Discretized Streams

Fault-Tolerant Streaming Computation at Scale

Authors: Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica

Presenter: Michael Lanthier
Slides adapted from [Tathagata Das' presentation at SOSP]

Motivation

Many big-data applications need to process large data streams in near-real time

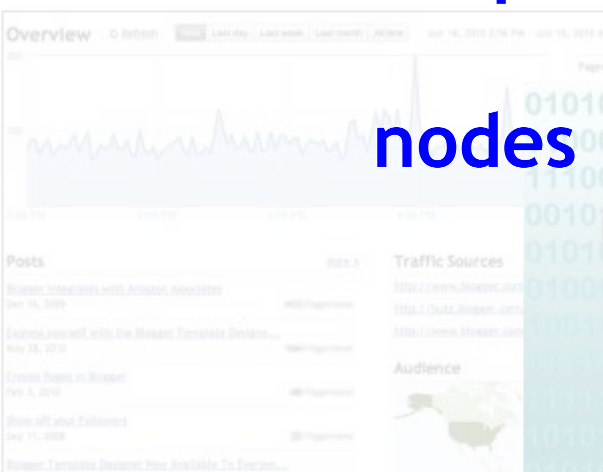
Website mon

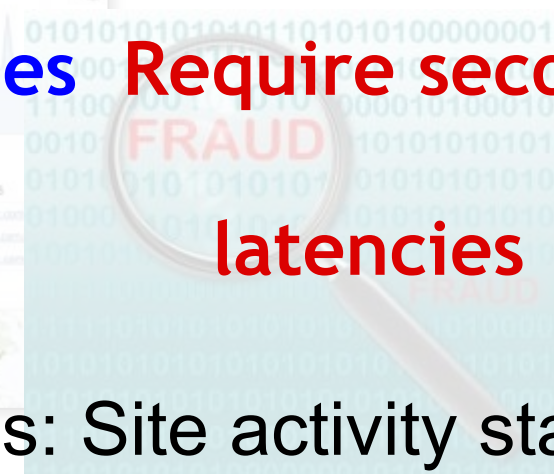
Require tens to hundreds of nodes

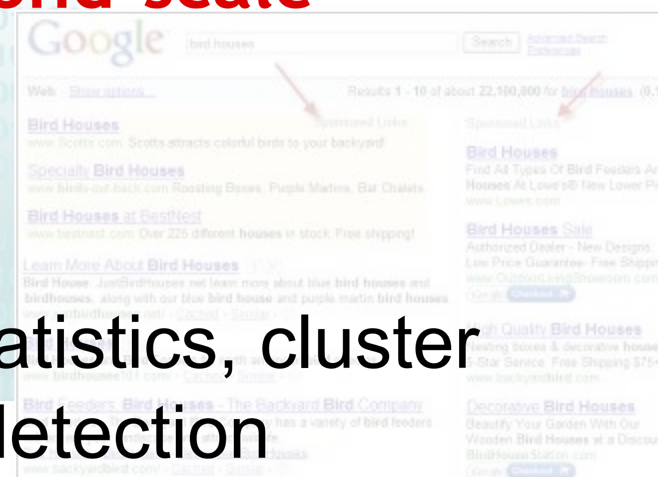
Fraud detection

Ad

Require second-scale latencies







Examples: Site activity statistics, cluster monitoring, spam detection

Challenge

- Stream processing systems must recover from failures and stragglers quickly and efficiently
  - – More important for streaming systems than batch systems

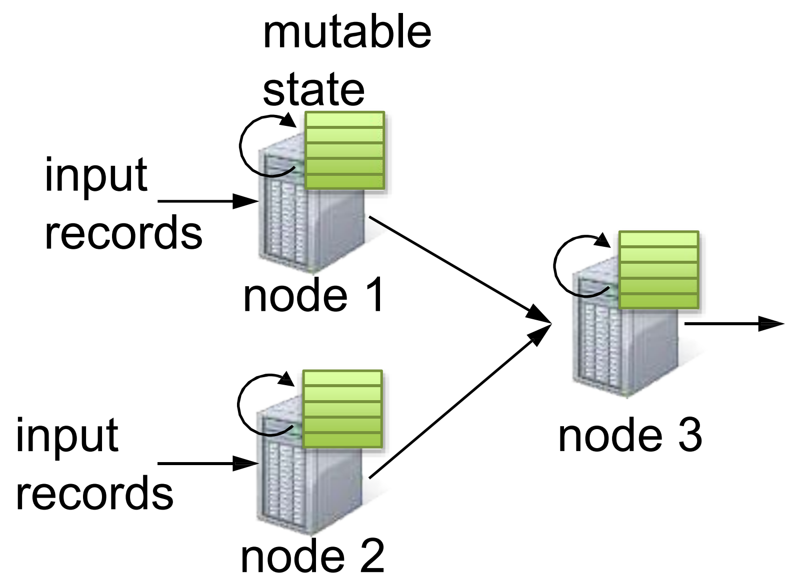
- Traditional streaming systems don't achieve these properties simultaneously

Outline

- Limitations of Traditional Streaming Systems
- Discretized Stream Processing (D-Streams)
- Spark Streaming
- Results
- Questions and Commentary

Traditional Streaming Systems

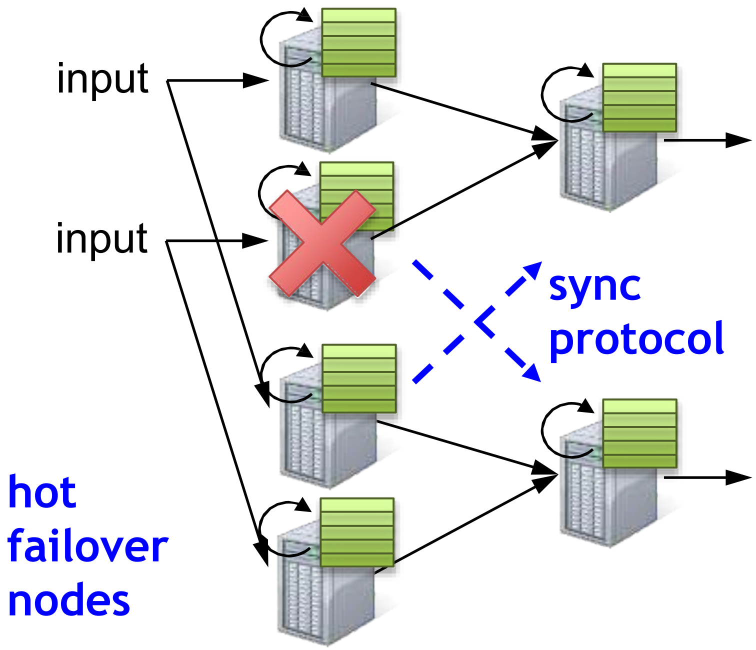
- *Continuous operator* model
  - Each node runs an operator with in-memory mutable state
  - For each input record, state is updated and new records are sent out



- Mutable state is lost if node fails
- Techniques such as node replication and upstream backup allow for fault tolerance.

Fault-tolerance in Traditional Systems

Node Replication [e.g. Borealis, Flux]



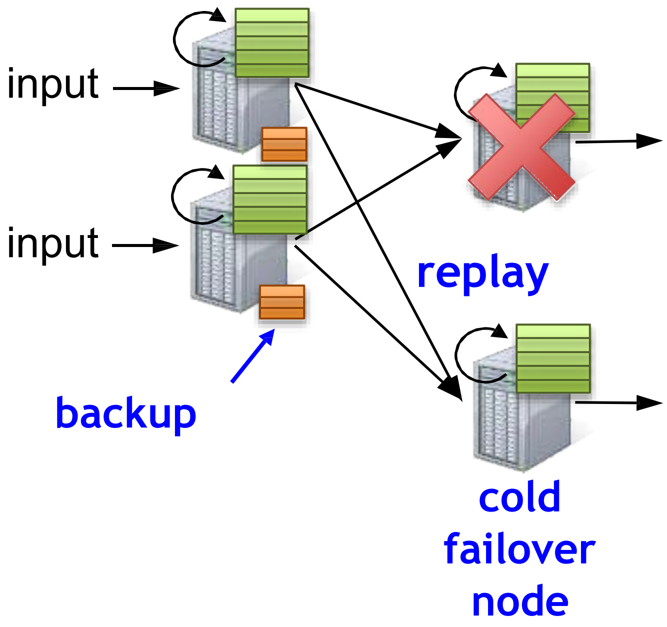
- Separate set of "hot failover" nodes
  process the same data streams
- Synchronization protocols ensures exact ordering of records in both sets
- On failure, the system switches over to the failover nodes

Fast recovery, but 2x hardware cost

Fault-tolerance in Traditional Systems

Upstream Backup [e.g. TimeStream, Storm]

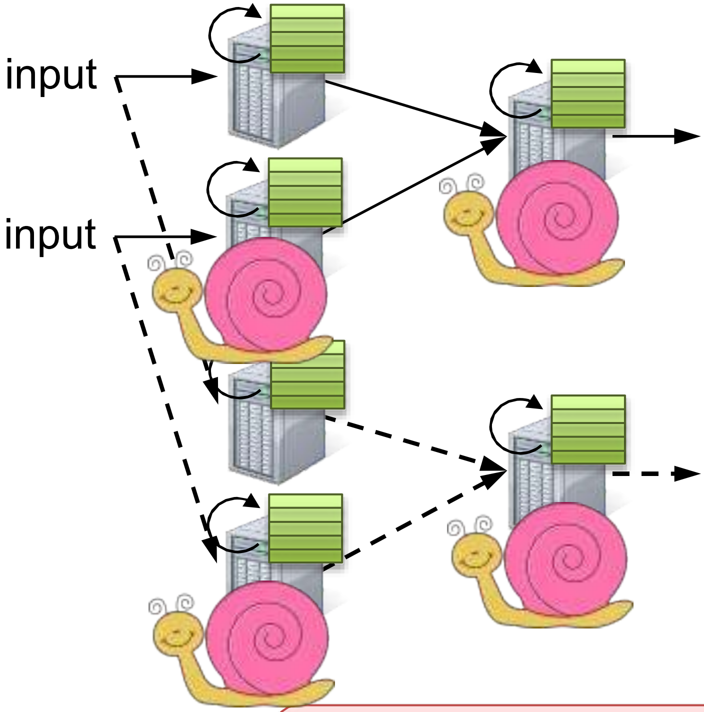
- Each node maintains backup of the forwarded records since last checkpoint
- A “cold failover” node is maintained
- On failure, upstream nodes replay the backup records *serially* to the failover node to recreate the state



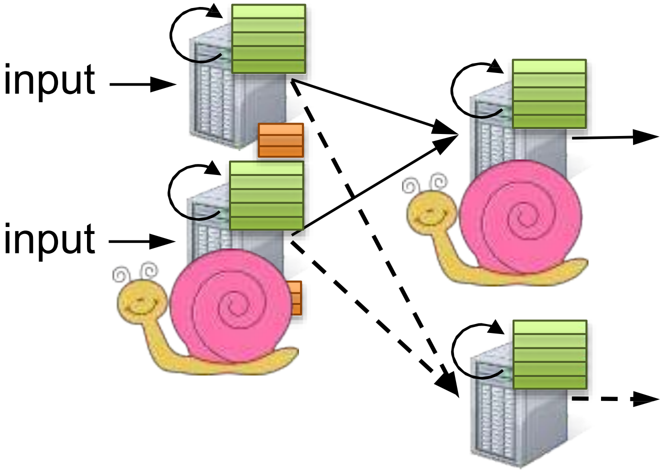
Only need 1 standby, but slow recovery

Slow Nodes in Traditional Systems

Node Replication



Upstream Backup



Neither approach handles stragglers

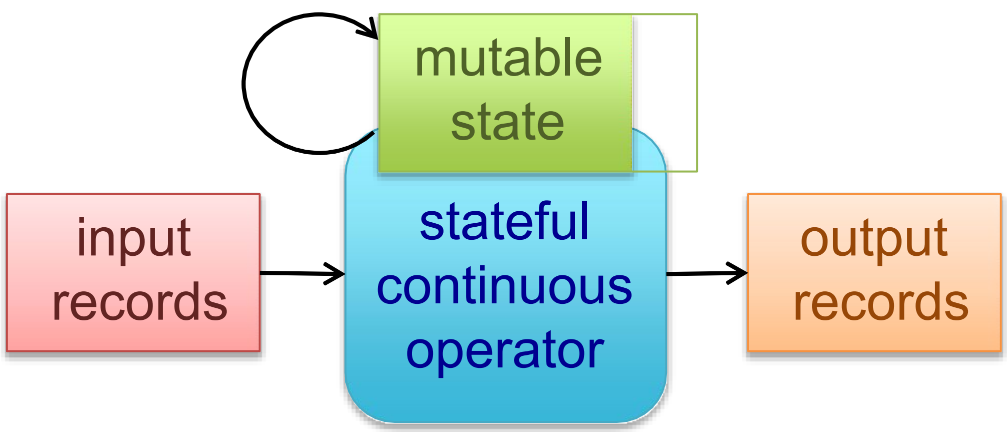
Spark Streaming's Goal

- Scales to hundreds of nodes
- Achieves second-scale latency
- Tolerate node failures and stragglers
- Sub-second fault and straggler recovery
- Minimal overhead beyond base processing

Why is it hard?

Stateful *continuous operators* tightly integrate “computation” with “mutable state”

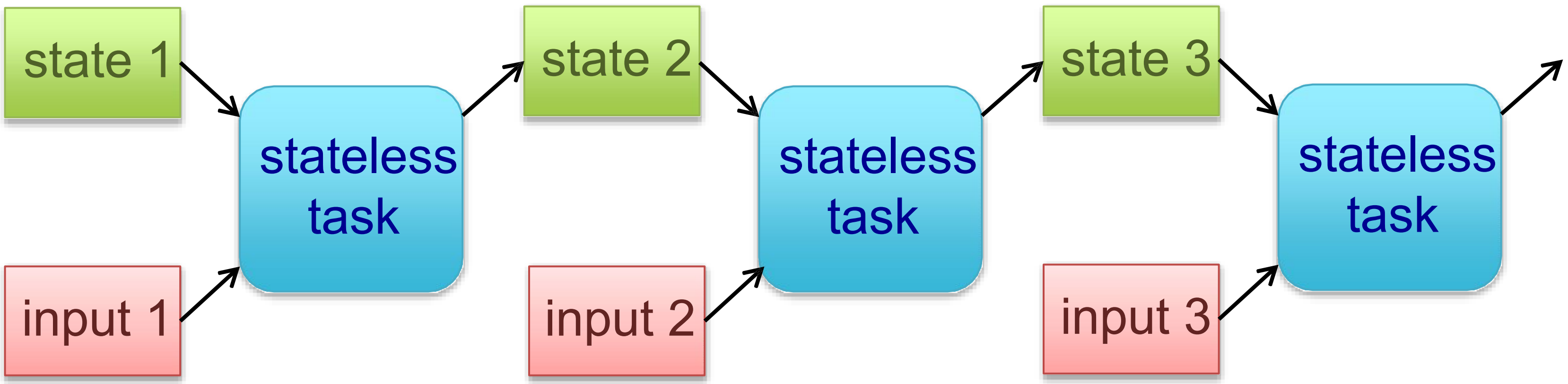
Makes it harder to define clear boundaries when computation and state can be moved around



Dissociate *computation* from *state*

- Make state *immutable* and break computation into *small*, *deterministic*, *stateless* tasks

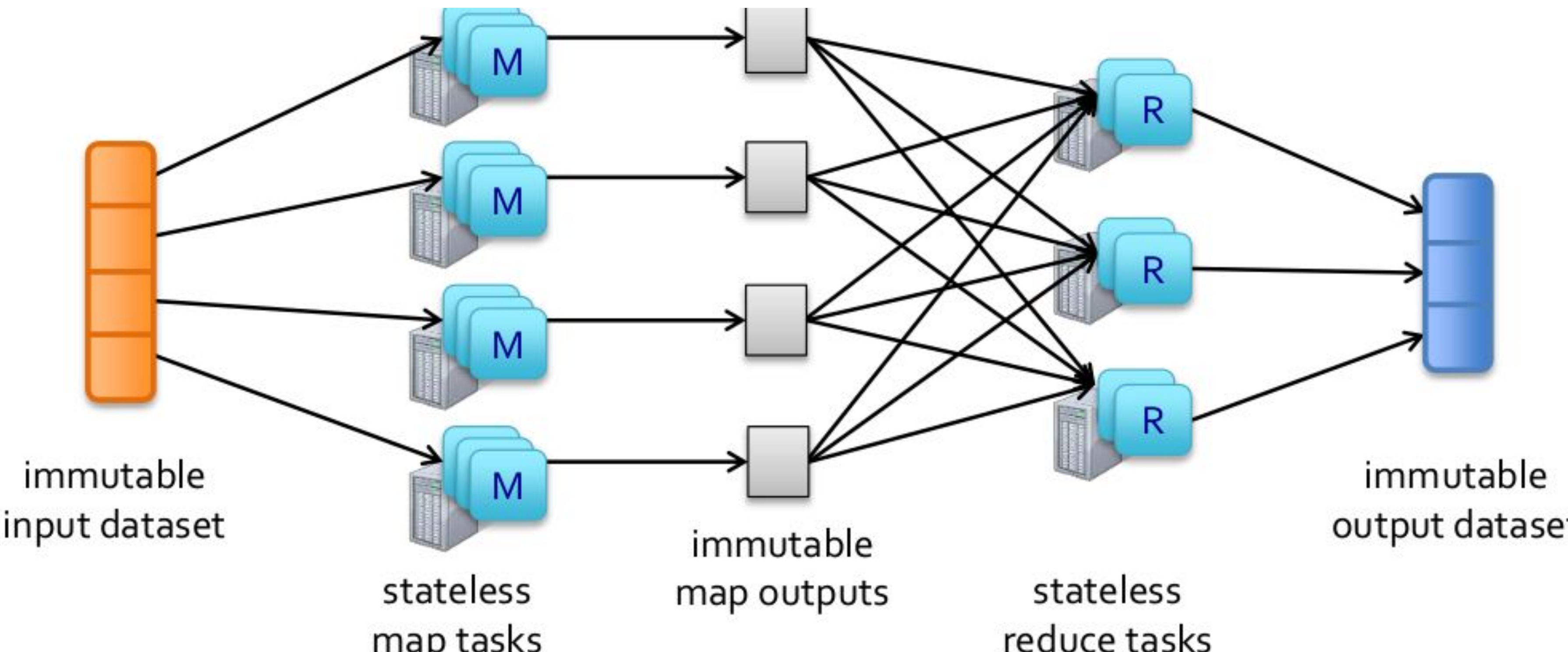
- Defines clear boundaries where state and computation can be moved around independently



Batch Processing Systems

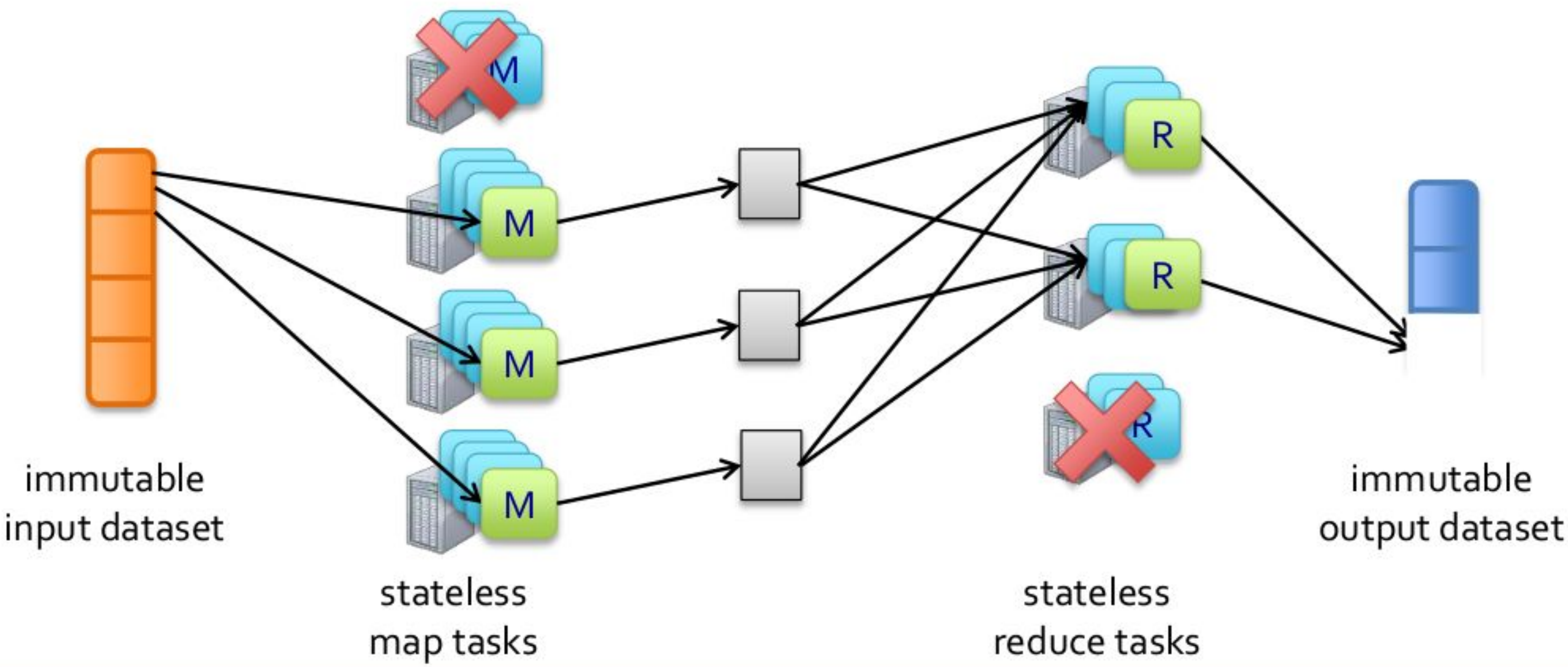
Batch processing systems like MapReduce and Spark divide

- Data into small partitions
- Jobs into small, deterministic, stateless map / reduce tasks



Parallel Recovery

Failed tasks are re-executed on the other nodes in parallel



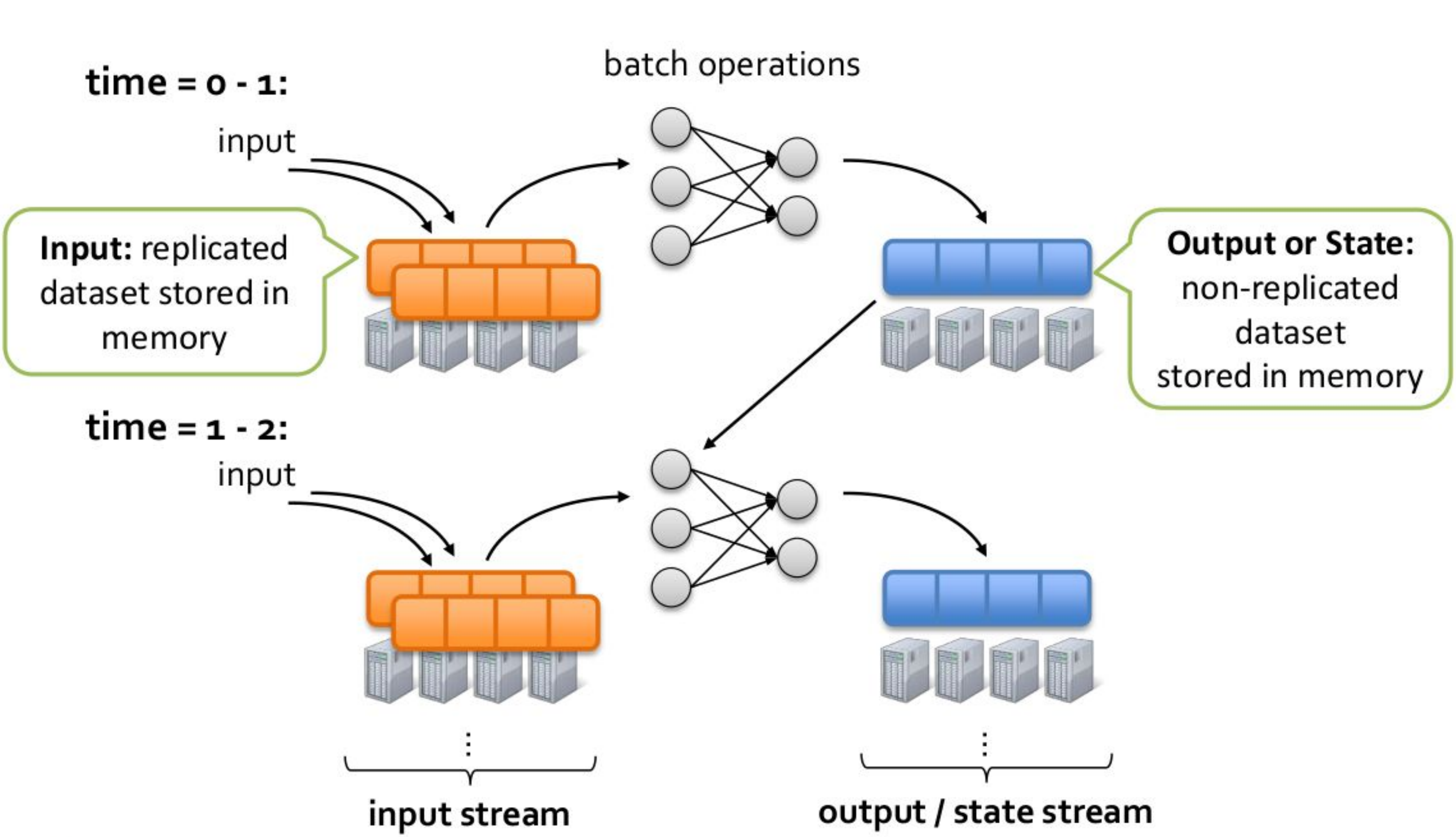
Discretized Stream Processing

Discretized Stream Processing

- Run a streaming computation as a series of small, deterministic batch jobs

- Store intermediate state data in memory

Discretized Stream Processing



Example: Counting page views

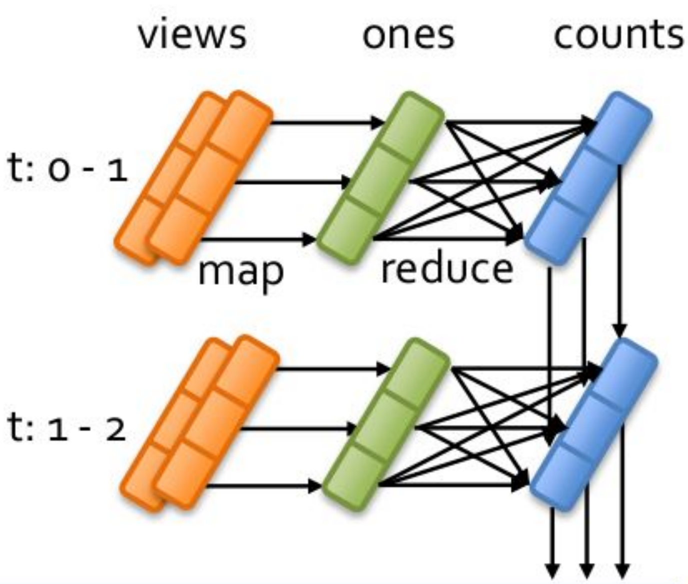
Discretized Stream (DStream) is a sequence of immutable, partitioned datasets

– Can be created from live data streams or by applying bulk, parallel transformations on other DStreams

creating a DStream

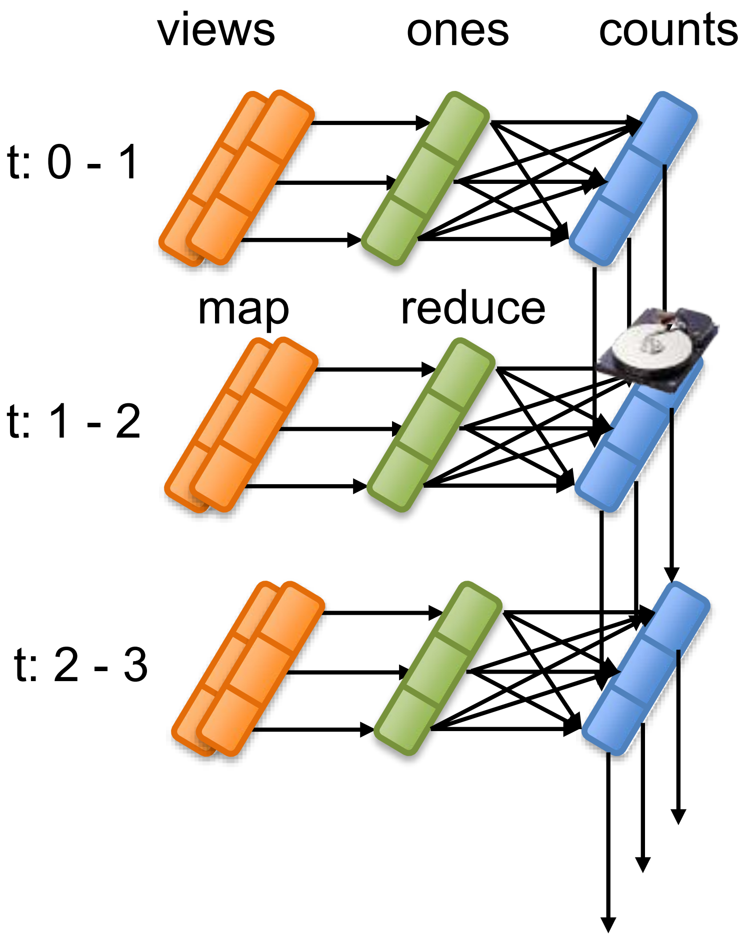
```
views = readStream("http:...", "1 sec")

ones = views.map(ev => (ev.url, 1))

counts = ones.runningReduce((x,y) => x+y)
```

transformation



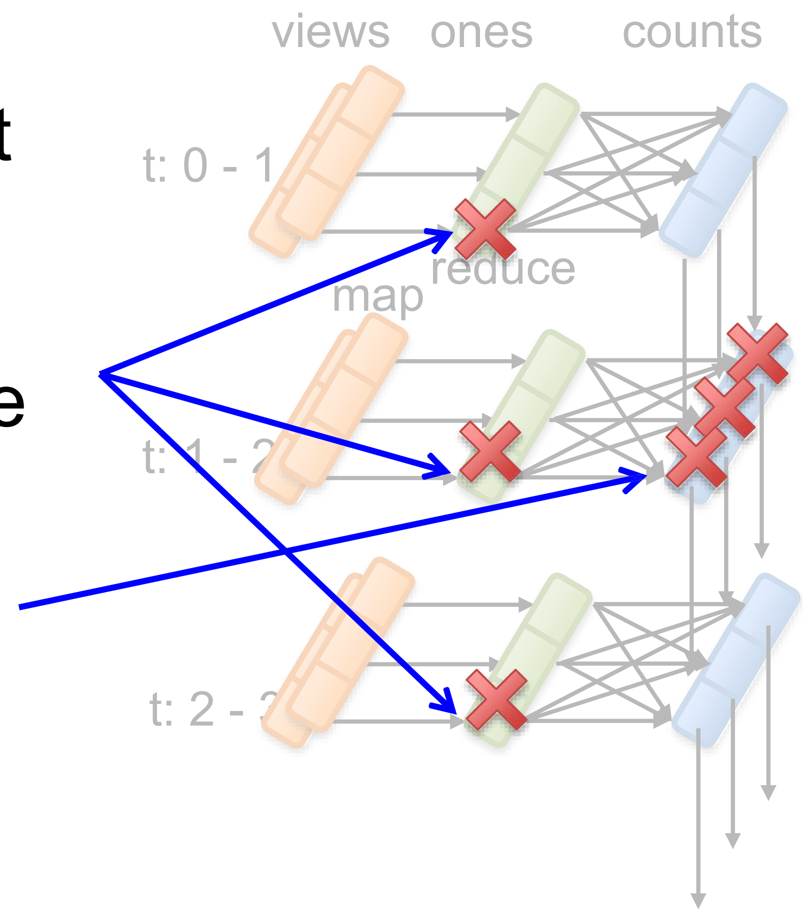
Lineage

- Datasets track operation lineage
- Datasets are periodically checkpointed asynchronously to prevent long lineages

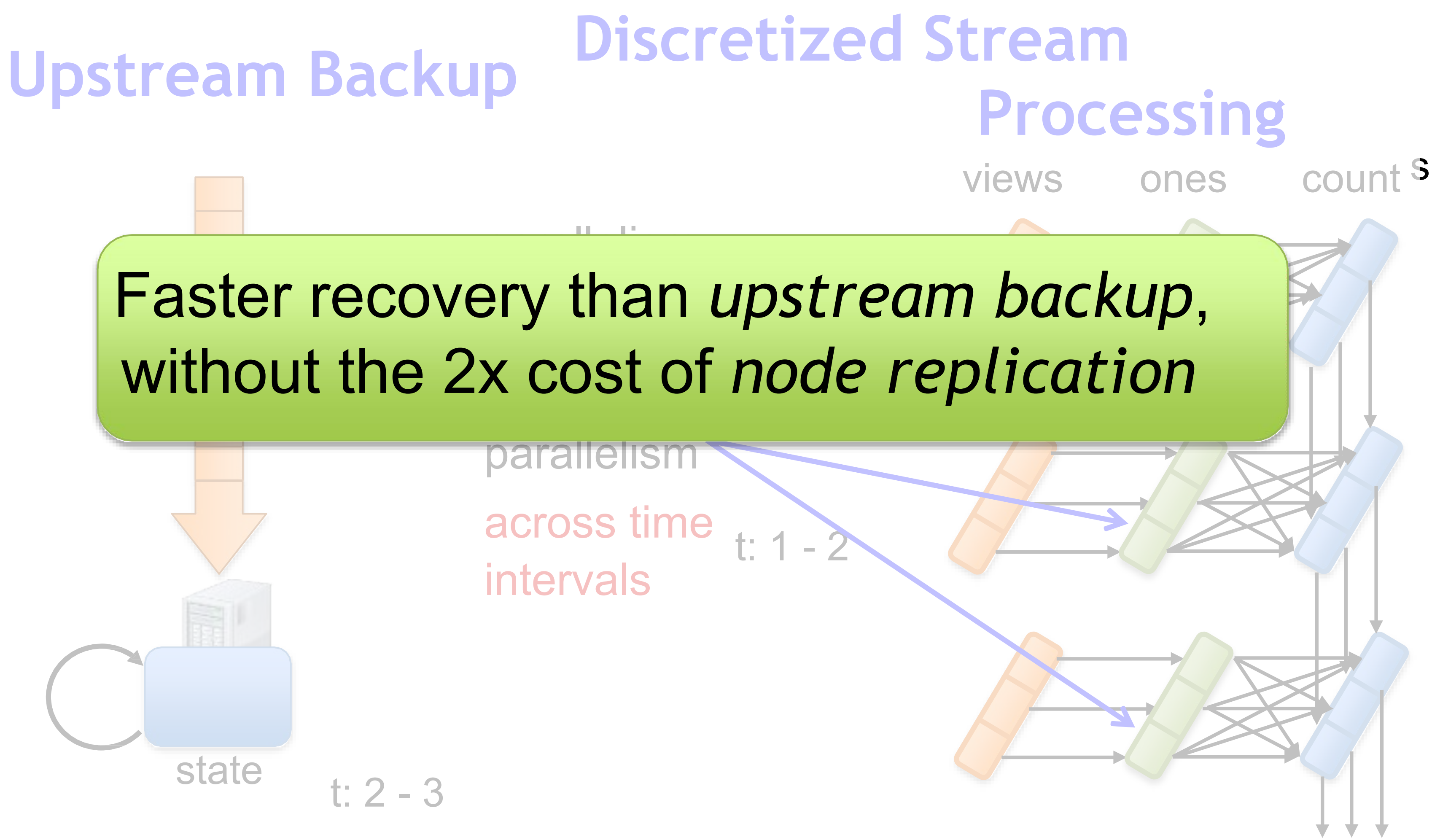


Parallel Fault Recovery

- Lineage is used to recompute partitions lost due to failures
- Datasets on different time steps recomputed in parallel
- Partitions within a dataset also recomputed in parallel



Comparison to Continuous Operators



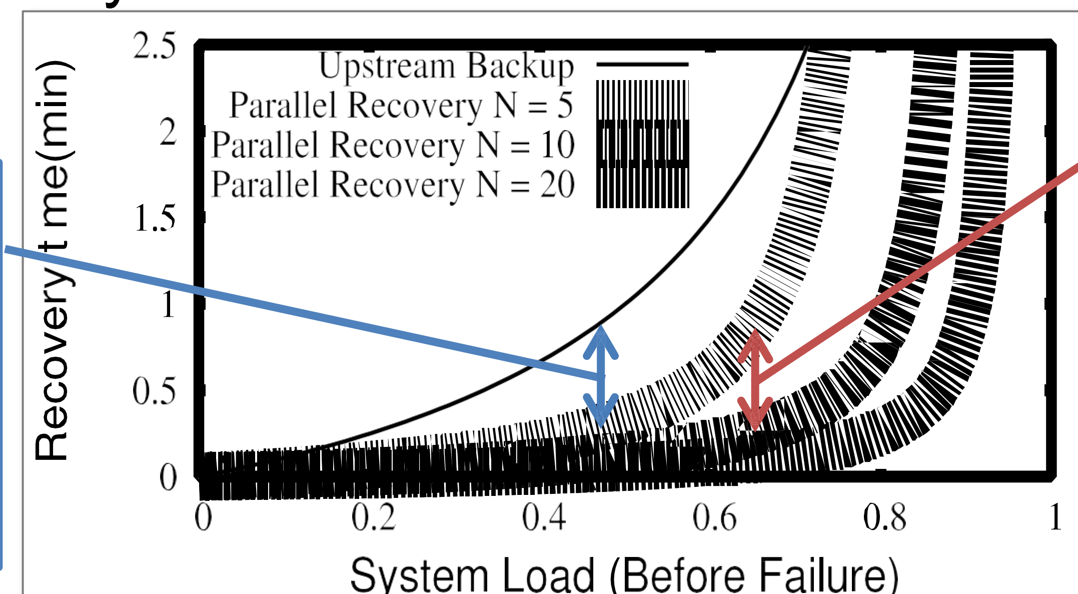
Faster recovery than *upstream backup*, without the 2x cost of *node replication*

How much faster than Upstream Backup?

Recover time = time taken to recompute and catch up

- Depends on available resources in the cluster
- Lower system load before failure allows faster recovery

Parallel recovery with 5 nodes faster than upstream backup



Parallel recovery with 10 nodes faster than 5 nodes

Parallel Straggler Recovery

- Straggler mitigation techniques
  - Detect slow tasks (e.g. 2X slower than other tasks)
  - Speculatively launch more copies of the tasks in parallel on other machines
- Masks the impact of slow nodes on the progress of the system

Handling Late Data

- System can wait to accumulate records.
  - Records can then be batched by an external timestamp.
- System can batch like normal and allow the application to deal with it.
  - Developers can apply a reduce later on to batch like timestamps together.

D-Stream Features

- Allows transformations on RDDs
- Data can output to external systems
- Windowing can be applied to group data in intervals
- Incremental aggregation can be used for operations like count and max.
- Allows for tracking of state of external objects.

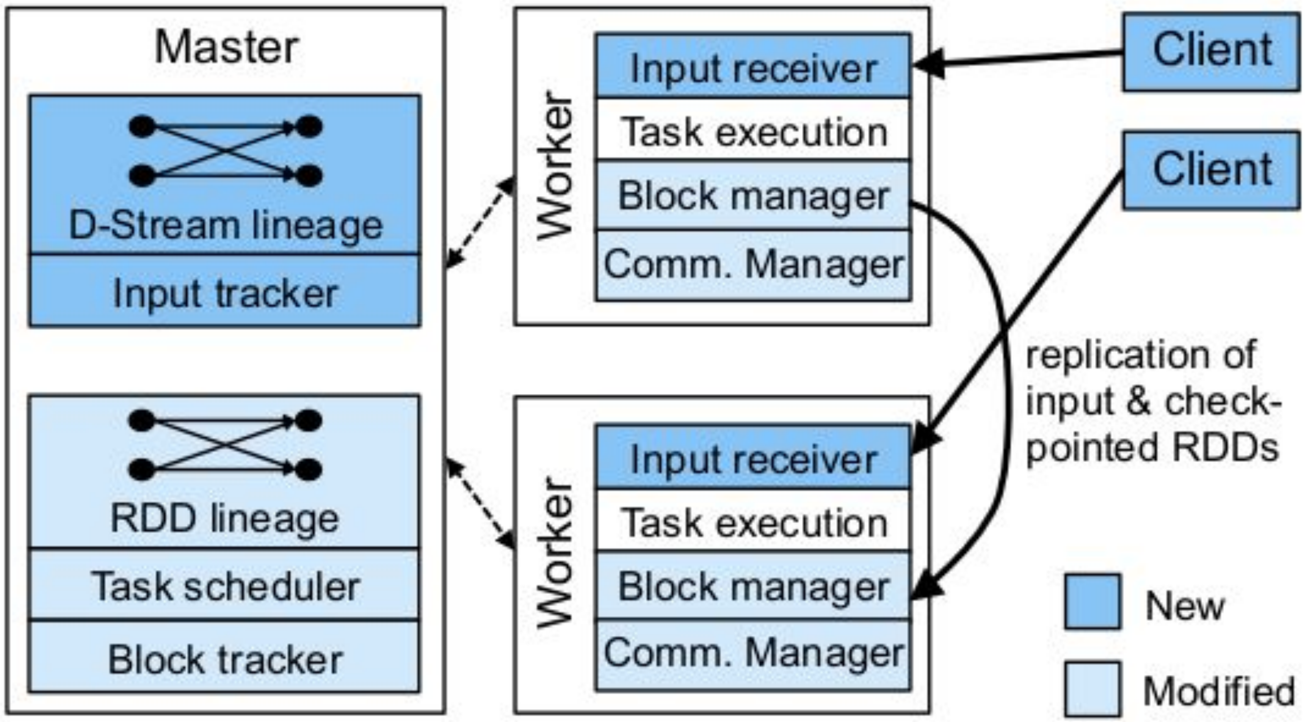
Overview

Aspect	D-Streams	Continuous proc. systems
Latency	0.5–2 s	1–100 ms unless records are batched for consistency
Consis-tency	Records processed atomically with interval they arrive in	Some systems wait a short time to sync operators before proceeding [5, 32]
Late records	Slack time or app-level correction	Slack time, out of order processing [22, 35]
Fault recovery	Fast parallel recovery	Replication or serial recovery on one node
Straggler recovery	Possible via speculative execution	Typically not handled
Mixing w/ batch	Simple unification through RDD APIs	In some DBs [14]; not in message queueing systems

Spark Streaming

- Implemented using Spark processing engine
  - Spark allows datasets to be stored in memory, and automatically recovers them using lineage

Architecture Overview



Optimizations for Streaming

- Increased ability to pipeline transformations.
- Enabled asynchronous checkpointing.
- Added master recovery.
  - State of computation written at beginning of each timestep
  - Workers connect to new master and report their RDD partitions.

Batch and Interactive Processing

- Discretized Streams creates a single programming and execution model for running streaming, batch and interactive jobs

- Combine live data streams with historic data

```
liveCounts.join(historicCounts).map(...)
```

- Interactively query live streams

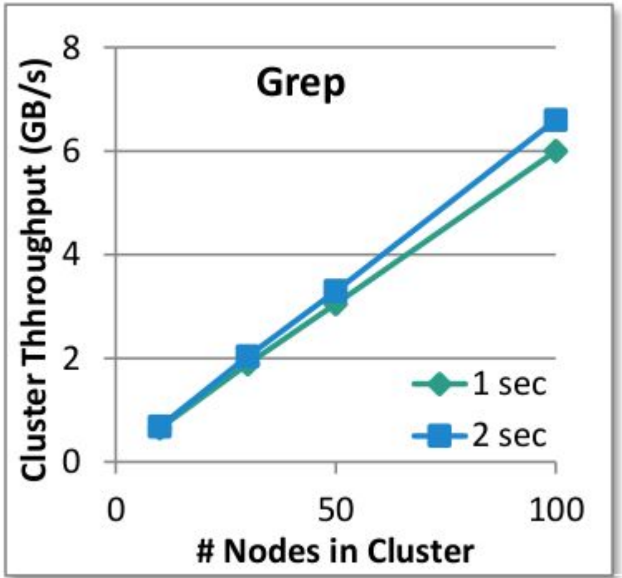
```
liveCounts.slice("21:00","21:05").count()
```

Evaluation

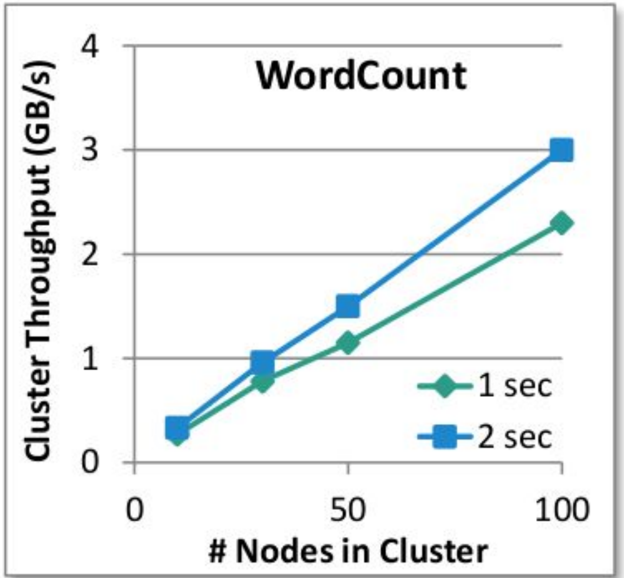
How fast is Spark Streaming?

Can process 60M records/second
on 100 nodes at 1 second latency

Tested with 100 4-core EC2 instances and 100 streams of text



Count the sentences having a keyword



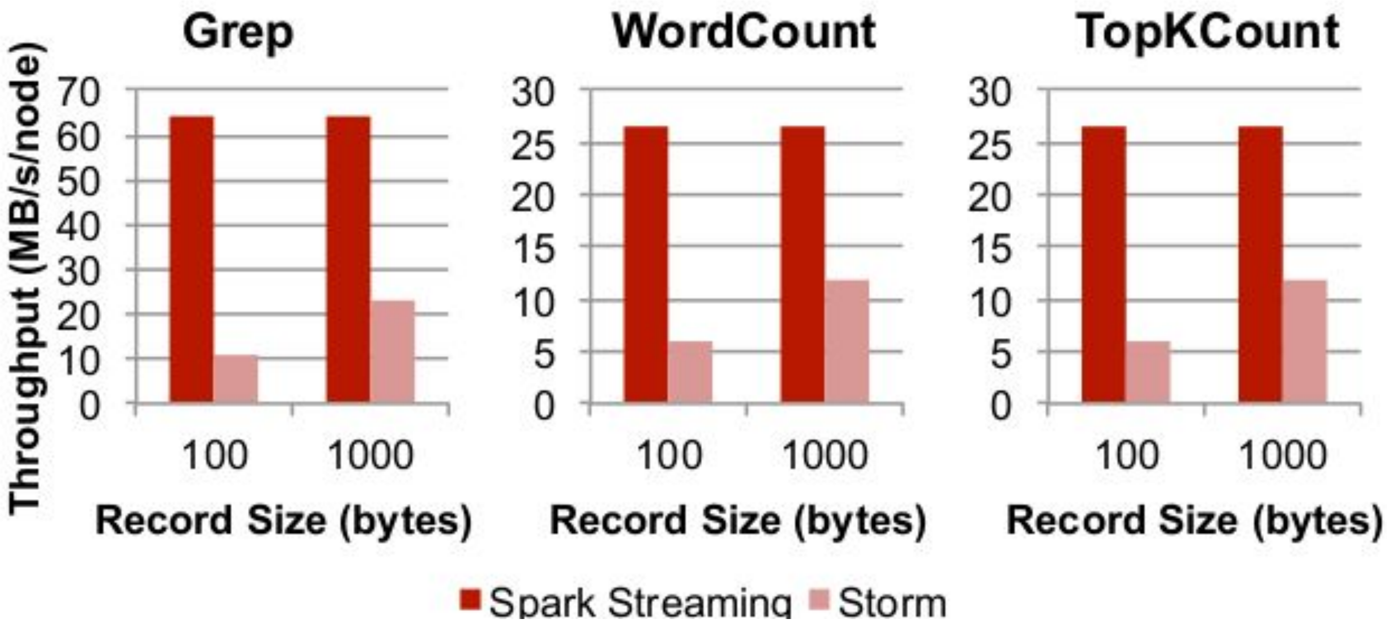
WordCount over 30 sec sliding window

How does it compare to others?

Throughput comparable to other commercial stream processing systems

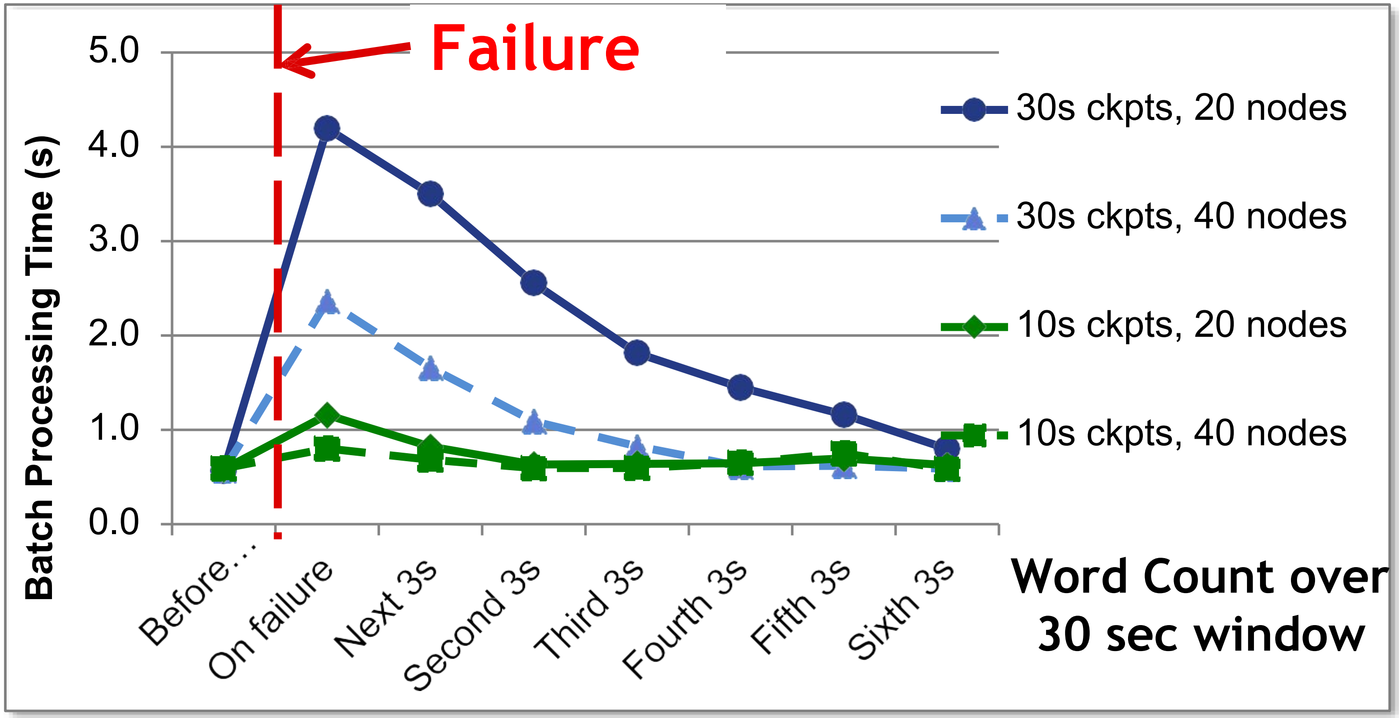
System	Throughput per core [records / sec]
Spark Streaming	160k
Oracle CEP	125k
Esper	100k
StreamBase	30k
Storm	30k

Comparison to Storm



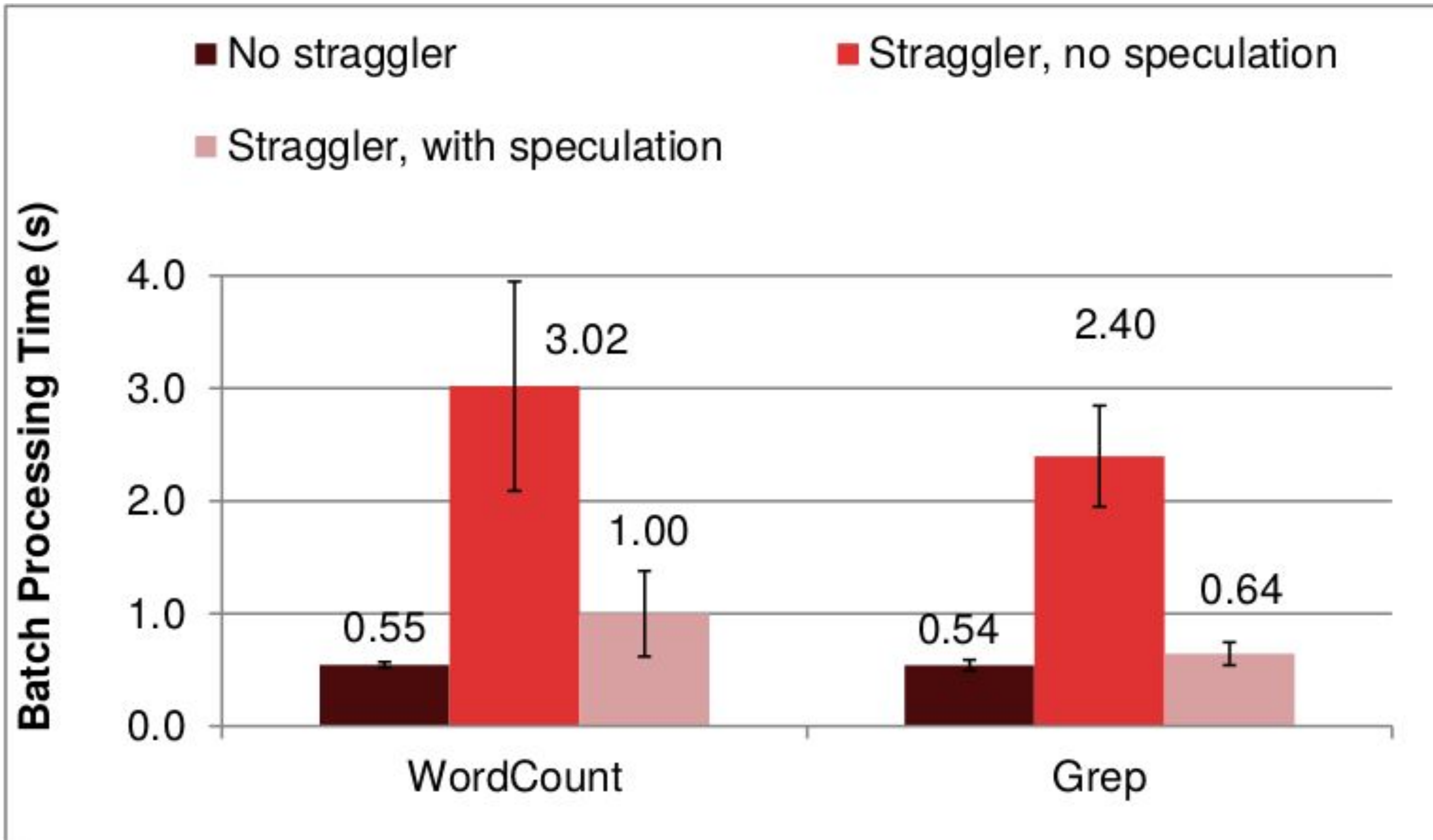
Fault Recovery

Recovery time improves with more frequent checkpointing and more nodes



Straggler Recovery

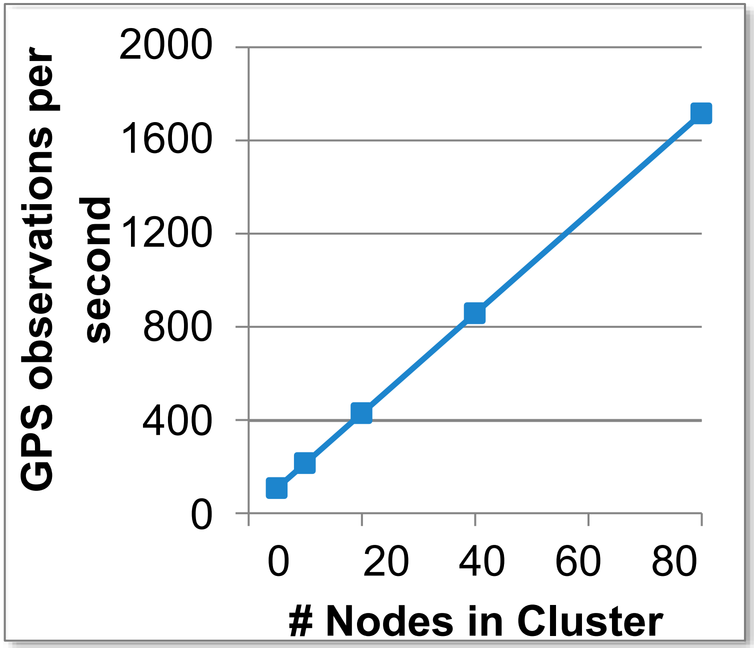
Speculative execution of slow tasks mask the effect of stragglers



App combining live + historic data

Mobile Millennium Project: Real-time estimation of traffic transit times using live and past GPS observations

- Markov chain Monte Carlo simulations on GPS observations
- Very CPU intensive
- Scales linearly with cluster size



Recent Related Work

- *Naiad* - distributed system for executing data parallel, cyclic dataflow programs.

- *Re-Stream* - Data streaming platform optimizing on energy efficiency.

- *SEEP* - Extends continuous operators to enable parallel recovery, but does not handle stragglers

Questions

- Do the metrics provided accurately represent common use cases? Are there other metrics that should have been provided?
- Could this system be effectively adapted to deal with sub second latency?
- Will the system scale linearly in a less controlled environment?

Comments

- Paper could have included more metric comparisons.
  - How does the system handle straggler and fault recovery under load?
  - What was the cost of checkpointing?
- Provided a seemingly effective solution for the problem they are trying to solve.