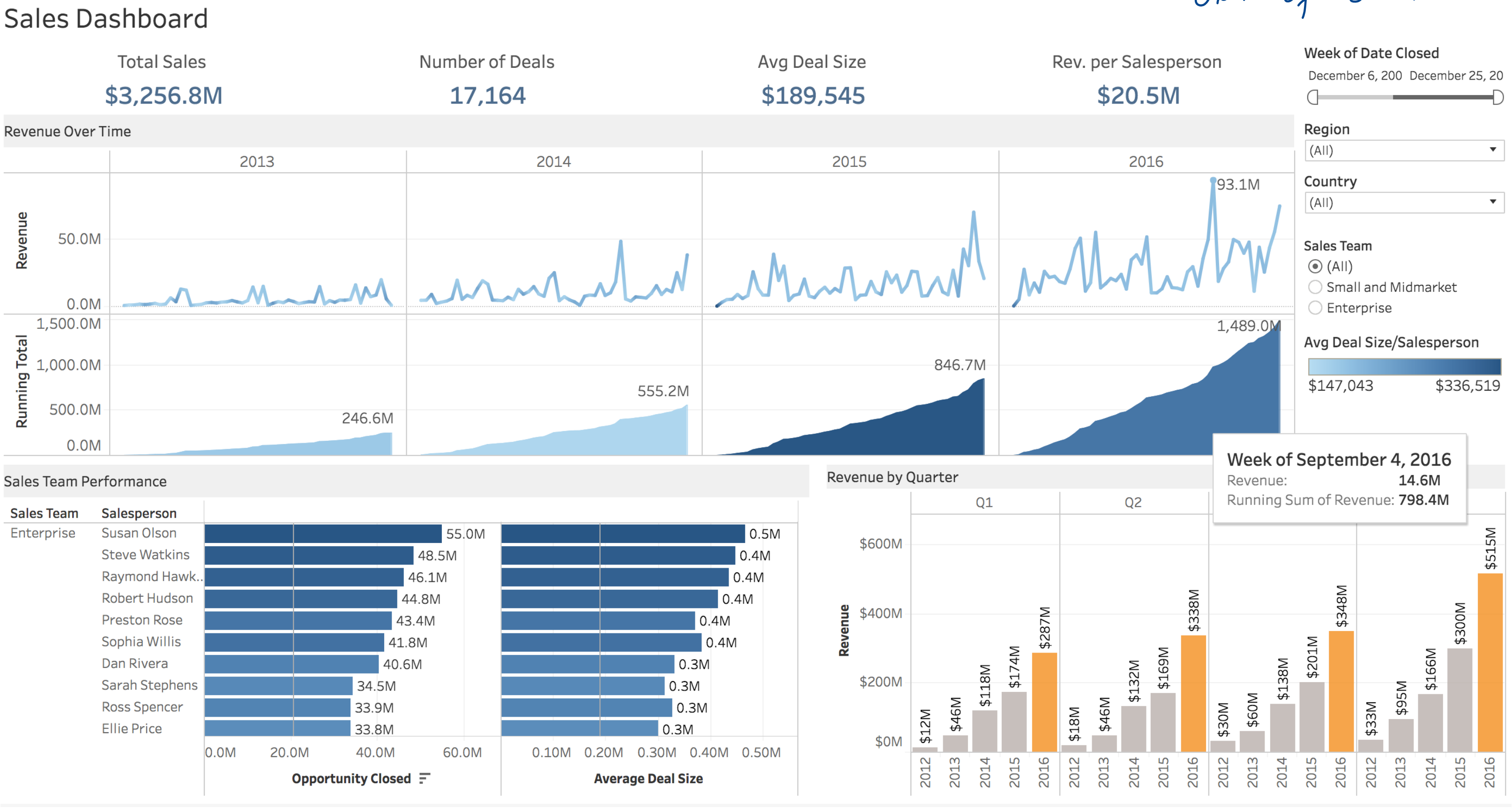
↳ how to express streaming queries

Apache Flink

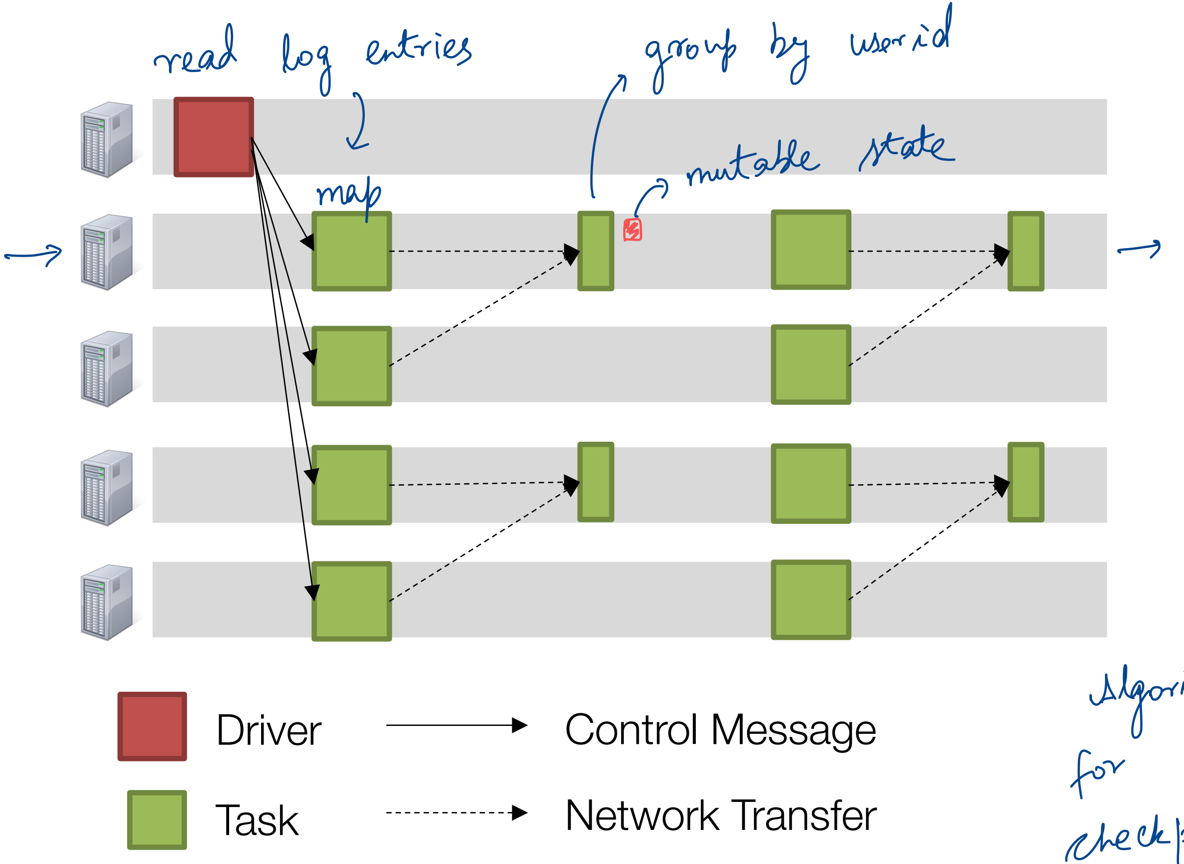
↳ realizes continuous operator

DASHBOARDS

data continuously over time → low latency
→ out of order delivery



CONTINUOUS OPERATOR MODEL



Long-lived operators

Mutable State

Distributed Checkpoints for Fault Recovery

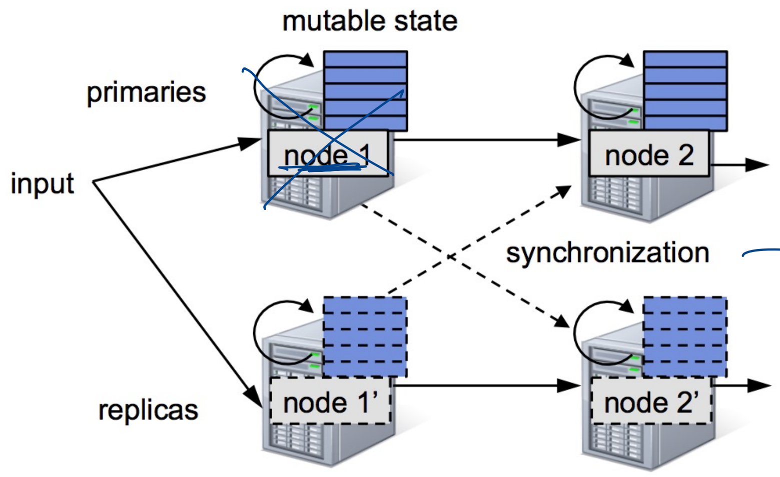
Stragglers ?

Algorithm for checkpointing

Flink

Naiad

CONTINUOUS OPERATORS



replicate operators across nodes

↳ more resources to support this scheme

same order of events to the replica operator

↳ overhead during processing

SPARK STREAMING: GOALS

1. Scalability to hundreds of nodes → high throughput

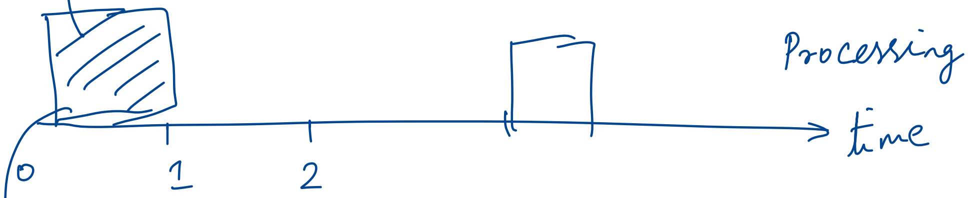
2. Minimal cost beyond base processing (no replication)

3. Second-scale latency → time between event arriving & it being reflected in the output

4. Second-scale recovery from faults and stragglers

DISCRETIZED STREAMS (DSTREAMS)

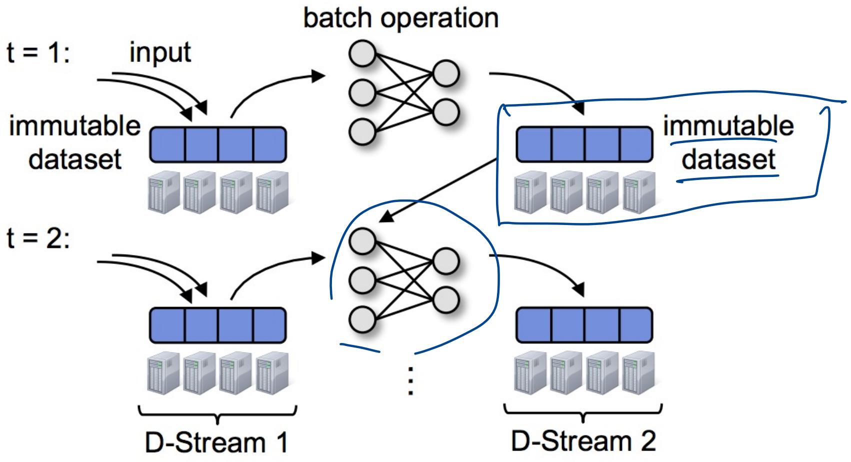
Could contain diff event times



batch → submit this for computation

↳ re-using batch computation frameworks

→ Save some state at the end of each batch and use that as input for next batch



new input

DStream API

EXAMPLE

read input

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

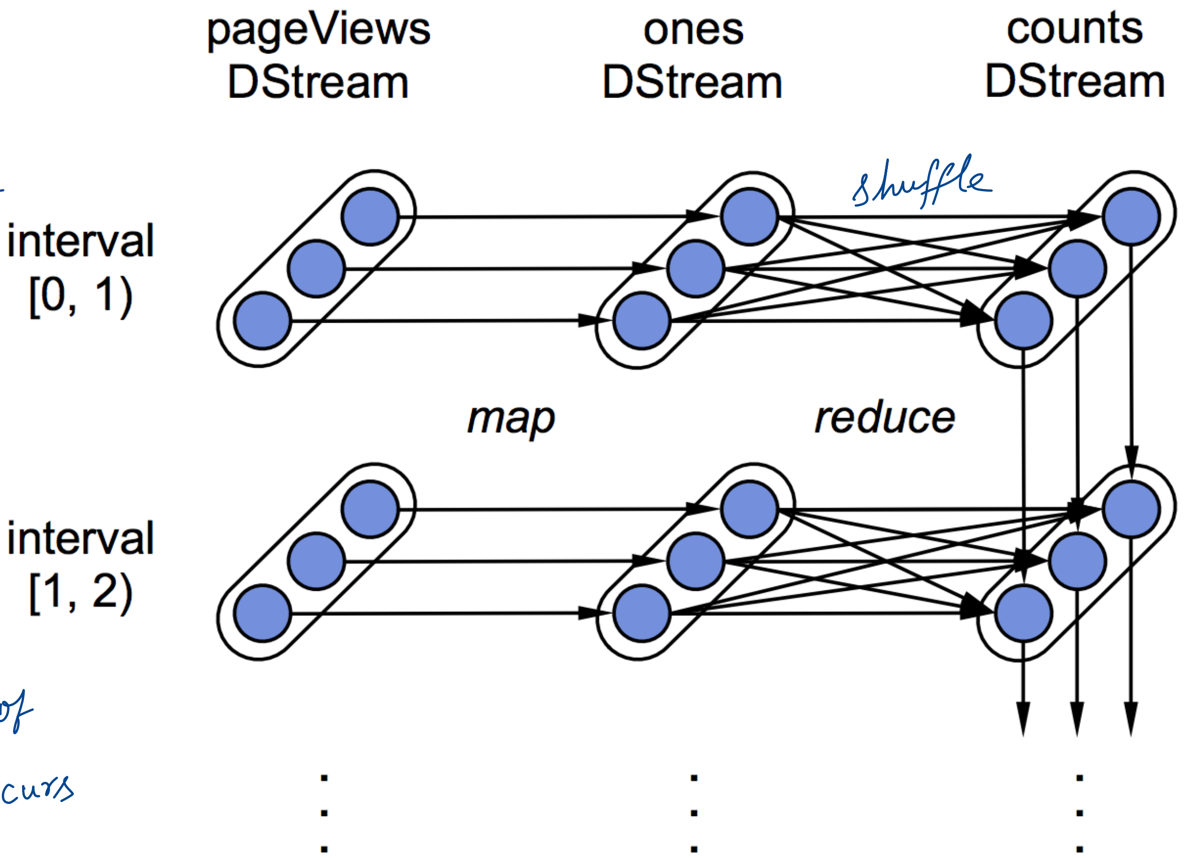
batch size

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

Key

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

aggregate number of times URL occurs



DSTREAM API

Transformations

Stateless: map, reduce, groupBy, join

similar to batch API

do not have dependencies across time steps

Stateful:

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

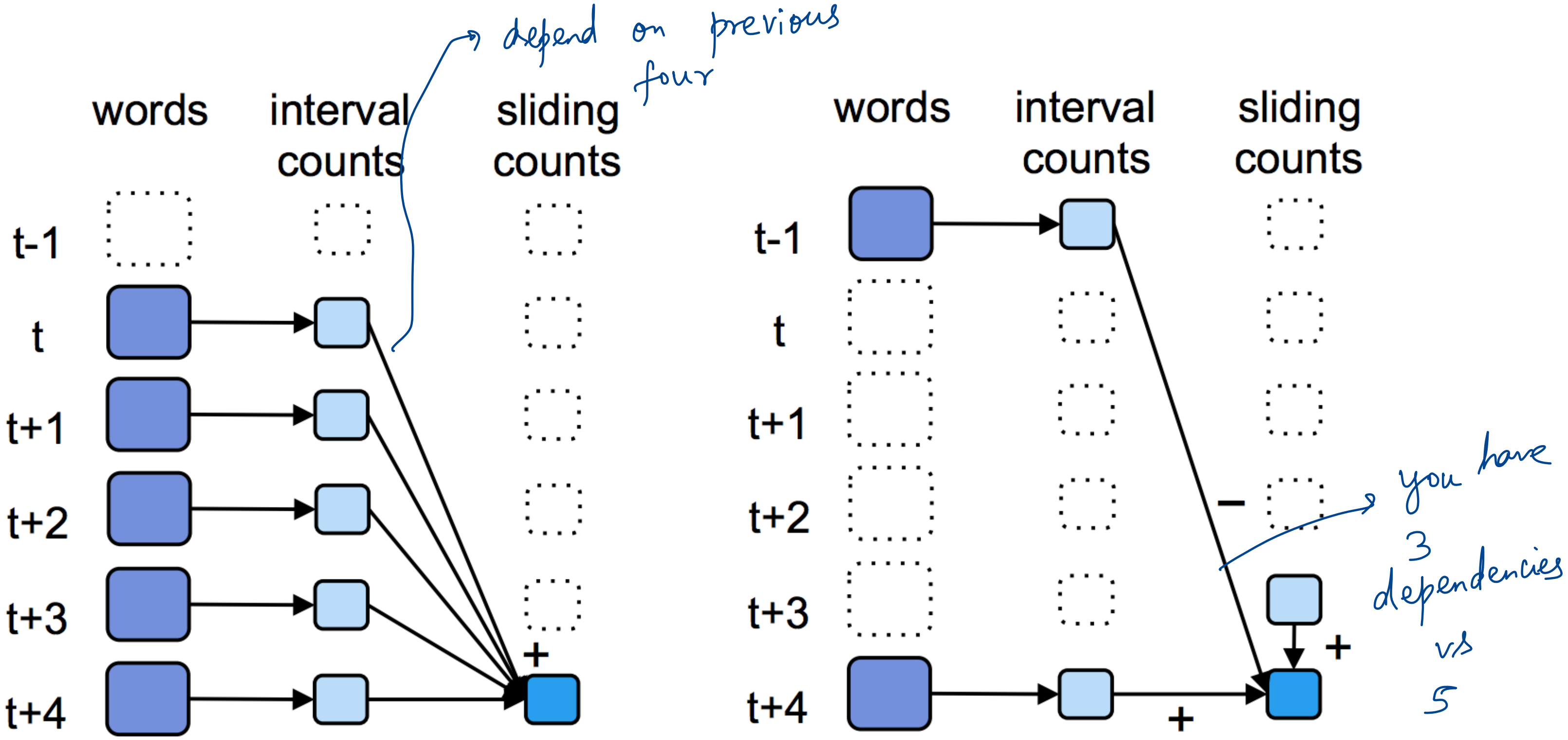
reduceByWindow("5s", (a, b) => a + b)

↳ creates a window & uses this reduction function

SLIDING WINDOW

5 seconds duration

Add previous 5 each time



(a) Associative only

(b) Associative & invertible

STATE MANAGEMENT

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

Similar in spirit to mutable state in flink

DStream

operator

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

Initialize state

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

Given key, prev state, new event → new state

```
  "30s")
```

Forget

key, event → what is initial value

→ examples include event time range / session Id etc.

STATE MANAGEMENT

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

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

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

   "30s")
```

Computation requires state!!

Stateless operator

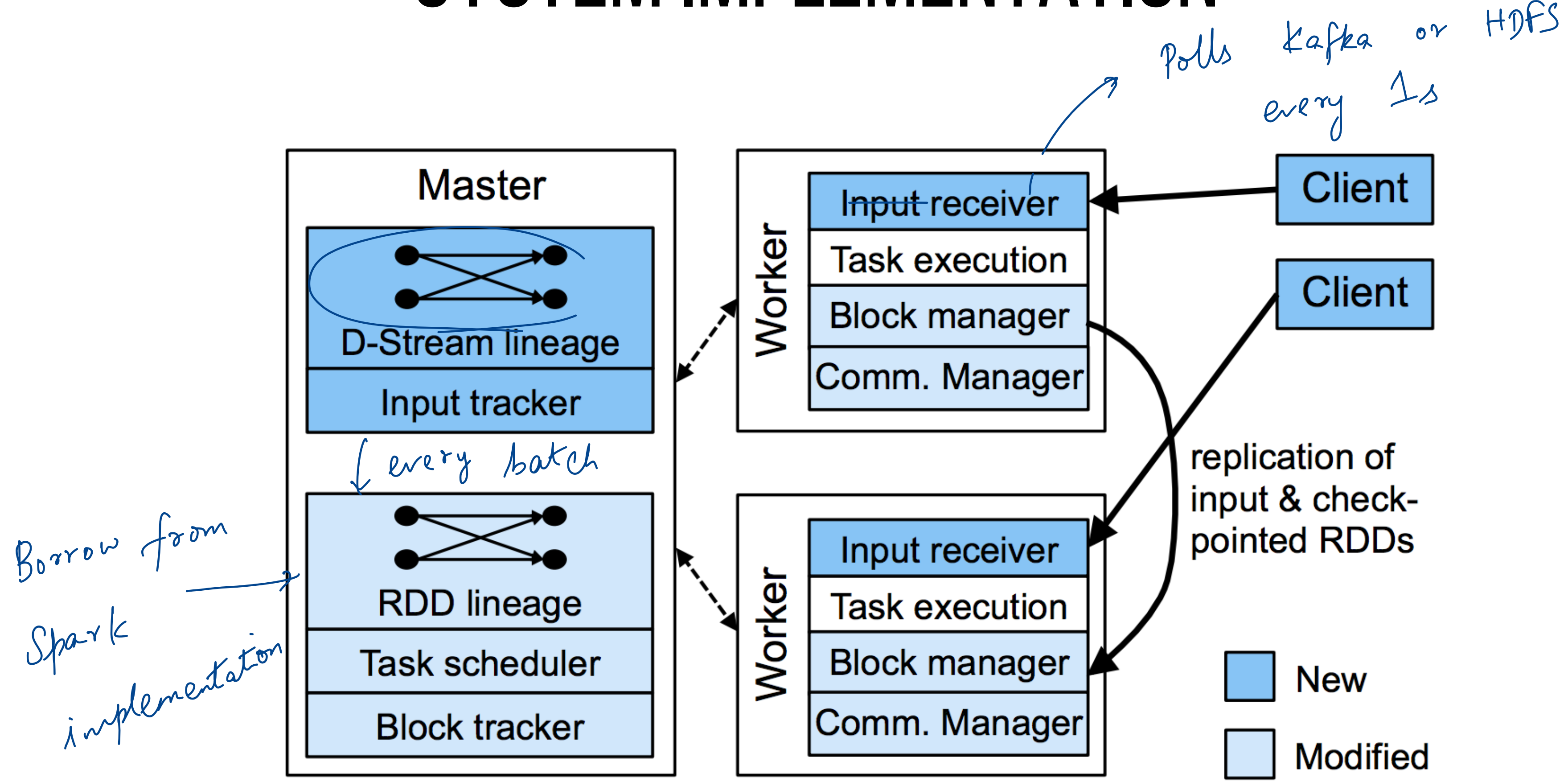
System manages state

vs.

Operator maintains state

System helps with checkpoints etc.

SYSTEM IMPLEMENTATION



OPTIMIZATIONS

start computation t = 2 when t = 1 is still running

Timestep Pipelining

No barrier across timesteps unless needed

Tasks from the next timestep scheduled before current finishes

simple to realize

Checkpointing

lineage can grow infinitely

Async I/O, as RDDs are immutable

Truncate lineage after checkpoint

background

FAULT TOLERANCE: PARALLEL RECOVERY

second scale

Worker failure

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

only need to replay tasks on this worker

Strategy

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

worker

4 tasks running → to run on 4 diff machines

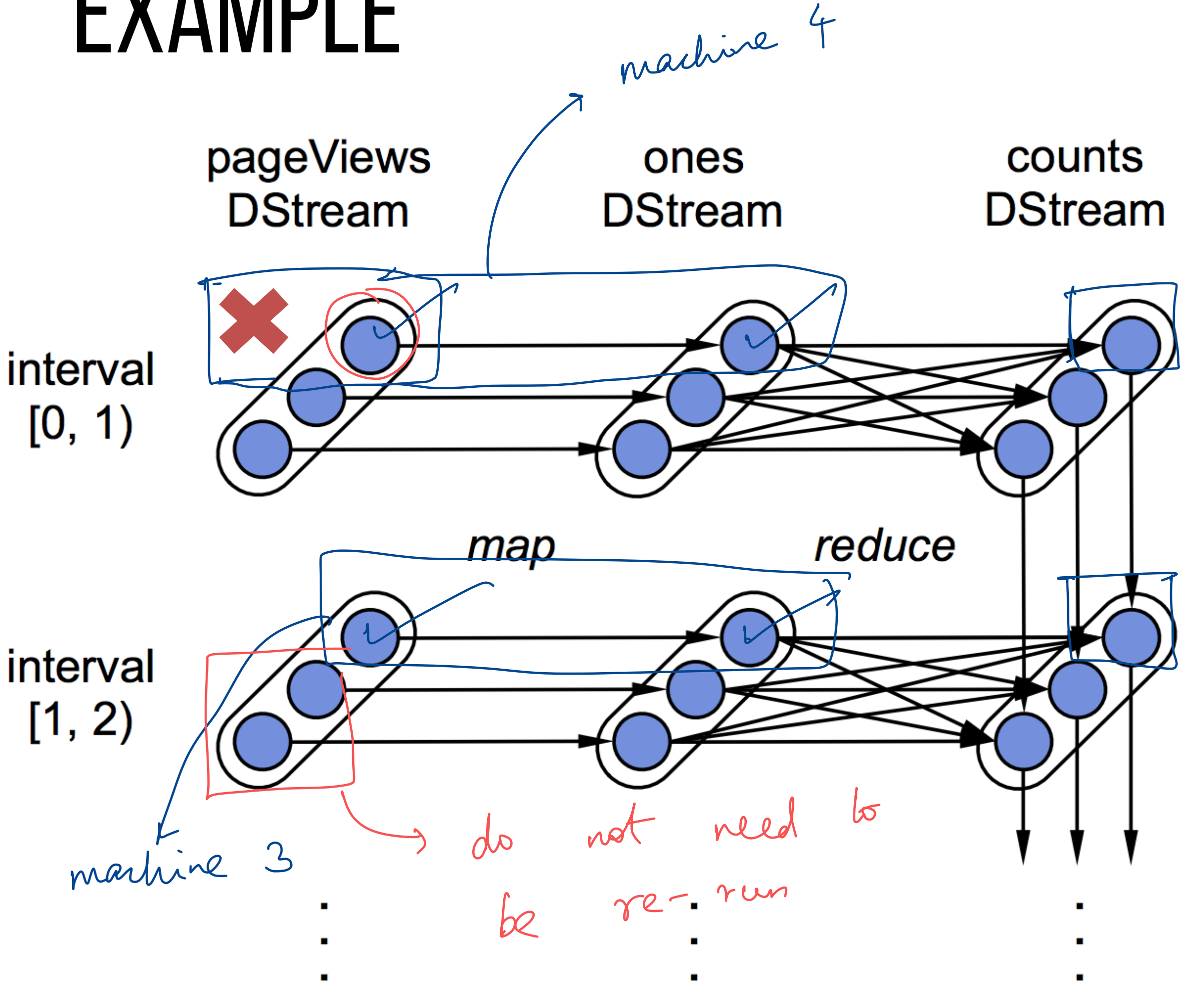
4 RDD state diff time steps → recover each on a diff machine

EXAMPLE

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

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

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



replayable input
↳ Kafka

FAULT TOLERANCE

Straggler Mitigation: Use speculative execution → operators are stateless multiple of them same time

checkpoint driver state

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).

similar to GFS recovery

SUMMARY

Micro-batches: New approach to stream processing

Simplifies fault tolerance, straggler mitigation

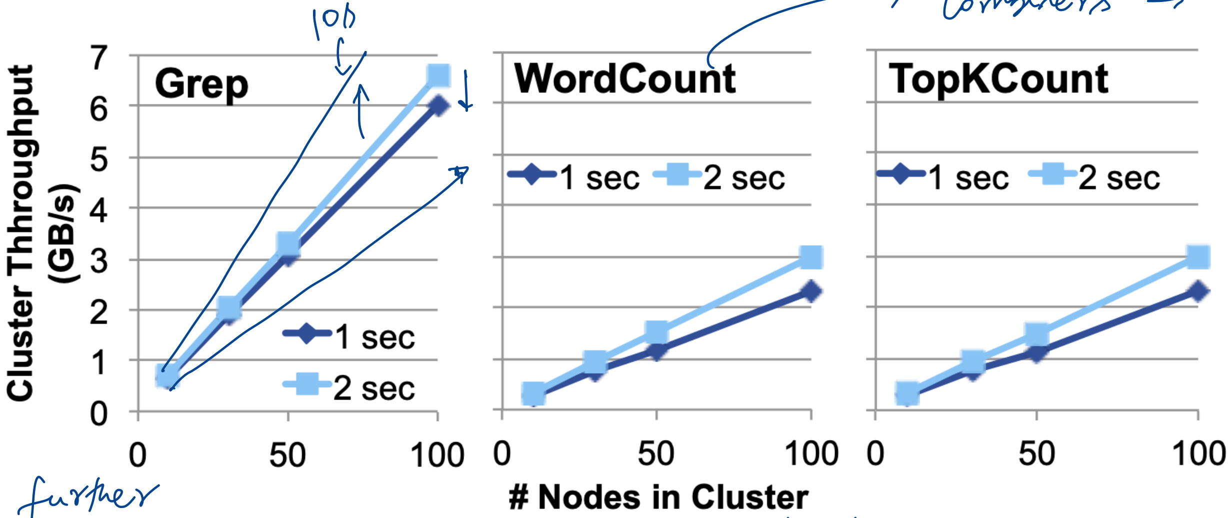
Unifying batch, streaming analytics

share code

DISCUSSION

https://forms.gle/RVtChgDQzbX16tqT7

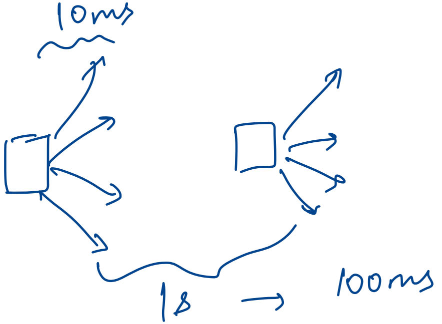
If the latency bound was made to 100ms, how do you think the above figure would change? What could be the reasons for it? what about 10s ?



Combiners → not as effectives at 100 ms

10s
↳ higher tput
capped at some limit
resource limits of cluster.

① 100ms
tput will further go down ?
↳ "overheads" will be higher
↳ scheduling, task launch
→ input reading



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

Flink

→ checkpoints are more expensive

↳ unreliable hardware

↳ don't want to use Flink

↳ cluster is small

→ FT is less of concern ?

Spark Streaming

↳ low latency

< second scale

then not Spark Streaming

→ streaming join with historical data spark has advantages

NEXT STEPS

Next class: Graph processing!

Midterm grades soon!