Discretized Streams

Fault-Tolerant Streaming Computation at Scale

Matei Zaharia, **Tathagata Das (TD)**, Haoyuan (HY) Li, Timothy Hunter, Scott Shenker, Ion Stoica

Berkeley
UNIVERSITY OF CALIFORNIA

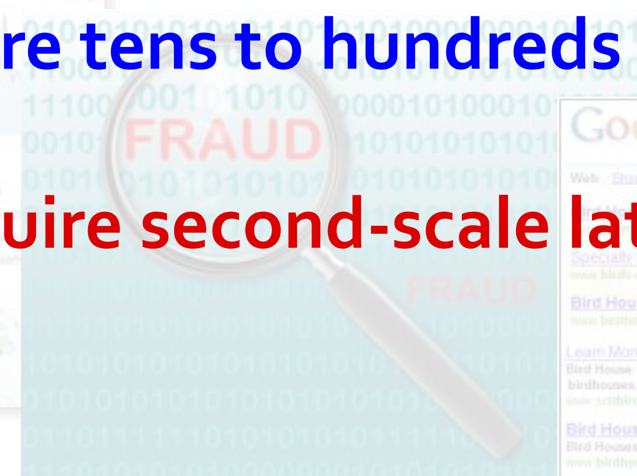
amplab

Motivation

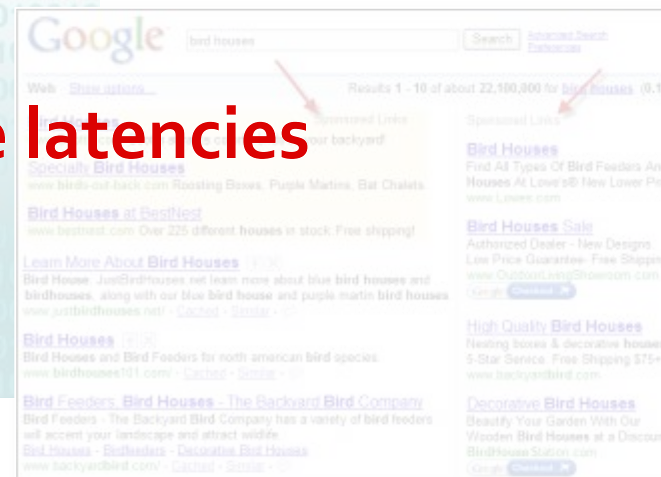
Many big-data applications need to process large data streams in near-real time

Website monitoring

Fraud detection



Ad monetization



Require tens to hundreds of nodes

Require second-scale latencies


ONE DOES NOT SIMPLY BUILD LARGE SYSTEMS
WITHOUT HANDLING FAILURES
imgflip.com

Challenge

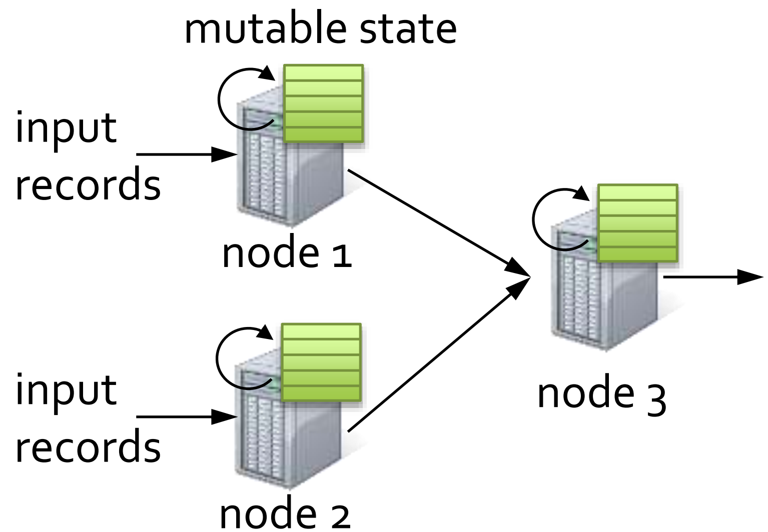
- Stream processing systems must recover from failures and stragglers quickly and efficiently
  - More important for streaming systems than batch systems
- Traditional streaming systems don't achieve these properties simultaneously

Outline

- Limitations of Traditional Streaming Systems
- Discretized Stream Processing
- Unification with Batch and Interactive Processing

Traditional Streaming Systems

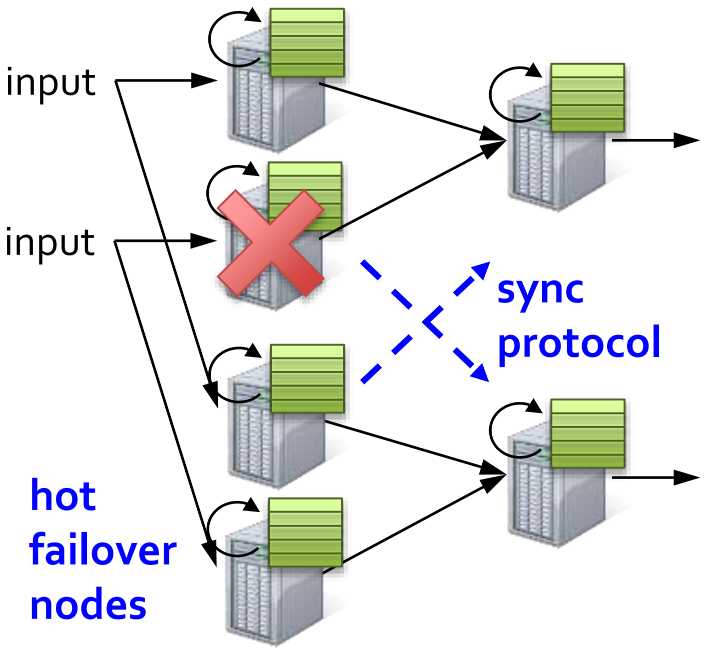
- *Continuous operator* model
  - Each node runs an operator with in-memory mutable state
  - For each input record, state is updated and new records are sent out



- Mutable state is lost if node fails
- Various techniques exist to make state fault-tolerant

Fault-tolerance in Traditional Systems

Node Replication [e.g. Borealis, Flux]



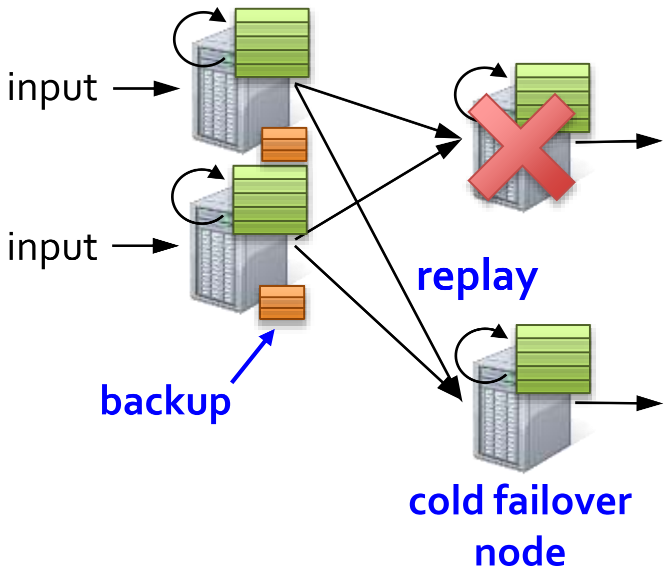
- Separate set of "hot failover" nodes process the same data streams
- Synchronization protocols ensures exact ordering of records in both sets
- On failure, the system switches over to the failover nodes

Fast recovery, but 2x hardware cost

Fault-tolerance in Traditional Systems

Upstream Backup [e.g. TimeStream, Storm]

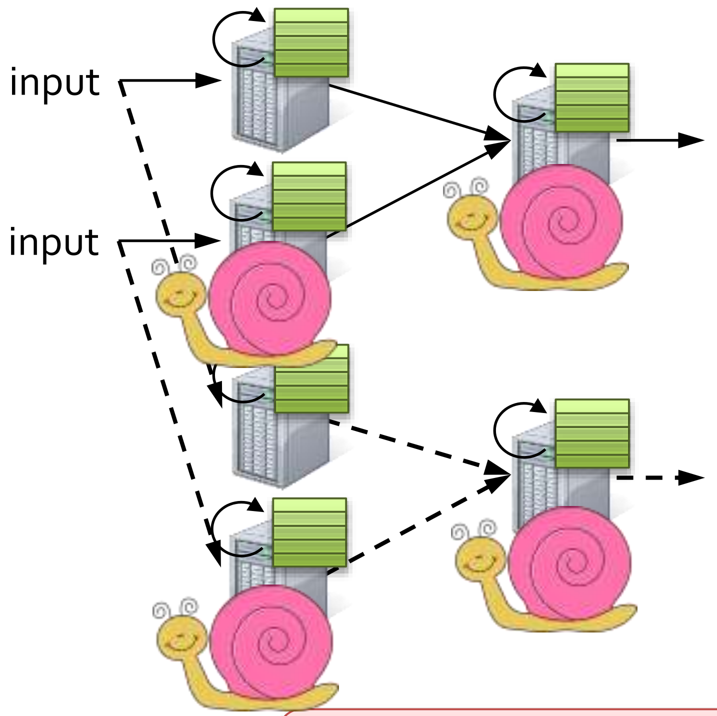
- Each node maintains backup of the forwarded records since last checkpoint
- A "cold failover" node is maintained
- On failure, upstream nodes replay the backup records *serially* to the failover node to recreate the state



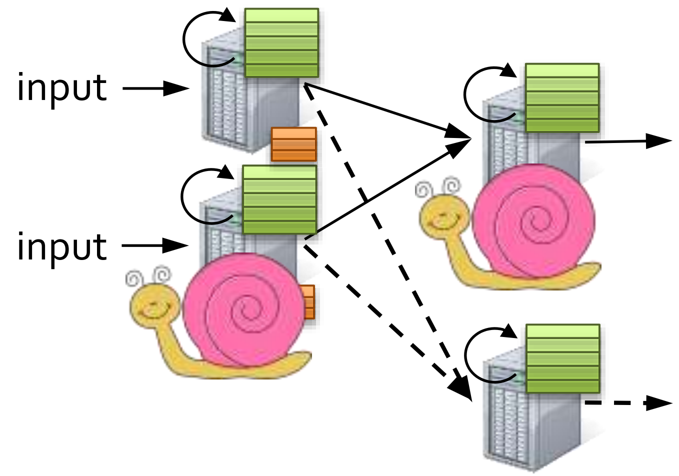
Only need 1 standby, but slow recovery

Slow Nodes in Traditional Systems

Node Replication



Upstream Backup



Neither approach handles stragglers

Our 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

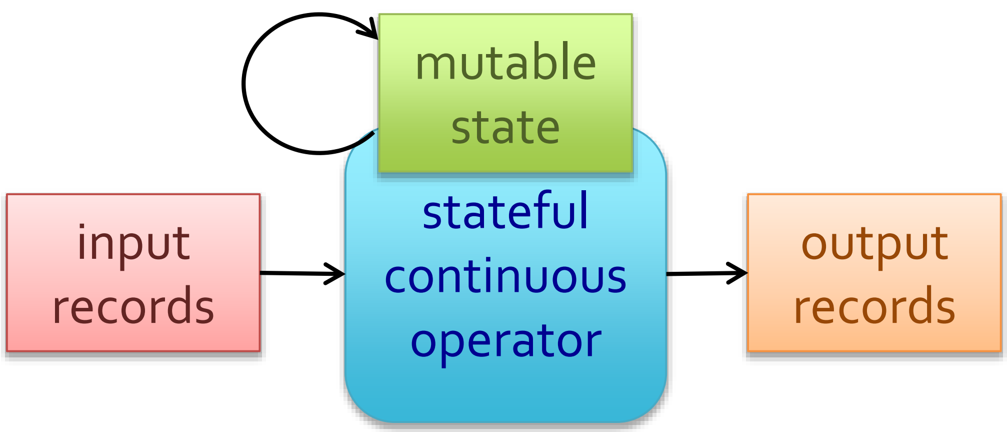
Our Goal

- Scales to hundreds of nodes
- Achieves
- Tolerate
- Sub-seco ry
- Minimal sing

Why is it hard?

Stateful *continuous operators* tightly integrate "computation" with "mutable state"

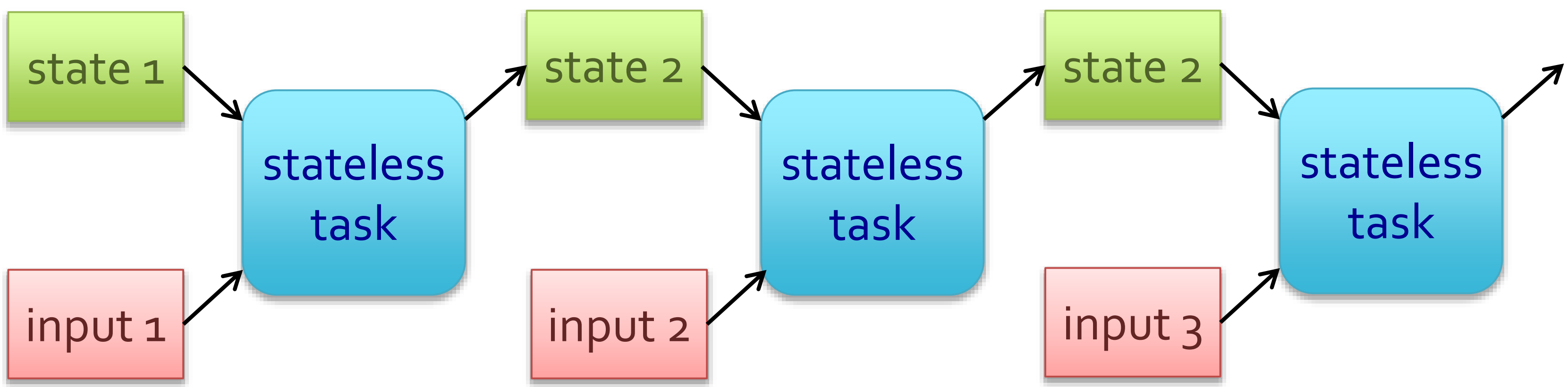
Makes it harder to define clear boundaries when computation and state can be moved around



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

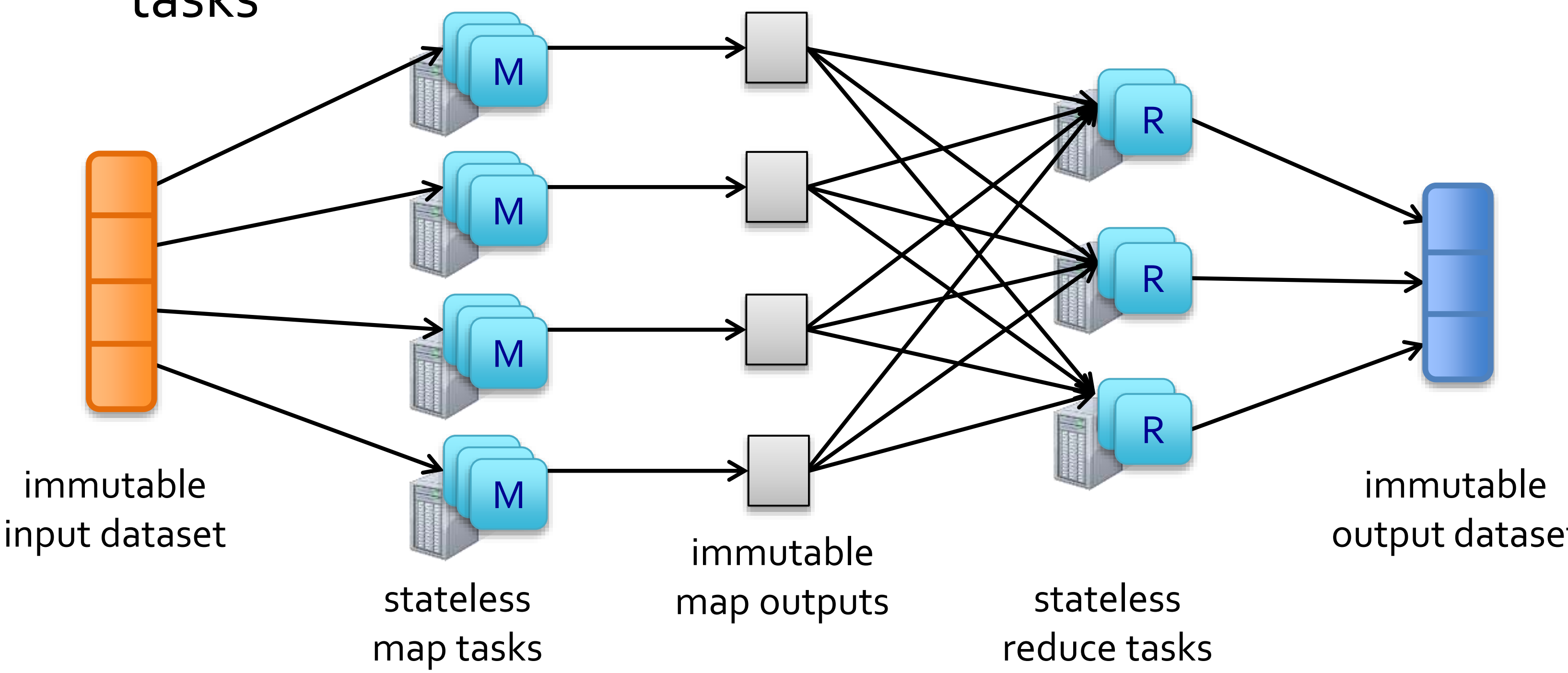


Batch Processing Systems!

Batch Processing Systems

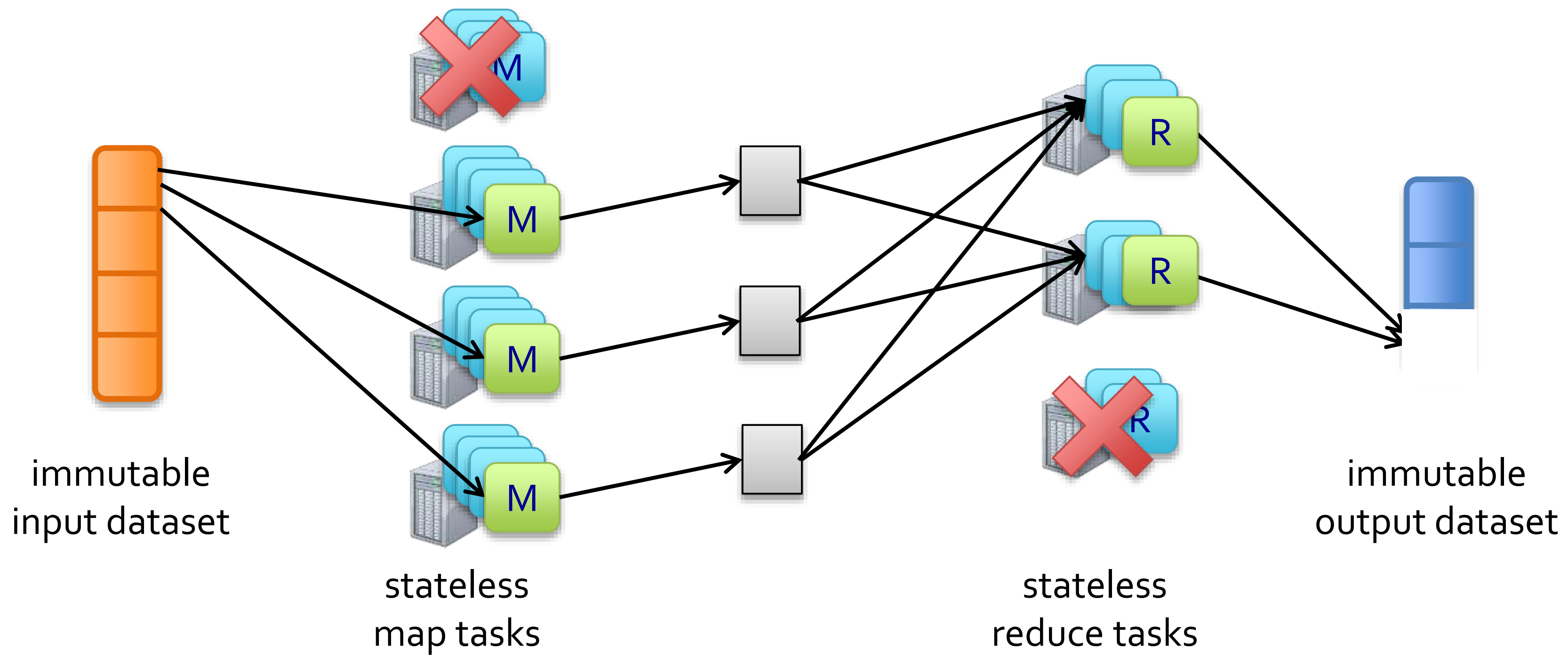
Batch processing systems like MapReduce divide

- 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

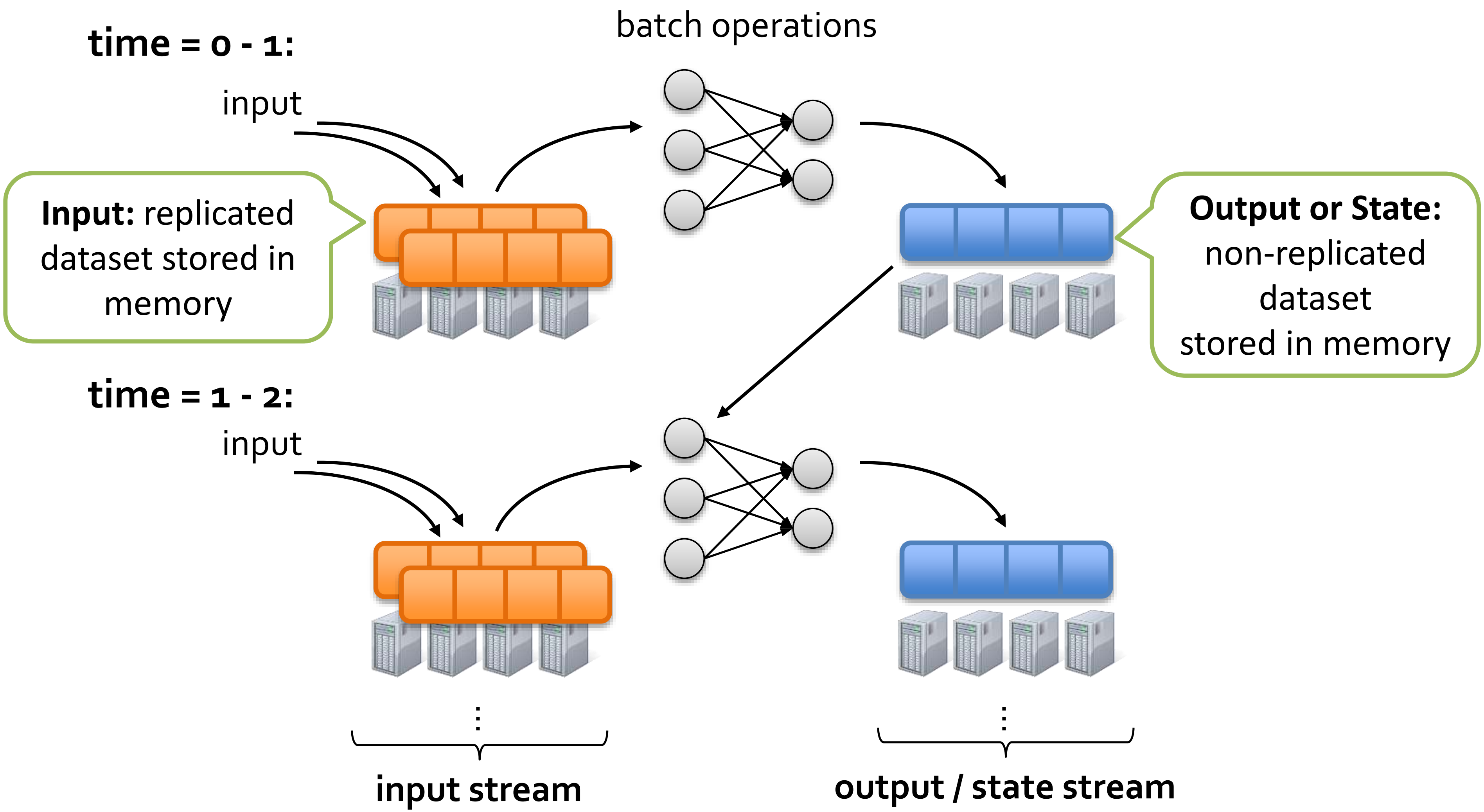
Discretized Stream Processing

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

Store intermediate state data in cluster memory

Try to make batch sizes as small as possible
to get second-scale latencies

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

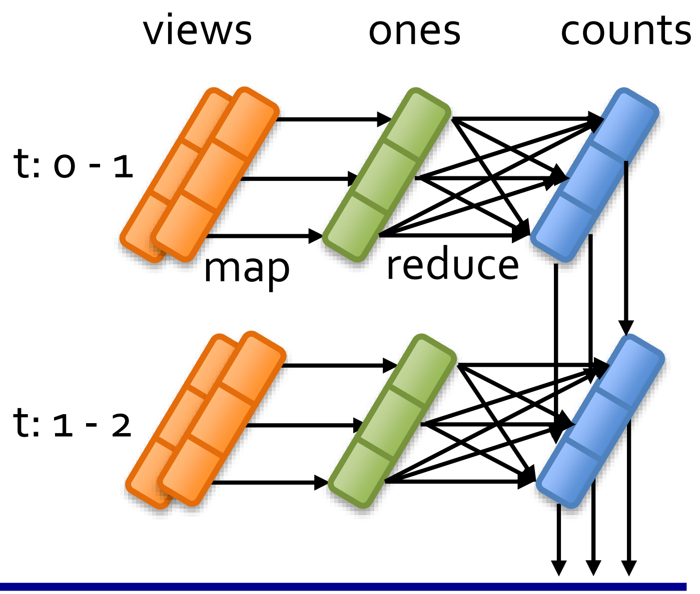
creating a DStream

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

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

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

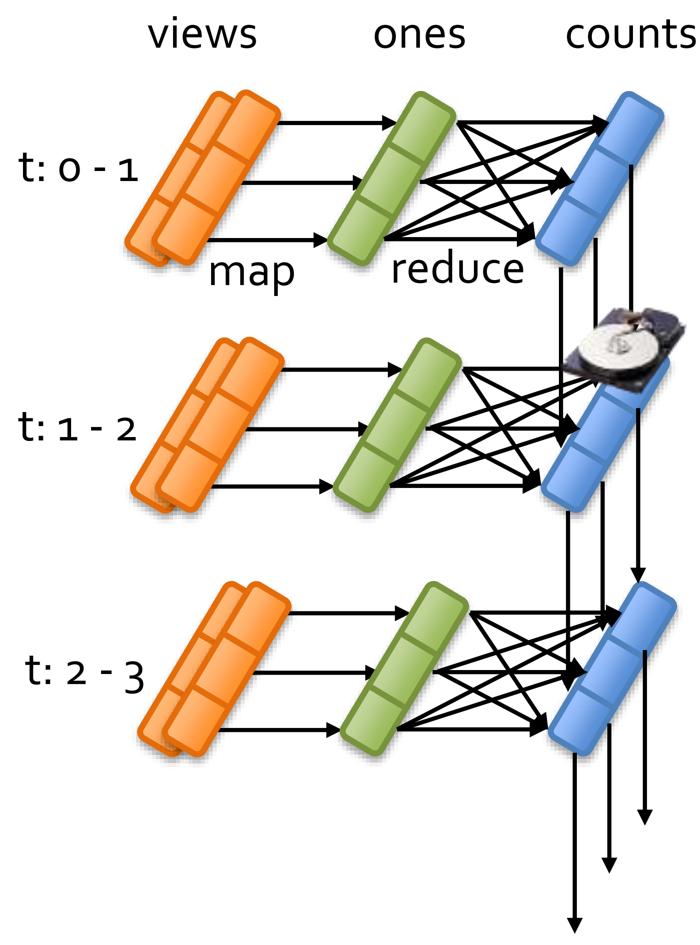
transformation



Fine-grained Lineage

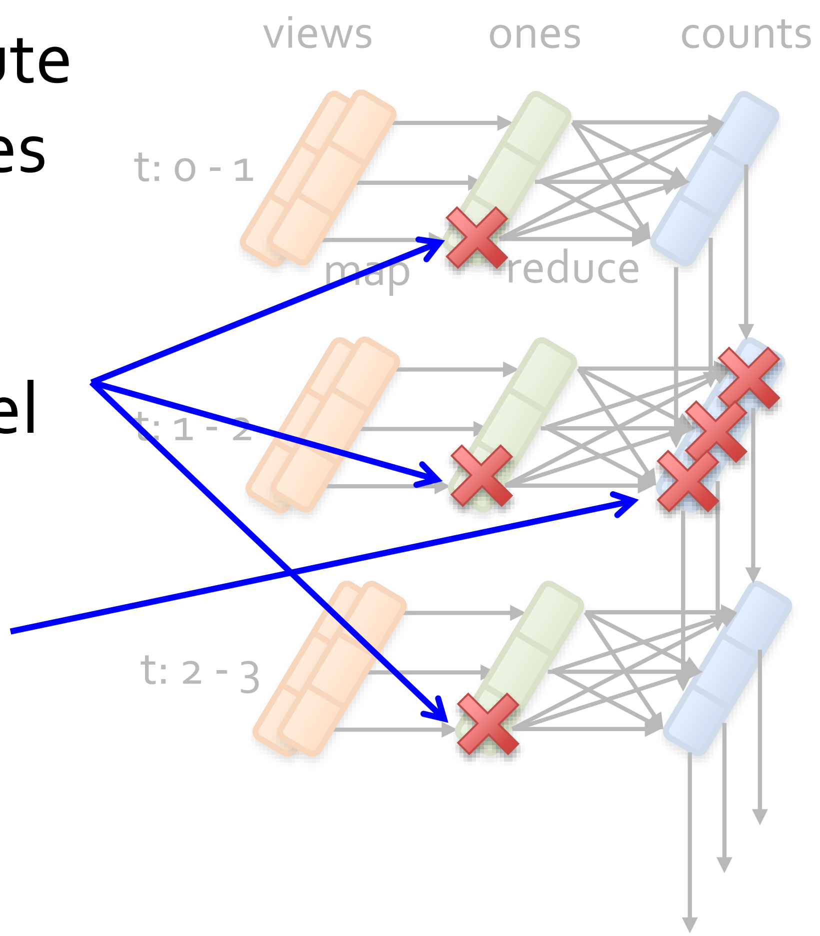
- Datasets track fine-grained operation lineage

- Datasets are periodically checkpointed asynchronously to prevent long lineages

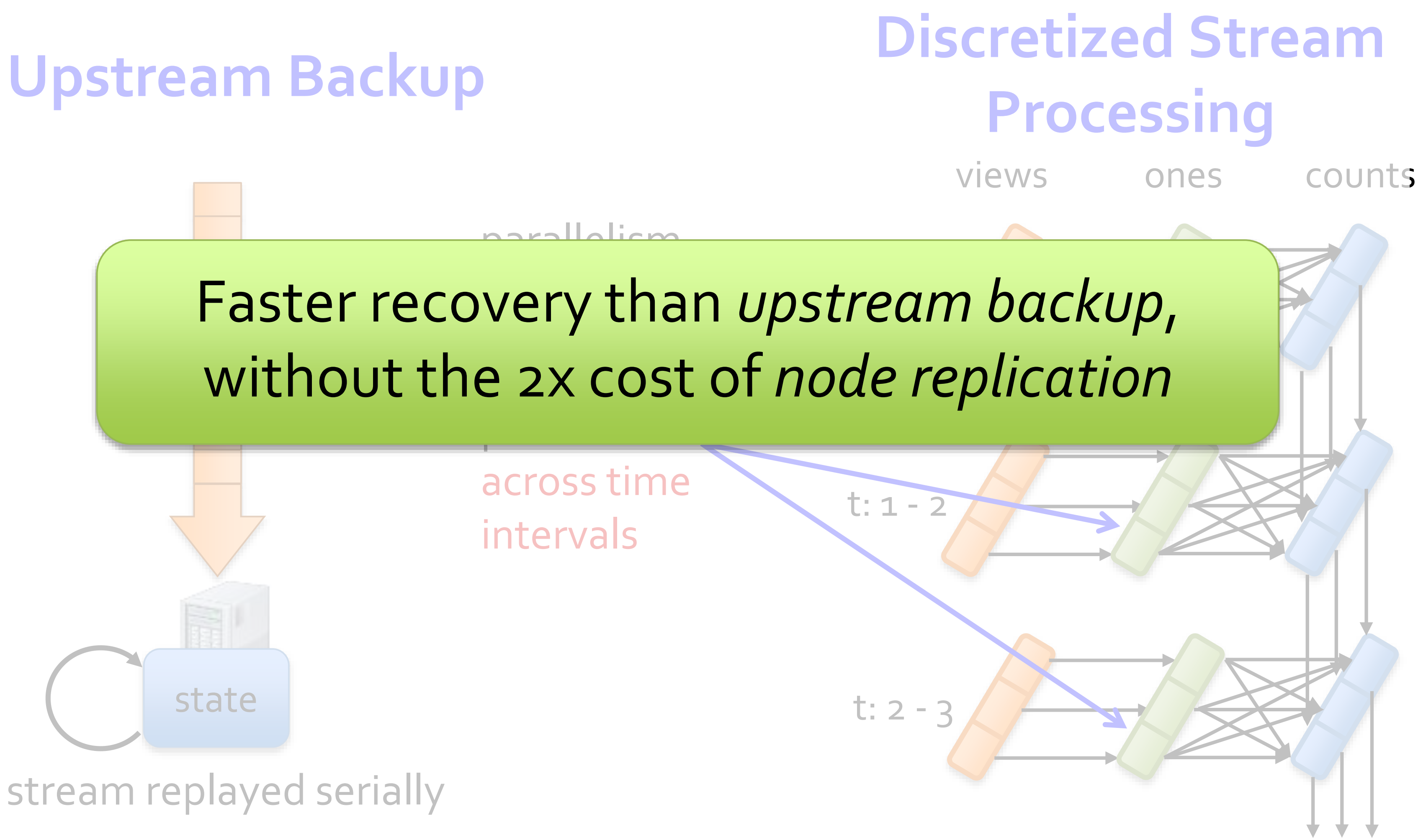


Parallel Fault Recovery

- 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 Upstream Backup



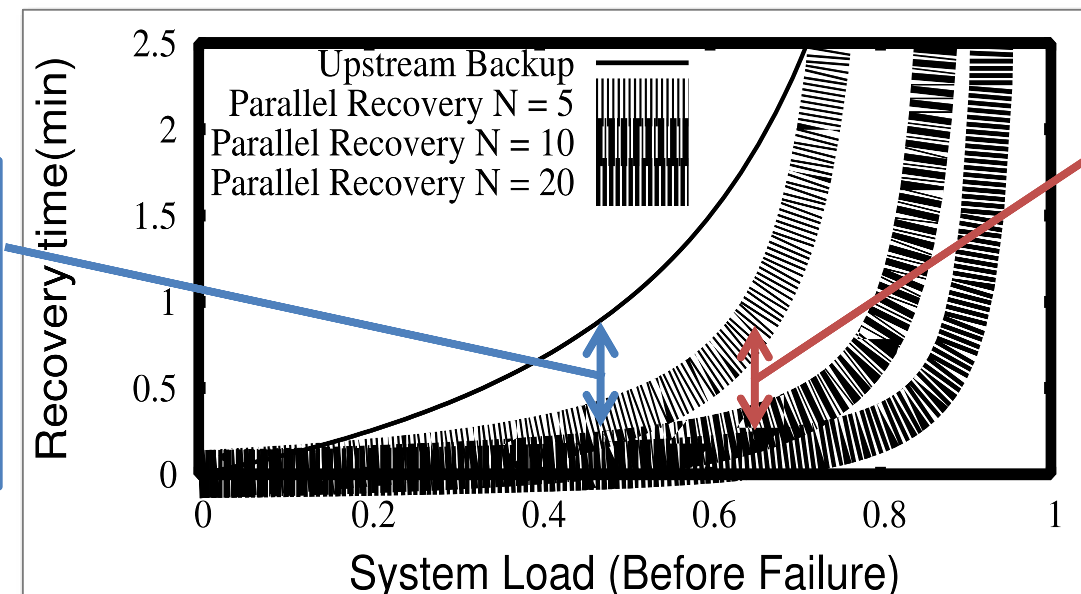
Faster recovery than *upstream backup*, without the 2x cost of *node replication*

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

Evaluation

Spark Streaming

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

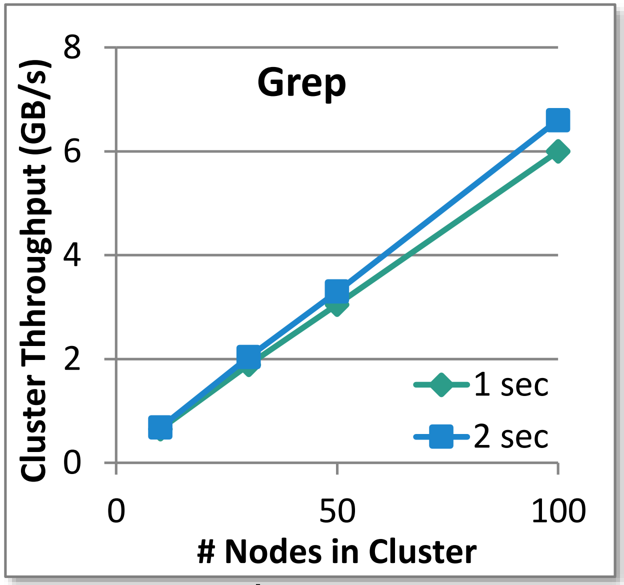
- Modifications required to reduce jobs launching overheads from seconds to milliseconds

[*Resilient Distributed Datasets - NSDI, 2012]

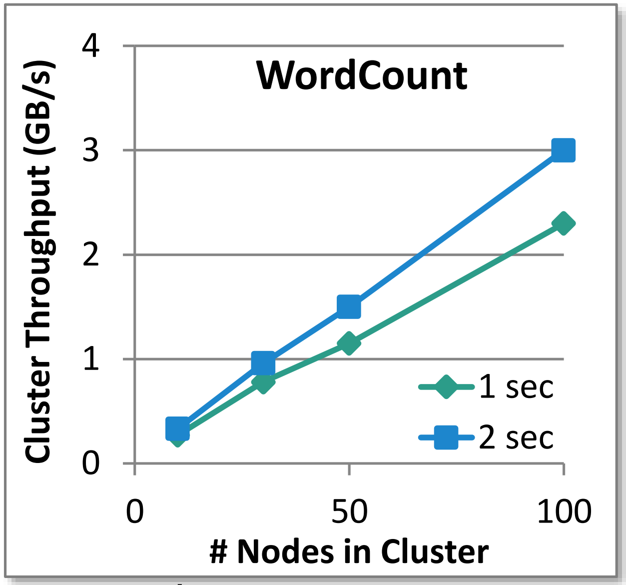
How fast is Spark Streaming?

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?

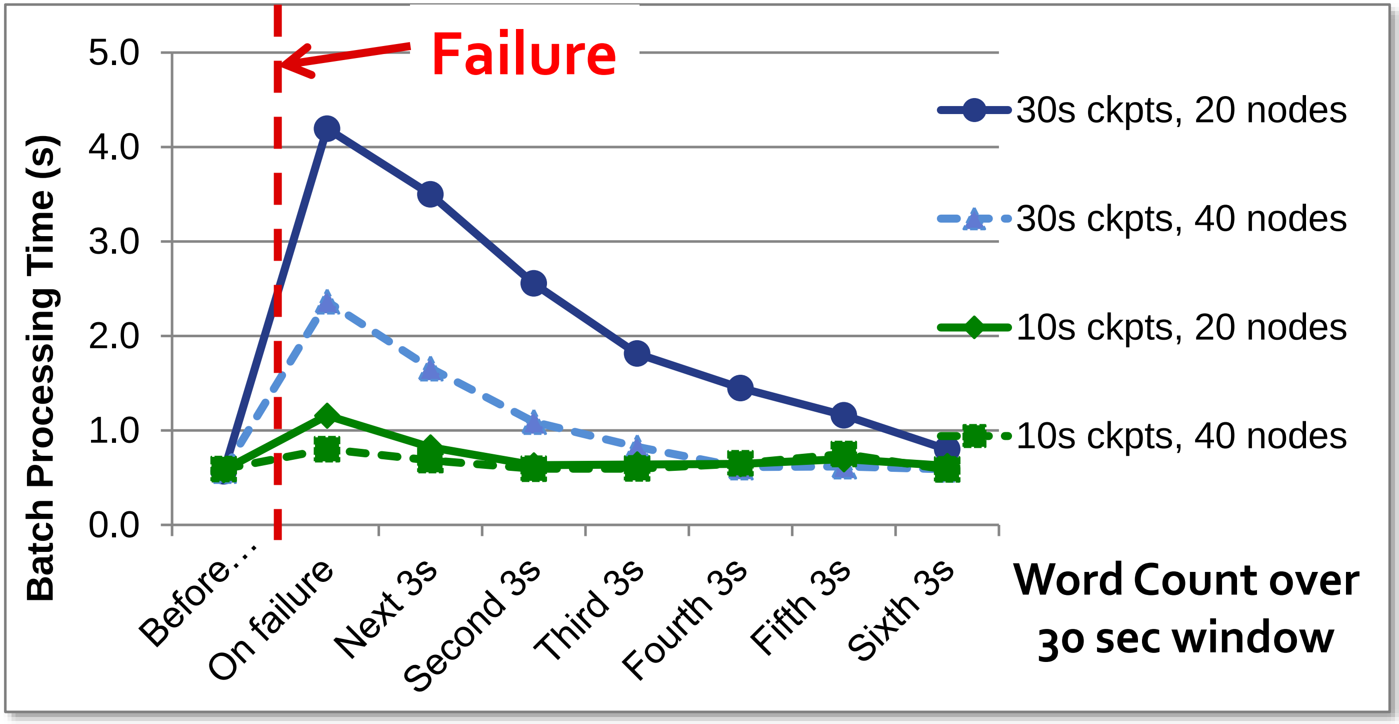
Throughput comparable to other commercial stream processing systems

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

[Refer to the paper for citations]

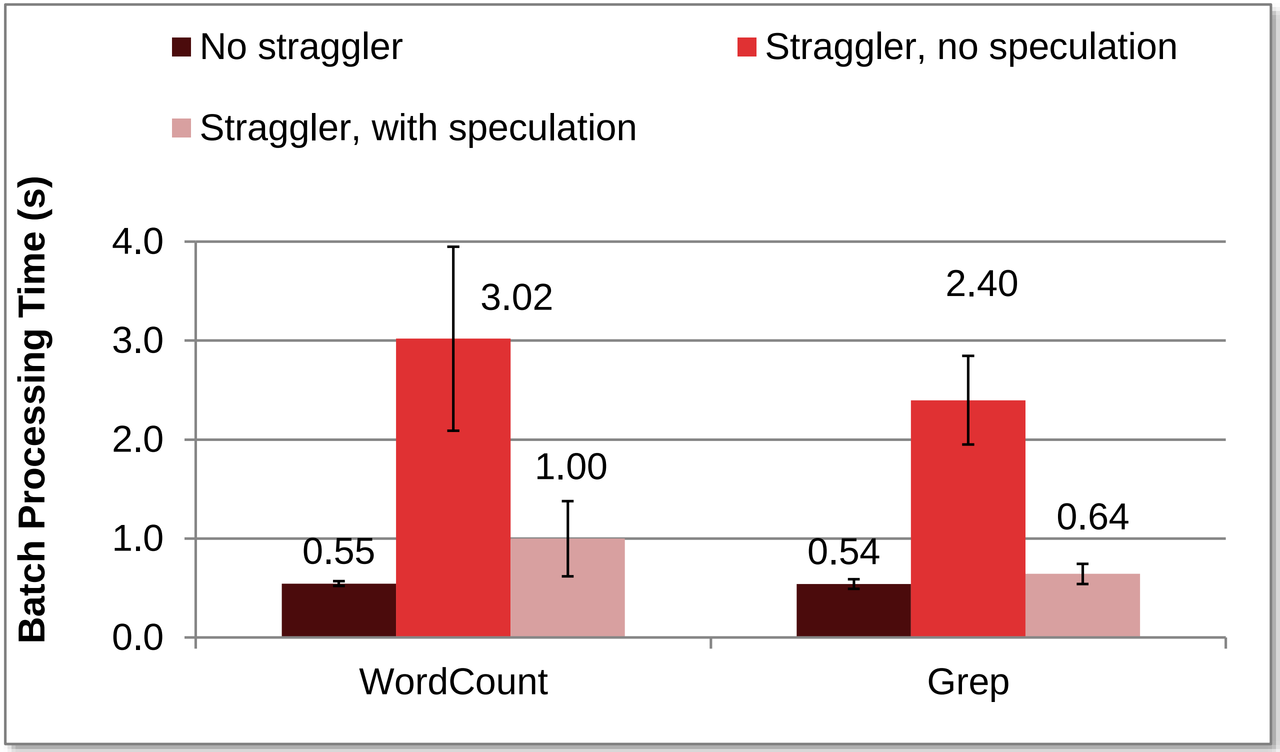
How fast can it recover from faults?

Recovery time improves with more frequent checkpointing and more nodes



How fast can it recover from stragglers?

Speculative execution of slow tasks mask the effect of stragglers



Unification with Batch and Interactive Processing

Unification with 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(...)
```

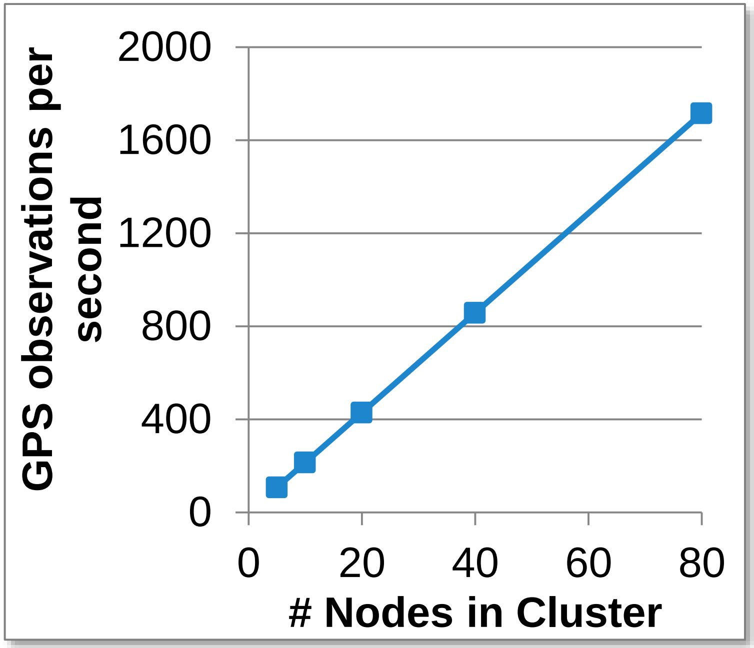
- Interactively query live streams

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

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* – Full cluster rollback on recovery
- *SEEP* – Extends continuous operators to enable parallel recovery, but does not handle stragglers
- *TimeStream* – Recovery similar to upstream backup
- *MillWheel* – State stored in BigTable, transactions per state update can be expensive

Takeaways

- Large scale streaming systems must handle faults and stragglers
- Discretized Streams model streaming computation as series of batch jobs
  - Uses simple techniques to exploit parallelism in streams
  - Scales to 100 nodes with 1 second latency
  - Recovers from failures and stragglers very fast
- Spark Streaming is open source - spark-project.org
  - Used in production by ~ 10 organizations!