Hot Data Analytics for Real-Time Streaming in IoT Platform

Dr. Rajiv Misra,Professor,
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss Real-time data processing in IoT edge platform with Spark Streaming and Sliding Window Analytics.

- We will also discuss a case study based on Twitter Sentiment Analysis using Streaming.

Preface

Content of this Lecture:

- In this lecture, we will discuss Real-time data processing in IoT edge platform with Spark Streaming and Sliding Window Analytics.

- We will also discuss a case study based on Twitter Sentiment Analysis using Streaming.

IoT platform: Overview

Things

Sensors
Automation
etc

Azure
Sphere
Azure Iot
Device
SDK

Edge

IoT Edge,
Databox Edge

Cloud IoT

Device
Provision,
Device security,
Device
messaging

IoT Central
IoT Hub,
IoT Hub DPS,
Digital Twins

Data Flow

Hot Path

Real-time
data
Processing

Stream Analytics,
Event Hub,
Functions,
Synapse,
Kafka,
Databricks

Warm Path

Small Batch
Processing

Data Lake,
Functions,
Data Factory,
Synapse,
Databricks,
Azure DBaas

Cold Path

Batch
Processing

Data Lake,
Data Factory,
Synapse,
Databricks,
Azure DBaas

Presentation

Reporting,
Dataset, APIs
Applications

Azure DBaas,
Power BI,
Synapse,
Azure App
services

Consumers

External
systems,
Report
Consumers,
Data
Integration

IoT platform: Data Flow

The data is routed to one of the three different paths i.e. the hot path or the cold path or the warm path.

Hot path data is the data that is processed in real time. It gets processed within seconds of that happening, so when the message hits the hot path it's processed and then it's presented to something in the consumption layer. The consumption layer consume that data immediately once it's been processed in the hot path.

The output from a hot path to a cold storage system can be written that is consumed by an api. The data is written in real time but the api might be querying that data that was written an hour ago.

The main thing about hotpath is that you're processing data in real time as it's happening however what's consuming that might be querying old data that was processed an hour ago. It could be something that's processing it and then presenting it in real time such as a dashboard that is constantly monitoring things in their present state as comes off of the hot path and into the consumption layer.

Data Flow

Hot Path

Real-time data Processing

Stream Analytics, Event Hub, Functions, Synapse, Kafka, Databricks

Warm Path

Small Batch Processing

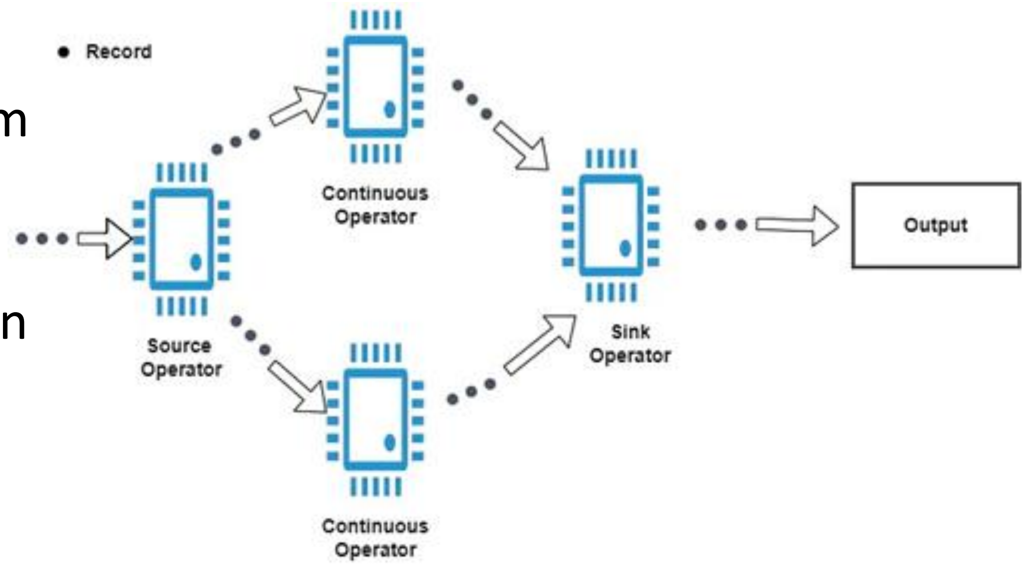
Data Lake, Functions, Data Factory, Synapse, Databricks, Azure DBaas

Cold Path

Batch Processing

Data Lake, Data Factory, Synapse, Databricks, Azure DBaas

IoT platform: Traditional Stream Processing



- Streaming data is received from data sources (e.g. live logs, system telemetry data, IoT device data, etc.) into some data ingestion system like Apache Kafka, Amazon Kinesis, etc.
- The data is then processed in parallel on a cluster.
- Results are given to downstream systems like HBase, Cassandra, Kafka, etc.
- There is a set of worker nodes, each of which runs one or more continuous operators. Each continuous operator processes the streaming data one record at a time and forwards the records to other operators in the pipeline.
- Data is received from ingestion systems via Source operators and given as output t downstream systems via sink operators.
- Continuous operators are a simple and natural model. However, this traditional architecture has also met some challenges with today's trend towards larger scale and more complex real-time analytics

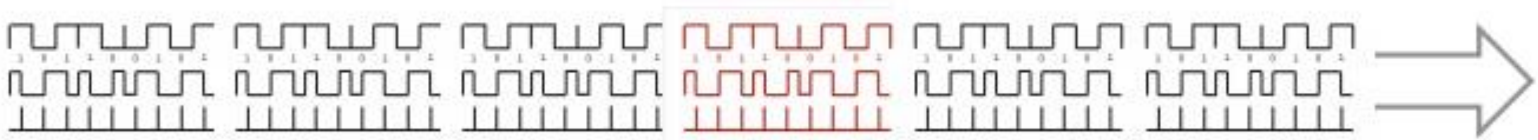
Traditional Stream Processing: Limitations

- **Fast Failure and Straggler Recovery** In real time, the system must be able to fastly and automatically recover from failures and stragglers to provide results which is challenging in traditional systems due to the static allocation of continuous operators to worker nodes.
- **Load Balancing** In a continuous operator system, uneven allocation of the processing load between the workers can cause bottlenecks. The system needs to be able to dynamically adapt the resource allocation based on the workload.
- **Unification of Streaming, Batch and Interactive Workloads** In many use cases, it is also attractive to query the streaming data interactively, or to combine it with static datasets (e.g. pre-computed models). This is hard in continuous operator systems which does not designed to new operators for ad-hoc queries. This requires a single engine that can combine batch, streaming and interactive queries.
- **Advanced Analytics with Machine learning and SQL Queries** Complex workloads require continuously learning and updating data models, or even querying the streaming data with SQL queries. Having a common abstraction across these analytic tasks makes the developer's job much easier.

Big Streaming Data Processing

Fraud detection in bank transactions

Anomalies in sensor data



Cat videos in tweets

How to Process Big Streaming Data

- Scales to hundreds of nodes
- Achieves low latency
- Efficiently recover from failures
- Integrates with batch and interactive processing

What people have been doing?

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

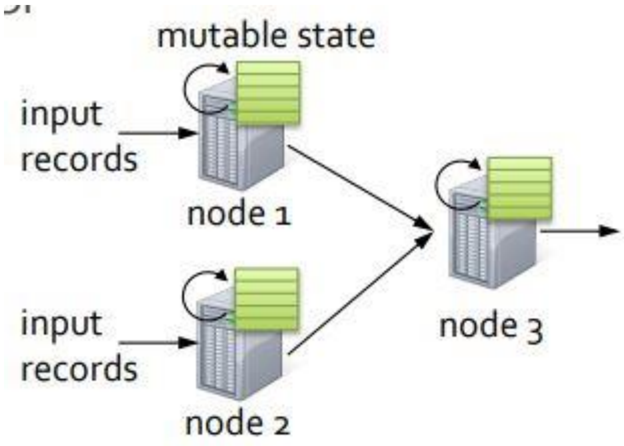
- Existing frameworks cannot do both
  - Either, stream processing of 100s of MB/s with low latency
  - Or, batch processing of TBs of data with high latency

What people have been doing?

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

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



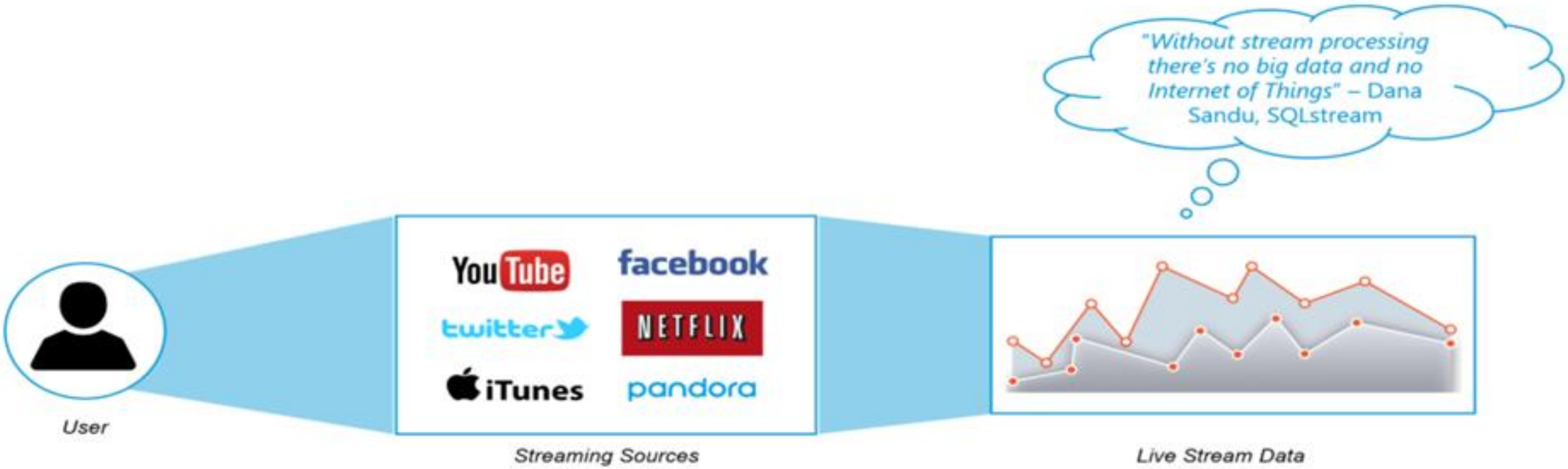
- Mutable state is lost if node fails

- Making stateful stream processing fault-tolerant is challenging!

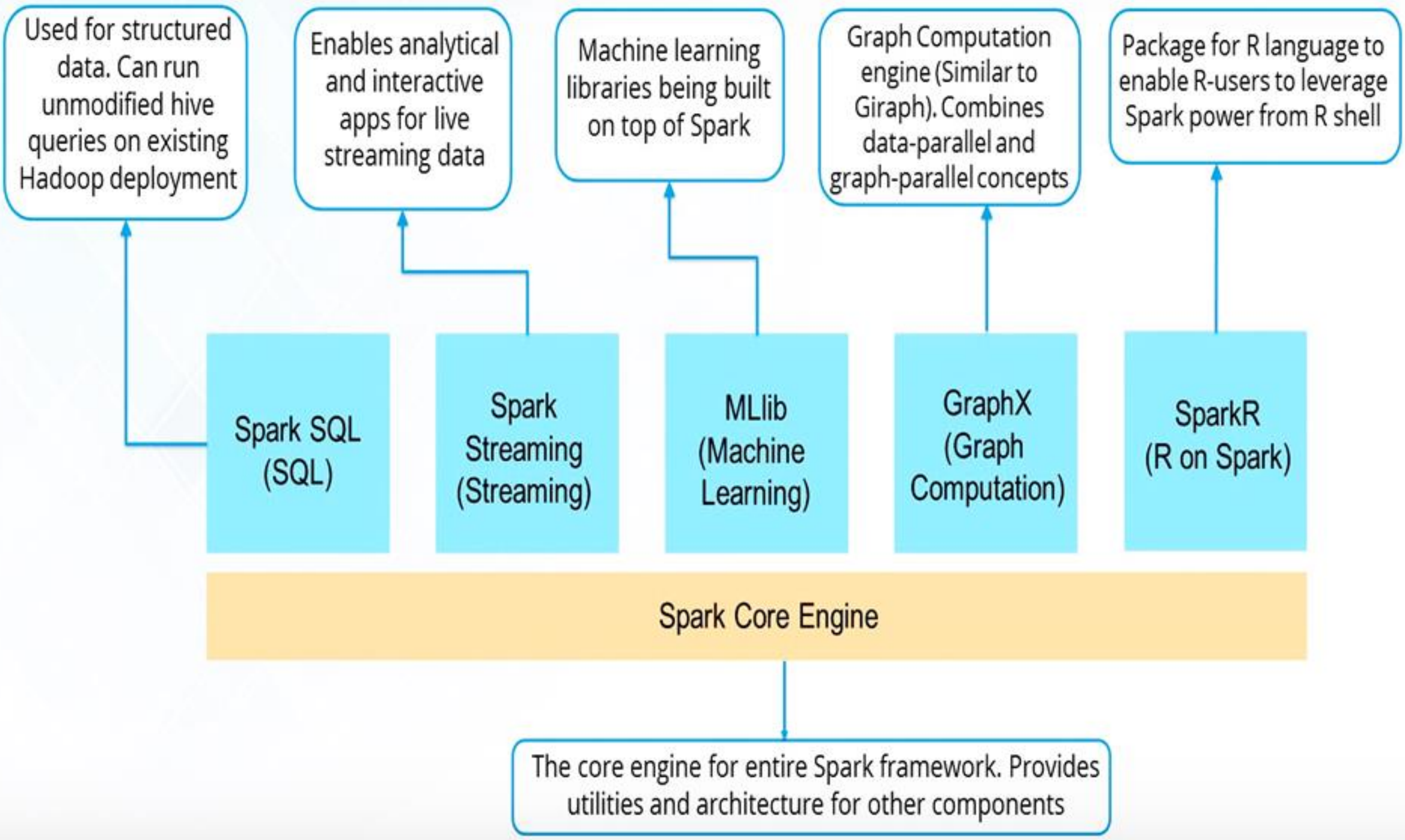
What is Streaming?

- Data Streaming is a technique for transferring data so that it can be processed as a steady and continuous stream.

- Streaming technologies are becoming increasingly important with the growth of the Internet.

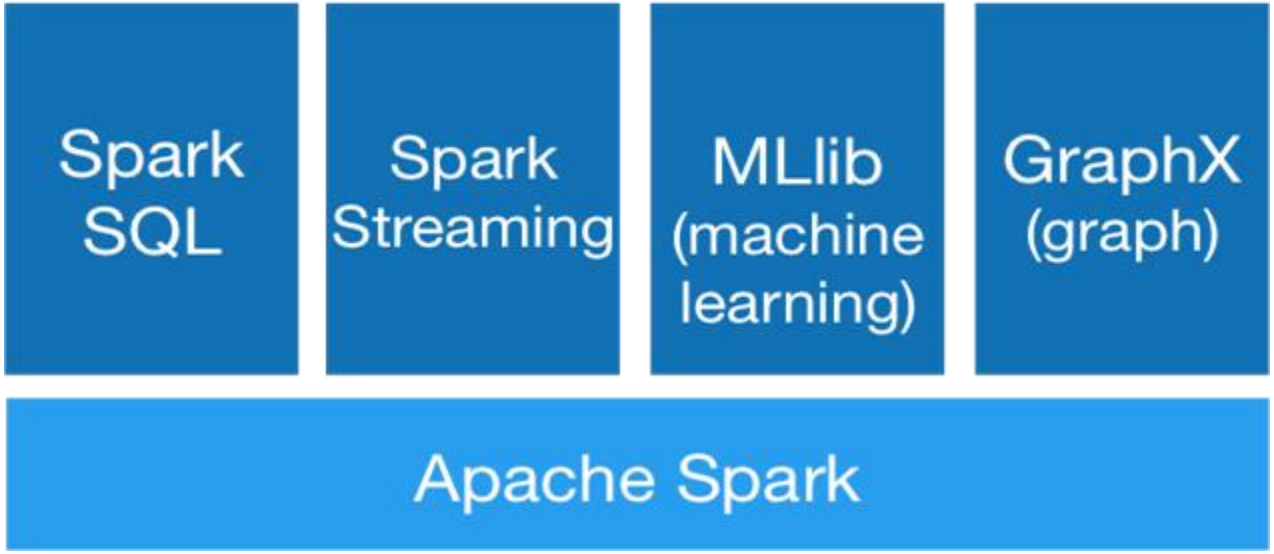


Spark Ecosystem



What is Spark Streaming?

- Extends Spark for doing big data stream processing
- Project started in early 2012, alpha released in Spring 2017 with Spark 0.7
- Moving out of alpha in Spark 0.9
- Spark Streaming has support built-in to consume from Kafka, Flume, Twitter, ZeroMQ, Kinesis, and TCP/IP sockets.
- In Spark 2.x, a separate technology based on Datasets, called Structured Streaming, that has a higher-level interface is also provided to support 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.

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

Why Spark Streaming ?

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

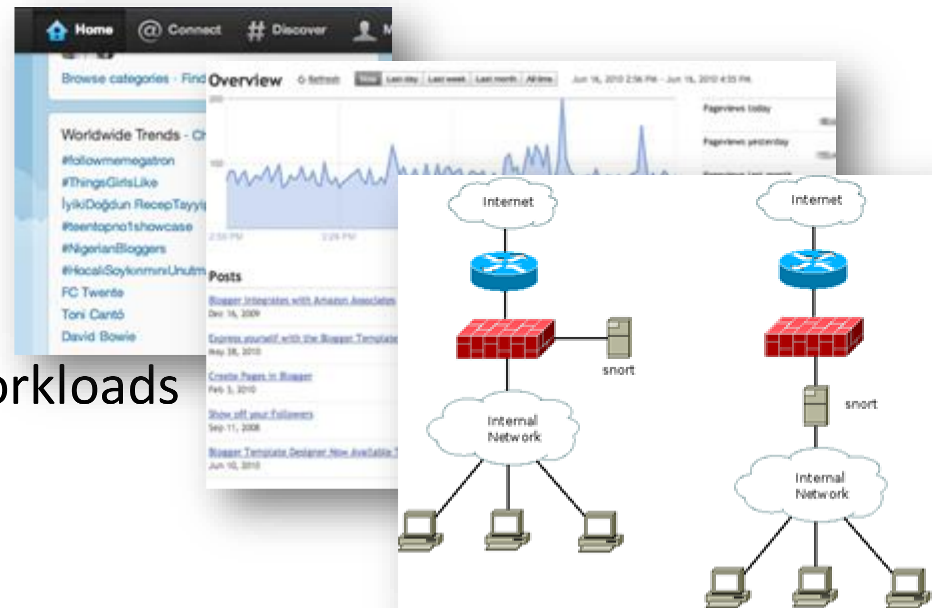
Website monitoring

Fraud detection

Ad monetization

Why Spark Streaming ?

- 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

Why Spark Streaming ?

- We can use Spark Streaming to stream real-time data from various sources like Twitter, Stock Market and Geographical Systems and perform powerful analytics to help businesses.

Spark Streaming is used to stream real-time data from various sources like Twitter, Stock Market and Geographical Systems and perform powerful analytics to help businesses.

Why Spark Streaming?

Need a framework for big data stream processing that

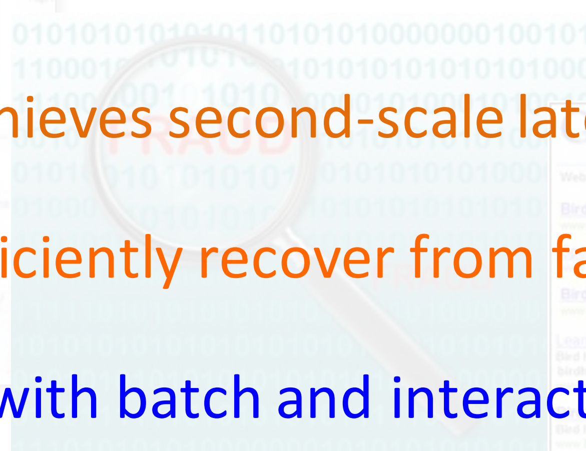
Scales to hundreds of nodes

Achieves second-scale latencies

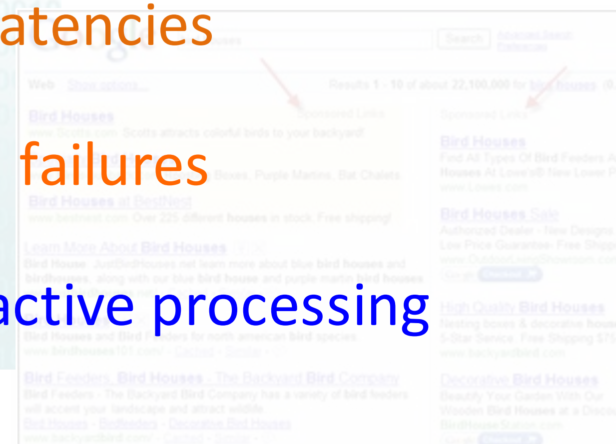
Efficiently recover from failures

Integrates with batch and interactive processing

Website monitoring

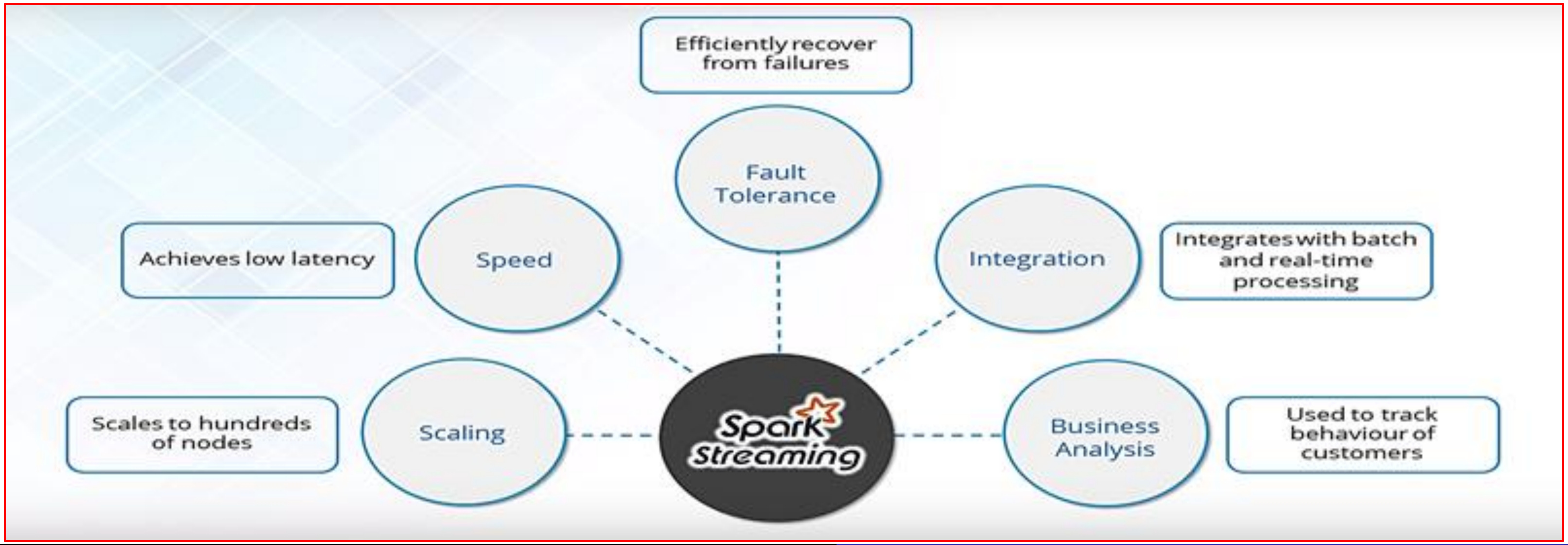


Ad monetization



Spark Streaming Features

- **Scaling:** Spark Streaming can easily scale to hundreds of nodes.
- **Speed:** It achieves low latency.
- **Fault Tolerance:** Spark has the ability to efficiently recover from failures.
- **Integration:** Spark integrates with batch and real-time processing.
- **Business Analysis:** Spark Streaming is used to track the behavior of customers which can be used in business analysis



Requirements

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

Batch vs Stream Processing

Batch Processing

- Ability to process and analyze data at-rest (stored data)
- Request-based, bulk evaluation and short-lived processing
- Enabler for **Retrospective, Reactive and On-demand Analytics**

Stream Processing

- Ability to ingest, process and analyze data in-motion in real- or near-real-time
- Event or micro-batch driven, continuous evaluation and long-lived processing
- Enabler for **real-time Prospective, Proactive and Predictive Analytics** for Next Best Action

Stream Processing + Batch Processing = All Data Analytics

real-time (now) historical (past)

Integration with Batch Processing

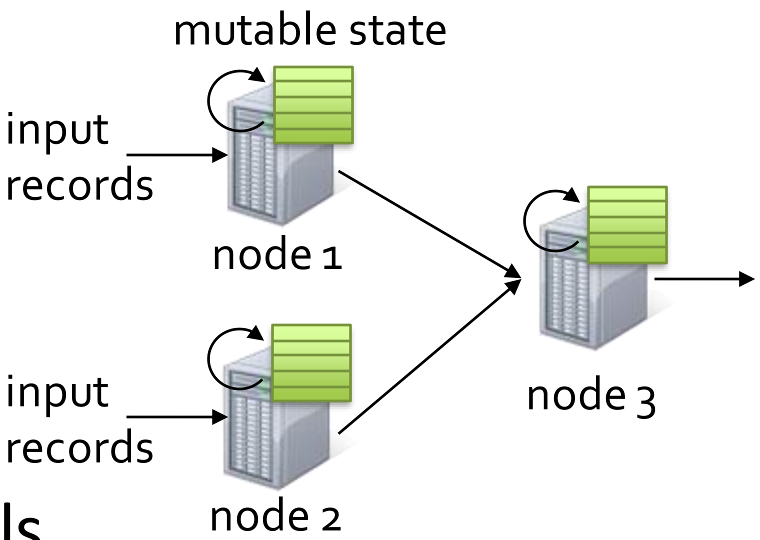
- Many environments require processing same data in live streaming as well as batch post-processing

- 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 two differe
  - Different programming models
  - Double implementation effort

Stateful Stream Processing

- Traditional model
  - Processing pipeline of nodes
  - Each node maintains mutable state
  - Each input record updates the state and new records are sent out

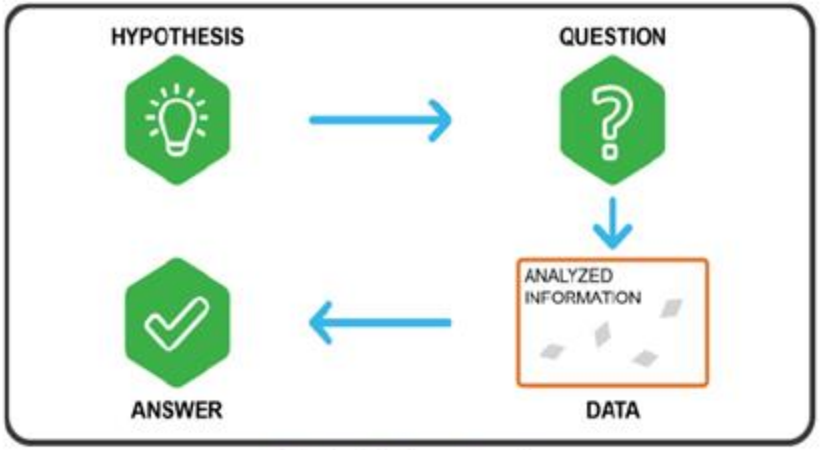


- Mutable state is lost if node fails
- Making stateful stream processing fault tolerant is challenging!

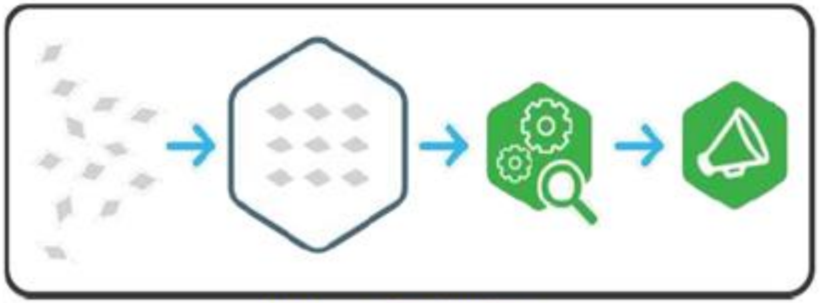
Modern Data Applications approach to Insights

Traditional Analytics

Structured & Repeatable
Structure built to store data



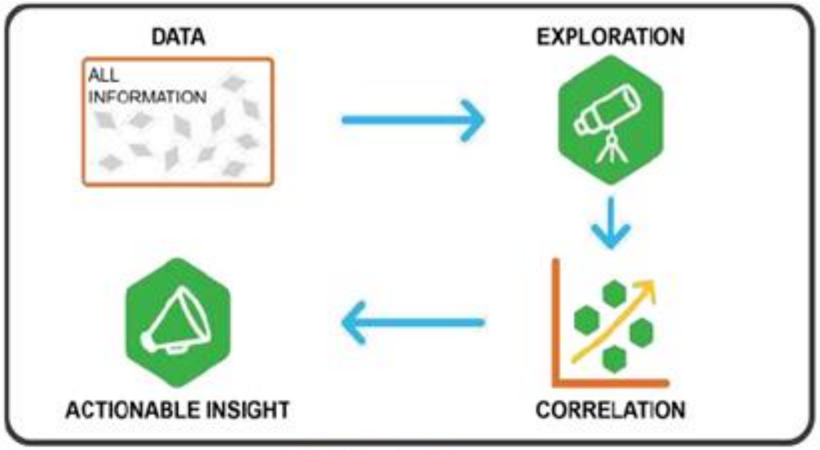
Start with hypothesis
Test against selected data



Analyze after landing...

Next Generation Analytics

Iterative & Exploratory
Data is the structure



Data leads the way
Explore *all* data, identify correlations

Analyze in motion...

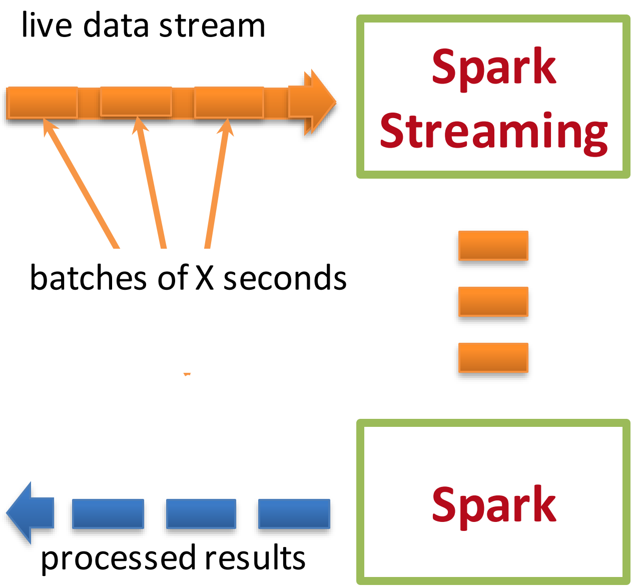
Existing Streaming Systems

- 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

How does Spark Streaming work?

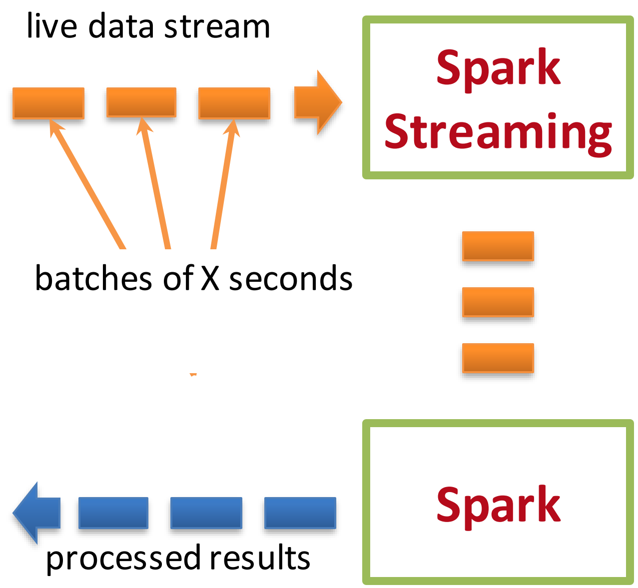
Run a streaming computation as a **series of very small, deterministic batch jobs**

- 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



How does Spark Streaming work?

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

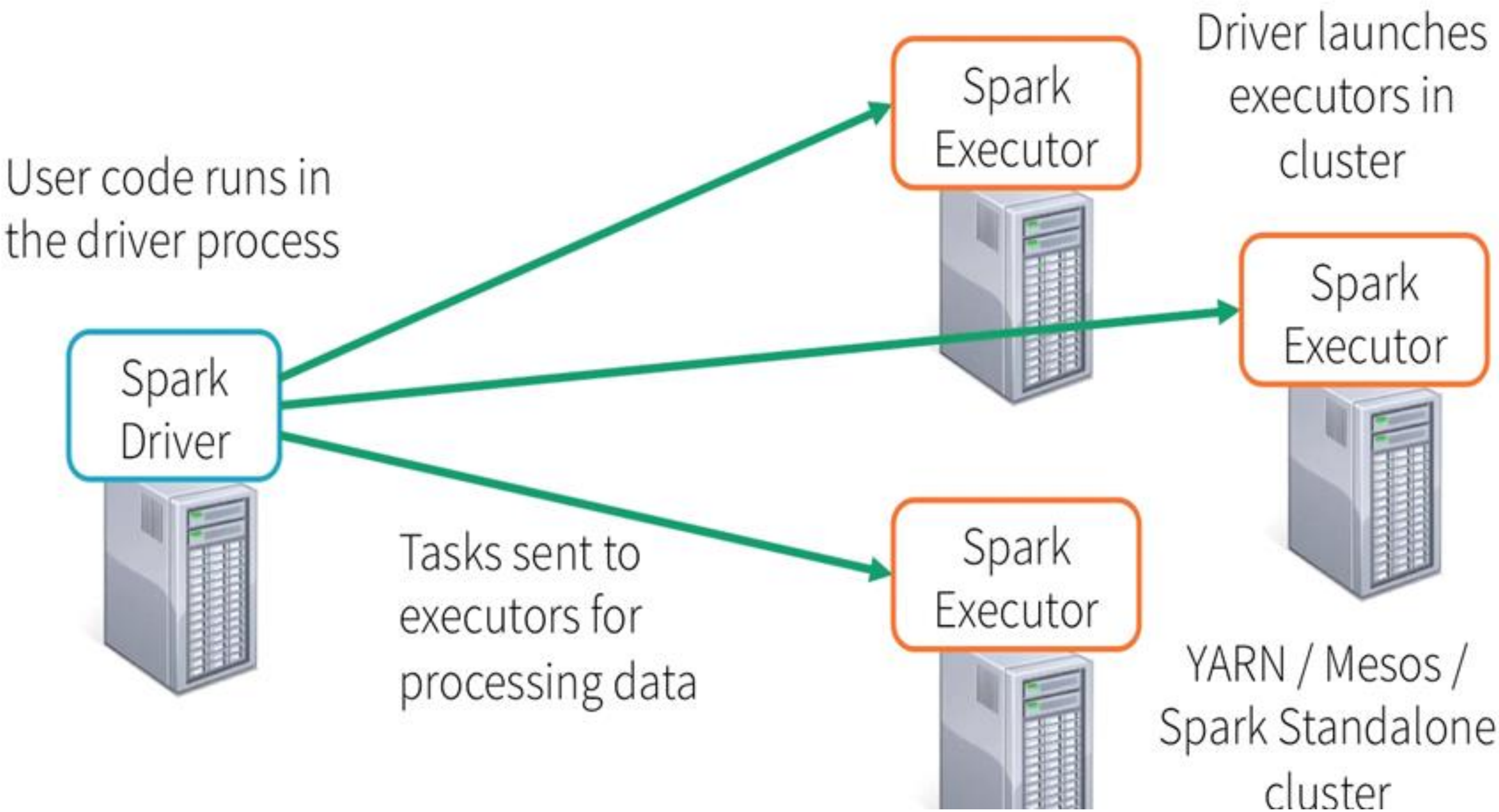


- Batch sizes as low as ½ second, latency of about 1 second
- Potential for combining batch processing and streaming processing in the same system

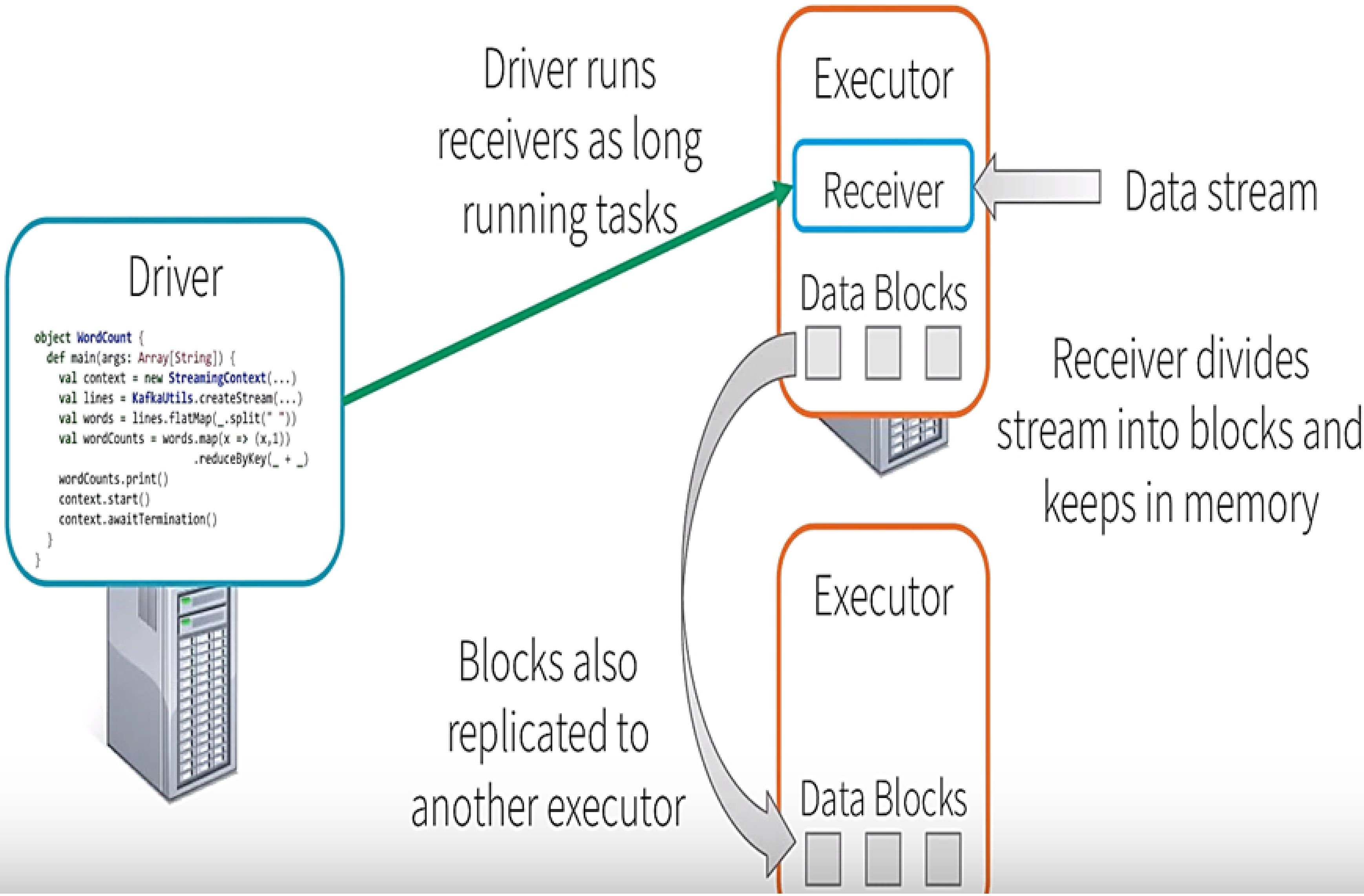
Word Count with Kafka

```
object WordCount {
  def main(args: Array[String]) {
    val context = new StreamingContext(new SparkConf(), Seconds(1))
    val lines = KafkaUtils.createStream(context, ...)
    val words = lines.flatMap(_.split(" "))
    val wordCounts = words.map(x => (x,1)).reduceByKey(_ + _)
    wordCounts.print()
    context.start()
    context.awaitTermination()
  }
}
```

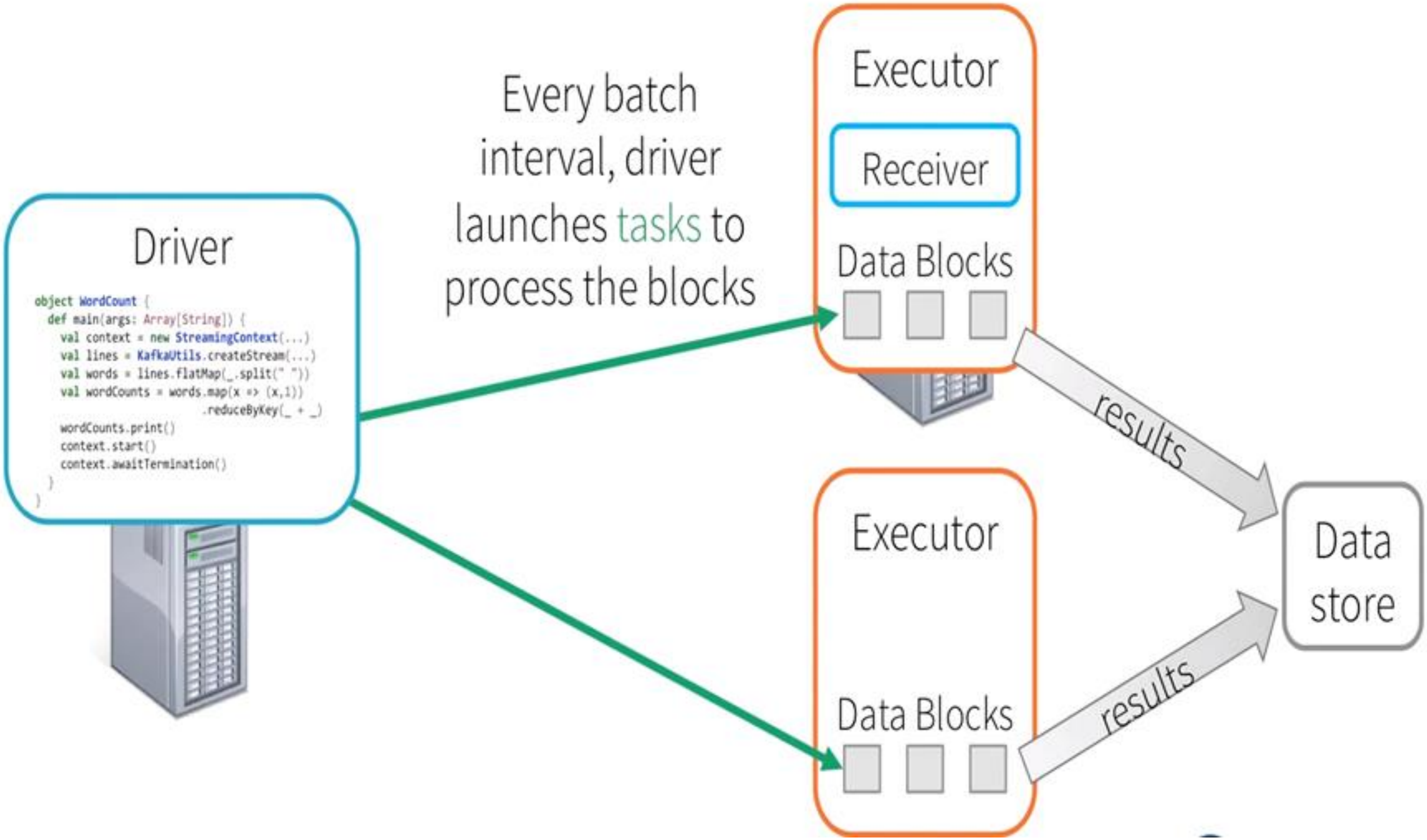
Any Spark Application



Spark Streaming Application: Receive data

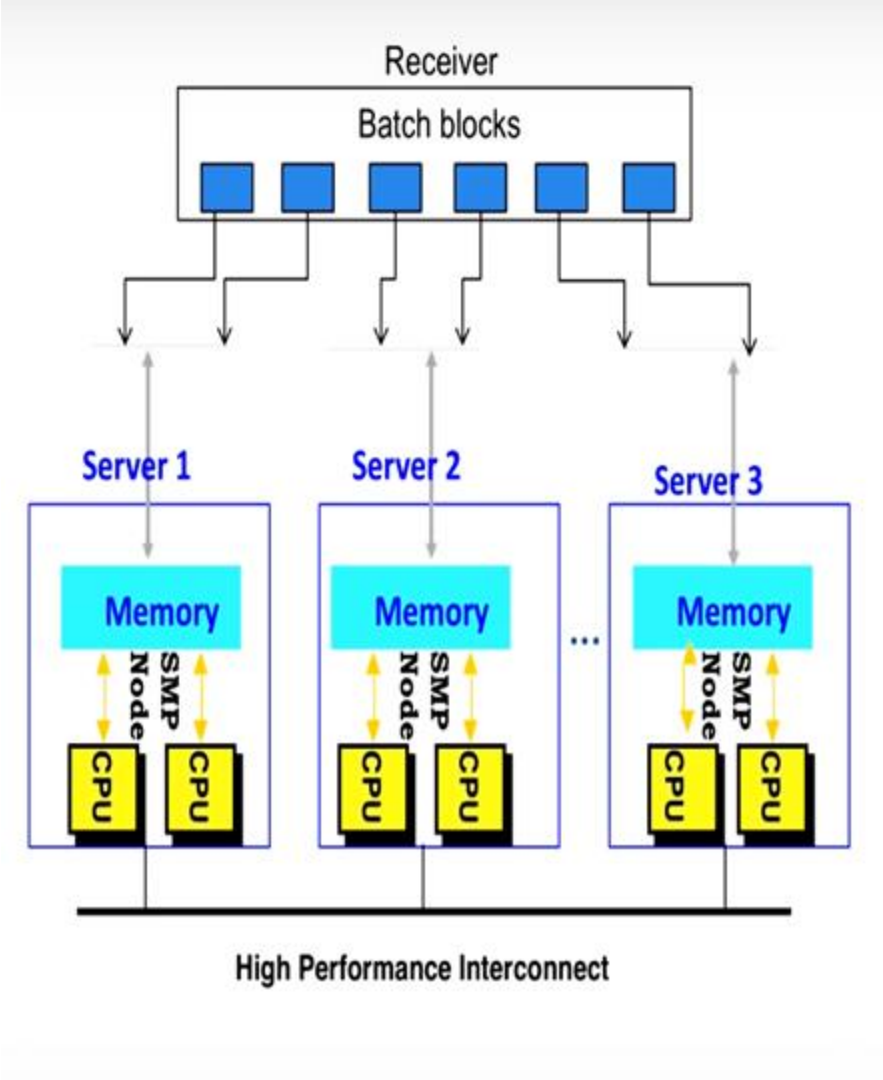


Spark Streaming Application: Process data



Spark Streaming Architecture

- Micro batch architecture.
- Operates on interval of time
- New batches are created at regular time intervals.
- Divides received time batch into blocks for parallelism
- Each batch is a graph that translates into multiple jobs
- Has the ability to create larger size batch window as it processes over time.



Spark Streaming Workflow

- Spark Streaming workflow has four high-level stages. The first is to stream data from various sources. These sources can be streaming data sources like Akka, Kafka, Flume, AWS or Parquet for real-time streaming. The second type of sources includes HBase, MySQL, PostgreSQL, Elastic Search, Mongo DB and Cassandra for static/batch streaming.
- Once this happens, Spark can be used to perform Machine Learning on the data through its MLlib API. Further, Spark SQL is used to perform further operations on this data. Finally, the streaming output can be stored into various data storage systems like HBase, Cassandra, MemSQL, Kafka, Elastic Search, HDFS and local file system.

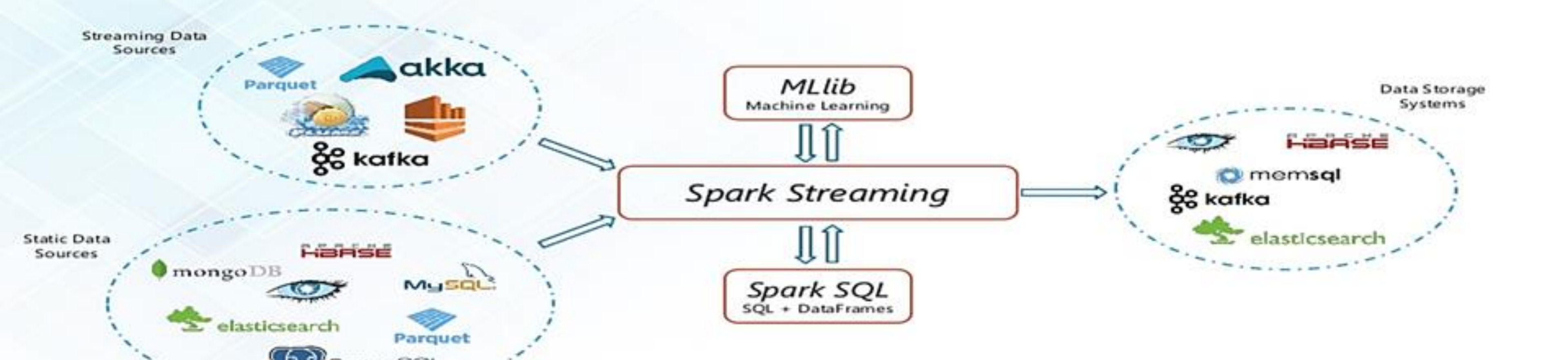


Figure: *Overview Of Spark Streaming*

Spark Streaming Workflow

Figure: *Data from a variety of sources to various storage systems*

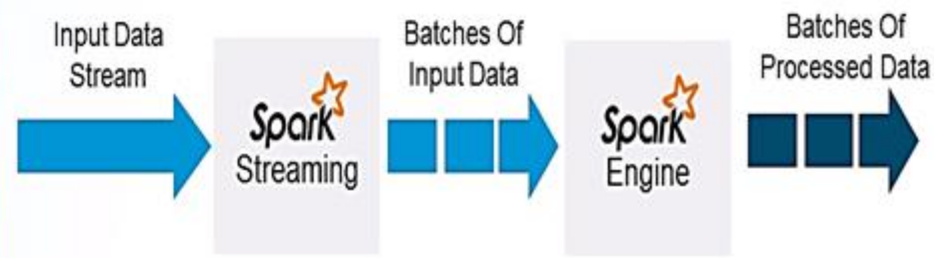


Figure: *Incoming streams of data divided into batches*

Figure: *Input data stream divided into discrete chunks of data*

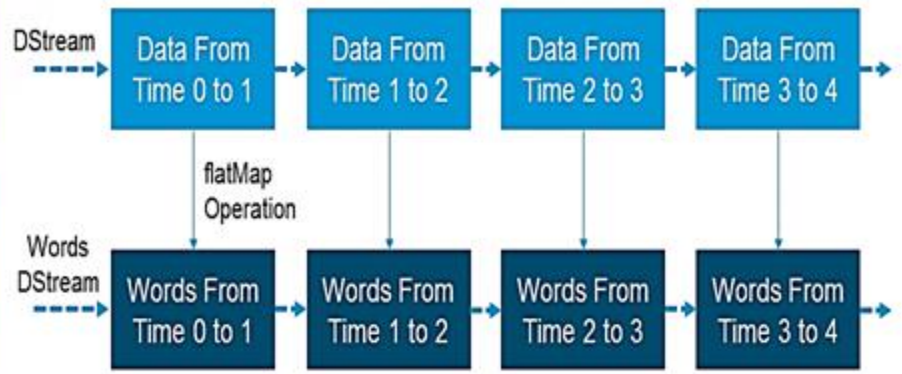


Figure: *Extracting words from an InputStream*

Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
```

DStream: a sequence of RDDs representing a stream of data

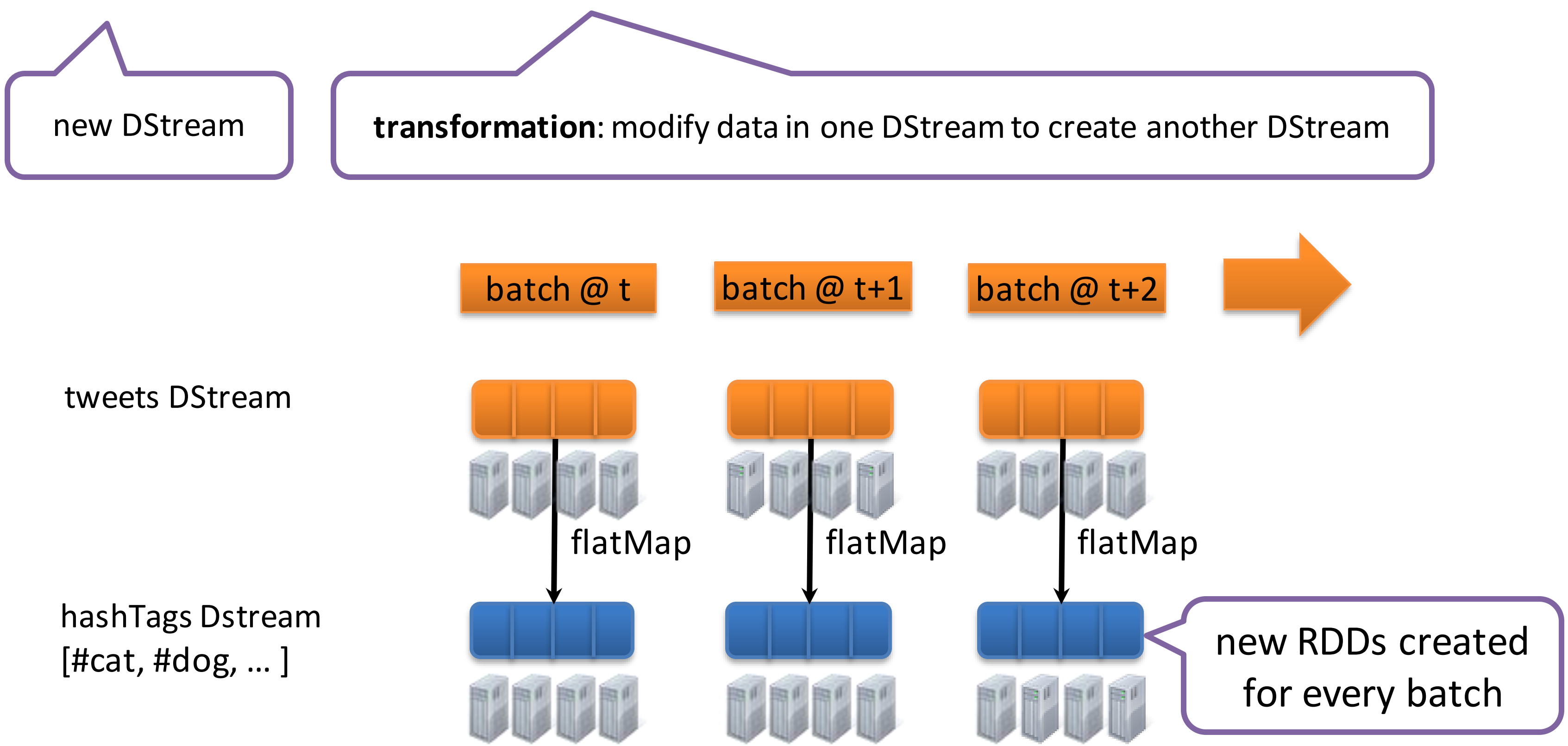
Twitter Streaming API

tweets DStream

stored in memory as an RDD
(immutable, distributed)

Example 1 – Get hashtags from Twitter

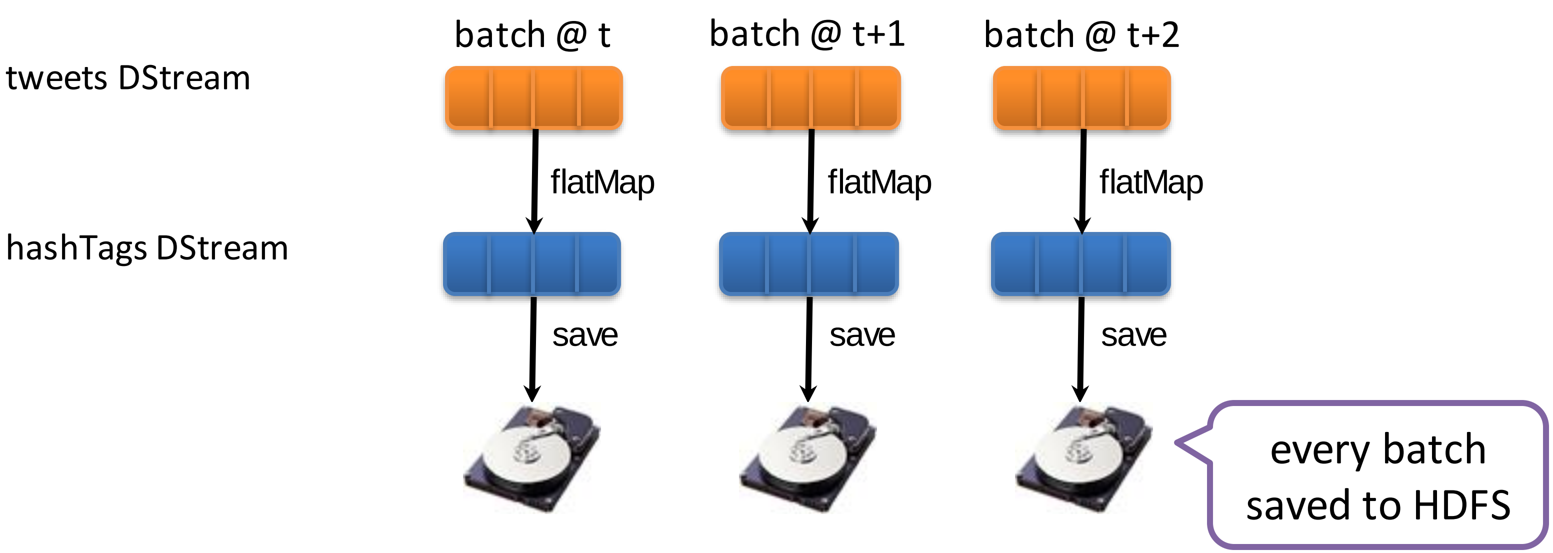
```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
```



Example 1– Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

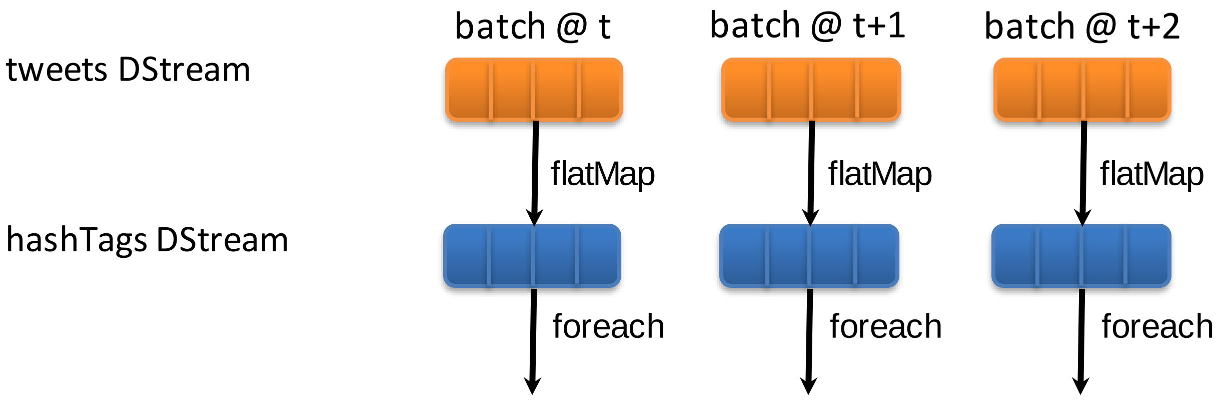
output operation: to push data to external storage



Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.foreach(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data



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

Java Example

Scala

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

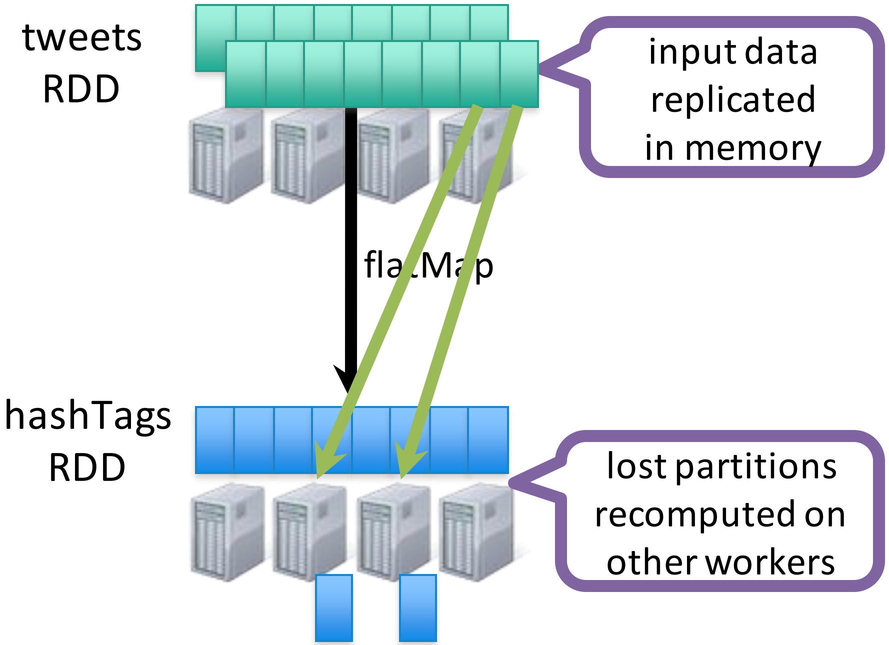
Java

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

Function object

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

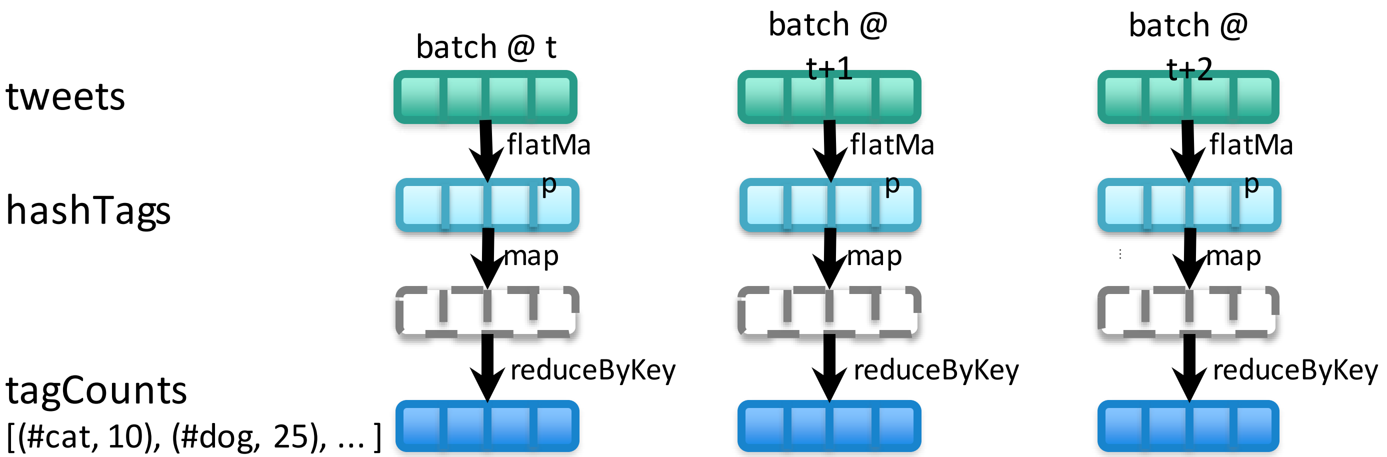


Key concepts

- **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()
```



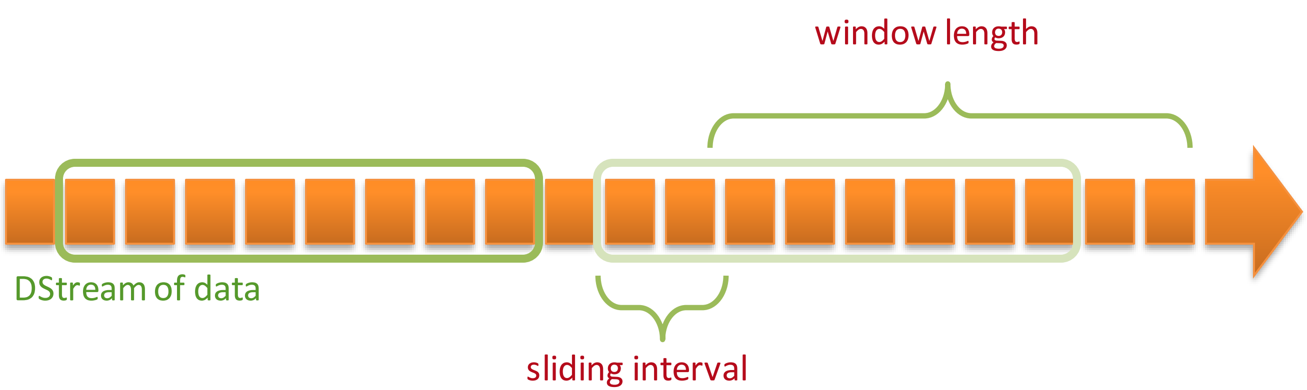
Example 3 – Count the hashtags over last 10 mins

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

sliding window operation

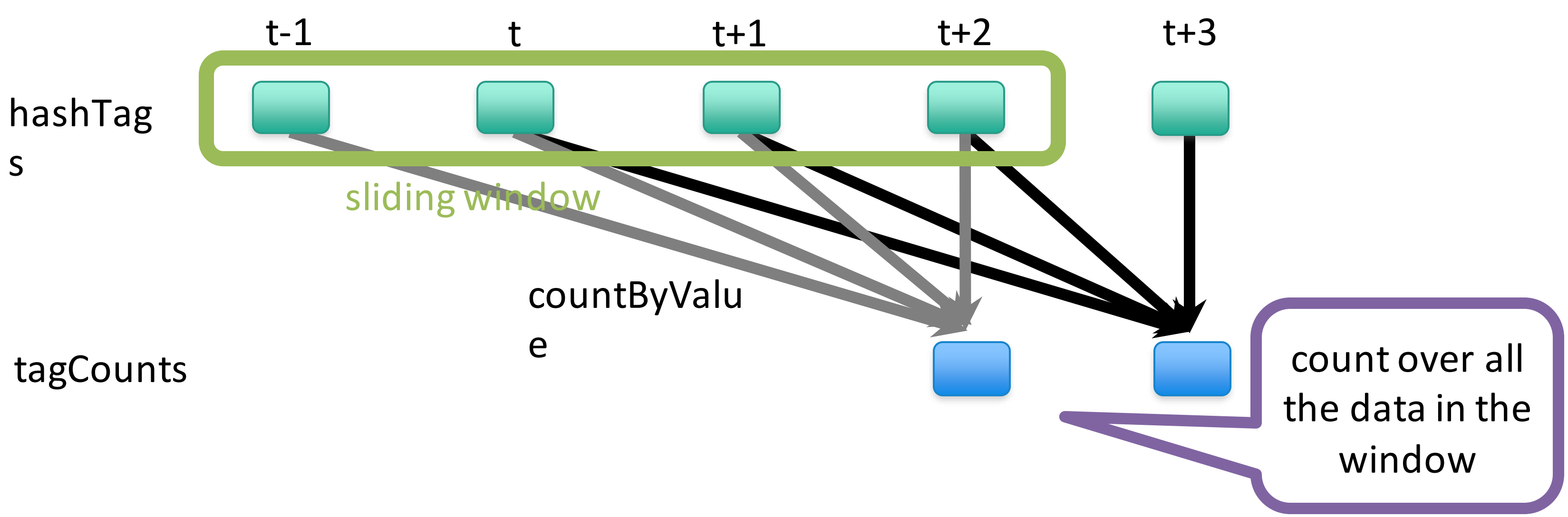
window length

sliding interval



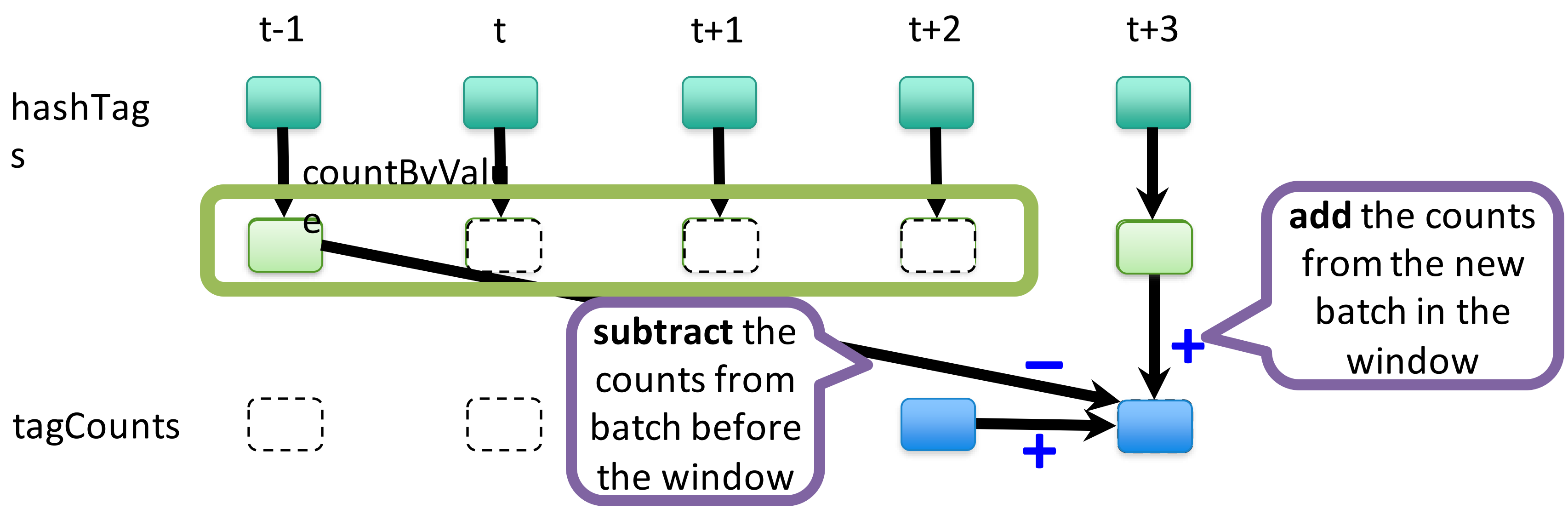
Example 3 – Counting the hashtags over last 10 mins

```
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), …)

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

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(spamHDFSFile).filter(...)
})
```

Spark Streaming-Dstreams, Batches and RDDs

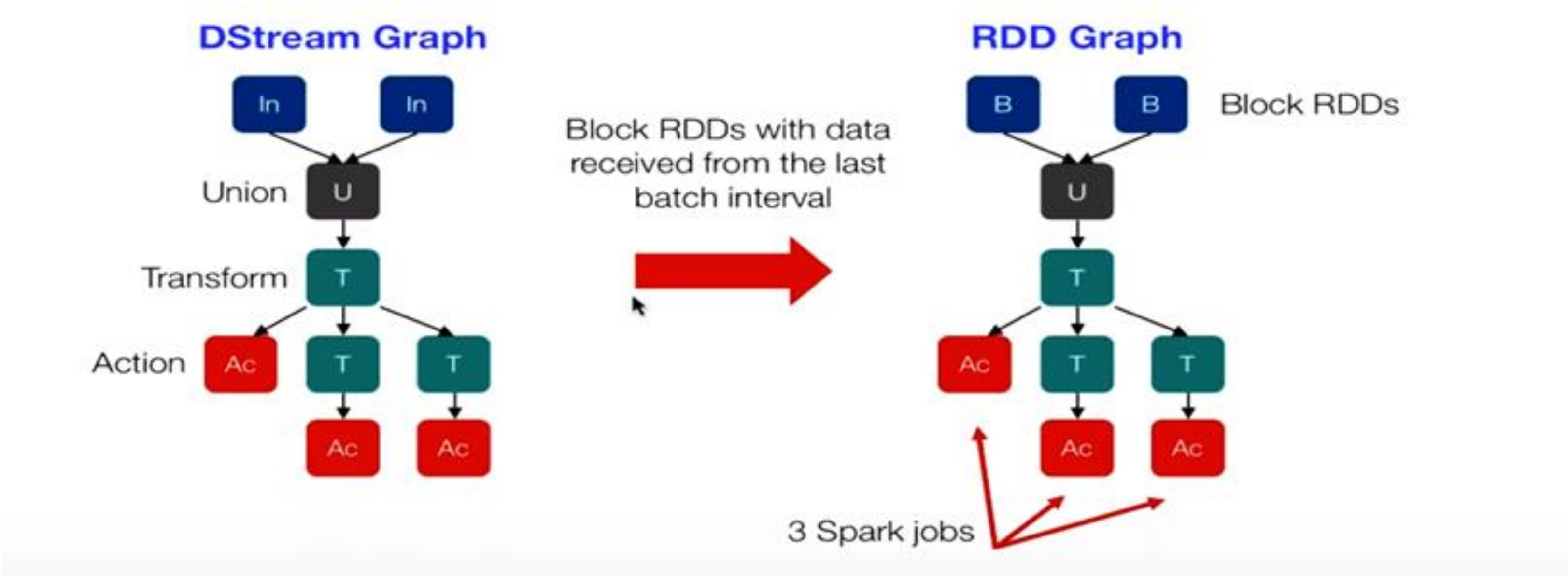
- These steps repeat for each batch.. Continuously
- Because we are dealing with Streaming data. Spark Streaming has the ability to “remember” the previous RDDs…to some extent.

DStreams + RDDs = Power

- Online machine learning
  - Continuously learn and update data models (*updateStateByKey* and *transform*)
- Combine live data streams with historical data
  - Generate historical data models with Spark, etc.
  - Use data models to process live data stream (*transform*)
- CEP-style processing
  - window-based operations (reduceByWindow, etc.)

From DStreams to Spark Jobs

- Every interval, an RDD graph is computed from the DStream graph
- For each output operation, a Spark action is created
- For each action, a Spark job is created to compute it



Input Sources

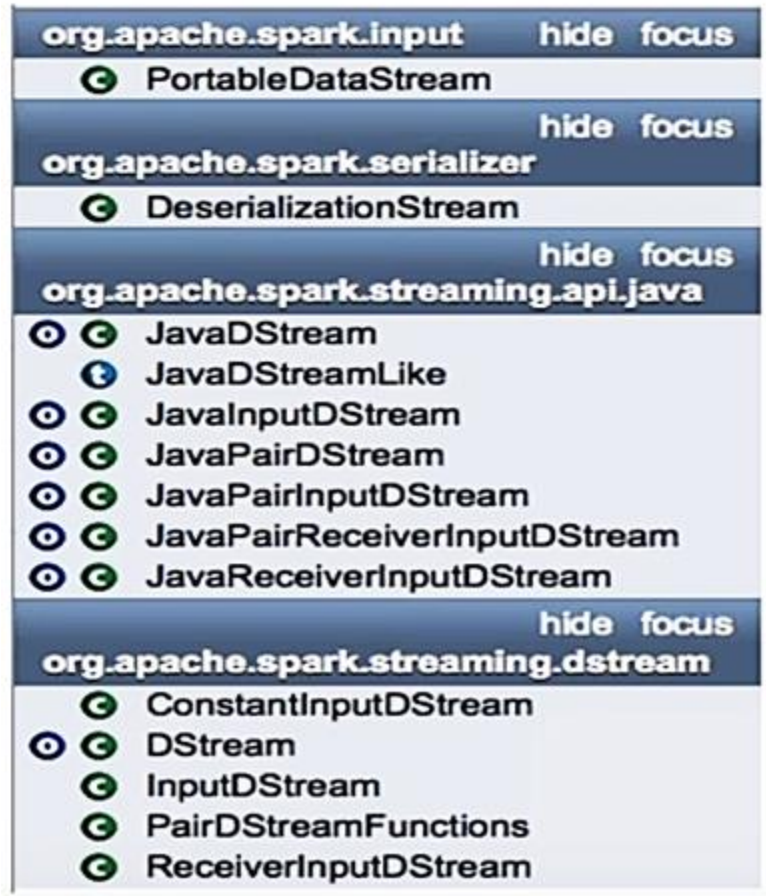
- Out of the box, we provide
  - Kafka, HDFS, Flume, Akka Actors, Raw TCP sockets, etc.
- Very easy to write a *receiver* for your own data source
- Also, generate your own RDDs from Spark, etc. and push them in as a "stream"

Current Spark Streaming I/O

- Input Sources
  - Kafka, Flume, Twitter, ZeroMQ, MQTT, TCP sockets
  - Basic sources: sockets, files, Akka actors
  - Other sources require receiver threads
- Output operations
  - Print(), saveAsTextFiles(), saveAsObjectFiles(), saveAsHadoopFiles(), foreachRDD()
  - foreachRDD can be used for message queues, DB operations and more

Dstream Classes

- Different classes for different languages (Scala, Java)
- Dstream has 36 value members
- Multiple types of Dstreams
- Separate Python API



Spark Streaming Operations

- All the Spark RDD operations
  - Some available through the transform() operation

map/flatmap	filter	repartition	union
count	reduce	countByValue	reduceByKey
join	cogroup	transform	updateStateByKey

- Spark Streaming window operations

window	countByWindow	reduceByWindow
reduceByKeyAndWindow	countByValueAndWindow	

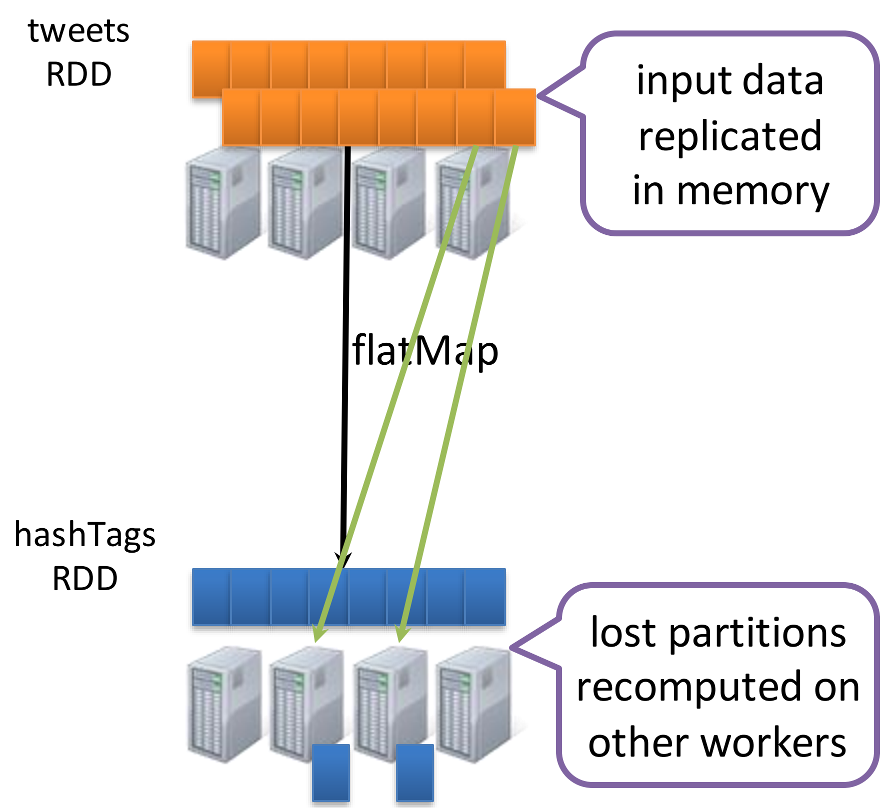
- Spark Streaming output operations

print	saveAsTextFiles	saveAsObjectFiles
saveAsHadoopFiles	foreachRDD	

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



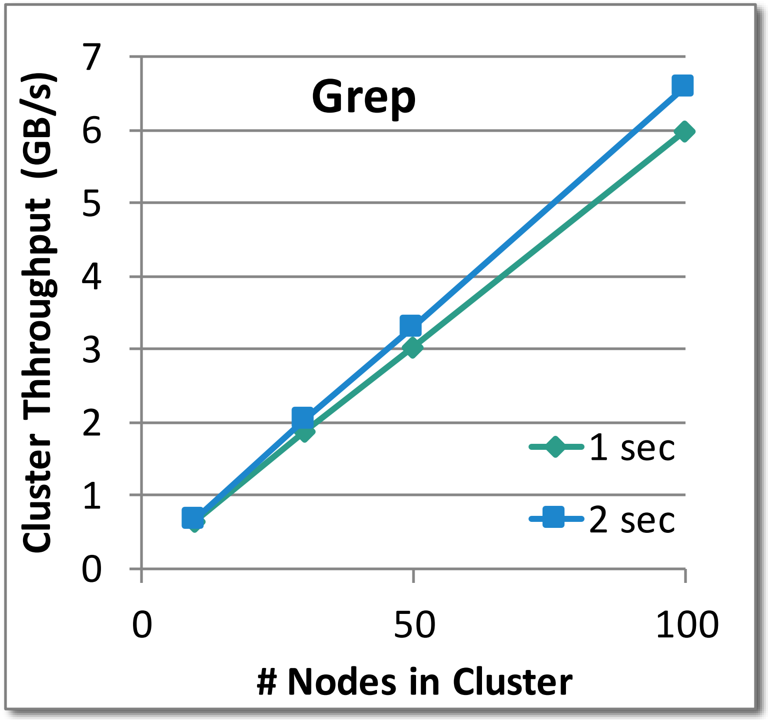
  - All transformations are fault-tolerant, and *exactly-once* transformations

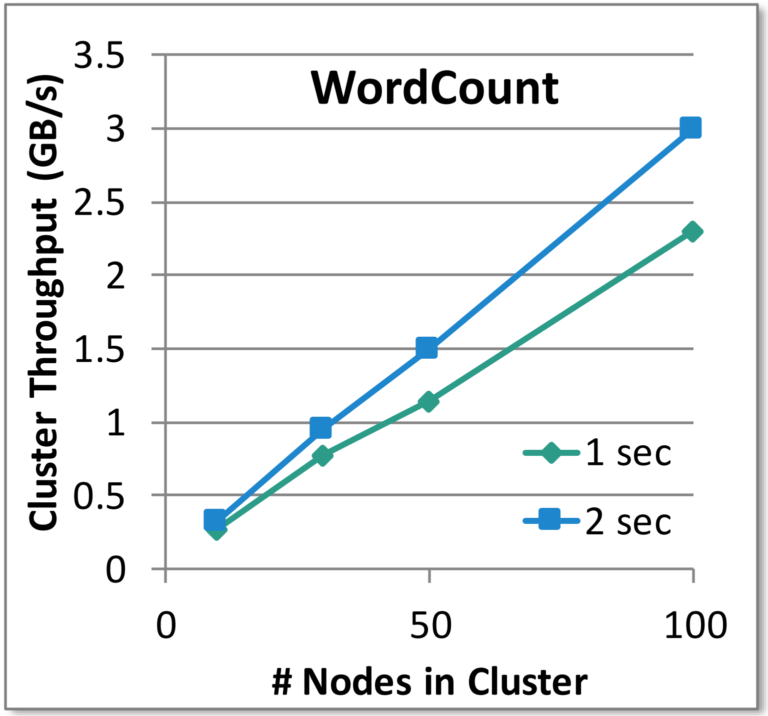
Fault-tolerance

- Received data is replicated among multiple Spark executors
  - Default factor: 2
- Checkpointing
  - Saves state on regular basis, typically every 5-10 batches of data
  - A failure would have to replay the 5-10 previous batched to recreate the appropriate RDDs
  - Checkpoint done to HDFS or equivalent
- Must protect the driver program
  - If the driver node running the Spark Streaming application fails
  - Driver must be restarted on another node.
  - Requires a checkpoint directory in the StreamingContext
- Streaming Backpressure
  - spark.streaming.backpressure.enabled
  - spark.streaming.receiver.maxRate

Performance

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

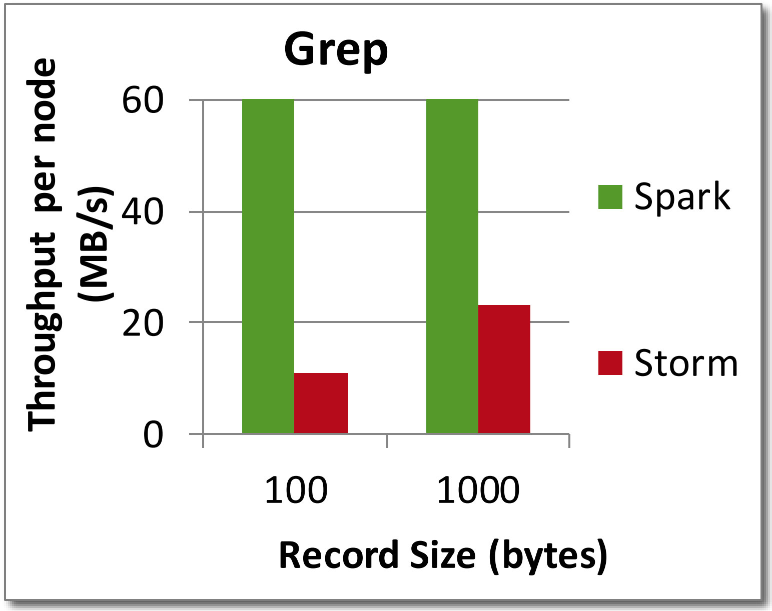


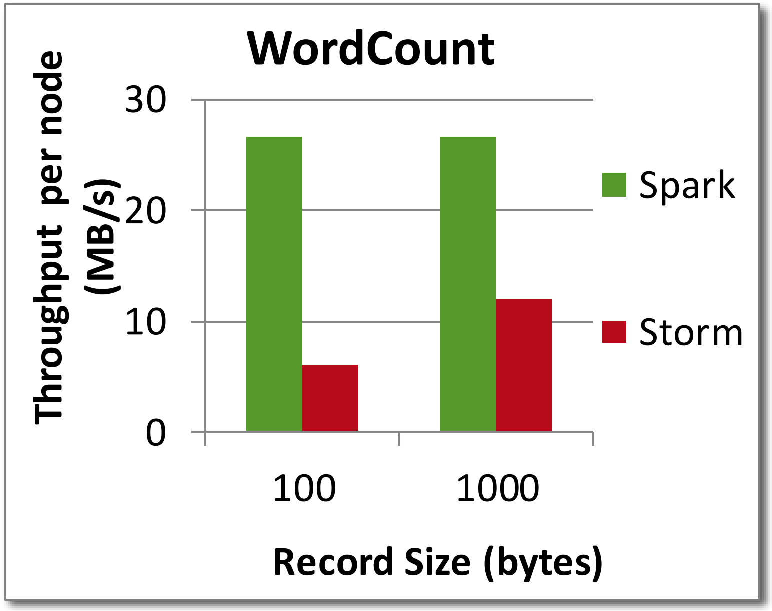


Comparison with other systems

Higher throughput than Storm

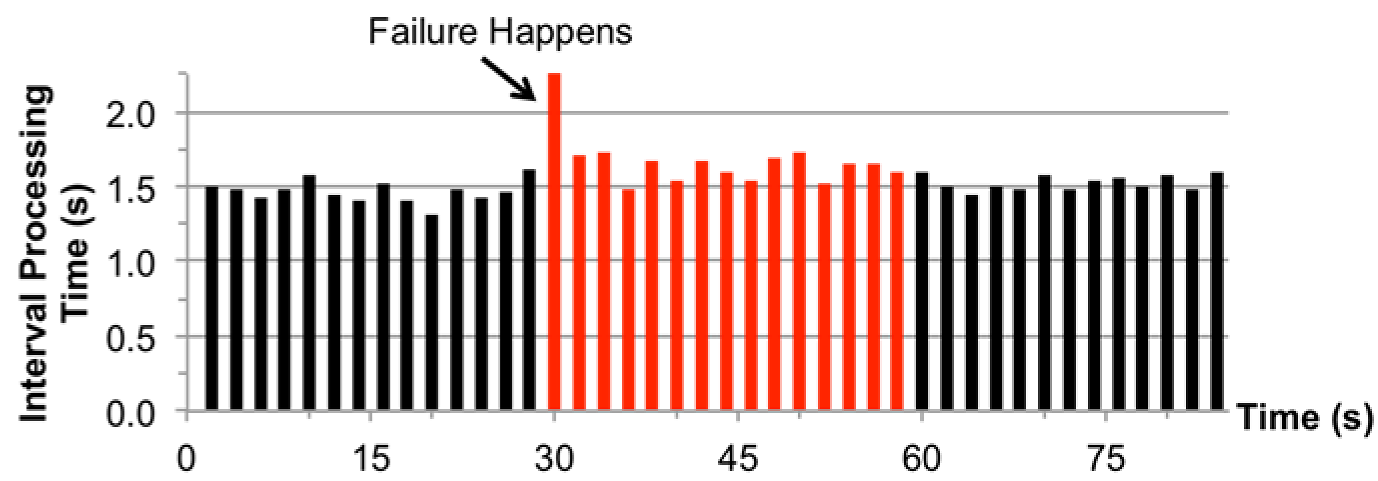
- Spark Streaming: **670k** records/sec/node
- Storm: **115k** records/sec/node
- Commercial systems: **100-500k** records/sec/node





Fast Fault Recovery

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

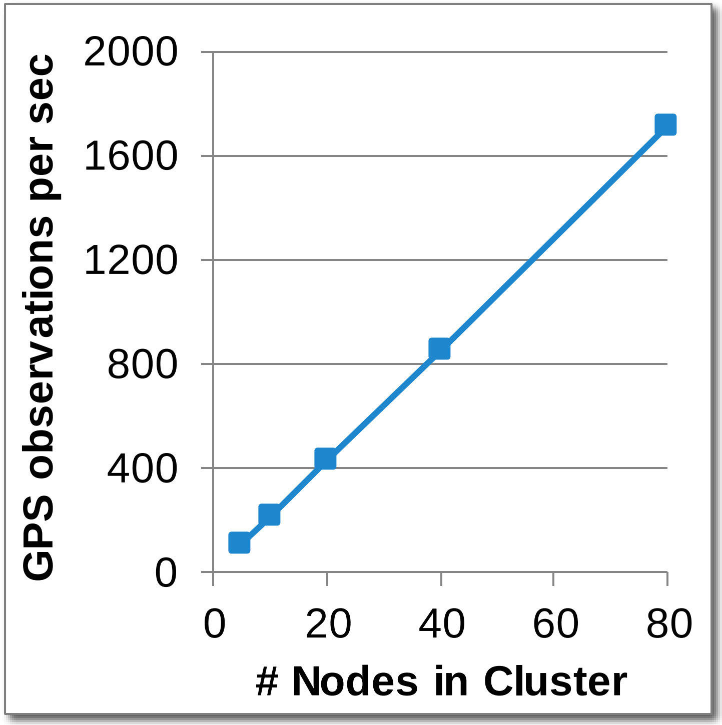


Sliding WordCount on 10 nodes with 30s checkpoint interval

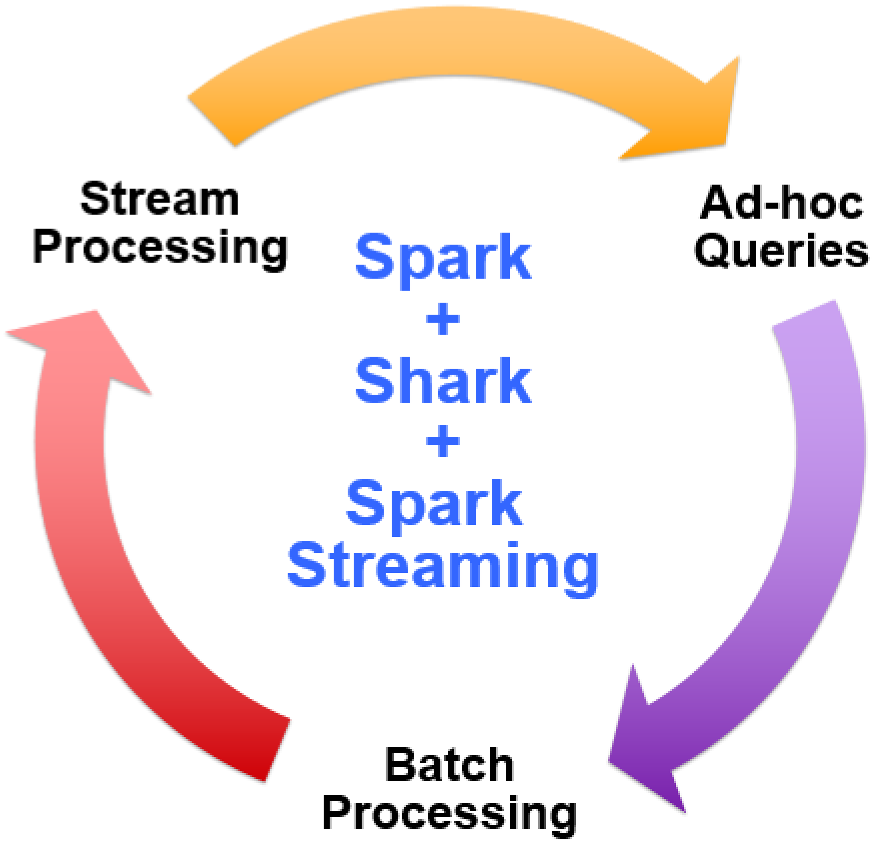
Real time application: Mobile Millennium Project

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://...")
```

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 = sc.kafkaStream(...)
    val filtered = stream.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
    ...
  }
}
```

Roadmap

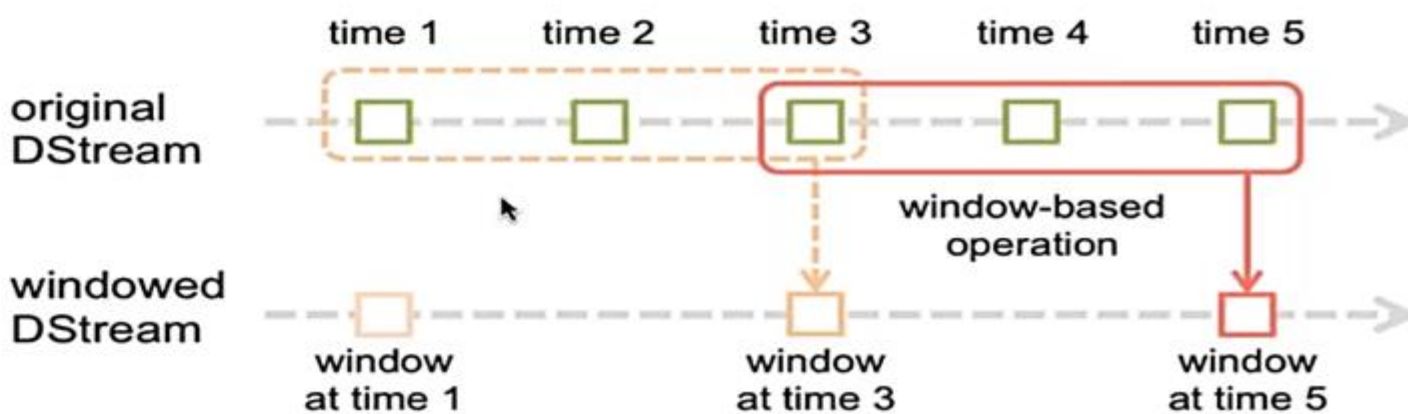
- Spark 0.8.1
  - Marked alpha, but has been quite stable
  - Master fault tolerance – manual recovery
    - Restart computation from a checkpoint file saved to HDFS
- Spark 0.9 in Jan 2014 – out of alpha!
  - Automated master fault recovery
  - Performance optimizations
  - Web UI, and better monitoring capabilities
- Spark v2.4.0 released in November 2, 2018

Sliding Window Analytics

Spark Streaming Windowing Capabilities

- **Parameters**
  - **Window length:** duration of the window
  - **Sliding interval:** interval at which the window operation is performed
  - Both the parameters must be a multiple of the batch interval

- A window creates a new DStream with a larger batch size



Spark Window Functions

Spark Window Functions for DataFrames and SQL

Introduced in Spark 1.4, Spark window functions improved the expressiveness of Spark DataFrames and Spark SQL. With window functions, you can easily calculate a moving average or cumulative sum, or reference a value in a previous row of a table. Window functions allow you to do many common calculations with DataFrames, without having to resort to RDD manipulation.

Aggregates, UDFs vs. Window functions

Window functions are complementary to existing DataFrame operations: aggregates, such as sum and avg, and UDFs. To review, aggregates calculate one result, a sum or average, for each group of rows, whereas UDFs calculate one result for each row based on only data in that row. In contrast, window functions calculate one result for each row based on a window of rows. For example, in a moving average, you calculate for each row the average of the rows surrounding the current row; this can be done with window functions.

Moving Average Example

- Let us dive right into the moving average example. In this example dataset, there are two customers who have spent different amounts of money each day.

- // Building the customer DataFrame. All examples are written in Scala with Spark 1.6.1, but the same can be done in Python or SQL.

```
val customers = sc.parallelize(List(("Alice", "2016-05-01", 50.00),
                                    ("Alice", "2016-05-03", 45.00),
                                    ("Alice", "2016-05-04", 55.00),
                                    ("Bob", "2016-05-01", 25.00),
                                    ("Bob", "2016-05-04", 29.00),
                                    ("Bob", "2016-05-06", 27.00))).
                               toDF("name", "date", "amountSpent")
```

Moving Average Example

// Import the window functions.

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._
```

// Create a window spec.

```
val wSpec1 =
Window.partitionBy("name").orderBy("date").rowsBetween(-1, 1)
```

- In this window spec, the data is partitioned by customer. Each customer's data is ordered by date. And, the window frame is defined as starting from -1 (one row before the current row) and ending at 1 (one row after the current row), for a total of 3 rows in the sliding window.

Moving Average Example

// Calculate the moving average

```
customers.withColumn( "movingAvg",
                      avg(customers("amountSpent")).over(wSpec1) ).show()
```

This code adds a new column, “movingAvg”, by applying the avg function on the sliding window defined in the window spec:

name	date	amountSpent	movingAvg
Alice	5/1/2016	50	47.5
Alice	5/3/2016	45	50
Alice	5/4/2016	55	50
Bob	5/1/2016	25	27
Bob	5/4/2016	29	27
Bob	5/6/2016	27	28

Window function and Window Spec definition

- As shown in the above example, there are two parts to applying a window function: (1) specifying the window function, such as avg in the example, and (2) specifying the window spec, or wSpec1 in the example. For (1), you can find a full list of the window functions here:
- https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.functions$
- You can use functions listed under “Aggregate Functions” and “Window Functions”.

- For (2) specifying a window spec, there are three components: partition by, order by, and frame.

1. “Partition by” defines how the data is grouped; in the above example, it was by customer. You have to specify a reasonable grouping because all data within a group will be collected to the same machine. Ideally, the DataFrame has already been partitioned by the desired grouping.
2. “Order by” defines how rows are ordered within a group; in the above example, it was by date.
3. “Frame” defines the boundaries of the window with respect to the current row; in the above example, the window ranged between the previous row and the next row.

Cumulative Sum

Next, let us calculate the cumulative sum of the amount spent per customer.

// Window spec: the frame ranges from the beginning (Long.MinValue) to the current row (0).

```
val wSpec2 =
Window.partitionBy("name").orderBy("date").rowsBetween(Long.MinValue, 0)
```

// Create a new column which calculates the sum over the defined window frame.

```
customers.withColumn( "cumSum",
  sum(customers("amountSpent")).over(wSpec2)  ).show()
```

name	date	amountSpent	cumSum
Alice	5/1/2016	50	50
Alice	5/3/2016	45	95
Alice	5/4/2016	55	150
Bob	5/1/2016	25	25
Bob	5/4/2016	29	54
Bob	5/6/2016	27	81

Data from previous row

In the next example, we want to see the amount spent by the customer in their previous visit.

```
// Window spec. No need to specify a frame in this case.
val wSpec3 = Window.partitionBy("name").orderBy("date")

// Use the lag function to look backwards by one row.
customers.withColumn("prevAmountSpent",
 lag(customers("amountSpent"), 1).over(wSpec3) ).show()
```

name	date	amountSpent	prevAmountSpent
Alice	5/1/2016	50	null
Alice	5/3/2016	45	50
Alice	5/4/2016	55	45
Bob	5/1/2016	25	null
Bob	5/4/2016	29	25
Bob	5/6/2016	27	29

Rank

- In this example, we want to know the order of a customer's visit (whether this is their first, second, or third visit).

// The rank function returns what we want.

```
customers.withColumn( "rank", rank().over(wSpec3) ).show()
```

name	date	amountSpent	rank
Alice	5/1/2016	50	1
Alice	5/3/2016	45	2
Alice	5/4/2016	55	3
Bob	5/1/2016	25	1
Bob	5/4/2016	29	2
Bob	5/6/2016	27	3

Case Study: Twitter Sentiment Analysis with Spark Streaming

Case Study: Twitter Sentiment Analysis

- Trending Topics can be used to create campaigns and attract larger audience. Sentiment Analytics helps in crisis management, service adjusting and target marketing.
- Sentiment refers to the emotion behind a social media mention online.
- Sentiment Analysis is categorising the tweets related to particular topic and performing data mining using Sentiment Automation Analytics Tools.
- We will be performing Twitter Sentiment Analysis as an Use Case or Spark Streaming.

Figure: *Facebook And Twitter Trending Topics*

Problem Statement

- To design a Twitter Sentiment Analysis System where we populate real-time sentiments for crisis management, service adjusting and target marketing.

Sentiment Analysis is used to:

- Predict the success of a movie
- Predict political campaign success
- Decide whether to invest in a certain company
- Targeted advertising
- Review products and services

Importing Packages

```
//Import the necessary packages into the Spark Program
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.SparkContext._
import org.apache.spark.streaming.twitter._
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark._
import org.apache.spark.rdd._
import org.apache.spark.rdd.RDD
import org.apache.spark.SparkContext._
import org.apache.spark.sql
import org.apache.spark.storage.StorageLevel
import scala.io.Source
import scala.collection.mutable.HashMap
import java.io.File
```

Twitter Token Authorization

```
object mapr {

 def main(args: Array[String]) {
 if (args.length < 4) {
 System.err.println("Usage: TwitterPopularTags <consumer key>
<consumer secret> " +
 "<access token> <access token secret> [<filters>]")
 System.exit(1)
 }

 StreamingExamples.setStreamingLogLevels()
 //Passing our Twitter keys and tokens as arguments for authorization
 val Array(consumerKey, consumerSecret, accessToken,
accessTokenSecret) = args.take(4)
 val filters = args.takeRight(args.length - 4)
```

DStream Transformation

```
// Set the system properties so that Twitter4j library used by twitter stream
// Use them to generate OAuth credentials
System.setProperty("twitter4j.oauth.consumerKey", consumerKey)
System.setProperty("twitter4j.oauth.consumerSecret", consumerSecret)
System.setProperty("twitter4j.oauth.accessToken", accessToken)
System.setProperty("twitter4j.oauth.accessTokenSecret",
accessTokenSecret)

val sparkConf = new
SparkConf().setAppName("Sentiments").setMaster("local[2]")
val ssc = new StreamingContext(sparkConf, Seconds(5))
val stream = TwitterUtils.createStream(ssc, None, filters)

//Input DStream transformation using flatMap
val tags = stream.flatMap { status =>
status.getHashtagEntities.map(_.getText)}
```

Results

Markers Properties Servers Data Source Explorer Snippets Console Scala Interpreter (TwitterStreaming)

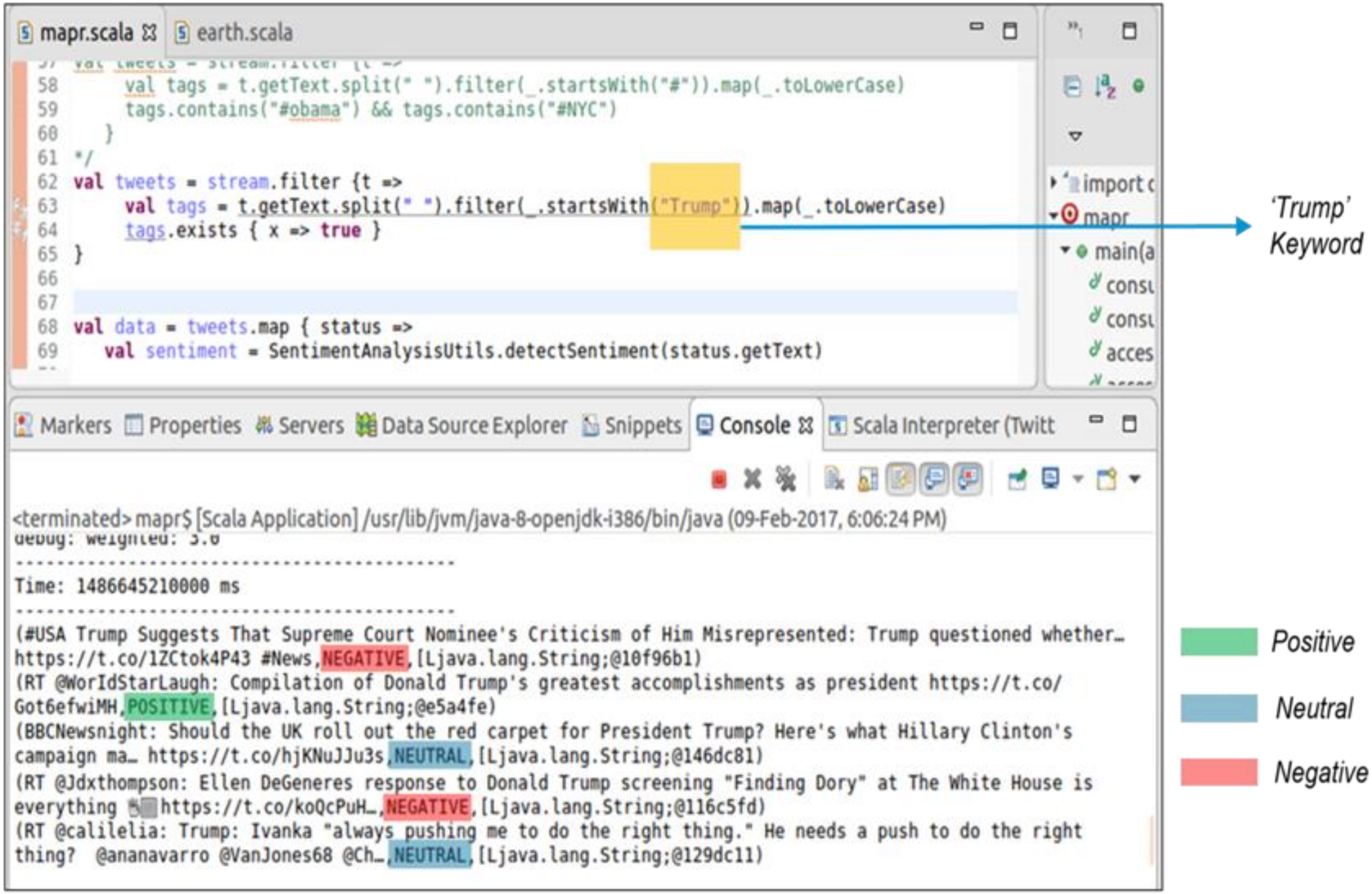
```
<terminated> mapr$ [Scala Application] /usr/lib/jvm/java-8-openjdk-i386/bin/java (09-Feb-2017, 11:56:26 AM)
debug: weighted: 1.0
-------------------------------------------
Time: 1486621640000 ms
-------------------------------------------
(東芝、半導体新棟を着工=メモリー製造、18年夏完成へ https://t.co/DU5goZAp25 #不動産 #投資 #マネー #株 #市況 #拡散,NEGATIVE,[Ljava.lang.String;@1a25ec3)
(RT @bts_bighit: [투표] Q. 투표하라는 말 지겹다?
아미: 좋아요~ 짜릿해!⚡ 늘 새로워! #방탄소년단 투표하는 게 최고야!

#가온차트어워드 https://t.co/OHDWR2smt4
#ShortyAwards https://t.…,NEGATIVE,[Ljava.lang.String;@121986a)
(RT @MukePL: Jeżeli na tym zdjęciu widzisz swój świat to daj RT. ♡ #oneDbestfans & #5SOSbestfans ♡ https://t.co/rn2EmNvjFp,NEGATIVE,[Ljava.lang.String;@1c3681d)
(RT @Horocasts: #Cancer most enduring quality is an unexpected silly sense of humor.,POSITIVE,[Ljava.lang.String;@174e1a2)
(I'm listening to "A Song For Mama" by @BoyzIIMen on @PandoraMusic. #pandora https://t.co/71n5Rw3CY0,NEUTRAL,[Ljava.lang.String;@95f6d4)
('Greenwashing' Costing Walmart $1 Million https://t.co/D8XO2RZMnP #Biodegradability #Compostability #biobased,NEGATIVE,[Ljava.lang.String;@1511e25)
(RT @camilasxdinah: Serayah representando a las camilizers   cuando un hombre se le acerca a Camila #CamilaBestFans https://t.co/8IggLo3RGn,NEGATIVE,[Ljava.lang.String;@78c835)
(RT @CamiIaVoteStats: #CamilaBestFans https://t.co/qsLxPQpD1n,NEUTRAL,[Ljava.lang.String;@16e7255)
(@tos 六甲道駅 https://t.co/0rKl8rlSb3 #TFB,NEGATIVE,[Ljava.lang.String;@1a3fe)
(Ilmar pro Marcos: "Vai dormir puta.. Bebe e fica aí com o cu quente." KKKKKKKKKKKKKKKKKKKKKKKK #BBB17,NEGATIVE,[Ljava.lang.String;@1516ece)
...

Adding annotator tokenize
```

Positive

Neutral

Negative

Sentiment for Trump



Applying Sentiment Analysis

- As we have seen from our Sentiment Analysis demonstration, we can extract sentiments of particular topics just like we did for ‘Trump’. Similarly, Sentiment Analytics can be used in crisis management, service adjusting and target marketing by companies around the world.

- Companies using Spark Streaming for Sentiment Analysis have applied the same approach to achieve the following:

1. Enhancing the customer experience
2. Gaining competitive advantage
3. Gaining Business Intelligence
4. Revitalizing a losing brand

References

- https://spark.apache.org/streaming/
- Streaming programming guide – spark.incubator.apache.org/docs/latest/streaming-programming-guide.html
- https://databricks.com/speaker/tathagata-das

Conclusion

- 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