BIG DATA

Streaming Spark

K V Subramaniam

Computer Science and Engineering

BIG DATA

Examples of Streaming

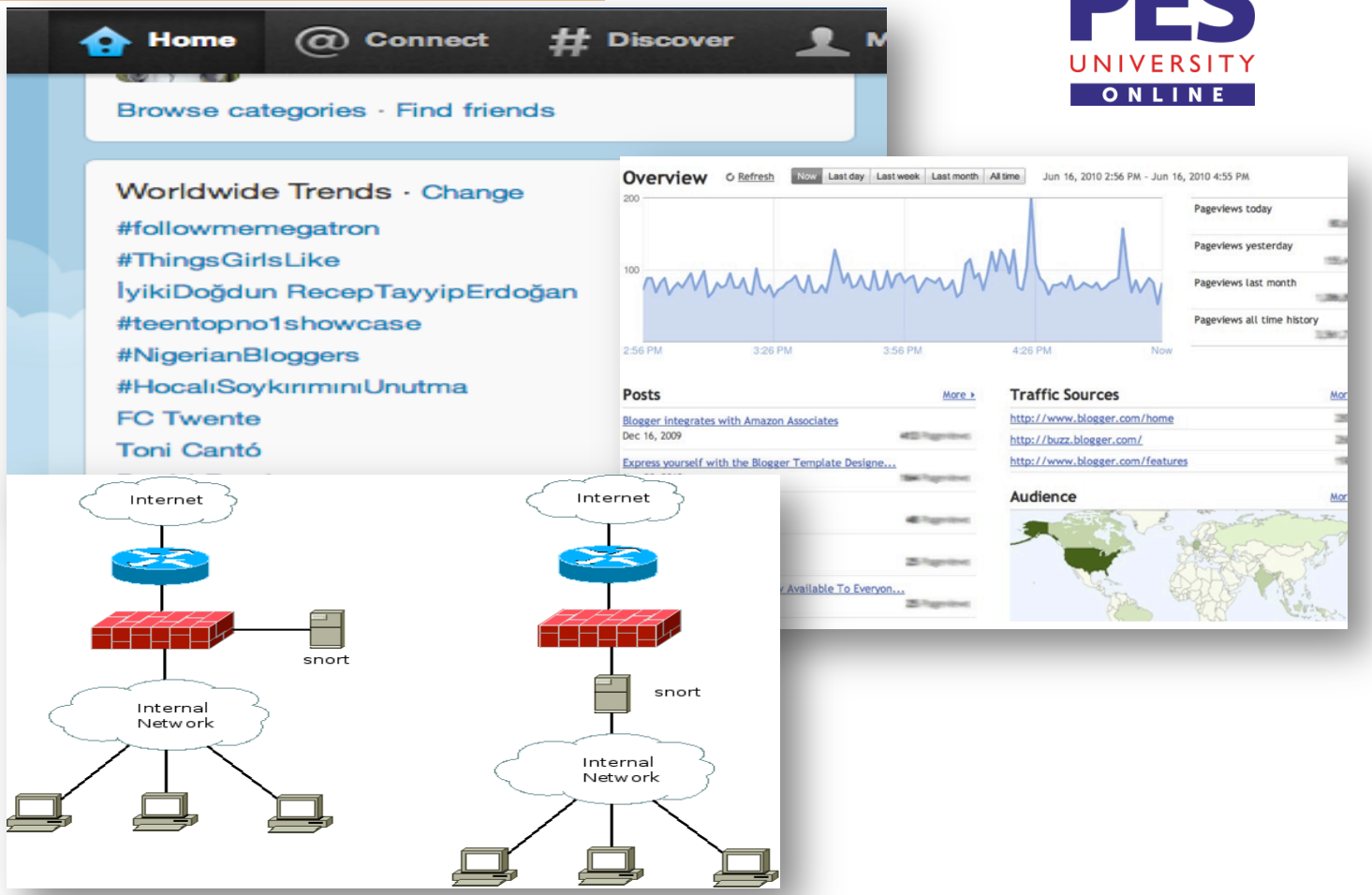
- Sensor data, e.g.,
  - Temperature sensor in the ocean
  - Reports GPS
  - 1 sensor every 150 square miles => 1,000,000 sensors
  - 10 readings/sec => 3.5 TB/day
- Images
  - London: 6 million video cameras
- Internet / Web traffic
  - Google: hundreds of millions of queries/ day
  - Yahoo: billions of clicks/day

BIG DATA

Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Intrusion detection systems
  - Transportation system - Uber
  - etc.
- Require large clusters to handle workloads

- Require latencies of few seconds



BIG DATA

Stream Data Model

- Multiple streams
- Different rates, not synchronized
- Archival store
  - Offline analysis, not real-time
- Working store
  - Disk or memory
  - Summaries
  - Parts of streams
- Queries
  - Standing queries
  - Ad-hoc queries

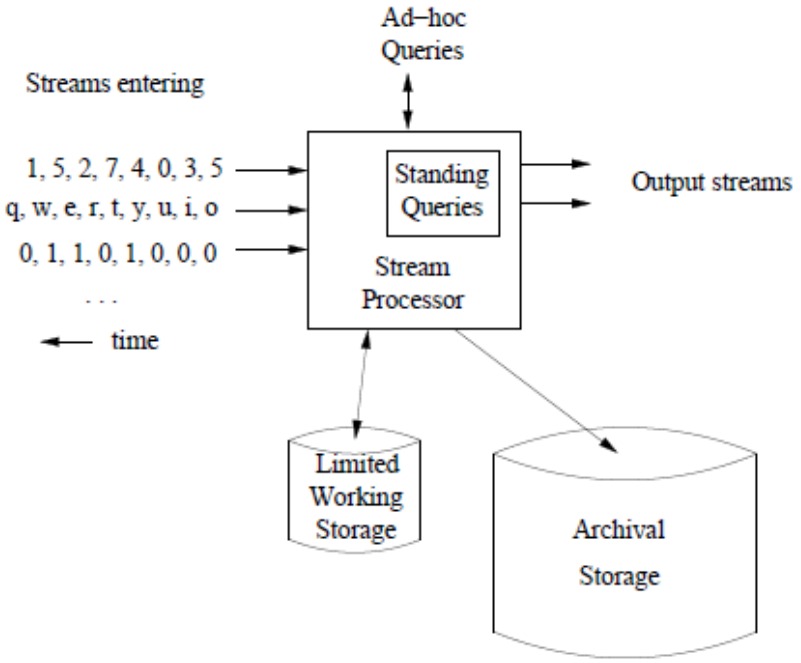


Figure 4.1: A data-stream-management system

Examples of Stream Queries

- **Standing queries**: produce outputs at appropriate time
  - Query is continuously running
  - Constantly reading new data
  - Query execution can be optimized
  - Example: maximum temperature ever recorded
- ***Ad hoc* query**: not predetermined, arbitrary query
  - Need to store stream
  - Approach: store sliding window in SQL DB
  - Do SQL query
  - Example: number of unique users over last 30 days
    - Store logins for last 30 days

BIG DATA

Exercise 1

Consider the queries on the right. Which among them are **STANDING QUERIES** and which are **AD HOC**?

- Alert when temperature > threshold
- Display average of last n temperature readings; n arbitrary
- List of countries from which visits have been received over last year
- Alert if website receives visit from a black-listed country

BIG DATA

Exercise 1 Solution

Consider the queries on the right. Which among them are **STANDING QUERIES** and which are **AD HOC**?

Solution

- Alert when temperature > threshold - standing
- Display average of last *n* temperature readings – *ad hoc*
- List of countries from which visits have been received over last year – *ad hoc*
- Alert if website receives visit from a black-listed country - standing

BIG DATA

Issues in Stream Processing

- Velocity
  - Streams can have high data rate
  - Need to process very fast
- Volume
  - Low data rate, but large number of streams
    - Ocean sensors, pollution sensors
- Need to store in memory
  - May not have huge memory
    - Approximate solutions

Need for a framework …

… for building such complex stream processing applications

But what are the requirements

from such a framework?

BIG DATA

Requirements

- **Scalable** to large clusters
- **Second**-scale latencies
- **Simple** programming model

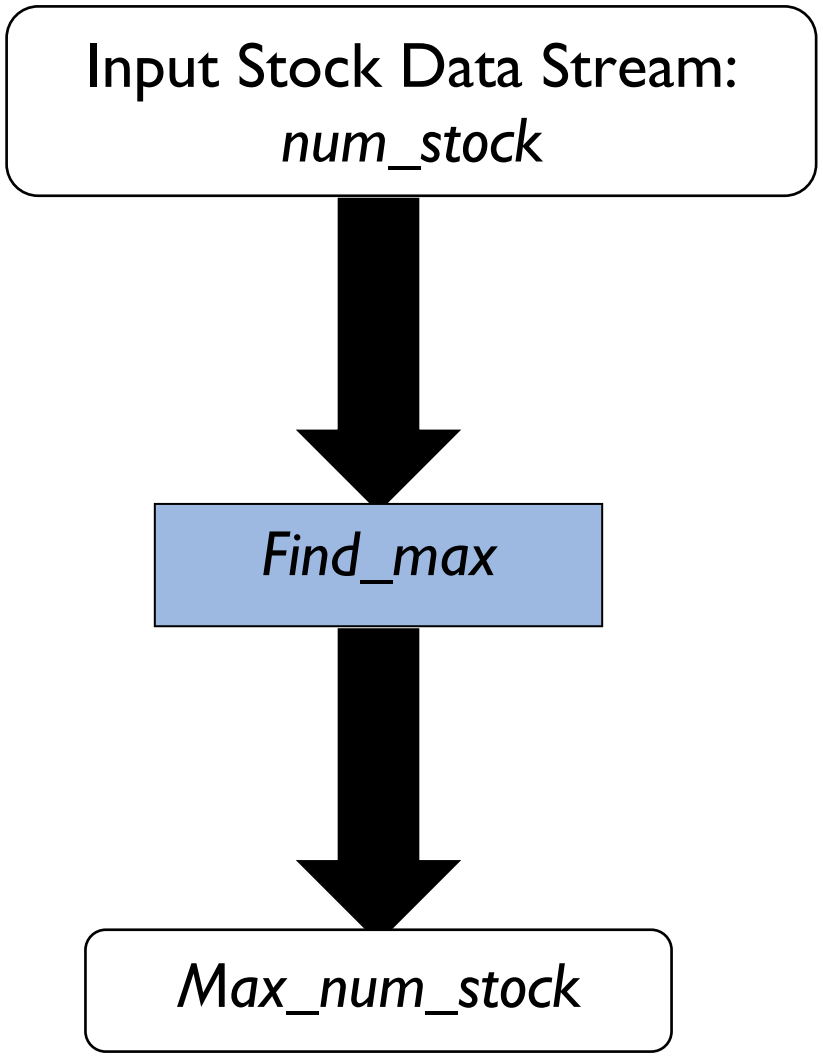
Need for integrated Batch and Stream processing

BIG DATA

Exercise 2

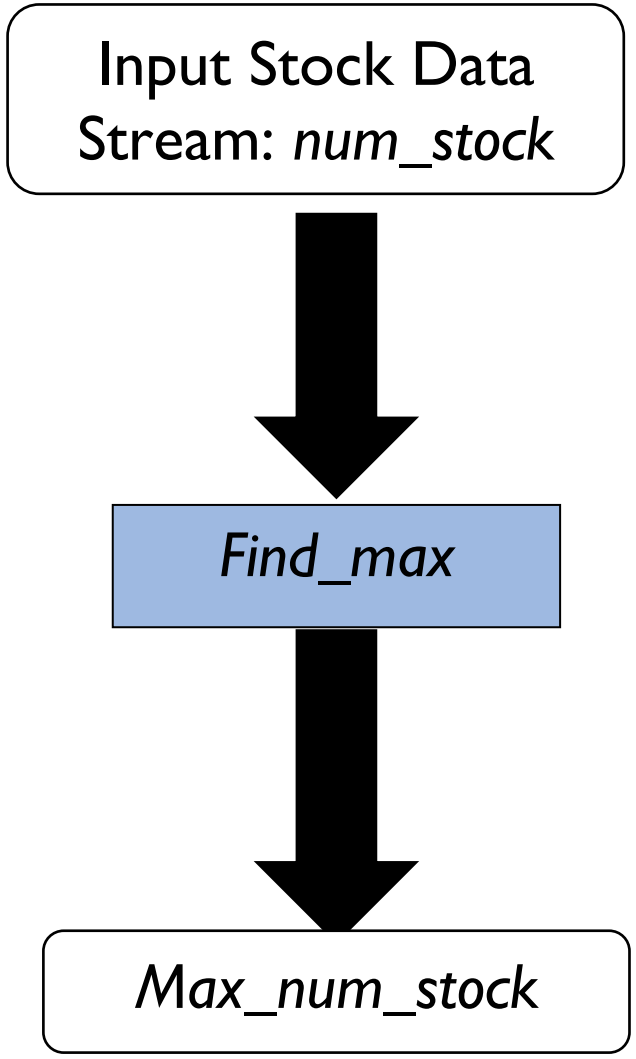
CAN WE USE HADOOP?

- Consider the simple program on the right.
- The input is a stream of records from the stock market.
  - Each time a stock is sold, a new record is created.
  - The record contains a field num_stock which is the number of stocks sold.
- Find_max is a program that updated a variable Max_num_stock which is the maximum of num_stock.



Exercise 2 - Solution CAN WE USE HADOOP?

- Write the pseudo-code for *Find_max*
  - *If num_stock > Max_num_stock*
    Max_num_stock = num_stock
- Can this be implemented in Hadoop?
  - We need to process one record at a time. Hadoop processes a full file in the Map
  - *Max_num_stock* is a global variable
- Does your solution assume that *Find_max* runs on a single node? Is this a reasonable assumption?
  - No, the number of transactions could be more than what a single node can handle



Similar problems can arise in Spark

Case study: Conviva, Inc.

- Real-time monitoring of online video metadata
  HBO, ESPN, ABC, SyFy, …

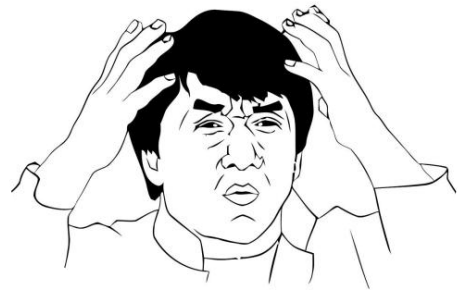
Since we can't use Hadoop

- Two processing stacks
  - Custom-built distributed stream processing system
    - 1000s complex metrics on millions of video sessions
    - Requires many dozens of nodes for processing
  - Hadoop backend for offline analysis
    - Generating daily and monthly reports
    - **Similar computation as the streaming system**

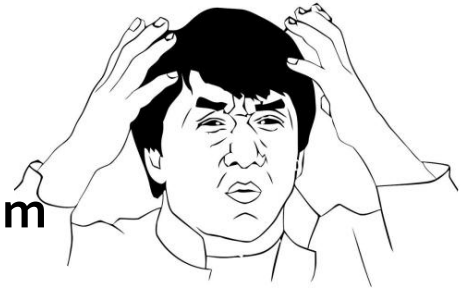
BIG DATA

Case study: XYZ, Inc.

- ***Any company who wants to process live streaming data has this problem***
- **Twice** the effort to implement any new function
- **Twice** the number of bugs to solve
- **Twice** the headache



- Two processing stacks
  - Custom-built distributed stream processing system
    - 1000s complex metrics on millions of video sessions
    - Requires many dozens of nodes for processing
  - Hadoop backend for offline analysis
    - Generating daily and monthly reports
    - **Similar computation as the streaming system**



BIG DATA

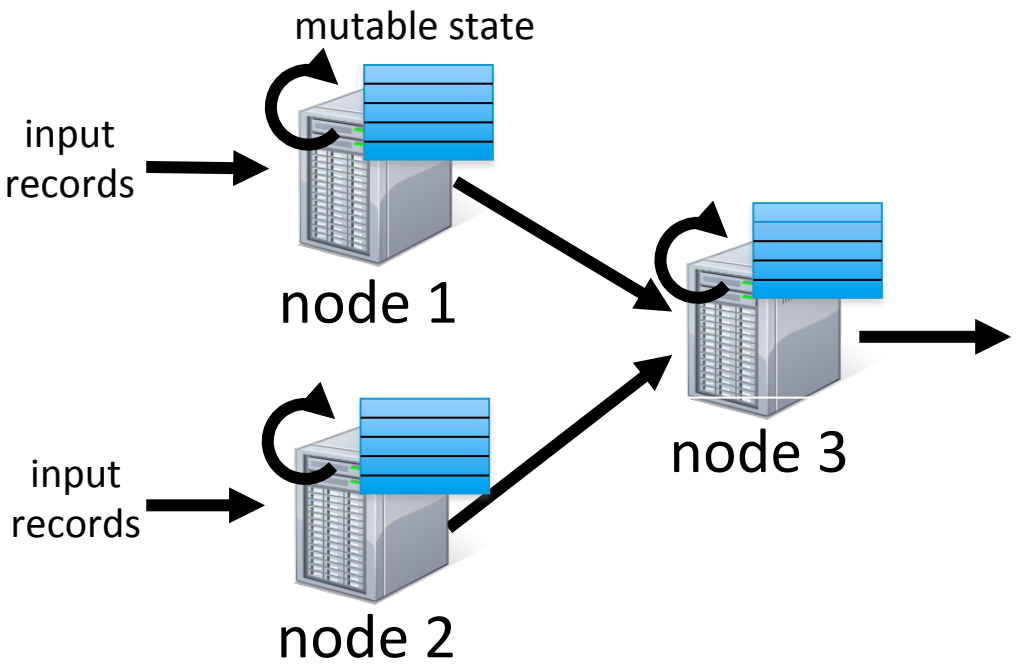
Requirements

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing

State in Stream processing

Stateful Stream Processing

- Traditional streaming systems have a event-driven **record-at-a-time** processing model
  - Each node has mutable state
  - For each record, update state & send new records
- State is lost if node dies!
- Making stateful stream processing be fault-tolerant is challenging

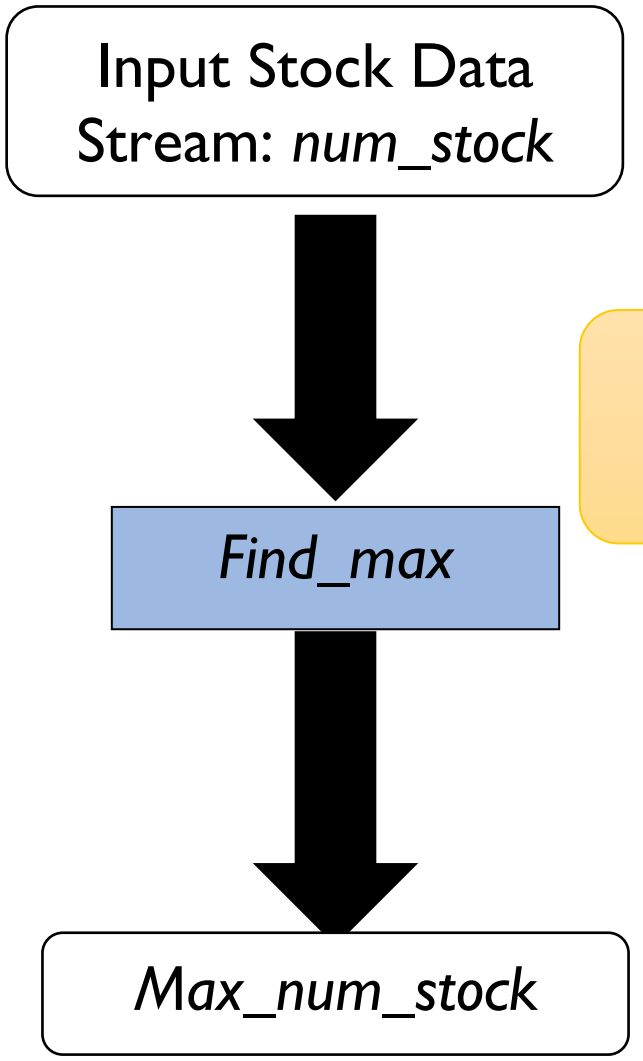


BIG DATA

Exercise 2 - Solution

CAN WE USE HADOOP?

- Write the pseudo-code for *Find_max*
  - *If num_stock > Max_num_stock*
    Max_num_stock = num_stock
- Can this be implemented in Hadoop?
  - We need to process one record at a time. Hadoop processes a full file in the Map
  - *Max_num_stock* is a global variable
- Does your solution assume that *Find_max* runs on a single node? Is this a reasonable assumption?
  - No, the number of transactions could be more than what a single node can handle



Similar problems can arise in Spark

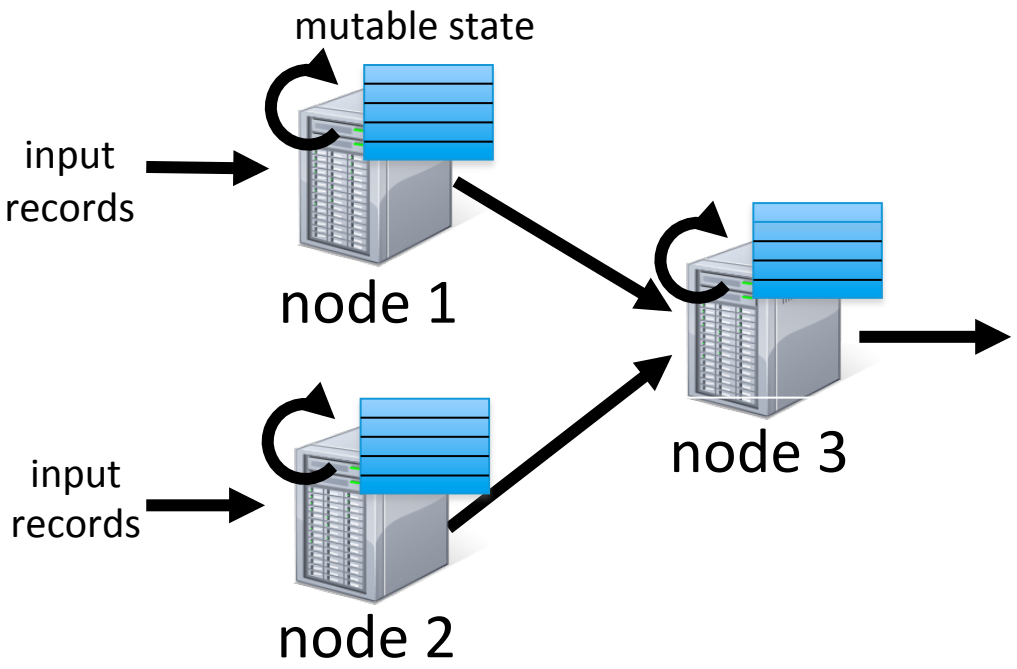
Exercise 3: Stateful Stream Processing

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

In the stock market example, where is the state?

- State is lost if node dies!

- Making stateful stream processing be fault-tolerant is challenging

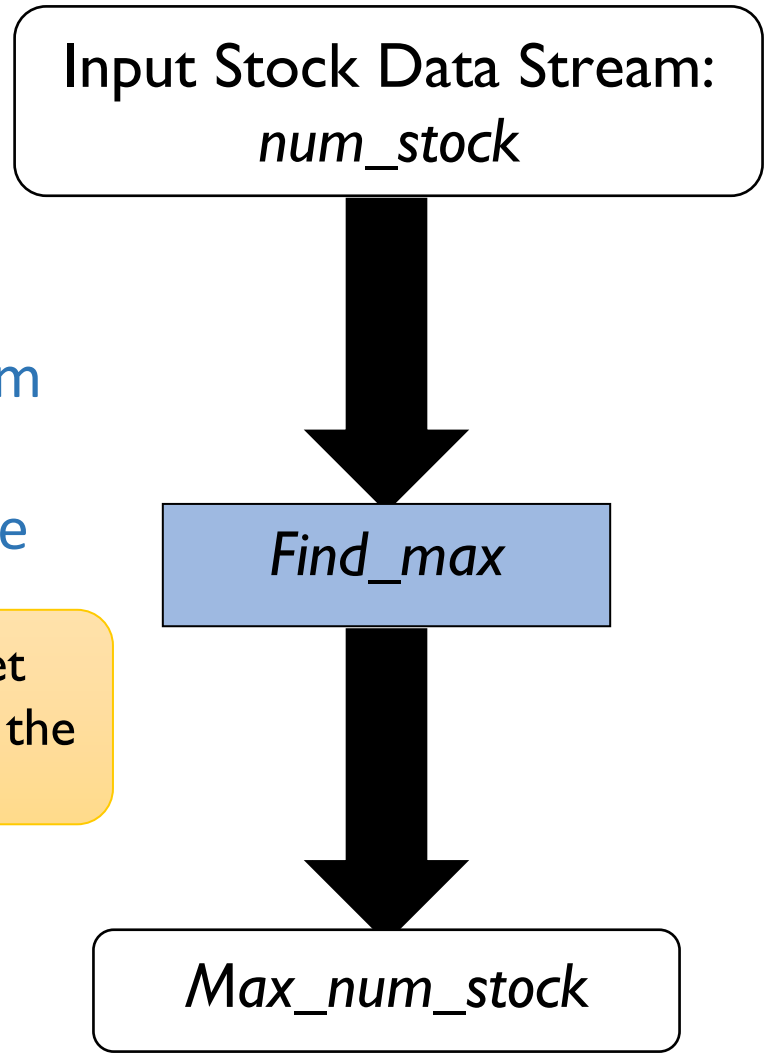


BIG DATA

Exercise 3: (solution): Stateful Stream Processing

- Write the pseudo-code for Find_max
  - If num_stock > Max_num_stock
    - Max_num_stock = num_stock
- Max_num_stock is a global state
  - Its value depends upon the entire stream sequence
  - The first num_stock could have been the largest

In the stock market example, where is the state?



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!

- Trident – Use transactions to update state
  - Processes each record exactly once
  - Per state transaction updates slow

BIG DATA

Requirements

- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing
- **Efficient fault-tolerance** in stateful computations

Spark Streaming

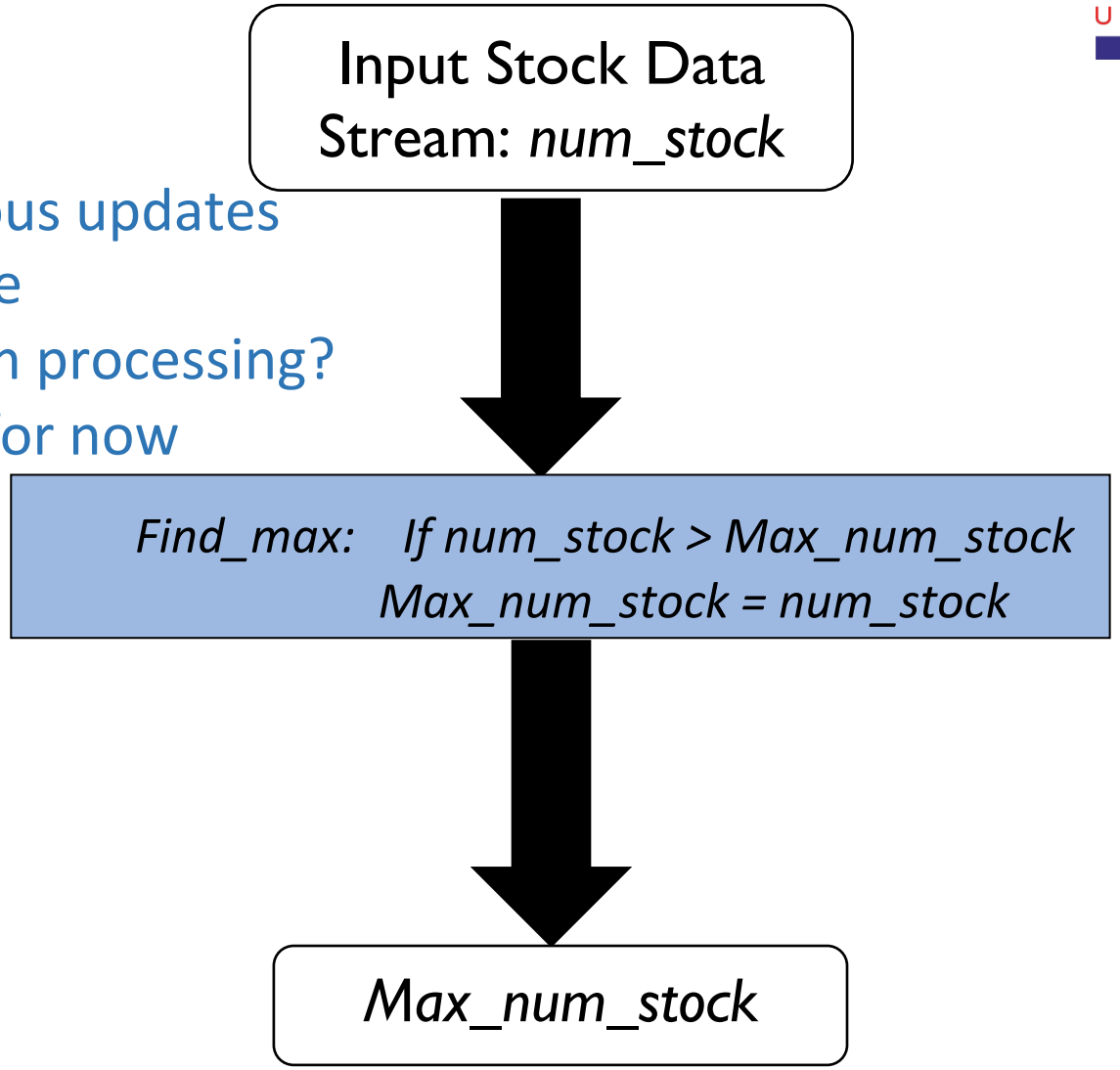
What is Spark Streaming?

- Framework for large scale stream processing
  - Scales to 100s of nodes
  - Can achieve second scale latencies
  - Integrates with Spark's batch and interactive processing
  - Provides a simple batch-like API for implementing complex algorithm
  - Can absorb live data streams from Kafka, Flume, ZeroMQ, etc.

Exercise 4:

Can we modify Hadoop?

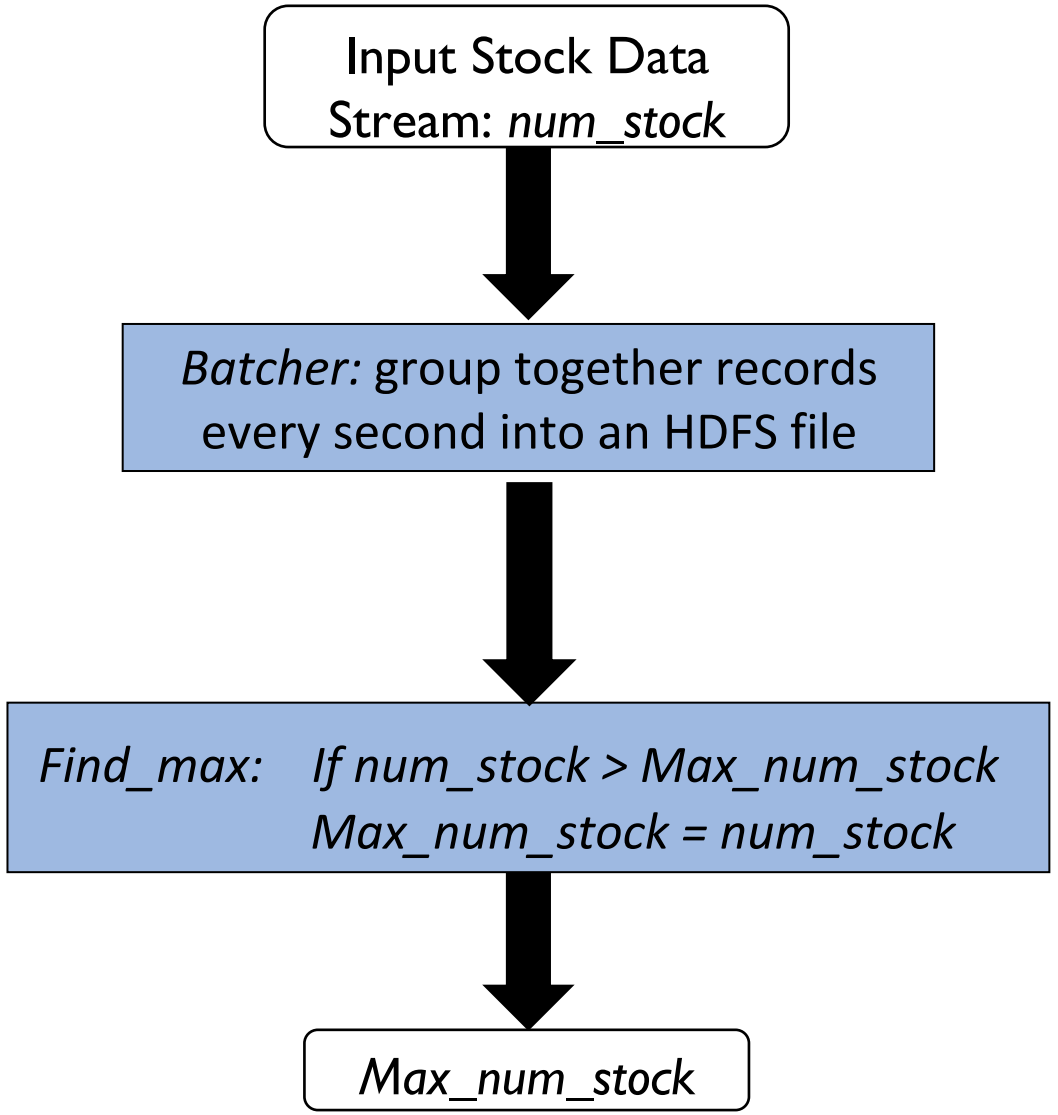
- Suppose we don't want instantaneous updates
- Say 1 second updates are acceptable
- Can we modify Hadoop to do stream processing?
- Ignore the global variable problem for now



BIG DATA

Exercise 4: Solution

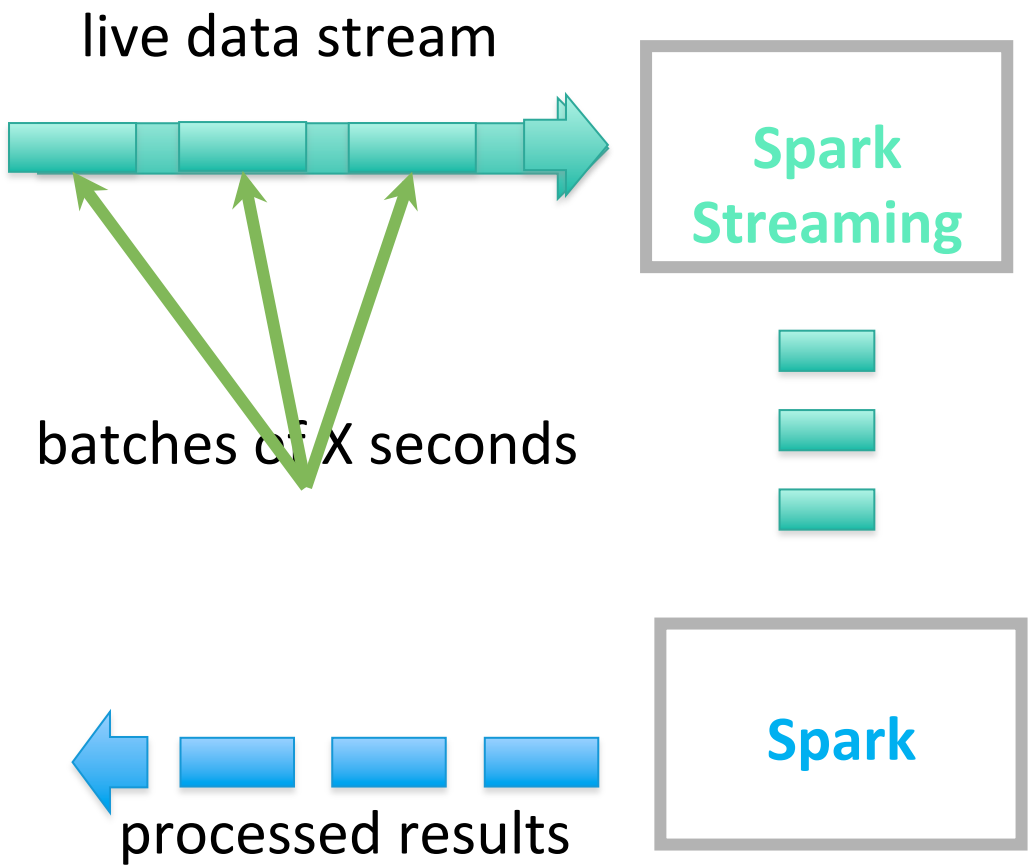
- We can *batch* input records into an HDFS file
- We can batch together the inputrecords every second
- This file can be processed using MapReduce
- We can produce an update every second
- We have ignored the global variable problem for now



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

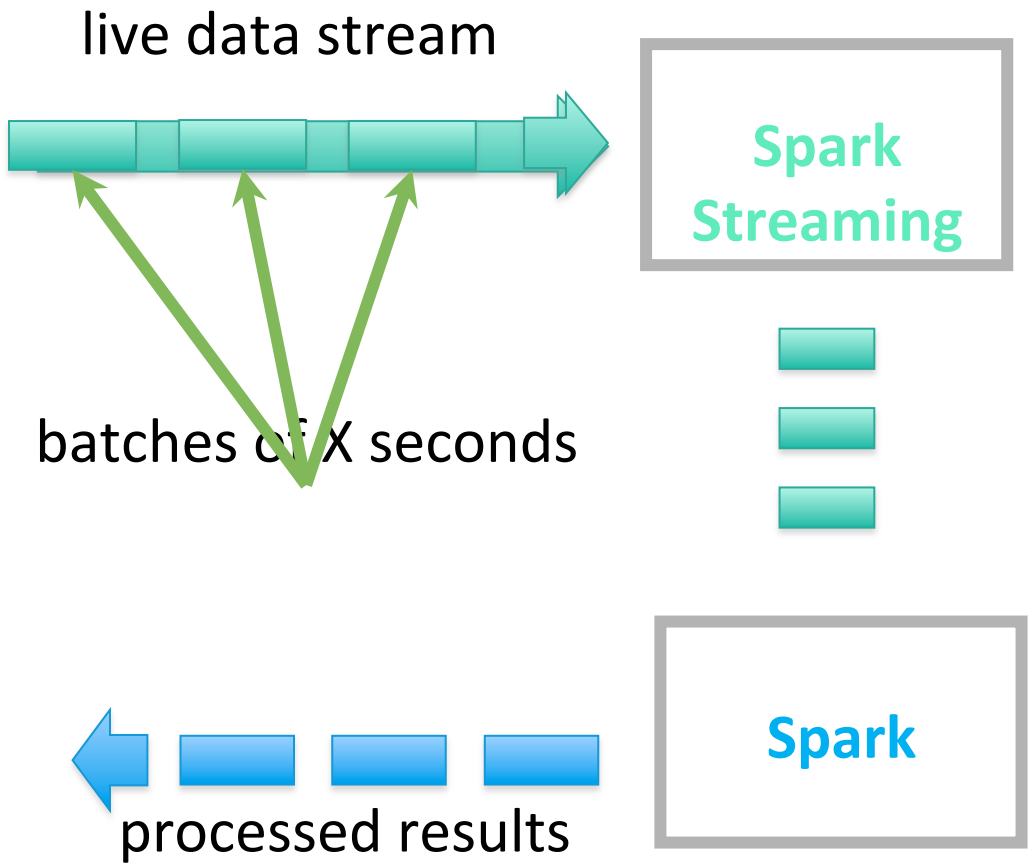


BIG DATA

Discretized Stream Processing

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

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



Streaming Spark - DStreams

Remember!!!!

- **In Spark (not Streaming Spark)**
  - Every variable is an **RDD**
  - There are two kinds of **RDDs**
  - ***Pair RDDs*** are RDDs that consist of key-value pairs
  - **Special operations**, such as *reduceByKey* are defined on *pair RDDs*

BIG DATA

Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
```

DStream: a sequence of RDD representing a stream of data

twitterstream returns a variable of type *DSTREAM*

Twitter Streaming API

tweets DStream

stored in memory as an RDD (immutable, distributed)

BIG DATA

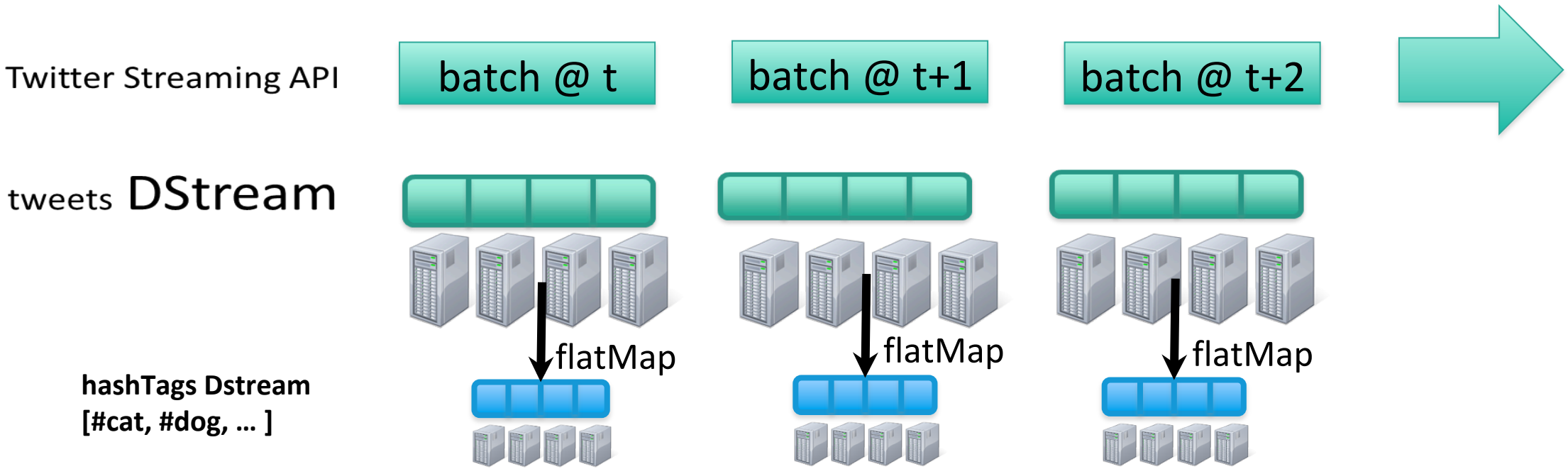
Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
```

new DStream

transformation: modify data in one Dstream to create another DStream

getTags is a function. A *DStream* is a sequence of RDDs. The function is applied to each RDD. The result is another *DStream*



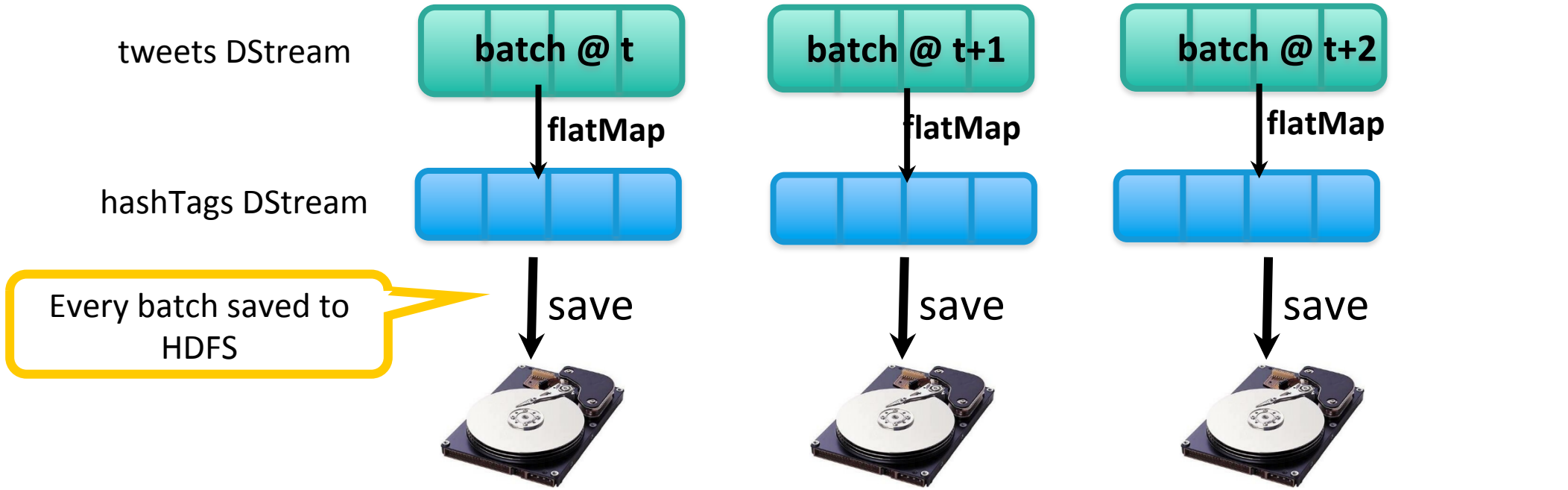
BIG DATA

Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))

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

output operation: to push data to external storage



Java Example

Scala

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

Java

```
JavaDStream<Status> tweets = ssc.twitterStream(<Twitter username>,
<Twitter password>)
JavaDstream<String> hashTags = tweets.flatMap(new Function<...> {  })
hashTags.saveAsHadoopFiles("hdfs://...")
```

Function object to define the transformation

Spark Streaming – Execution of jobs

BIG DATA

Streaming Spark Flow

Dstreams and Receivers

- Twitter, HDFS, Kafka, Flume

Transformations

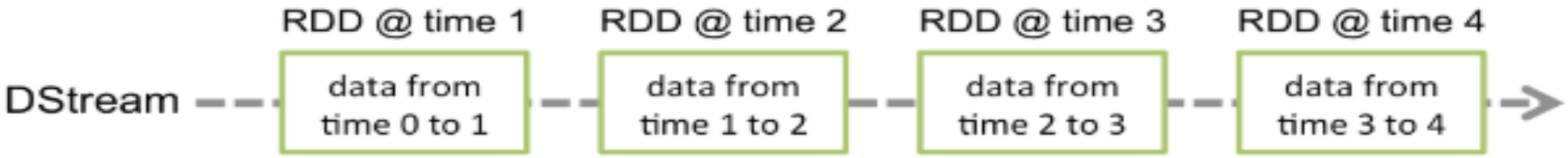
- Standard RDD operations – map, countByValue, reduce, join, …
- Stateful operations – window, countByValueAndWindow, …

Output Operations on Dstreams

- saveAsHadoopFiles – saves to HDFS
- foreach – do anything with each batch of results

Dstreams and Receivers

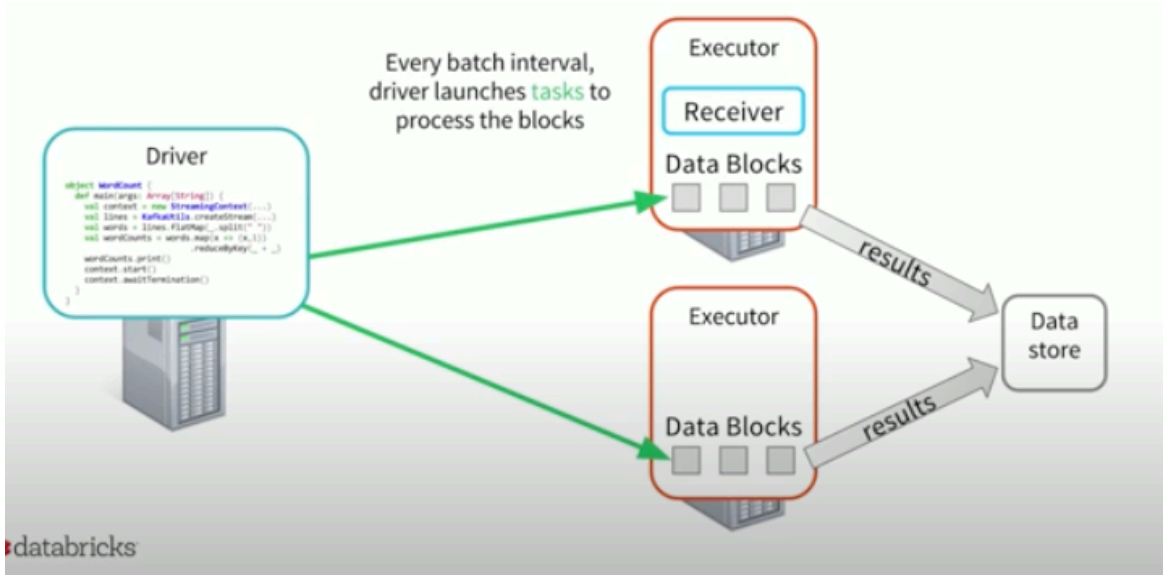
- **Streaming Spark processes data in batches**
  - Together termed the Dstream
- **Every Dstream is associated with a Receiver**
  - Receivers read data from a source and store into Spark Memory for processing
  - Types of sources
    - Basic – file systems and sockets
    - Advanced – Kafka, flume
- **Relationship between Dstream and RDD**



BIG DATA

Dstreams and Receivers

- Streaming Spark processes job
- Starts a Receiver on an executor as a long running job
- Driver starts tasks to process blocks on every interval



Source: https://www.youtube.com/watch?v=NHfggxItokg

Spark Streaming – stateless and stateful processing

BIG DATA

Transformations in Spark

- Stateless transformations
- Stateful Transformations

BIG DATA

Stateless transformations in Spark

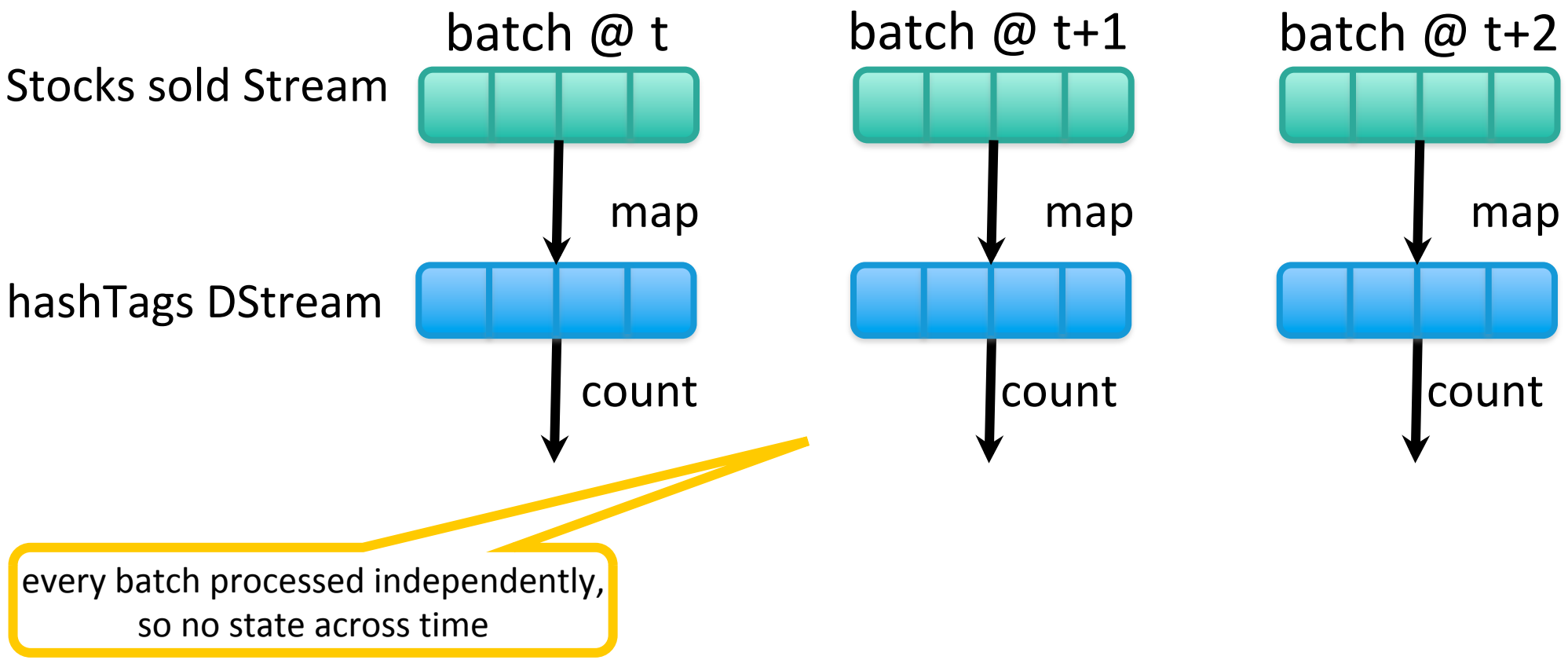
- Transformation is applied to every batch of data independently.
- No information is carried forward between one batch and the next batch
- **Examples**
  - Map()
  - FlatMap()
  - Filter()
  - Repartition()
  - reduceByKey()
  - groupByKey()

Class Exercise : Stateless stream processing (10 mins)

- Consider a Dstream on stock quotes generated similar to earlier that contains
  - A sequence of tuples that contain <company name, stock sold>
  - Need to find total shares sold per company in the last 1 minute
- Show Streaming spark design for the same.

BIG DATA

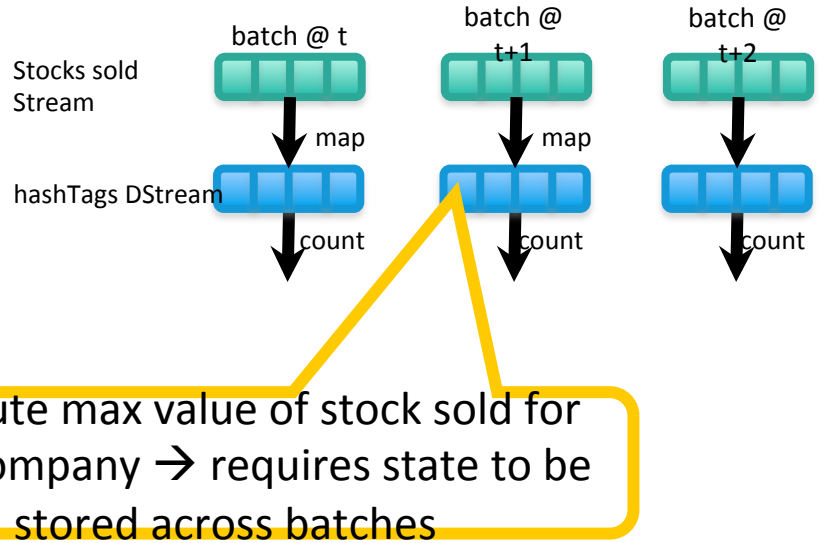
Solution – count stock in every window



BIG DATA

Stateful transformations

- Sometimes we need to keep some state across different batches of data
- For example, what's the max amount of stock sold across the whole day for a company?
  - In this case we need to store max value for each company
- How do we store state across batches?
- First we ensure that data is in pairRDDs
  - Key, value format
  - Helps to ensure that we have a state per key



Stateful processing

BIG DATA

Stateful transformations

- Spark provides two options
  - Window operator – when we want a state to be maintained across short periods of time

  - Session based – where state is maintained for the entire session

BIG DATA

Example 3 – Count the hashtags over last 10 mins

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

sliding window operation

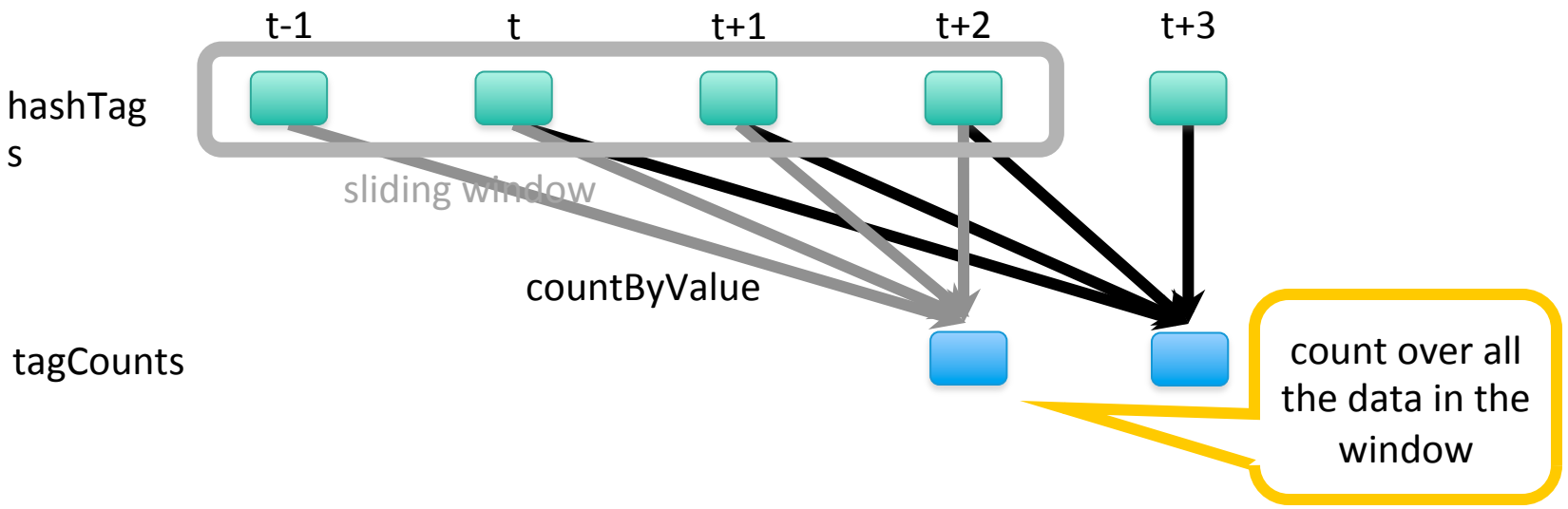
window length

sliding interval

BIG DATA

Example 3 – Counting the hashtags over last 10 mins

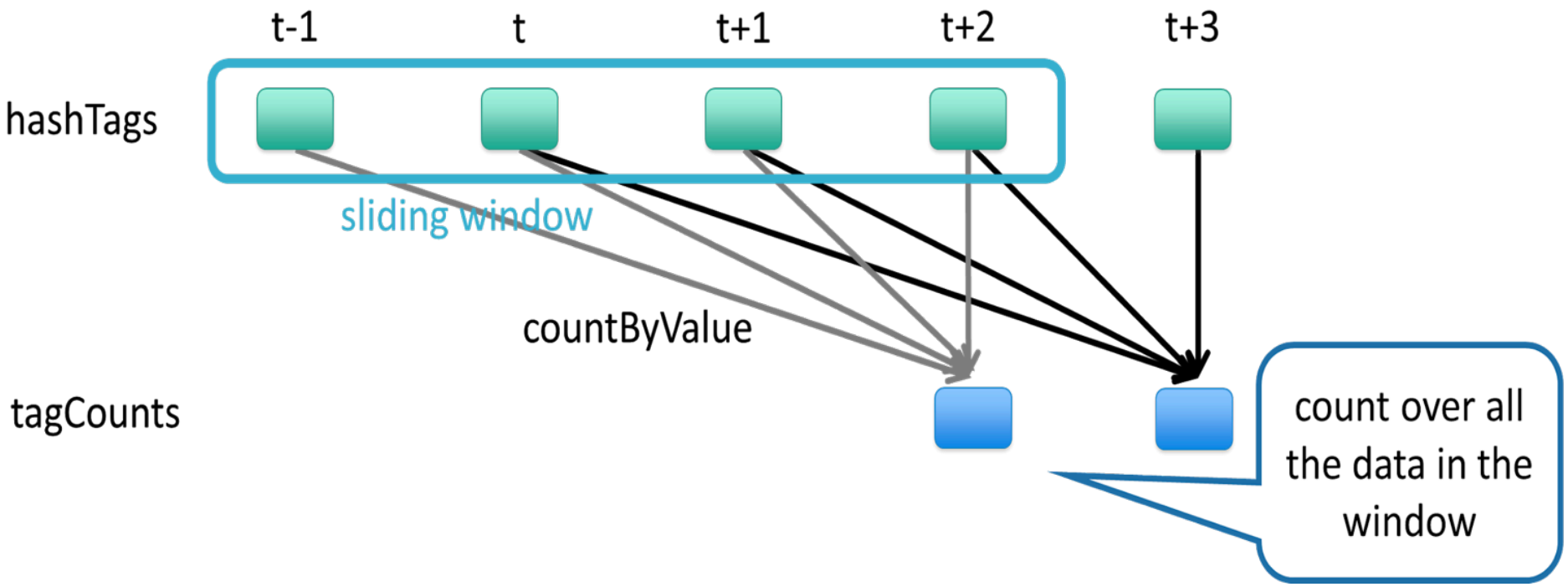
```
val tagCounts = hashTags.window(Minutes(10),Seconds(1)).countByValue()
```



BIG DATA

Class Exercise: (5 mins)

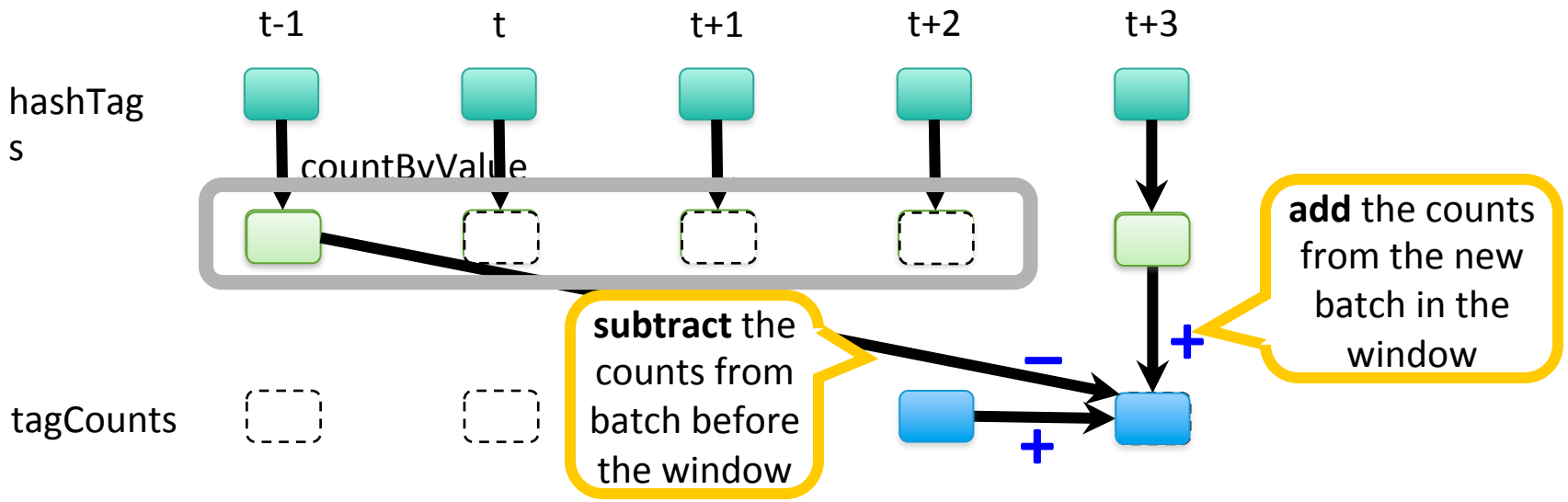
```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



How can we make this count operation more efficient?

Smart window-based countByValue

```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```



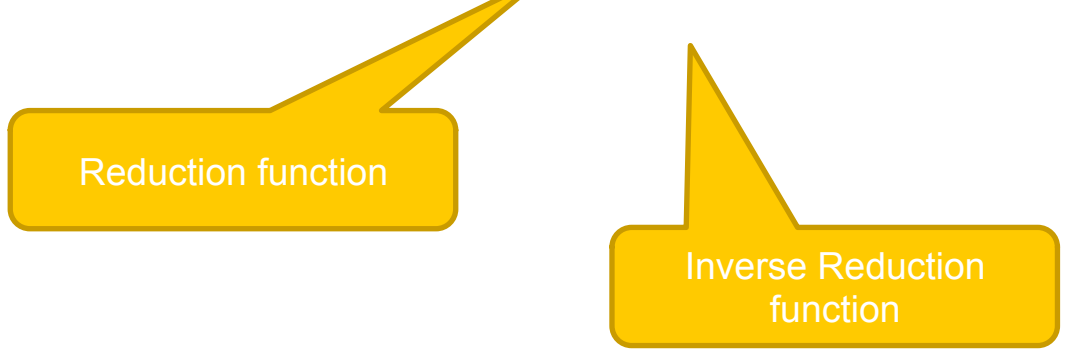
BIG DATA

Smart window-based reduce

- Technique to incrementally compute count generalizes to many reduce operations
  - Need a function to “inverse reduce” (“subtract” for counting)

- Could have implemented counting as:

```
hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), …)
```



BIG DATA

Session based state

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

```
tweets.updateStateByKey(tweet => updateMood(tweet))
```

- `updateStateByKey` uses the current mood and the mood in the tweet to update the user's mood

BIG DATA

Exercise 4 – Maintaining State (10 minutes)

- Consider the code to the right
  - What has to be the structure of the RDD *tweets*?
    - Hint – note that *updateStateByKey* needs a key
  - What does the function *updateMood* do?
    - Hint – note that it should update per-user mood

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

```
tweets.updateStateByKey(tweet =>
updateMood(tweet))
```

BIG DATA

Exercise 4 – Maintaining State: solution

- What has to be the structure of the RDD *tweets*?
  - Must consist of key value pairs with user as key and mood as value
    - Dinkar Happy
    - KVS VeryHappy
    - …
- What does the function *updateMood* do?
  - Compute the new mood based upon the old mood and tweet
- Suppose user Jack (key) tweets “Eating icecream” (value)

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

```
tweets.updateStateByKey(tweet =>
updateMood(tweet))
```

Exercise 4 – Maintaining State: solution

- Consider the code to the right
  - What has to be the structure of the RDD *tweets*?
    - Must consist of key value pairs with user as key and mood as value
      - Dinkar Happy
      - KVS VeryHappy
      - …
  - Suppose user Dinkar (key) tweets "Eating icecream" (value)
    - *updateStateByKey* finds the current mood – Happy
    - current mood (Happy) and tweet (Eating icecream) is passed to *updateMood*
    - *updateMood* calculates new mood as VeryHappy
    - *updateStateByKey* stores the new mood for Dinkar as VeryHappy

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

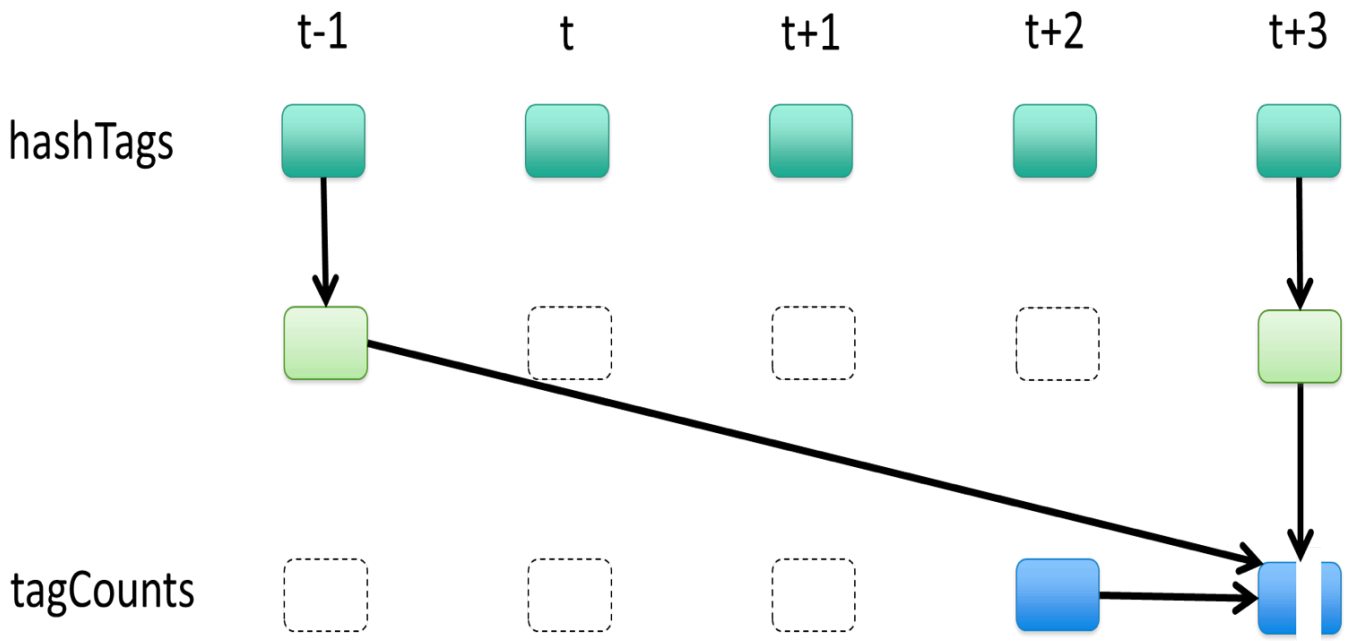
```
tweets.updateStateByKey(tweet =>
updateMood(tweet))
```

Fault tolerant Stateful processing

BIG DATA

Fault-tolerant Stateful Processing

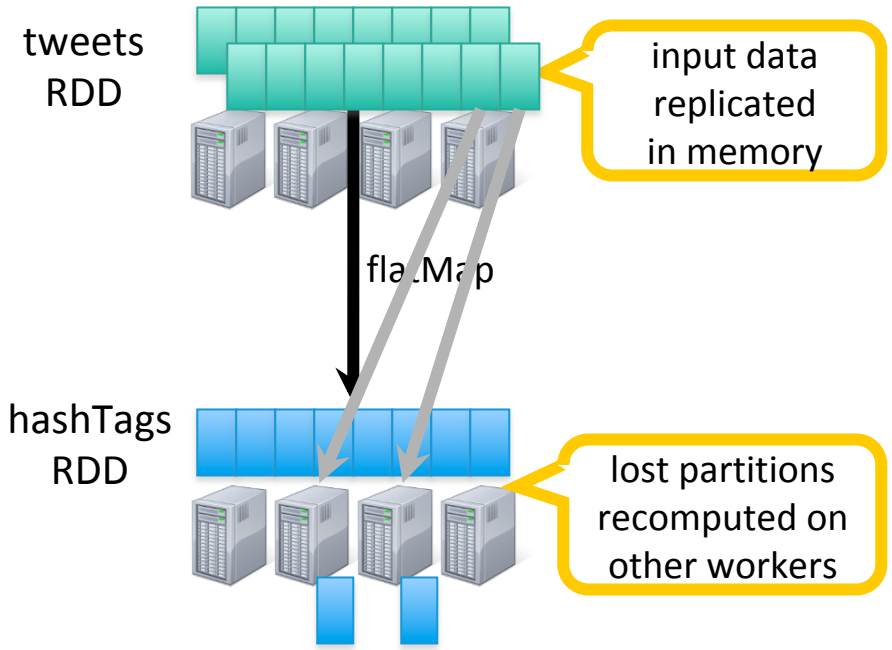
- All intermediate data are RDDs, hence can be recomputed if lost.



BIG DATA

Fault-tolerance

- RDDs remember the sequence of operations that created it from the original fault-tolerant input data
- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant
- Data lost due to worker failure, can be recomputed from input data



BIG DATA

Fault Tolerance

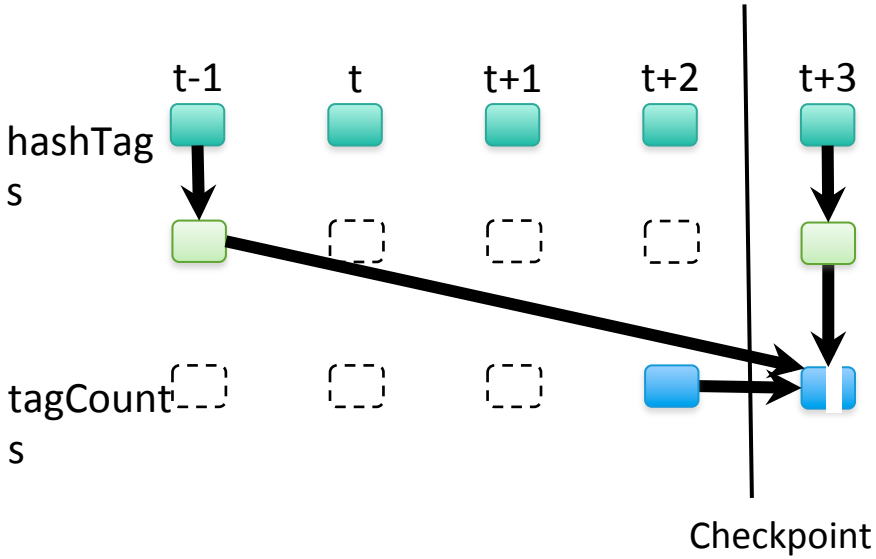
- What happens if there is a failure with
  - Stateless processing
  - Stateful processing

- Stateless
  - Previous history is not required.
  - Processing can just be recomputed
- Stateful
  - State from previous batches required for computation.
  - How much state to retain?

BIG DATA

Checkpointing

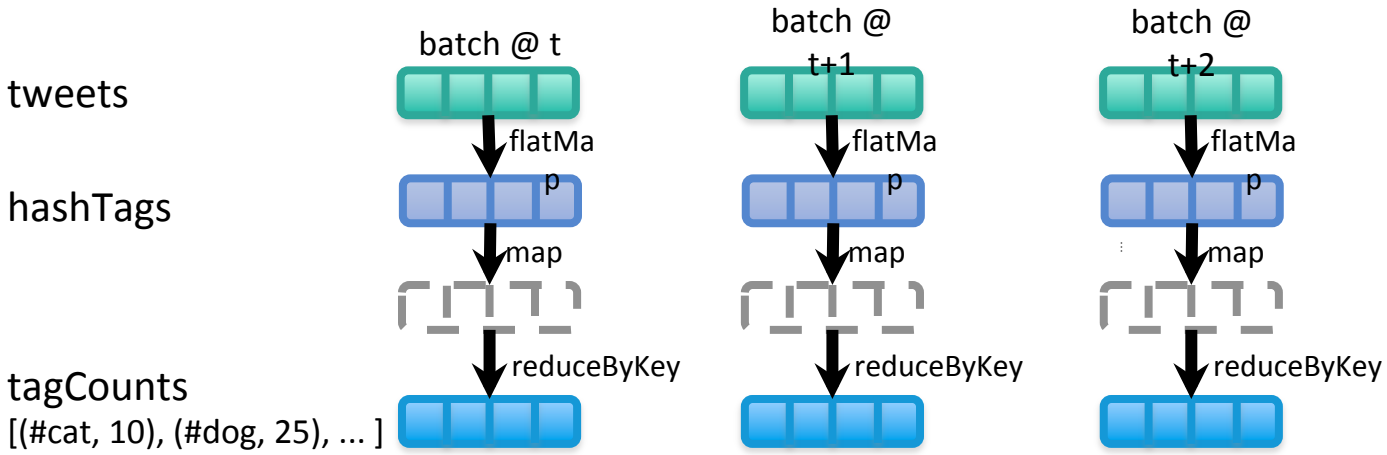
- Sometimes there may be too much data to be stored
  - For a streaming algorithm, may have to store all the streams
- Checkpointing
  - Stores an RDD
  - Forgets the lineage
  - A checkpoint at *t+2* will
    - store the hashTags and tagCounts at t+2
    - Forget the rest of the lineage



BIG DATA

Example – Count the hashtags

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.countByValue()
```



BIG DATA

Other Interesting Operations

- Maintaining arbitrary state, track sessions
  - Maintain per-user mood as state, and update it with his/her tweets

```
tweets.updateStateByKey(tweet => updateMood(tweet))
```

- Do arbitrary Spark RDD computation within DStream
  - Join incoming tweets with a spam file to filter out bad tweets

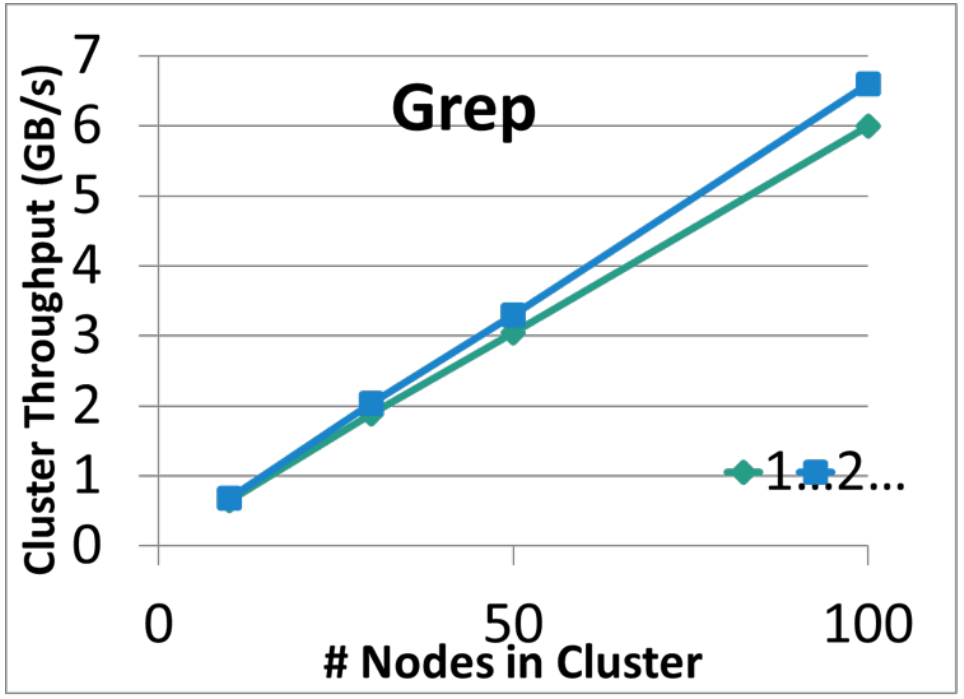
```
tweets.transform(tweetsRDD => {
        tweetsRDD.join(spamHDFSFile).filter(...)
})
```

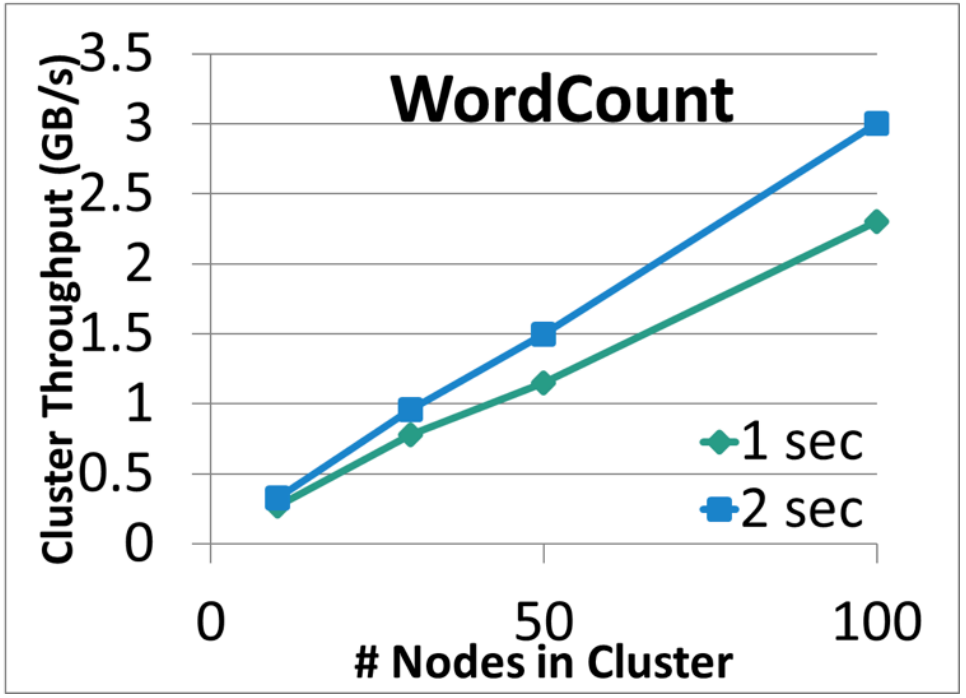
Performance

BIG DATA

Performance

- Can process **6 GB/sec (60M records/sec)** of data on 100 nodes at **sub-second** latency
  - Tested with 100 streams of data on 100 EC2 instances with 4 cores each

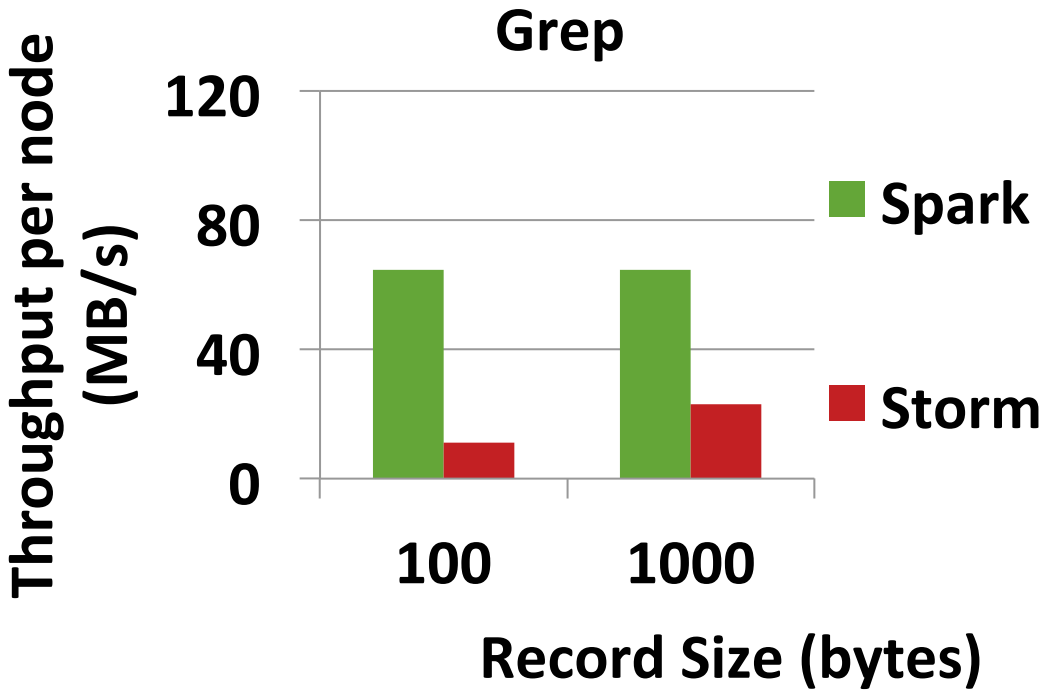


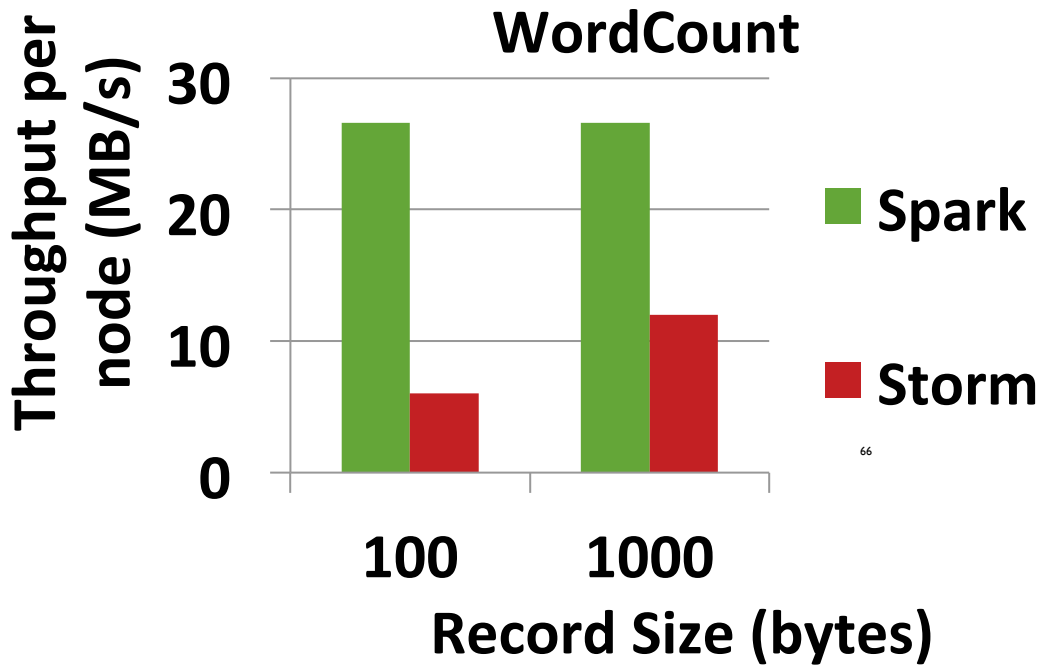


Comparison with Storm and S4

Higher throughput than Storm

- Spark Streaming: **670k** records/second/node
- Storm: **115k** records/second/node
- Apache S4: 7.5k records/second/node

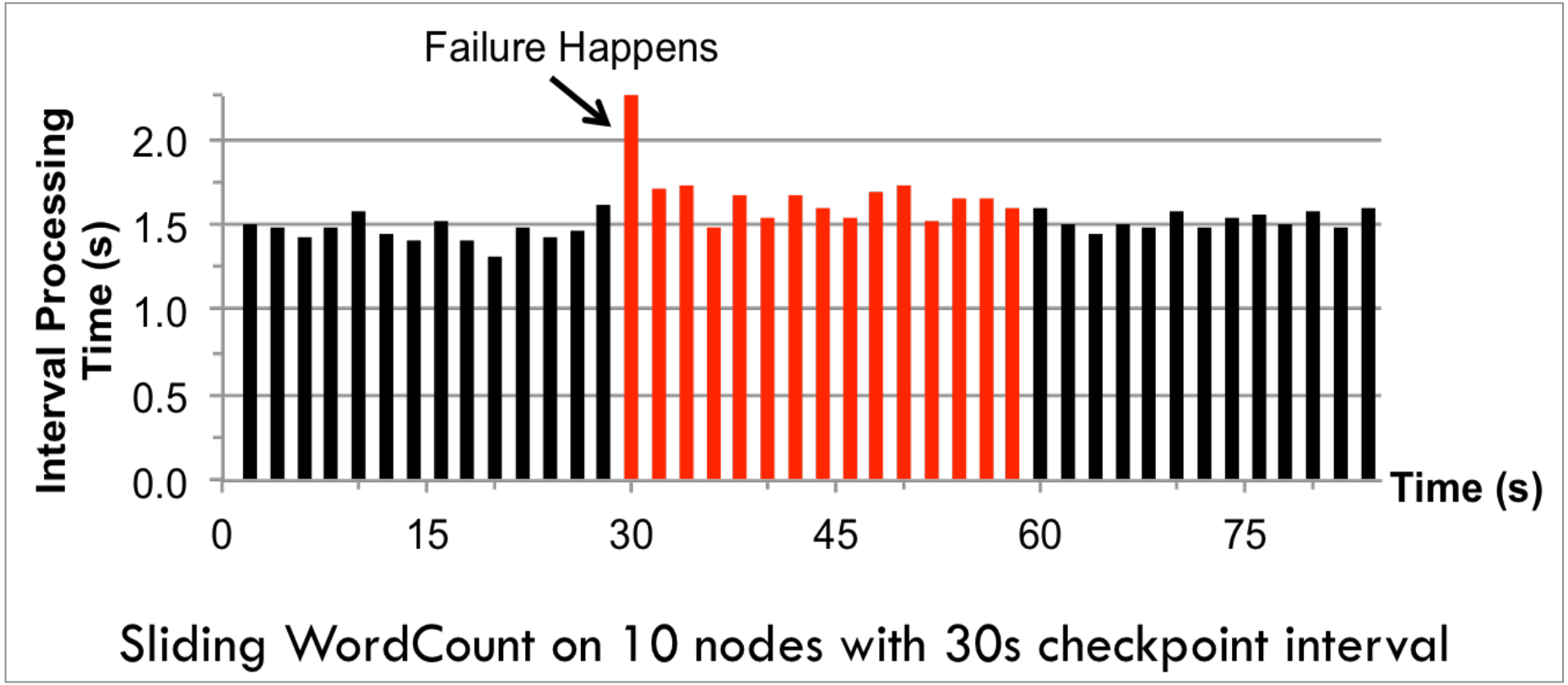




BIG DATA

Fast Fault Recovery

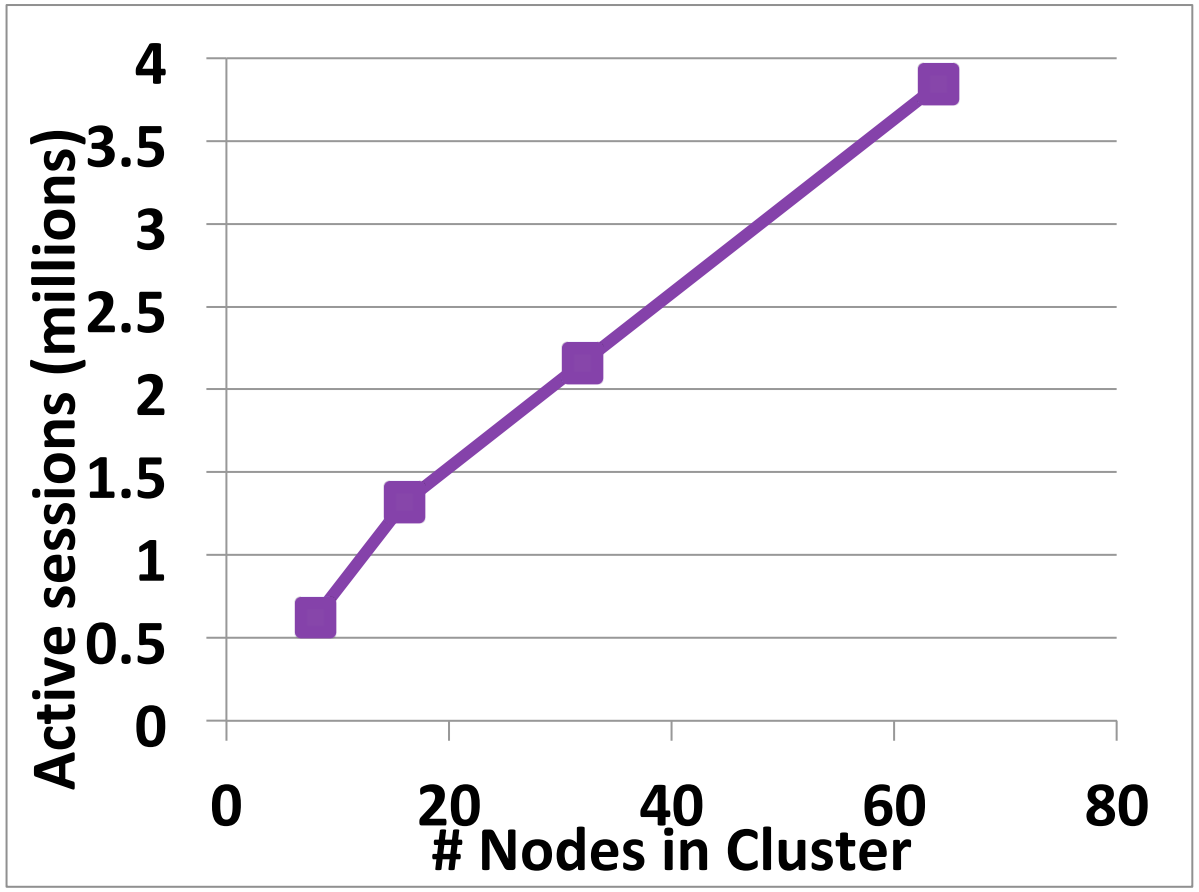
- Recovers from faults/stragglers within **1 sec**



BIG DATA

Real Applications: Conviva

- Real-time monitoring of video metadata

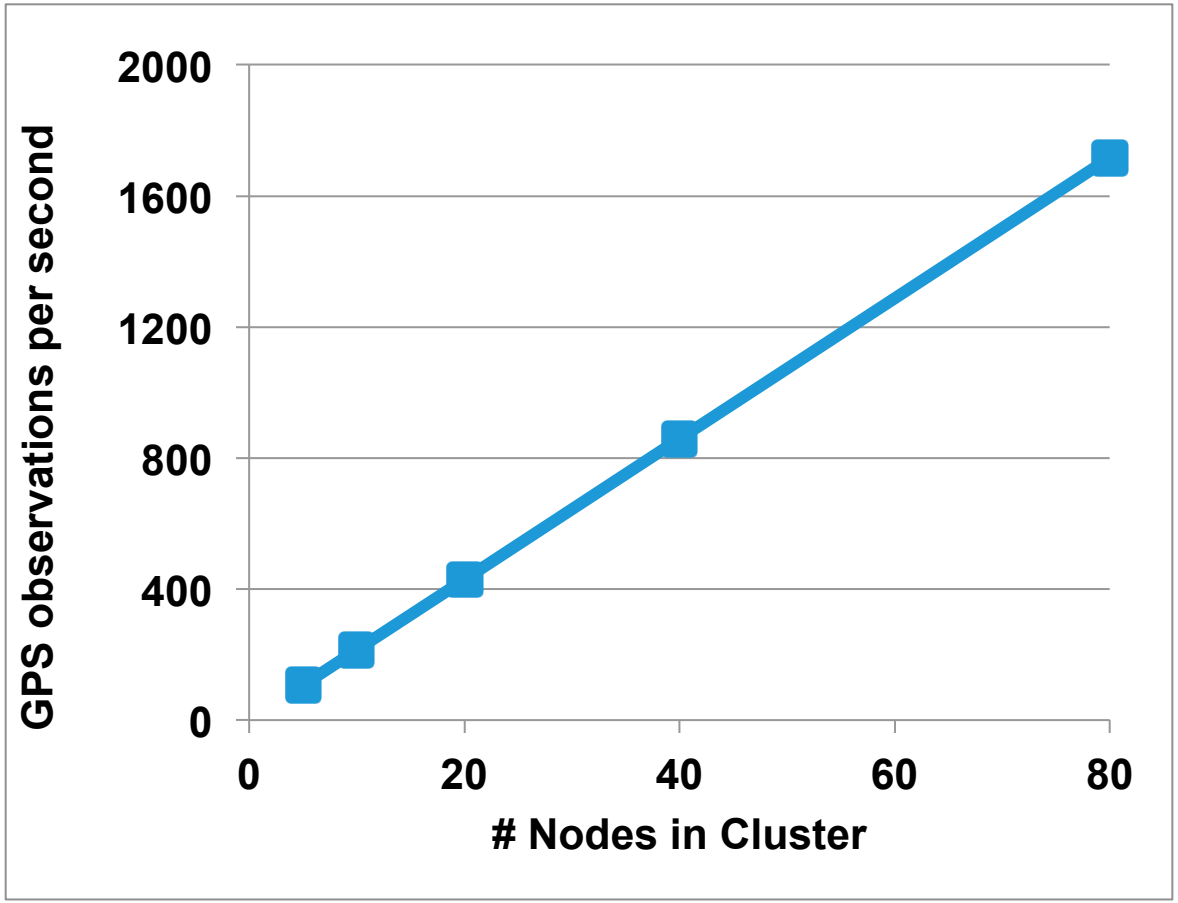


- Achieved 1-2 second latency
- Millions of video sessions processed
- Scales linearly with cluster size

BIG DATA

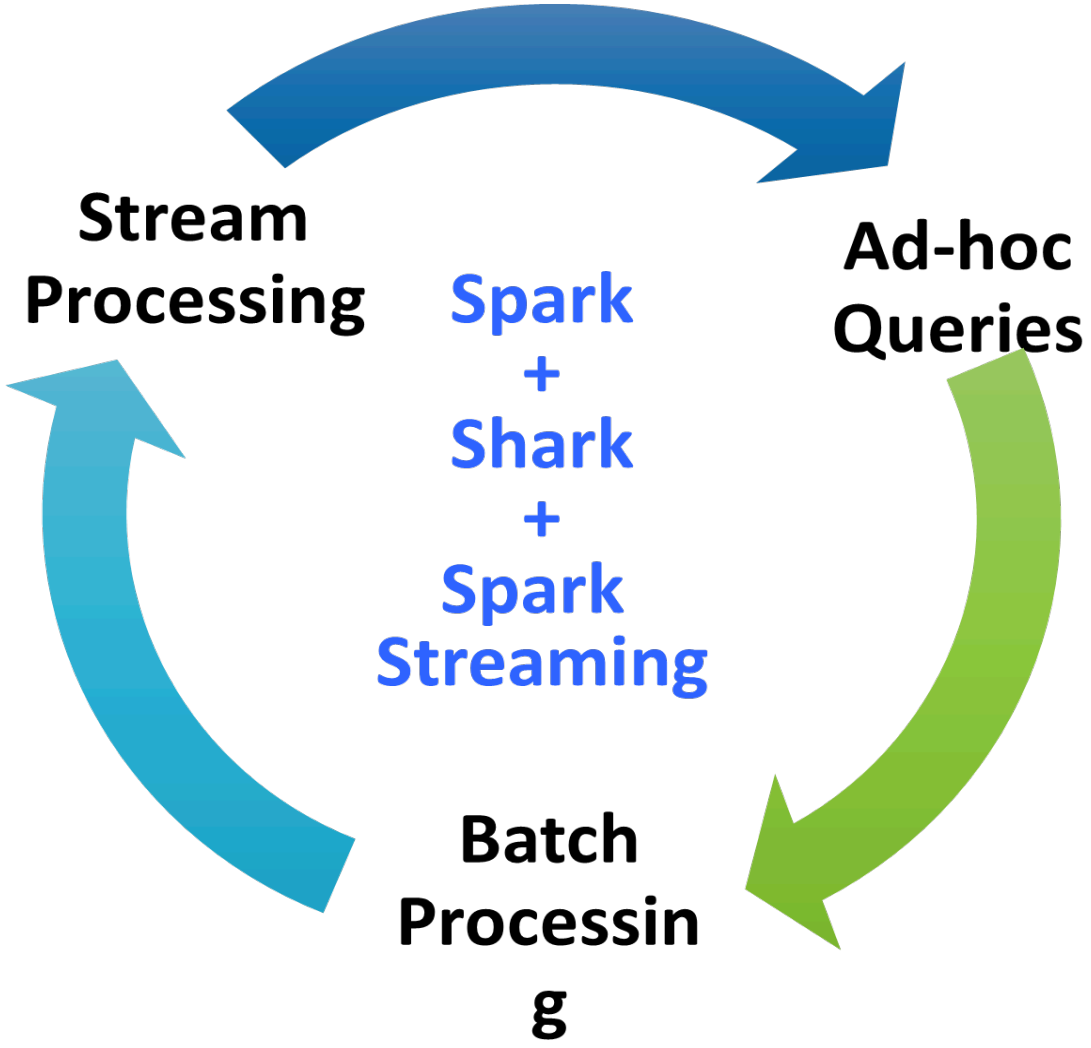
Real Applications: Mobile Millennium Project

- Traffic transit time estimation using online machine learning on GPS observations



- Markov chain Monte Carlo simulations on GPS observations
- Very CPU intensive, requires dozens of machines for useful computation
- Scales linearly with cluster size

Vision - one stack to rule them all



BIG DATA

Spark program vs Spark Streaming program

Spark Streaming program on Twitter stream

```
val tweets = ssc.twitterStream(<Twitter username>,
<Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

Spark program on Twitter log file

```
val tweets = sc.hadoopFile("hdfs://...")
val hashTags = tweets.flatMap (status => getTags(status))
hashTags.saveAsHadoopFile("hdfs://...")
```

BIG DATA

Vision - one stack to rule them all

- Explore data interactively using Spark Shell / PySpark to identify problems
- Use same code in Spark stand-alone programs to identify problems in production logs
- Use similar code in Spark Streaming to identify problems in live log streams

```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
```

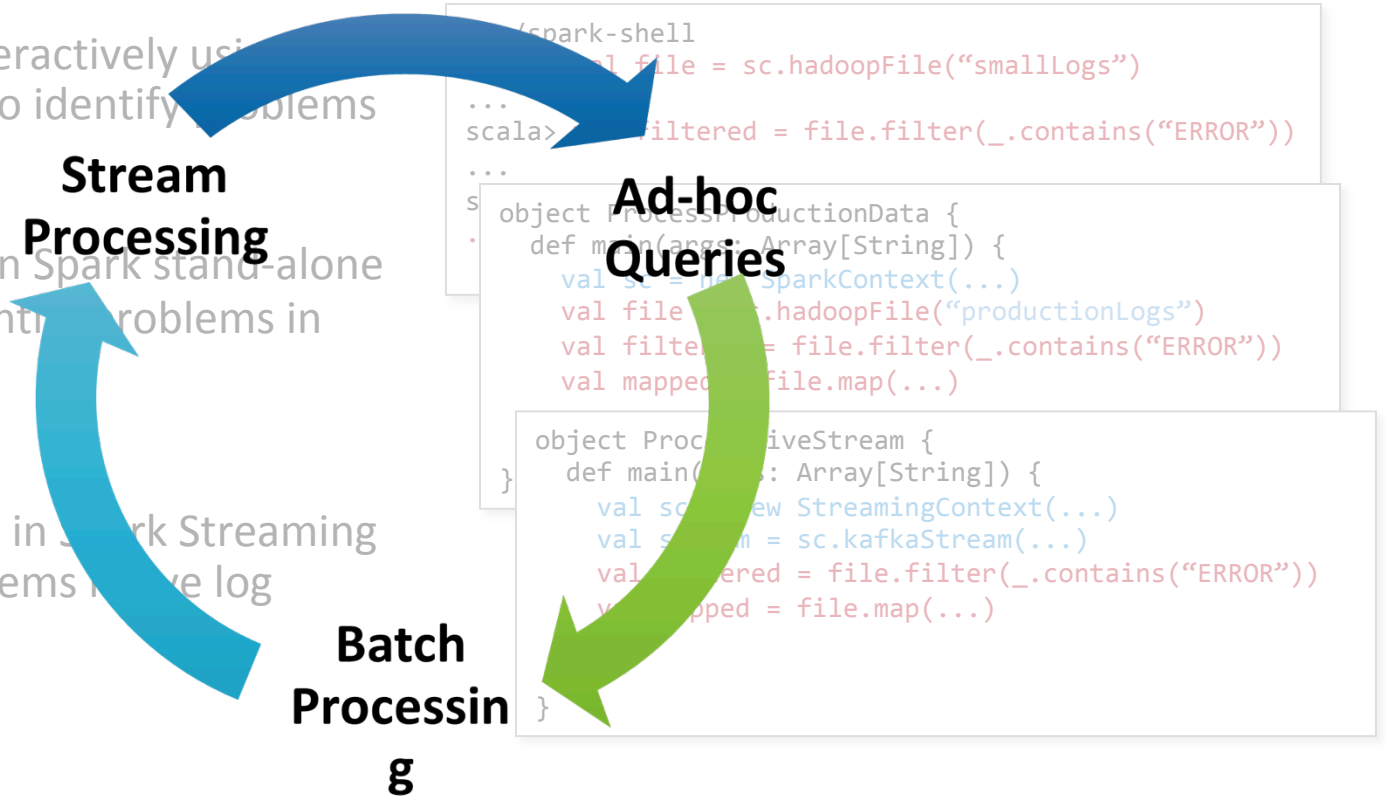
```
object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = file.map(...)
```

```
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = file.map(...)
    ...
  }
}
```

BIG DATA

Vision - one stack to rule them all

- Explore data interactively using Spark Shell / PySpark to identify problems
- Use same code in Spark stand-alone programs to identify problems in production logs
- Use similar code in Spark Streaming to identify problems live log streams



BIG DATA

Alpha Release with Spark 0.7

- Integrated with Spark 0.7
  - Import **spark.streaming** to get all the functionality
- Both Java and Scala API
- Give it a spin!
  - Run locally or in a cluster
- Try it out in the hands-on tutorial later today

BIG DATA

Limitations

- Streaming Spark processes data in batches
  - [?] Near Real Time

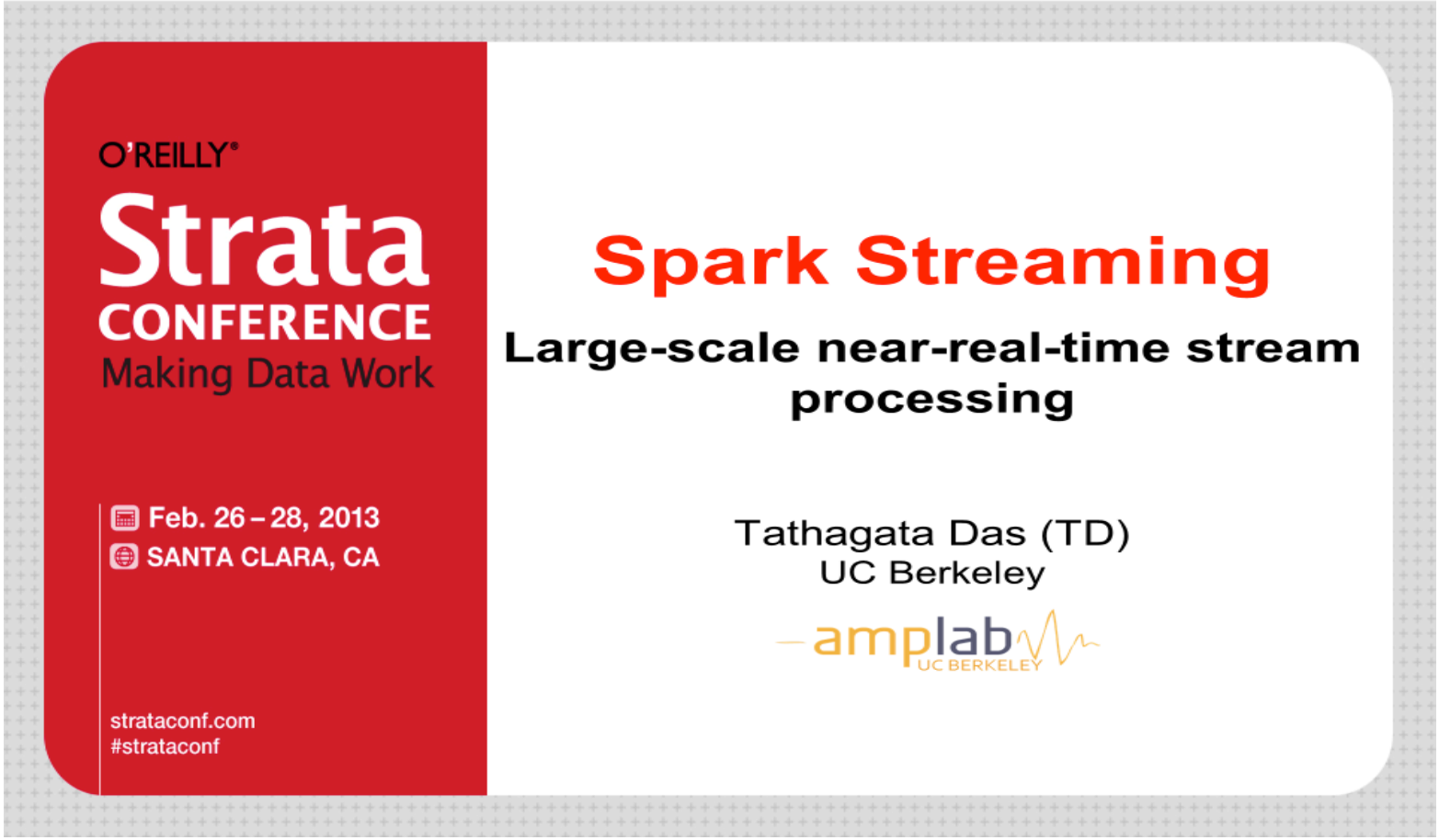
  - Not necessarily acceptable for certain scenarios

Summary

- Stream processing framework that is ...
  - Scalable to large clusters
  - Achieves second-scale latencies
  - Has simple programming model
  - Integrates with batch & interactive workloads
  - Ensures efficient fault-tolerance in stateful computations

- For more information, checkout the paper: http://tinyurl.com/dstreams

Source:



Video : Use Cases and Design Patterns for SPARK STREAMING,
https://www.youtube.com/watch?v=NHfggxItokg