CSCI 381/780
Cloud Computing

Streaming Analytics

Jun Li
Queens College

QUEENS
COLLEGE

MapReduce

- Batch Processing => Need to wait for entire computation on large dataset to complete
- Not intended for long-running stream-processing

Challenges

- Large amounts of data => Need for real-time views of data
  - Social network trends, e.g., Twitter real-time search
  - Website statistics, e.g., Google Analytics
  - Intrusion detection systems, e.g., in most data centers

- Process large amounts of data
  - With latencies of few seconds
  - With high throughput

Storm

- Apache Project
- http://storm.apache.org/
- Highly active JVM project
- Multiple languages supported via API
  - Python, Ruby, etc.

- Used by over 30 companies including
  - Twitter: For personalization, search
  - Flipboard: For generating custom feeds
  - Weather Channel, WebMD, etc.

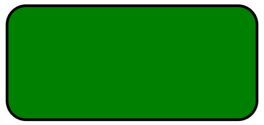
Tuple

- An ordered list of elements

- E.g., <tweeter, tweet>
  - E.g., <“Miley Cyrus”, “Hey! Here’s my new song!”>
  - E.g., <“Justin Bieber”, “Hey! Here’s MY new song!”>

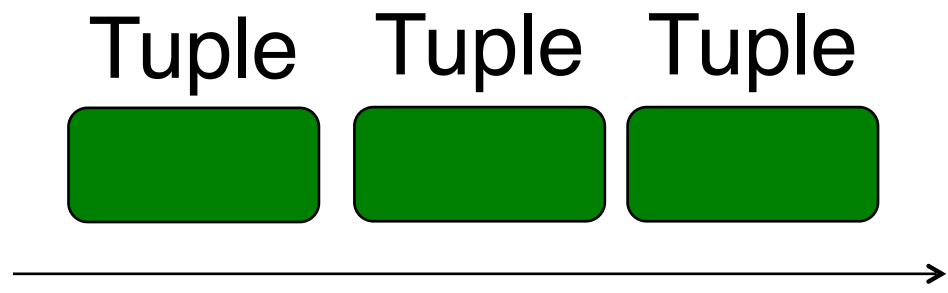
- E.g., <URL, clicker-IP, date, time>
  - E.g., <coursera.org, 101.102.103.104, 4/4/2014, 10:35:40>
  - E.g., <coursera.org, 101.102.103.105, 4/4/2014, 10:35:42>

Tuple



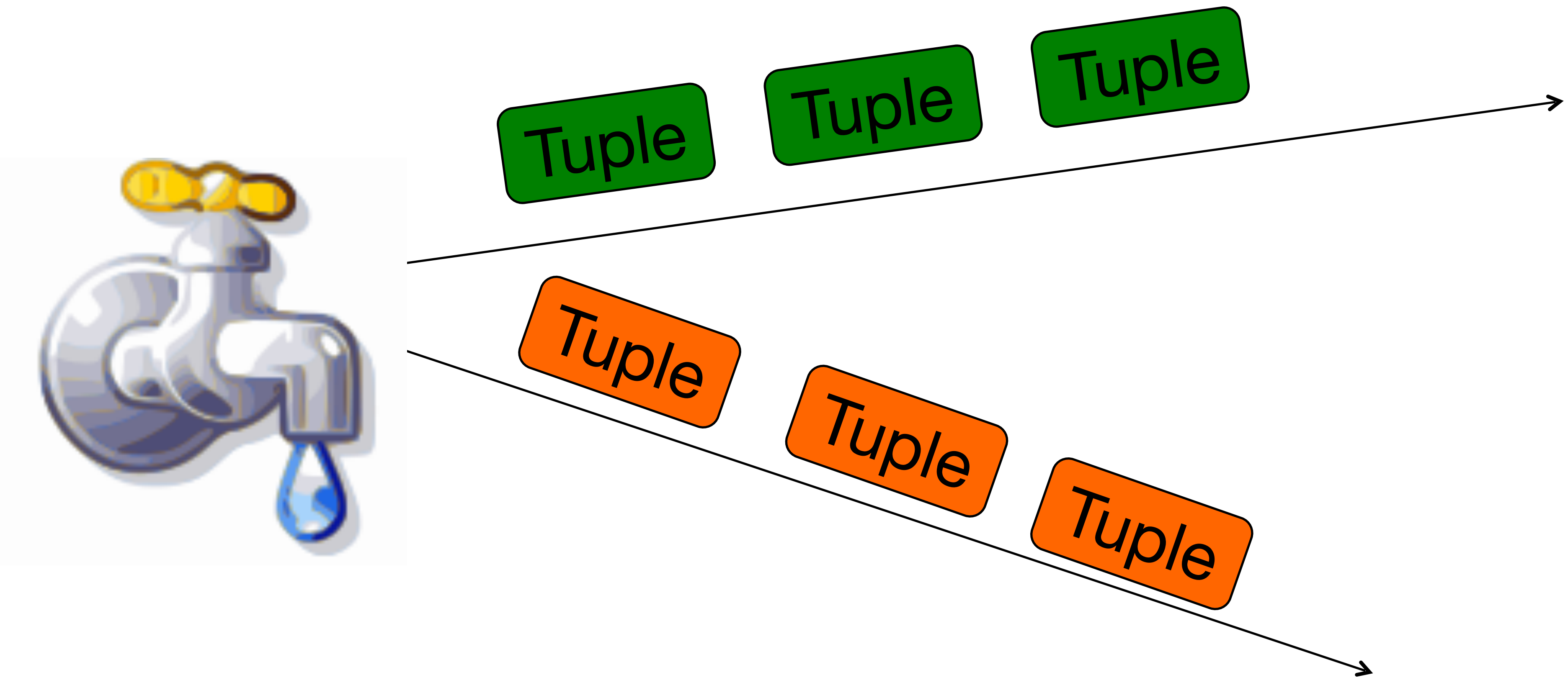
Stream

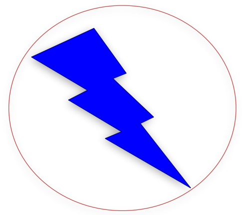
- Sequence of tuples
  - Potentially unbounded in number of tuples
- Social network example:
  - <“Miley Cyrus”, “Hey! Here’s my new song!”>,
    <“Justin Bieber”, “Hey! Here’s MY new song!”>,
    <“Rolling Stones”, “Hey! Here’s my old song that’s still a super-hit!”>, …
- Website example:
  - <coursera.org, 101.102.103.104, 4/4/2014, 10:35:40>, <coursera.org, 101.102.103.105, 4/4/2014, 10:35:42>, …



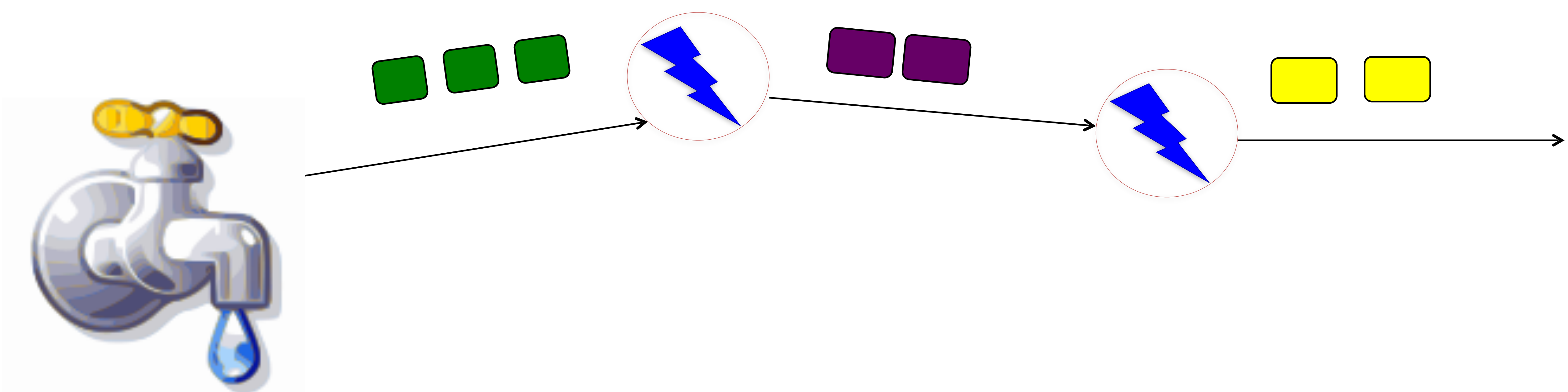
Spout

- A Storm entity (process) that is a source of streams
- Often reads from a crawler or DB



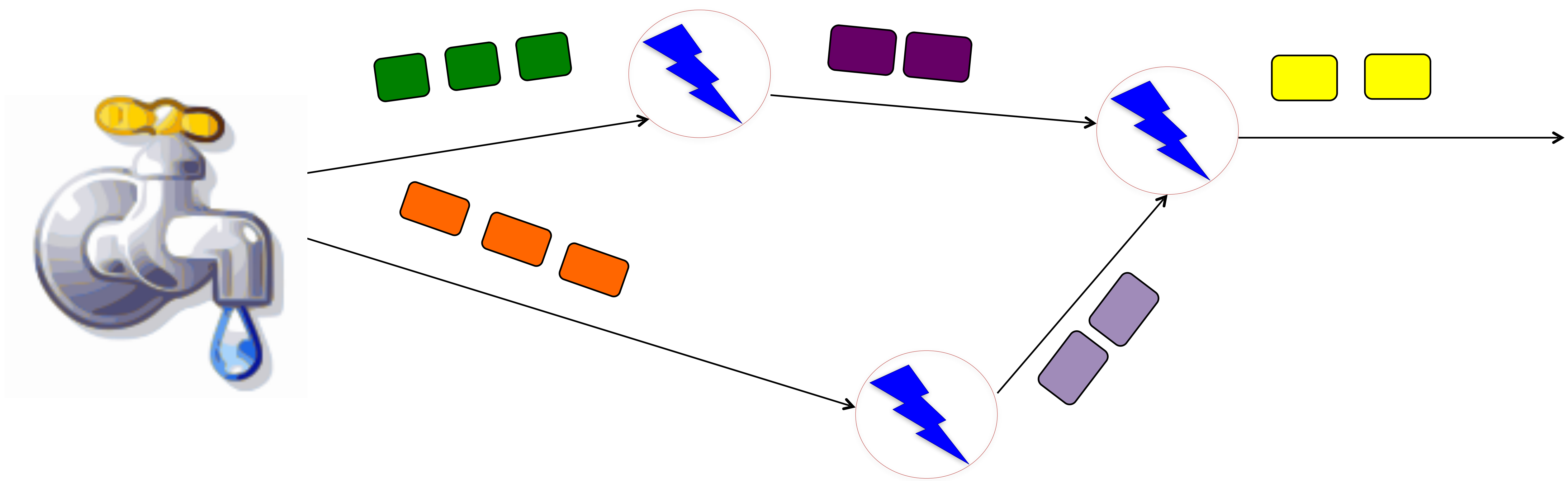
Bolt 

- A Storm entity (process) that
    - Processes input streams
    - Outputs more streams for other bolts



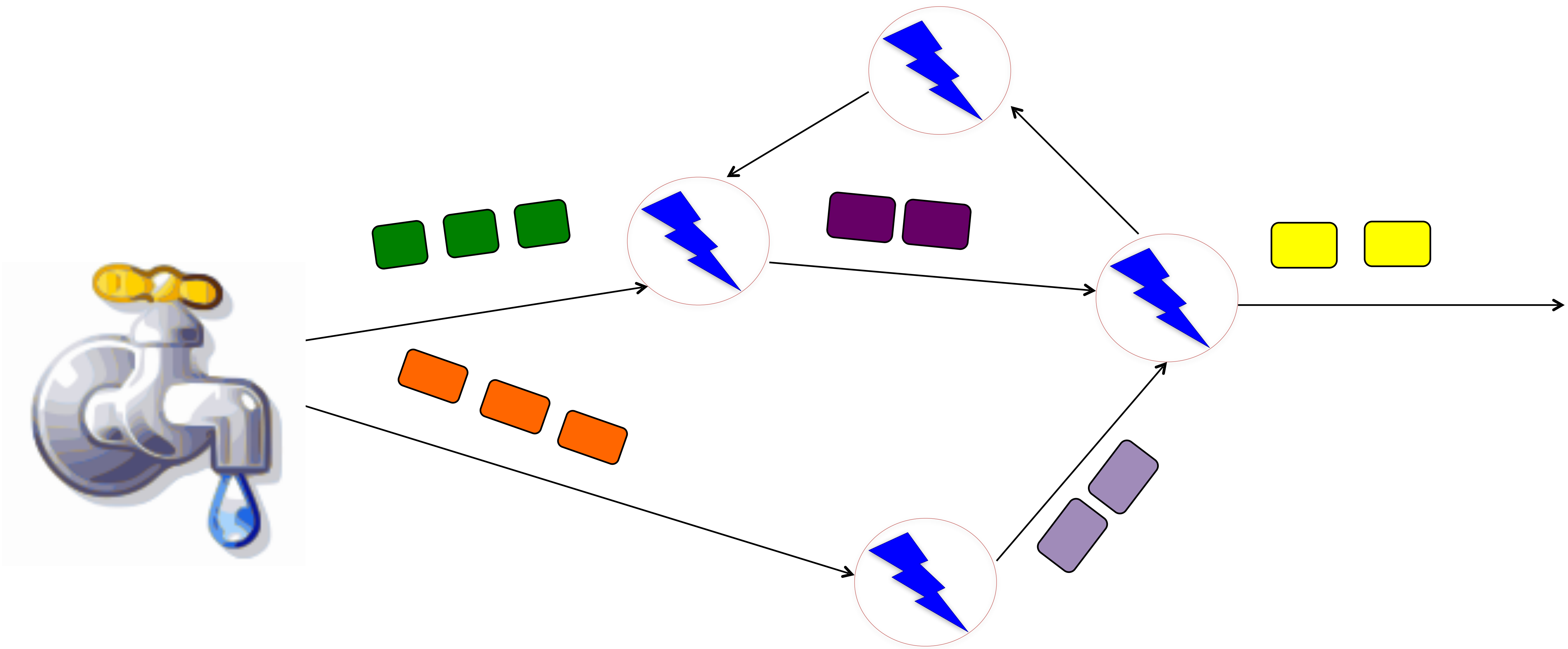
Topology

- A directed graph of spouts and bolts (and output bolts)
- Corresponds to a Storm "application"



Topology

- Can have cycles if the application requires it



Bolts come in many Flavors

- Operations that can be performed
  - Filter: forward only tuples which satisfy a condition
  - Joins: When receiving two streams A and B, output all pairs (A,B) which satisfy a condition
  - Apply/transform: Modify each tuple according to a function
  - And many others

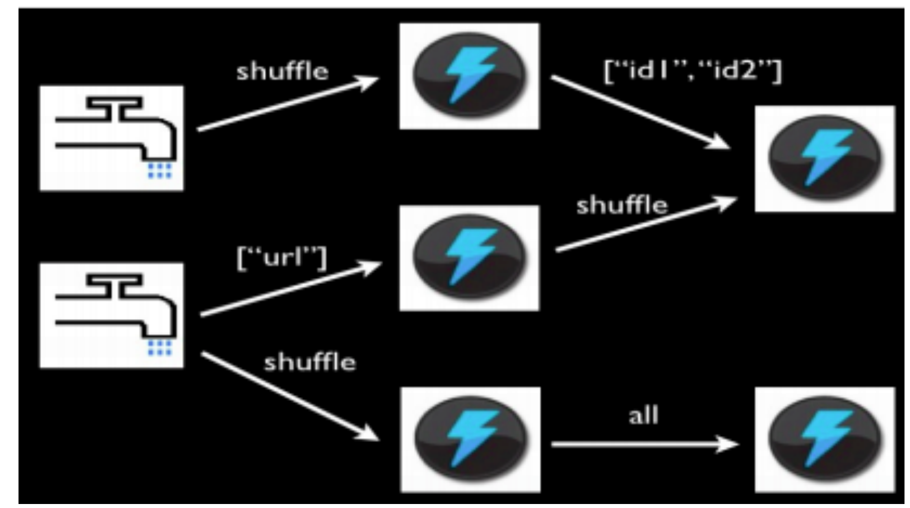
- But bolts need to process a lot of data
  - Need to make them fast

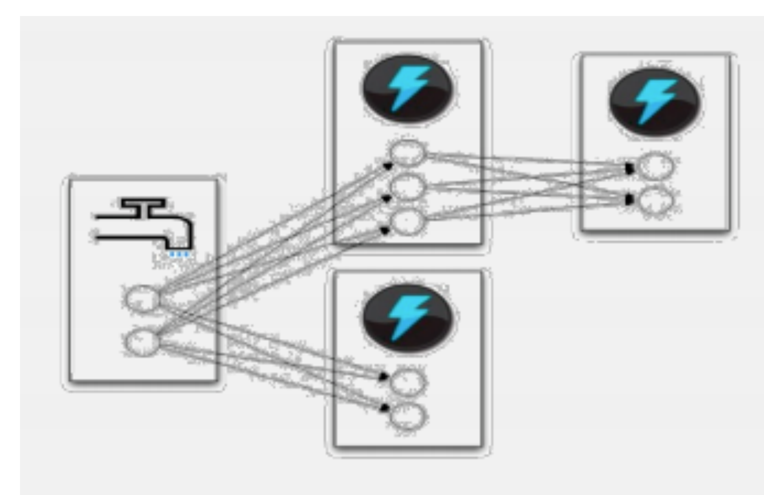
Parallelizing Bolts

- Have multiple processes (“tasks”) constitute a bolt
- Incoming streams split among the tasks
- Typically each incoming tuple goes to one task in the bolt
  - Decided by “Grouping strategy”
- Three types of grouping are popular

Storm Task

- Spouts and bolts execute as many tasks across the cluster
- When a tuple is emitted, which task does it go to? User programmable:
  - Shuffle grouping: pick a random task
  - Fields grouping: consistent hashing on a subset of tuple fields
  - All grouping: send to all tasks
  - Global grouping: pick task with lowest id





Storm Cluster

- Master node
  - Runs a daemon called *Nimbus*
  - Responsible for
    - Distributing code around cluster
    - Assigning tasks to machines
    - Monitoring for failures of machines
- Worker node
  - Runs on a machine (server)
  - Runs a daemon called *Supervisor*
  - Listens for work assigned to its machines
  - Runs “Executors”(which contain groups of tasks)
- Zookeeper
  - Coordinates Nimbus and Supervisors communication
  - All state of Supervisor and Nimbus is kept here

Failures

- A tuple is considered failed when its topology (graph) of resulting tuples fails to be fully processed within a specified timeout

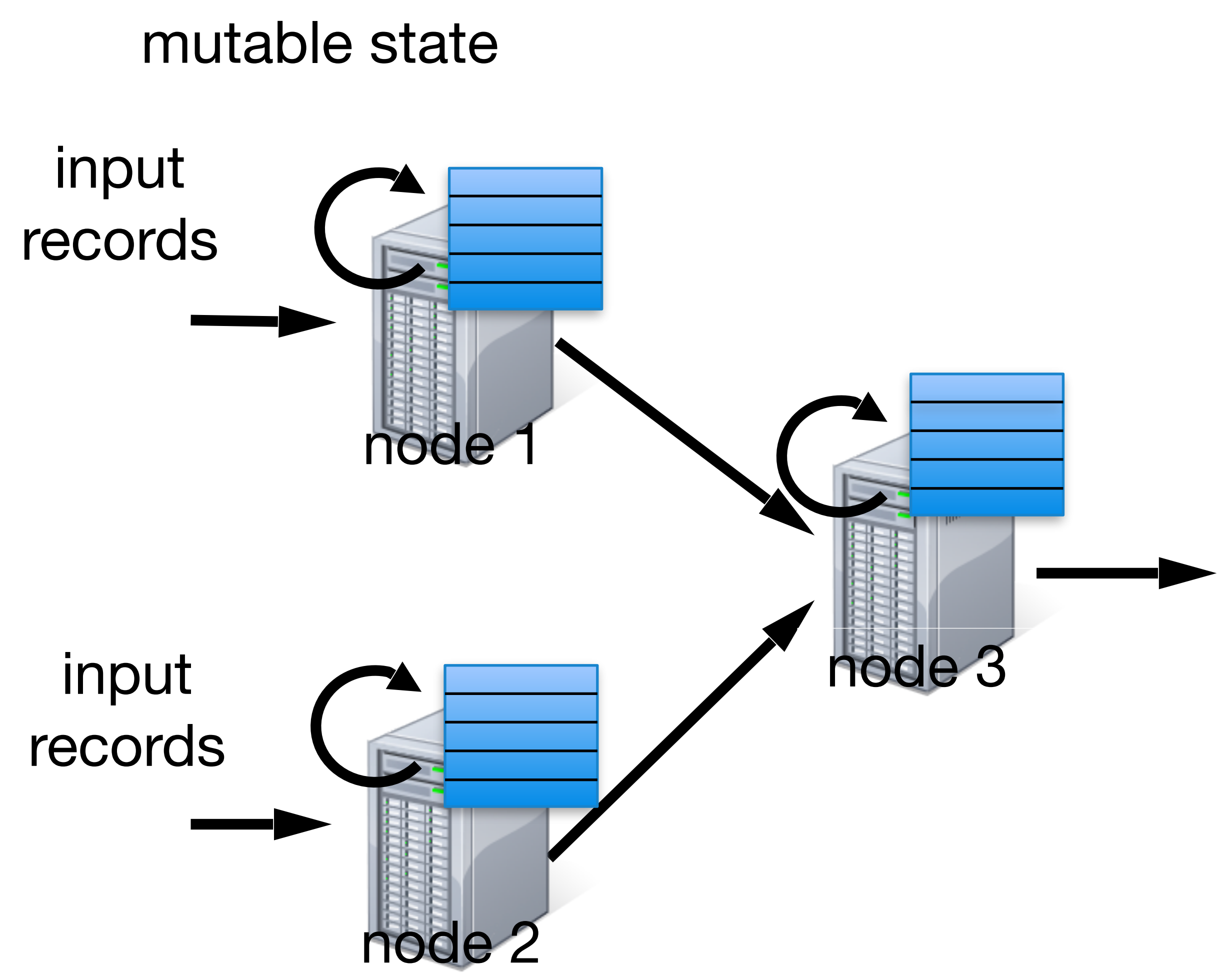
- Anchoring: Anchor an output to one or more input tuples
  - Failure of one tuple causes one or more tuples to replayed

API For Fault-Tolerance (OutputCollector)

- Emit(tuple, output)
  - Emits an output tuple, perhaps anchored on an input tuple (first argument)
- Ack(tuple)
  - Acknowledge that you (bolt) finished processing a tuple
- Fail(tuple)
  - When a tuple fails, Storm will attempt to replay it from its source and reprocess it.
  - If the tuple cannot be replayed or reprocessed, Storm will consider it a permanent failure and remove it from its tracking data structures.
- Must remember to ack/fail each tuple
  - Each tuple consumes memory. Failure to do so results in memory leaks.

Stateful Stream Processing

- Traditional streaming systems have a record-at-a-time processing model
  - Each node has mutable state
  - For each record, update state and send new records



- State is lost if node dies!
- Making stateful stream processing be fault-tolerant is challenging

Existing Streaming Systems

- Storm
  - Replays record if not processed by a node
  - Processes each record *at least once*
  - May update mutable state twice!
  - Mutable state can be lost due to failure!

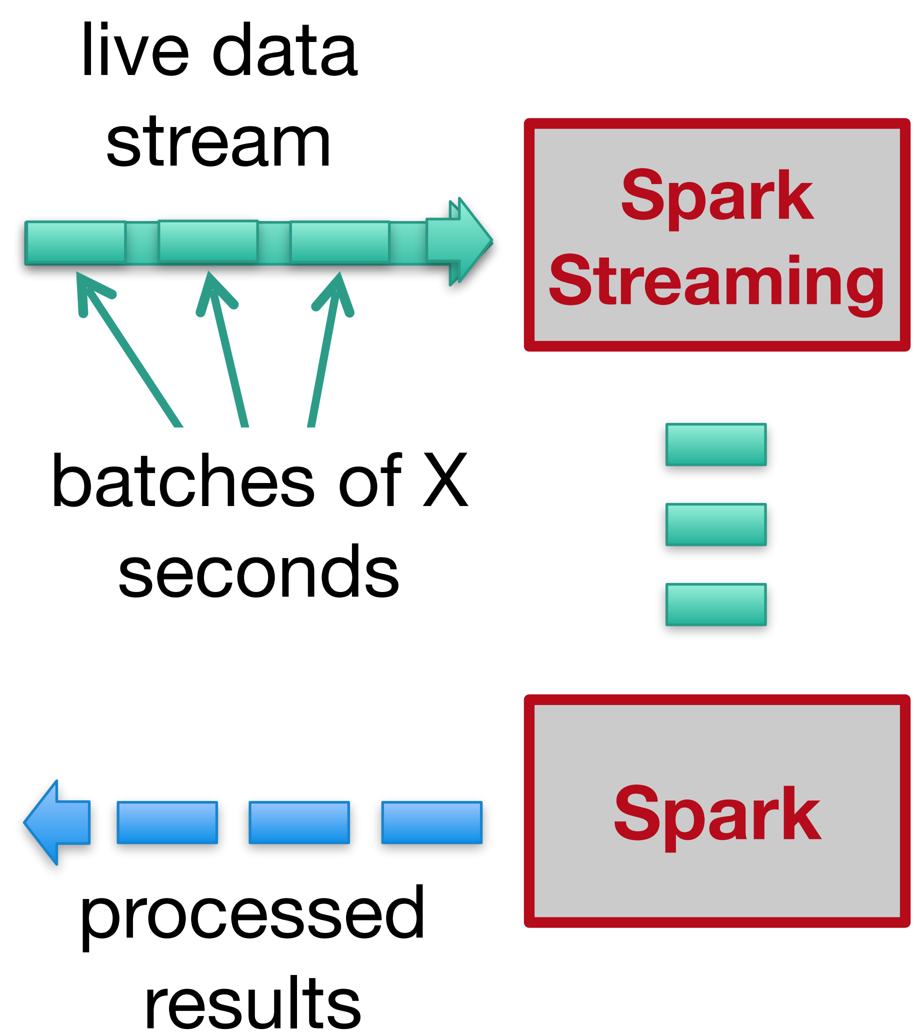
What is Spark Streaming?

- Extends Spark for doing large scale stream processing
- Scales to 100s of nodes and achieves second scale latencies
- Efficient and fault-tolerant stateful stream processing
- Simple batch-like API for implementing complex algorithms

Discretized Stream Processing

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

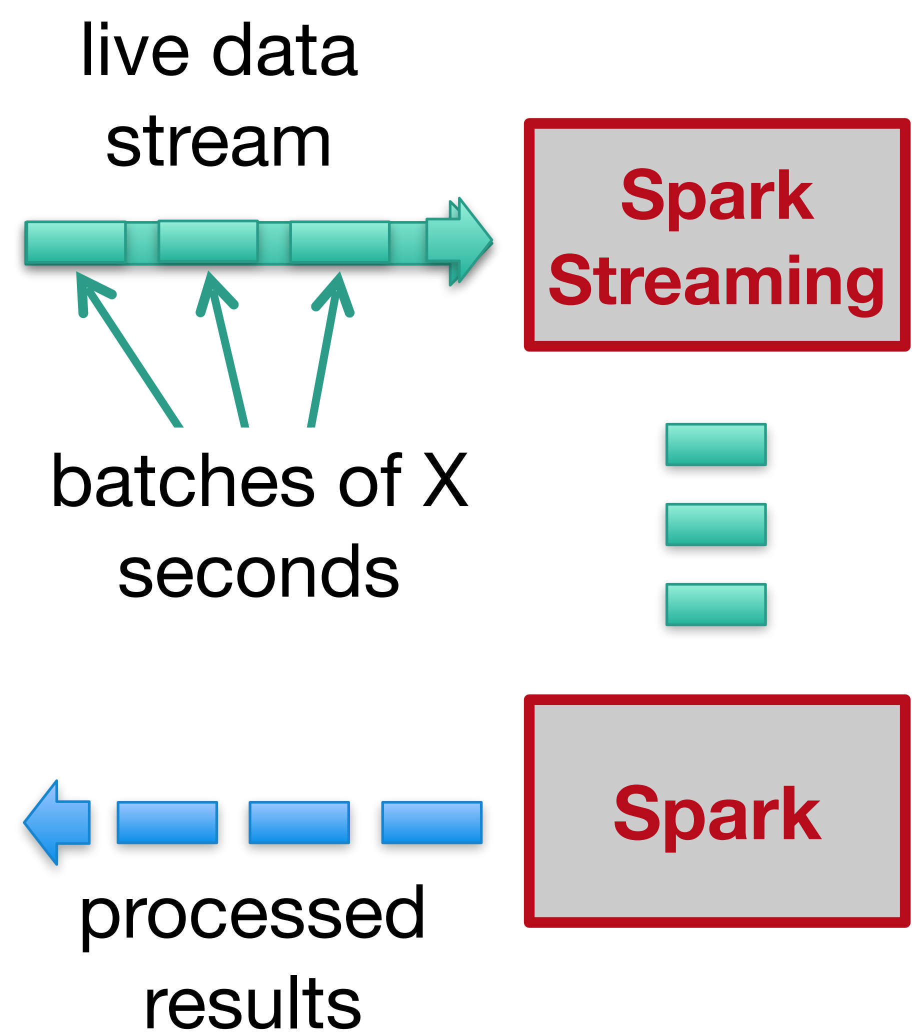
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Discretized Stream Processing

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

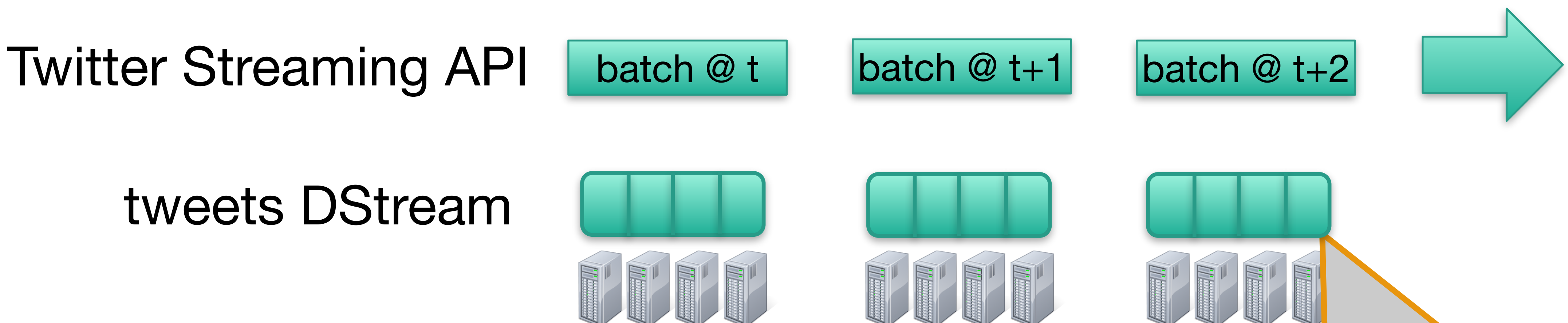
- Batch sizes as low as ½ second, latency of about 1 second
- Potential for combining batch processing and streaming processing in the same system



Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
```

DStream: a sequence of RDDs representing a stream of data



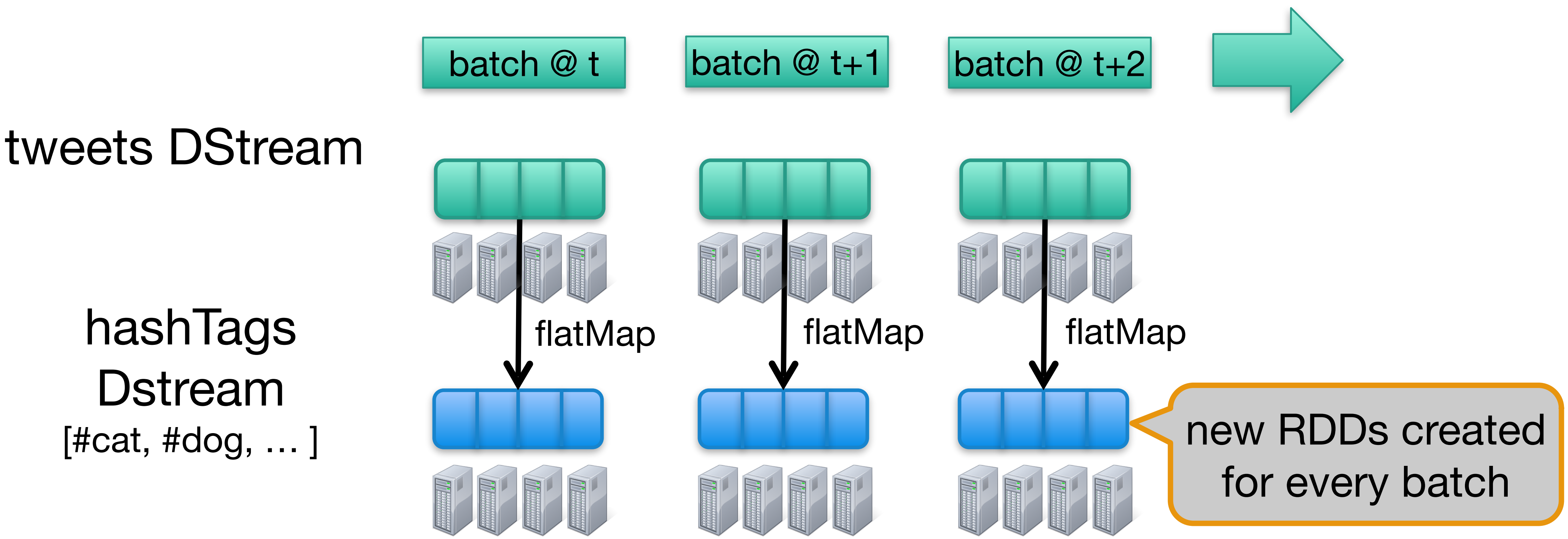
stored in memory as an RDD (immutable, distributed)

Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

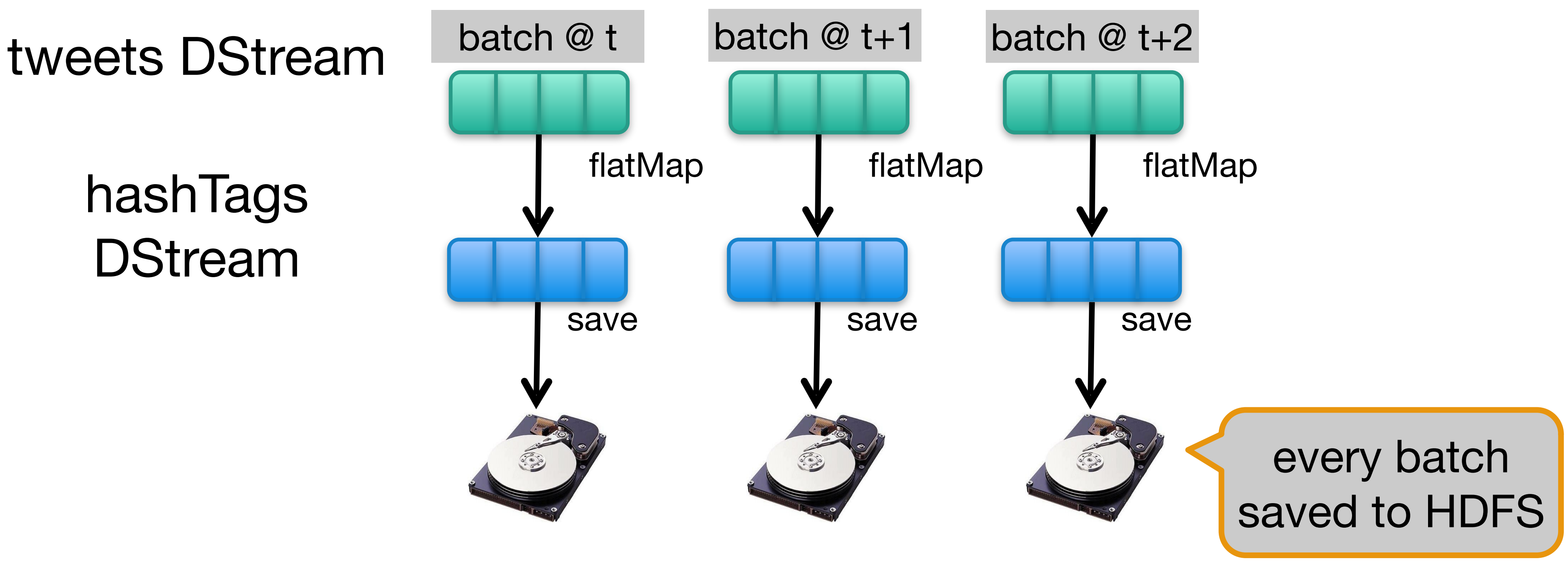
transformation: modify data in one DStream to create another DStream



Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

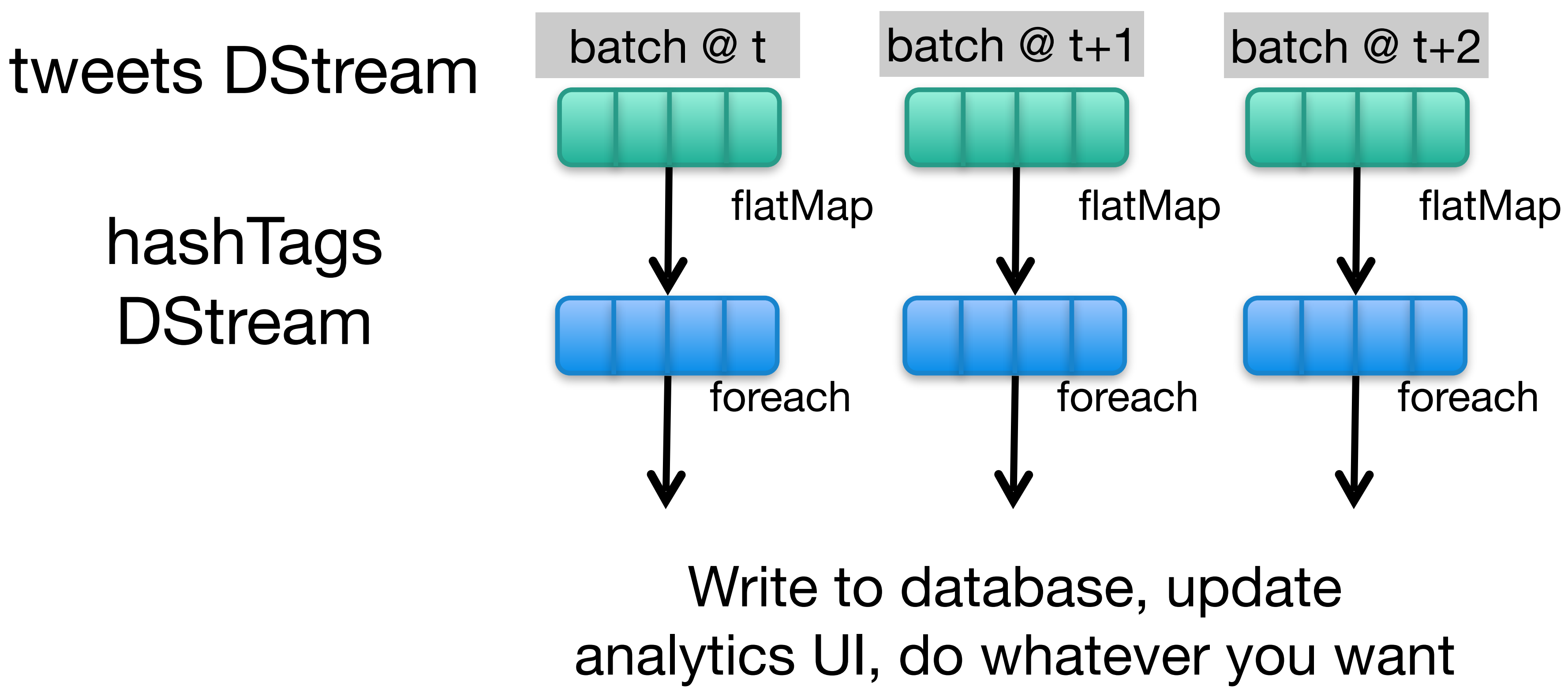
output operation: to push data to external storage



Example – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.foreach(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data



Java Example

Scala

```
val tweets = ssc.twitterStream()

val hashTags = tweets.flatMap (status => getTags(status))

hashTags.saveAsHadoopFiles("hdfs://...")
```

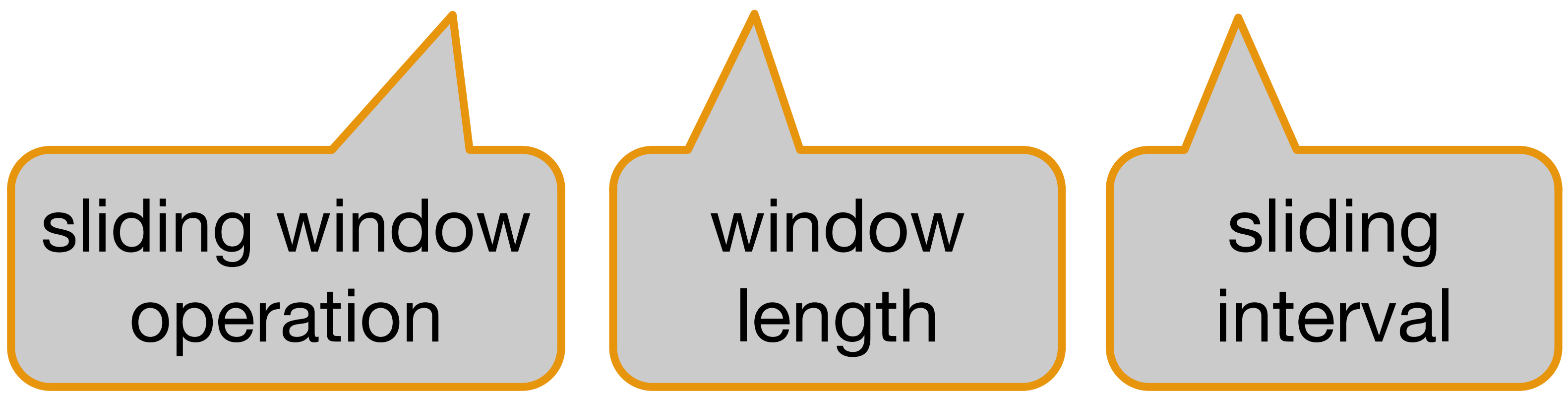
Java

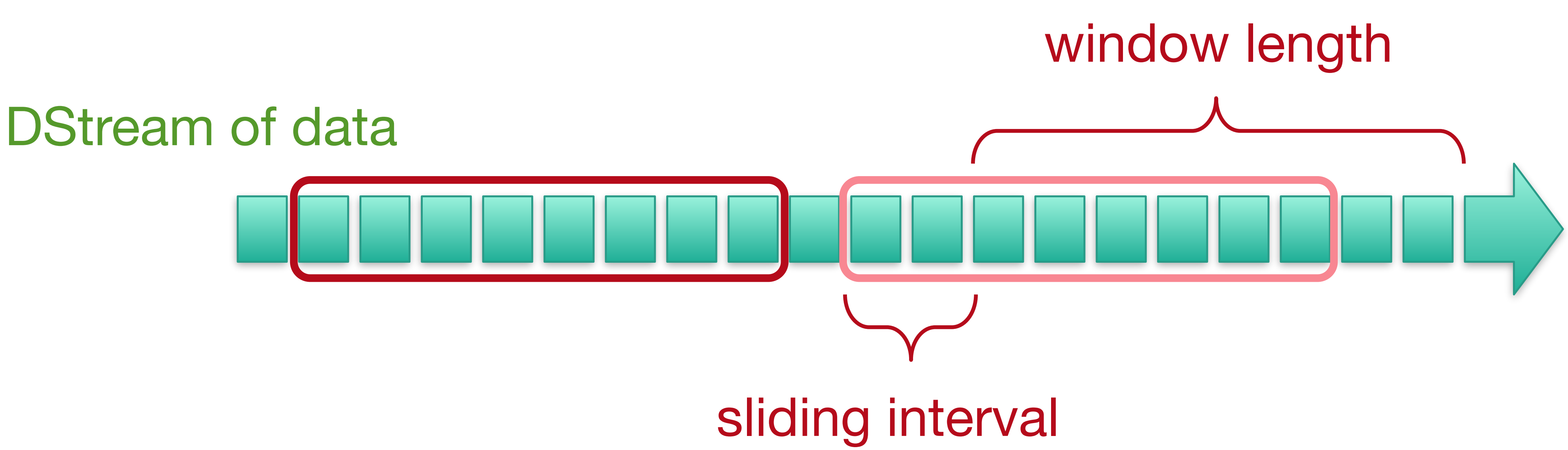
```
JavaDStream<Status> tweets = ssc.twitterStream()

JavaDstream<String> hashTags = tweets.flatMap(new Function<...> {  })

hashTags.saveAsHadoopFiles("hdfs://...")
```

Function object

Window-based Transformations

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
```





Arbitrary Stateful Computations

Specify function to generate new state based on previous state and new data

- Example: Maintain per-user mood as state, and update it with their tweets

```
updateMood(newTweets, lastMood) => newMood
moods = tweets.updateStateByKey(updateMood _)
```

Arbitrary Combinations of Batch and Streaming Computations

Inter-mix RDD and DStream operations!

- Example: Join incoming tweets with a spam HDFS file to filter out bad tweets

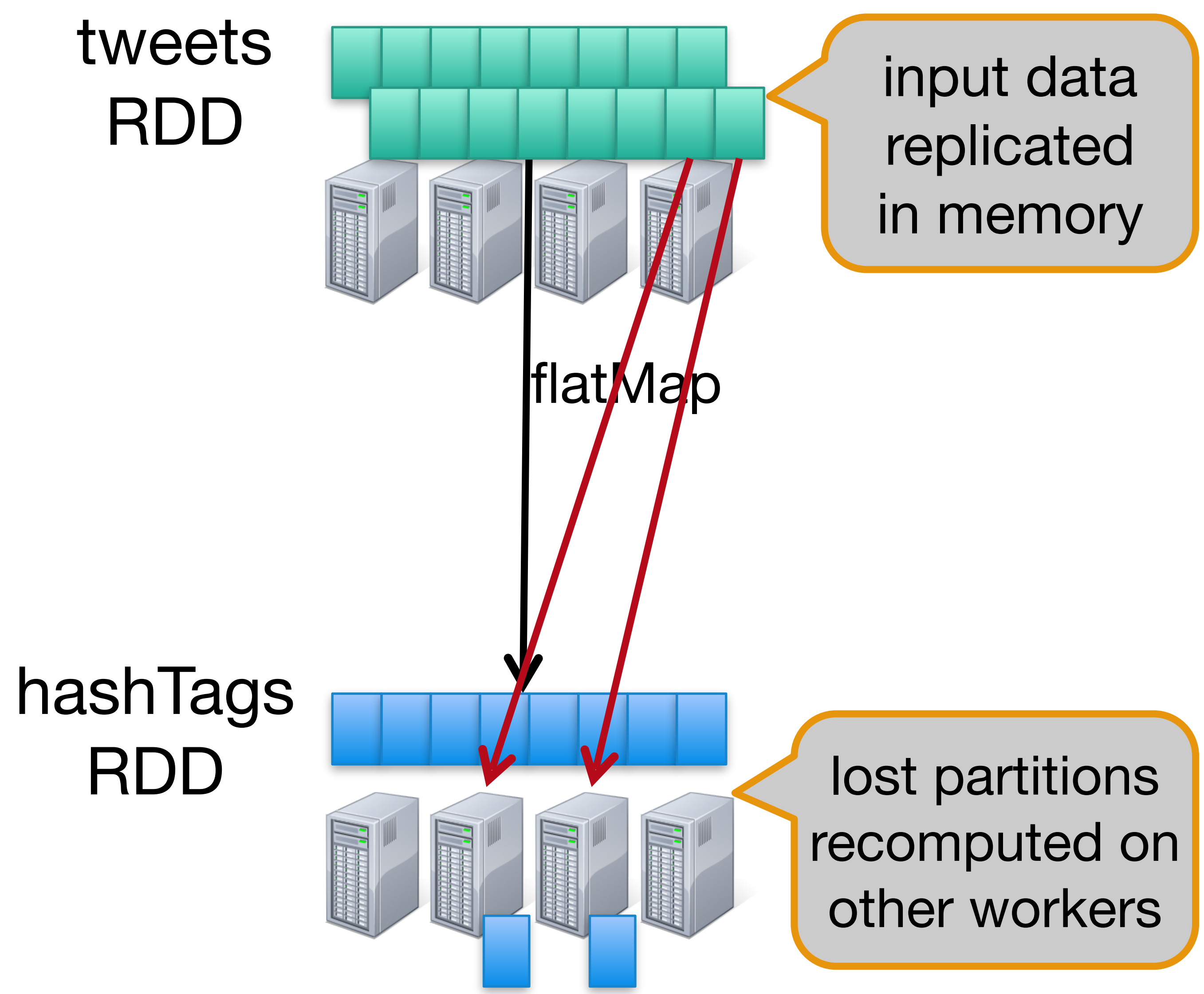
```
 tweets.transform(tweetsRDD => {
tweetsRDD.join(spamHDFSFile).filter(...)
 })
```

DStream Input Sources

- Out of the box Spark Streaming provides
  - Kafka
  - HDFS
  - Flume
  - Akka Actors
  - Raw TCP sockets
- Very easy to write a ***receiver*** for your own data source

Fault-tolerance: Worker

- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data



- All transformed data is fault-tolerant, and exactly-once transformations

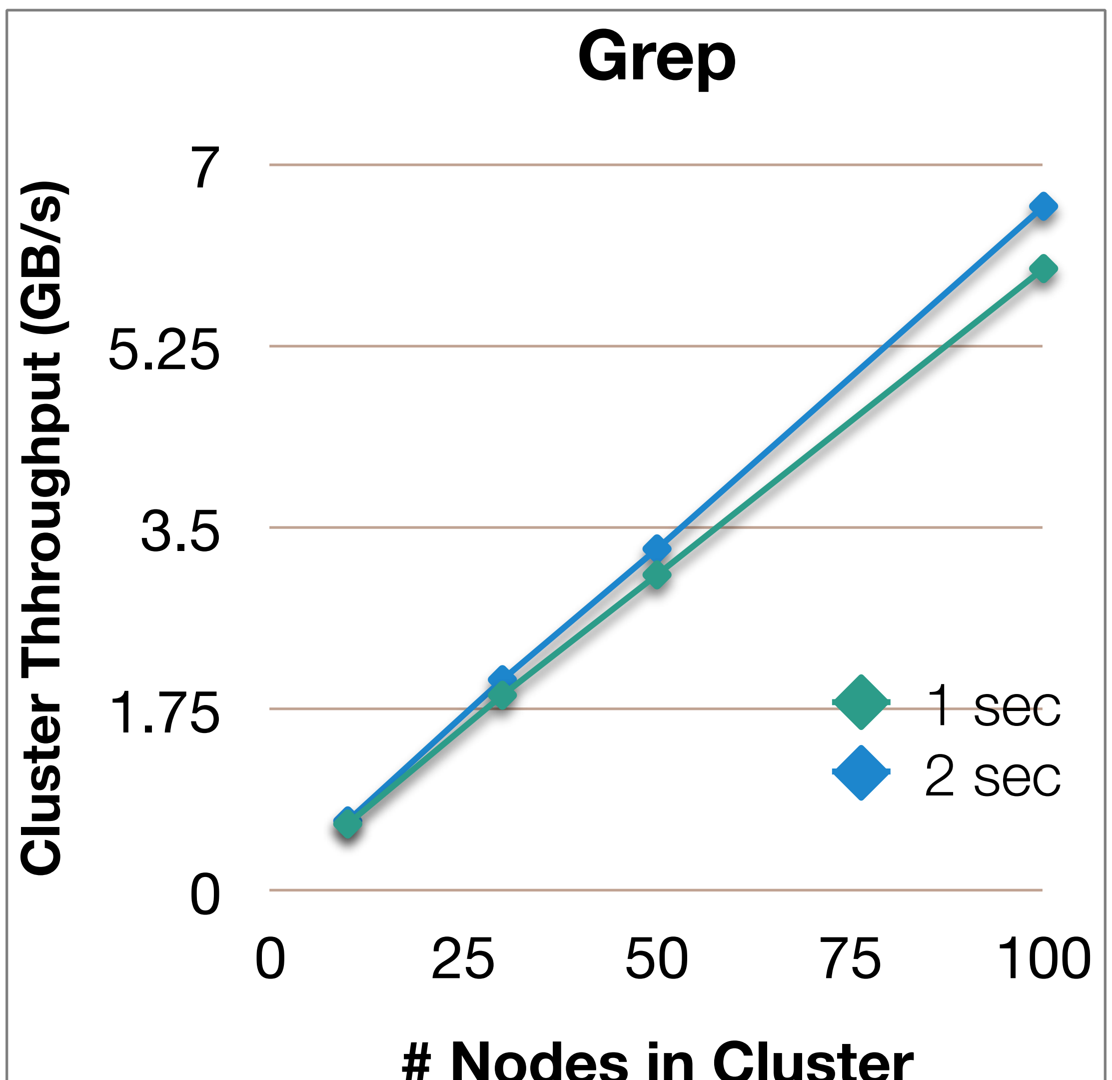
Fault-tolerance: Master

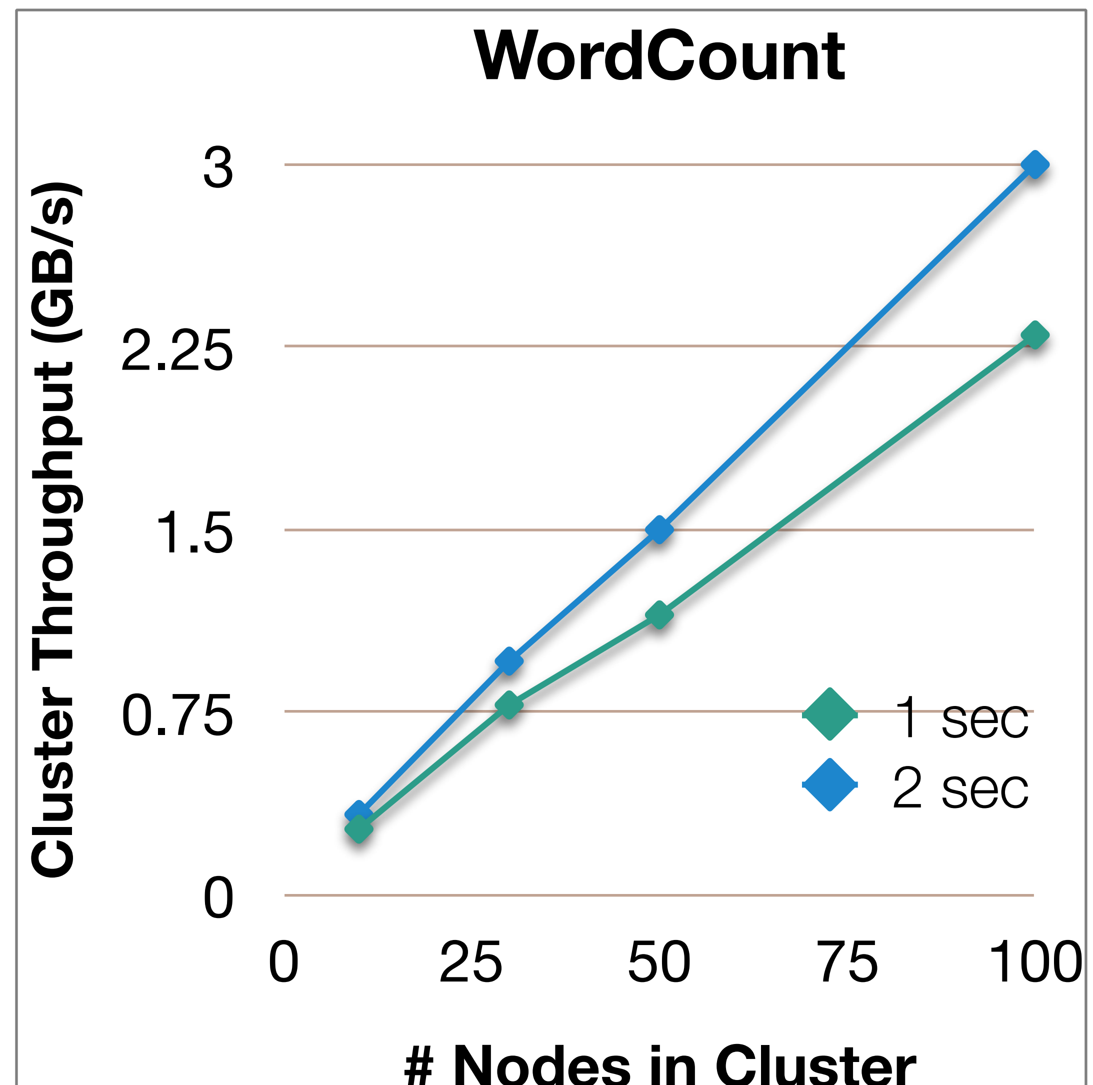
- Master saves the state of the DStreams to a checkpoint file
  - Checkpoint file saved to HDFS periodically
- If master fails, it can be restarted using the checkpoint file
- Automated master fault recovery coming soon

Performance

Can process **6 GB/sec (60M records/sec**) of data on 100 nodes at **sub-second** latency

- Tested with 100 text streams on 100 EC2 instances with 4 cores each

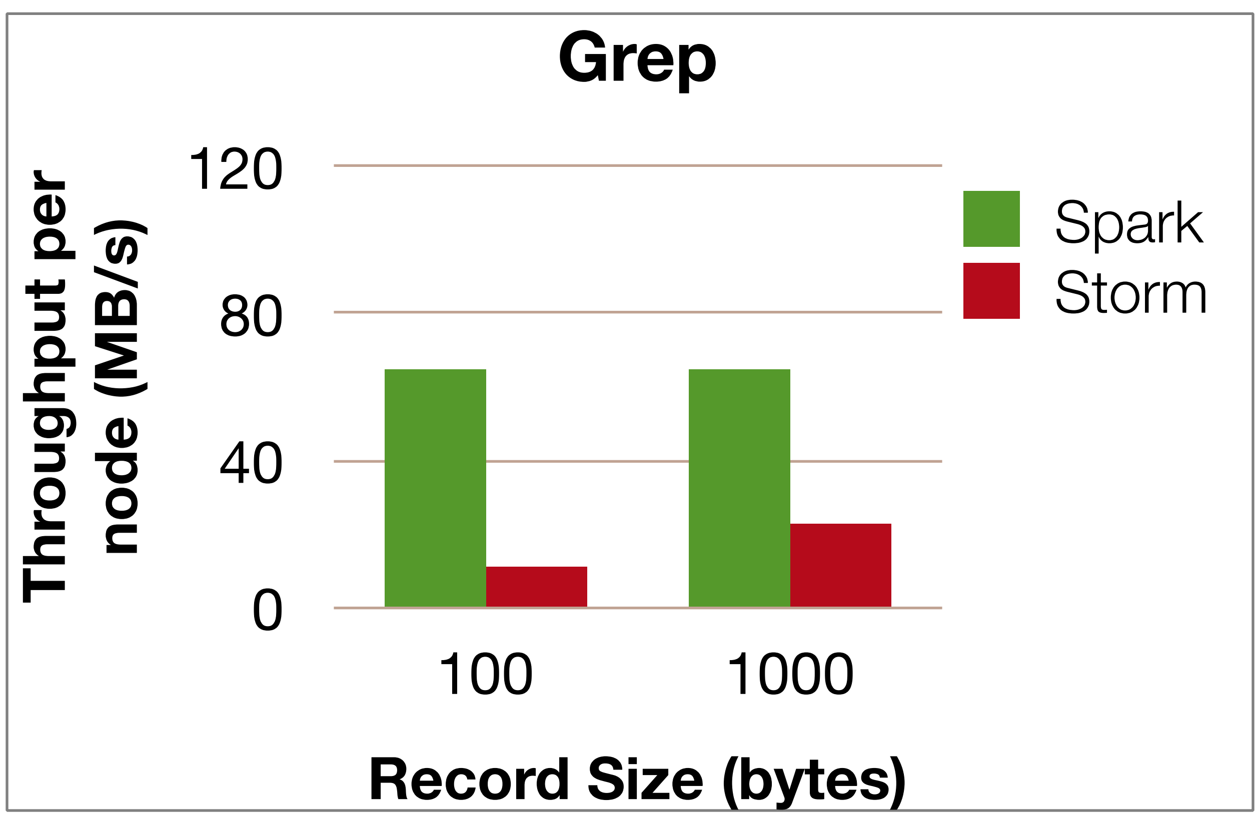


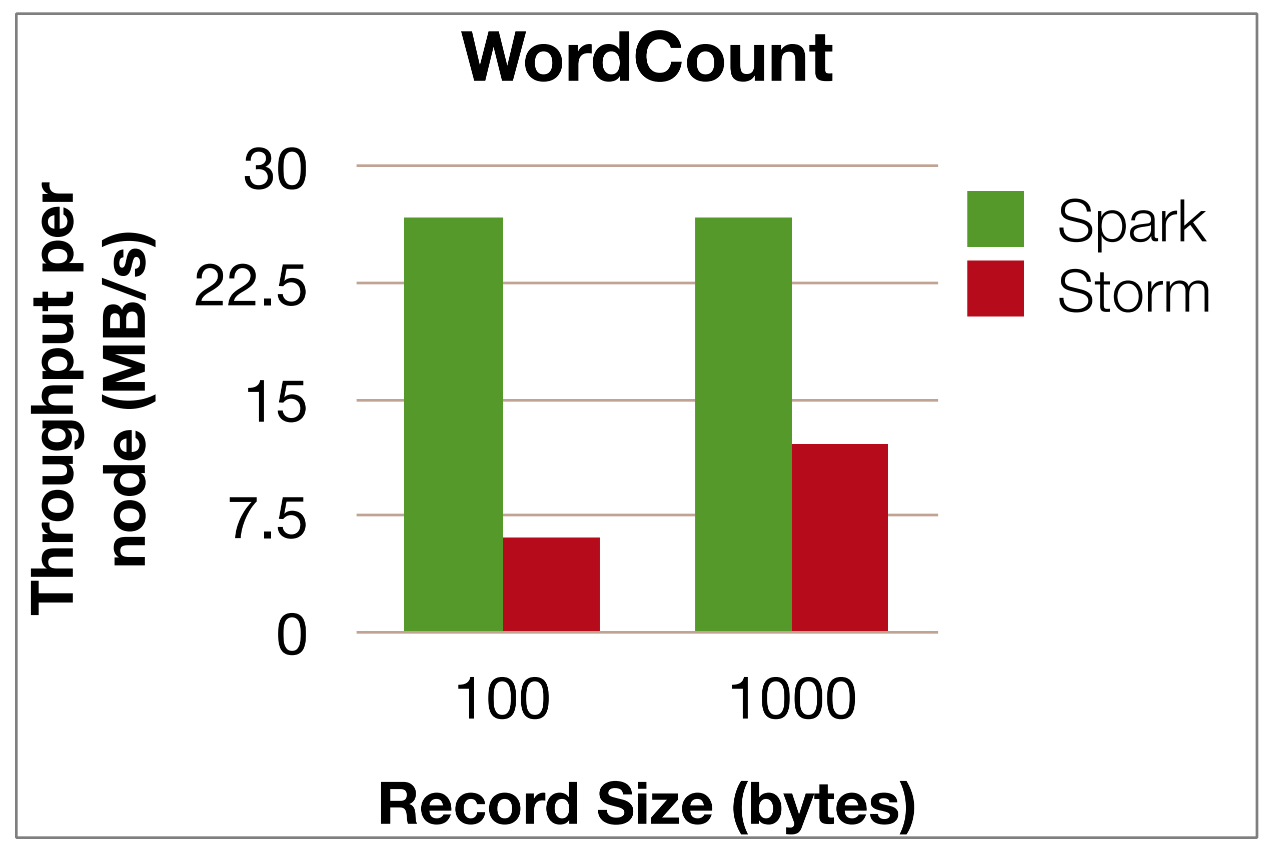


Comparison with Storm

Higher throughput than Storm

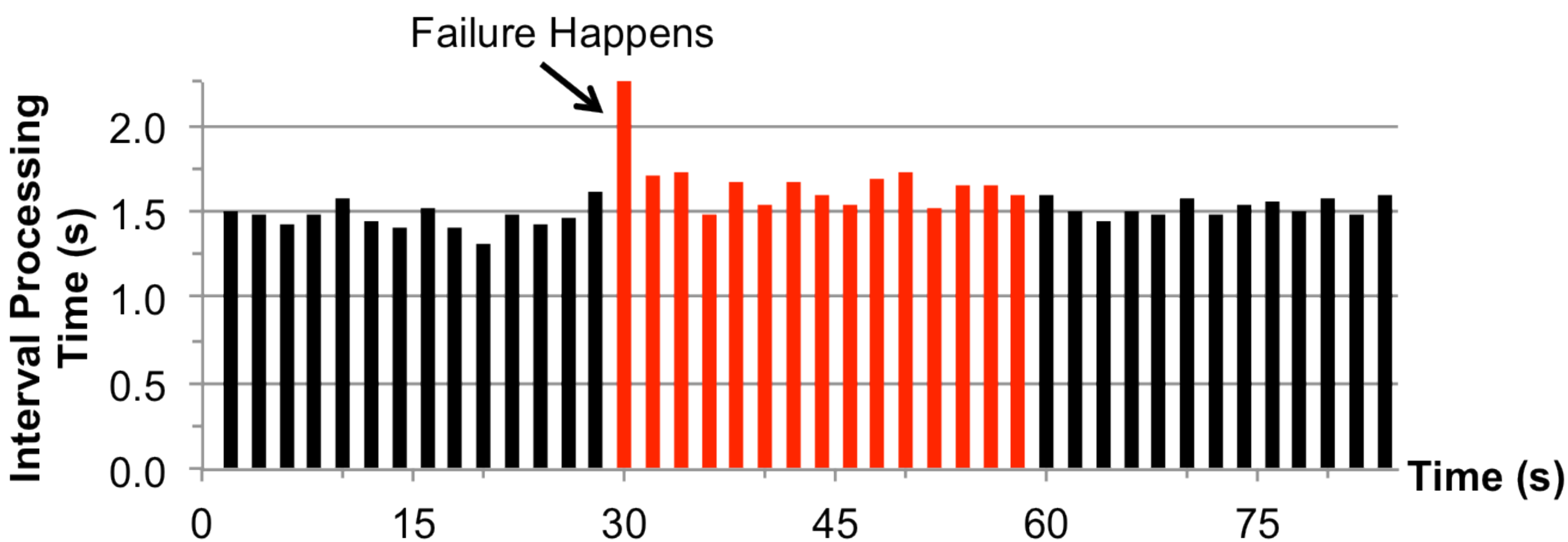
- Spark Streaming: **670k** records/second/node
- Storm: **115k** records/second/node





Fast Fault Recovery

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



Sliding WordCount on 10 nodes with 30s checkpoint interval