Spark Streaming

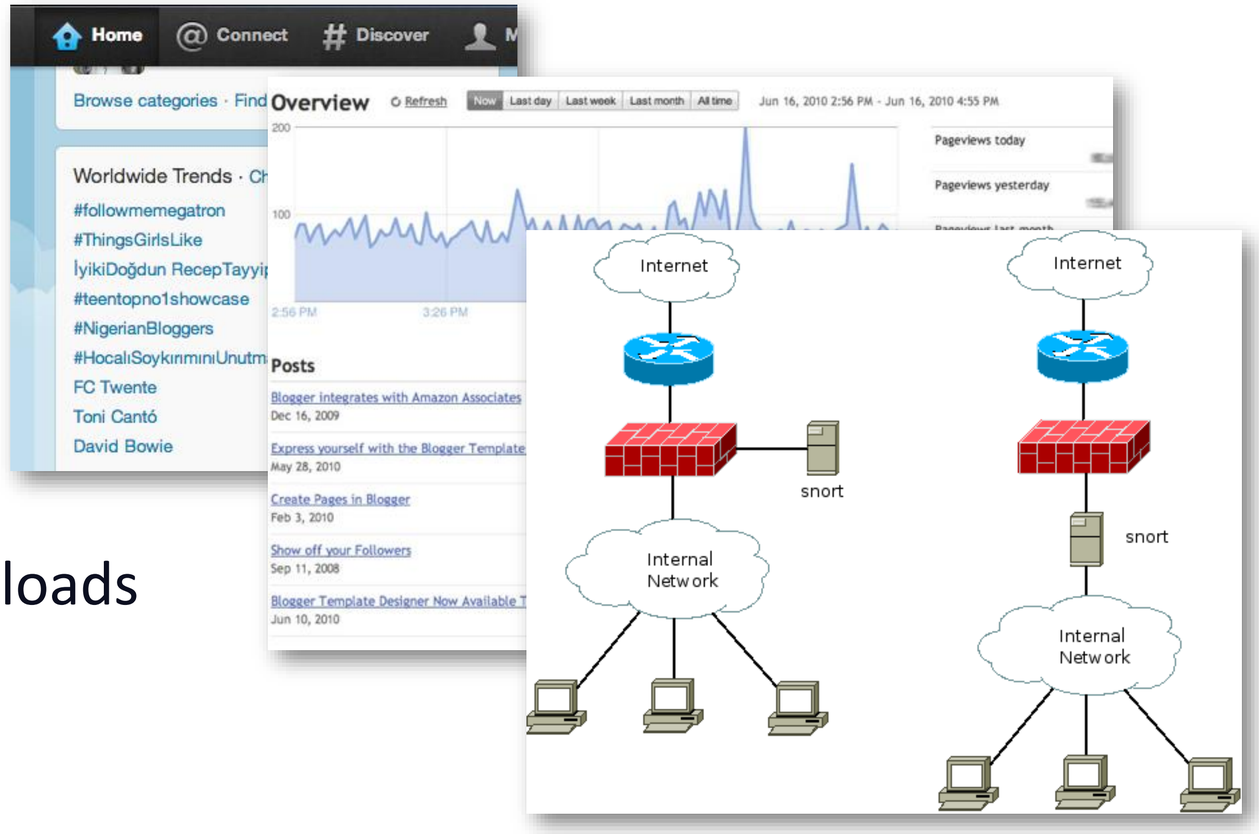
Large-scale near-real-time stream processing

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.

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.



- Require large clusters to handle workloads

- Require latencies of few seconds

Need for a framework …

… for building such complex stream processing applications

But what are the requirements
from such a framework?

Requirements

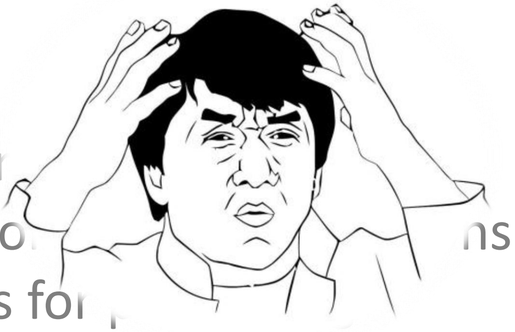
- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model

Case study: Conviva, Inc.

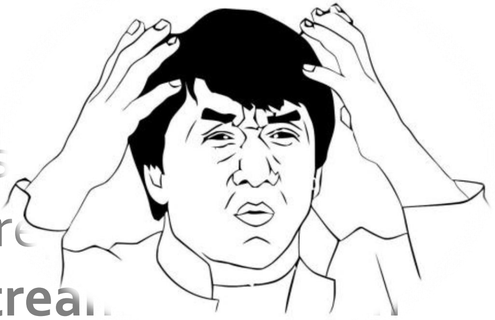
- Real-time monitoring of online video metadata
  - HBO, ESPN, ABC, SyFy, …
- Two processing stacks
  - Custom-built distributed stream processing system
    - 1000s complex metrics on millions of video sessions
    - Requires many dozens of nodes for processing
  - Hadoop backend for offline analysis
    - Generating daily and monthly reports
    - **Similar computation as the streaming system**

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-b… …ctrib… …eam pr…
    - 1000… …ex me… …n millio… …ns
    - Requi… …any dozens of nodes for …
  - Hadoop b… …d fo… …analys…
    - Gene… …aily a… …thly re…
    - **Simila… …putation as the strea…**



Requirements

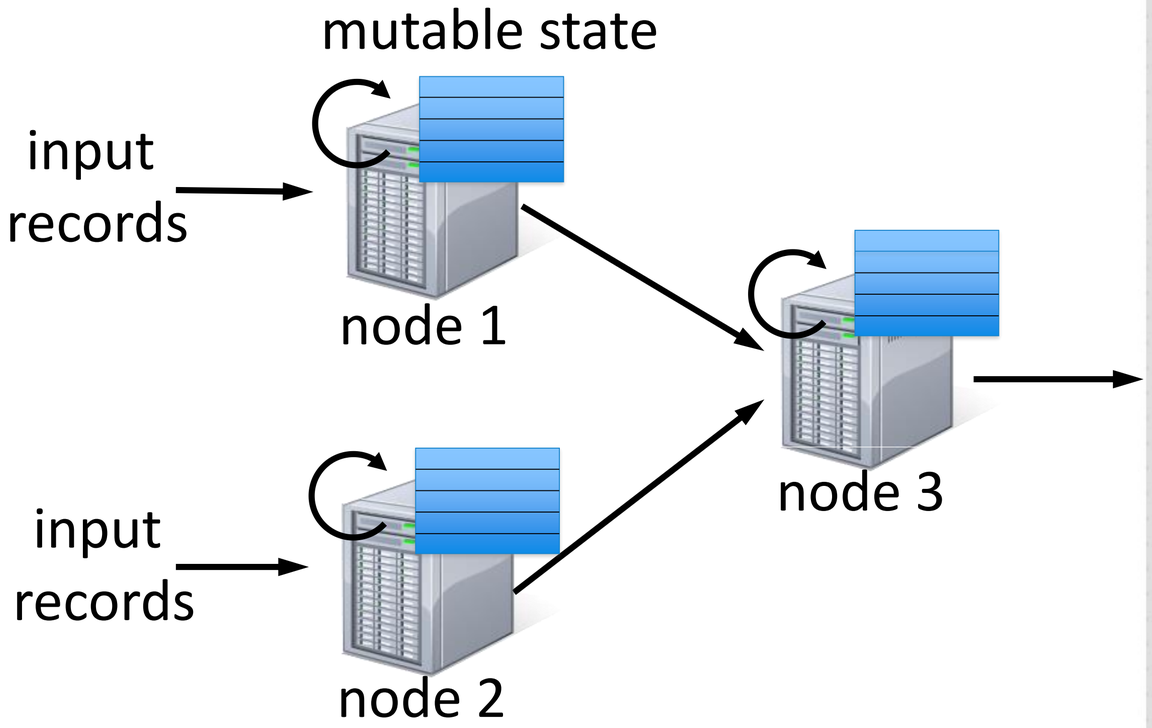
- **Scalable** to large clusters
- **Second-scale** latencies
- **Simple** programming model
- **Integrated** with batch & interactive processing

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



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

Requirements

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

Spark Streaming

Spark Streaming

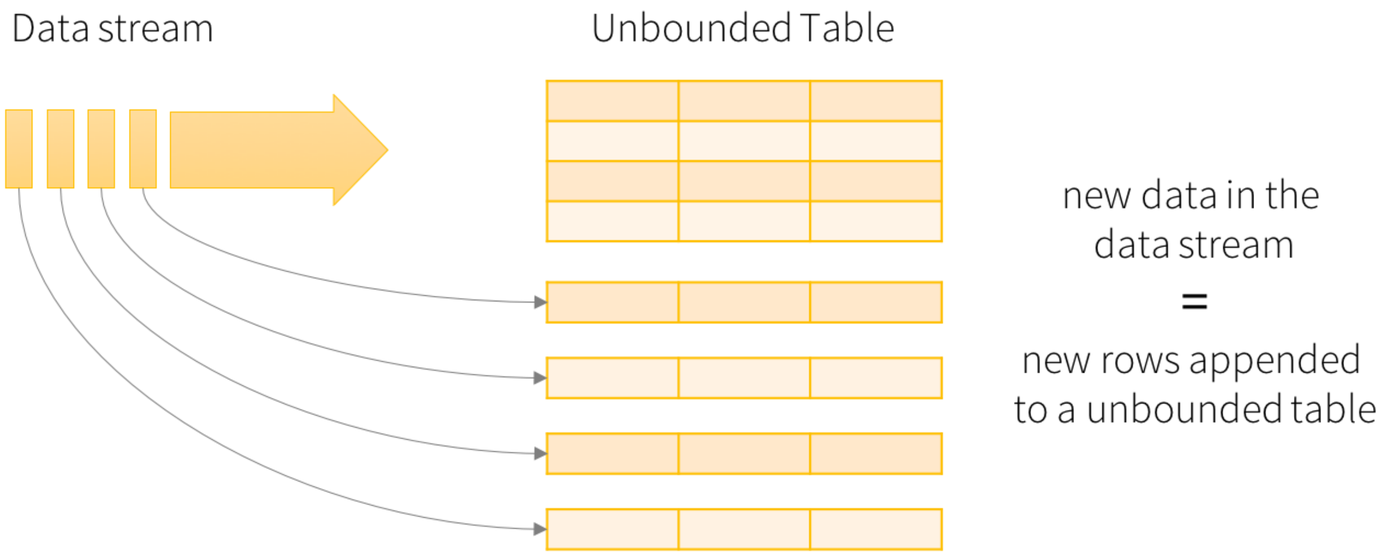
- Stream processing and batch processing use same API
- Two types of Streams –
  - DStreams
  - Structured Streams (Datasets/Dataframes)

DStreams

- Series of RDDs
- Basic Sources
  - File systems, socket connections
- Advance Sources
  - Kafka, Flume, Kinesis,..

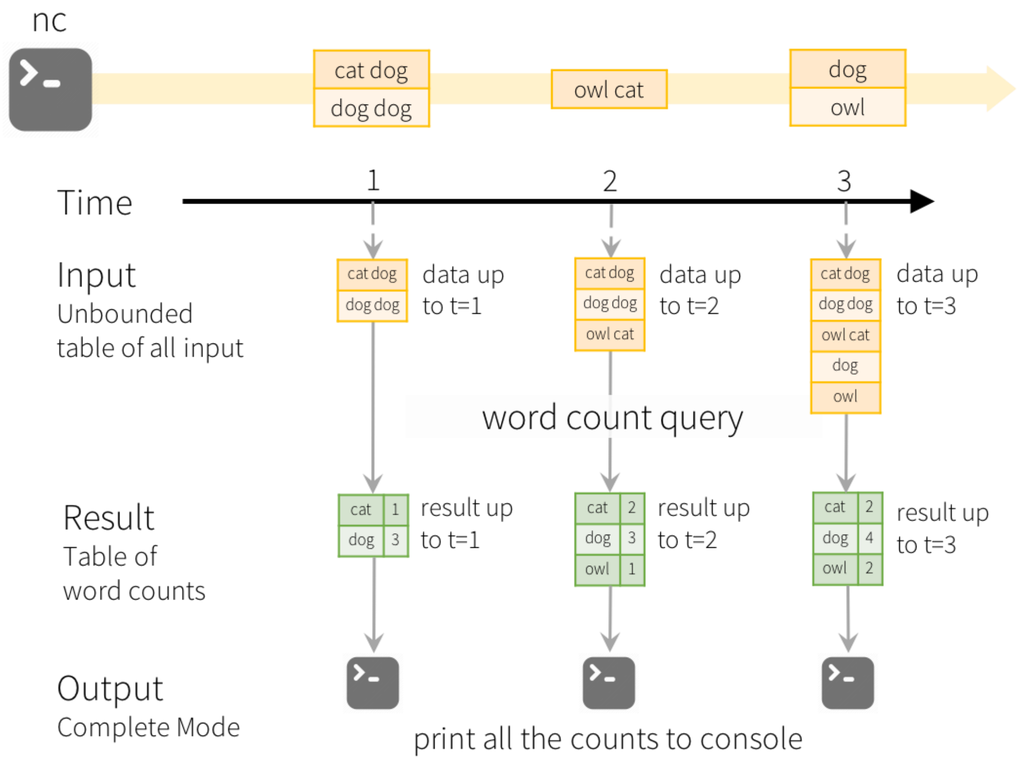
Structured Streaming

- Built on Spark SQL Engine
- Dataset/Dataframe API
- Late events aggregated correctly
- Exactly once semantics
  - Stream source must track progress (i.e. Kafka offsets)
  - Stream Sink must be idempotent



Data stream as an unbounded table

Source: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html



Model of the Quick Example

https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

Spark ML Pipelines

Pipeline consists of a sequence of PipelineStages
which are either Transformers or Estimators

- **Transformer**
  transform() converts one DataFrame to another DataFrame
- **Estimator**
  fit() accepts a DataFrame and produces a Model
    a Model is a Transformer

Bayes Model

- Very fast, simple
- $P(c\,|x) = \frac{P(x|c)P(c)}{P(x)}$

Demo: Analyze Twitter Streams with Spark ML and Spark Streaming

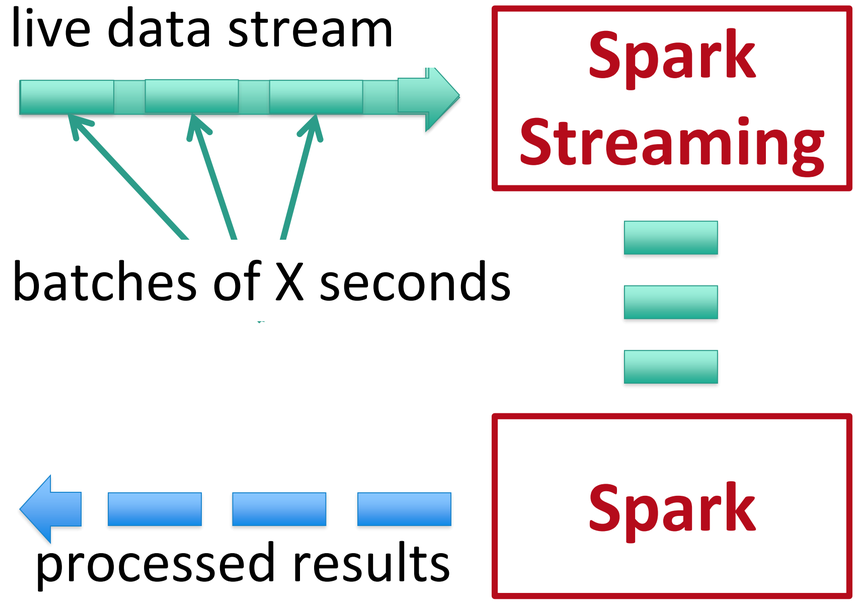
- **Part 1**: Create Naïve Bayes Model with training data
- **Part 2**: Sentiment Analysis of live Twitter Stream using Naïve Bayes Model and Spark Dstreams
- **Part 3**: Structured Streaming example – get Twitter trending hashtags in 10 minute windows using Twitter timestamp
- https://github.com/ekraffmiller/SparkStreamingDemo

map(*func*)	map(*func*) returns a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	flatMap(*func*) is similar to map(*func*) but each input item can be mapped to 0 or more output items and returns a new DStream by passing each source element through a function *func*.
filter(*func*)	filter(*func*) returns a new DStream by selecting only the records of the source DStream on which *func* returns true.
reduce(*func*)	reduce(*func*) returns a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func*.
groupBy(*func*)	groupBy(*func*) returns the new RDD which basically is made up with a key and corresponding list of items of that group.

Discretized Stream Processing

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

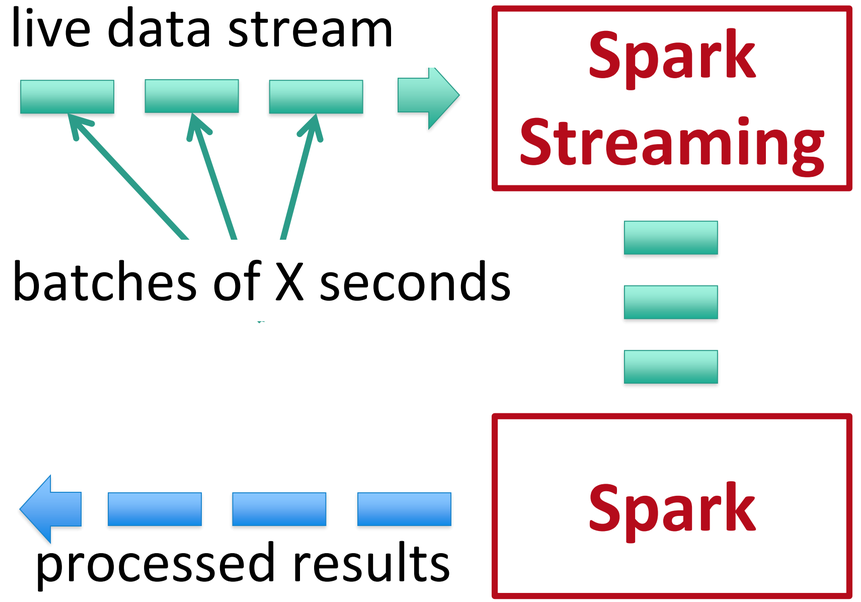
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Discretized Stream Processing

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

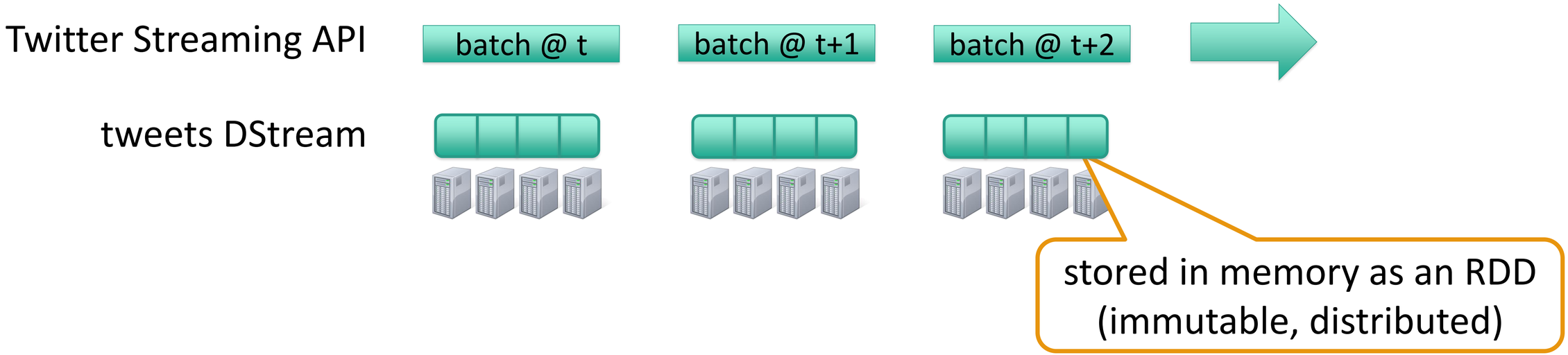
- Batch sizes as low as ½ second, latency ~ 1 second
- Potential for combining batch processing and streaming processing in the same system



Example 1 – Get hashtags from Twitter

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

DStream: a sequence of RDD representing a stream of data

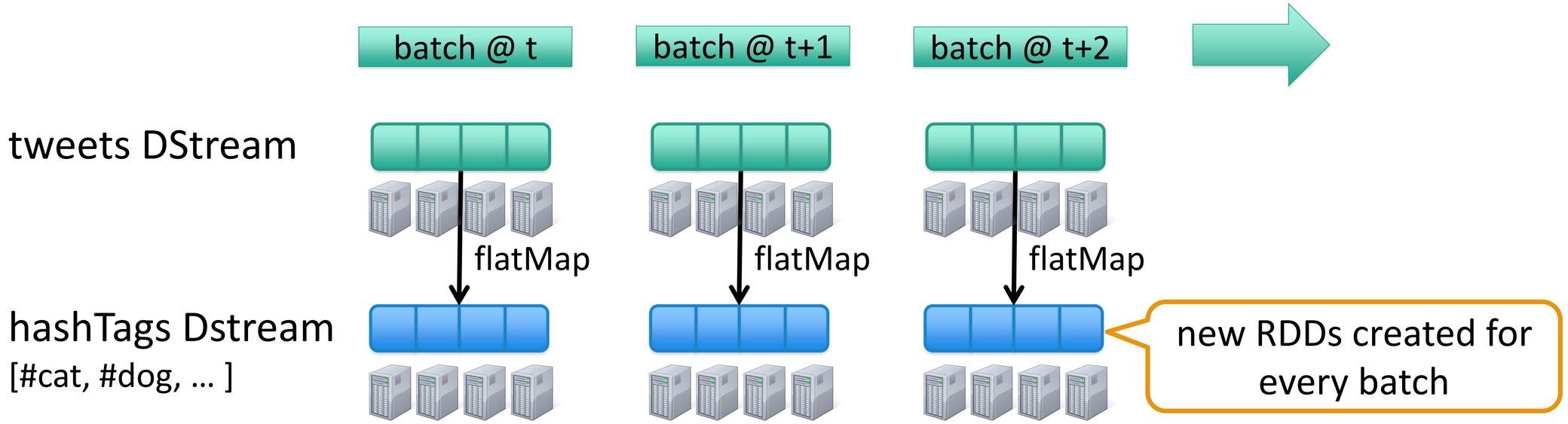


Example 1 – Get hashtags from Twitter

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

new DStream

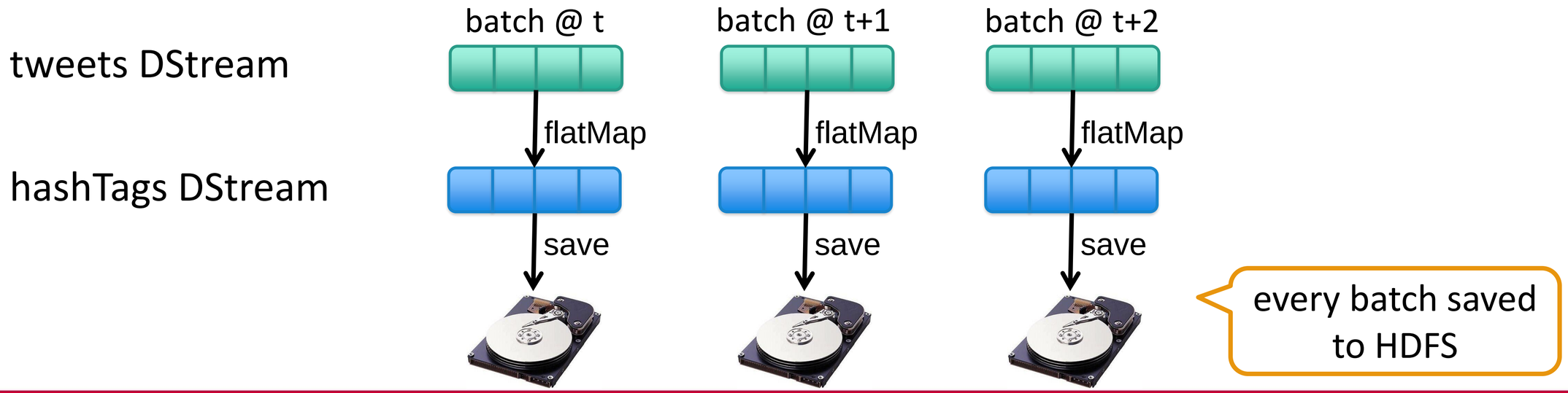
transformation: modify data in one Dstream to create another DStream



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://...")
```

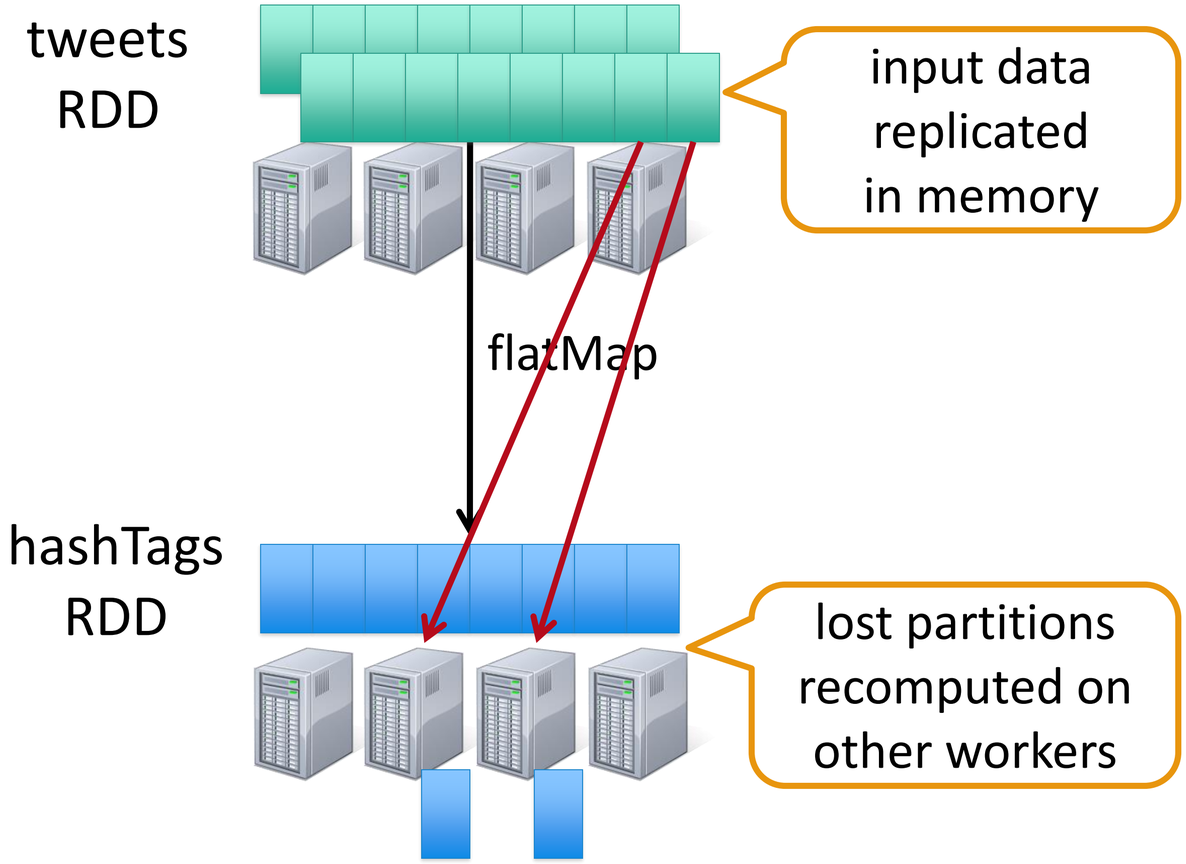
Java

```
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

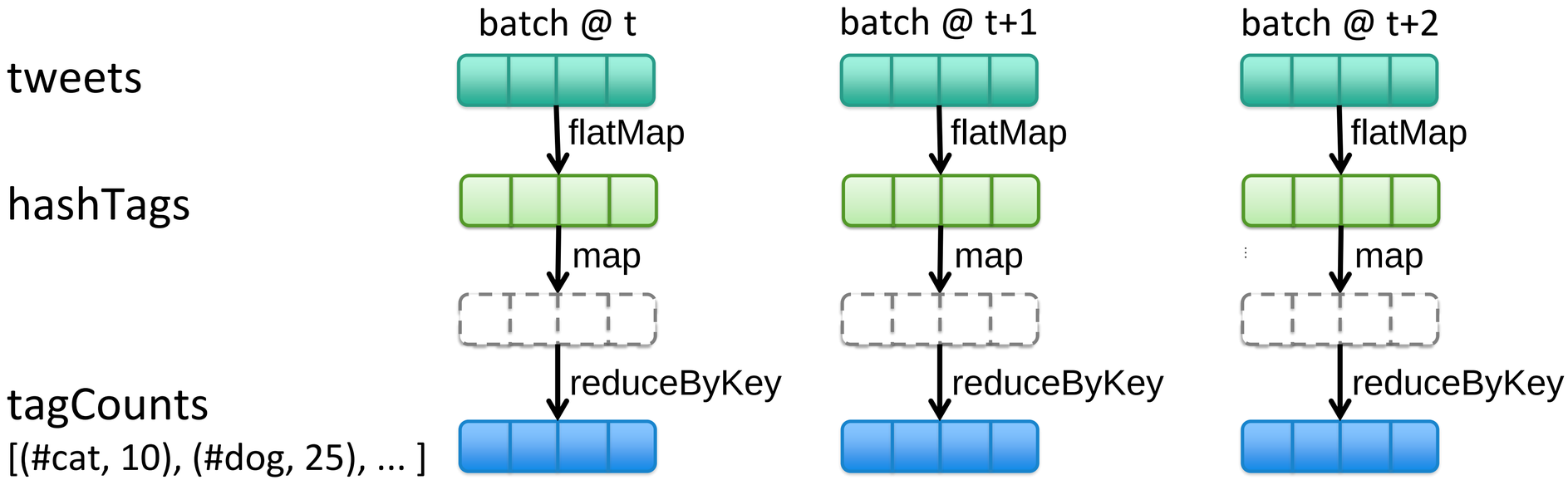
- **DStream** – sequence of RDDs representing a stream of data
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets

- **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()
```

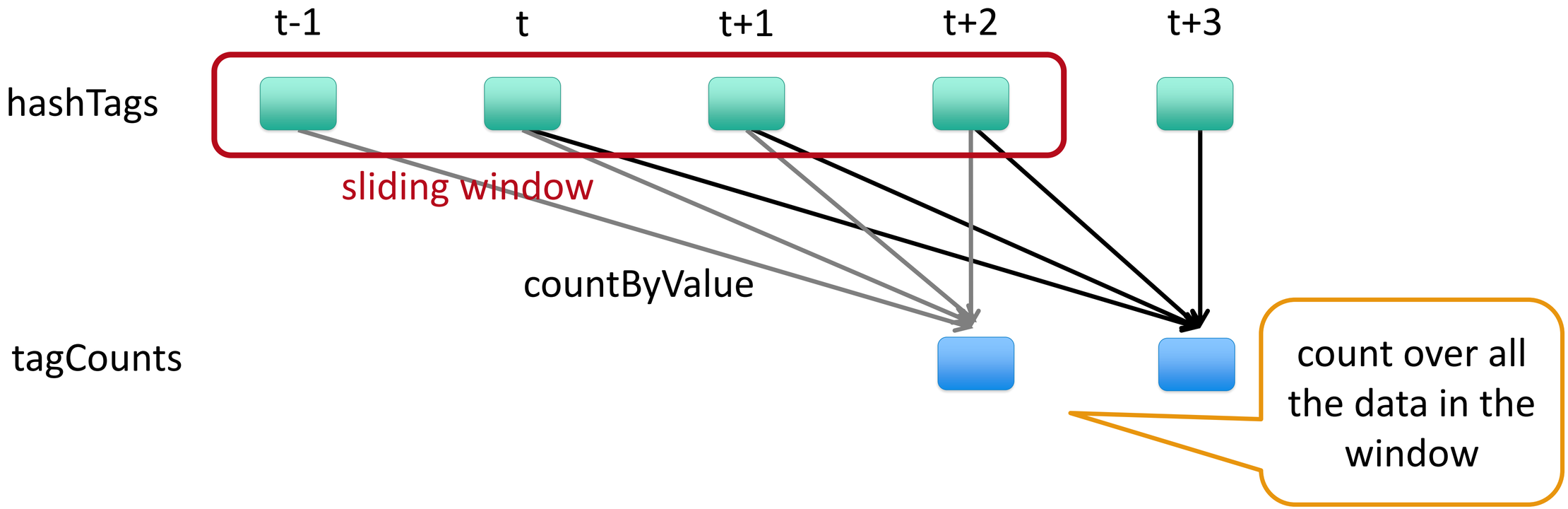
sliding window operation

window length

sliding interval

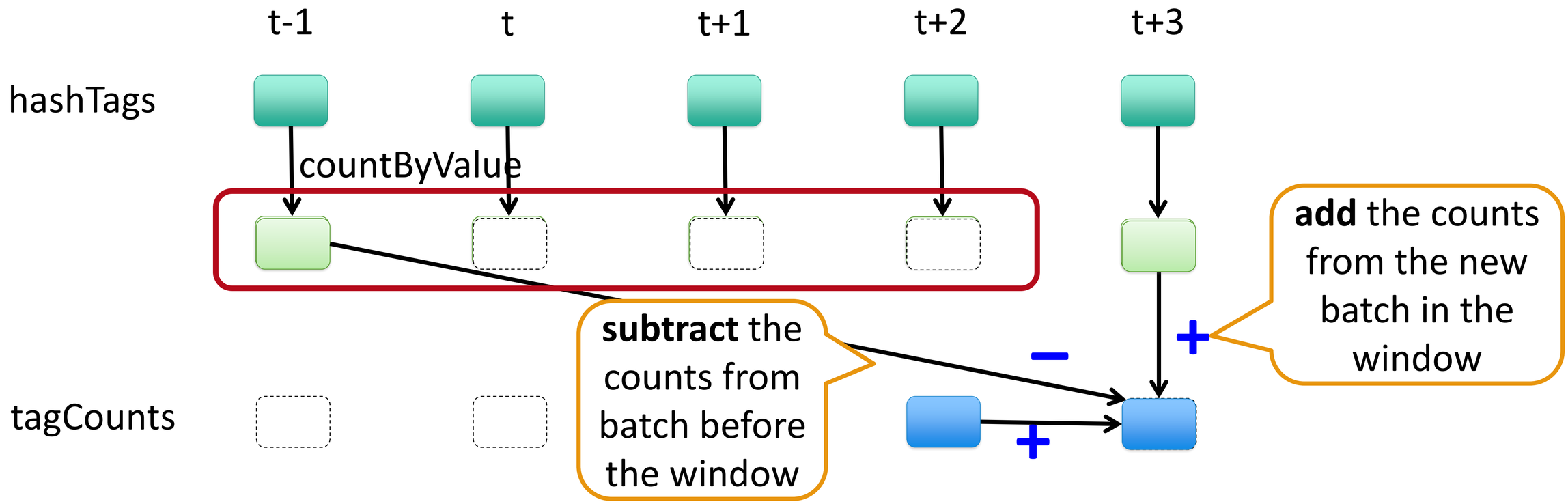
Example 3 – Counting the hashtags over last 10 mins

```
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)

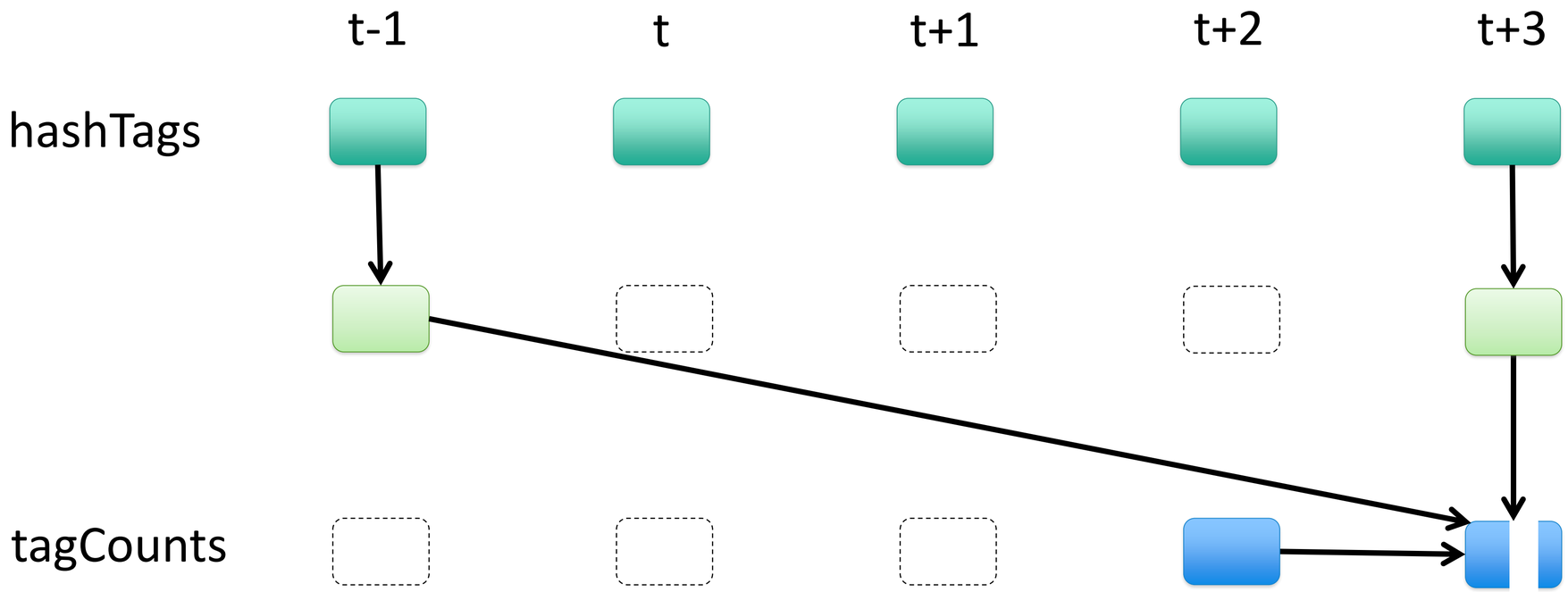
- Could have implemented counting as:

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

Demo

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))
```

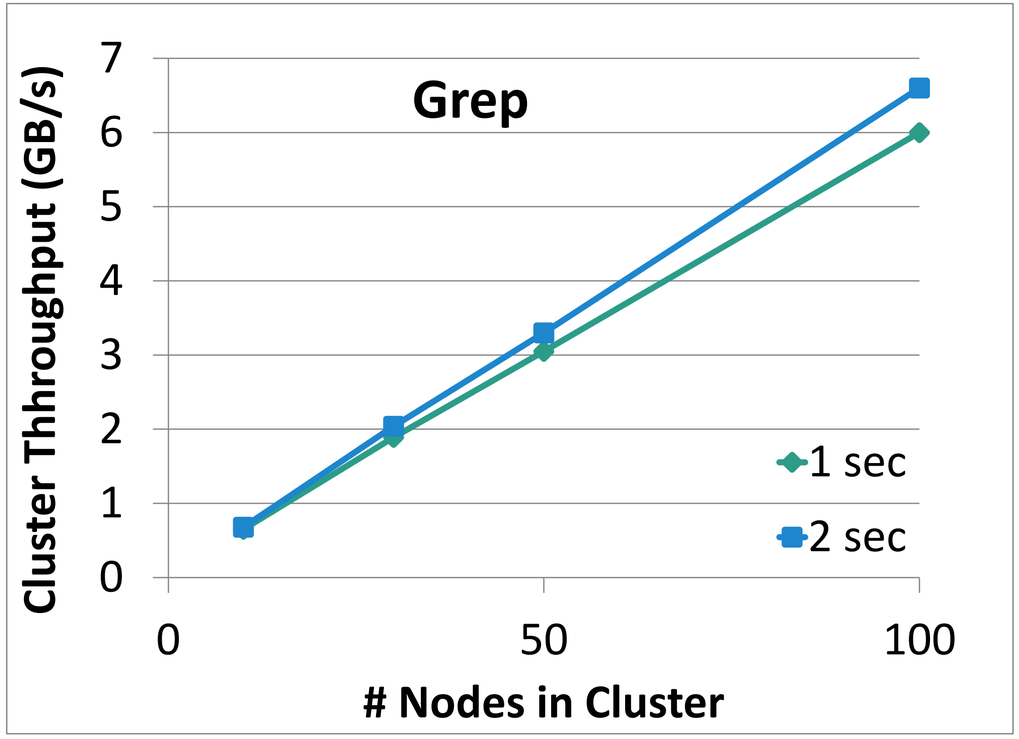
- Do arbitrary Spark RDD computation within DStream
  - Join incoming tweets with a spam file to filter out bad tweets

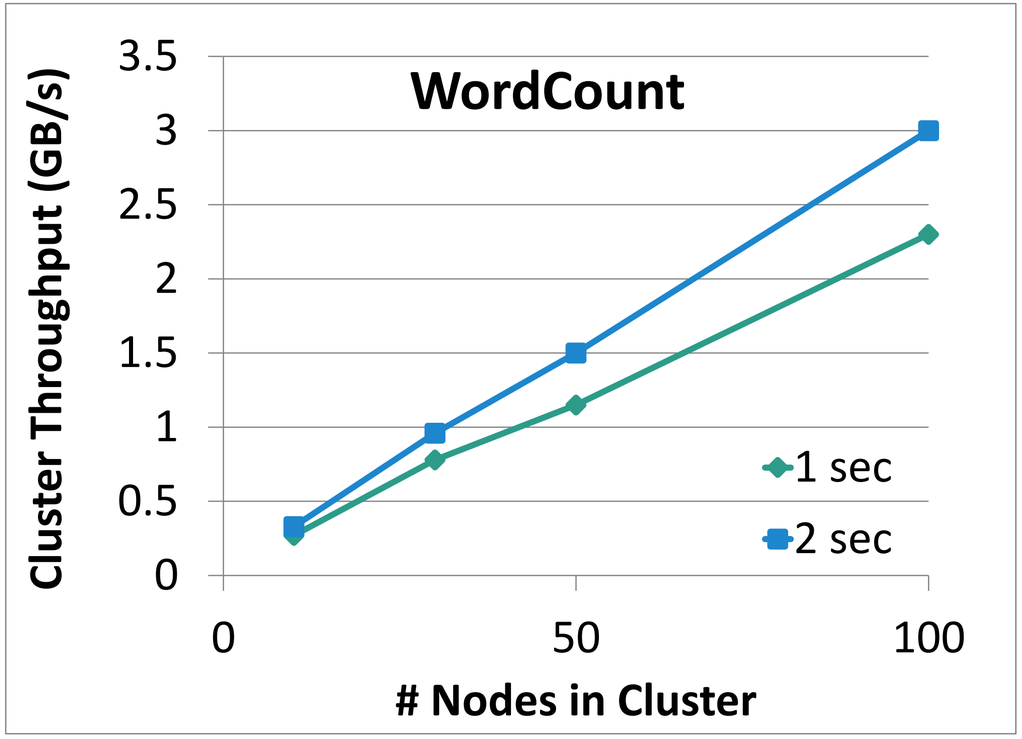
```
tweets.transform(tweetsRDD => {
   tweetsRDD.join(spamHDFSFile).filter(...)
})
```

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

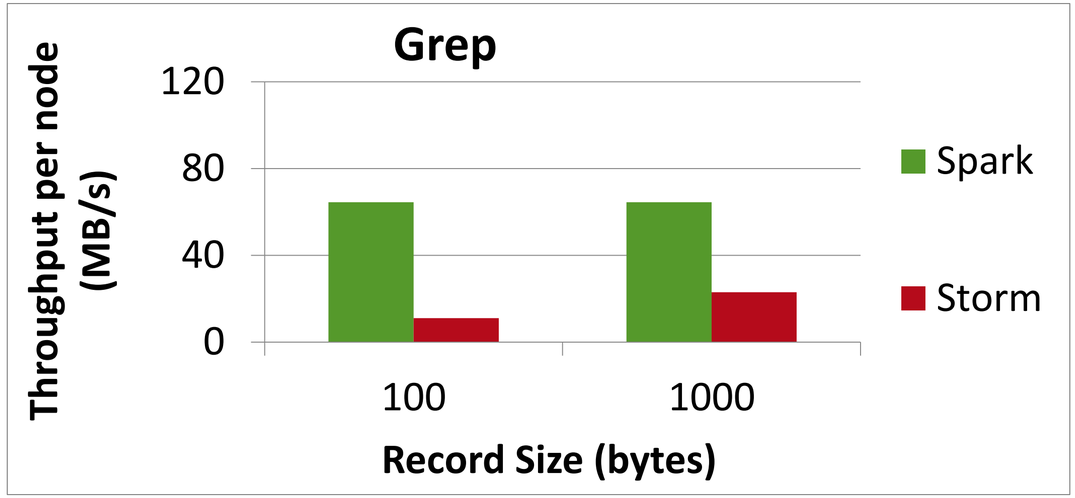


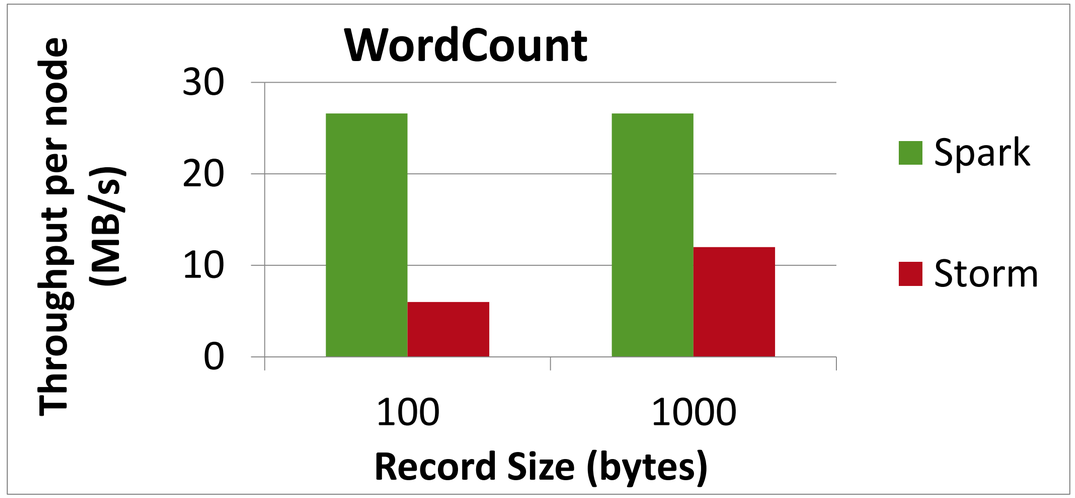


Comparison with Storm and S4

Higher throughput than Storm

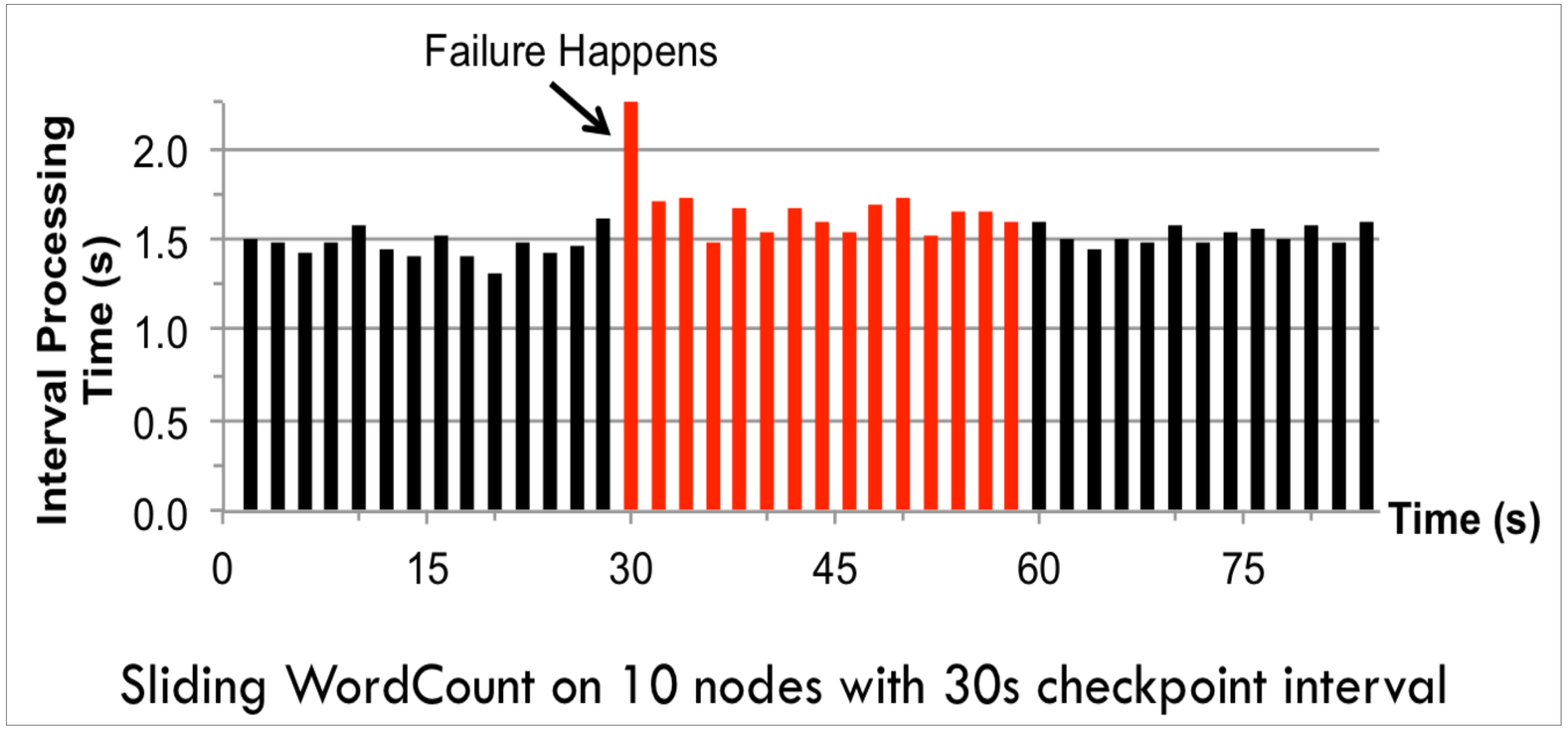
- Spark Streaming: **670k** records/second/node
- Storm: **115k** records/second/node
- Apache S4: 7.5k records/second/node





Fast Fault Recovery

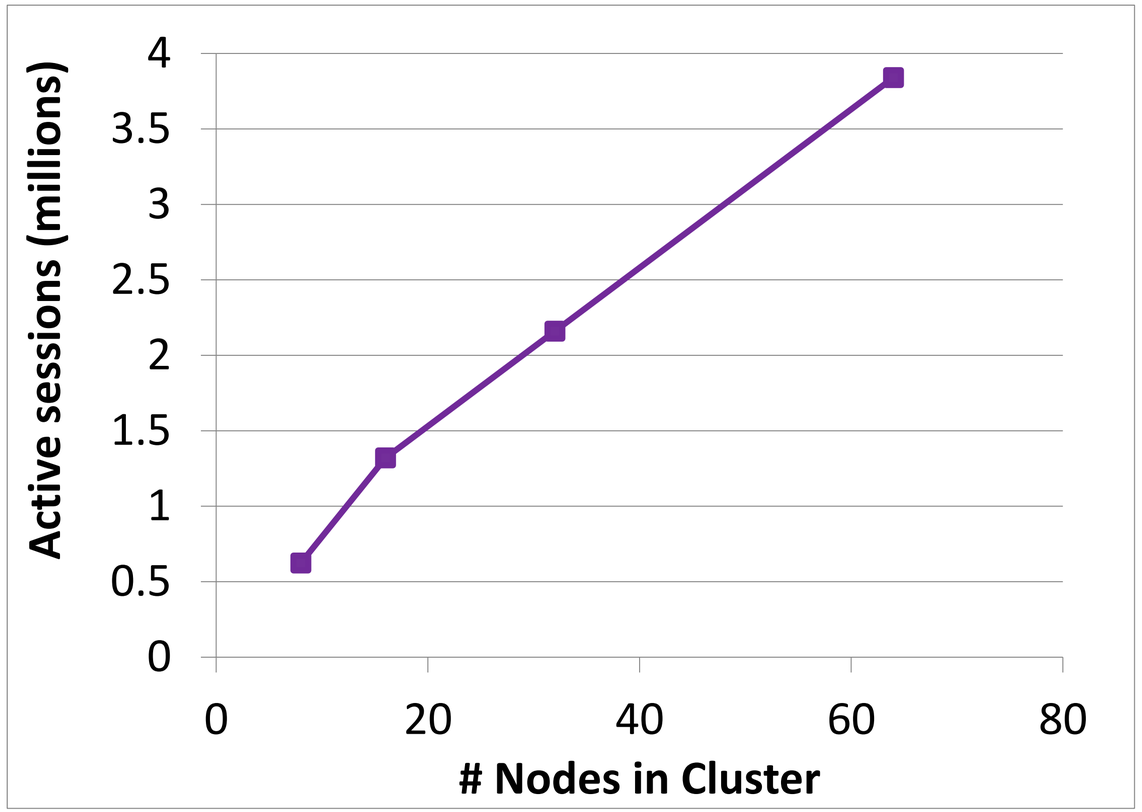
Recovers from faults/stragglers within **1 sec**



Real Applications: Conviva

Real-time monitoring of video metadata

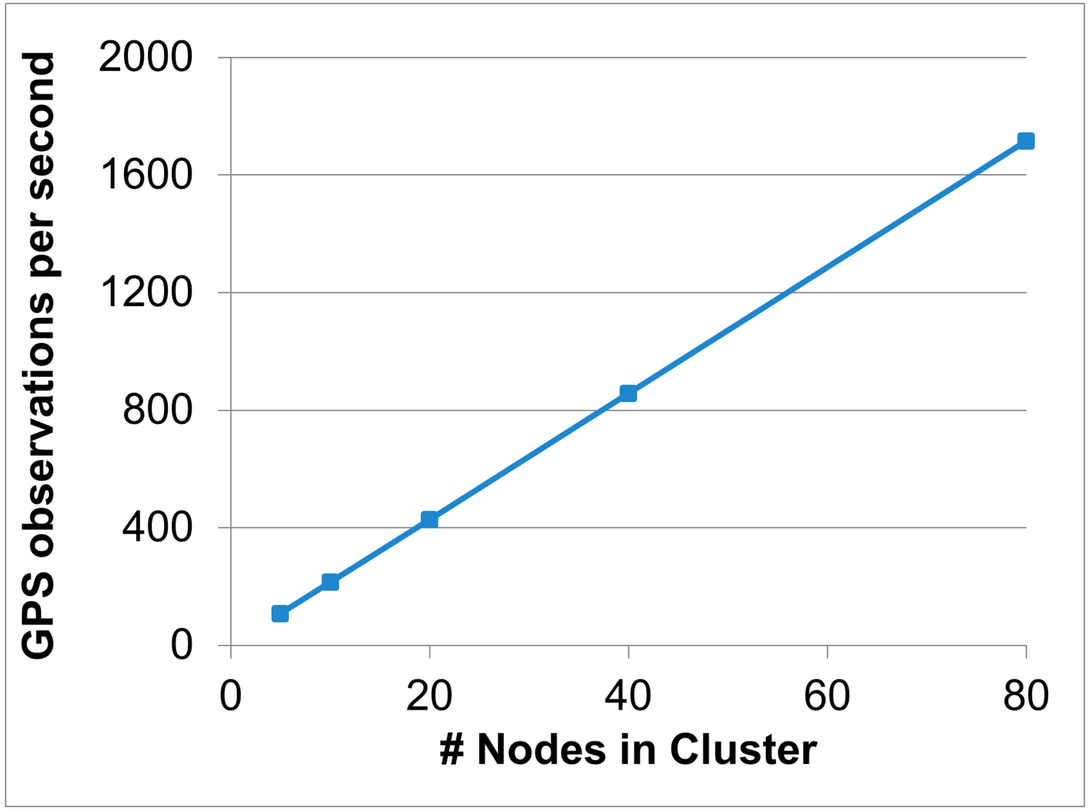
- Achieved 1-2 second latency
- Millions of video sessions processed
- Scales linearly with cluster size



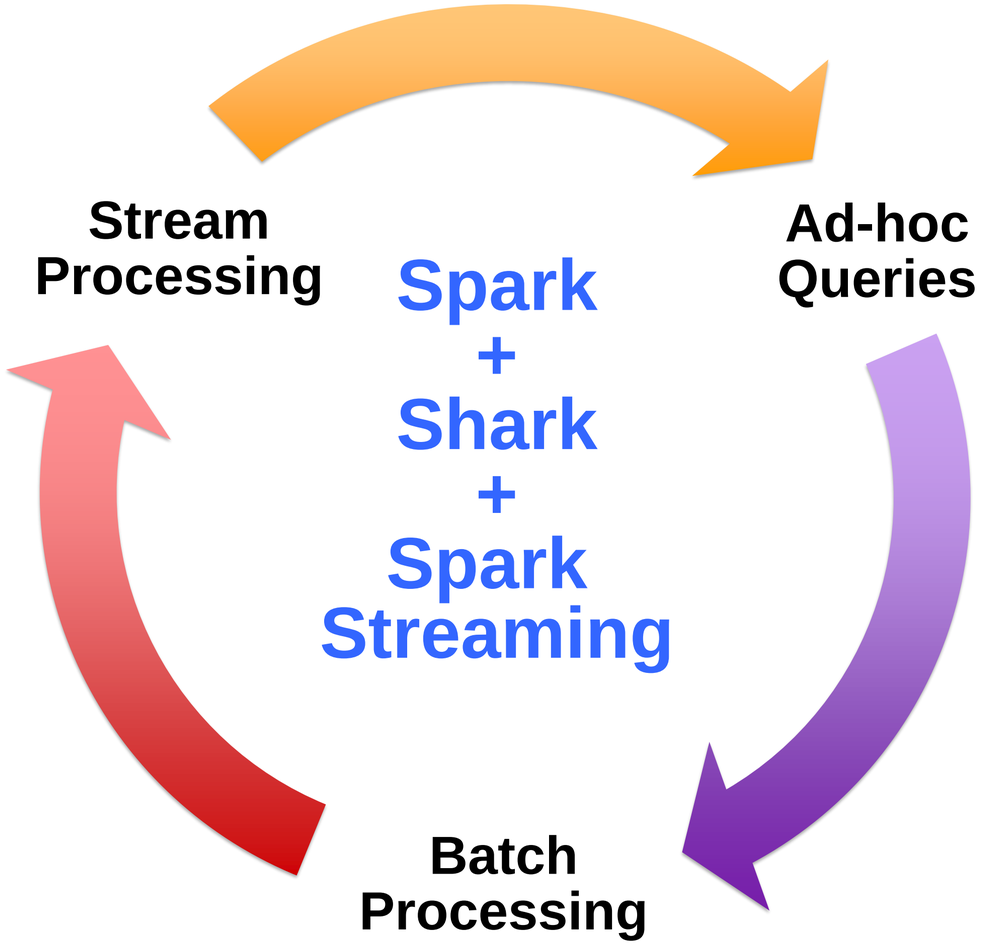
Real Applications: Mobile Millennium Project

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*



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

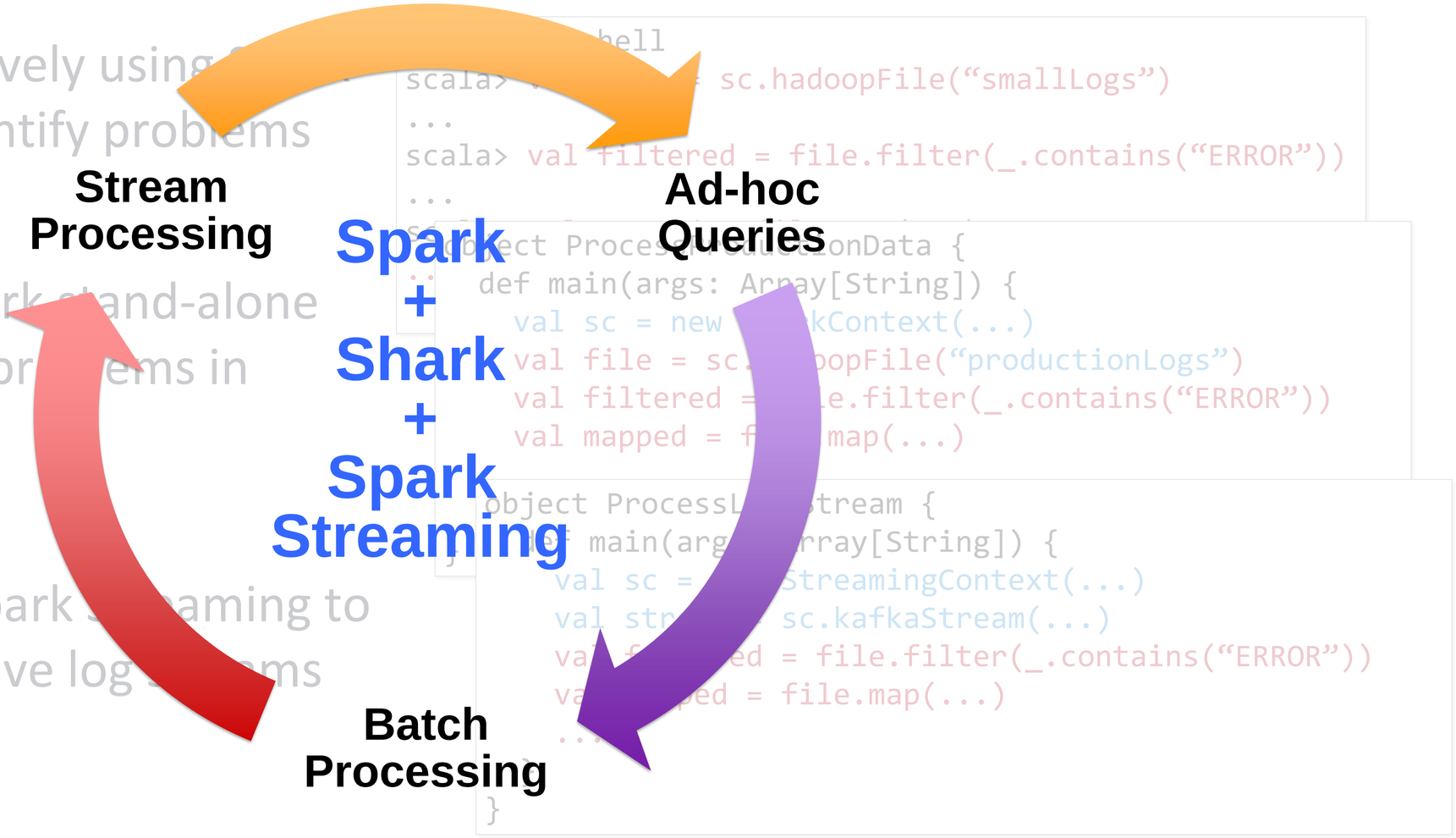
```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
```

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

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



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

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

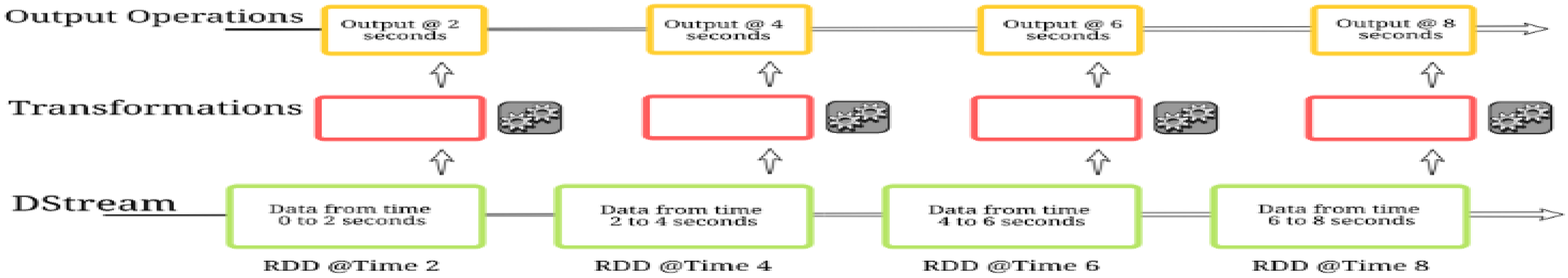
- For more information, checkout our paper: http://tinyurl.com/dstreams

Discretized Streams

Discretized Streams, or DStreams, represent a continuous stream of data. Here, either the data stream is received directly from any source or is received after we've done some processing on the original data.

The very first step of building a streaming application is to define the batch duration for the data resource from which we are collecting the data. If the batch duration is 2 seconds, then the data will be collected every 2 seconds and stored in an RDD. And the chain of continuous series of these RDDs is a DStream which is immutable and can be used as a distributed dataset by Spark.
a typical data science project. During the data pre-processing stage, we need to transform variables, including converting categorical ones into numeric, creating bins, removing the outliers and lots of other things. Spark maintains a history of all the transformations that we define on any data. So, whenever any fault occurs, it can retrace the path of transformations and regenerate the computed results again.

We want our Spark application to run 24 x 7 and whenever any fault occurs, we want it to recover as soon as possible. But while working with data at a massive scale, Spark needs to recompute all the transformations again in case of any fault. This, as you can imagine, can be quite expensive.



Caching

Here's one way to deal with this challenge. We can store the results we have calculated (cached) temporarily to maintain the results of the transformations that are defined on the data. This way, we don't have to recompute those transformations again and again when any fault occurs.

DStreams allow us to keep the streaming data in memory. This is helpful when we want to compute multiple operations on the same data.

Checkpointing

Caching is extremely helpful when we use it properly but it requires a lot of memory. And not everyone has hundreds of machines with 128 GB of RAM to cache everything.

This is where the concept of Checkpointing will help us.

Checkpointing is another technique to keep the results of the transformed dataframes. It saves the state of the running application from time to time on any reliable storage like HDFS. However, it is slower and less flexible than caching.

We can use checkpoints when we have streaming data. The transformation result depends upon previous transformation results and needs to be preserved in order to use it. We also checkpoint metadata information, like what was the configuration that was used to create the streaming data and the results of a set of DStream operations, among other things.

How to use a Machine Learning Model to Make Predictions on Streaming Data using PySpark

We'll work with a real-world dataset to detect hate speech in Tweets. to classify Tweets from other Tweets. We will use a training sample of Tweets and labels, where label '1' denotes that a Tweet is hate speech and label '0' denotes otherwise.

Setting up the Project Workflow

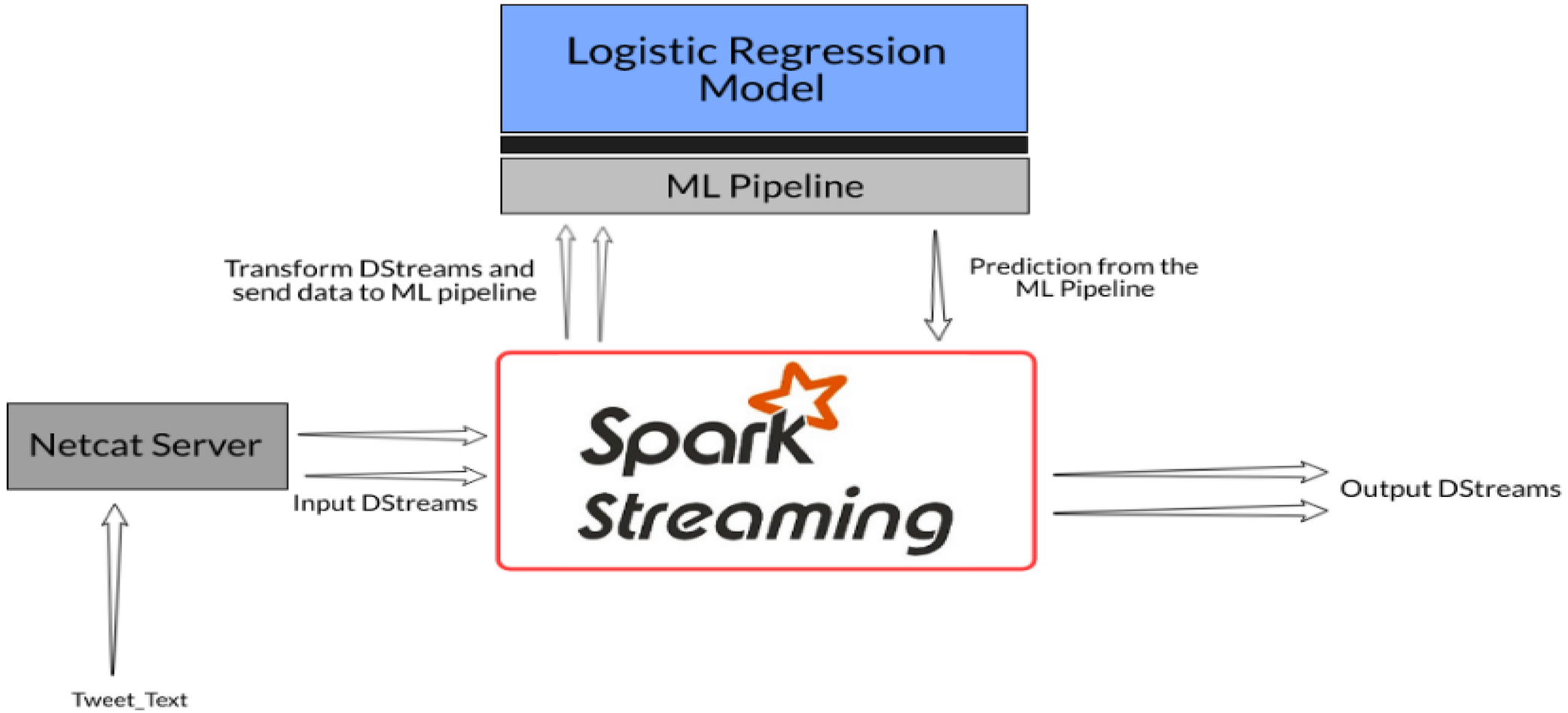
Model Building: We will build a Logistic Regression Model pipeline to classify whether the tweet contains hate speech or not. Here, our focus is not to build a very accurate classification model but to see how to use any model and return results on streaming data

Initialize Spark Streaming Context: Once the model is built, we need to define the hostname and port number from where we get the streaming data

Stream Data: Next, we will add the tweets from the netcat server from the defined port, and the Spark Streaming API will receive the data after a specified duration

Predict and Return Results: Once we receive the tweet text, we pass the data into the machine learning pipeline we created and return the predicted sentiment from the model

workflow


Logistic Regression Model
ML Pipeline
Transform DStreams and send data to ML pipeline
Prediction from the ML Pipeline
Netcat Server
Input DStreams
Spark Streaming
Output DStreams
Tweet_Text

Training the Data for Building a Logistic Regression Model

We have data about Tweets in a CSV file mapped to a label.
We will use a logistic regression model to predict whether the tweet contains hate speech or not. If yes, then our model will predict the label as 1 (else 0).

First, we need to define the schema of the CSV file. Otherwise, Spark will consider the data type of each column as string. Read the data and check if the schema is as defined or not:

```
+---+-----+--------------------+
| id|label|               tweet|
+---+-----+--------------------+
|  1|    0| @user when a fat...|
|  2|    0|@user @user thank...|
|  3|    0|  bihday your maj...|
|  4|    0|#model   i love u...|
|  5|    0| factsguide: soci...|
+---+-----+--------------------+
only showing top 5 rows
```

```
root
 |-- id: integer (nullable = true)
 |-- label: integer (nullable = true)
 |-- tweet: string (nullable = true)
```

```
# importing required libraries
from pyspark import SparkContext
from pyspark.sql.session import SparkSession
from pyspark.streaming import StreamingContext
import pyspark.sql.types as tp
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer, OneHotEncoderEstimator, VectorAssembler
from pyspark.ml.feature import StopWordsRemover, Word2Vec, RegexTokenizer
from pyspark.ml.classification import LogisticRegression
from pyspark.sql import Row
# initializing spark session
sc = SparkContext(appName="PySparkShell")
spark = SparkSession(sc)
# define the schema
my_schema = tp.StructType([
  tp.StructField(name= 'id',          dataType= tp.IntegerType(),  nullable= True),
  tp.StructField(name= 'label',       dataType= tp.IntegerType(),  nullable= True),
  tp.StructField(name= 'tweet',       dataType= tp.StringType(),   nullable= True)
])
# read the dataset
my_data = spark.read.csv('twitter_sentiments.csv',
                         schema=my_schema,
                         header=True)
# view the data
my_data.show(5)
# print the schema of the file
my_data.printSchema()
```

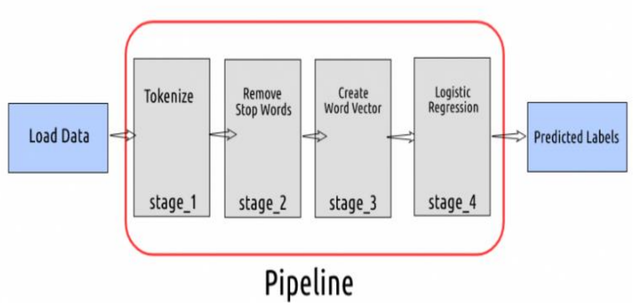
Defining the Stages of our Machine Learning Pipeline

Now that we have the data in a Spark dataframe, we need to define the different stages in which we want to transform the data and then use it to get the predicted label from our model.

In the first stage, we will use the *RegexTokenizer* to convert Tweet text into a list of words.

Then, we will remove the stop words from the word list and create word vectors.

In the final stage, we will use these word vectors to build a logistic regression model and get the predicted sentiments.




```
. # define stage 1: tokenize the tweet text
stage_1 = RegexTokenizer(inputCol= 'tweet' , outputCol= 'tokens', pattern= '\\W')
# define stage 2: remove the stop words
stage_2 = StopWordsRemover(inputCol= 'tokens', outputCol= 'filtered_words')
# define stage 3: create a word vector of the size 100
stage_3 = Word2Vec(inputCol= 'filtered_words', outputCol= 'vector', vectorSize= 100)
# define stage 4: Logistic Regression Model
model = LogisticRegression(featuresCol= 'vector', labelCol= 'label'
```

Setup our Machine Learning Pipeline

Let’s add the stages in the Pipeline object and we will then perform these transformations in order. Fit the pipeline with the training dataset and now, whenever we have a new Tweet, we just need to pass that through the pipeline object and transform the data to get the predictions:

```
# setup the pipeline
pipeline = Pipeline(stages= [stage_1, stage_2, stage_3, model])

# fit the pipeline model with the training data
pipelineFit = pipeline.fit(my_data)
```

Stream Data and Return Results

Let’s say we receive hundreds of comments per second and we want to keep the platform clean by blocking the users who post comments that contain hate speech. So, whenever we receive the new text, we will pass that into the pipeline and get the predicted sentiment.

We will define a function get_prediction which will remove the blank sentences and create a dataframe where each row contains a Tweet.

So, initialize the Spark Streaming context and define a batch duration of 3 seconds. This means that we will do predictions on data that we receive every 3 seconds:

```
# define a function to compute sentiments of the received tweets
def get_prediction(tweet_text):
        try:
    # filter the tweets whose length is greater than 0
            tweet_text = tweet_text.filter(lambda x: len(x) > 0)
    # create a dataframe with column name 'tweet' and each row will contain the tweet
            rowRdd = tweet_text.map(lambda w: Row(tweet=w))
    # create a spark dataframe
            wordsDataFrame = spark.createDataFrame(rowRdd)
    # transform the data using the pipeline and get the predicted sentiment
            pipelineFit.transform(wordsDataFrame).select('tweet','prediction').show()
        except :
            print('No data')
# initialize the streaming context
ssc = StreamingContext(sc, batchDuration= 3)

# Create a DStream that will connect to hostname:port, like localhost:9991
lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2]))
# split the tweet text by a keyword 'TWEET_APP' so that we can identify which set of words is from a single tweet
words = lines.flatMap(lambda line : line.split('TWEET_APP'))

# get the predicted sentiments for the tweets received
words.foreachRDD(get_prediction)
# Start the computation
ssc.start()
# Wait for the computation to terminate
ssc.awaitTermination()
```