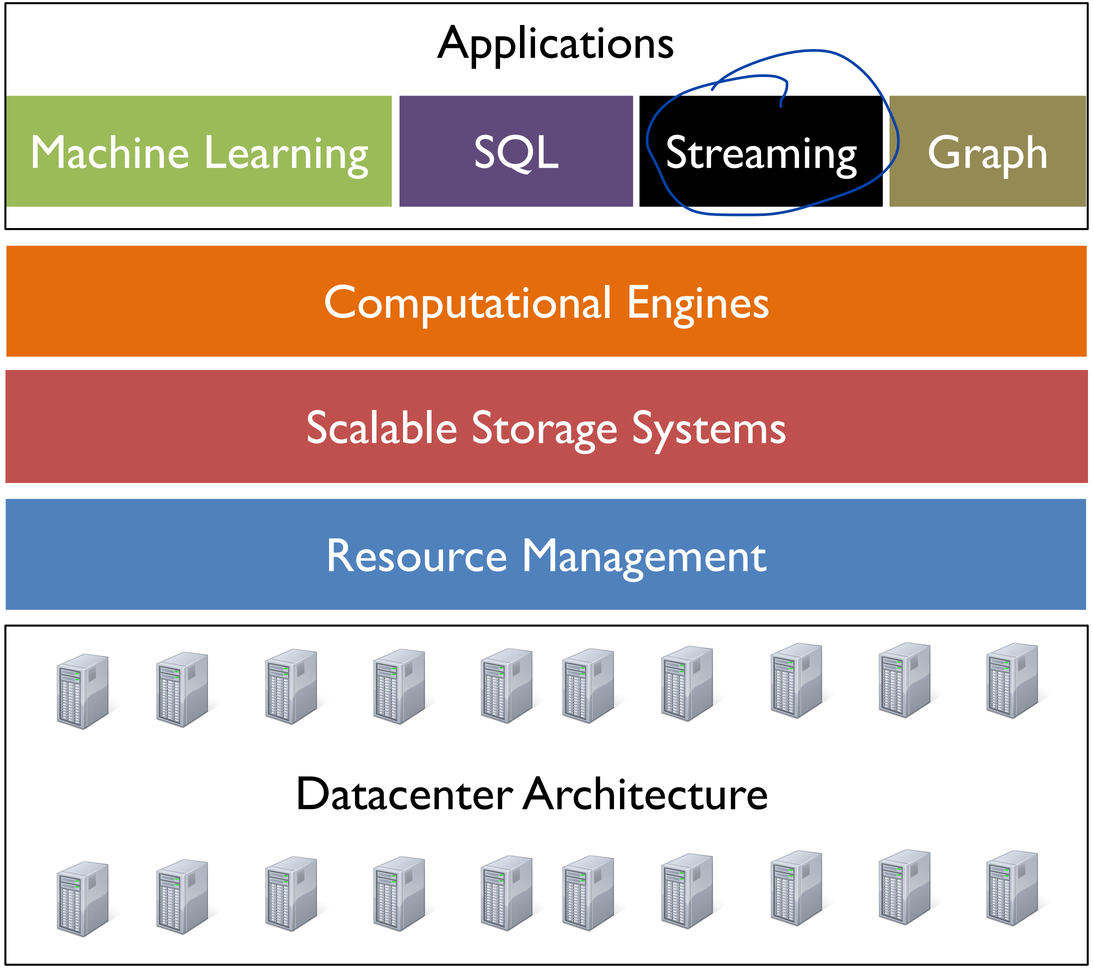
CS 744: SPARK STREAMING

Shivaram Venkataraman

Fall 2019

ADMINISTRIVIA

- Midterm grades this week
- Course Projects sign up for meetings → Nov 21, 25

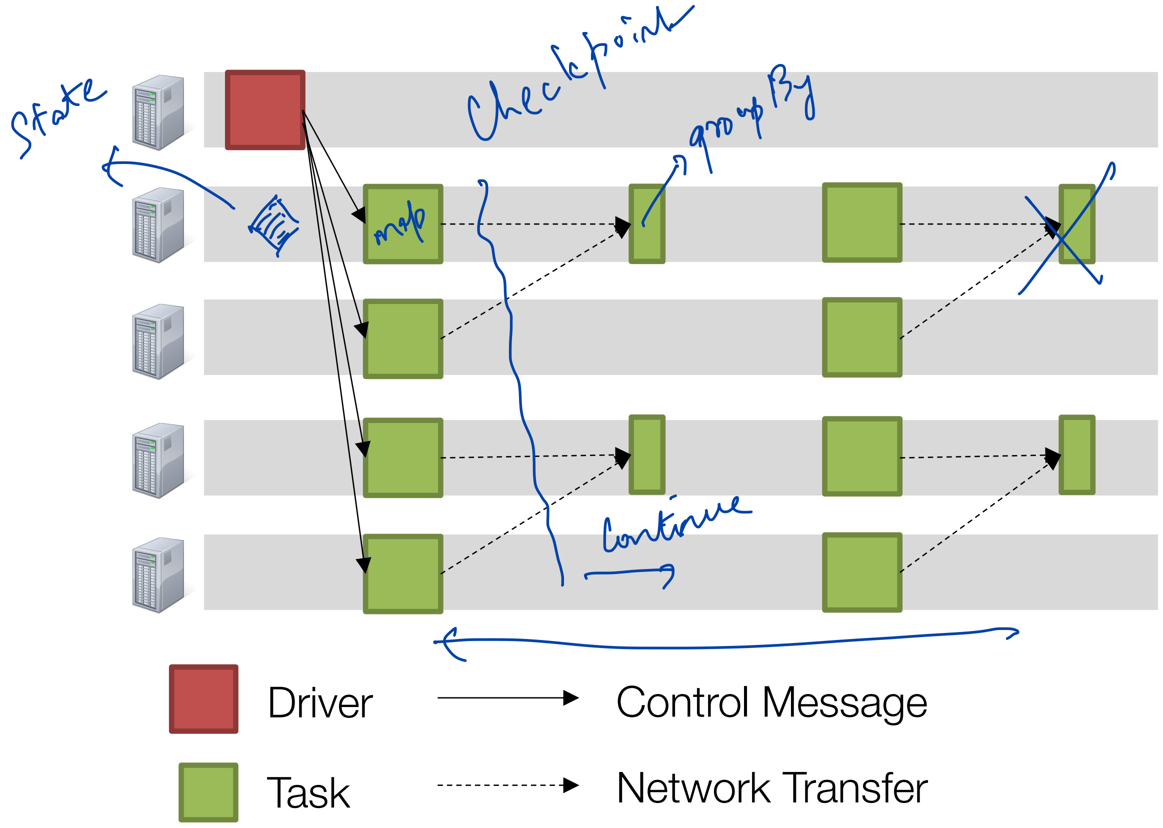


Dataflow model

→ Queries

Properties of Data, Queries

CONTINUOUS OPERATOR MODEL



Long-lived operators

Mutable State

Distributed Checkpoints for Fault Recovery

Prevent

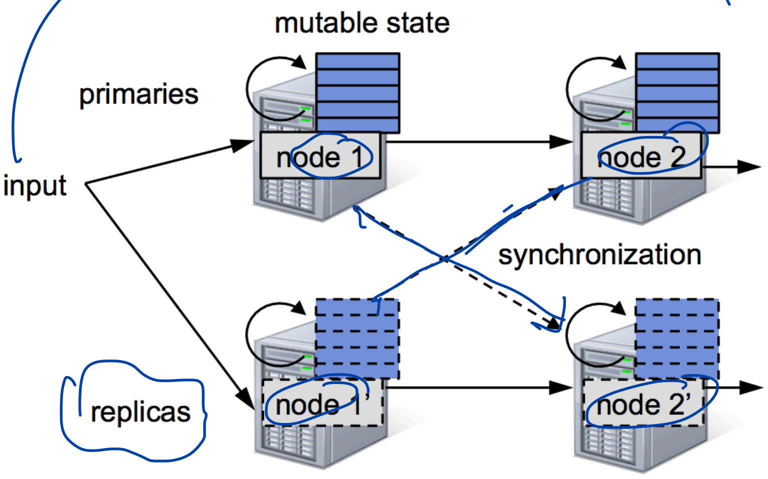
Stragglers ?

Flink

Naiad

CONTINUOUS OPERATORS

order of data



resource / memory overhead

group By (country)

Country	Pageview

mutable state differences across replicas

Country	Pageview

Upstream backup

recovery is very fast

strict ordering

↳ synchronization

might not mitigate stragglers

SPARK STREAMING: GOALS

1. Scalability to hundreds of nodes

$\lessapprox$ 2x overhead

2. Minimal cost beyond base processing (no replication)
3. $\boxed{\text{Second-scale}}$ latency → Revisit
4. Second-scale recovery from faults and stragglers

DISCRETIZED STREAMS (DSTREAMS)

short, deterministic tasks

from t = 0

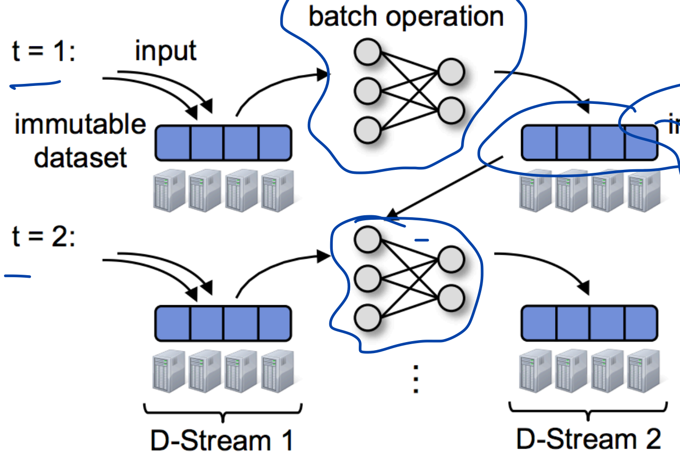
0 t: 1 2 3

→ Stateless operation

→ replicate / recover operation } low overhead

t = 1

manage State



→ run ops in parallel

→ each batch can be run in ||l

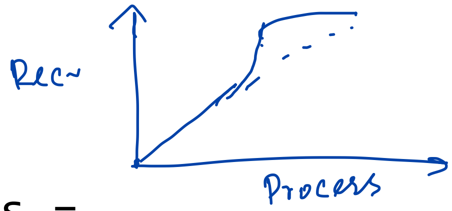
t = 2

→ recovery or straggler mitigation in parallel] → Low latency recovery

Dependencies within / across timestamp

EXAMPLE

similar to RDD

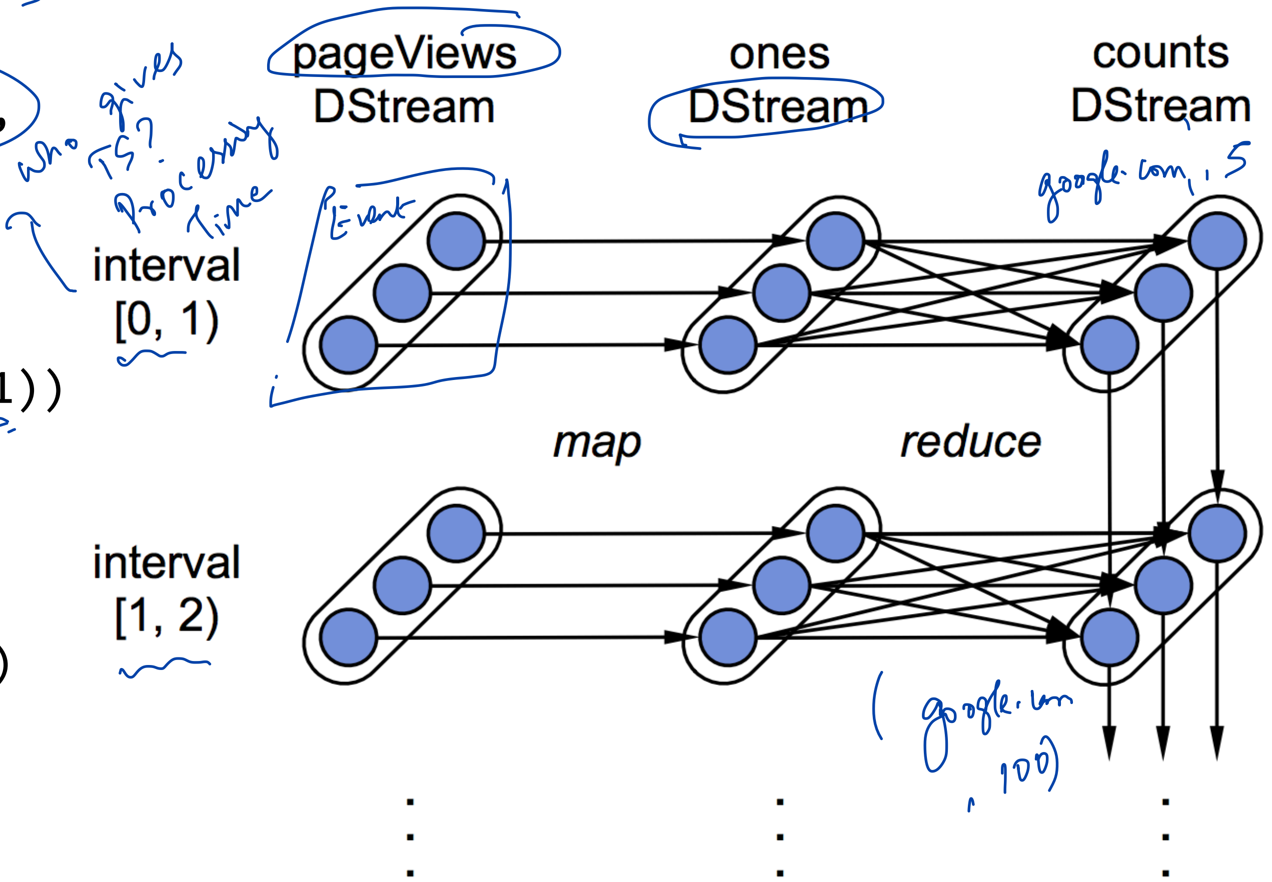



```
pageViews =
  readStream(http://...,
             "1s")
```

batch duration

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

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



ARCHITECHTURE

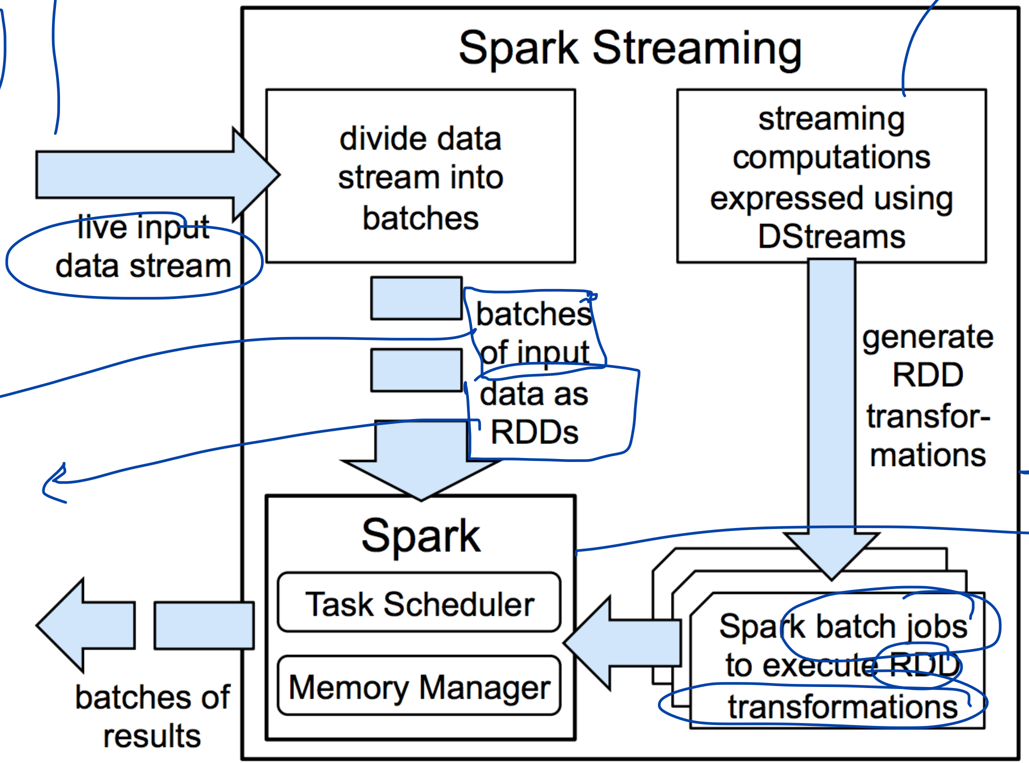
read the FS every ls
subtract before that
Force

.count()   foreach{ }

DStream Computation
/directory

Socket, FileSystem

stream input



batched across time

t = (0, 1)

FS
↳ ls recursive /directory

→ diff against prev version

→ name files

F1
F2
F3

DSTREAM API

Transformations → on DStream

map

Stream → Stream

1:1 mapping

Stateless: map, reduce, groupBy, join

not available on RDDs

Stateful:

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

Sliding window

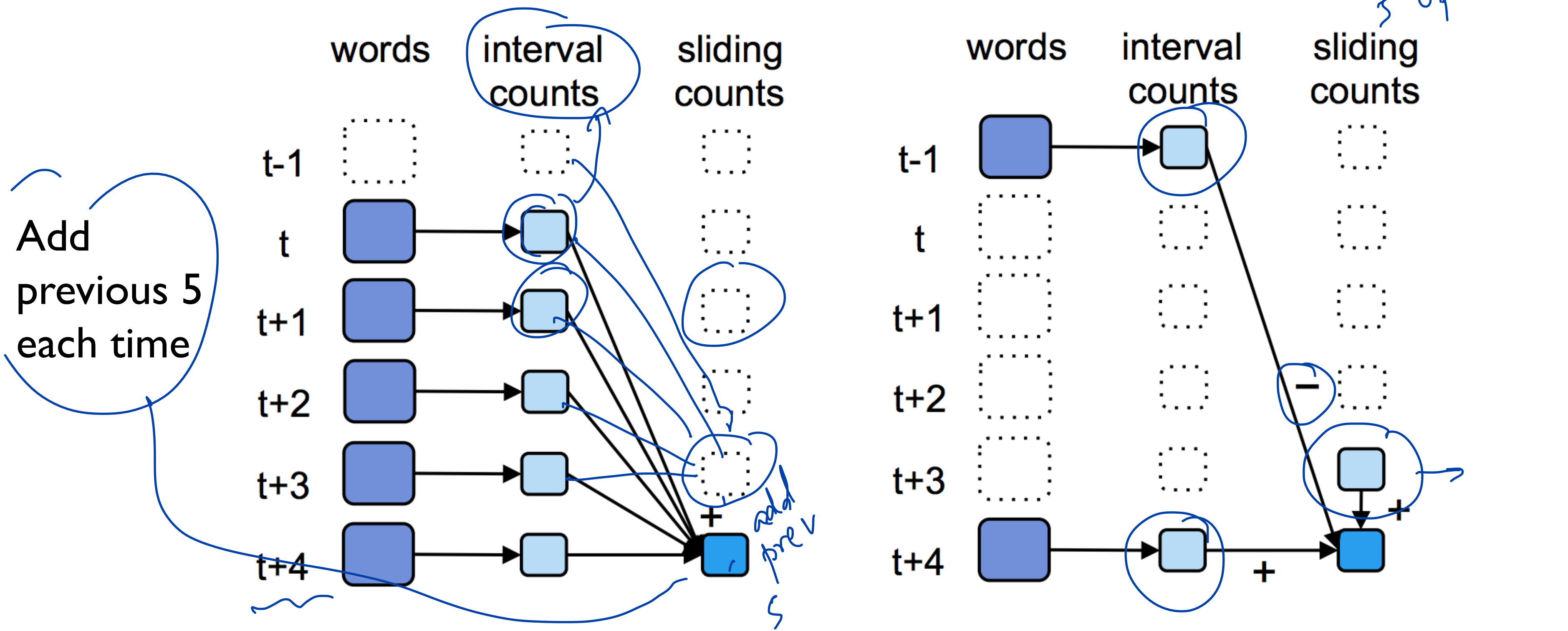
reduceByWindow("5s", (a, b) => a + b)

↓

Operate across windows

reduce By Window (5s, a+b)

SLIDING WINDOW



(a) Associative only (b) Associative & invertible

STATE MANAGEMENT

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

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

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

  "30s")
```

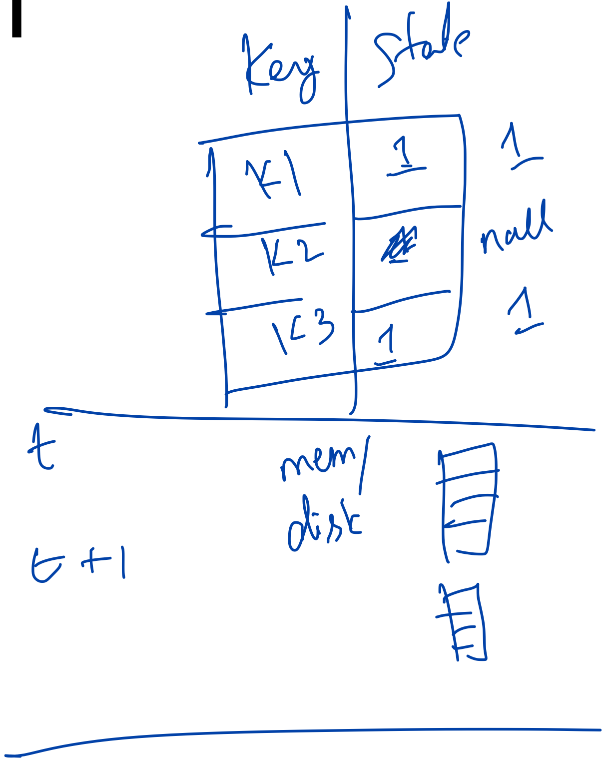
→ Initialize

if event is exit state

forget state

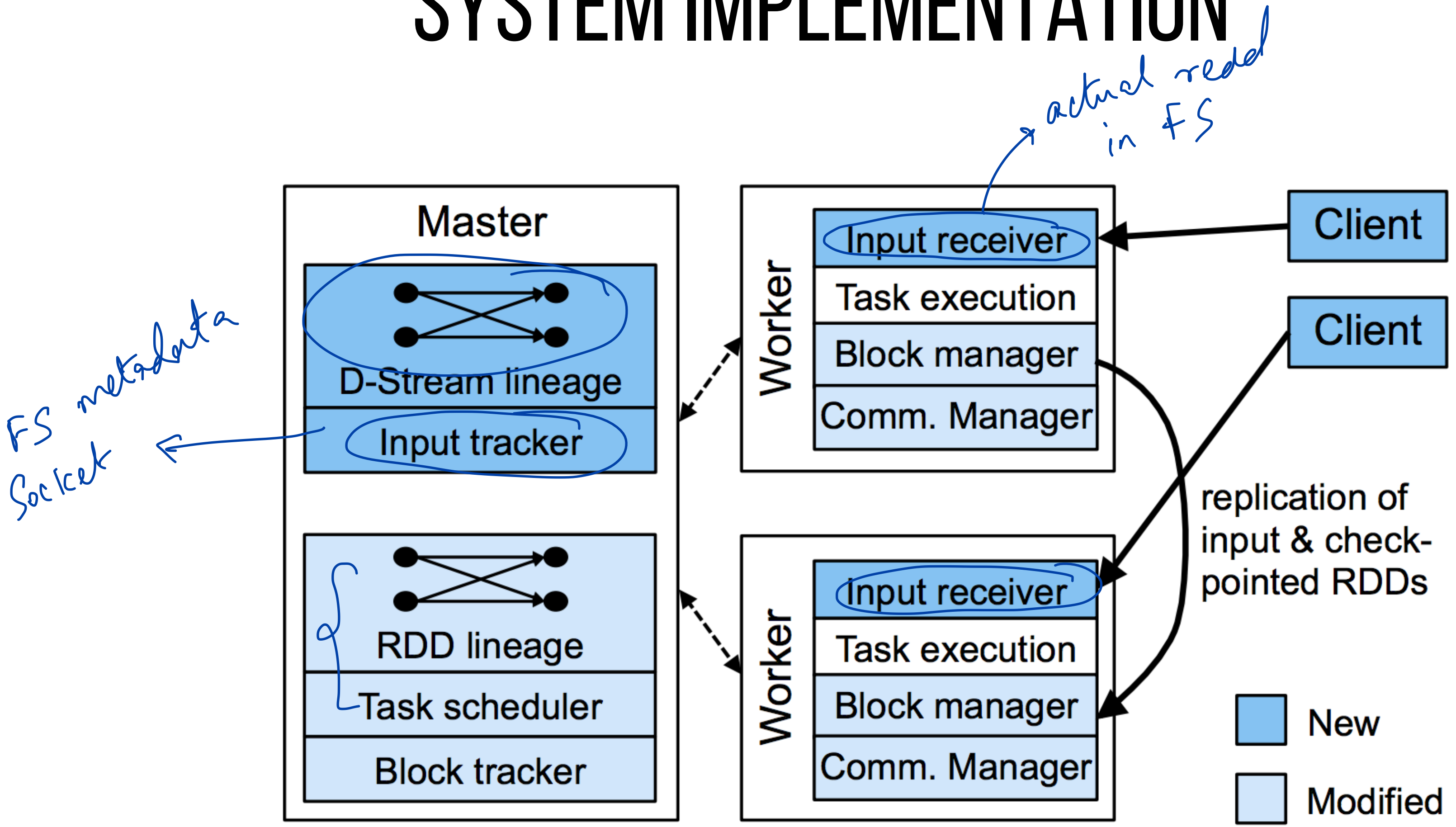
old state

new event



Key, old state, event → new state

SYSTEM IMPLEMENTATION



OPTIMIZATIONS

running { t [0, 1] → submit Job (rdd[,)

t [1, 2] → submit Job ( )

Timestep Pipelining

- No barrier across timesteps unless needed
- Tasks from the next timestep scheduled before current finishes

map t [1, 2]

Checkpointing

- Async I/O, as RDDs are immutable
- Forget lineage after checkpoint

lineage chain very long

State

window

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*

EXAMPLE

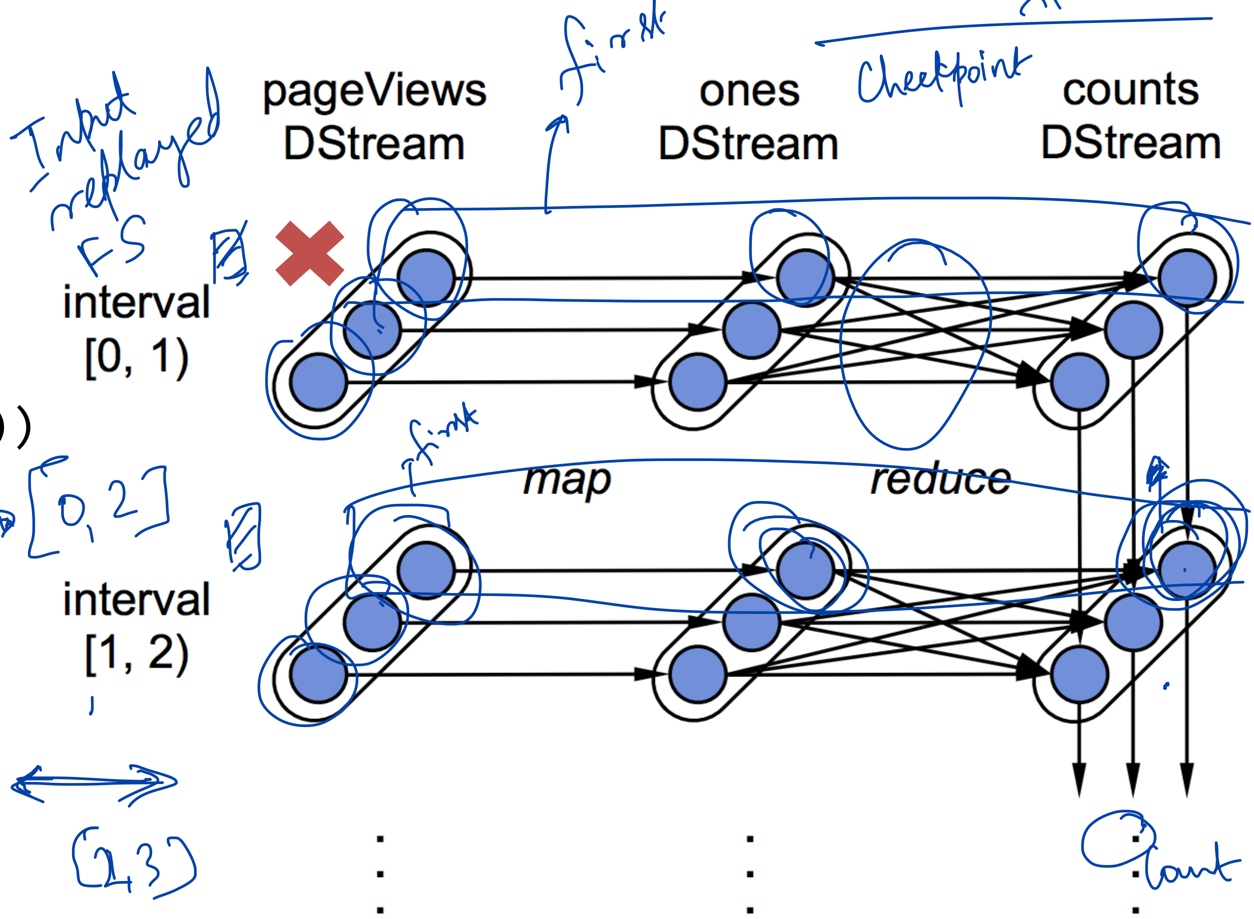
parallelism available across timesteps

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

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

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

Associative



FAULT TOLERANCE

Straggler Mitigation

Use speculative execution

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

Master Recovery NEW!

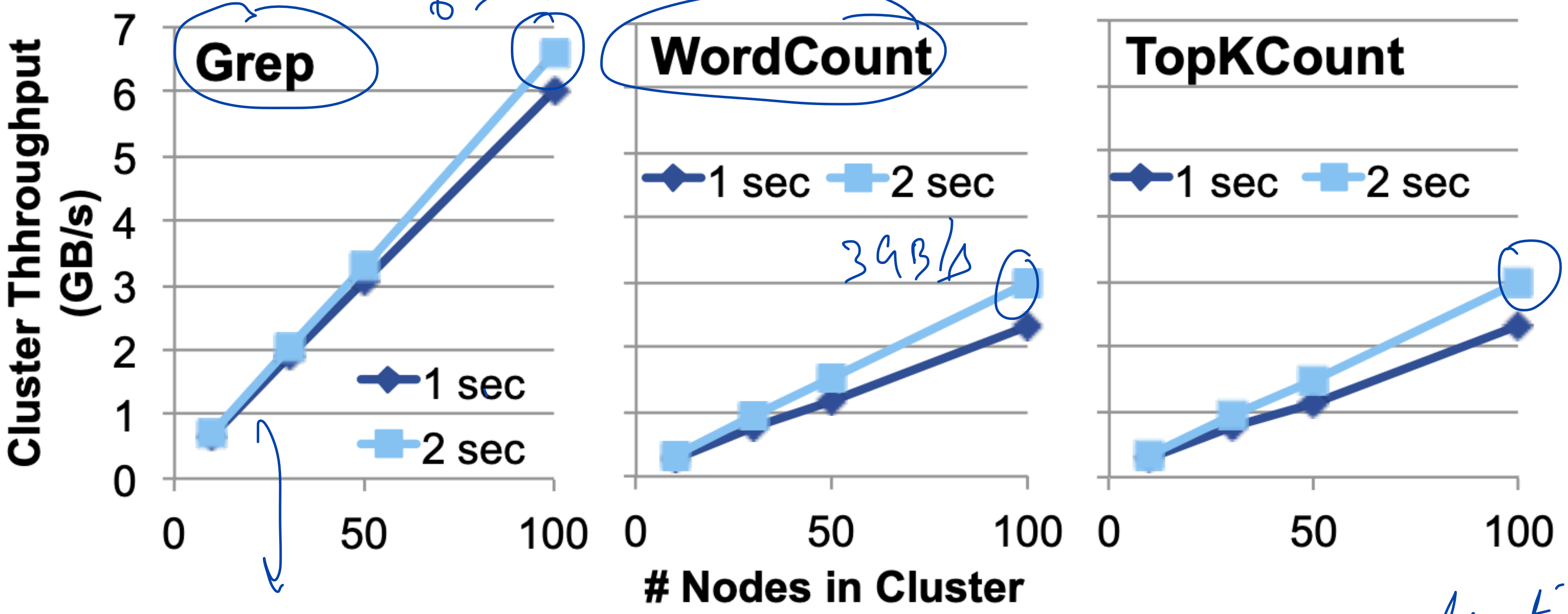
- At each timestep, save graph of DStreams and Scala function objects
- Workers connect to a new master and report their RDD partitions
- Note: No problem if a given RDD is computed twice (determinism).

DISCUSSION

https://forms.gle/xUvzC1bdV7H48mTM8

Double timestep ≠ double tput

larger window ⇒ more tput

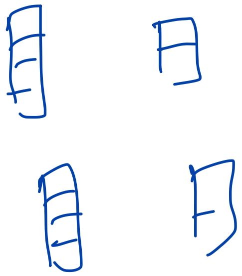


linear scaling

having no co-ordination ⇒ more tput

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

(key, count)



Combiner example

→ Throughput might decrease

↳ Overhead of creating RDDs

↓

↳ Take more time to process than 100ms?

→ Linear scaling might not work → Lots of tasks

→ Network

Driver O] very short time | Memory management] = 100ms

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 - low latency

- incremental output

* Applications that might have fewer stragglers

Spark Streaming

lower latency

recovery

API simple

NEXT STEPS

Next class: Graph processing

Sign up for project check-ins!

SHORTCOMINGS?

Expressiveness

- Current API requires users to “think” in micro-batches

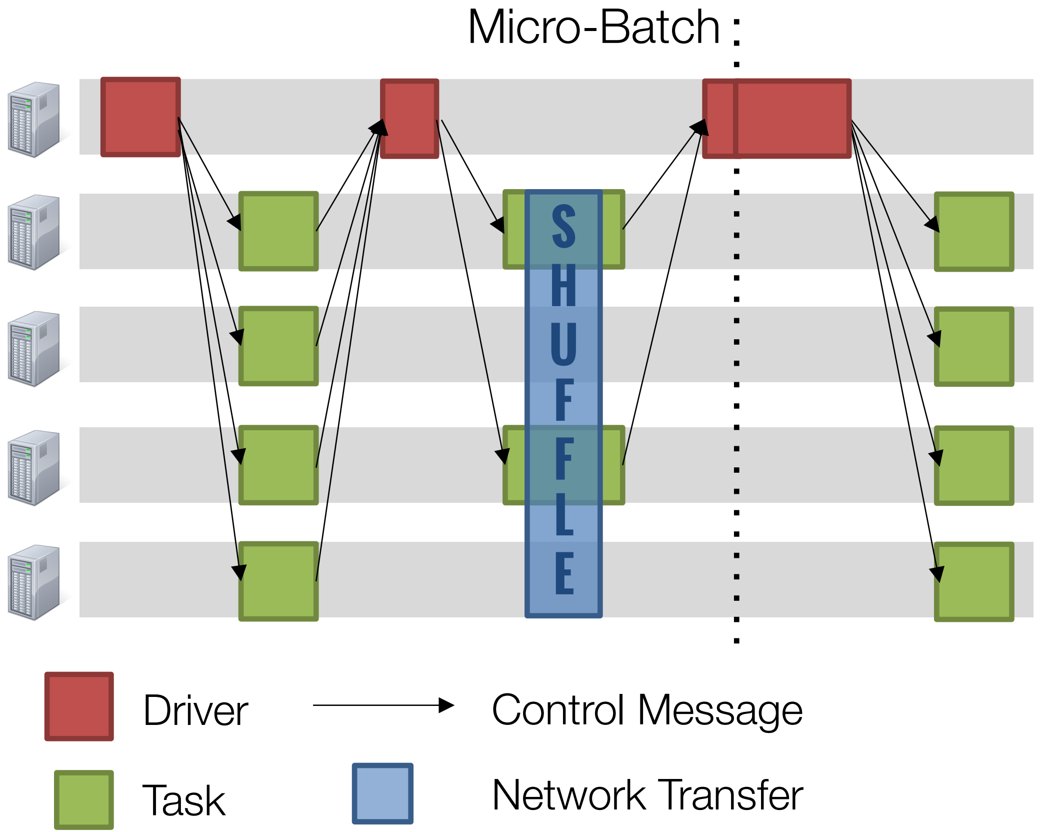
Setting batch interval

- Manual tuning. Higher batch → better throughput but worse latency

Memory usage

- LRU cache stores state RDDs in memory

COMPUTATION MODEL: MICRO-BATCHES



SUMMARY

Micro-batches: New approach to stream processing

Higher latency for fault tolerance, straggler mitigation

Unifying batch, streaming analytics