Discretized Streams

Fault Tolerance Streaming Computation at Scale

Ion Stoica
UC Berkeley

Joint work with: Matei Zaharia, **Tathagata Das (TD)**, Haoyuan Li (HY), Timothy Hunter, Scott Shenker

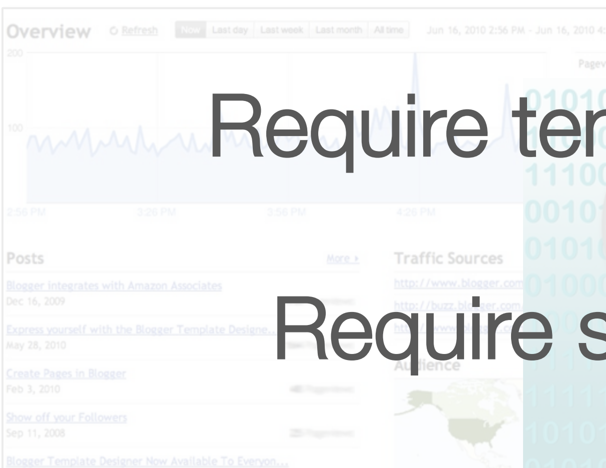
Why Care?

Data is important as the decisions it enables

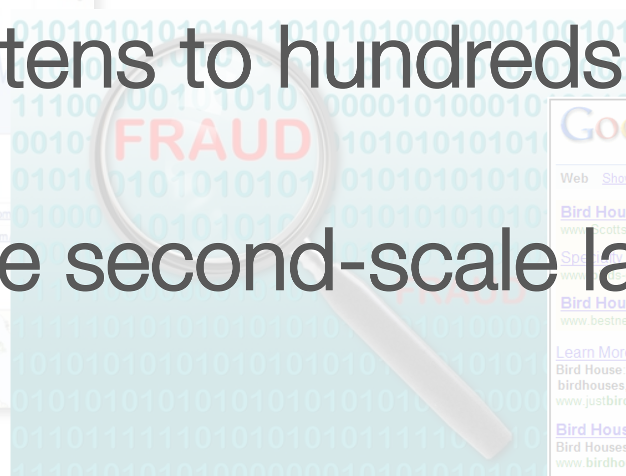
Decisions on fresh data better than on stale data

» More and more apps want to process large data streams

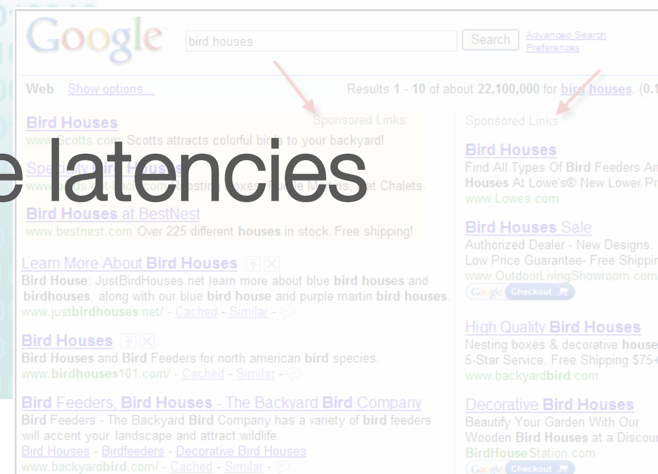
Website monitoring



Fraud detection



Ad monetization



Require tens to hundreds of nodes

Require second-scale latencies

Why Hard beyond Scale & Latency?

Typically run 24x7 services

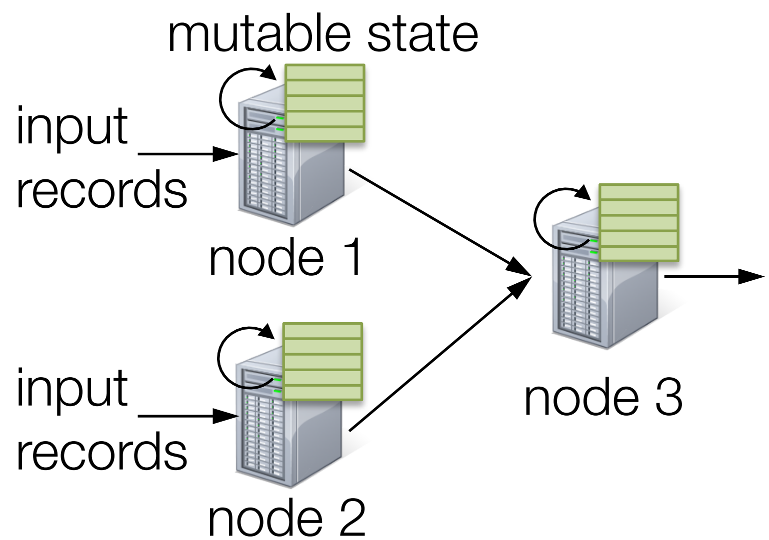
Need to recover from failure very fast, e.g., sub-second recovery time

» Need to handle stragglers as well

Traditional systems either *inefficient* or *slow*

Traditional Streaming Systems

DAGs of stateful operators



Traditional Streaming Systems

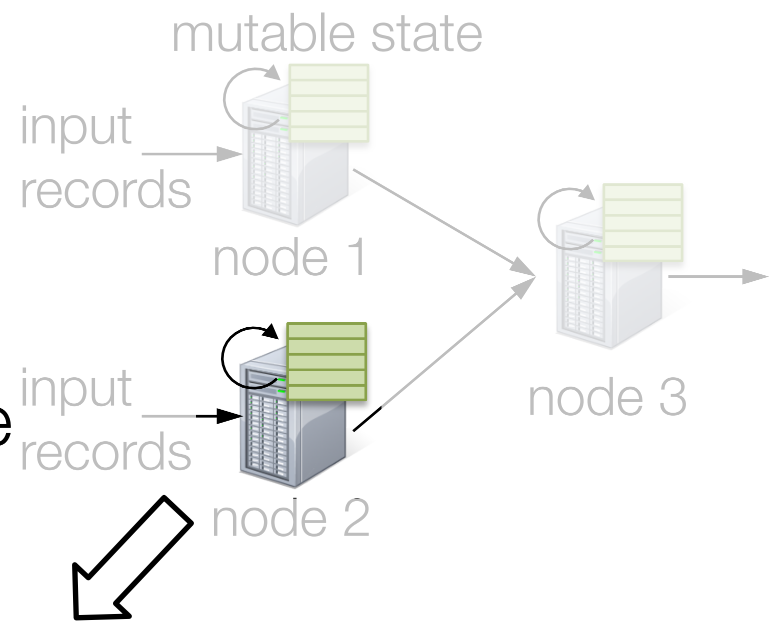
DAGs of stateful operators

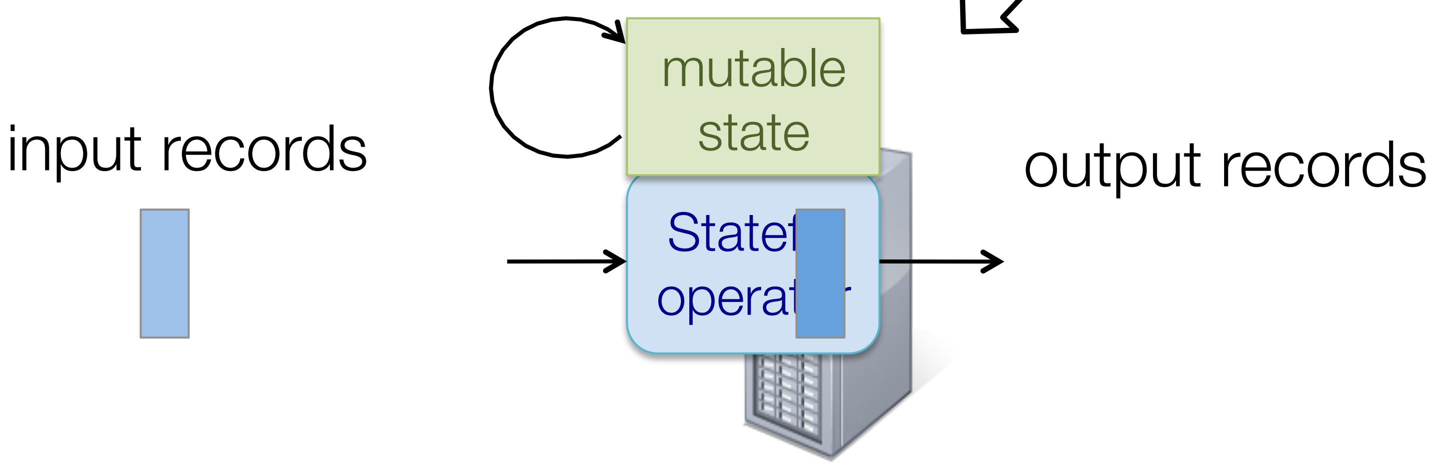
Each operator

» Get record

» Process record and update state

» Eventually emit a new record





Traditional Streaming Systems

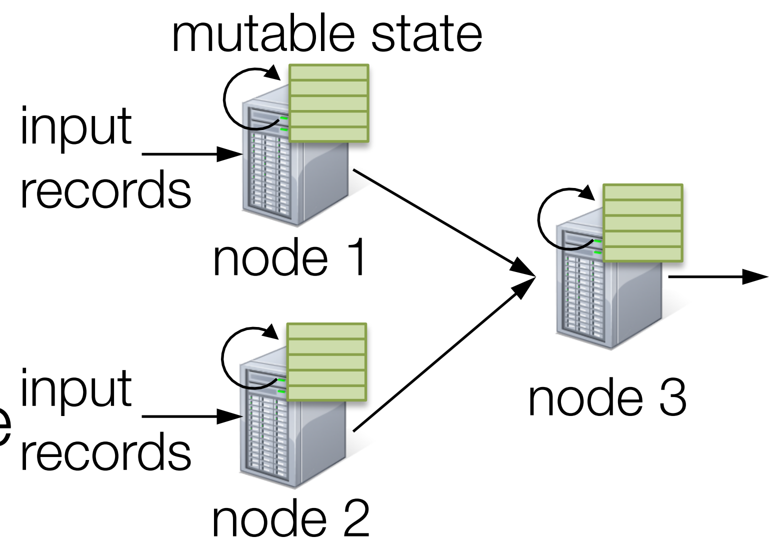
DAGs of stateful operators

Each operator:

» Get record

» Process record and update state

» Eventually emit a new record

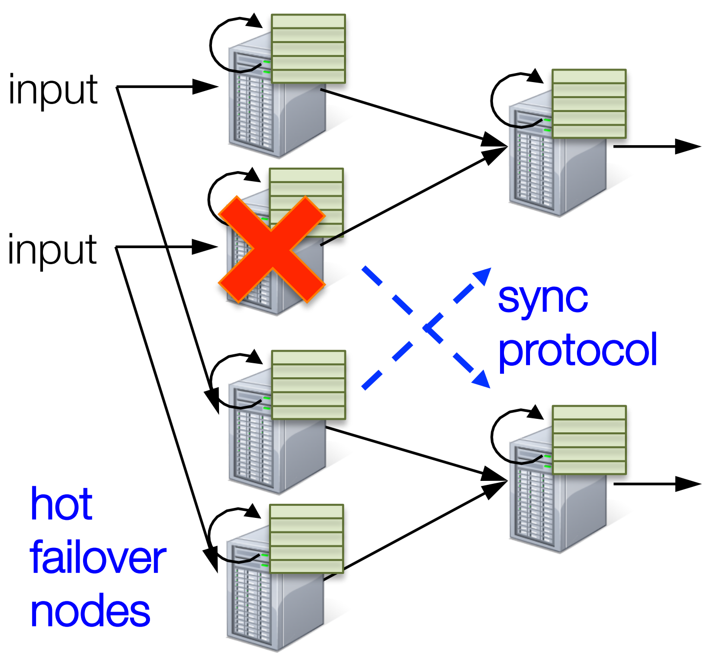


State is lost if node fails

Two general techniques for fault tolerance

Replication

Examples: Borealis, Flux



Separate set of "hot failover" nodes process the same data streams

Sync. protocols ensures exact ordering of records in both sets

On failure, the system switches over to the failover nodes

Fast recovery, but up to 2x hardware cost

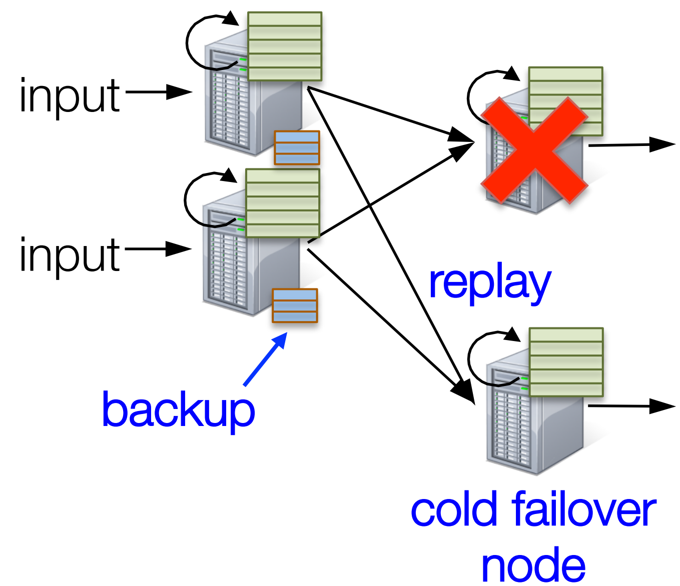
Upstream Backup

Examples: TimeStream, Storm

Each node backups forwarded records

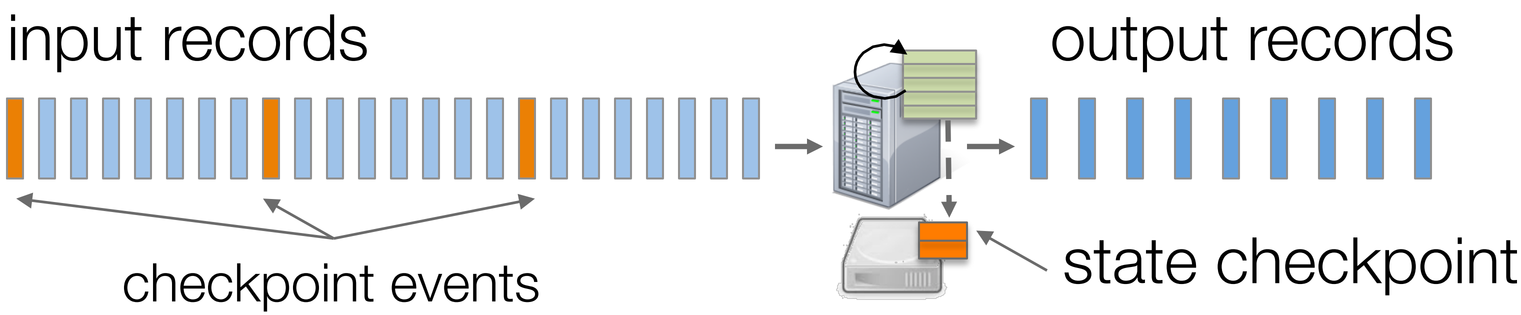
Maintain “cold failover”

On failure, upstream nodes replay the backup records *serially*

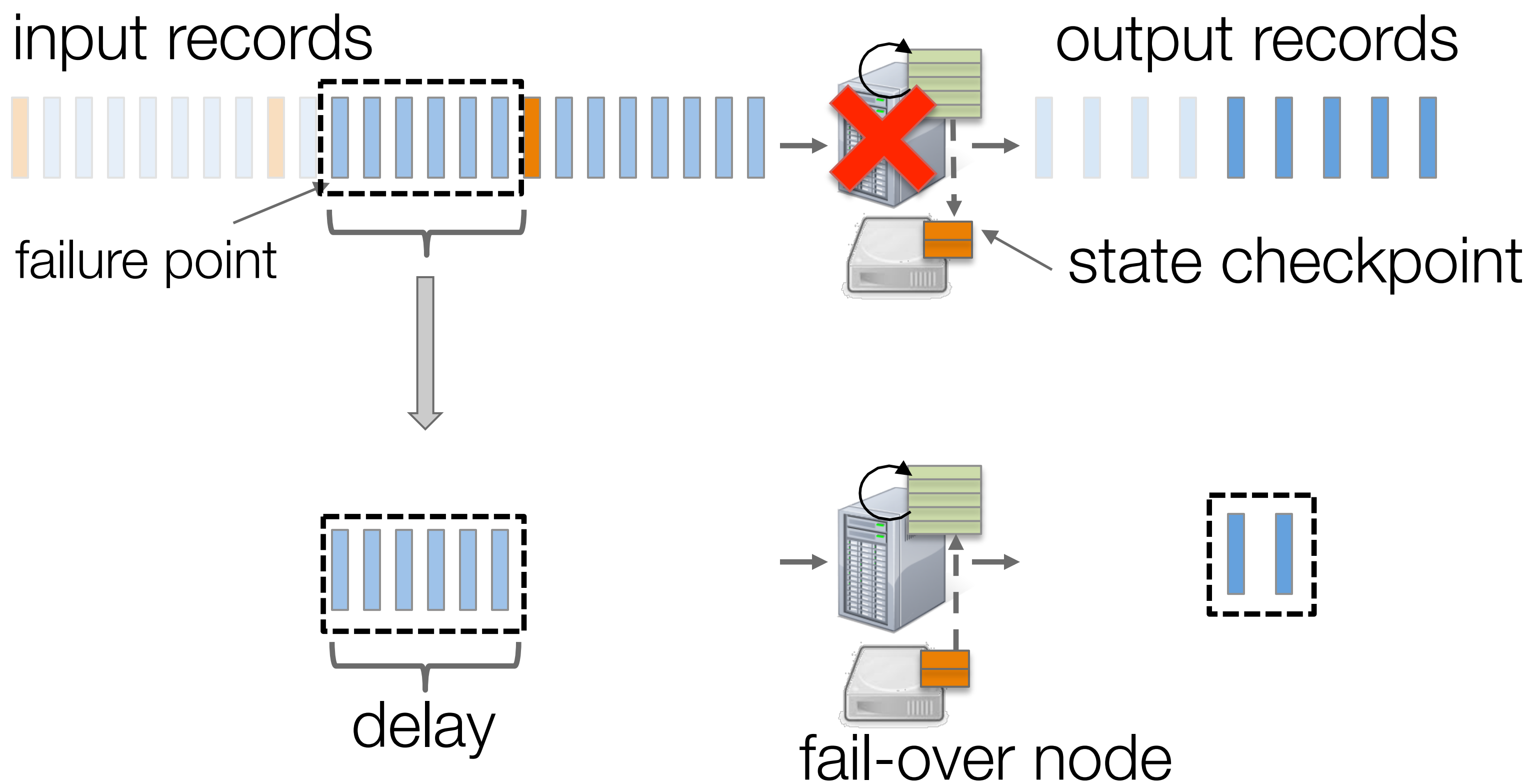


Only need one standby, but slow recovery

Understanding upstream Backup



Understanding upstream Backup

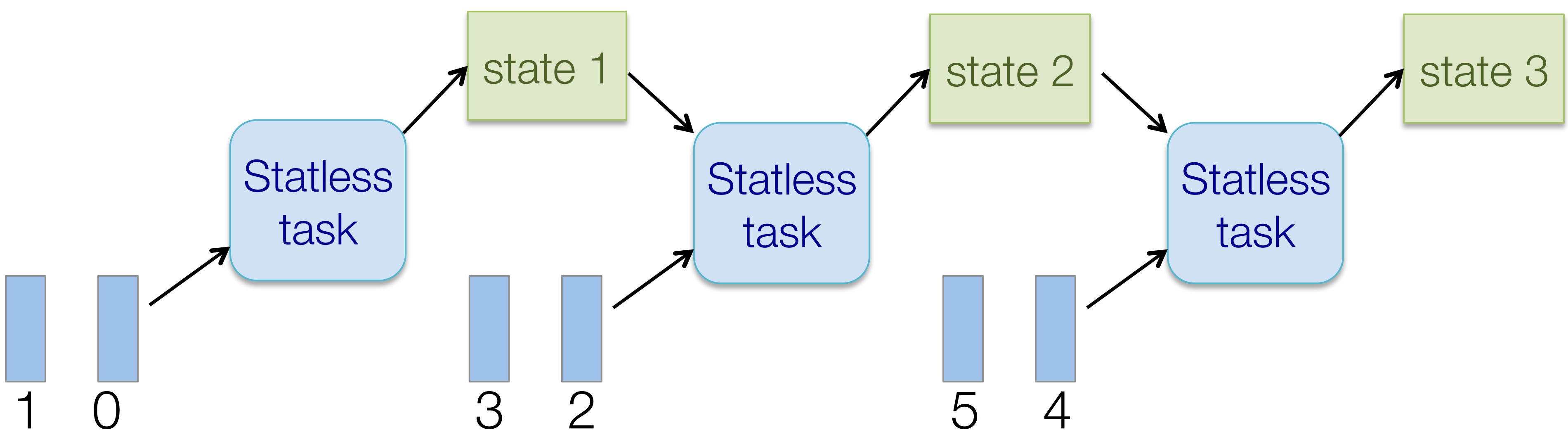


Delay as large as checkpoint interval

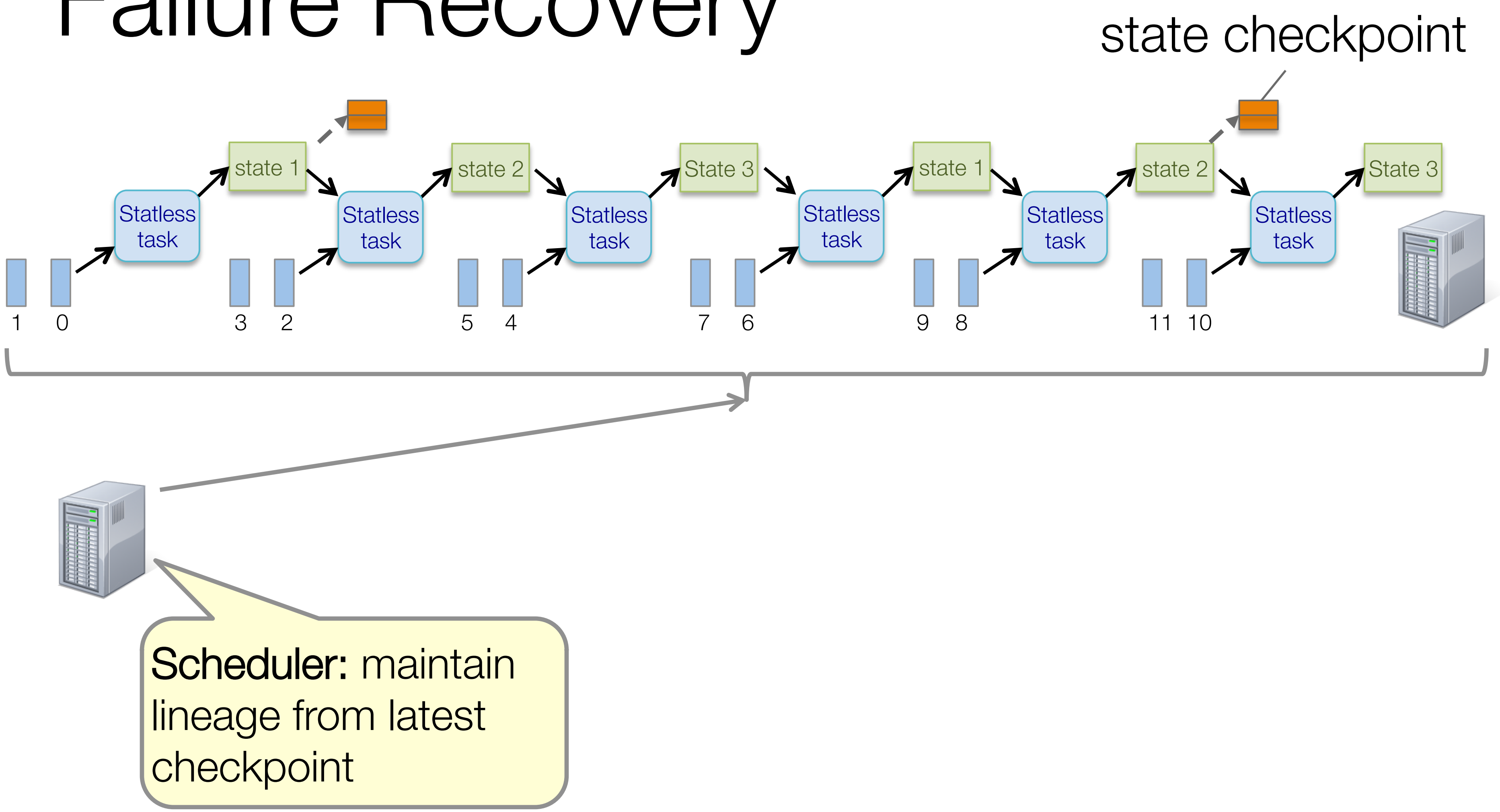
Key Idea: Stateless Tasks

Split computation in small ***stateless*** tasks

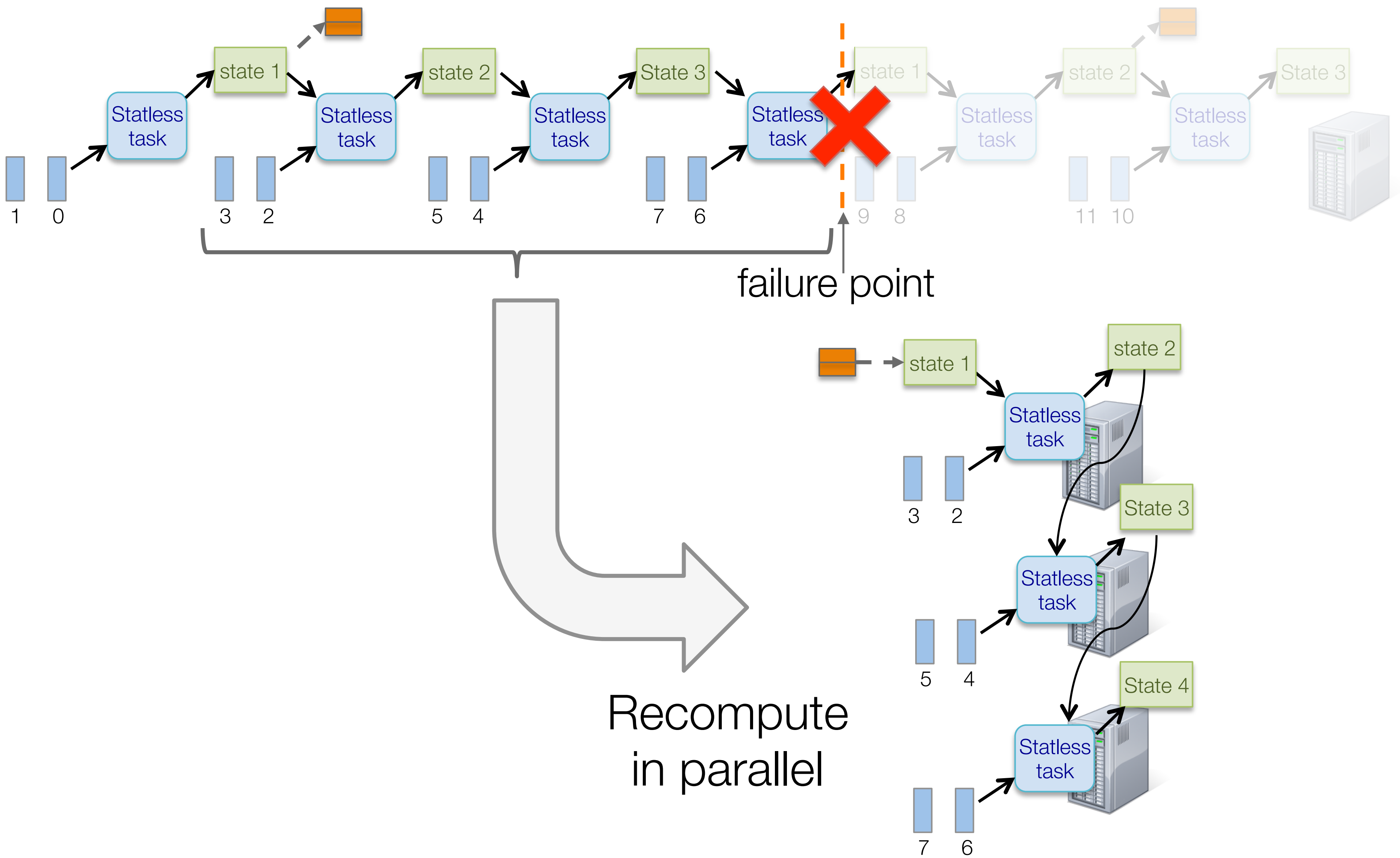
Naturally define boundaries where computation can be moved around



Failure Recovery



Failure Recovery



Discretized Stream Processing

Discretized Stream Processing

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

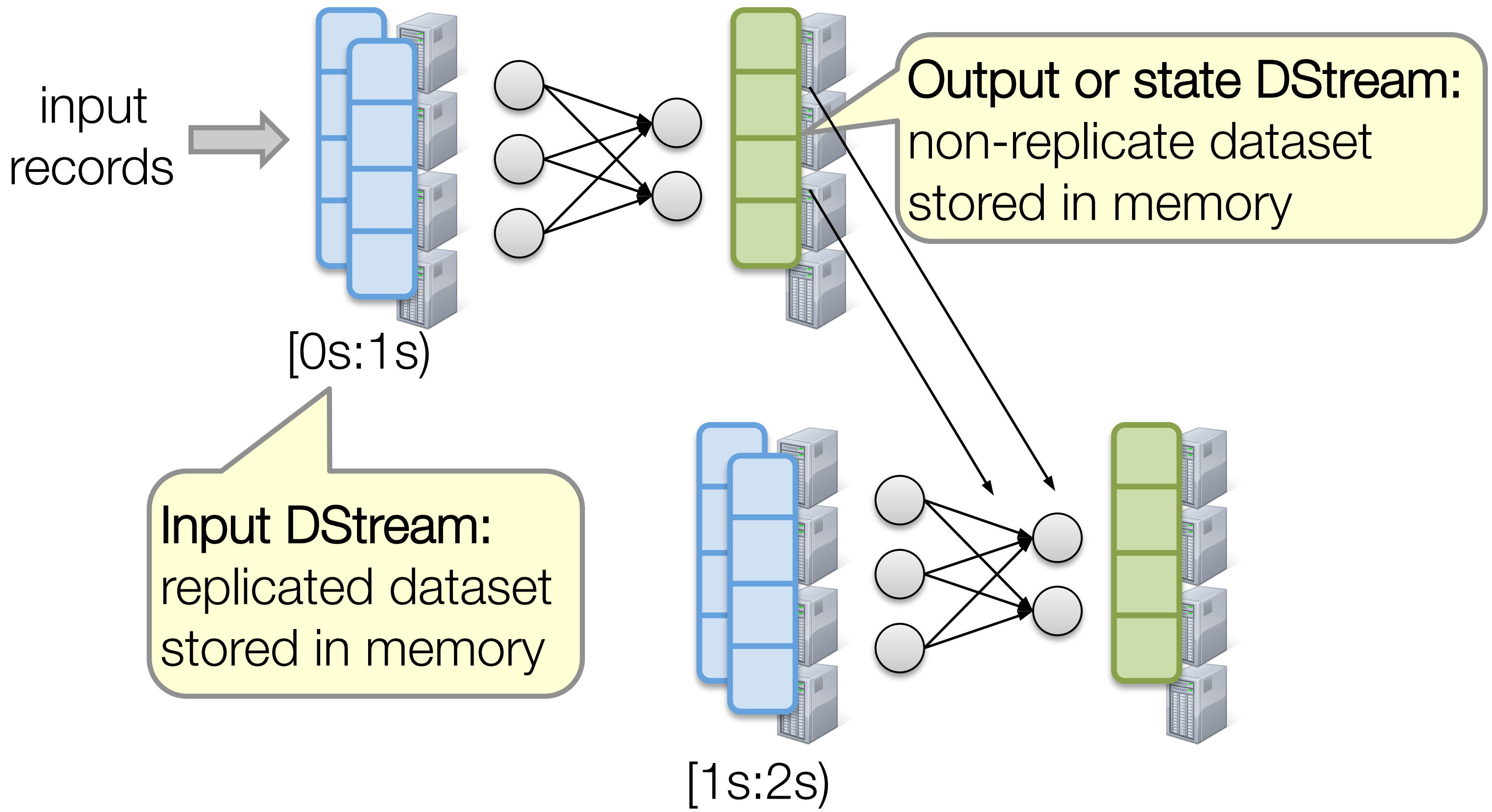
Keep lineage since last checkpoint

Challenge: make data batches as small as possible

Discretized Stream Processing

DStream: seq. of immutable, partitioned datasets

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

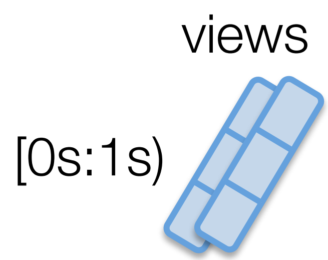


Example: Counting page views

Input DStream: split incoming records into 1s batches

creating a DStream

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



Example: Counting page views

Input DStream: split incoming records into 1s batches

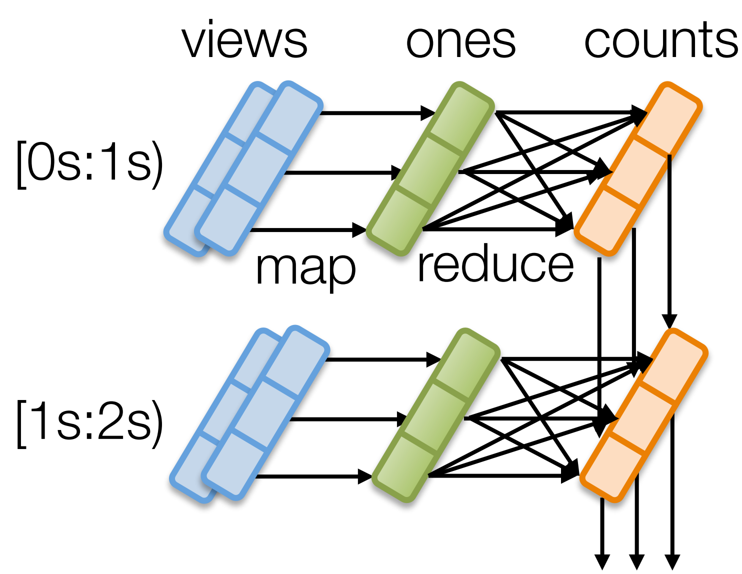
creating a DStream

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

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

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

transformation

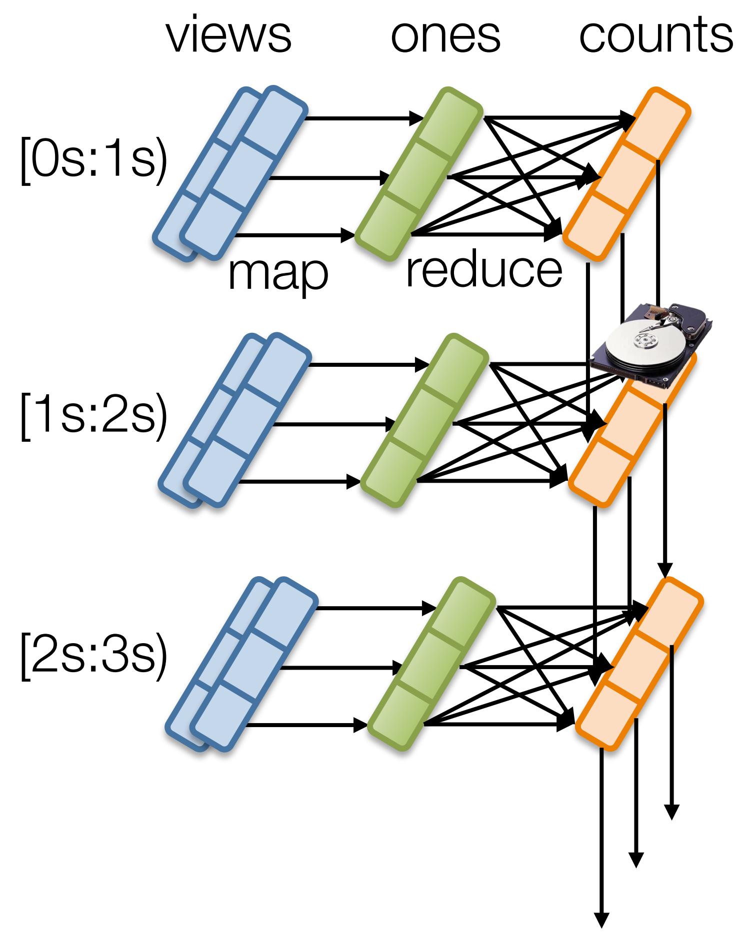


Fine-grained Lineage

Track fine-grained operation lineage

Datasets are periodically checkpointed

» Asynchronously to prevent long lineages

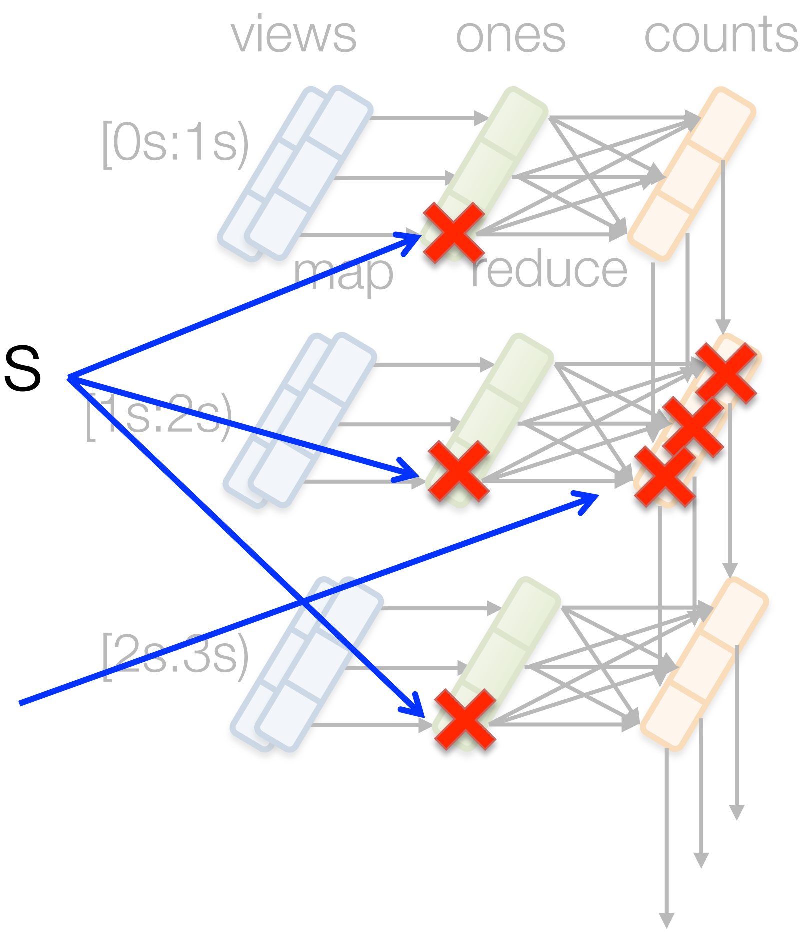


Parallel Fault Recovery

Use lineage to recompute lost partitions

Datasets in different batches recomputed in parallel

Partitions within a dataset also recomputed in parallel

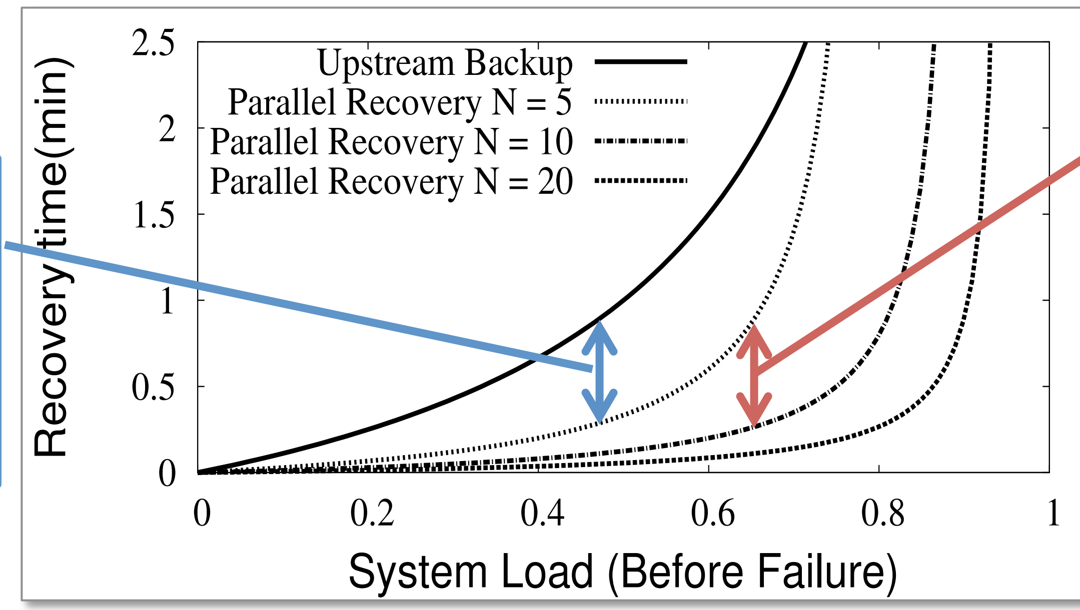


How much faster than Upstream Backup?

Recovery time = time to recompute & 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 with 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

Mask the impact of slow nodes on the progress of the system

Evaluation

Spark Streaming

Implemented on top of Spark*

» Supports in-memory storage and recovery via lineage

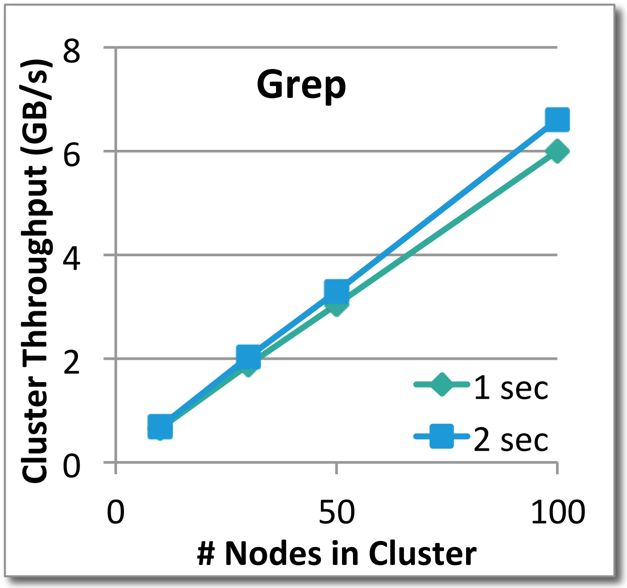
Numerous performance optimization

[*Resilient Distributed Datasets - NSDI, 2012]

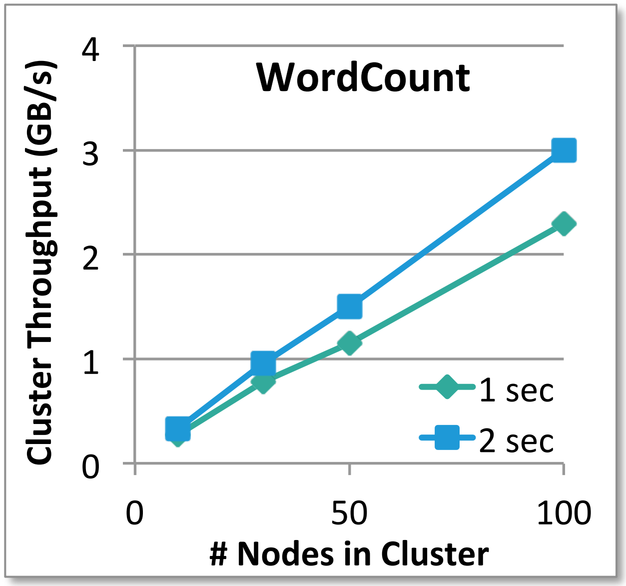
How fast is Spark Streaming?

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

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



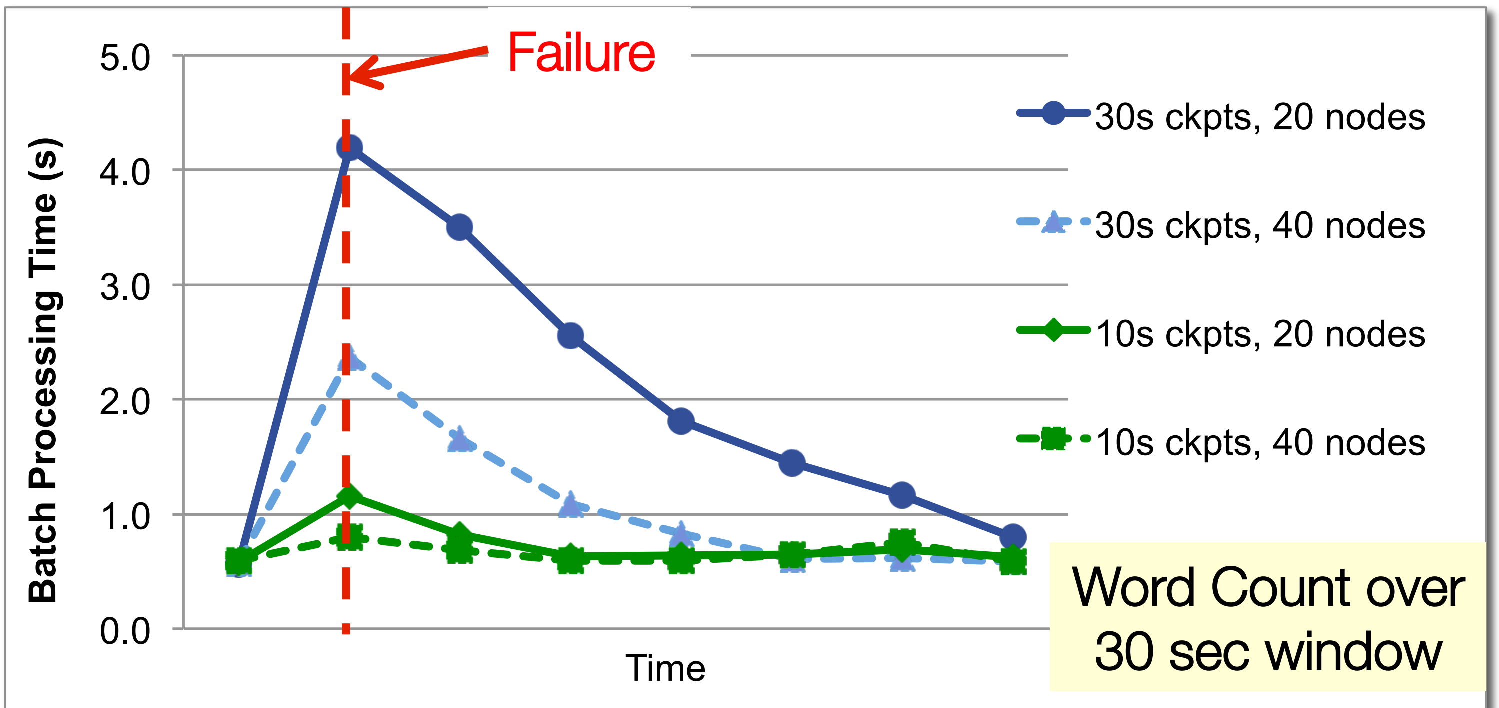
Count the sentences having a keyword



WordCount over 30 sec sliding window

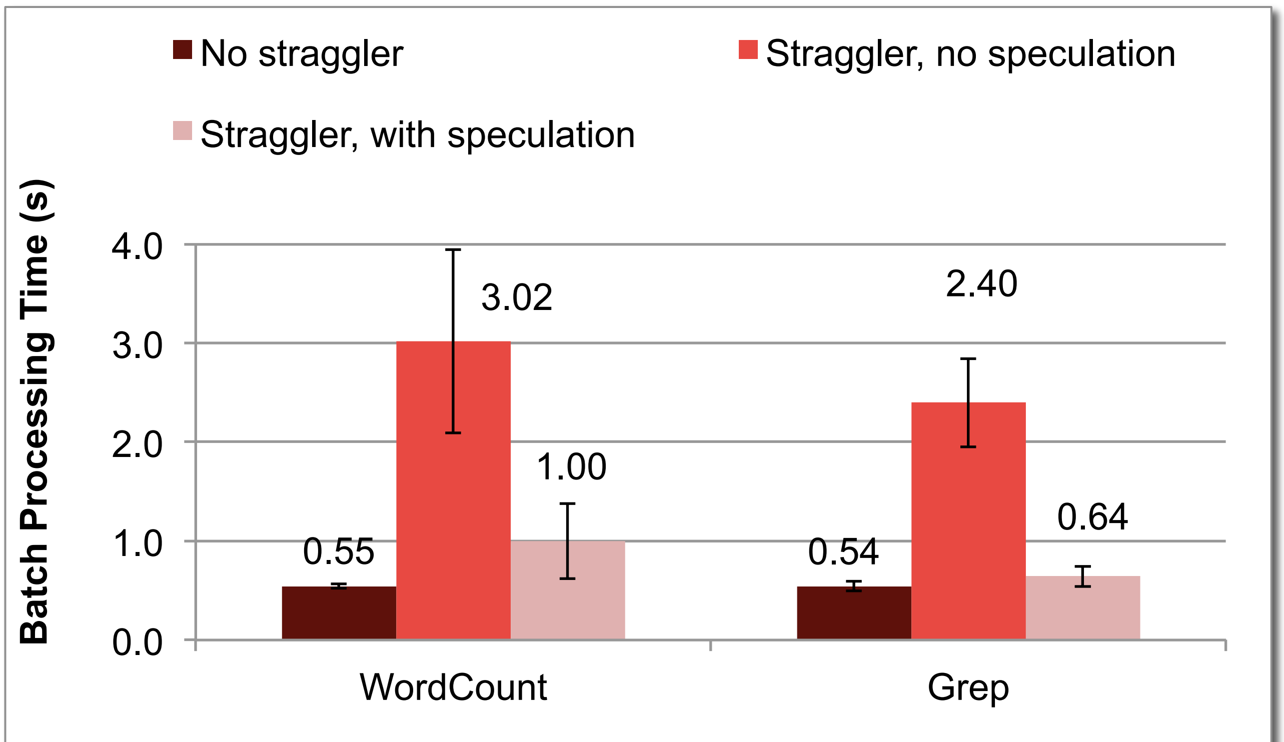
Fault Recovery

Recovery time improves with more frequent checkpointing and more nodes



Straggler Mitigation

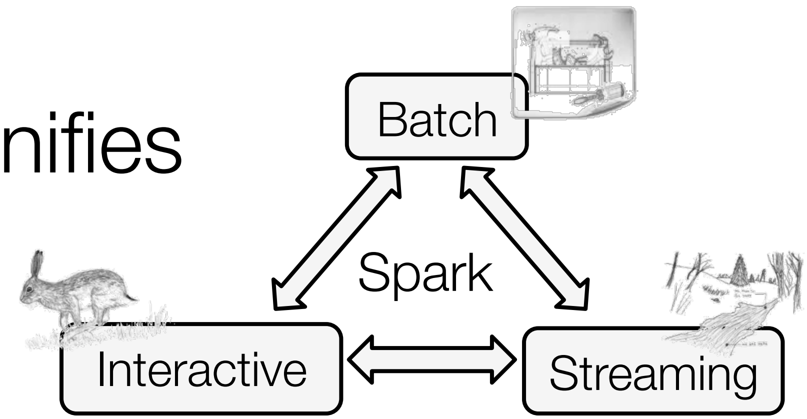
Speculative execution of slow tasks mask the effect of stragglers



Unification

Spark + SprakStreaming unifies

» Batch
» Interactive
» Streaming



Combine live data streams with historic data

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

Interactively query live streams

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

Summary

Large scale streaming systems must handle failures and stragglers

Discretized Streams model streaming computation as series of batch jobs

» Naturally 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!

Exciting Future Work

Trade between latency and throughput by dynamically adjusting batch size

Partial computation to handle tight latency

» Don't wait for stragglers

» Expose partially executed DAG

» (Eventually) update results when straggler finish

Dynamic optimization of execution plan

» Use measurements from previous job to optimize execution of next job