DATABRICKS

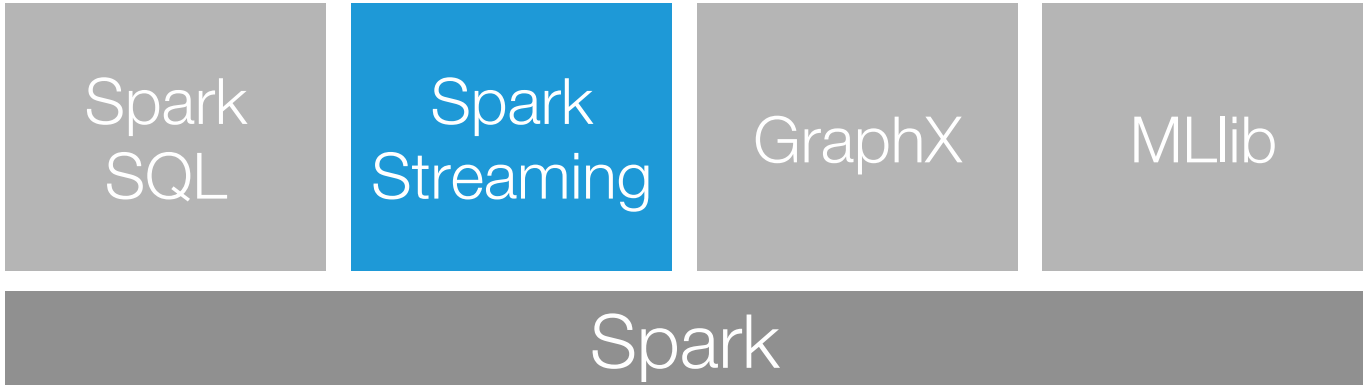
Spark Streaming

Tathagata Das (TD)

Whoami

- Core committer on Apache Spark
- Lead developer on Spark Streaming
- On leave from PhD program in UC Berkeley

What is Spark Streaming?



- Extends Spark for doing big data stream processing
- Started in 2012, alpha released in 2013 with Spark 0.7, graduated in early 2014 with Spark 0.9

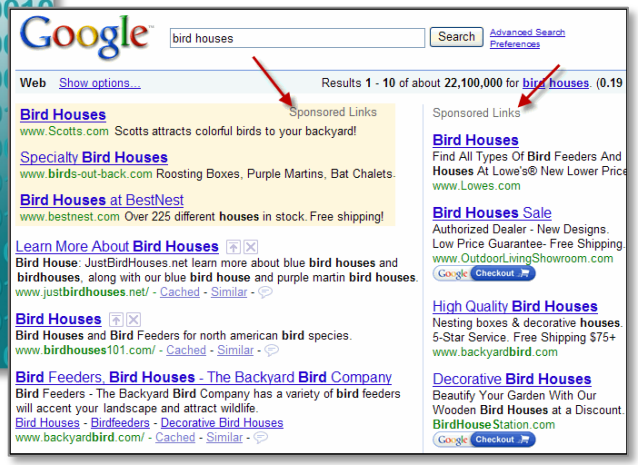
Why Spark Streaming?

Many big-data applications need to process large data streams in realtime

Website monitoring

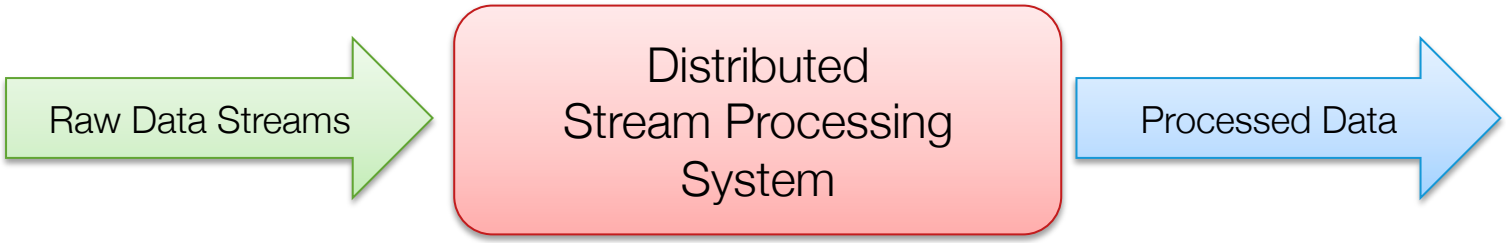
Fraud detection

Ad monetization



DATABRICKS

Why Spark Streaming?



Scales to hundreds of nodes

Achieves low latency

Efficiently recover from failures

Integrates with batch and interactive processing

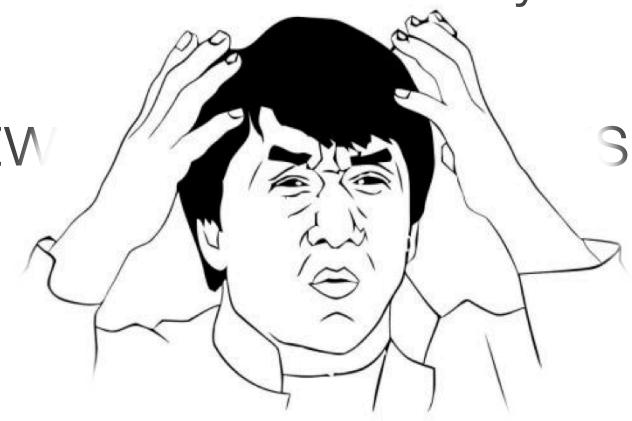
DATABRICKS

Integration with Batch Processing

- Many environments require processing same data in live streaming as well as batch post-processing

- 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

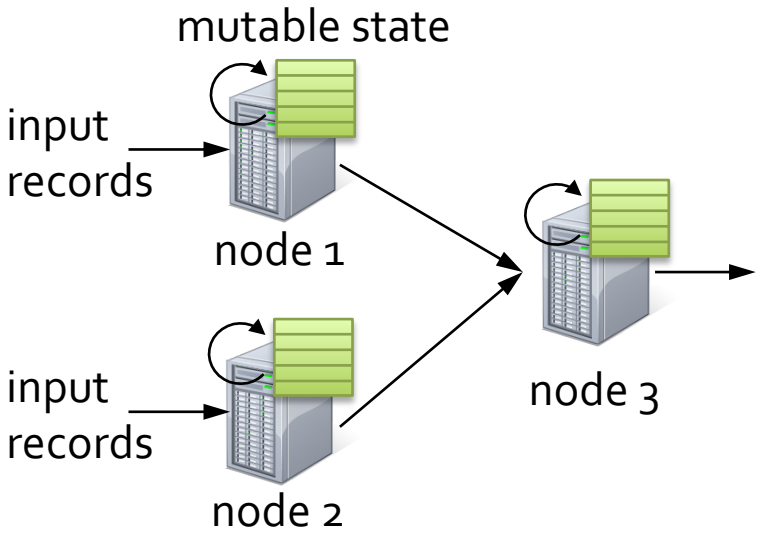
- Extremely painful to maintain tw s
  - Different programming models
  - Doubles implementation effort



DATABRICKS

Fault-tolerant Stream Processing

- Traditional processing model
  - Pipeline 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!

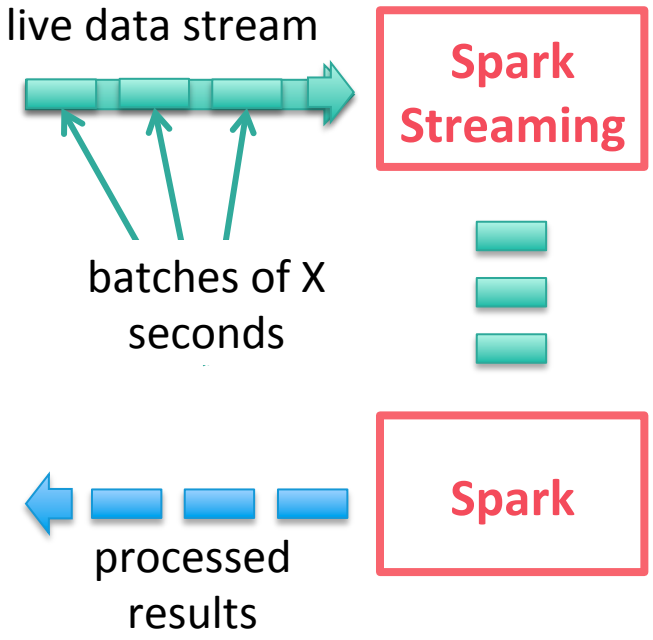
- Trident – Use transactions to update state
  - Processes each record *exactly once*
  - Per-state transaction to external database is slow

Spark Streaming

Spark Streaming

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

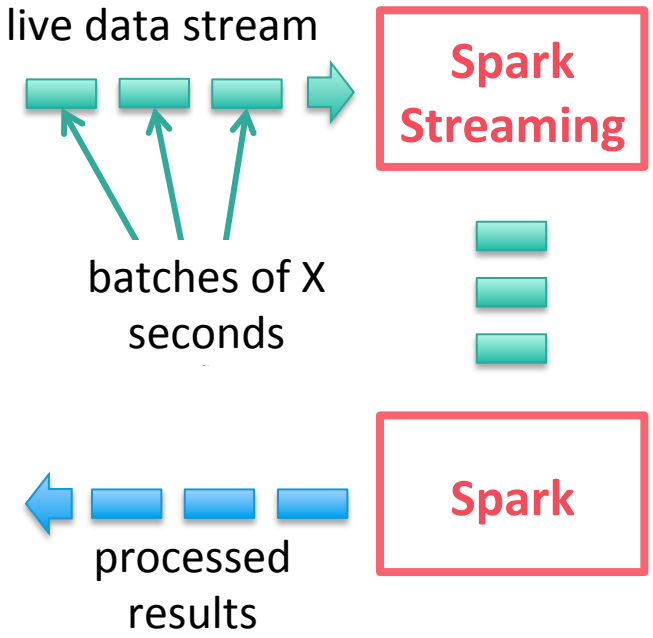
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Spark Streaming

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

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



Programming Model - DStream

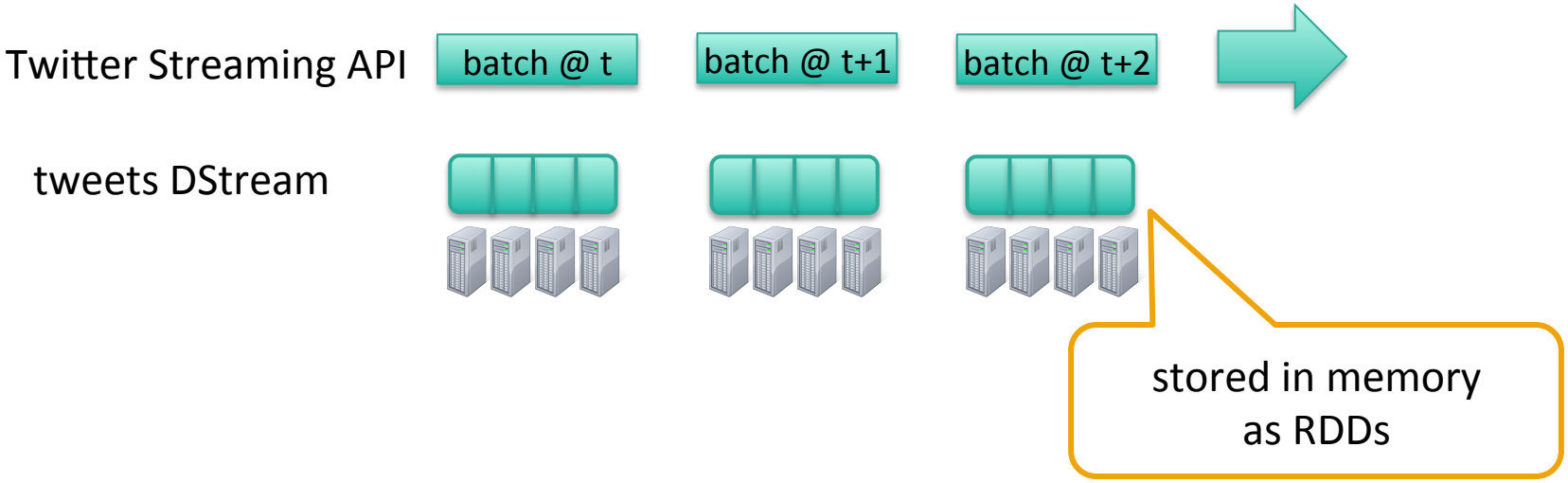
- Discretized Stream (DStream)
  - Represents a stream of data
  - Implemented as a sequence of RDDs

- DStreams can be either…
  - Created from streaming input sources
  - Created by applying transformations on existing DStreams

Example – Get hashtags from Twitter

```
val ssc = new StreamingContext(sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream(ssc, auth)
```

Input DStream

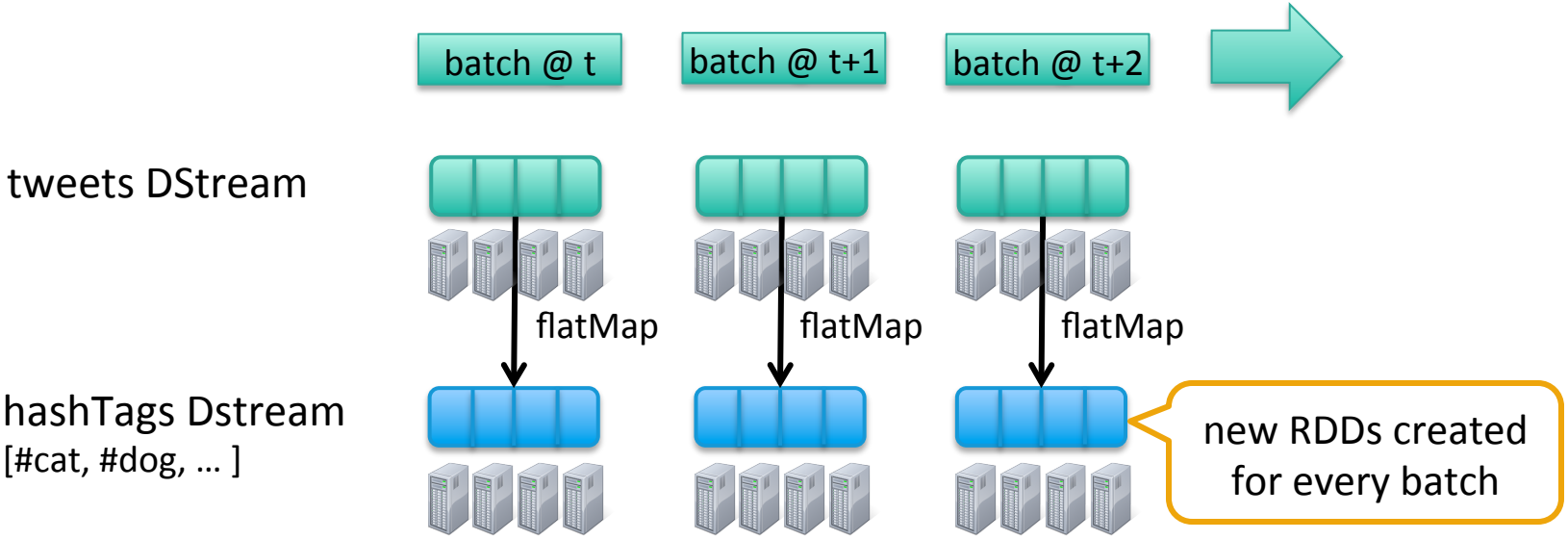


Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
```

transformed DStream

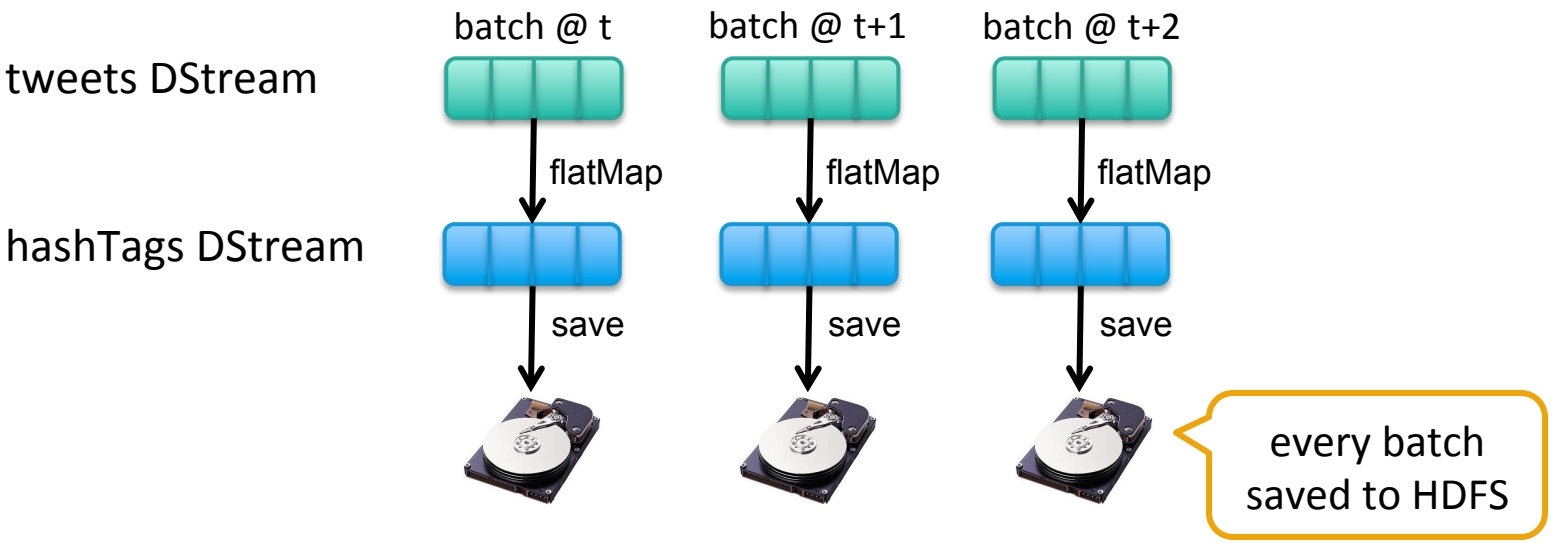
transformation: modify data in one DStream to create another DStream



Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

output operation: to push data to external storage

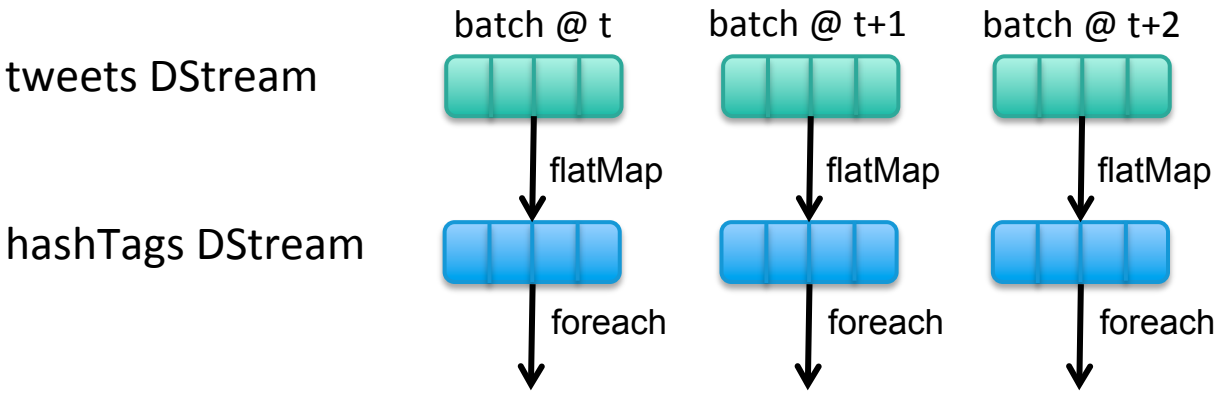


DATABRICKS

Example – Get hashtags from Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.foreachRDD(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data



Write to a database, update analytics UI, do whatever you want

Languages

Scala API

```
val tweets = TwitterUtils.createStream(ssc, auth)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

Java API


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

Python API

```
...soon
```

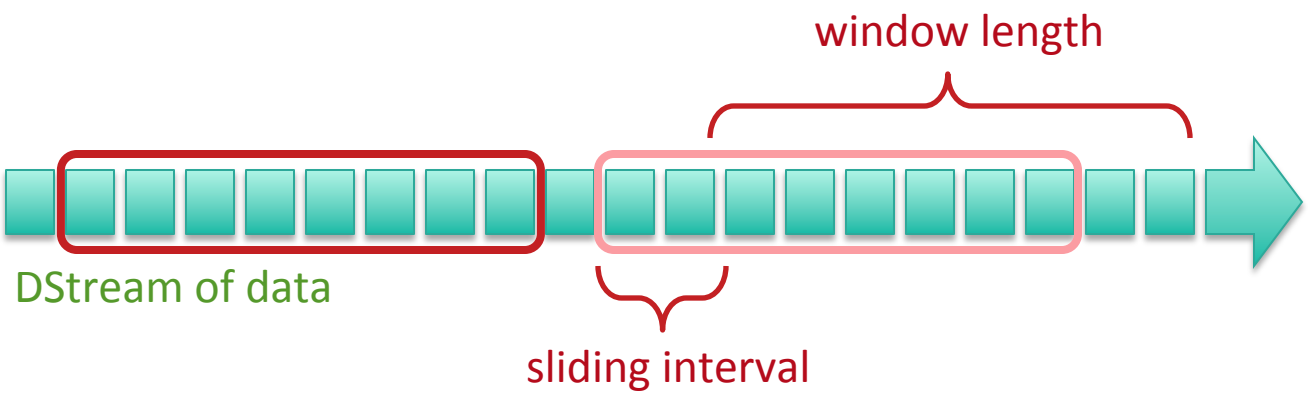
Window-based Transformations

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

sliding window operation

window length

sliding interval



DATABRICKS

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

```
def updateMood(newTweets, lastMood) => newMood

val moods = tweetsByUser.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(spamFile).filter(...)
})
```

DStreams + RDDs = Power

- Combine live data streams with historical data
  - Generate historical data models with Spark, etc.
  - Use data models to process live data stream

- Combine streaming with MLlib, GraphX algos
  - Offline learning, online prediction
  - Online learning and prediction

- Query streaming data using SQL
  - select * from table_from_streaming_data

Databricks Keynote Demo

```
val ssc = new StreamingContext(sc, Seconds(5))
val sqlContext = new SQLContext(sc)
val tweets = TwitterUtils.createStream(ssc, auth)
val transformed = tweets.filter(isEnglish).window(Minutes(1))

transformed.foreachRDD { rdd =>
  // Tweet is a case class containing necessary
  rdd.map(Tweet.apply(_)).registerAsTable("tweets")
}
```

```
SELECT text FROM tweets WHERE similarity(tweet) > 0.01
SELECT getClosestCountry(lat, long) FROM tweets
```

Advantage of an Unified Stack

- Explore data interactively to identify problems
- Use same code in Spark for processing large logs
- Use similar code in Spark Streaming for realtime processing

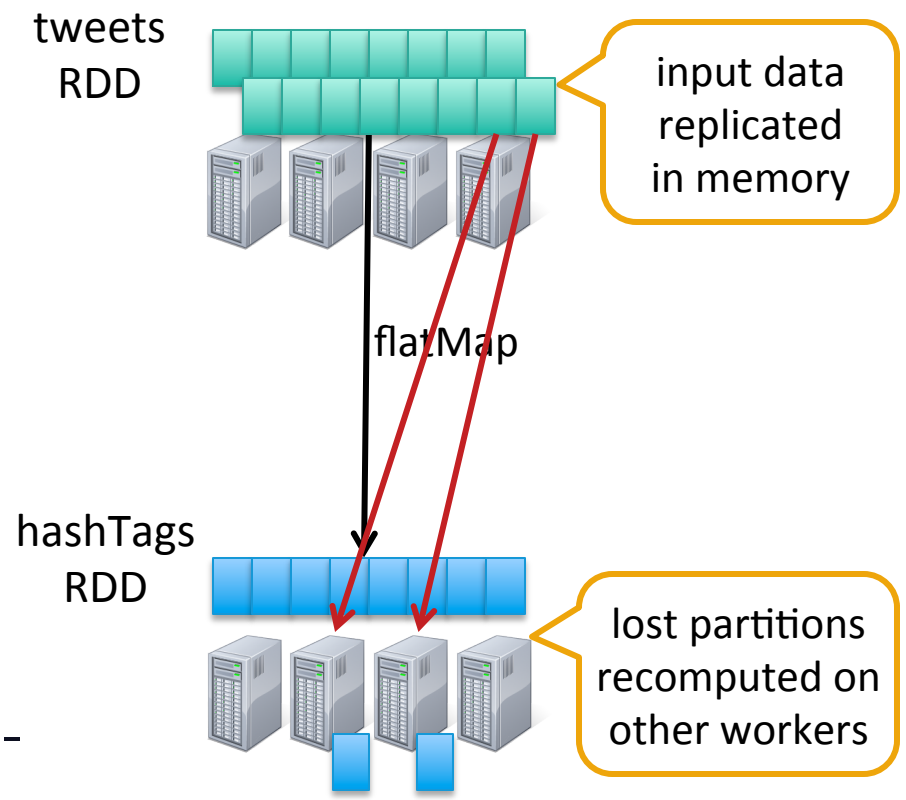
```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = filtered.map(...)
```

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

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

Fault-tolerance

- 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 transformations are fault-tolerant, and *exactly-once* transformations



Input Sources

- Out of the box, we provide
  - Kafka, Flume, Akka Actors, Raw TCP sockets, HDFS, etc.
- Very easy to write a custom *receiver*
  - Define what to when receiver is started and stopped
- Also, generate your own sequence of RDDs, etc. and push them in as a “stream”

Future Directions!

- Better integration with SQL, MLlib, GraphX
- Improved integration with different sources
- Improved control over data rates
- Python API

Today's Tutorial

- Process Twitter data stream to find most popular hashtags over a window
- Requires a Twitter account
  - Need to setup Twitter authentication to access tweets
  - All the instructions are in the tutorial
- Your account will be safe!
  - No need to enter your password anywhere, only the keys
  - Destroy the keys after the tutorial is done

Conclusion

Spark 1.0.0 Overview Programming Guides API Docs Deploying More

Spark Streaming Programming Guide

Overview

Spark Streaming is an extension of the core Spark API that allows enables high-throughput, fault-tolerant stream processing of live data streams. Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ or plain old TCP sockets and be processed using complex algorithms expressed with high-level functions like `map`, `reduce`, `join` and `window`. Finally, processed data can be pushed out to filesystems, databases, and live dashboards. In fact, you can apply Spark's in-built machine learning algorithms, and graph processing algorithms on data streams.

DATABRICKS