CS 744: BIG DATA SYSTEMS

Shivaram Venkataraman

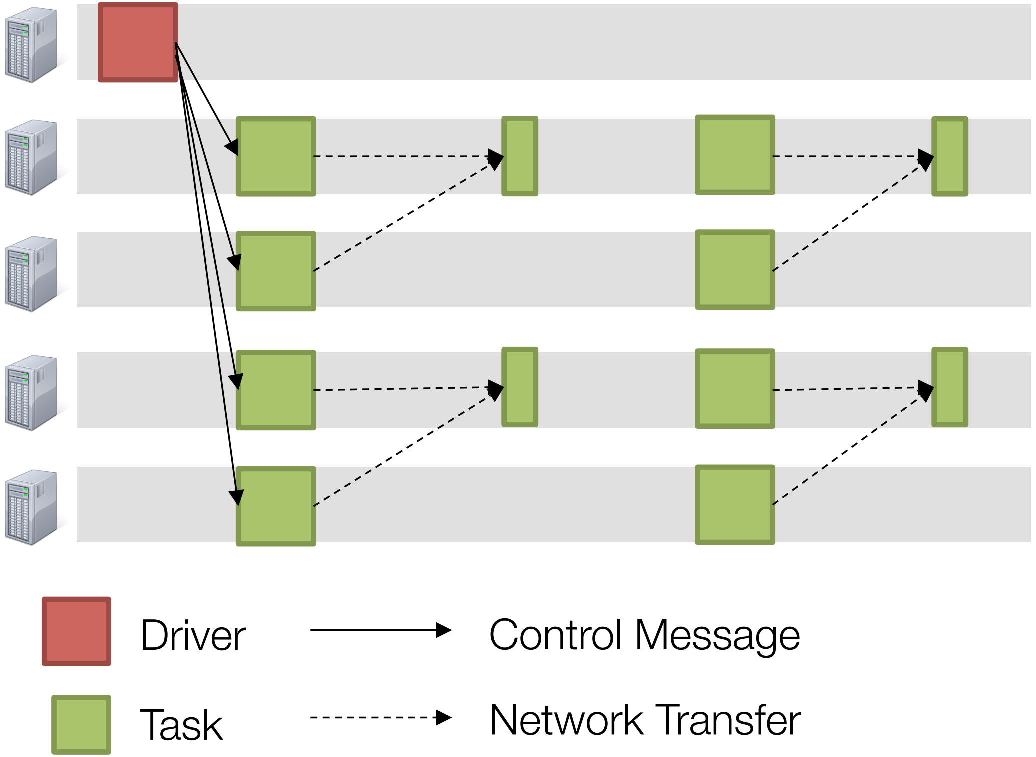
Fall 2018

ADMINISTRIVIA

- Assignment 2, Midterm grades this week
- Course Projects: round 2 meetings next Friday
- Next Tuesday: Guest speaker for first part

WHAT WE KNOW SO FAR

CONTINUOUS OPERATOR MODEL



Long-lived operators

Mutable State

Distributed Checkpoints
High overhead for Fault Recover

Stragglers ?

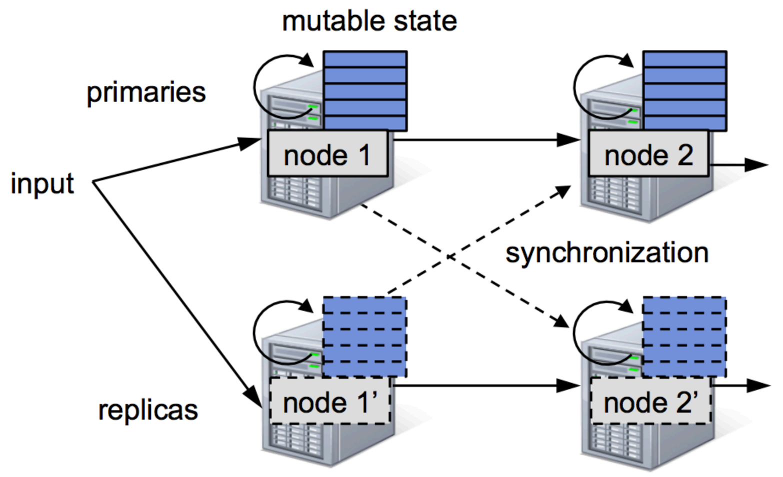
Flink

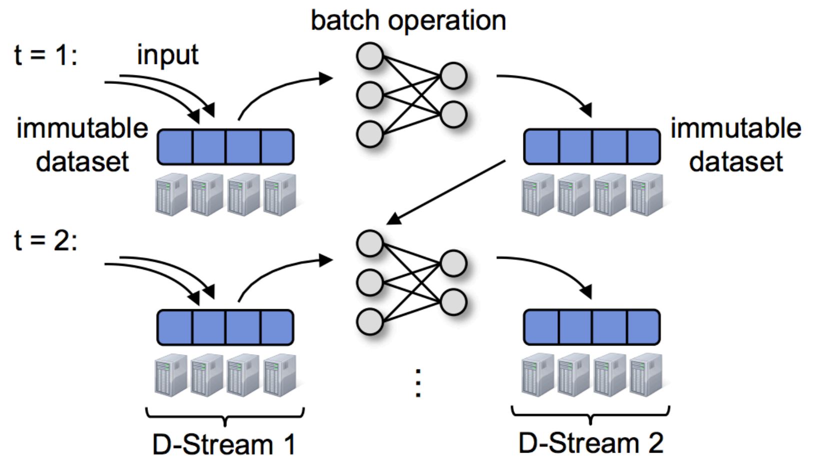
Naiad

GOALS

1. Scalability to hundreds of nodes
2. Minimal cost beyond base processing (no replication)
3. Second-scale latency
4. Second-scale recovery from faults and stragglers

DISCRETIZED STREAMS





DISCRETIZED STREAMS (DSTREAMS)

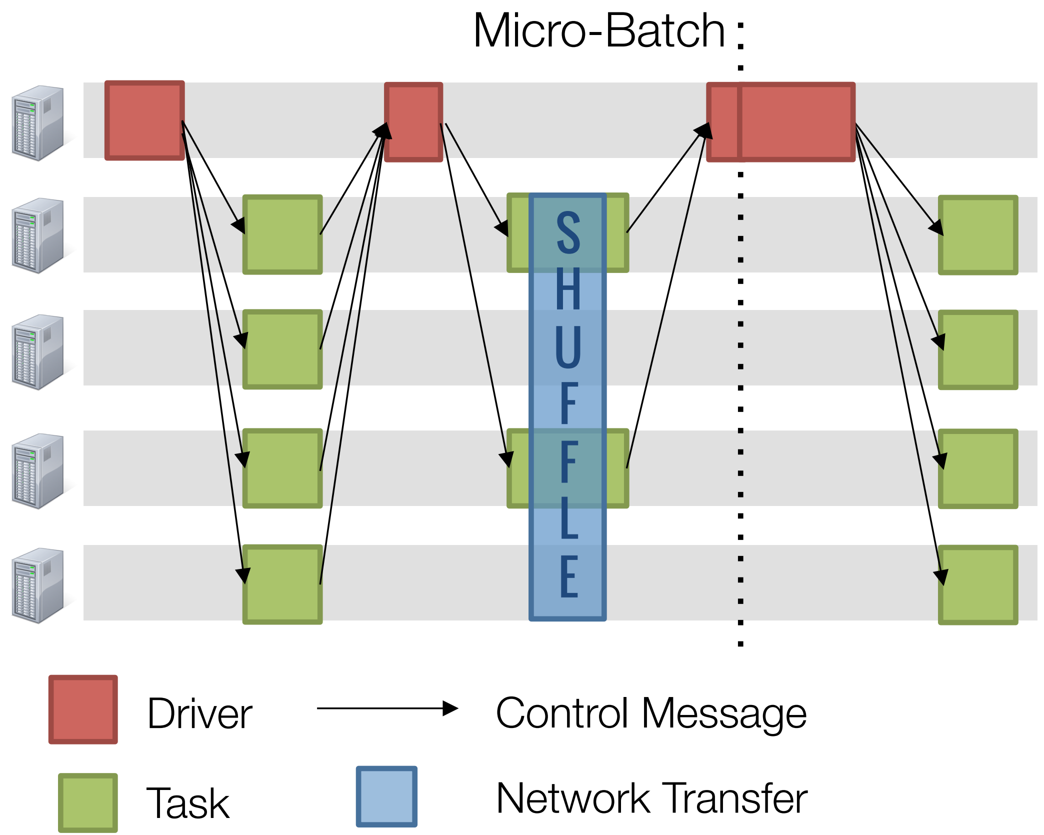
Approach

- Use *short, stateless, deterministic tasks*
- Store state across tasks as in-memory RDDs
- Fine-grained tasks → Parallel recovery / speculation

Model

- Chunk inputs into a number of micro-batches
- Processed via parallel operations (i.e., map, reduce, groupBy etc.)
- Save intermediate state as RDD / write output to external systems

COMPUTATION MODEL: MICRO-BATCHES

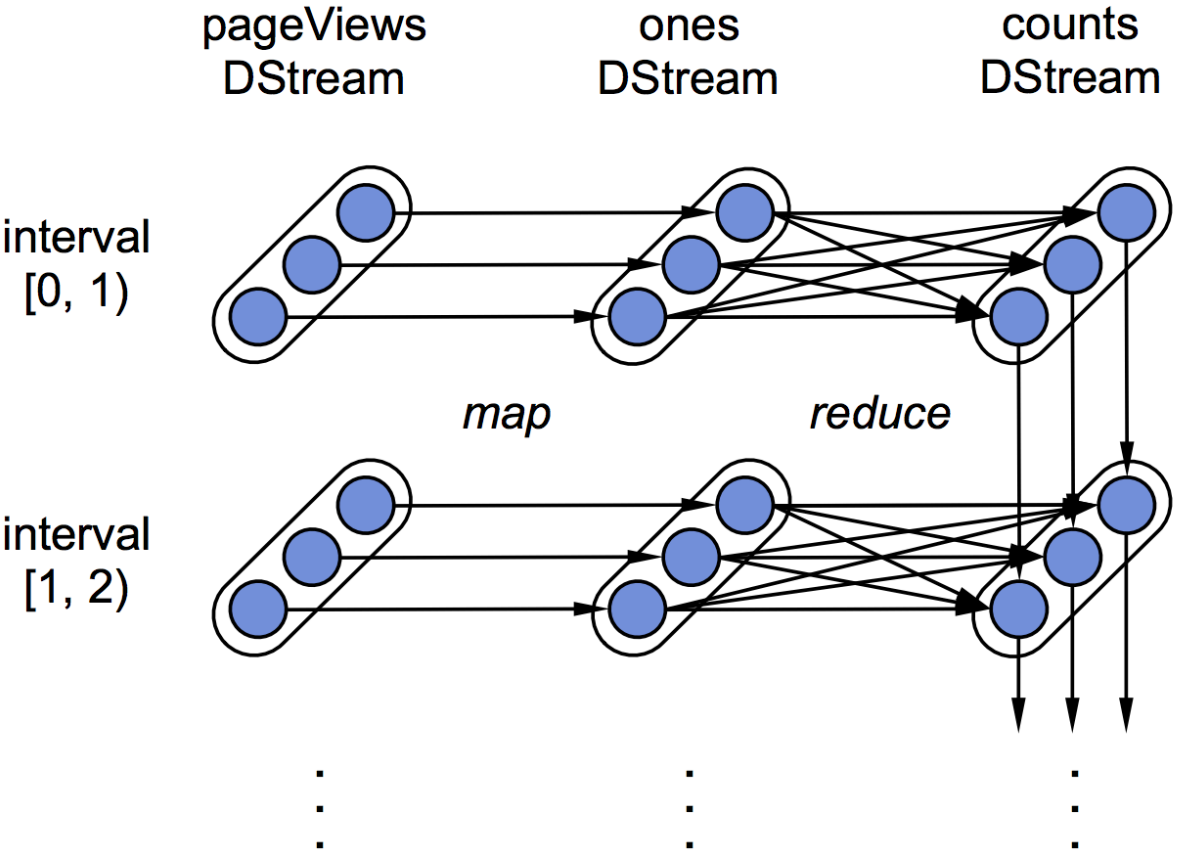


EXAMPLE

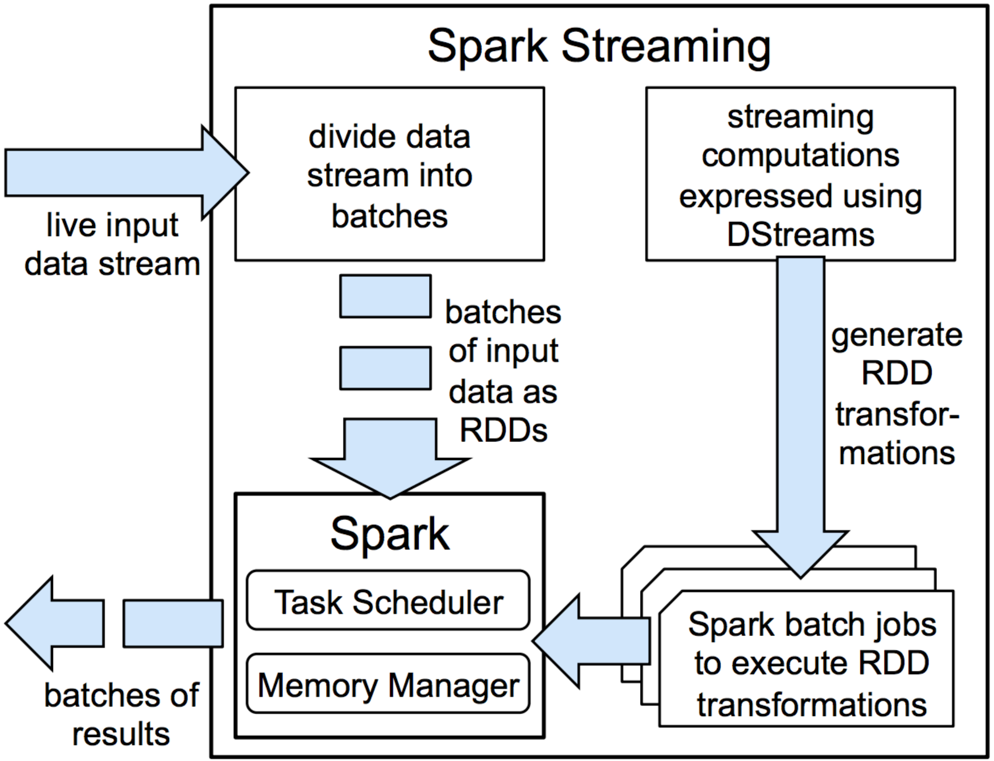
```
pageViews =
  readStream(http://...,
             "1s")

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

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



ARCHITECHTURE



DSTREAM API

Output operations

save output to external database / filesystem

Transformations

Stateless: map, reduce, groupBy, join

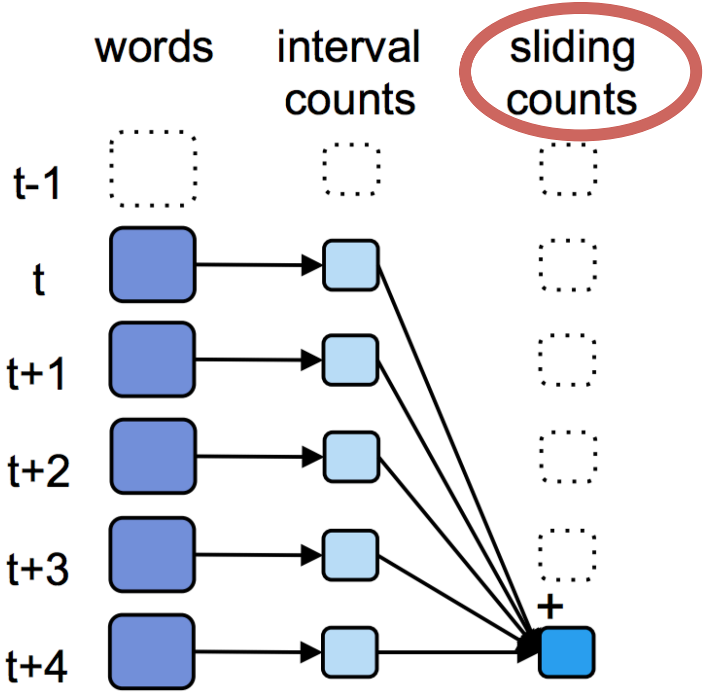
Stateful:

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

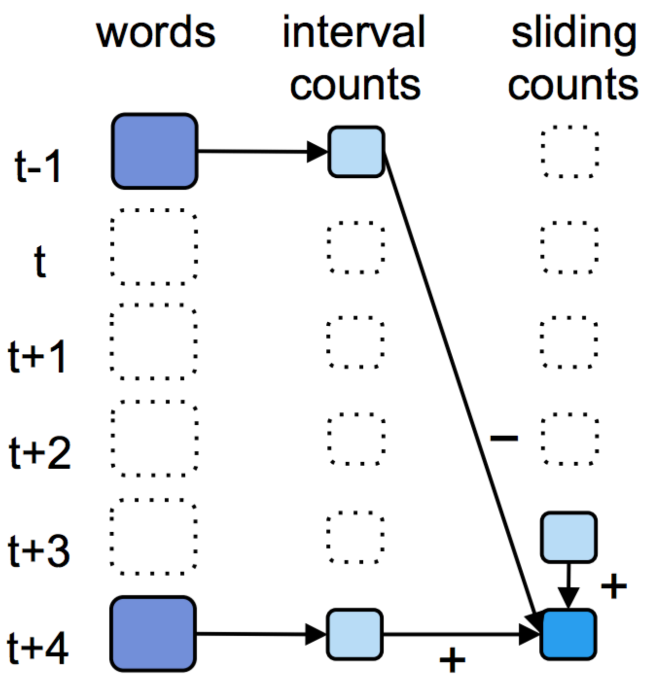
reduceByWindow("5s", (a, b) => a + b) → incremental aggregation

ASSOCIATIVE, INVERTIBLE

Add previous 5 each time



(a) Associative only



(b) Associative & invertible

Subtract previous and add current

OTHER ASPECTS

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

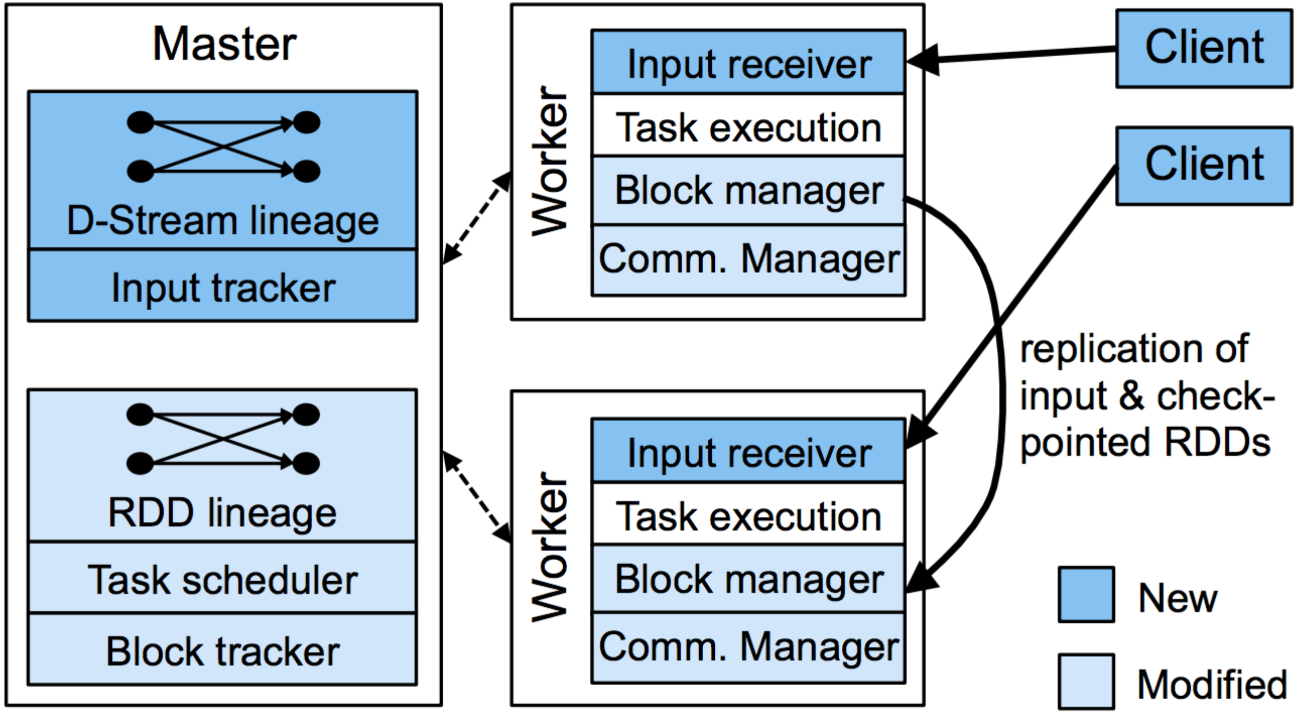
- Initialize: Create a State from the first event
- Update: Return new State given, old state and event
- Timeout for dropping old states.

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

Unifying batch and stream

- Join DStream with static RDD
- Attach console and query existing RDDs
- Shared codebase, functions etc.

SYSTEM IMPLEMENTATION



OPTIMIZATIONS

Network Communication

Rewrote Spark's data plane to use asynchronous I/O

Timestep Pipelining

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

FAULT TOLERANCE: PARALLEL RECOVERY

Worker failure

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

Strategy

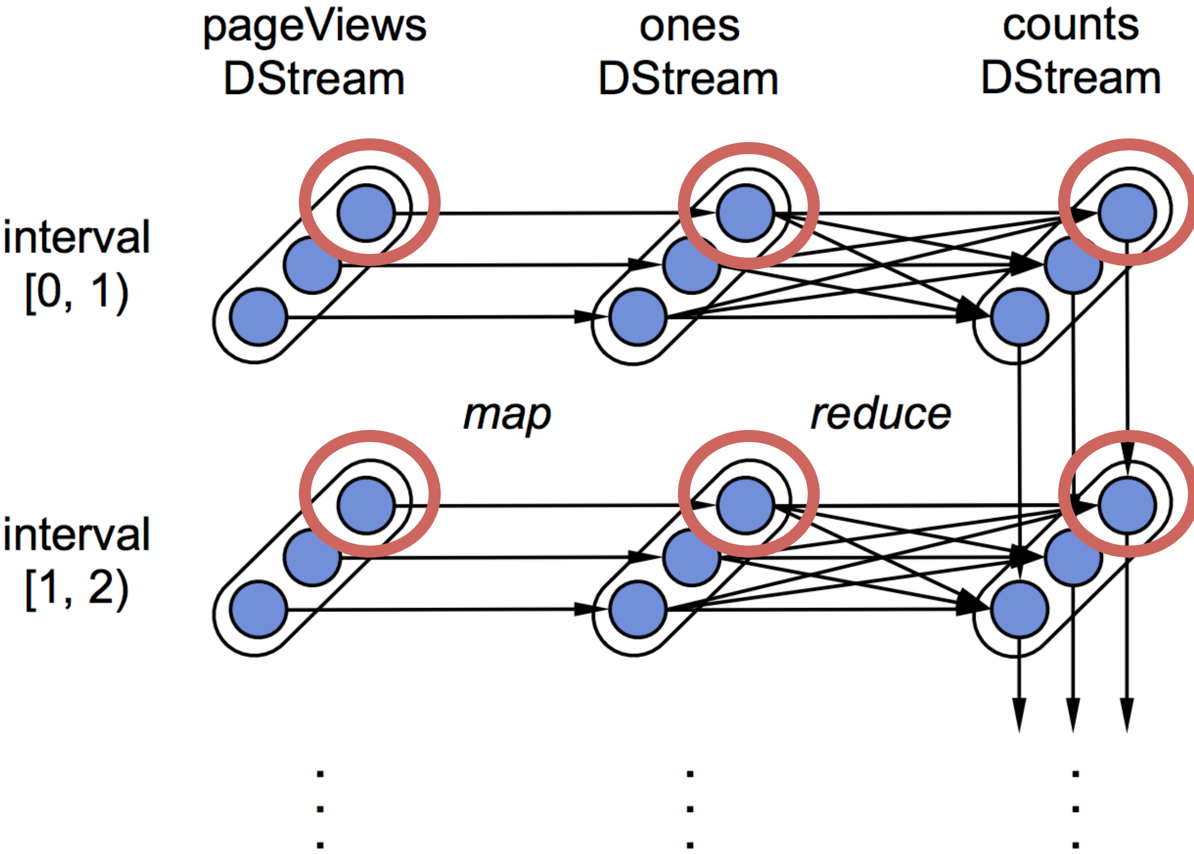
- Run all independent recovery tasks in parallel
- Parallelism from partitions *in timestep* and *across timesteps*

EXAMPLE

```
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

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

Master Recovery

- At each timestep, write out 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/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

SUMMARY

Micro-batches: New approach to stream processing

Higher latency for fault tolerance, straggler mitigation

Unifying batch, streaming analytics