CSE 4/587
Data Intensive Computing

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Dr. Shamshad Parvin
shamsadp@buffalo.edu
313 Davis Hall

Spark Streaming

References

- **Spark Streaming Programming Guide**
- http://spark.apache.org/docs/latest/streaming-programming-guide.html
- **Apache Spark documentation**
- ○ http://spark.apache.org/
- **Advanced Analytics with Spark** by S. Ryza, U. Laserson, S. Owen and J. Wills
- **Data brick website :**

https://www.databricks.com/glossary/what-is-spark-streaming
- **Discretized Streams: A Fault-Tolerant Model for Scalable Stream Processing,** *Matei Zaharia ,Tathagata Das et al.*

https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-259.pdf

What Is Spark?

- Parallel execution engine for big data processing
- Easy to use: 2-5x less code than Hadoop MR
  - High level API's in Python, Java, and Scala
- Fast: up to 100x faster than Hadoop MR
  - Can exploit in-memory when available
  - Low overhead scheduling, optimized engine
- **General**: support multiple computation models

A Short History

- Started at UC Berkeley in 2009
- Open Source: 2010
- Apache Project: 2013
- Today: most popular big data processing engine

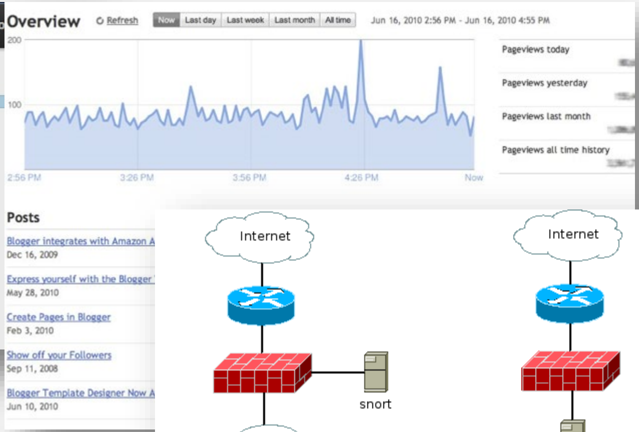
Interactive Streaming Vs Batch

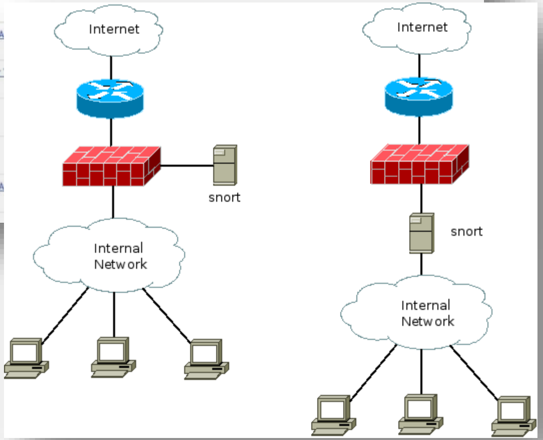
- **Interactive Streaming** : Continuous flow of data/event
  example: Tweet event , stream of log messages

- **Batch :** Data collected in regular interval of time

Motivation

- Many important applications must process large streams of live data and provide results in near-real-time
  - Social network trends
  - Website statistics
  - Intrusion 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
- Achieves Low latencies
- Simple programming model
- Efficient recovery from failure
- Integrates batch and interactive processing

Why MR is not a Solution for streaming Real Data

- Great for large amounts of static data
  - – Data is not moving!

- For streams: only for large windows
  - High latency, low efficiency

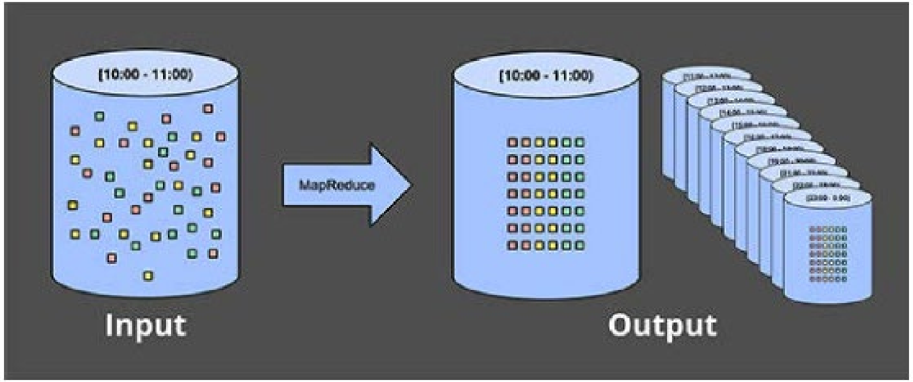


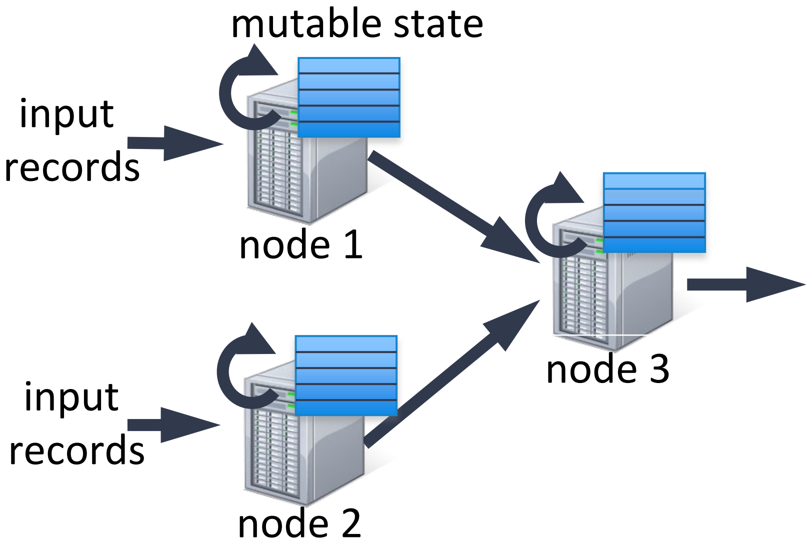
Figure : Tyler Akidau

What people have been doing ?

- 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
- Build two stacks – one for batch, one for streaming
- Extremely painful to maintain two different stacks
  - Different programming models
  - Doubles implementation effort
  - Doubles operational effort

Fault-tolerant Stream Processing

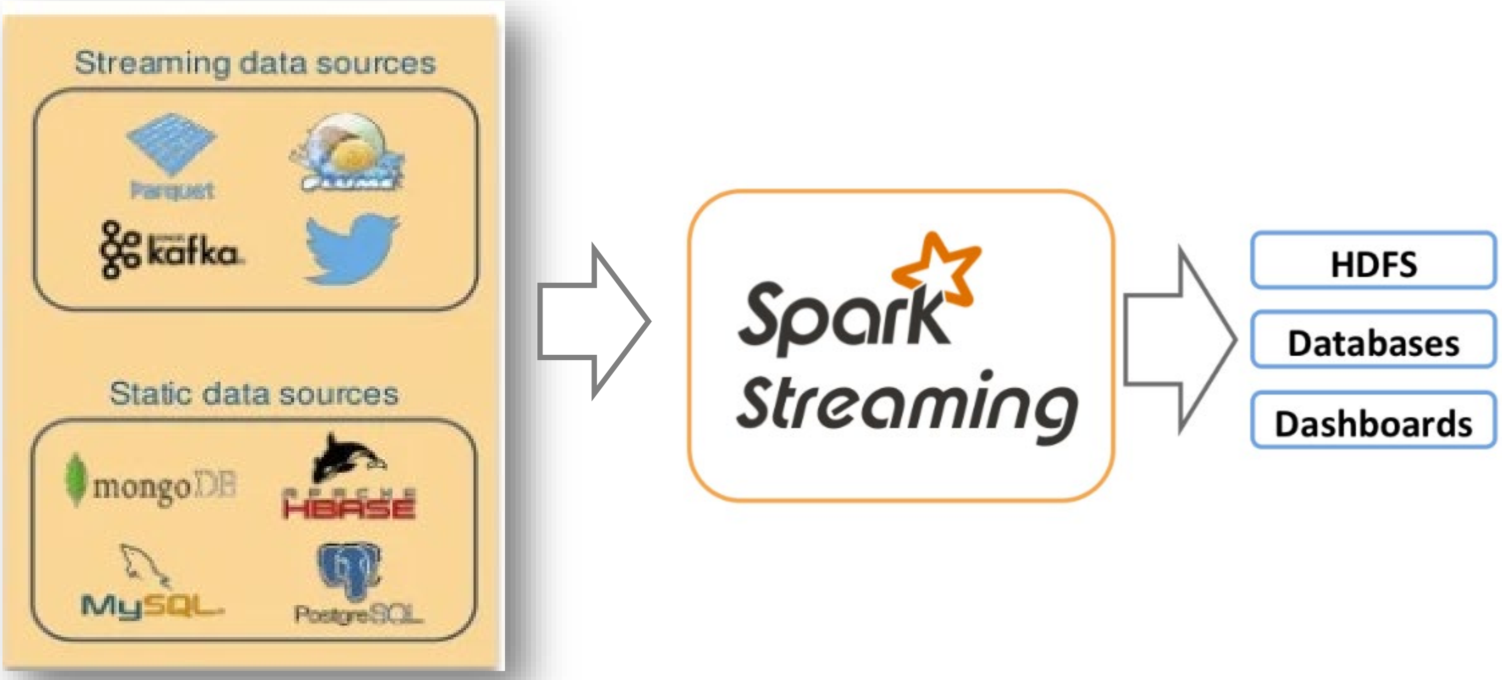
- Traditional processing model
  - --Pipeline of nodes
  - – Each node maintains mutable state
  - – Each input record updates the state
  - --and new records are sent out
- Mutable state is lost if node fails
- Making stateful stream processing
- Fault-tolerant is challenging!



What is Spark Streaming?

- Spark Streaming is a scalable fault-tolerant streaming processing system that natively supports both batch and streaming workloads.
- It is an extension of the Spark API that process live data stream in a real time

Data Source of Spark Streaming



Data Source of Spark Streaming

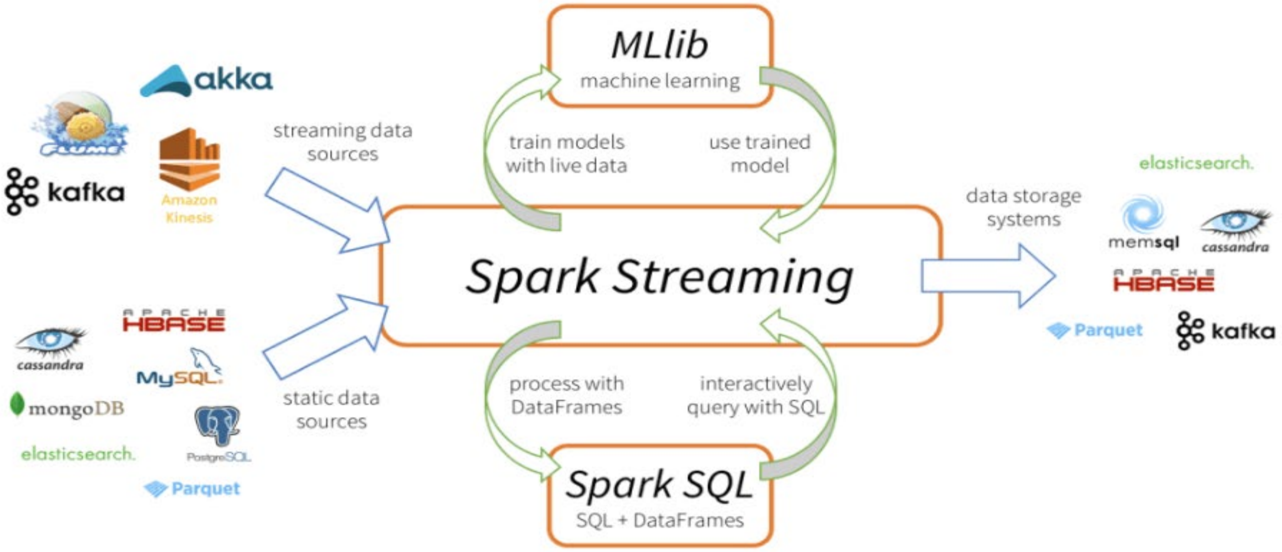


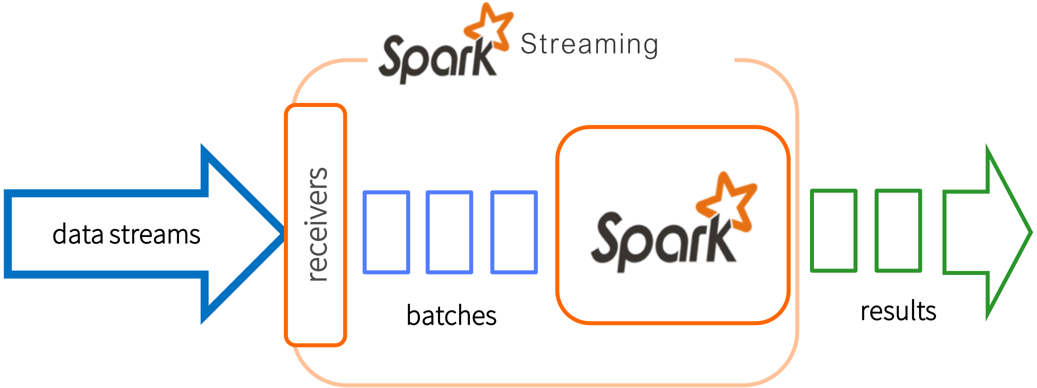
Image : databricks

Major Aspects of Spark Streaming

- Fast recovery from failures and stragglers
- Better load balancing and resource usage
- Combining of streaming data with static datasets and interactive queries
- Native integration with advanced processing libraries (SQL, machine learning, graph processing)

How does it work?

- Data streams are chopped into batches of few secs
- SPARK treats each batches of data s RDDs and process them using RDD operator
- Each batch is processed in Spark
- Results pushed out in batches



Spark Streaming Programming Model

Discretized Stream (DStream)

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

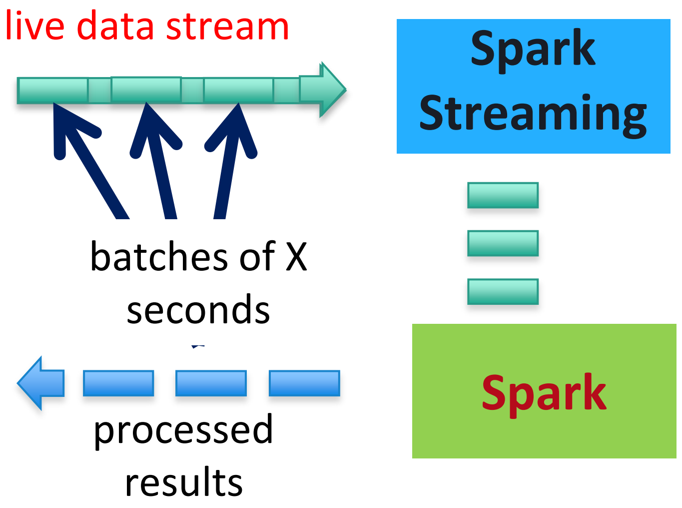
DStreams API very similar to RDD API

- Create input DStreams from different sources
- Apply parallel operations

Discretized Stream Processing

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

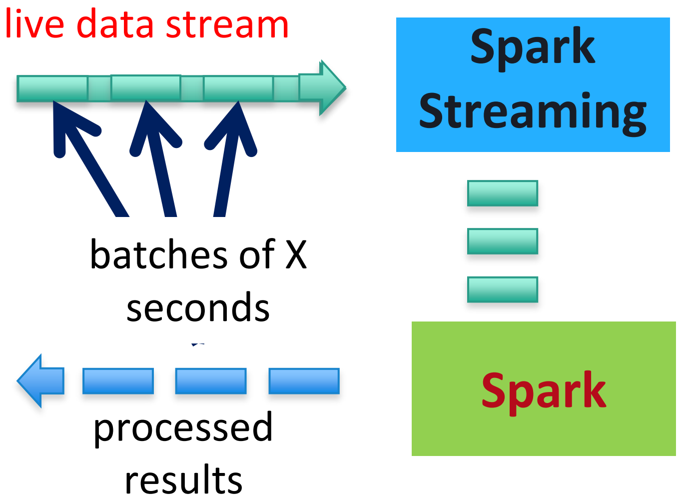
- Chop up the live stream into batches of X seconds
- Spark treats each batch of data as RDDs and processes them using RDD operations
- Finally, the processed results of the RDD operations are returned in batches



Discretized Stream Processing

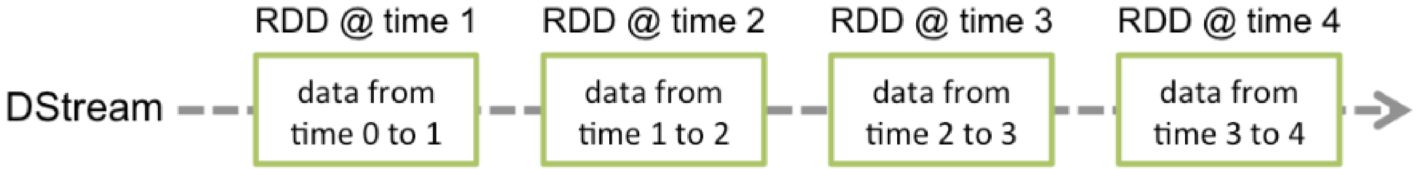
Run a streaming computation as a **series of very small, deterministic batch jobs**

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



DStream

- The basic high-level abstraction for streaming in spark is called DStream or discretized stream
- Dstream can be created
- --from input data streams from sources such as kafka,flume and Kinesis
- A Dstream is represented as a sequence of RDDs.



Streaming Context

- A **StreamingContext** object has to be created which is the main entry point of all Spark Streaming functionality.
- A **StreamingContext** object can be created from a Sparkcontext object.
- Define the input sources by creating input DStreams.
- Define the streaming computations by applying transformation and output operations to DStreams.
- Start receiving data and processing it using (start ())
- Wait for the processing to be stopped (manually or due to any error) (awaitTermination ())

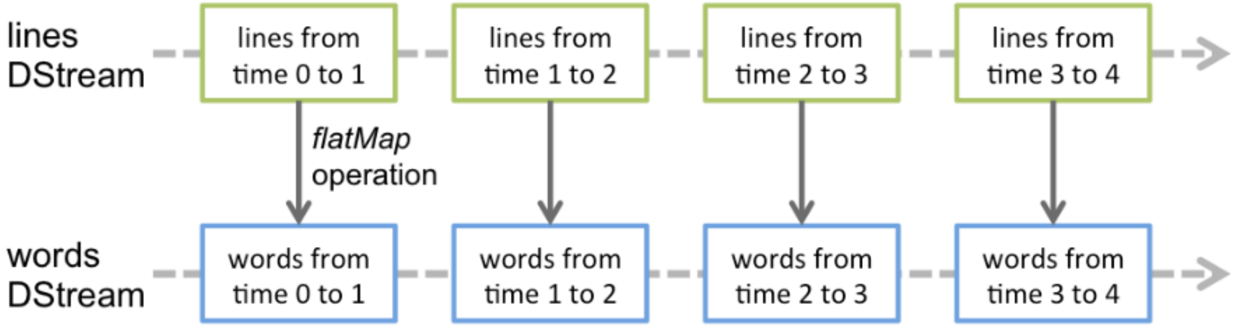
The processing can be manually stopped using, (stop ())

Streaming Context

```python
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext(master, appName)
ssc = StreamingContext(sc, 1)
```

Operations Applied on DStream

- Any operation applied on a DStream translates to operations on the underlying RDDs.
- converting a stream of lines to words, by applying the operation ‘flatmap’ on each RDD in the lines DStream



Input DStreams and Receivers

- Input DStreams are DStreams representing the stream of input data received f
- Spark Streaming provides two categories of built-in streaming built-in streaming sources.
- *Basic sources*: Sources directly available in the StreamingContext API. Examples: file systems, and socket connections.
- *Advanced sources*: Sources like Kafka, Kinesis, etc. are available through extra utility classes.

Input DStreams and Receivers

two kinds of receivers:

1. *Reliable Receiver* - A *reliable receiver* correctly sends acknowledgment to a reliable source when the data has been received and stored in Spark with replication.
2. *Unreliable Receiver* - An *unreliable receiver* does *not* send acknowledgment to a source. This can be used for sources that do not support acknowledgment, or even for reliable sources when one does not want or need to go into the complexity of acknowledgment.

Files Streams

- Besides sockets, the StreamingContext API provides methods for creating DStreams from files
- Reading data from files on any file system compatible with the HDFS API (that is, HDFS, S3, NFS, etc.)
- Spark Streaming will monitor a directory and process any files created in that directory

Transformations on DStreams

- Similar to that of RDDs, transformations allow the data from the input DStream to be modified. DStreams support many of the transformations available on normal Spark RDD's.

Transformation	Meaning
map(*func*)	Return a new DStream by passing each element of the source DStream through a function *func*.
flatMap(*func*)	Similar to map, but each input item can be mapped to 0 or more output items.
filter(*func*)	Return a new DStream by selecting only the records of the source DStream on which *func* returns true.
repartition(*numPartitions*)	Changes the level of parallelism in this DStream by creating more or fewer partitions.
union(*otherStream*)	Return a new DStream that contains the union of the elements in the source DStream and *otherDStream*.
count()	Return a new DStream of single-element RDDs by counting the number of elements in each RDD of the source DStream.
reduce(*func*)	Return a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function *func* (which takes two arguments and returns one). The function should be associative and commutative so that it can be computed in parallel.

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

DStream Example

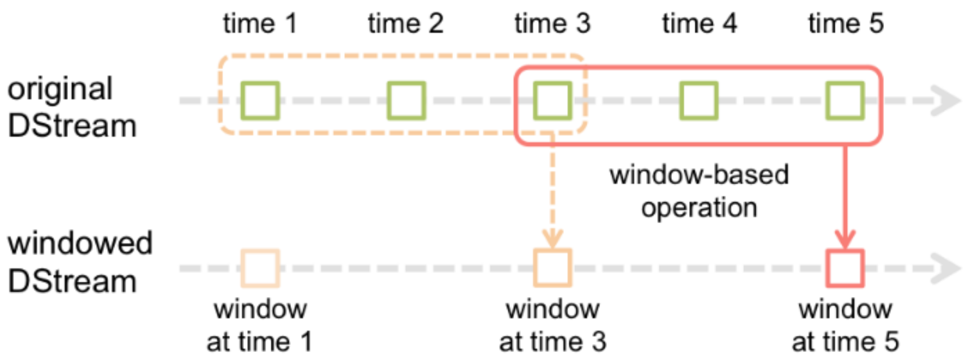
```
// Create a DStream that will connect to a server
// listening on a TCP socket, say <IP>:9990
val ssc = new StreamingContext(conf, Seconds(5))
val lines = ssc.socketTextStream("<Some_IP>", 9990)

// Word count again
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word.trim, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()

// Start the computation
ssc.start()
// Wait for the application to terminate
ssc.awaitTermination()
// ssc.stop() forces application to stop
```

Stateless vs Stateful Operations

- By design streaming operators are stateless they know nothing about any previous batches
- Stateful operations have a dependency on previous batches of data continuously accumulate metadata overtime

Windowed Stream processing

- Spark Streaming allows you to apply transformations over a sliding window of data knows as *windowed computations*,

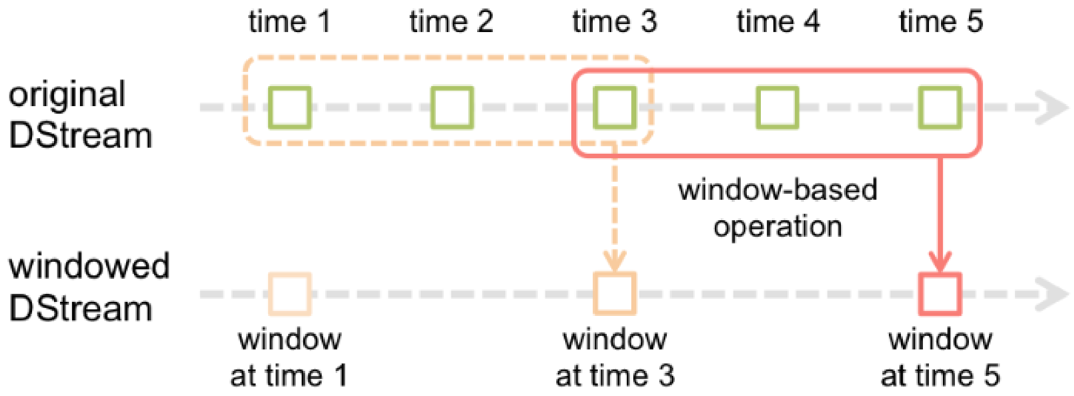


Windowed Stream processing

Any window operation needs to specify two parameters:

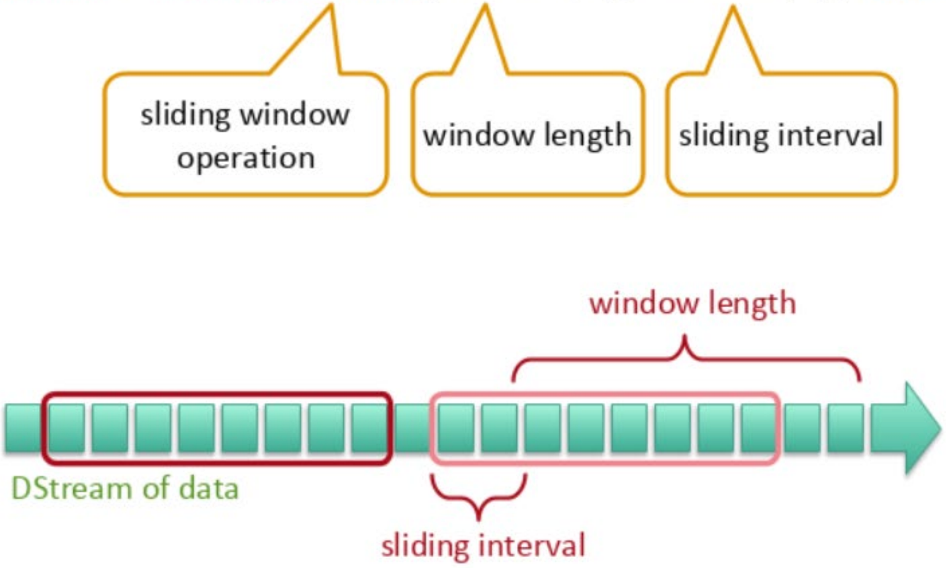
window length - The duration of the window (3 in the figure).

sliding interval - The interval at which the window operation is performed (2 in the figure).



Window Based Transformtion

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



Real World Application Example

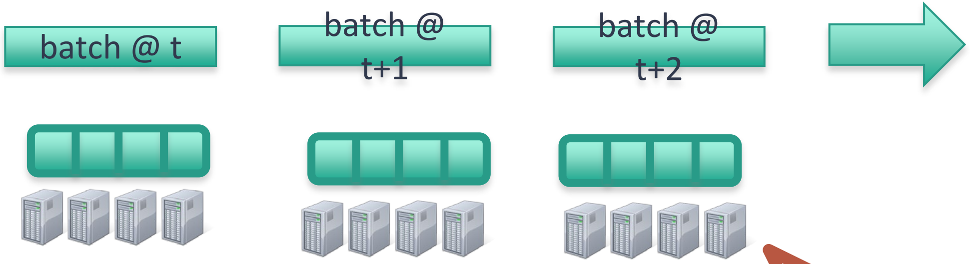
Example 1 – Get hashtags from Twitter

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

DStream: a sequence of RDD representing a stream of data

Twitter Streaming API

tweets DStream



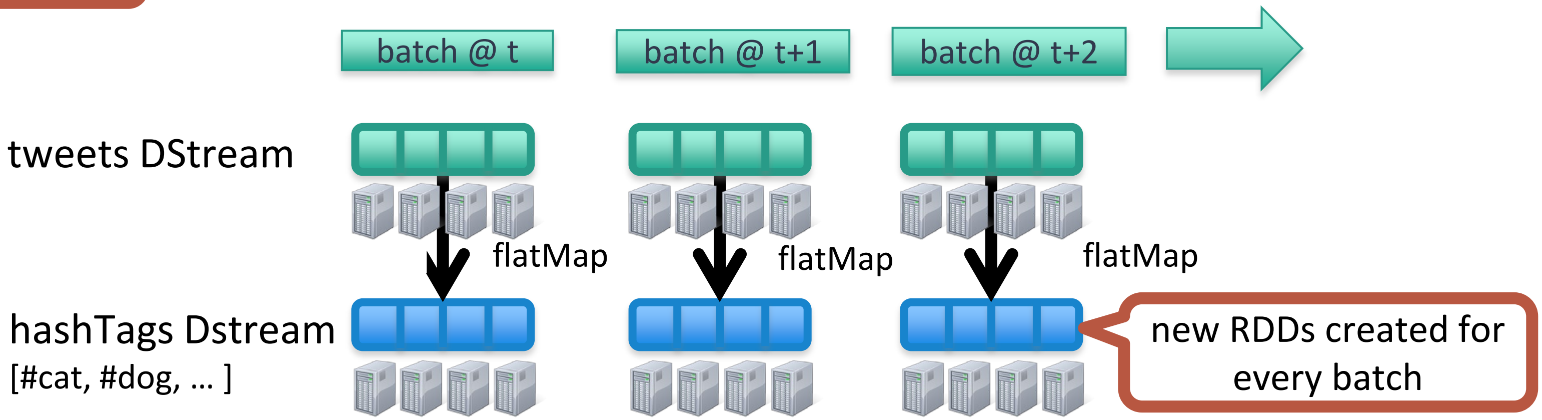
stored in memory as an RDD (immutable, distributed)

Example 1 – Get hashtags from Twitter

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

new DStream

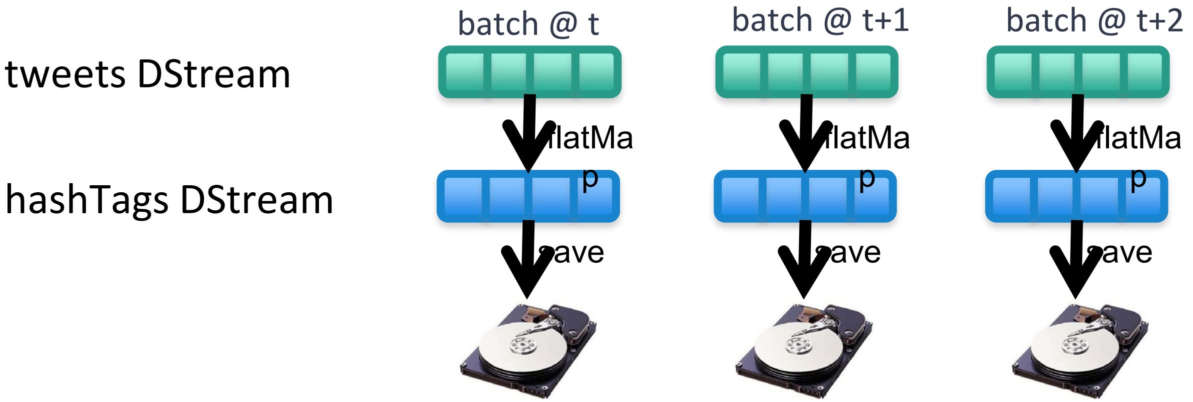
transformation: modify data in one Dstream to create another DStream



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



every batch saved to HDFS