SparkStreaming

Large scale near-realtime stream processing

Tathagata Das (**TD**)
UC Berkeley

amplab
UC BERKELEY

Motivation

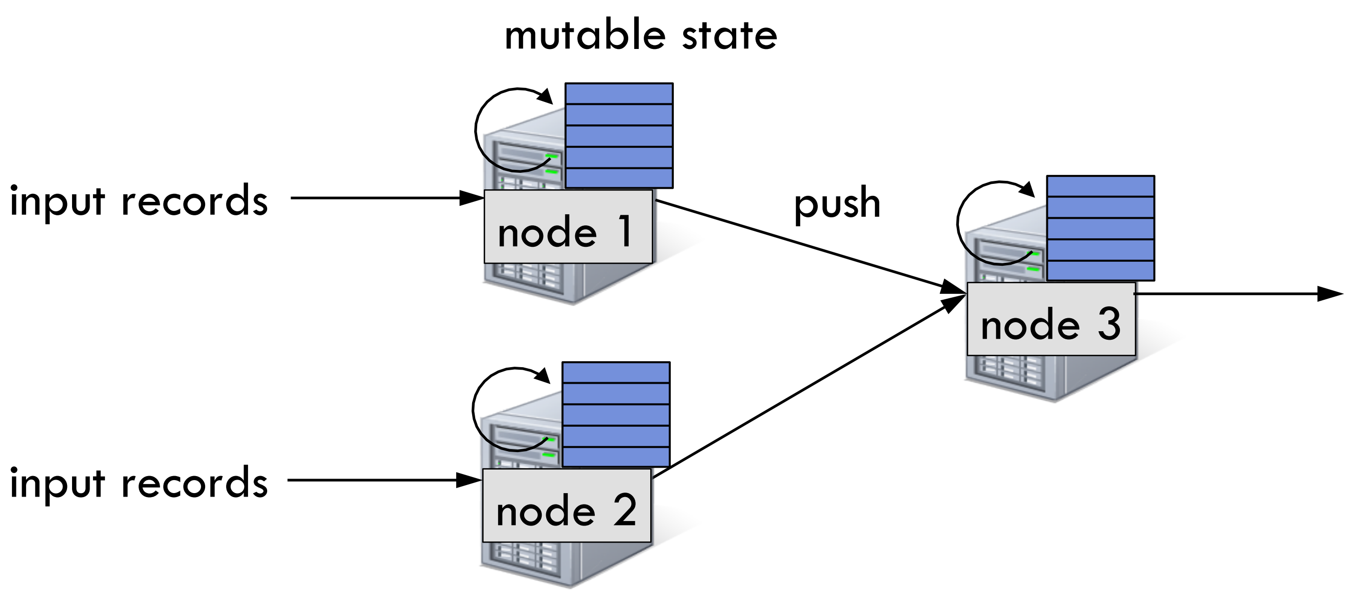
- Many important applications must process large data streams at second-scale latencies
  - Site statistics, intrusion detection, spam filtering,...
- Scaling these apps to 100s of nodes, require ...
  - **Fault-tolerance:** for both
  - **Efficiency:** for being cost-

Current streaming frameworks don't meet both goals together

- Also would like to have ...
  - Simple programming model
  - Integration with batch + ad hoc queries

Traditional Streaming Systems

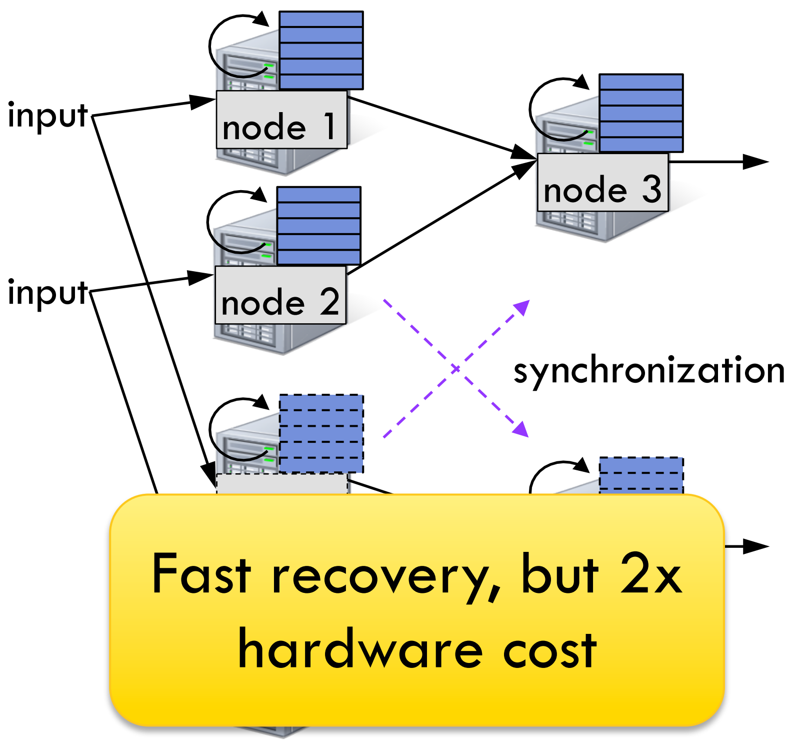
- **Record-at-a-time** processing model
  - Each node has mutable state
  - For each record, update state & send new records



Traditional Streaming Systems

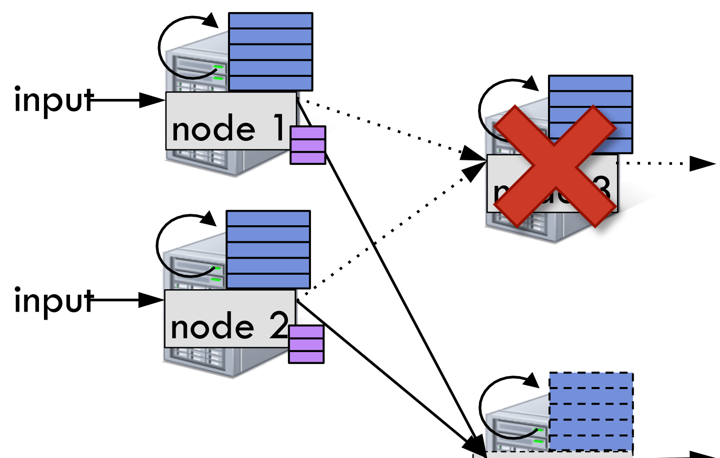
Fault tolerance via *replication* or *upstream backup*

Replication



Fast recovery, but 2x hardware cost

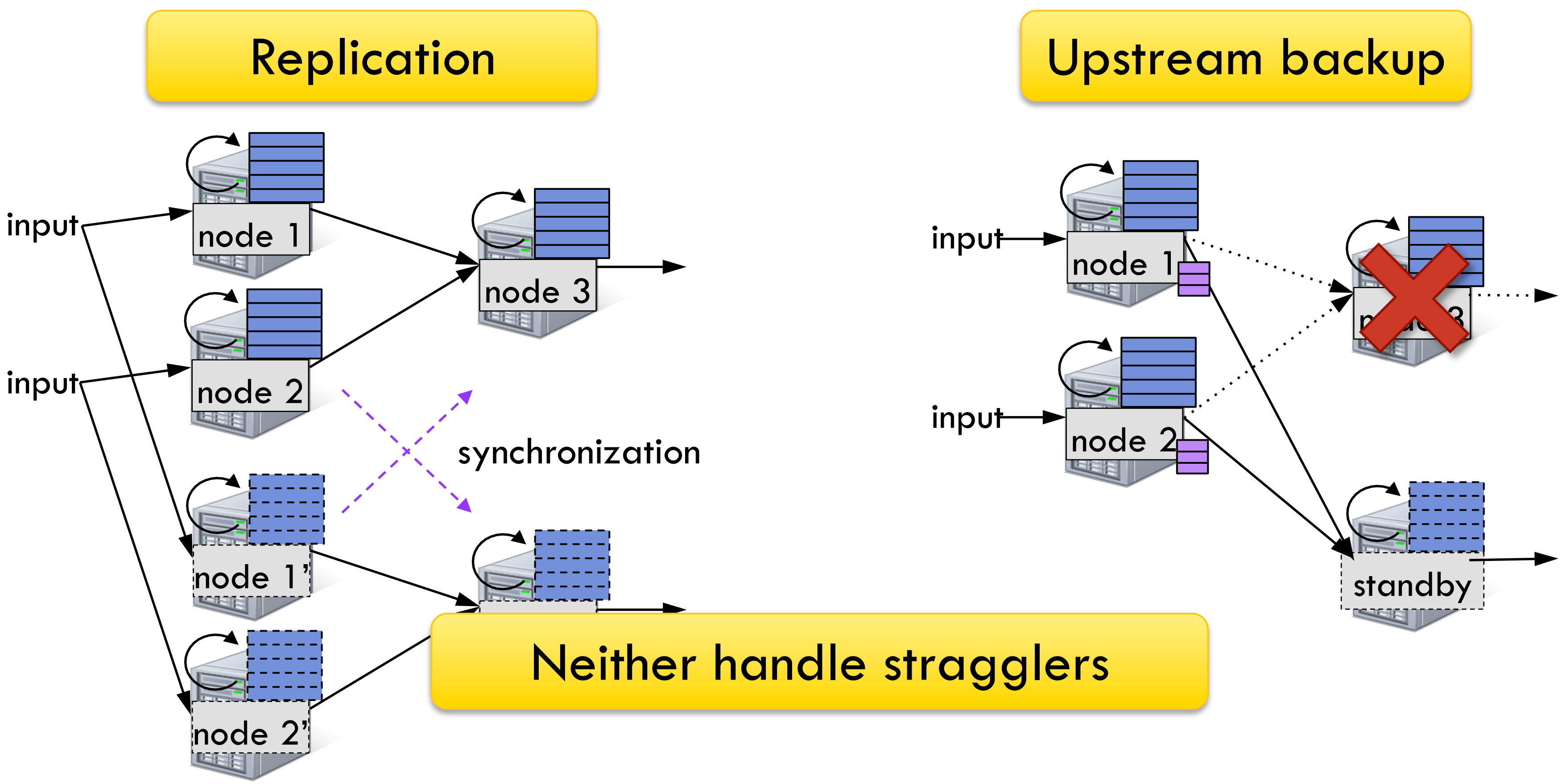
Upstream backup



Only need 1 standby, but slow to recover

Traditional Streaming Systems

Fault tolerance via *replication* or *upstream backup*



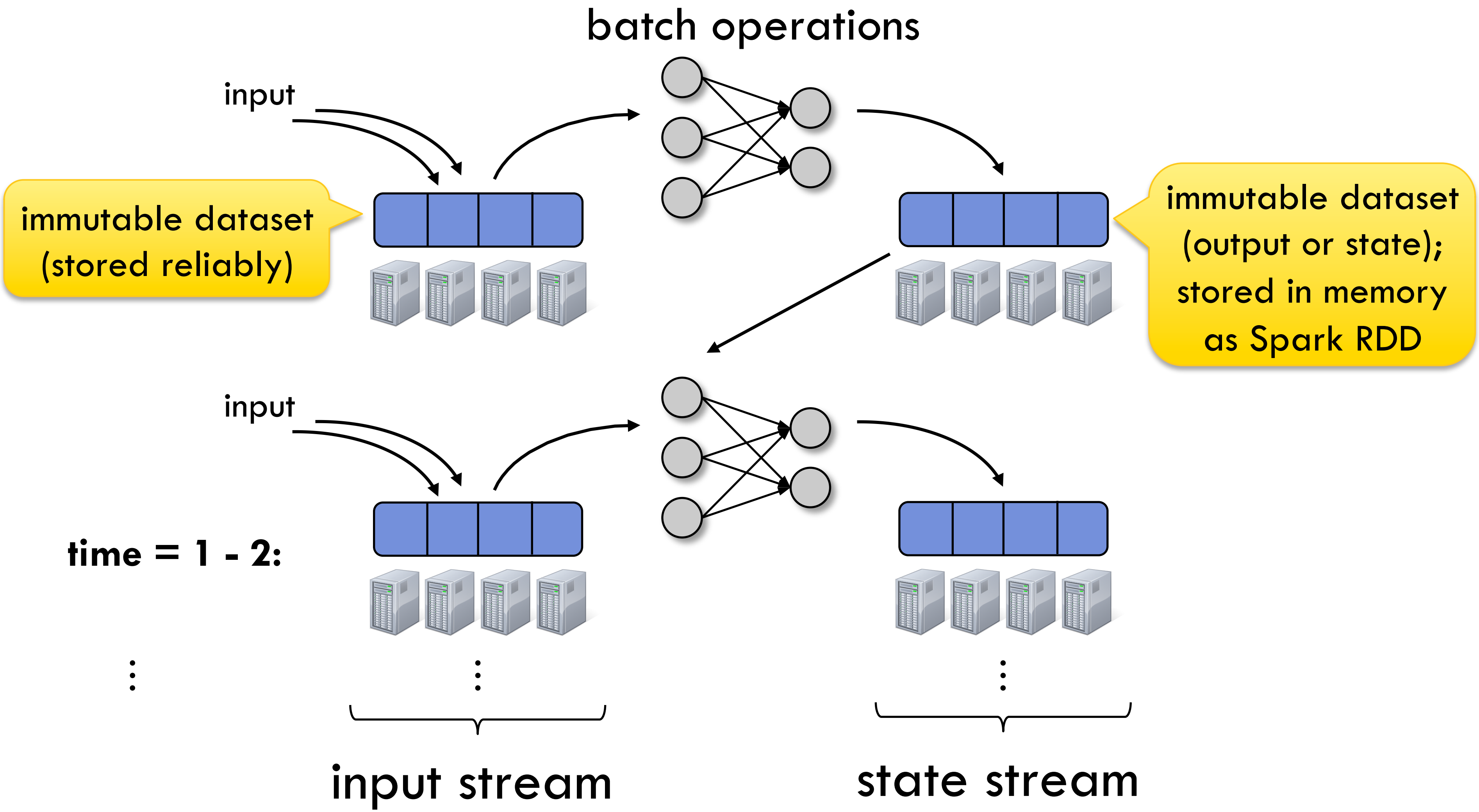
Observation

- Batch processing models, like MapReduce, do provide fault tolerance efficiently
  - Divide job into deterministic tasks
  - Rerun failed/slow tasks in parallel on other nodes

Idea

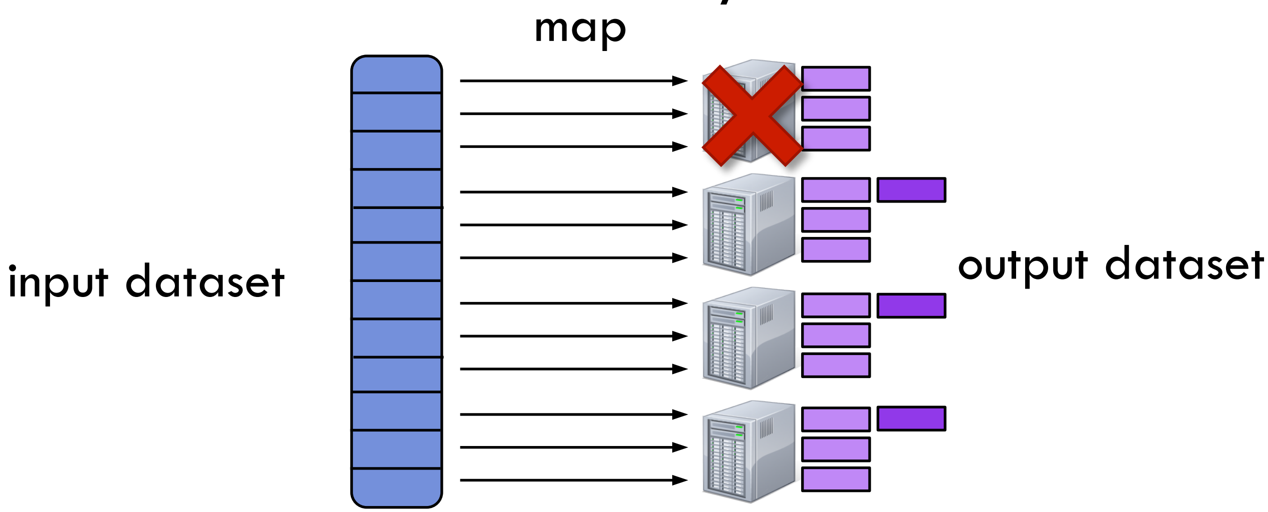
- Idea: run a streaming computation as a **series of very small, deterministic batch jobs**
  - Eg. Process stream of tweets in 1 sec batches
  - Same recovery schemes at smaller timescale
- Try to make batch size as small as possible
  - Lower batch size → lower end-to-end latency
- State between batches kept in memory
  - Deterministic stateful ops → fault-tolerance

Discretized Stream Processing



Fault Recovery

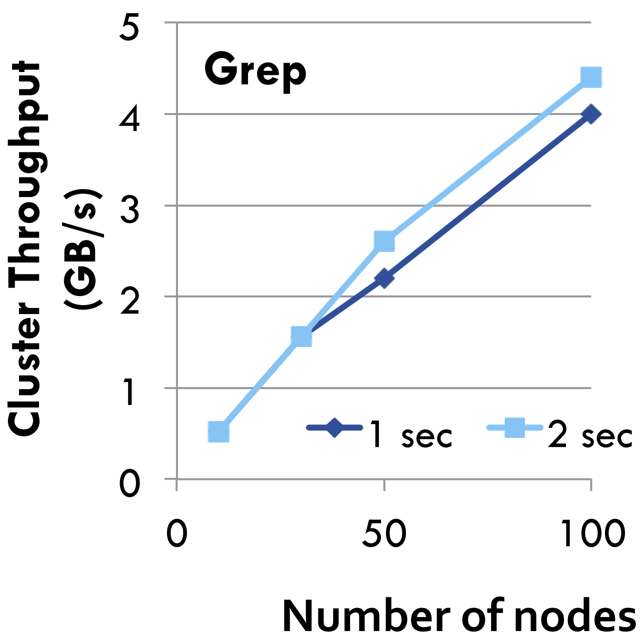
- All dataset modeled as RDDs with dependency graph → fault-tolerant with full replication
- Fault/straggler recovery is done **in parallel** on other nodes → fast recovery

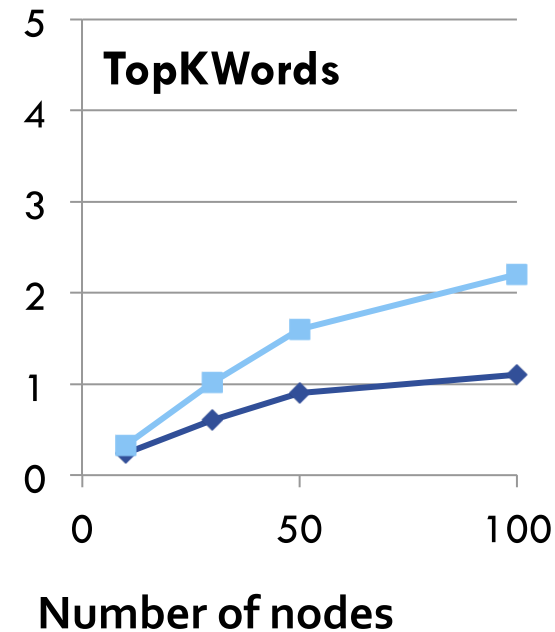


Fast recovery without the cost of full replication

How Fast Can It Go?

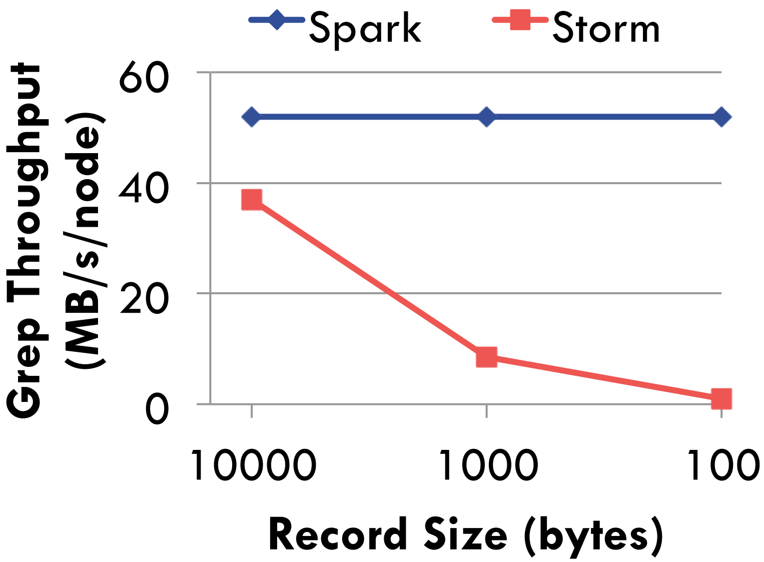
- Prototype can process **4 GB/s** (**40M records/s**) of data on 100 nodes at **sub-second** latency

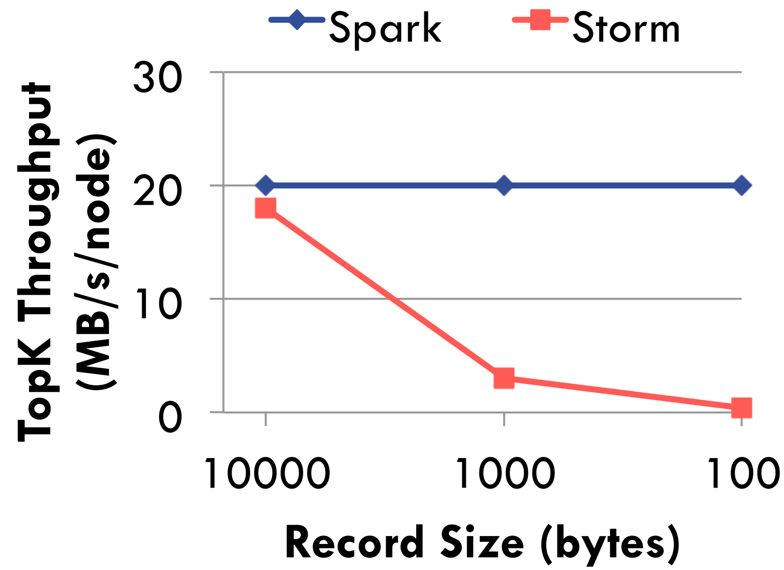




Max throughput with a given latency bound (1 or 2s)

Comparison with Storm

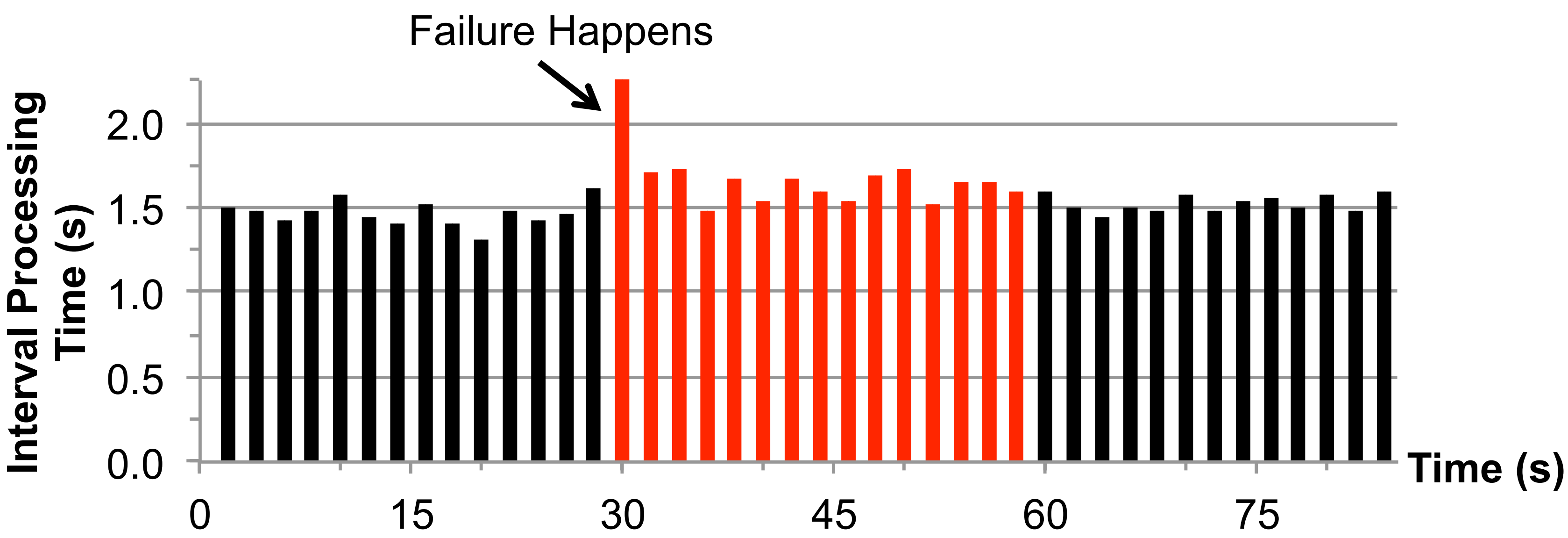




- Storm limited to 10,000 records/s/node
- Also tried Apache S4: 7000 records/s/node
- Commercial systems report O(100K)

How Fast Can It Recover?

- Recovers from faults/stragglers within **1 sec**



Sliding WordCount on 10 nodes with 30s checkpoint interval

Programming Interface

- A Discretized Stream or **DStream** is a sequence of RDDs
  - Represents a stream of data
  - API *very similar* to RDDs

- DStreams can be created…
  - Either from live streaming data
  - Or by transforming other DStreams

DStream Operators

- *Transformations*
  - Build new streams from existing streams
  - Existing RDD ops (map, etc) + new “stateful” ops

- *Output operators*
  - Send data to outside world (save results to external storage, print to screen, etc)

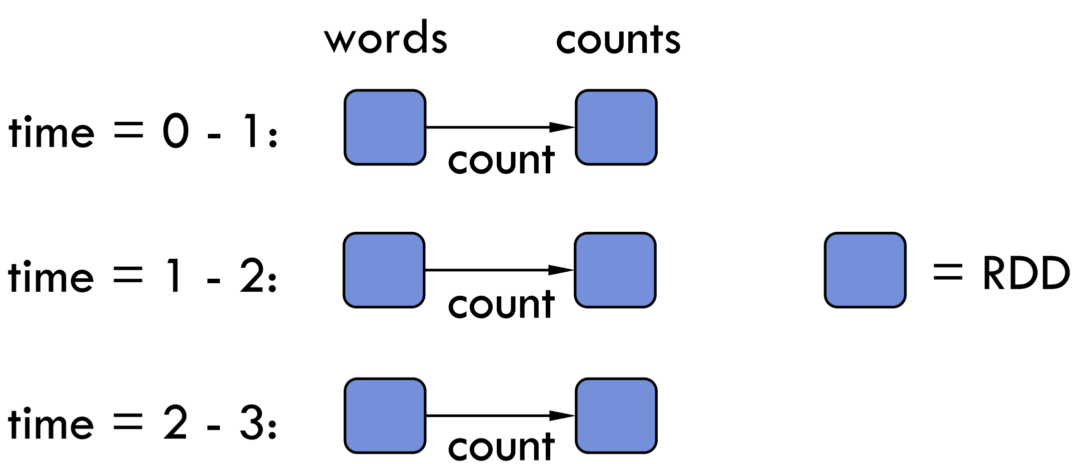
Example 1

Count the words received every second

```
words = createNetworkStream("http://...")
counts = words.count()
```

DStreams

transformation

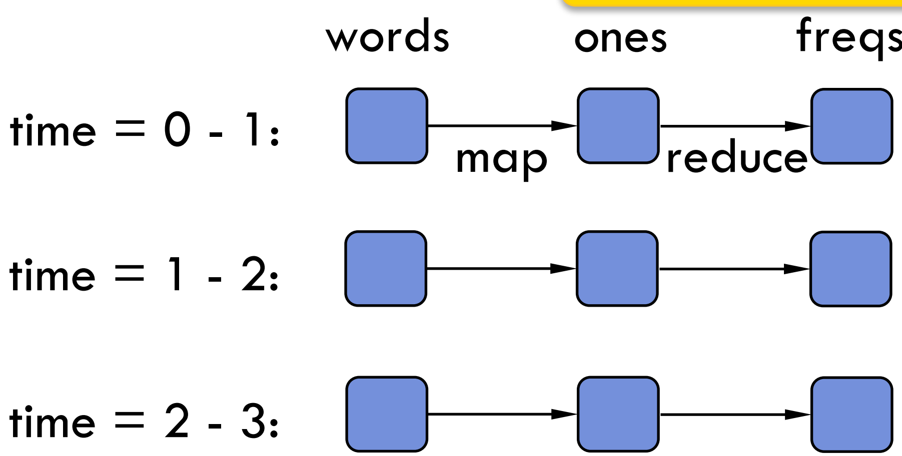


Example 2

Count frequency of words received every second

```
words = createNetworkStream("http://...”)

ones = words.map(w => (w, 1))

freqs = ones.reduceByKey(_ + _)
```

Scala function literal



Example 2: Full code

```
// Create the context and set the batch size
val ssc = new SparkStreamContext("local", "test")
ssc.setBatchDuration(Seconds(1))


// Process a network stream
val words = ssc.createNetworkStream("http://...")
val ones = words.map(w => (w, 1))
val freqs = ones.reduceByKey(_ + _)
freqs.print()


// Start the stream computation
ssc.run
```

Example 3

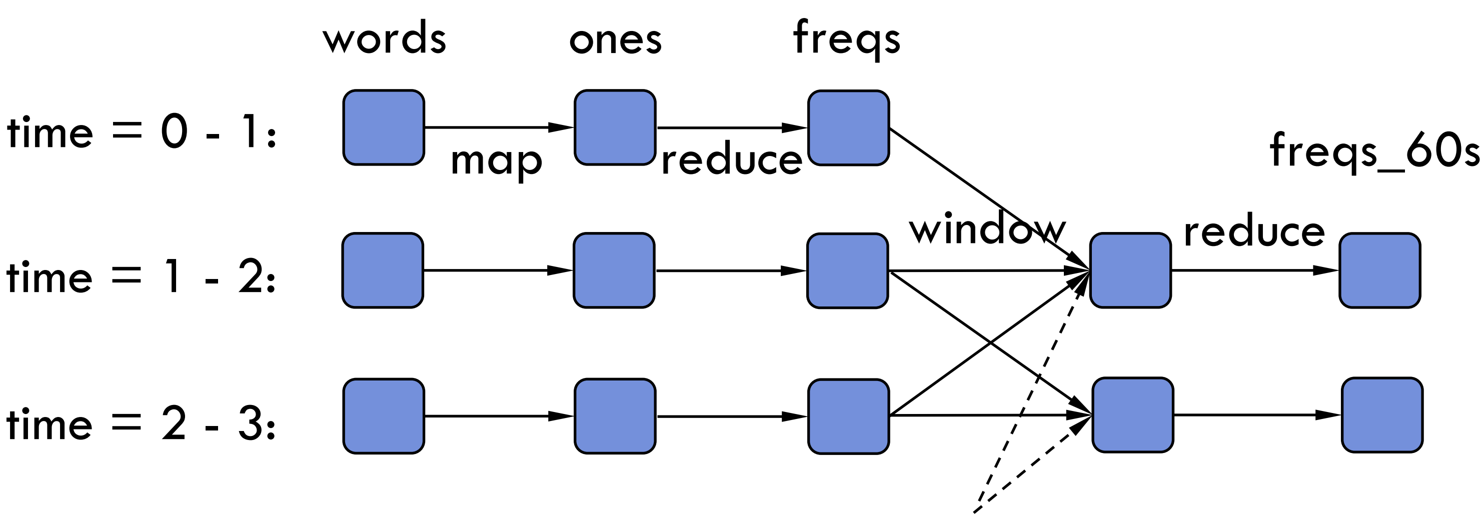
Count frequency of words received in last minute

```
ones = words.map(w => (w, 1))

freqs = ones.r

freqs_60s = freqs.window(Seconds(60), Second(1))

                     reduceByKe          )
```

sliding window operator

window length

window movement

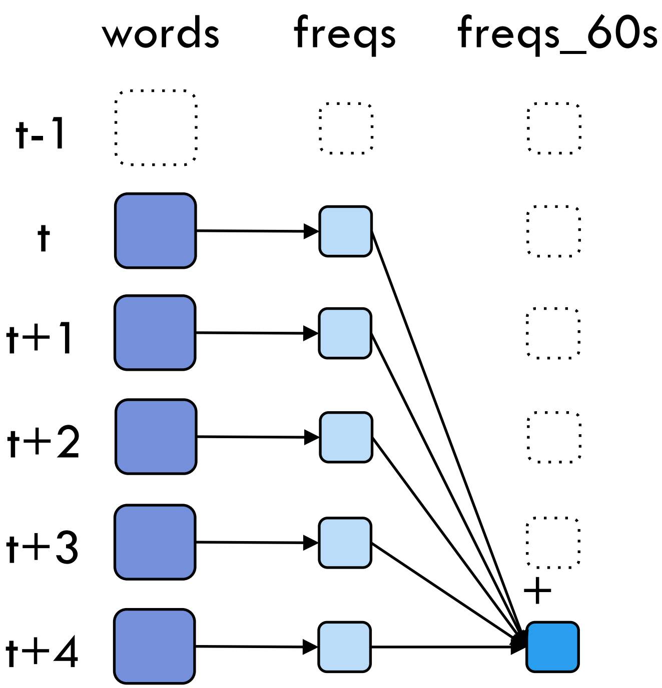


Simpler window-based reduce

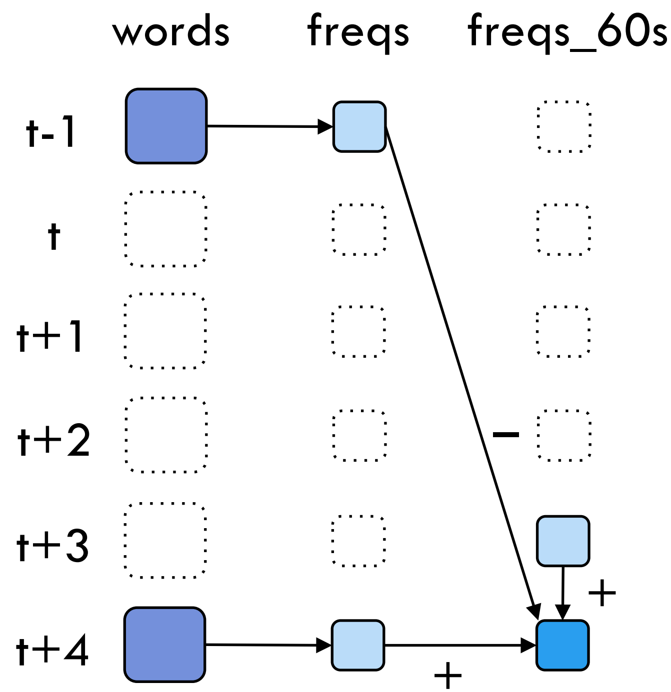
```
freqs = ones.reduceByKey(_ + _)

freqs_60s = freqs.window(Seconds(60), Second(1))
                 .reduceByKey(_ + _)
```

```
freqs = ones.reduceByKeyAndWindow(_ + _, Seconds(60), Seconds(1))
```

“Incremental” window operators



Aggregation function



Invertible aggregation function

Smarter window-based reduce

```
freqs = ones.reduceByKey(_ + _)

freqs_60s = freqs.window(Seconds(60), Second(1))
                 .reduceByKey(_ + _)
```

```
freqs = ones.reduceByKeyAndWindow(_ + _, Seconds(60), Seconds(1))
```

```
freqs = ones.reduceByKeyAndWindow(
                   _ + _, _ - _, Seconds(60), Seconds(1))
```

Output Operators

- *save*: write results to any Hadoop-compatible storage system (e.g. HDFS, HBase)

```
freqs.save("hdfs://...")
```

- *foreachRDD*: run a Spark function on each RDD

```
freqs.foreachRDD(freqsRDD => {
    // any Spark/Scala processing, maybe save to database
})
```

Live + Batch + Interactive

- Combining DStreams with historical datasets

```
freqs.join(oldFreqs).map(...)
```

- Interactive queries on stream state from the Spark interpreter

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

One stack to rule them all

- The promise of a unified data analytics stack
  - Write algorithms only once
  - Cuts complexity of maintaining separate stacks for live and batch processing
  - Query live stream state instead of waiting for import

- Feedback very exciting

- Some recent experiences …

Implementation on Spark

- Optimizations on current Spark
  - Optimized scheduling for < 100ms tasks
  - New block store with fast NIO communication
  - Pipelining of jobs from different time intervals

- Changes already in dev branch of Spark on http://www.github.com/mesos/spark

- An alpha will be released with Spark 0.6 soon

More Details

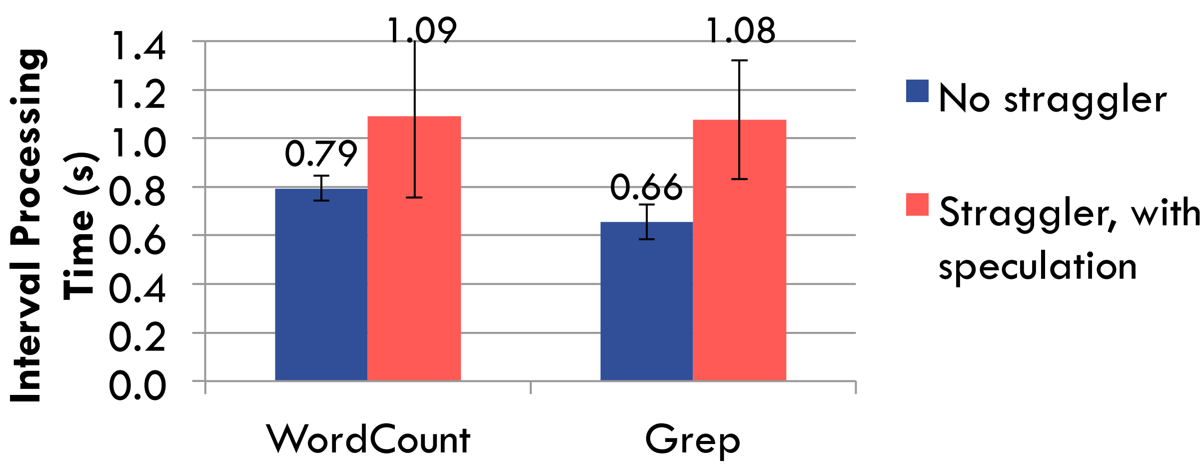
- You can find more about SparkStreaming in our paper: http://tinyurl.com/dstreams

Thank you!

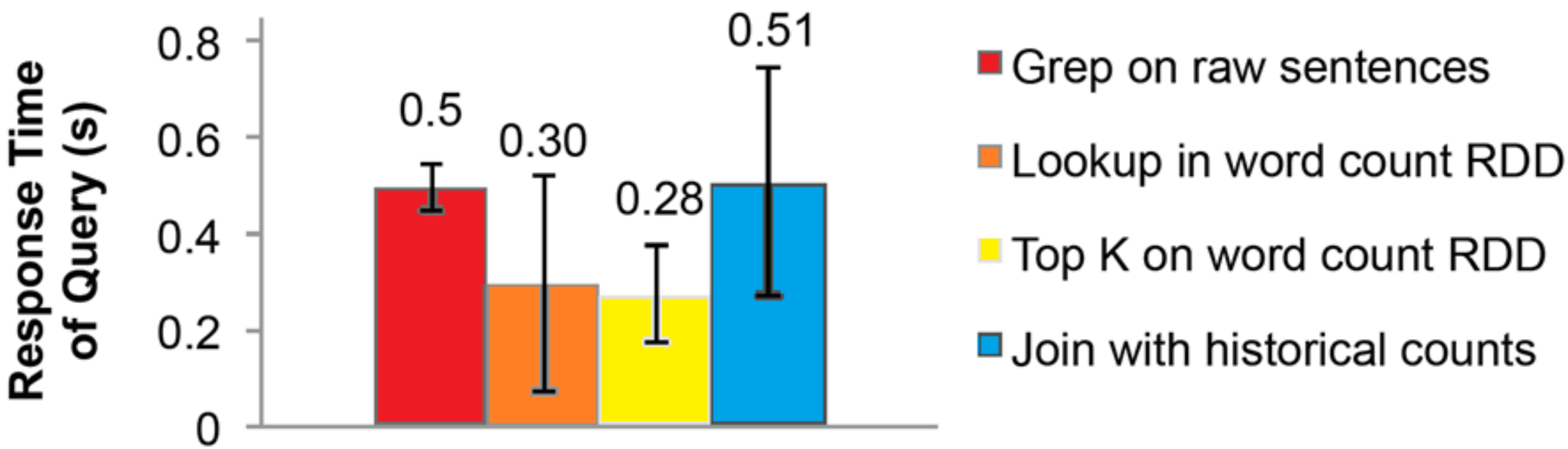
Fault Recovery

Failures:

Stragglers:



Interactive Ad-Hoc 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
- Approximate query processing, load shedding
  - Do not support the loss of arbitrary nodes
  - Different math because drop rate is known exactly
- Parallel recovery (MapReduce, GFS, RAMCloud, etc)

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