Big Data

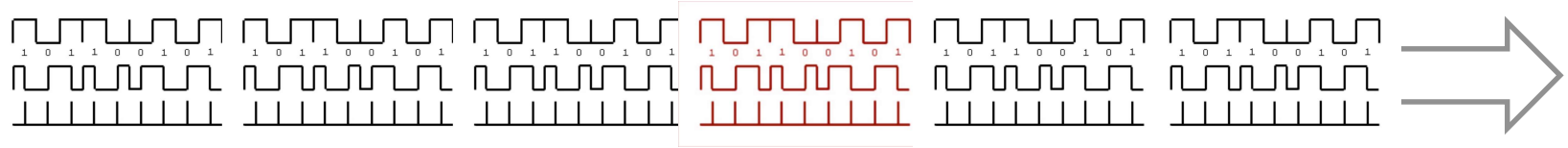
Big *Streaming* Data

DATABRICKS

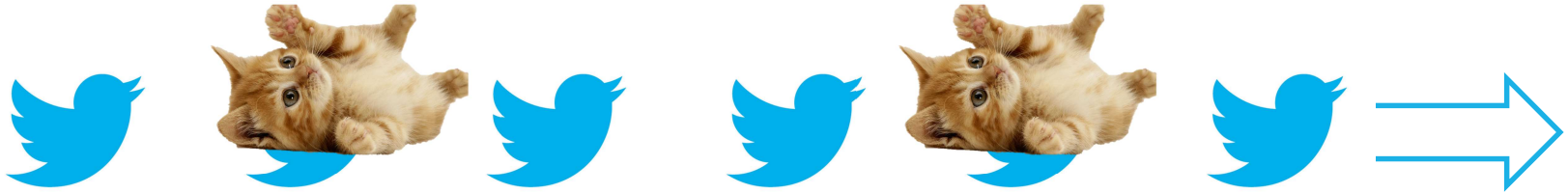
Big *Streaming* Data Processing

Fraud detection in bank transactions

Anomalies in sensor data



Cat videos in tweets



DATABRICKS

How to Process Big *Streaming* Data

Scales to hundreds of nodes

Achieves low latency

Efficiently recover from failures

Integrates with batch and interactive processing

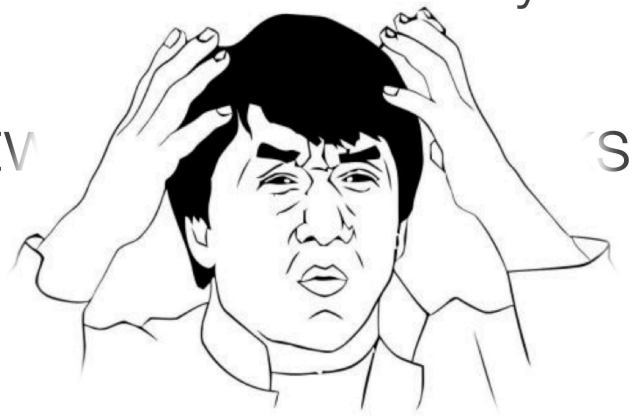
DATABRICKS

What people have been doing?

> Build two stacks – one for batch, one for streaming
  - Often both process same data

> 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
  - Doubles operational effort

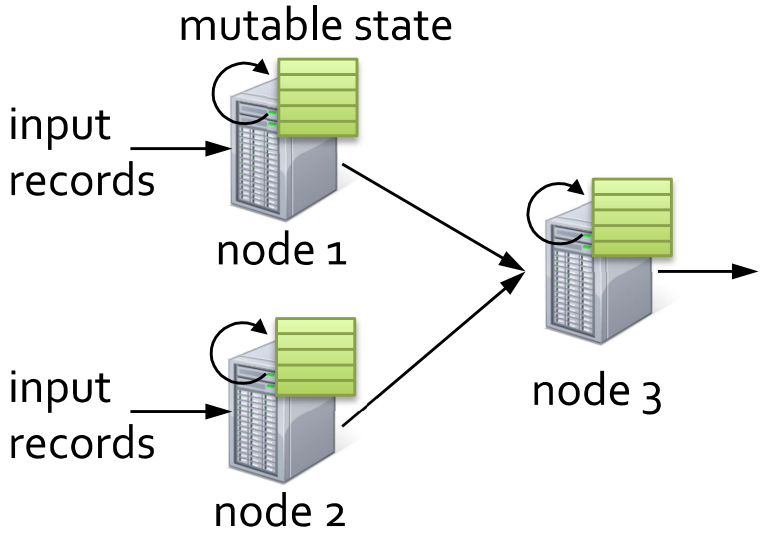


DATABRICKS

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

Streaming

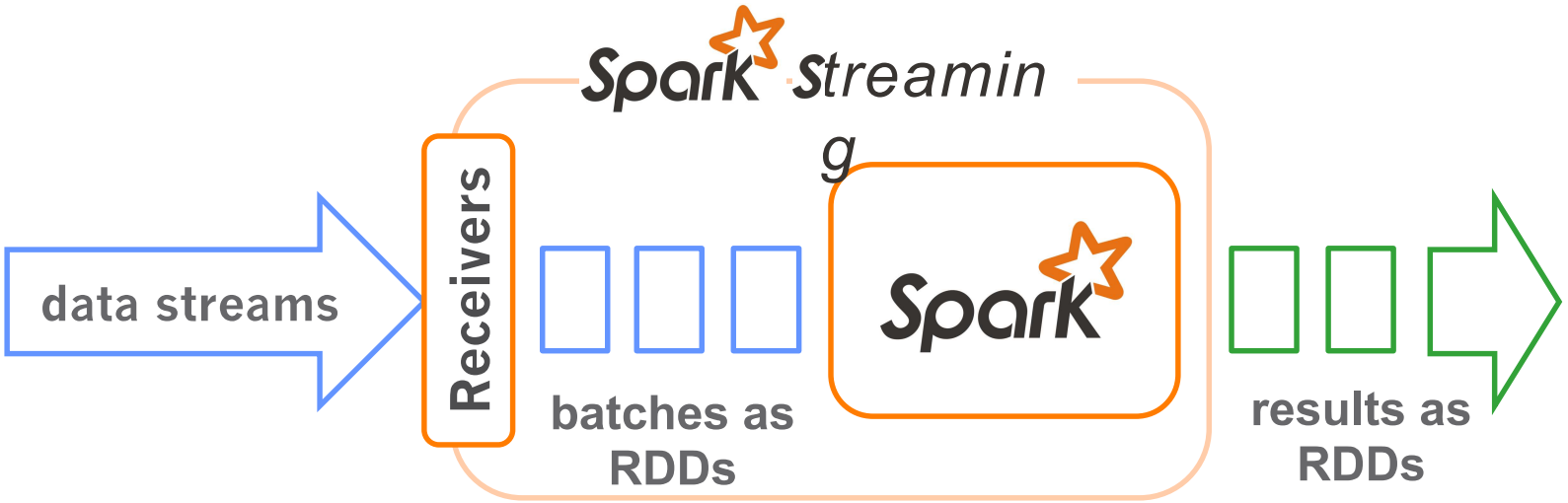
DATABRICKS

What is Spark Streaming?

- Receive data streams from input sources, process them in a cluster, push out to databases/ dashboards
- Scalable, fault-tolerant, second-scale latencies

How does Spark Streaming work?

- Chop up data streams into batches of few secs
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Processed results are pushed out in batches



DATABRICKS

Spark Streaming Programming Model

> *Discretized Stream (DStream)*

- Represents a stream of data
- Implemented as a sequence of RDDs

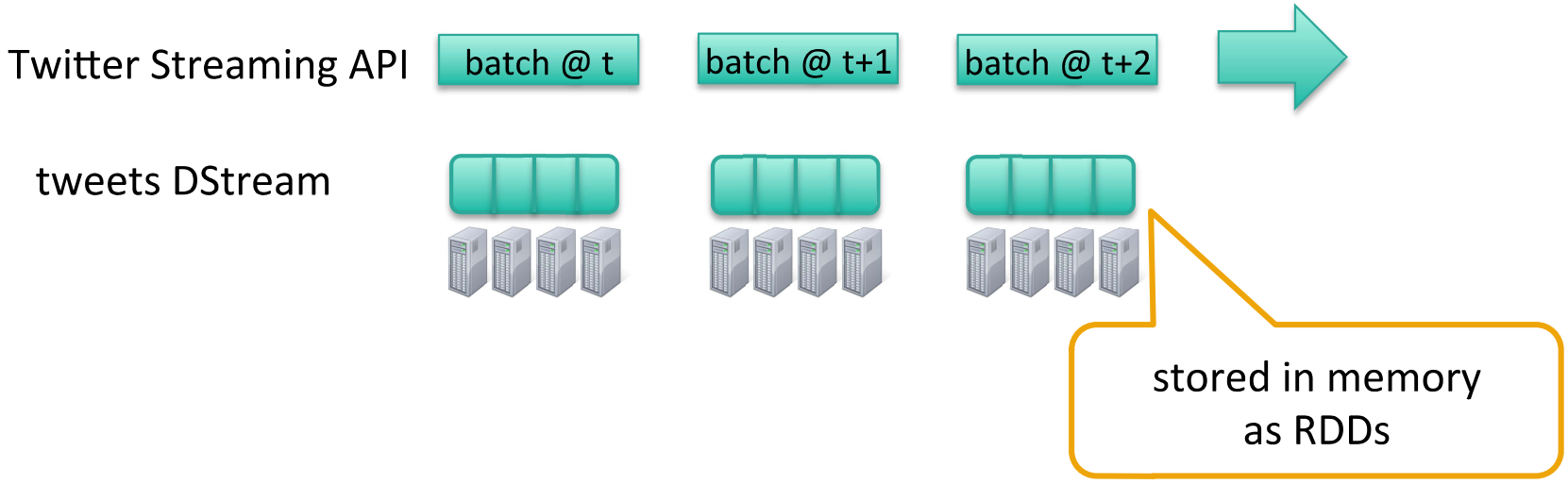
> DStreams API very similar to RDD API

- Functional APIs in Scala, Java
- Create input DStreams from different sources
- Apply parallel operations

Example – Get hashtags from Twitter

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

Input DStream



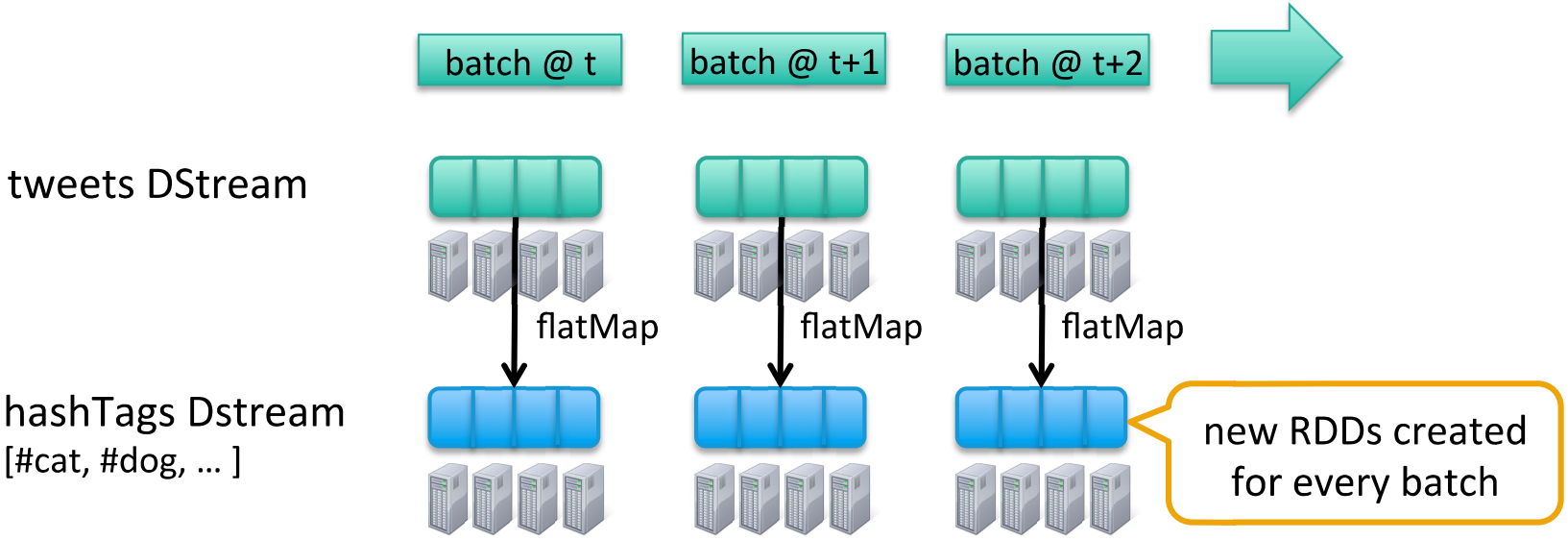
DATABRICKS

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

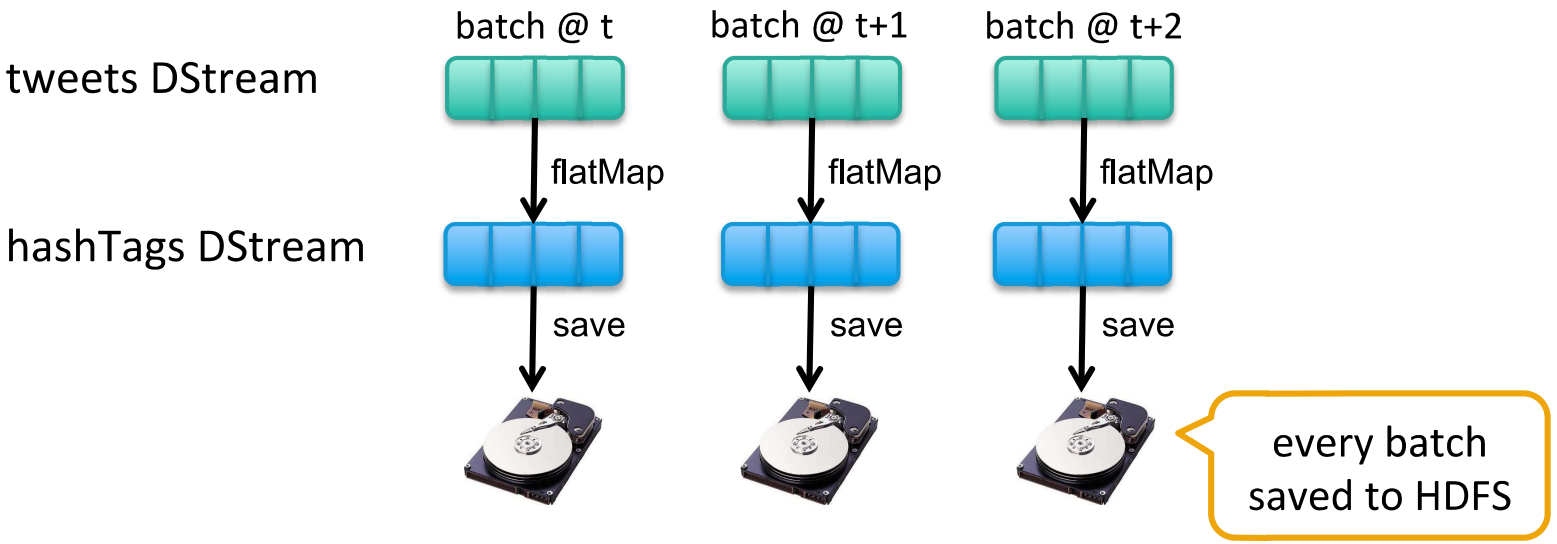


DATABRICKS

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

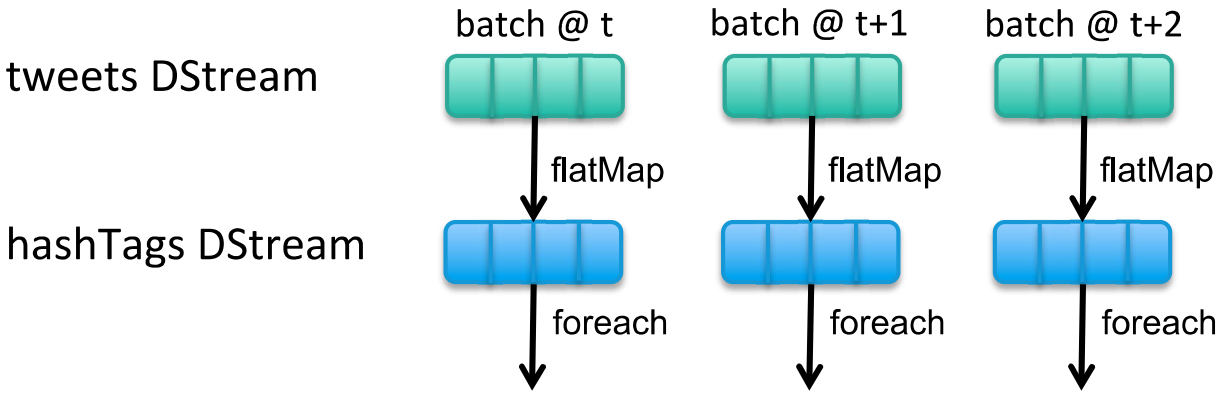


DATABRICKS

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

DATABRICKS

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
```

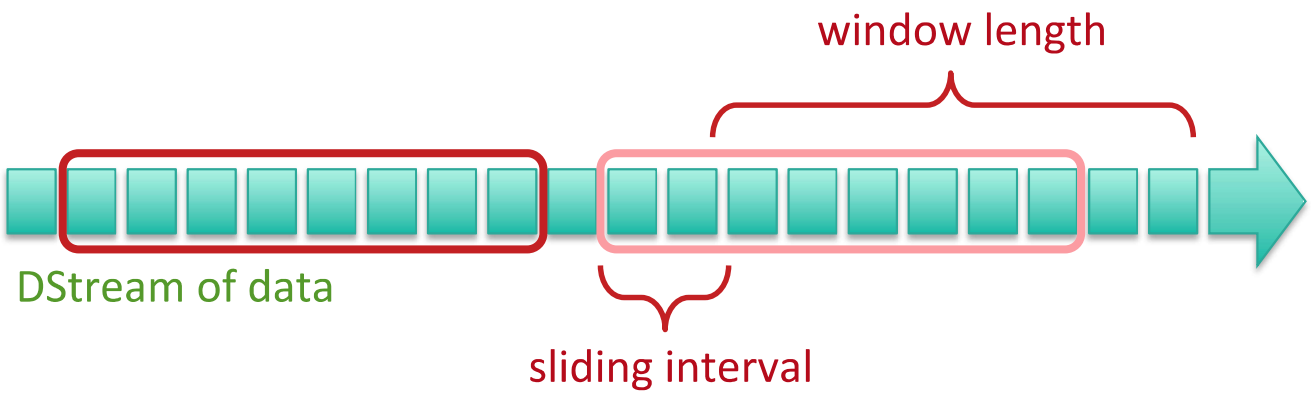
Window-based Transformations

```
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!

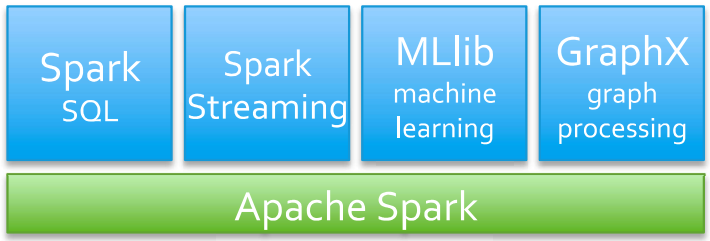
- Example: Join incoming tweets with a spam HDFS file to filter out bad tweets

```
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



- Interactively query streaming data using SQL
  - select * from table_from_streaming_data

DATABRICKS

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(...)
...
```

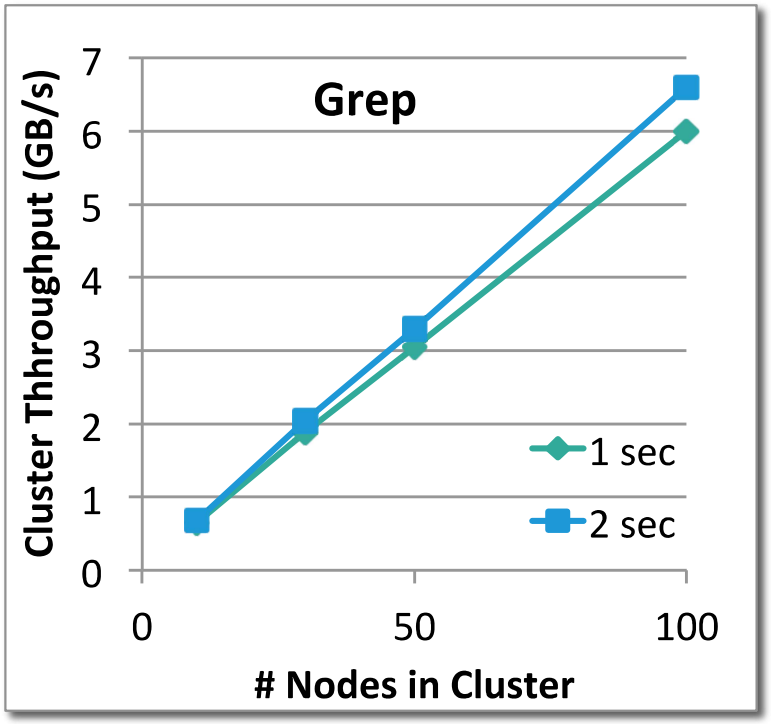
```
object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
    ...
  }
}
```

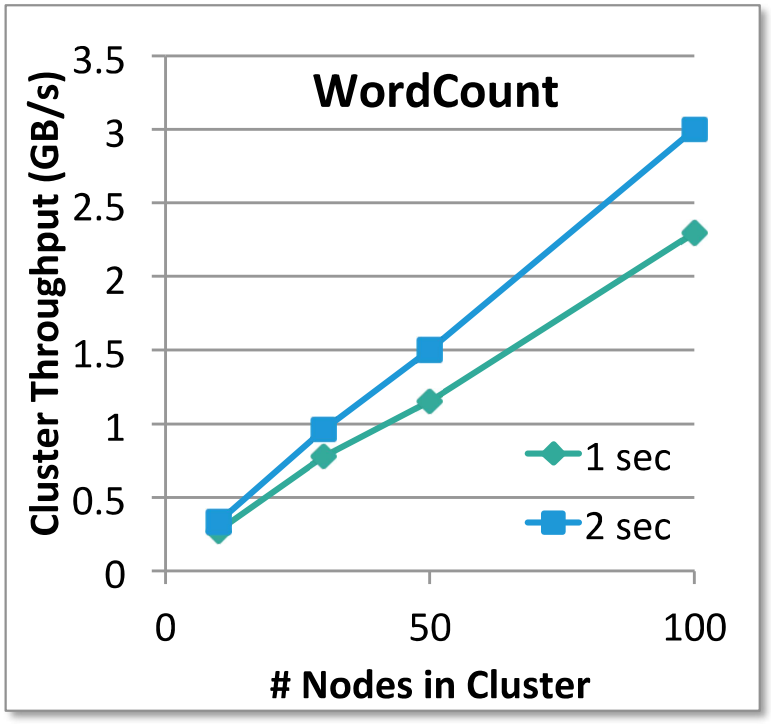
```
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = KafkaUtil.createStream(...)
    val filtered = stream.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
    ...
  }
}
```

DATABRICKS

Performance

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

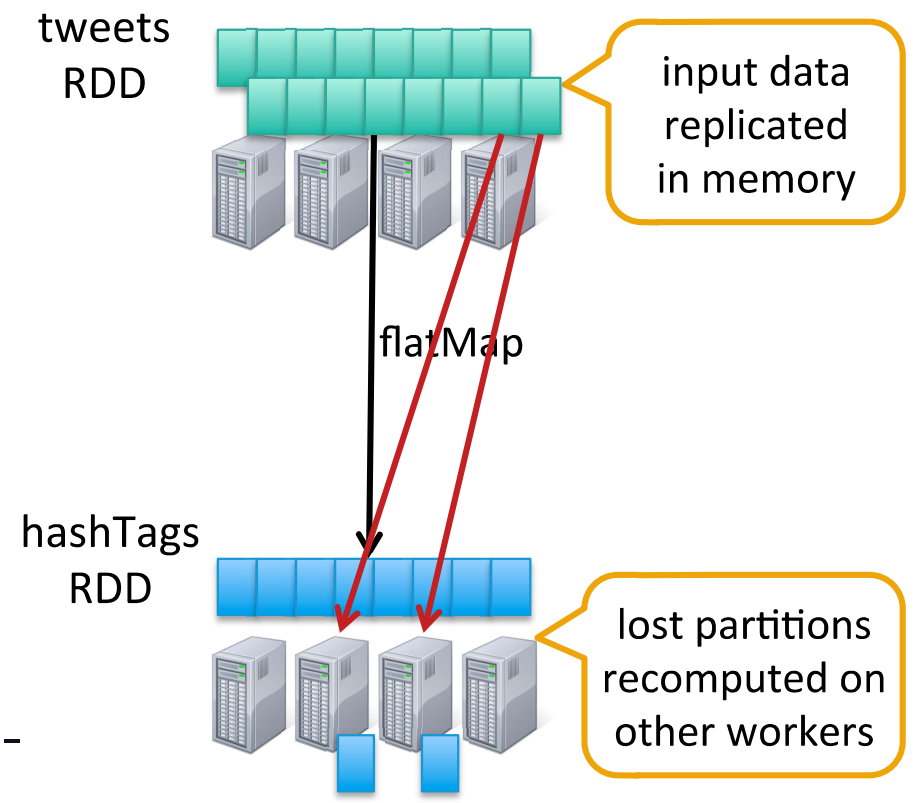




DATABRICKS

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



DATABRICKS

Input Sources

- Out of the box, we provide
  - Kafka, Flume, Kinesis, 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”

Output Sinks

- HDFS, S3, etc (Hadoop API compatible filesystems)
- Cassandra (using Spark-Cassandra connector)
- Hbase (integrated support coming to Spark soon)
- Directly push the data anywhere

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