Discretized Streams

An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters

Matei Zaharia, Tathagata Das,
Haoyuan Li, Scott Shenker, Ion Stoica

amplab
UC BERKELEY

Motivation

- Many important applications need to process large data streams arriving in real time
  - User activity statistics (e.g. Facebook's Puma)
  - Spam detection
  - Traffic estimation
  - Network intrusion detection
- Our target: large-scale apps that must run on tens-hundreds of nodes with O(1 sec) latency

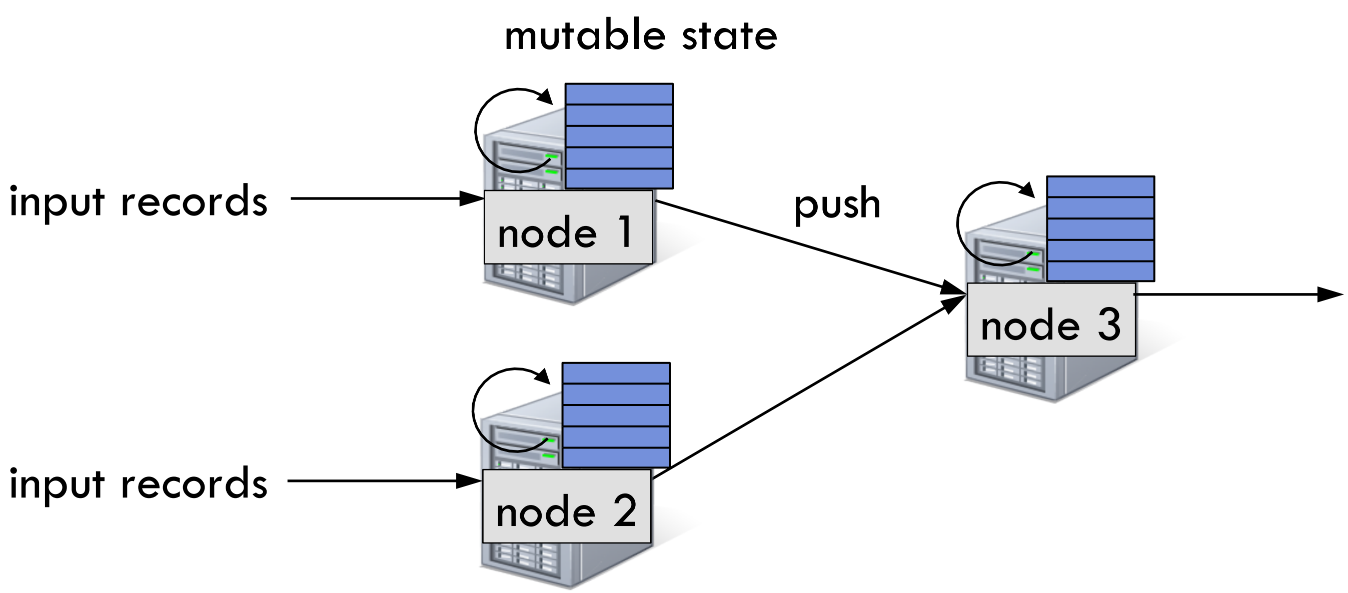
Challenge

- To run at large scale, system has to be both:
  - **Fault-tolerant:** recover quickly from failures and stragglers
  - **Cost-efficient:** do not require significant hardware beyond that needed for basic processing

- Existing streaming systems don’t have both properties

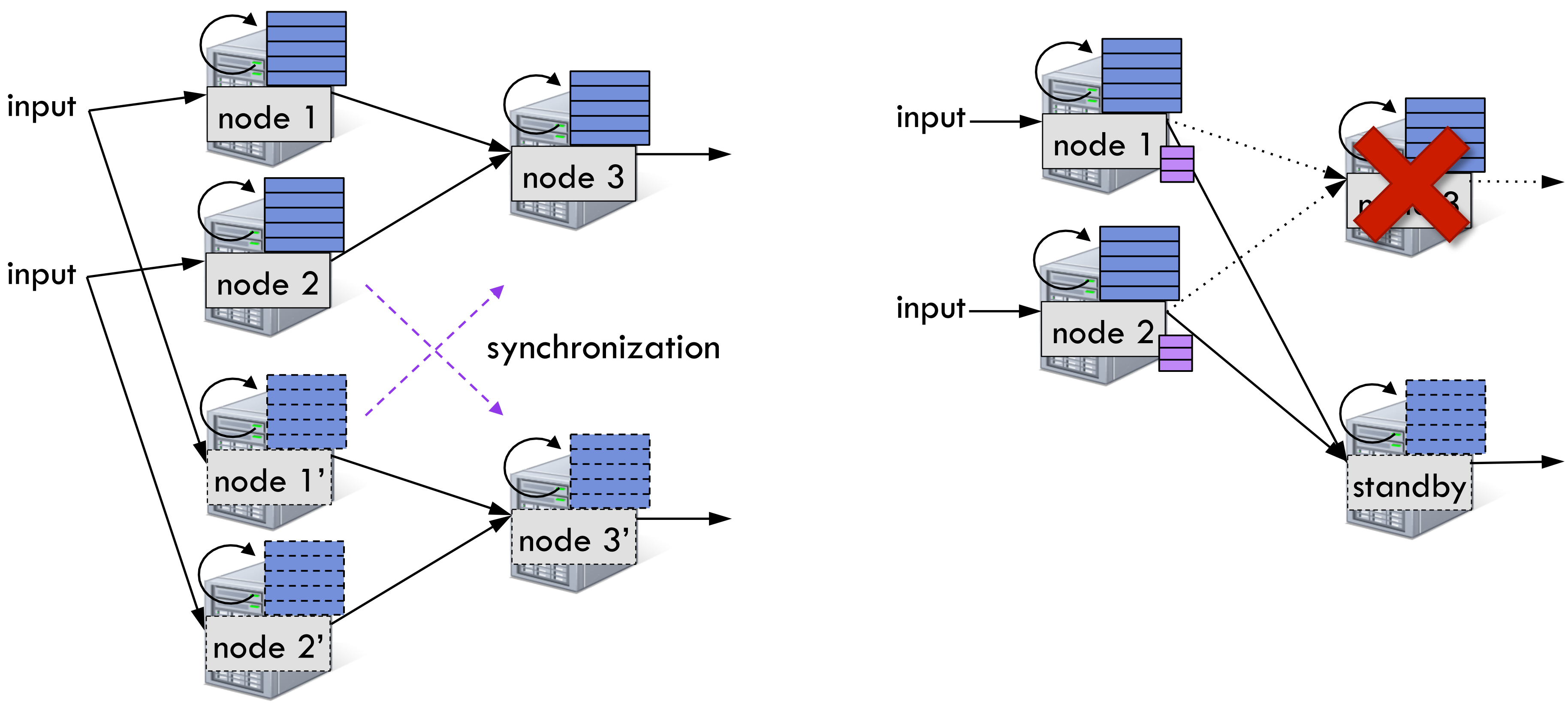
Traditional Streaming Systems

- "Record-at-a-time" processing model
  - Each node has mutable state
  - For each record, update state & send new records



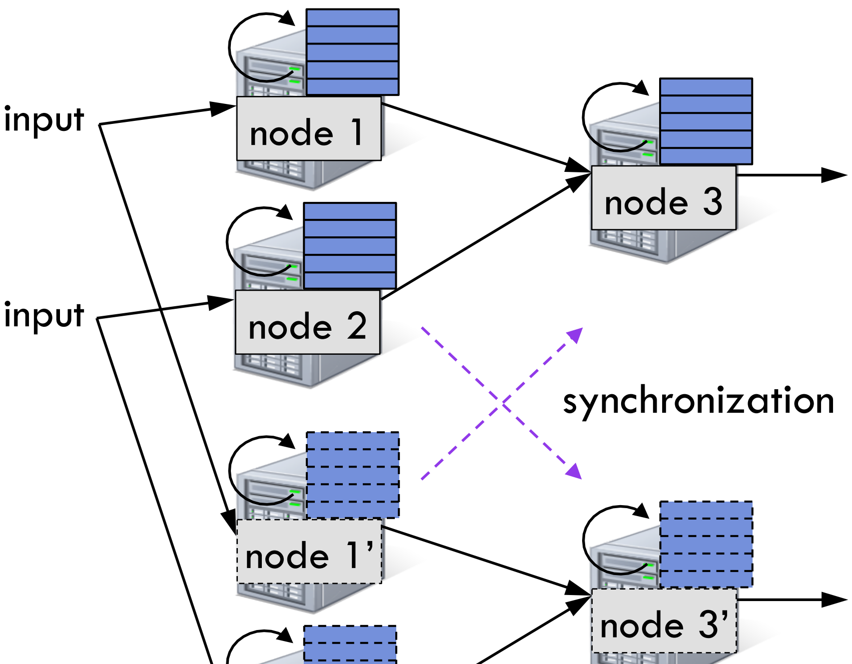
Traditional Streaming Systems

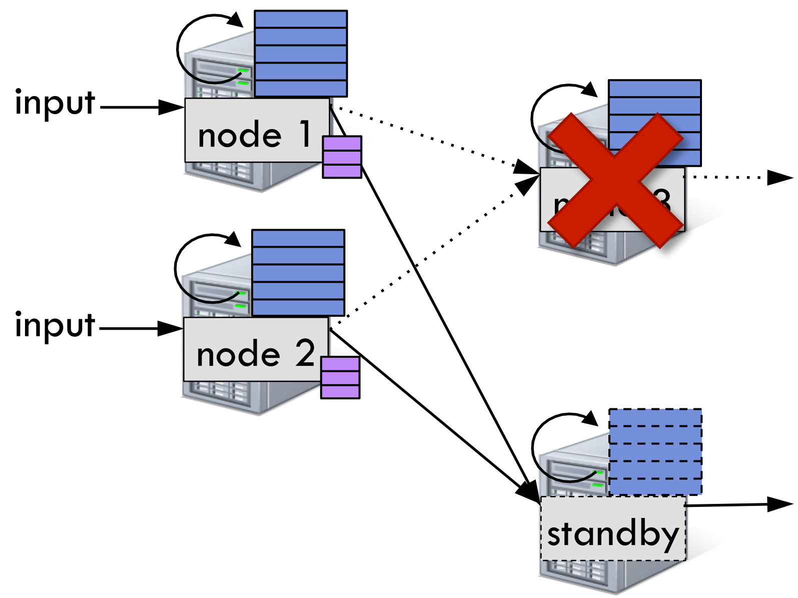
Fault tolerance via **replication** or **upstream backup:**



Traditional Streaming Systems

Fault tolerance via **replication** or **upstream backup**:



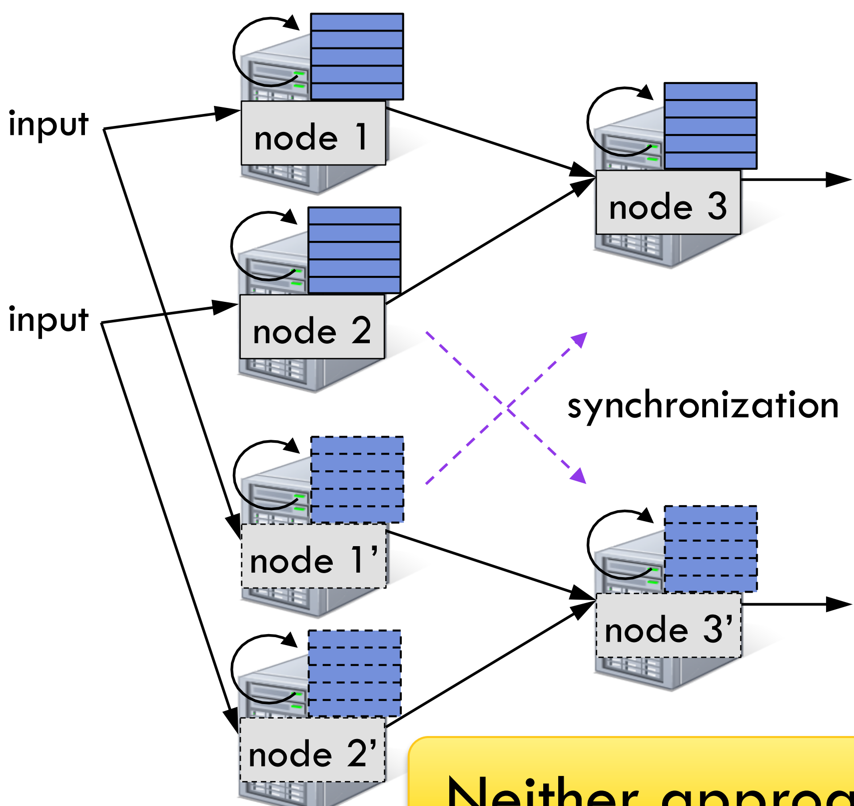


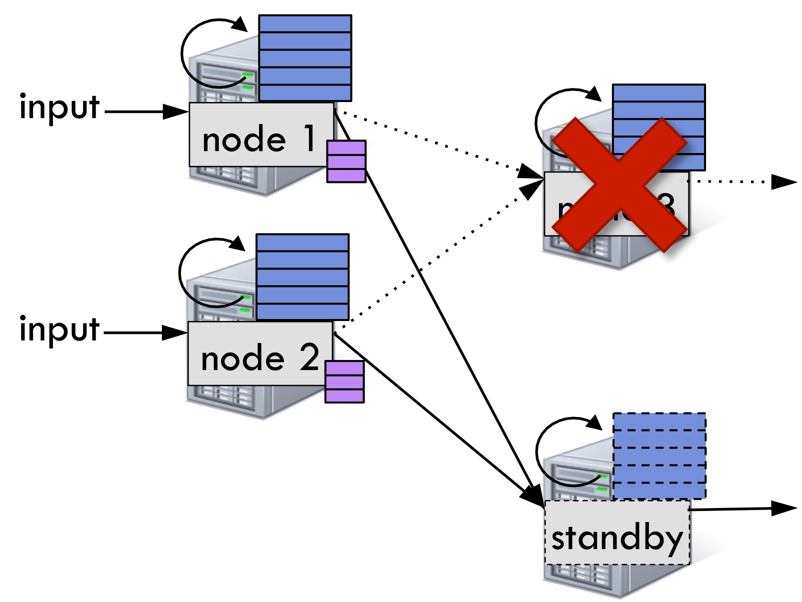
Fast recovery, but 2x hardware cost

Only need 1 standby, but slow to recover

Traditional Streaming Systems

Fault tolerance via **replication** or **upstream backup:**



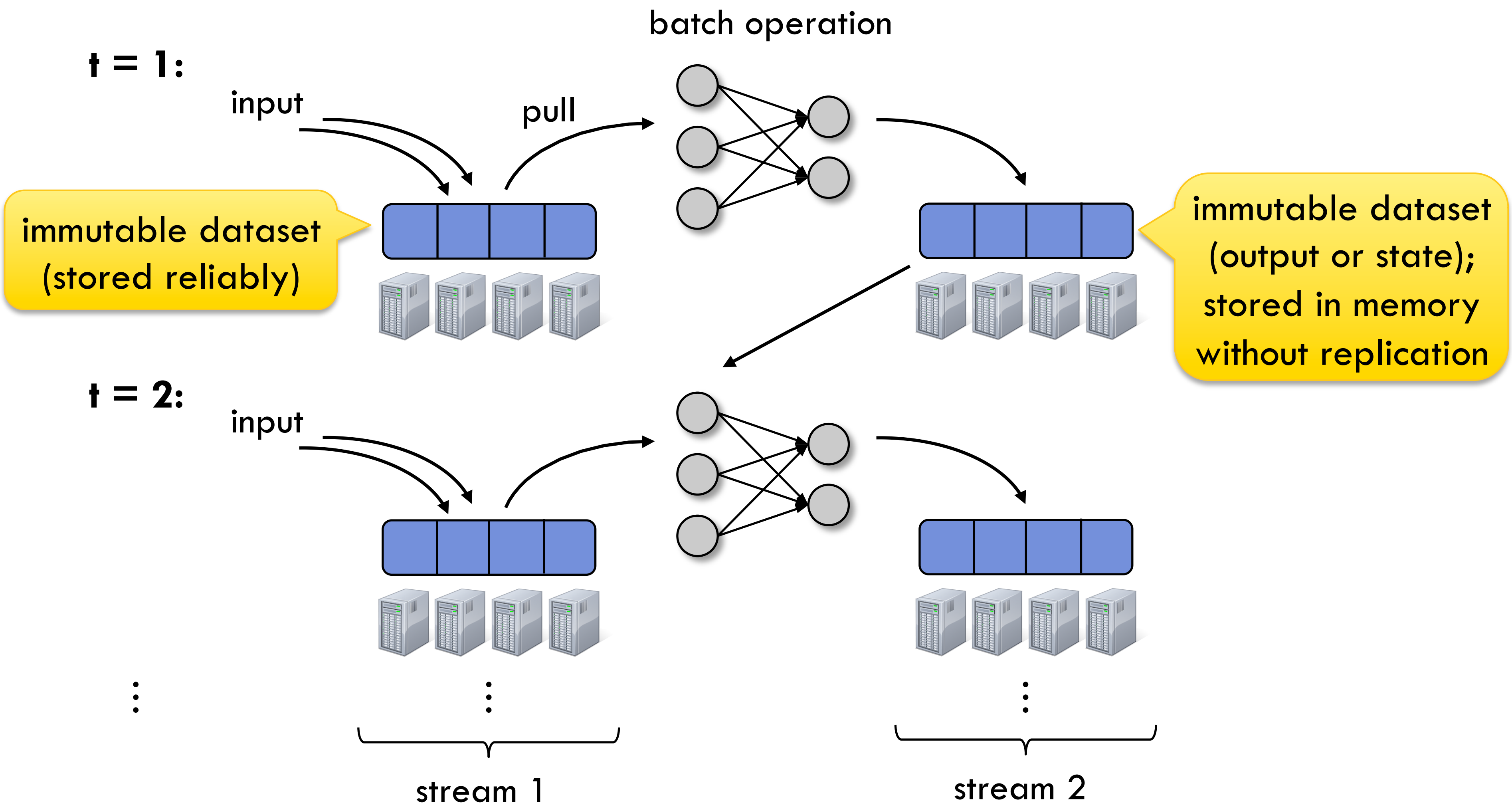


Neither approach tolerates stragglers

Observation

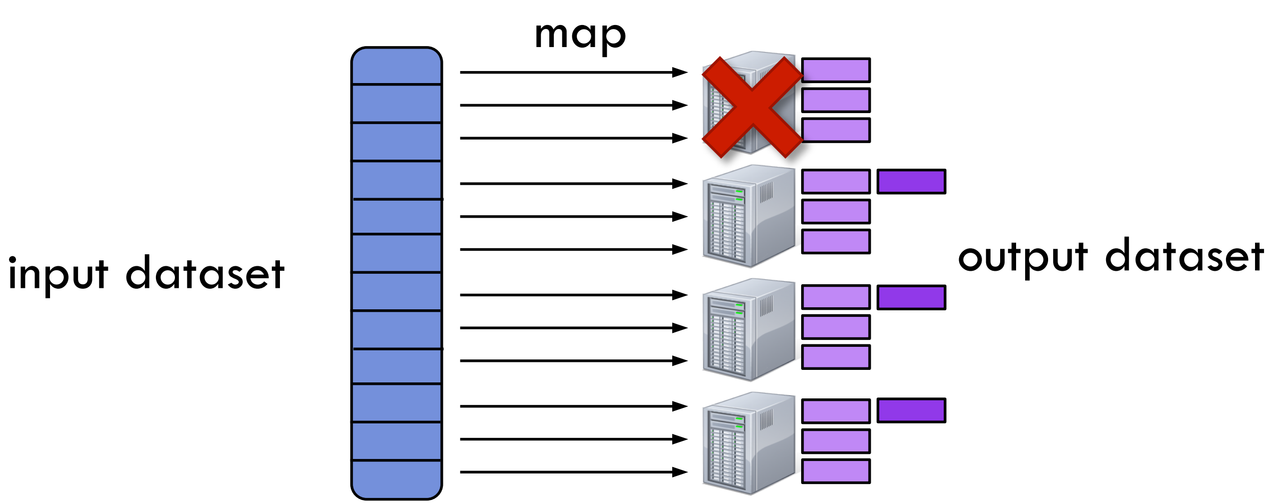
- **Batch** processing models for clusters (e.g. MapReduce) provide fault tolerance efficiently
  - Divide job into deterministic tasks
  - Rerun failed/slow tasks in parallel on other nodes
- Idea: run a streaming computation as a series of very small, deterministic batches
  - Same recovery schemes at much smaller timescale
  - Work to make batch size as small as possible

Discretized Stream Processing



Parallel Recovery

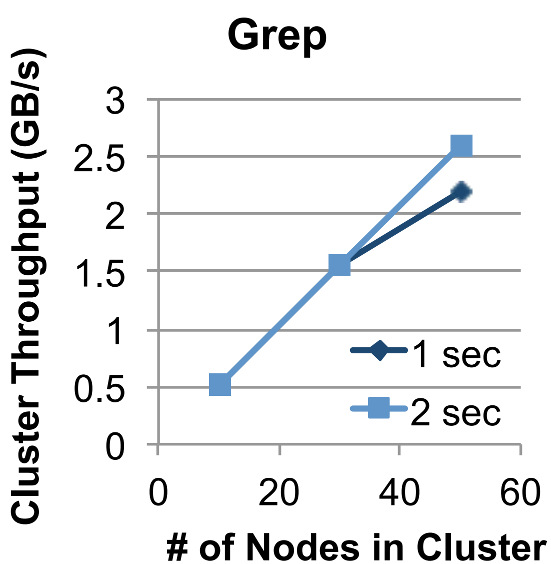
- Checkpoint state datasets periodically
- If a node fails/straggles, recompute its dataset partitions **in parallel** on other nodes

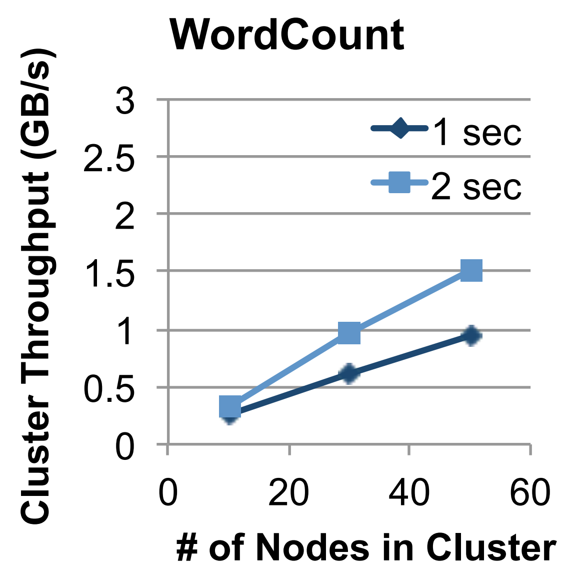


Faster recovery than upstream backup, without the cost of replication

How Fast Can It Go?

- Prototype built on the Spark in-memory computing engine can process **2 GB/s (20M records/s)** of data on 50 nodes at **sub-second** latency

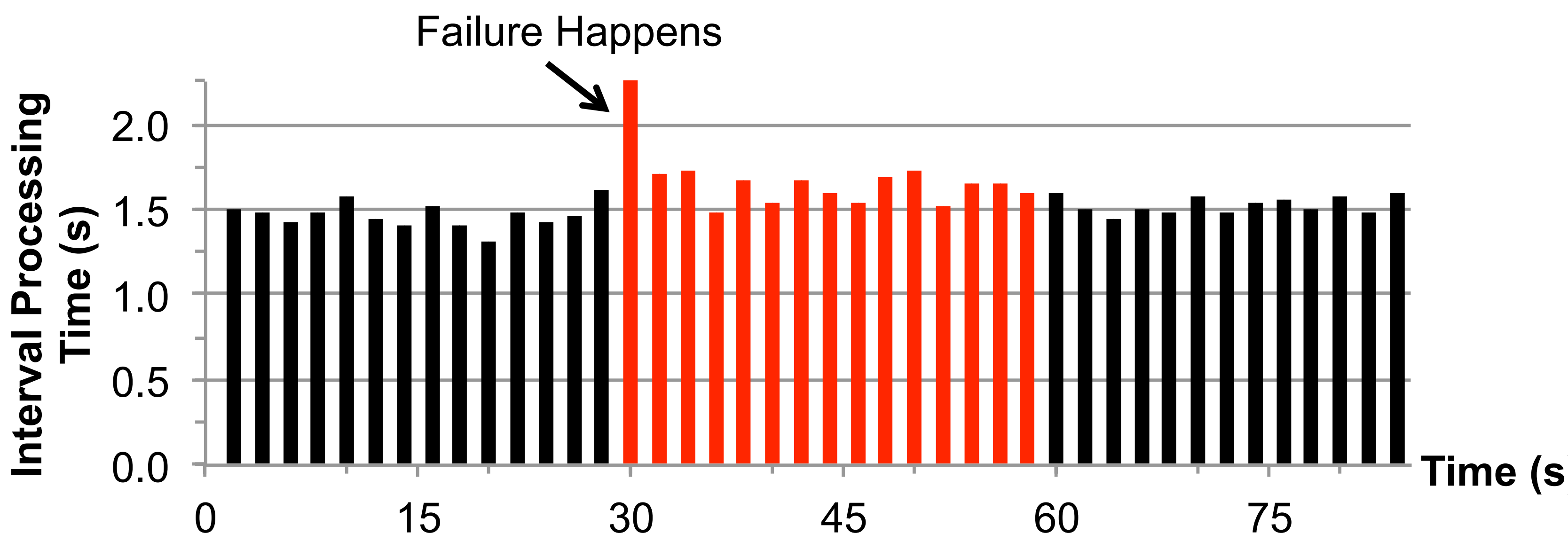




Max throughput within a given latency bound (1 or 2s)

How Fast Can It Go?

- Recovers from failures within **1 second**



Sliding WordCount on 10 nodes with 30s checkpoint interval

Programming Model

- A discretized stream (*D-stream*) is a sequence of immutable, partitioned datasets
  - Specifically, **resilient distributed datasets** (RDDs), the storage abstraction in Spark

- Deterministic *transformations* operators produce new streams

API

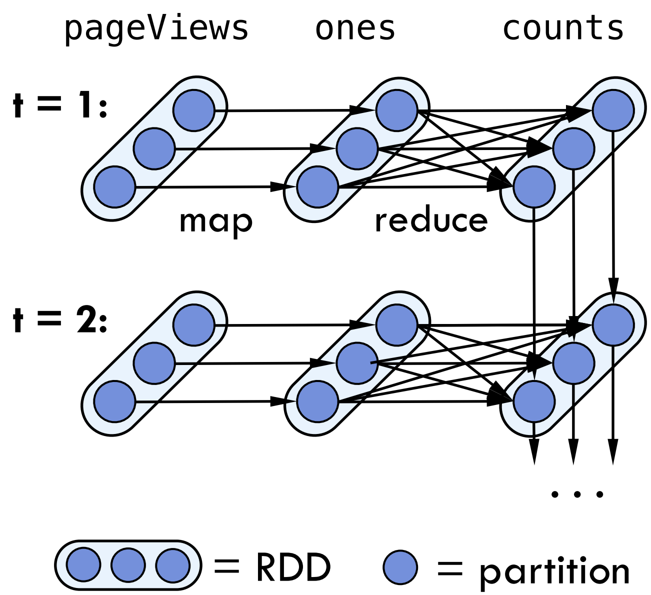
- LINQ-like language-integrated API in Scala
- New "stateful" operators for windowing

```
pageViews = readStream("...", "1s")

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

counts = ones.runningReduce(_ + _)
```

Scala function literal

```
sliding = ones.reduceByWindow(
                  "5s", _ + _, _ - _)
```

Incremental version with "add" and "subtract" functions



Other Benefits of Discretized Streams

- Consistency: each record is processed atomically

- Unification with batch processing:
  - Combining streams with historical data

```
pageViews.join(historicCounts).map(...)
```

  - Interactive ad-hoc queries on stream state

```
pageViews.slice("21:00", "21:05").topK(10)
```

Conclusion

- D-Streams forgo traditional streaming wisdom by **batching** data in small timesteps
- Enable efficient, new parallel recovery scheme
- Let users seamlessly intermix streaming, batch and interactive queries

Related Work

- Bulk incremental processing (CBP, Comet)
  - Periodic (~5 min) batch jobs on Hadoop/Dryad
  - On-disk, replicated FS for storage instead of RDDs
- Hadoop Online
  - Does not recover stateful ops or allow multi-stage jobs
- Streaming databases
  - Record-at-a-time processing, generally replication for FT
- Parallel recovery (MapReduce, GFS, RAMCloud, etc)
  - Hwang et al [ICDE'07] have a parallel recovery protocol for streams, but only allow 1 failure & do not handle stragglers

Timing Considerations

- D-streams group input into intervals based on when records arrive at the system
- For apps that need to group by an "external" time and tolerate network delays, support:
  - **Slack time:** delay starting a batch for a short fixed time to give records a chance to arrive
  - **Application-level correction:** e.g. give a result for time t at time t+1, then use later records to update incrementally at time t+5

D-Streams vs. Traditional Streaming

Concern	Discretized Streams	Record-at-a-time Systems
Latency	0.5–2s	1-100 ms
Consistency	Yes, batch-level	Not in msg. passing systems; some DBs use waiting
Failures	Parallel recovery	Replication or upstream bkp.
Stragglers	Speculation	Typically not handled
Unification with batch	Ad-hoc queries from Spark shell, join w. RDD	Not in msg. passing systems; in some DBs