Spark Streaming

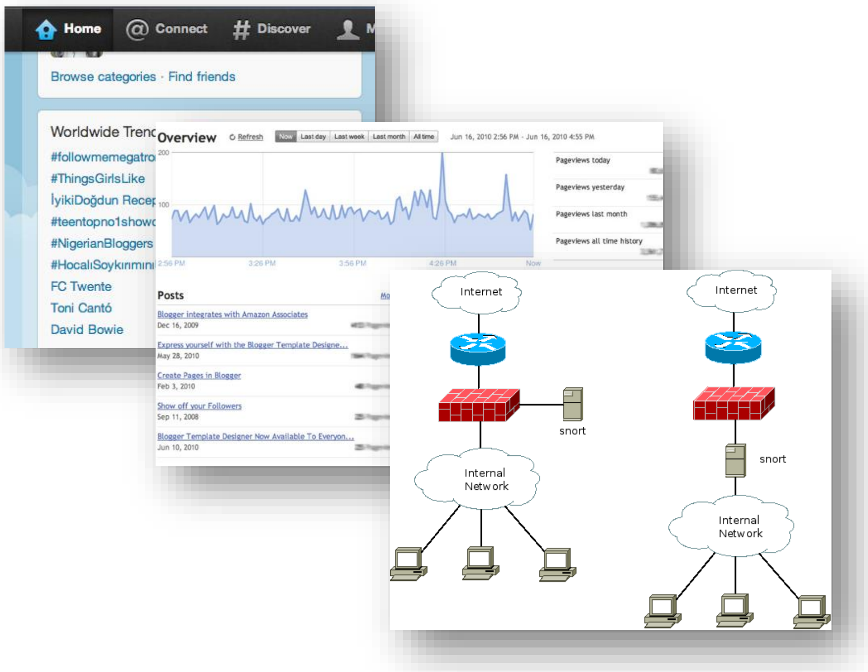
Guido Salvaneschi

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.

Motivation

- Many important applications must process large streams of live data and provide results in **near**-real-time
  - Social network trends
  - Website statistics
  - Intrustion detection systems
  - etc.



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

Example: Monitoring

Real-time monitoring of online video metadata

Two processing stacks:

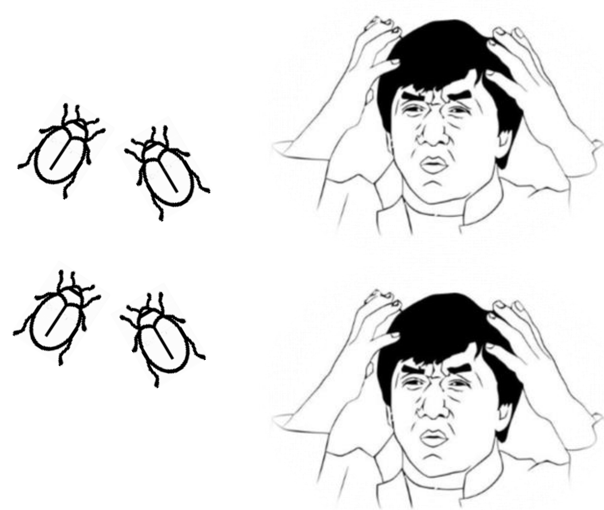
- Custom-built distributed stream processing system
  - 1000s complex metrics on millions of video sessions
  - Requires many dozens of nodes for processing
- Hadoop backend for offline analysis
  - Generating daily and monthly reports
  - Similar computation as the streaming system

Integrated batch & interactive processing ?

Two stacks

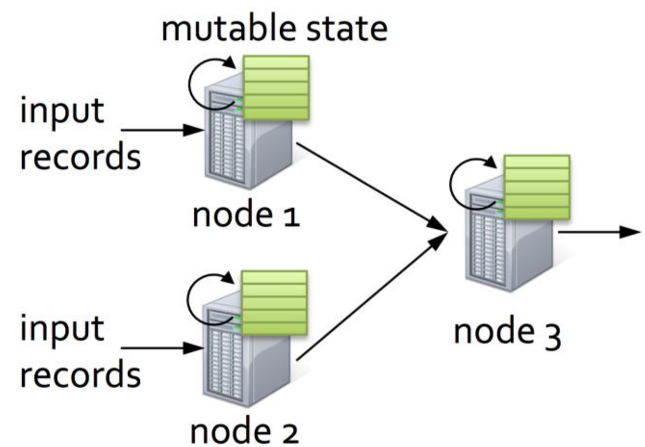
- Existing frameworks cannot do both
  - Either, **stream** processing of 100s of MB/s with **low latency**
  - Or, **batch** processing of TBs of data with **high latency**

- Extremely painful to maintain two different stacks
  - Different programming models
  - Doubles implementation effort
  - Doubles operational effort



Fault-tolerant Stream Processing

- Traditional processing model
- Pipeline (graph) of nodes
- Each node maintains mutable state
- Each input record updates the state and new records are sent out



- Mutable state is lost if node fails
- Making stateful stream processing 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!

- Trident
  - high-level abstraction for doing realtime computing on top of Storm
  - Use transactions to update state
  - Processes each record *exactly once*
  - Per-state transaction to external database is slow

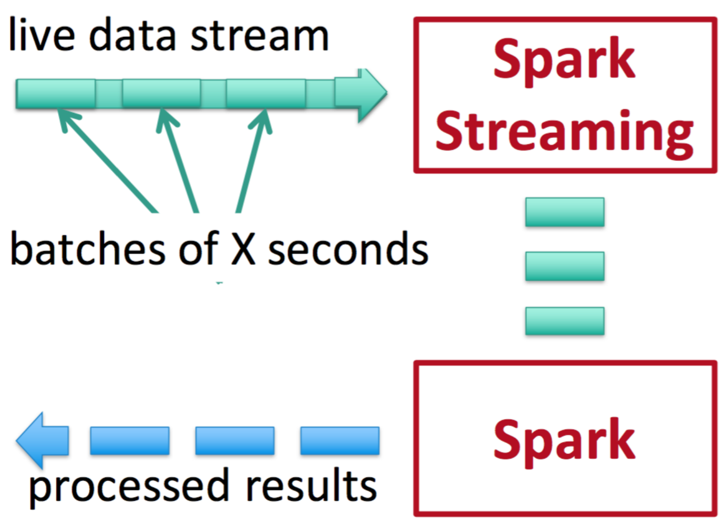
Spark streamaing

- Receive data streams from input sources
- Process them in a cluster
- Push out to databases/ dashboards

- Scalable, fault-tolerant, second-scale latencies

Microbatching

- Run a streaming computation as a series of very small, deterministic batch jobs
  - Chop up data streams into batches of few secs
    - Batch sizes as low as ½ second, latency ~ 1 second
  - Spark treats each batch of data as RDDs and processes them using RDD operations
  - Processed results are pushed out in batches



Spark Streaming Programming model

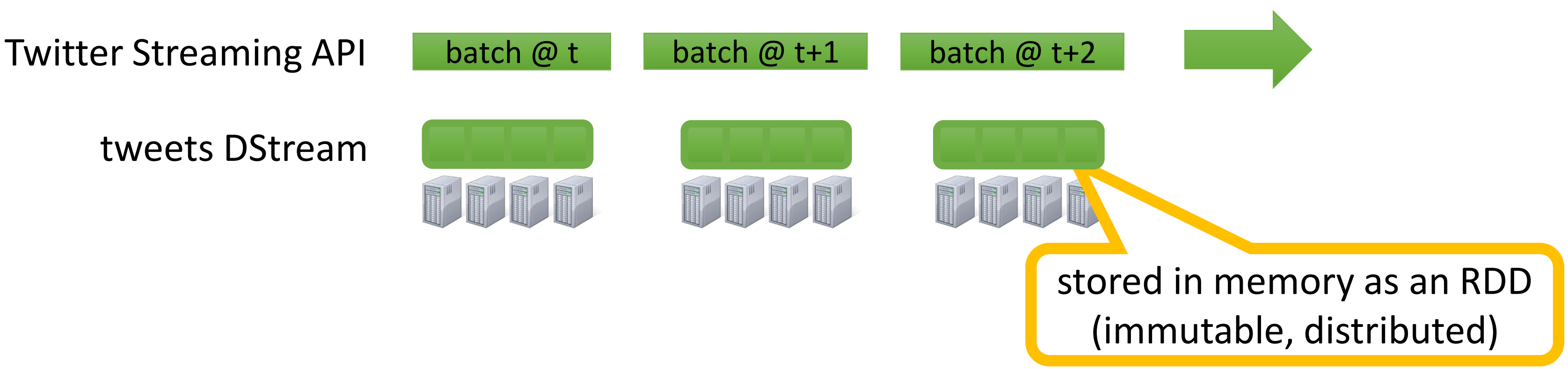
- *Discretized Stream (DStream)*
  - Represents a stream of data
  - Implemented as a sequence of RDDs

- DStreams API very similar to RDD API
  - Functional APIs in Scala, Java
  - Create input DStreams from different sources
  - Apply parallel operations

Example 1 – Get hashtags from Twitter

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

DStream: a sequence of RDD representing a stream of data

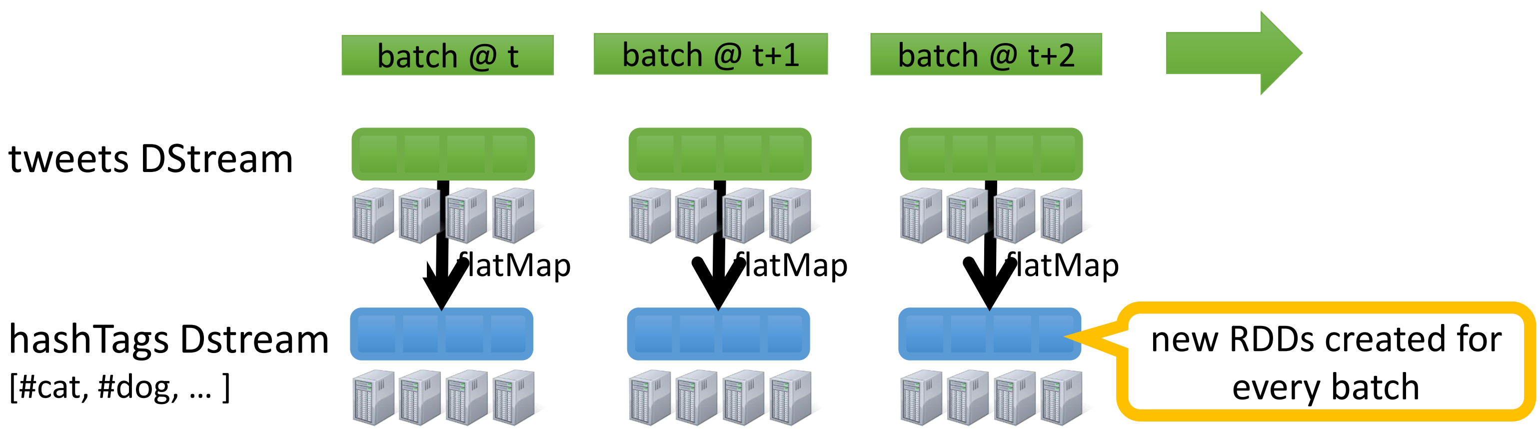


Example 1 – Get hashtags from Twitter

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

new DStream

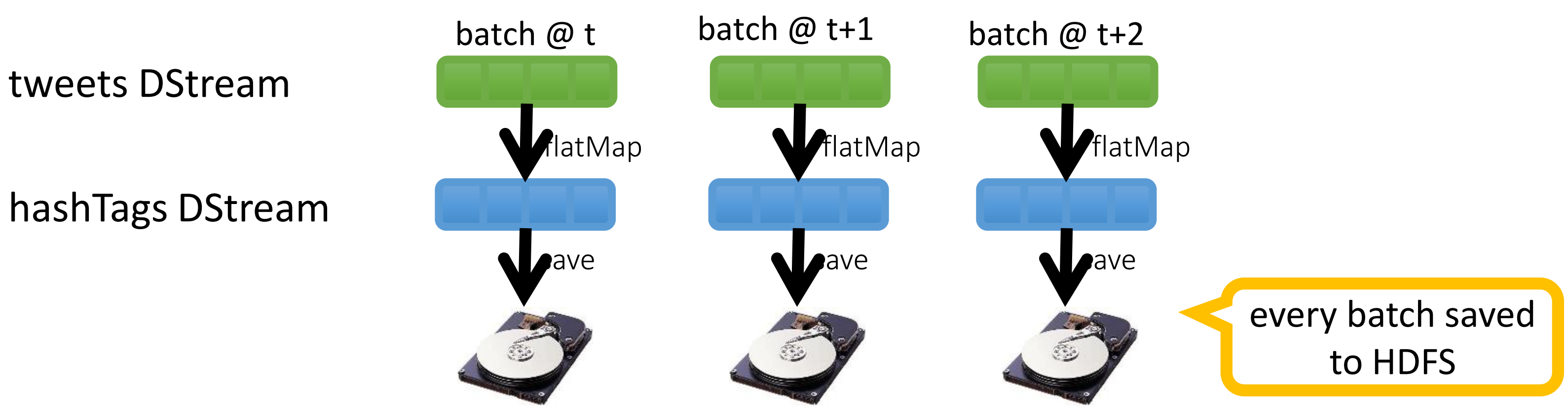
transformation: modify data in one Dstream to create another DStream



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://...")
```

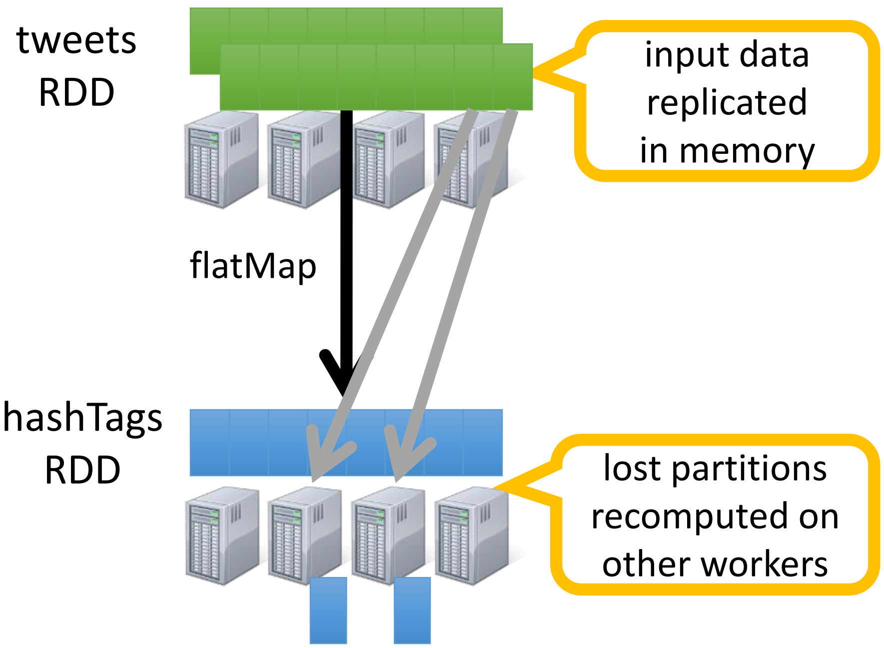
Java

```
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

Fault-tolerance

- RDDs are 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

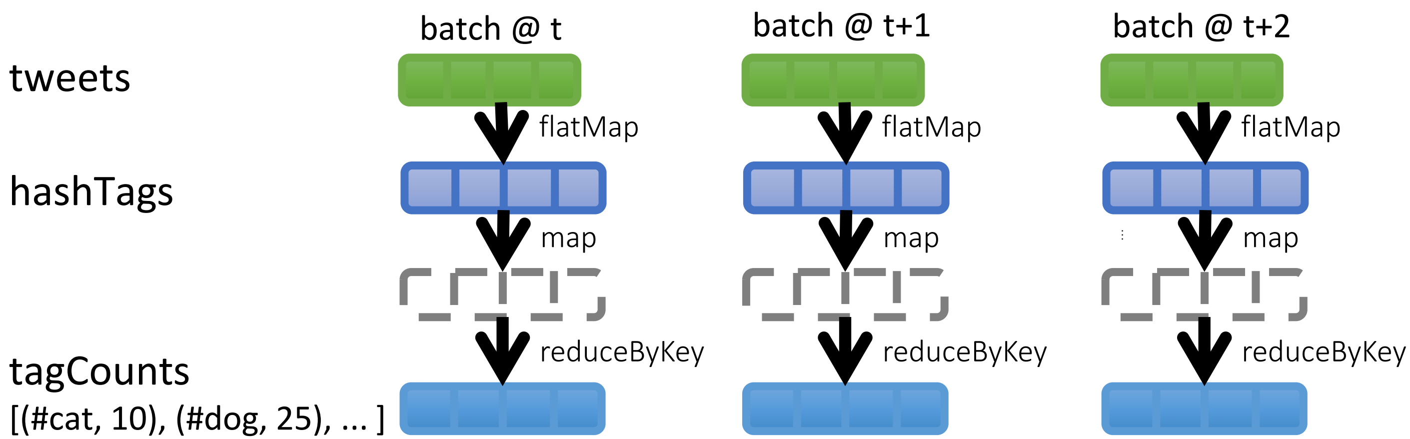


Key concepts

- **DStream** – sequence of RDDs representing a stream of data
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets
- **Transformations** – modify data from on DStream to another
  - Standard RDD operations – map, countByValue, reduce, join, …
  - Stateful operations – window, countByValueAndWindow, …
- **Output Operations – send data to external entity**
  - saveAsHadoopFiles – saves to HDFS
  - foreach – do anything with each batch of results

Example 2 – Count the hashtags

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



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()
```

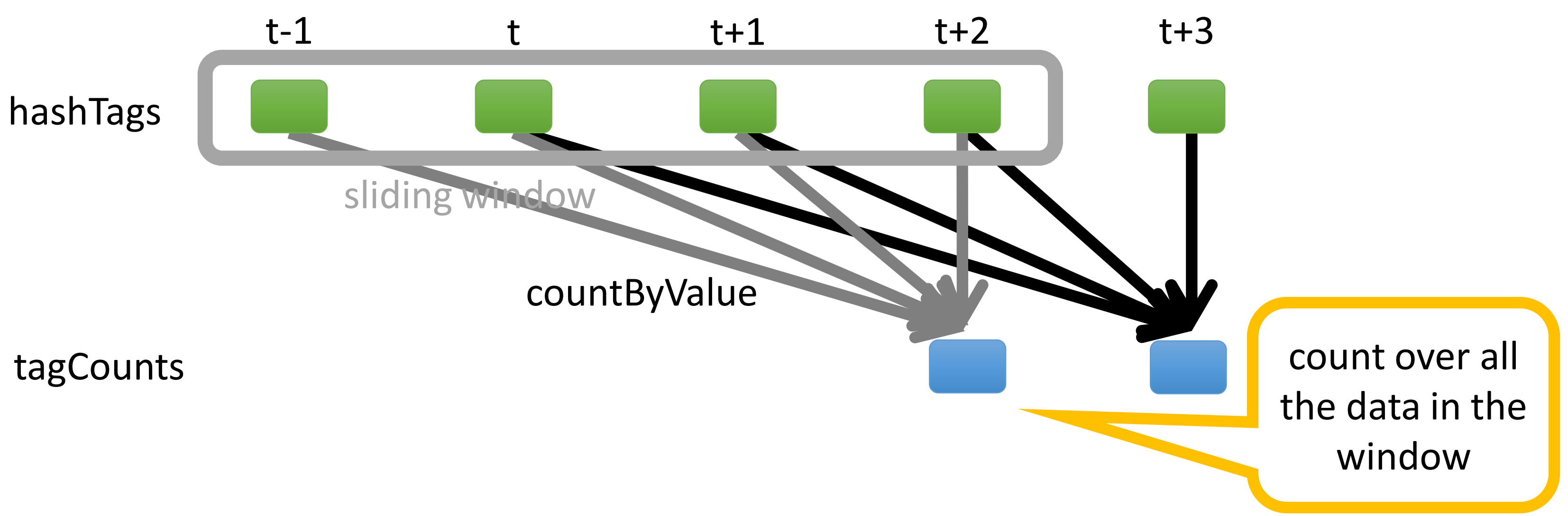
sliding window operation

window length

sliding interval

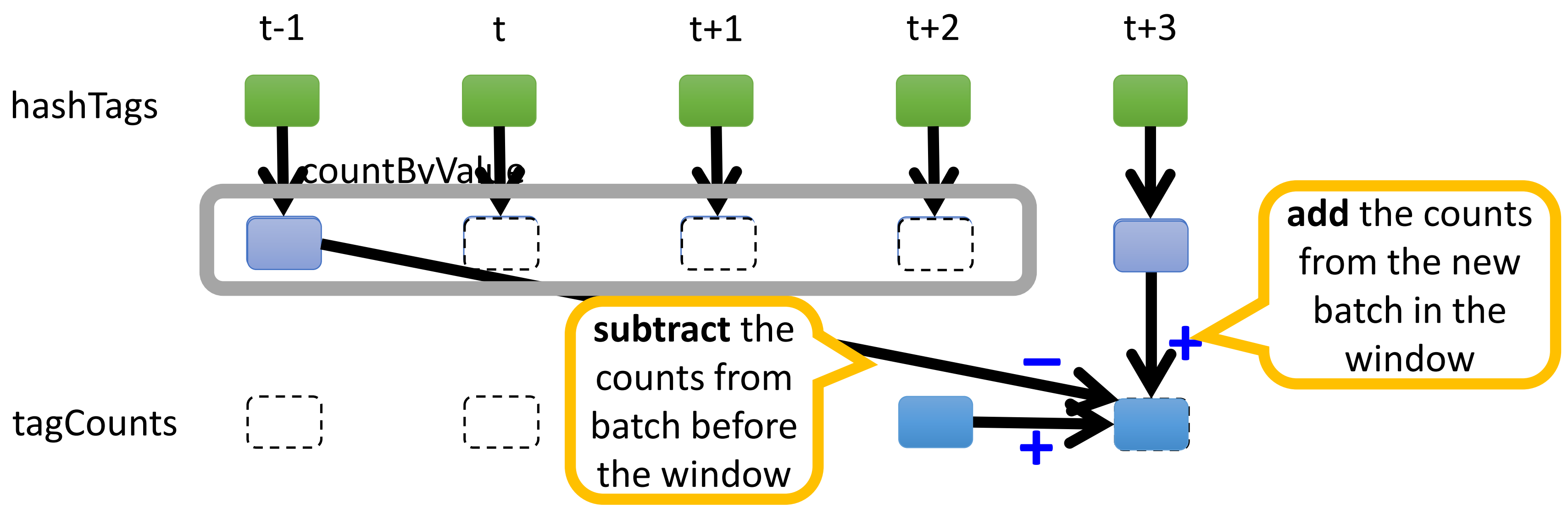
Example 3:
Counting the hashtags over last 10 mins

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



Smart window-based countByValue

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



Smart window-based *reduce*

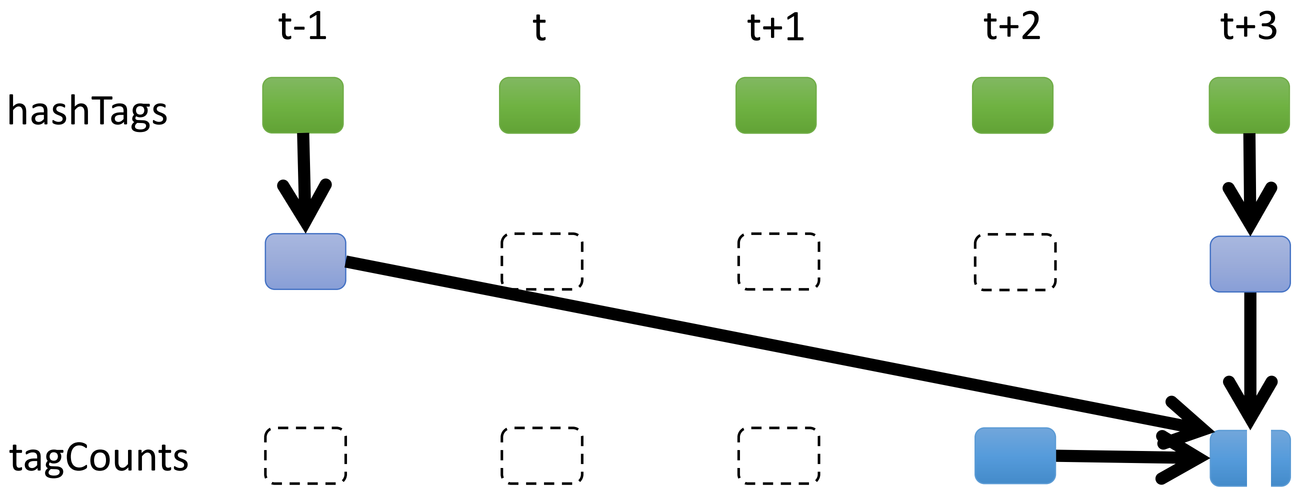
- Technique to incrementally compute count generalizes to many reduce operations
  - Need a function to “inverse reduce” (“subtract” for counting)

- Could have implemented counting as:

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

Fault-tolerant Stateful Processing

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



Fault-tolerant Stateful Processing

- State data not lost even if a worker node dies
  - Does not change the value of your result
- *Exactly once* semantics to all transformations
  - No double counting!

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))
```

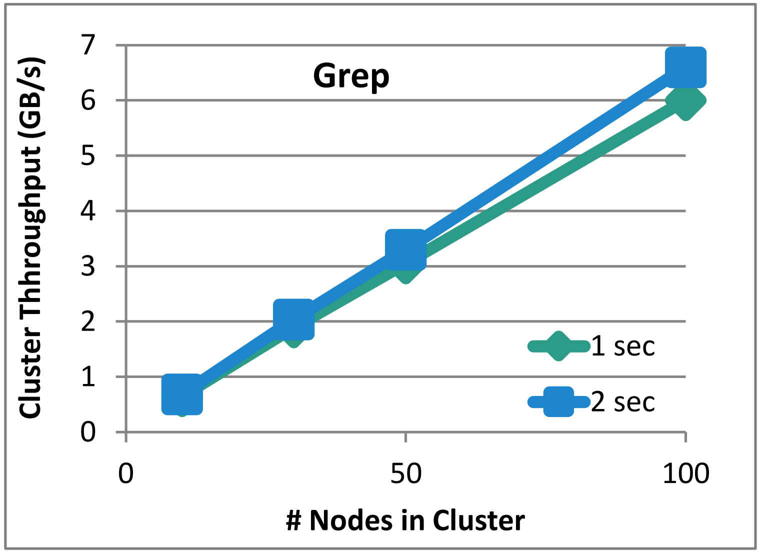
- Do arbitrary Spark RDD computation within DStream
  - Join incoming tweets with a spam file to filter out bad tweets

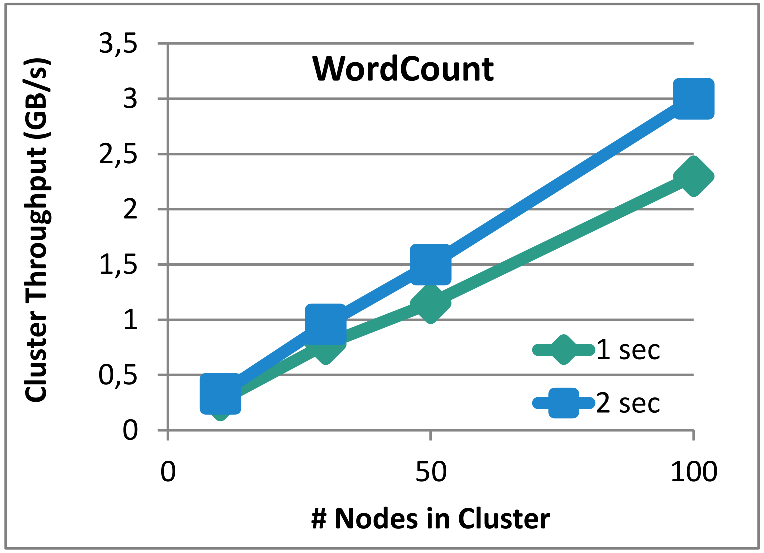
```
tweets.transform(tweetsRDD => {
        tweetsRDD.join(spamHDFSFile).filter(...)
})
```

Performance

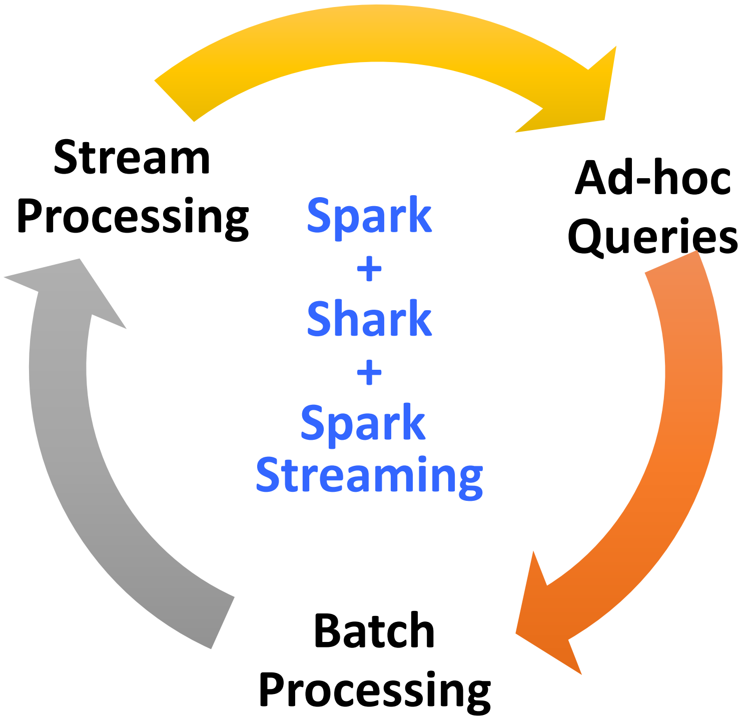
Can process **6 GB/sec (60M records/sec**) of data on 100 nodes at **sub-second** latency

- Tested with 100 streams of data on 100 EC2 instances with 4 cores each





Vision - *one stack to rule them all*



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://...")
```

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"))
...
s
.
```

```
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(...)
    ...
  }
}
```

Questions?

Sources & References

Lecture mostly mased on:

- http://ampcamp.berkeley.edu/wp-content/uploads/2013/02/large-scale-near-real-time-stream-processing-tathagata-das-strata-2013.pdf

Thoughts on realtime analytics:

- https://iwringer.wordpress.com/tag/bigdata/