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.*

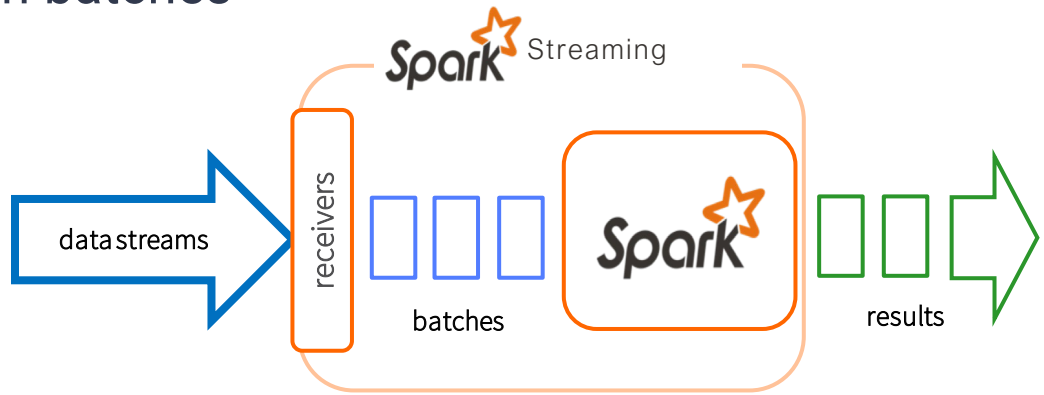
https://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-259.pdf

What is Spark 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

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

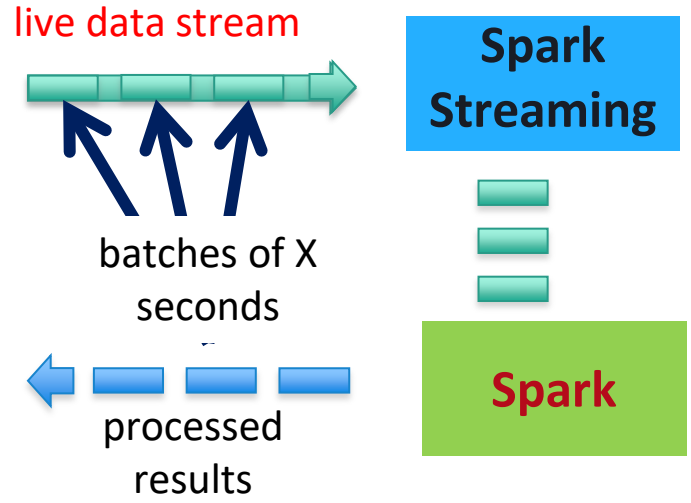
Discretized Stream Processing

- The model of spark streaming treats streaming data as a series of deterministic batch computations on discrete time intervals.
- The data received in each interval is stored reliably across the cluster to form an input dataset for that interval.
- Once the time interval completes, this dataset is processed via deterministic parallel operations, such as map, reduce and groupBy to produce new datasets representing either program outputs or intermediate state
- D-Stream is a sequence of immutable, partitioned datasets (specifically RDDs) that can be acted on through deterministic operators.

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



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

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

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

Counting a Page View Example

- Counting of page view events by URL.

PageViews: creates a D-Stream by an event stream over HTTP, and groups these into 1-second intervals.

```
pageViews=readStream("http://...", "1s")
ones = pageViews.map(event => (event.url, 1))
counts = ones.runningReduce((a, b) => a + b)
```

performs a running count of these using a stateful runningReduce operator.

transforms the event stream to get a DStream of (URL, 1) pairs called ones

Counting a Page View Example

The system will launch map tasks every second to process the new events dataset for that second.

Then it will launch reduce tasks that take as input both the results of the maps and the results of the previous interval's reduces, stored in an RDD.

These tasks will produce a new RDD with the updated counts.

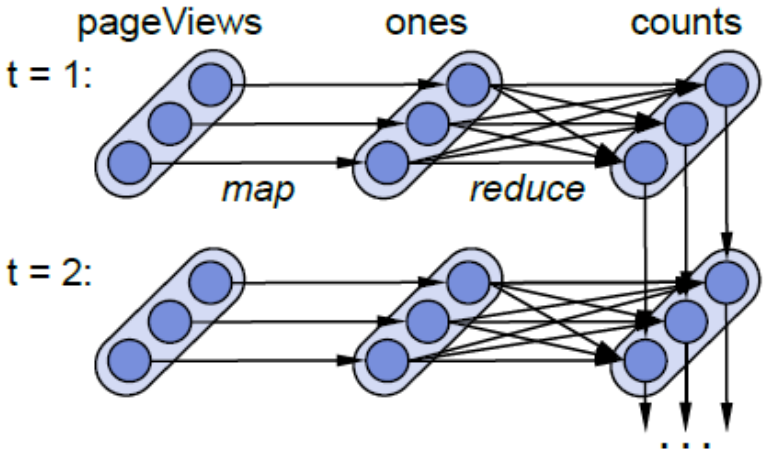
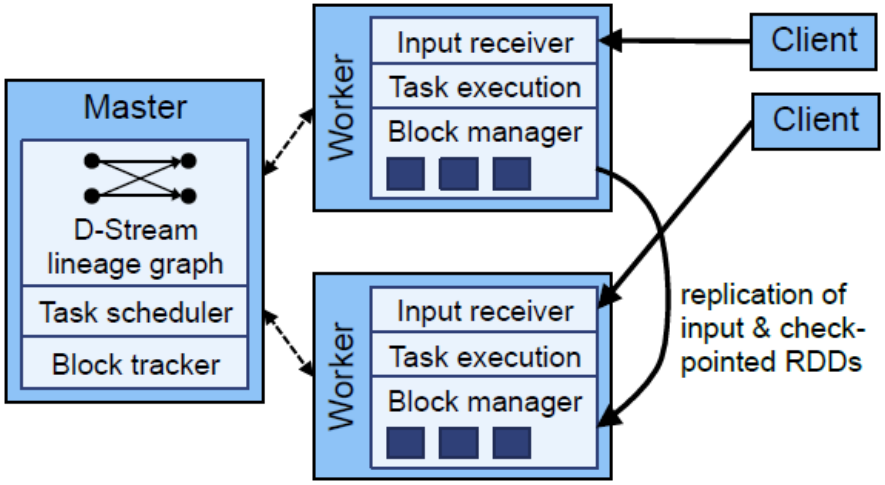


Figure : Lineage graph for RDDs in the view count program. Each oval is an RDD, whose partitions are shown as circles.

Components of Spark Streaming

Spark Streaming consists of three components, shown in Figure:

- A master that tracks the D-Stream lineage graph and schedules tasks to compute new RDD partitions.
- Worker nodes that receive data, store the partitions of input and computed RDDs, and execute tasks.
- A client library used to send data into the system.

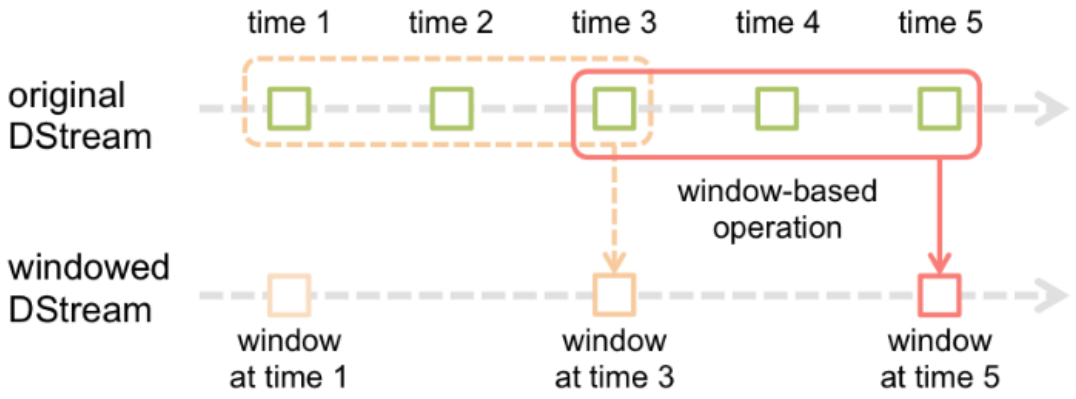


Window Computation on Stream processing

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()
```

Window operations

Transformation	Meaning
window(*windowLength*, *slideInterval*)	Return a new DStream which is computed based on windowed batches of the source DStream.
countByWindow(*windowLength*, *slideInterval*)	Return a sliding window count of elements in the stream.
reduceByWindow(*func*, *windowLength*, *slideInterval*)	Return a new single-element stream, created by aggregating elements in the stream over a sliding interval using *func*. The function should be associative and commutative so that it can be computed correctly in parallel.

Join Operations

- **Stream-stream joins**
- Streams can be very easily joined with other streams

```
stream1 = ...
stream2 = ...
joinedStream = stream1.join(stream2)
```

- Here, in each batch interval, the RDD generated by stream1 will be joined with the RDD generated by stream2

Join Operations

- **Stream-stream joins**
- You can also do `leftOuterJoin`, `rightOuterJoin`, `fullOuterJoin`.
- It is often very useful to do joins over windows of the streams.

```
windowedStream1 = stream1.window(20)
windowedStream2 = stream2.window(60)
joinedStream = windowedStream1.join(windowedStream2)
```

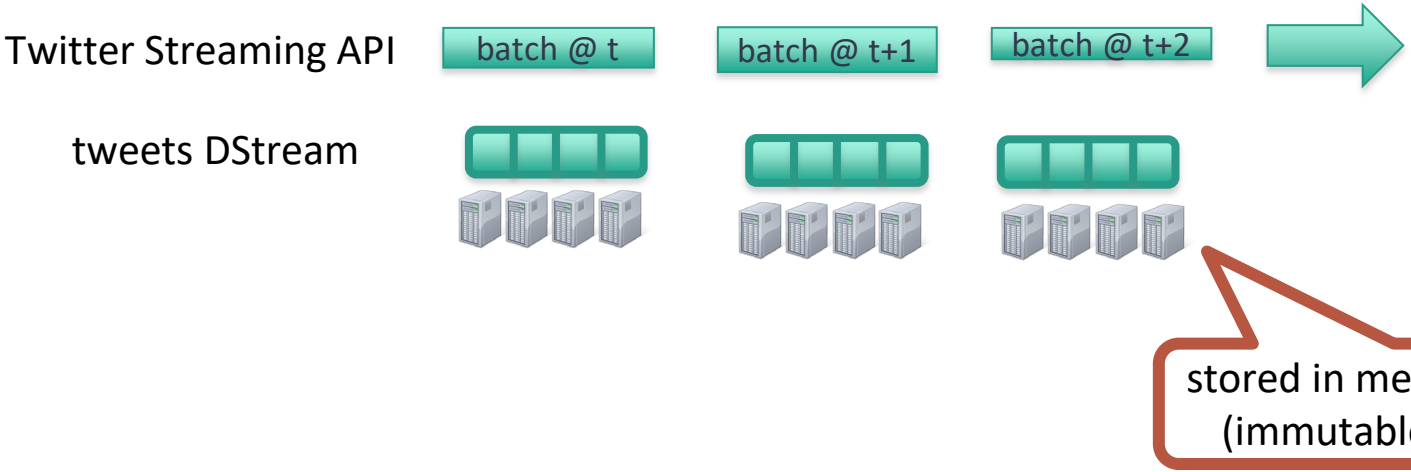
Setting the Right Batch Interval

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

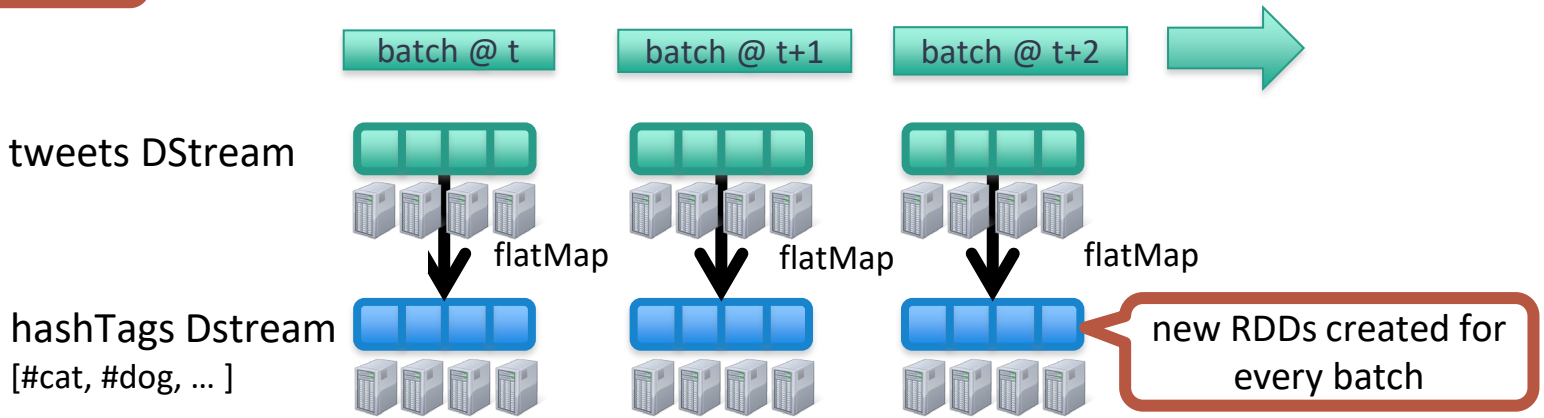


Example 1 – Get hashtags from Twitter

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

new DStream

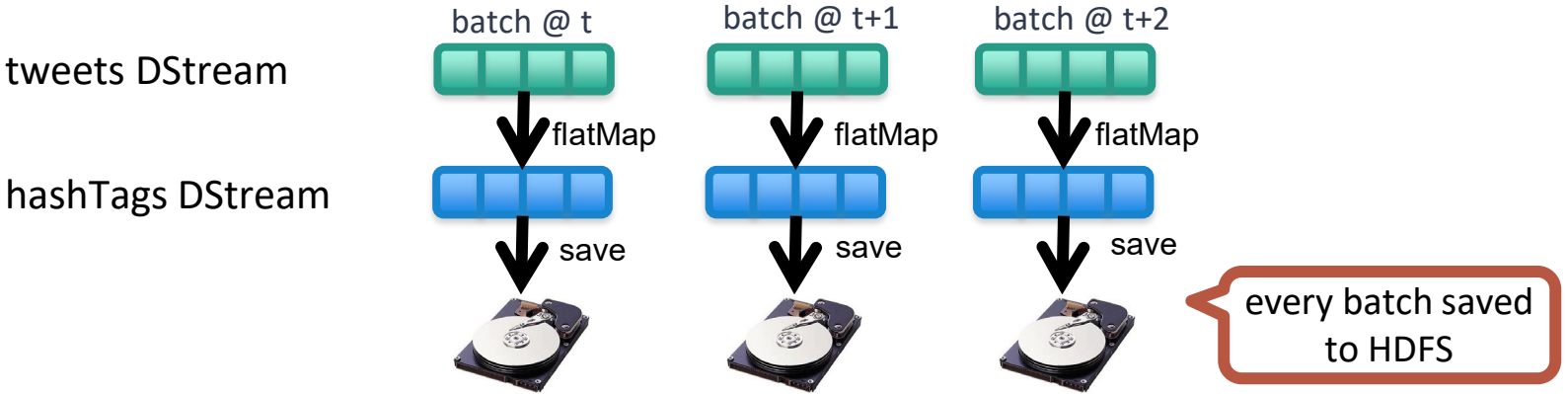
transformation: modify data in one Dstream to create another DStream



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

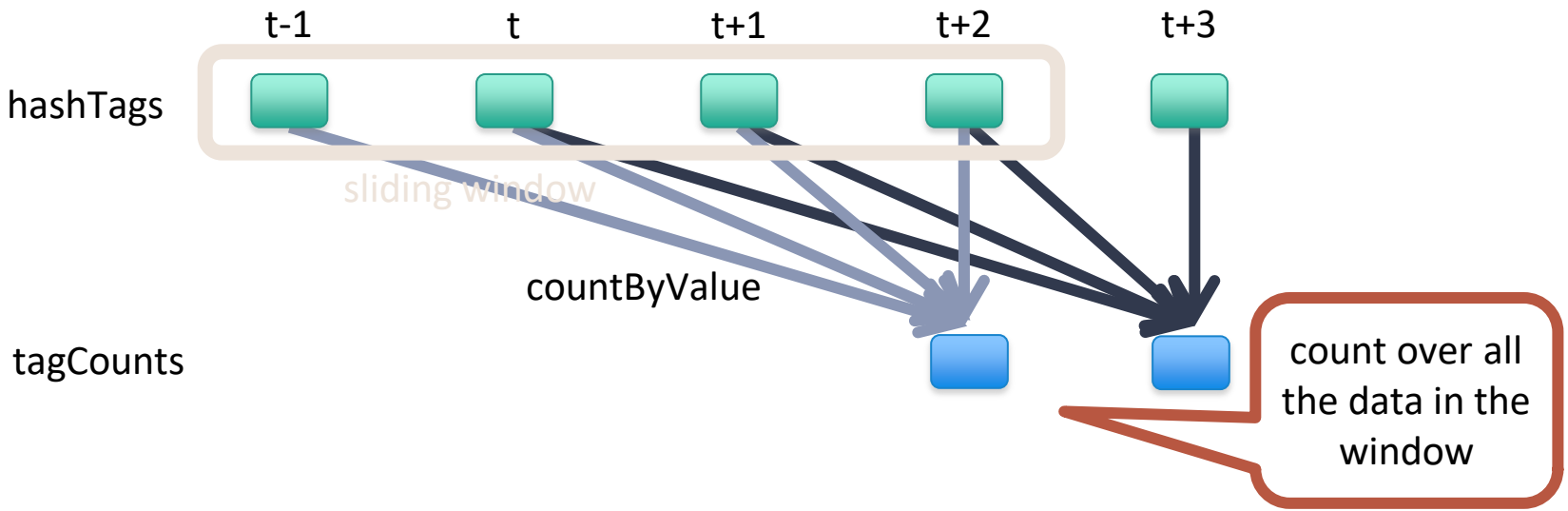


Example 3 – Count the hashtags over last 10 mins

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap (status => getTags(status))
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```

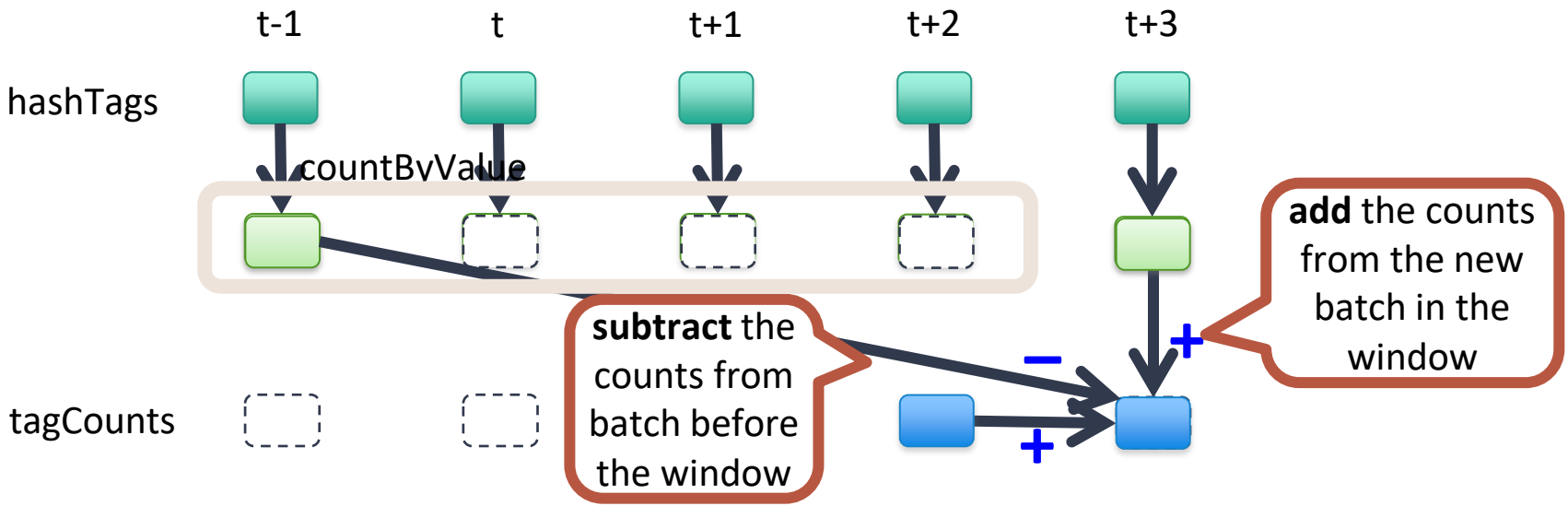
Example 3 – Counting the hashtags over last 10 mins

```
val tagCounts = hashTags.window(Minutes(10), Seconds(1)).countByValue()
```



Smart window-based countByValue

```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```



Fault Tolerant feature of Spark Streaming

Caching/persistence

- Dtreams also allows developers to persist the stream’s data into memory .
- Using persist( ) method , Dstream will automatically persist every RDD of that Dstream in memory
- This is useful if the data in the DStream will be computed multiple times
- DStreams generated by window-based operations are automatically persisted in memory, without the developer calling persist().
- For input streams that receive data over the network (such as, Kafka, sockets, etc.), the default persistence level is set to replicate the data to two nodes for fault-tolerance.

Check pointing Spark Streaming

- Check-pointing is the process to make streaming applications resilient to failure
- To ensure streaming applications operate in 24/7 , we should check-pointenough information for a storage system that is fault tolerant

Two types of data that are checkpointed :

Metadata check-pointing is primarily needed for recovery from driver failures. Includes - configuration, operations, incomplete batches

Data check-pointing

Store generated RDDs to HDFS . This is necessary in some stateful transformations

Check pointing Spark Streaming

- **When to enable Checkpointing**
- Checkpointing must be enabled for applications with any of the following requirements:

--*Usage of stateful transformations*

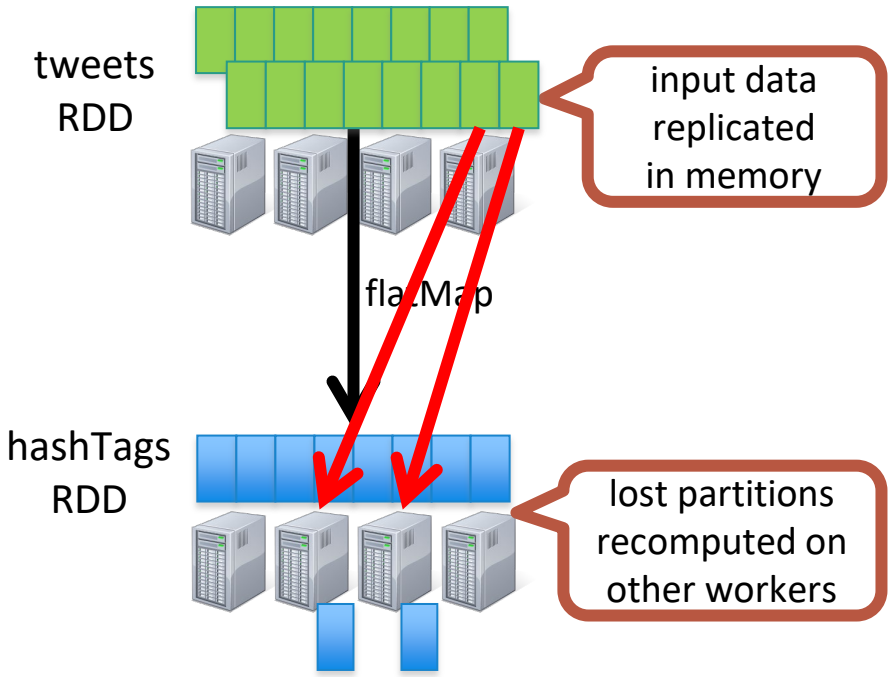
--*Recovering from failures of the driver running the application*

Fault-tolerance :Master

- Master saves the state of the DStreams to a checkpoint file
- -Checkpoiint file saved to HDFS periodically
- If master fails, it can be restarted using the checkpoint file

Fault-tolerance :Worker

- 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

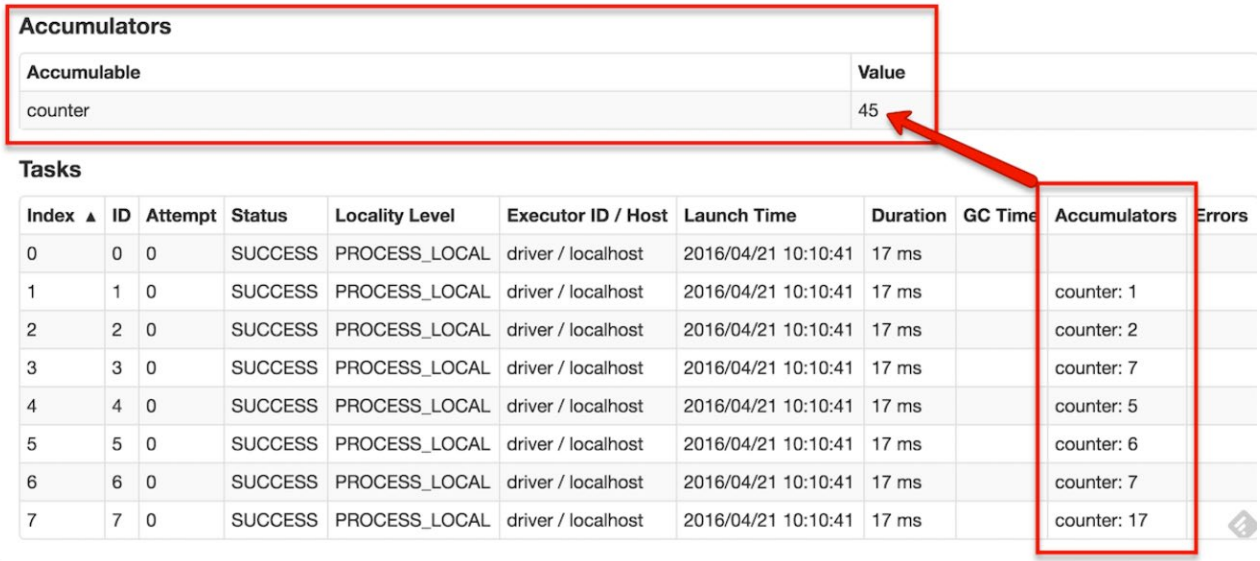


Accumulators in Spark Streaming

- Accumulators are variables that are only “added” to through an associative and commutative operation
- They can be used to implement counters (as in MapReduce) or sums.
- Spark natively supports accumulators of numeric types, and programmers can add support for new types.

Accumulators in Spark Streaming

- A named accumulator (in this instance counter) will display in the web UI for the stage that modifies that accumulator



Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

Tracking accumulators in the UI can be useful for understanding the progress of running stages