

# **Big data: architectures and data analytics**

## **Spark Streaming**

## What is Spark Streaming?

- Spark Streaming is a framework for large scale stream processing
  - Scales to 100s of nodes
  - Can achieve second scale latencies
  - Provides a simple **batch-like** API for implementing complex algorithm
  - Can absorb live data streams from Kafka, Flume, ZeroMQ, Twitter, ...

## What is Spark Streaming?



## 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
  - ...

## Requirements

- Scalable to large clusters
- Second-scale latencies
- Simple programming model
- Efficient fault-tolerance in stateful computations

## Other 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
- Storm Trident
  - Uses transactions to update state
  - Processes each record exactly once
  - Per state transaction updates slow

7

## Spark Streaming

8

## Discretized Stream Processing

- Spark streaming runs a streaming computation as a series of very small, deterministic batch jobs
- It splits each input stream in “portions” and processes one portion at a time (in the incoming order)
  - The same computation is applied on each portion of the stream
  - Each portion is called **batch**

9

## Discretized Stream Processing

- Spark streaming
  - Splits the live stream into batches of X seconds
  - 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



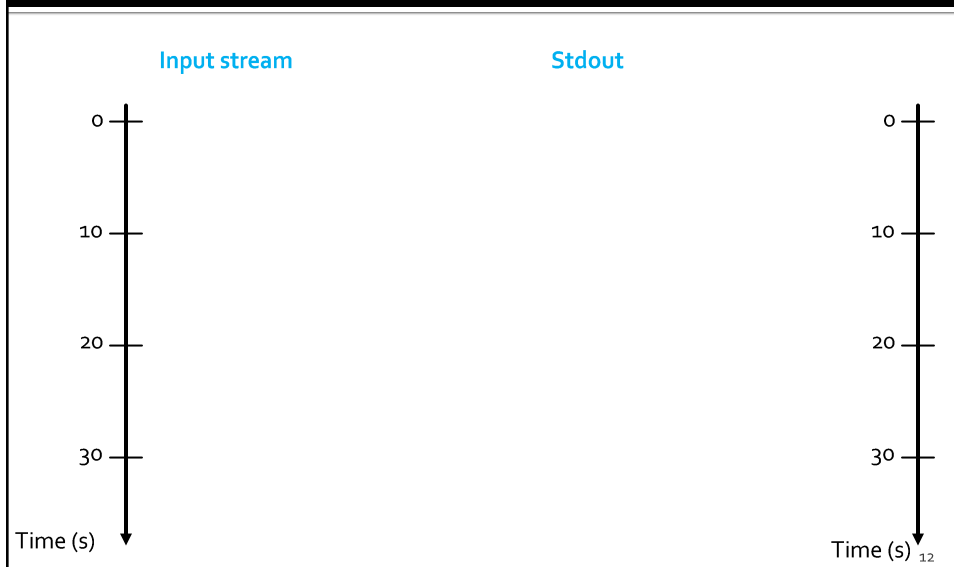
10

## Word count – Spark Streaming version

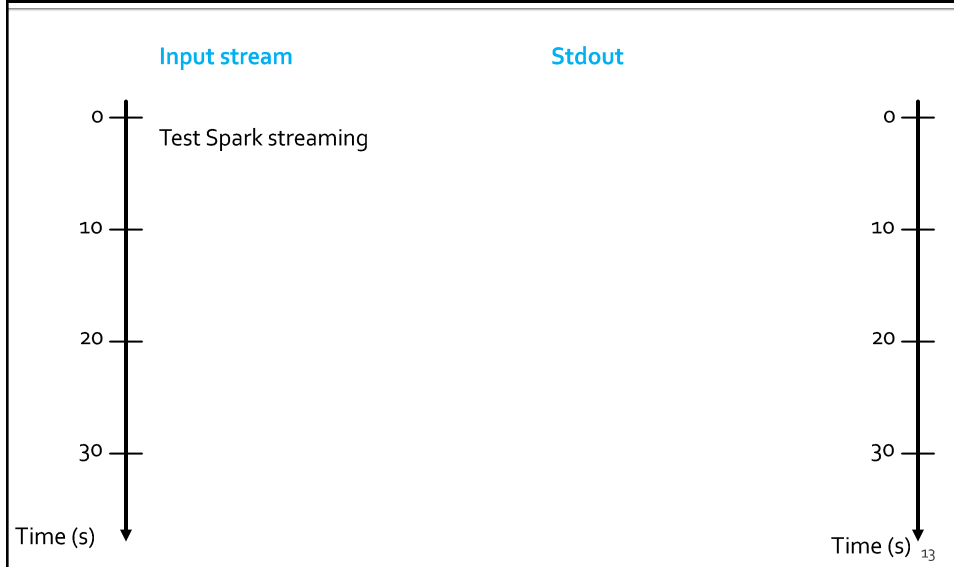
- Problem specification
  - Input: a stream of sentences
  - Split the input stream in batches of 10 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch
    - i.e., execute the word count problem for each batch of 10 seconds

11

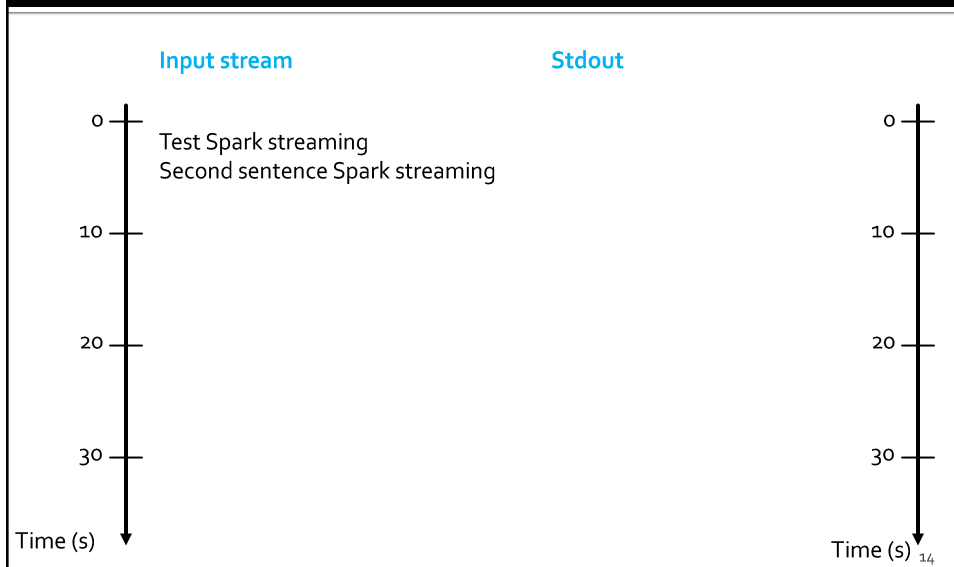
## Word count – Spark Streaming version



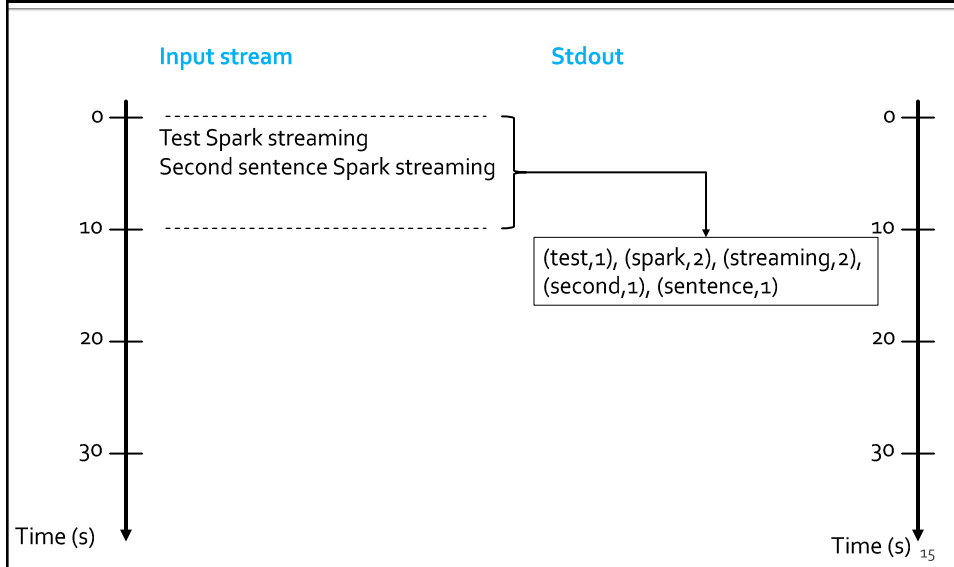
## Word count – Spark Streaming version



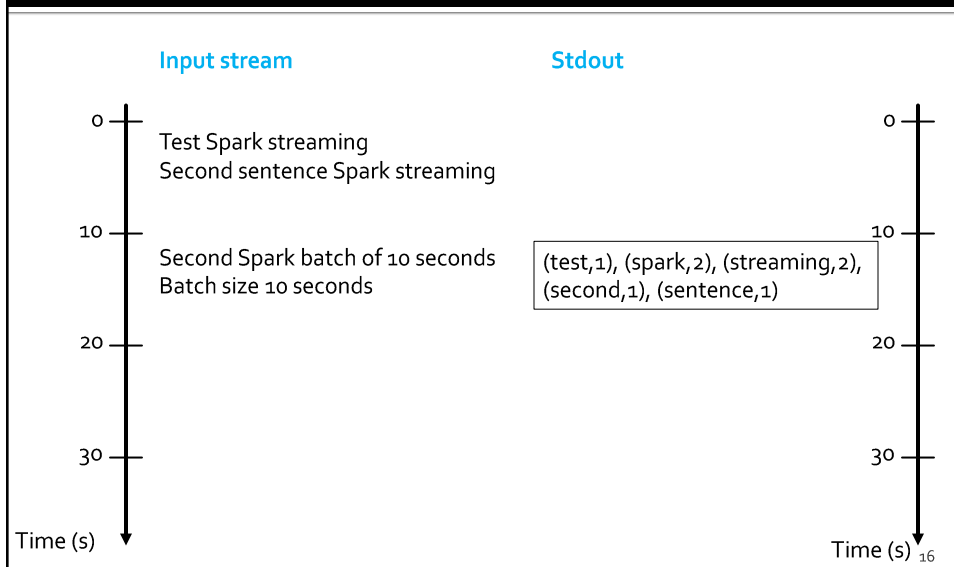
## Word count – Spark Streaming version



## Word count – Spark Streaming version

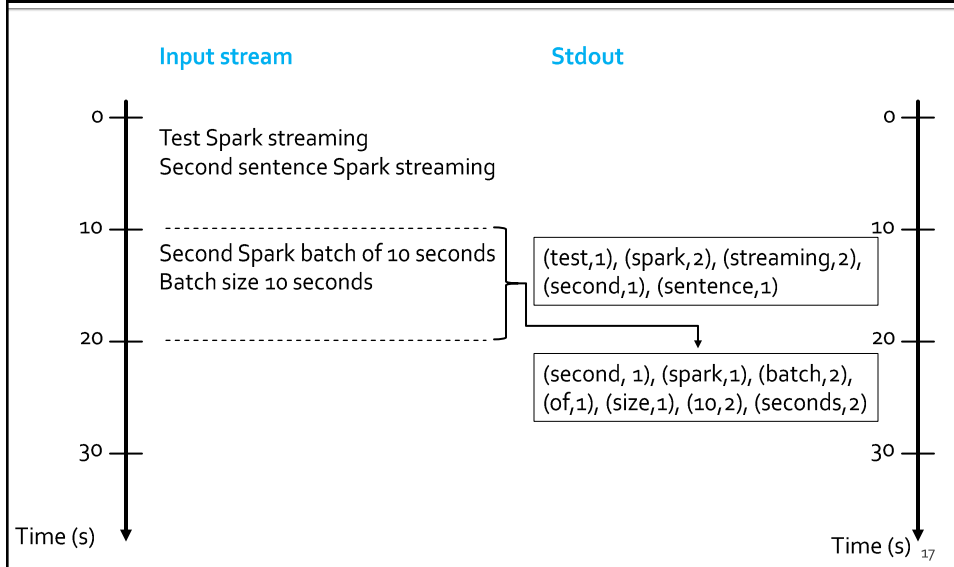


## Word count – Spark Streaming version

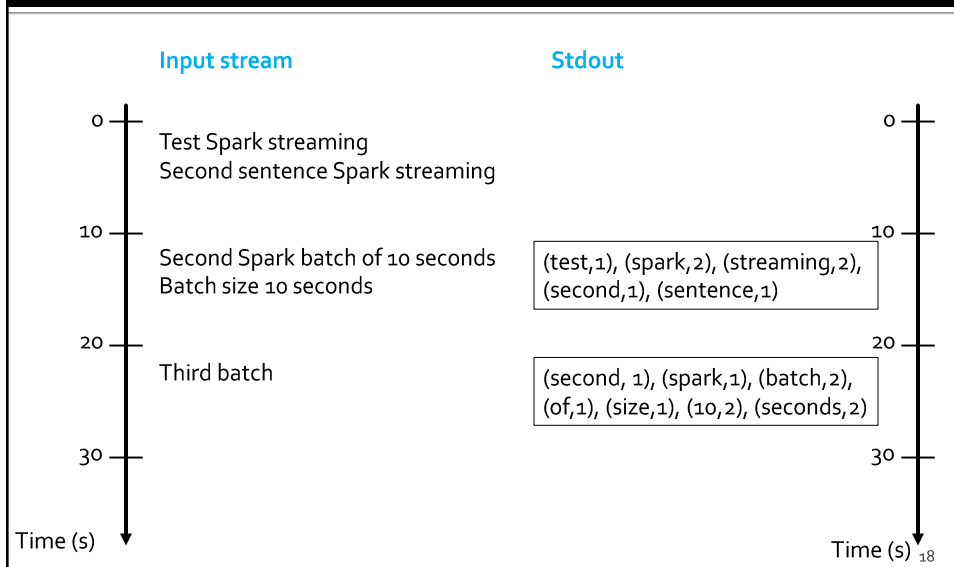




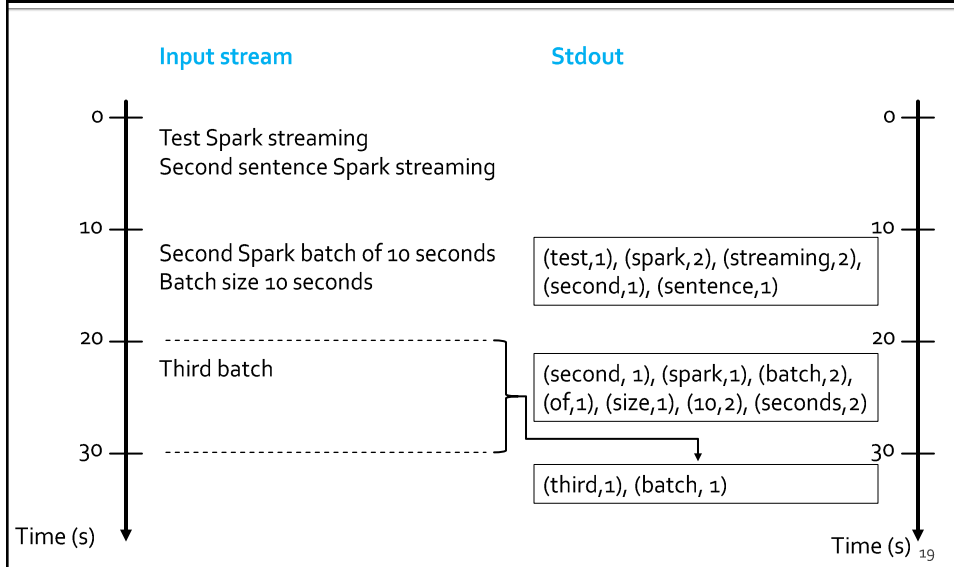
## Word count – Spark Streaming version



## Word count – Spark Streaming version



## Word count – Spark Streaming version



## Key concepts

### ■ DStream

- Sequence of RDDs representing a discretized version of the input stream of data
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets
- One RDD for each batch of the input stream

### ■ PairDStream

- Sequence of PairRDDs representing a stream of pairs

## Key concepts

### ■ Transformations

- Modify data from one DStream to another
- Standard RDD operations
  - map, countByValue, reduce, join, ...
- Window and Stateful operations
  - window, countByValueAndWindow, ...

### ■ Output Operations (actions)

- Send data to external entity
  - saveAsHadoopFiles, saveAsTextFile, ...

## Fault-tolerance

- DStreams remember the sequence of operations that created them 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

## Basic Structure of a Spark Streaming Program (1)

- Define a Spark Streaming Context object
  - Define the size of the batches (in seconds) associated with the Streaming context
- Specify the input stream and define a DStream based on it
- Specify the operations to execute for each batch of data
  - Use transformations and actions similar to the ones available for “standard” RDDs

## Basic Structure of a Spark Streaming Program (2)

- Invoke the start method
  - To start processing the input stream
- Wait until the application is killed or the timeout specified in the application expires
  - If the timeout is not set and the application is not killed **the application will run forever**

## Spark Streaming Context

- The Spark Streaming Context is defined by using the `JavaStreamingContext(SparkConf sparkC, Duration batchDuration)` constructor of `JavaStreamingContext`
- The `batchDuration` parameter specifies the “size” of the batches
- Example

```
JavaStreamingContext jssc =
new JavaStreamingContext(conf,Durations.seconds(10));
```

  - The input streams associated with this context will be split in batches of 10 seconds

## Input Streams

- The input Streams can be generate from different sources
  - TCP socket, Kafka, Flume, Kinesis, Twitter
  - Also an HDFS folder can be used as “input stream”

## Input Streams: TPC socket

- A DStream can be associated with the content emitted by a TCP socket
- `socketTextStream(String hostname, int port_number)` is used to create a DStream based on the textual content emitted by a TPC socket
- Example
 

```
JavaReceiverInputDStream<String> lines =
    jssc.socketTextStream("localhost", 9999);
```

  - "Store" the content emitted by localhost:9999 in the lines DStream

27

## Input Streams: (HDFS) folder

- A DStream can be associated with the content of an input (HDFS) folder
  - Every time a new file is inserted in the folder, the content of the file is "stored" in the associated DStream and processed
- `textFileStream(String folder)` is used to create a DStream based on the content of the input folder

28

## Input Streams: (HDFS) folder

- Example

```
JavaDStream<String> lines =  
    jssc.textFileStream(inputFolder);
```

- "Store" the content of the files inserted in the input folder in the lines Dstream
- Every time new files are inserted in the folder their content is "stored" in the current "batch" of the stream

29

## Input Streams: other sources

- Usually DStream objects are defined on top of streams emitted by specific applications that emit real-time streaming data
  - E.g., Apache Kafka, Apache Flume, Kinesis, Twitter
- You can also write your own applications for generating streams of data
  - However, Kafka, Flume and similar tools are usually a more reliable and effective solutions for generating streaming data

30

## Transformations

- Analogously to standard RDDs, also DStream are characterized by a set of transformations
  - When applied to DStream objects, transformations return a new DStream Object
  - The transformation is applied on one batch (RDD) of the input DStream at a time and returns a batch (RDD) of the new DStream
    - i.e., each batch (RDD) of the input DStream is associated with exactly one batch (RDD) of the returned DStream
- Many of the available transformations are the same transformations available for standard RDDs

31

## Basic Transformations on DStreams

- **map(func)**
  - Returns a new DStream by passing each element of the source DStream through a function **func**
- **flatMap(func)**
  - Each input item can be mapped to 0 or more output items. Returns a new DStream
- **filter(func)**
  - Returns a new DStream by selecting only the records of the source DStream on which **func** returns true

32



## Basic Transformations on DStreams

- **reduce(func)**
  - Returns a new DStream of single-element RDDs by aggregating the elements in each RDD of the source DStream using a function **func**. The function should be associative so that it can be computed in parallel
- **reduceByKey(func)**
  - When called on a PairDStream of (K, V) pairs, returns a new PairDStream of (K, V) pairs where the values for each key are aggregated using the given reduce function
- **countByValue()**
  - When called on a DStream of elements of type K, returns a new PairDStream of (K, Long) pairs where the value of each key is its frequency in each batch of the source DStream

33

## Basic Transformations on DStreams

- **count()**
  - Returns a new DStream of single-element RDDs by counting the number of elements in each batch (RDD) of the source Dstream
    - i.e., it counts the number of elements in each input batch (RDD)
- **union(otherStream)**
  - Returns a new DStream that contains the union of the elements in the source DStream and otherDStream.
- **join(otherStream)**
  - When called on two PairDStreams of (K, V) and (K, W) pairs, return a new PairDStream of (K, (V, W)) pairs with all pairs of elements for each key.

34

## Basic Transformations on DStreams

- **cogroup(otherStream)**
  - When called on a PairDStream of (K, V) and (K, W) pairs, return a new DStream of (K, Seq[V], Seq[W]) tuples

35

## Advanced transformation on DStreams

- **transform(func)**
  - It is a specific transformation of DStreams
  - It returns a new DStream by applying an RDD-to-RDD function to every RDD of the source Dstream
    - This can be used to do arbitrary RDD operations on the DStream
  - For example, the functionality of joining every batch in a data stream with another dataset (a standard RDD) is not directly exposed in the DStream API
    - However, you can use transform to do that

36

## Advanced transformation on DStreams

- **transformToPair(func)**
  - It is a specific transformation of PairDStreams
  - It returns a new PairDStream by applying a PairRDD-to-RPairDD function to every PairRDD of the source PairDStream
  - It must be used instead of transform when working with PairDStreams/PairRDDs

37

## Basic Output Operations (actions) on DStreams

- **print()**
  - Prints the first 10 elements of every batch of data in a DStream on the driver node running the streaming application
    - Useful for development and debugging

38

## Basic Output Operations (actions) on DStreams

- **saveAsTextFiles(prefix, [suffix])**
  - Saves the content of the DStream on which it is invoked as text files
    - One folder for each batch
    - The folder name at each batch interval is generated based on prefix, time of the batch (and suffix):  
"prefix-TIME\_IN\_MS[.suffix]"
  - It is not directly available for JavaDStream objects
    - A Scala DStream object must be created from a JavaDStream by invoking the `dstream()` method.
    - `saveAsTextFiles` can be invoked on the returned Scala Dstream
  - Example
    - `Counts.dstream().saveAsTextFiles(outputPathPrefix, "");`

39

## Start and run the computation

- The **start()** method of the `JavaSparkStreamingContext` class is used to start the application on the input stream(s)
- The **awaitTerminationOrTimeout(long milliseconds)** method is used to specify how long the application will run
- The **awaitTerminationOrTimeout()** method is used to **run** the application **forever**
  - Until the application is explicitly killed

40

## Example: Word count – Spark Streaming version

- Problem specification
  - Input: a stream of sentences retrieved from localhost:9999
  - Split the input stream in batches of 10 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch
    - i.e., execute the word count problem for each batch of 10 seconds
  - Store the results also in an HDFS folder

41

## Example: Word count – Spark Streaming version

```
package it.polito.bigdata.spark.StreamingWordCount;
import .....
public class SparkDriver{

    public static void main(String[] args) {
        String outputPathPrefix;
        outputPathPrefix=args[0];

        // Create a configuration object and set the name of the application
        SparkConf conf=new SparkConf().setAppName("SparkStreaming word count");

        // Create a Spark Streaming Context object
        JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(10));

        // Create a (Receiver) DStream that will connect to localhost:9999
        JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost", 9999);
```

42

## Example: Word count – Spark Streaming version

```
// Apply the "standard" transformations to perform the word count task
// However, the "returned" RDDs are DStream/PairDStreamRDDs
JavaDStream<String> words = lines.flatMap(new Split());

JavaPairDStream<String, Integer> wordsOnes = words.mapToPair(new WordOne());

JavaPairDStream<String, Integer> wordsCounts = wordsOnes.reduceByKey(new Sum());

// Print on the stdout 10 pairs of the wordsCounts PairDStream
wordsCounts.print();
```

43

## Example: Word count – Spark Streaming version

```
// Store the result in the output folders with prefix outputPathPrefix
wordsCounts.dstream().saveAsTextFiles(outputPathPrefix, "");

// Start the computation
jssc.start();

// Await until termination or timeout
jssc.awaitTerminationOrTimeout(120000);

jssc.close();
}
}
```

44

## Window operation

- Spark Streaming also provides windowed computations
  - It allows you to apply transformations over a sliding window of data
    - Each window contains a set of batches of the input stream
    - Windows can be overlapped
      - i.e., the same batch can be included in many consecutive windows

45

## Window operation

- Graphical example



- Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream

46

## Window operation

- In the example, the operation
  - is applied over the last 3 time units of data (i.e., the last 3 batches of the input DStream)
    - Each window contains the data of 3 batches
  - and slides by 2 time units

47

## Window operation: parameters

- Any window operation needs to specify two parameters:
  - Window length
    - The duration of the window (3 in the example)
  - Sliding interval
    - The interval at which the window operation is performed (2 in the example)
- These two parameters must be multiples of the batch interval of the source DStream

48

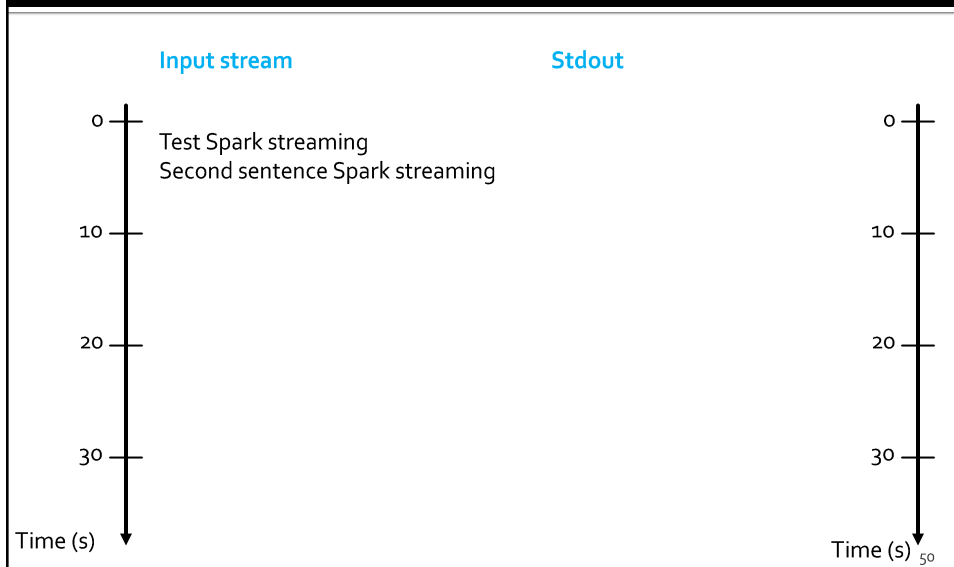


## Word count and Window

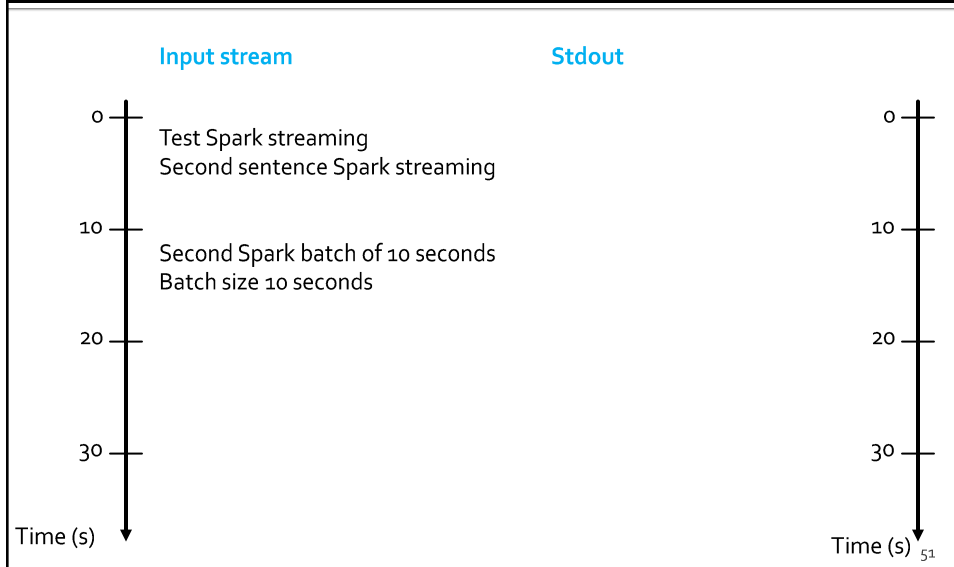
- Problem specification
  - Input: a stream of sentences
  - Split the input stream in batches of 10 seconds
  - Define windows with the following characteristics
    - Window length: 20 seconds (i.e., 2 batches)
    - Sliding interval: 10 seconds (i.e., 1 batch)
  - Print on the standard output, for each window, the occurrences of each word appearing in the window
    - i.e., execute the word count problem for each window

49

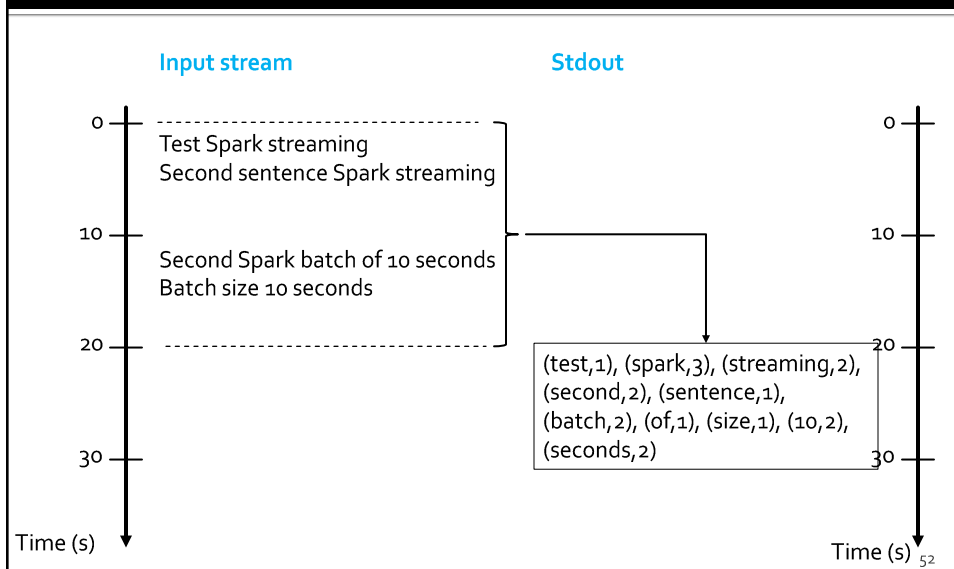
## Word count and Window



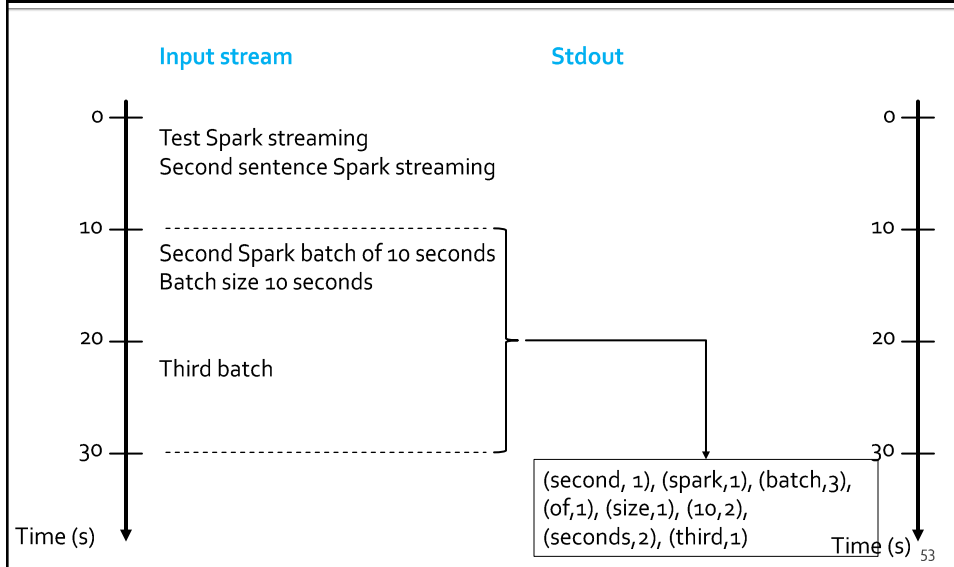
## Word count and Window



## Word count and Window



## Word count and Window



## Basic Window Transformations

- **window(windowLength, slideInterval)**
  - Returns a new DStream which is computed based on windowed batches of the source DStream.
- **countByWindow(windowLength, slideInterval)**
  - Returns a new single-element stream containing the number of elements of each window
    - The returned object is a `JavaDStream<Long>`. However, it contains only one value for each window (the number of elements of the last analyzed window)
- **reduceByWindow(func, windowLength, slideInterval)**
  - Returns a new single-element stream, created by aggregating elements in the stream over a sliding interval using `func`. The function should be associative so that it can be computed correctly in parallel

54

## Basic Window Transformations

- `countByValueAndWindow(windowLength, slideInterval)`
  - When it is called on a `PairDStream` of (K, V) pairs, returns a new `PairDStream` of (K, Long) pairs where the value of each key is its frequency within a sliding window

55

## Checkpoints

- A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic (e.g., system failures, JVM crashes, etc.)
- For this to be possible, Spark Streaming needs to checkpoint enough information to a fault-tolerant storage system such that it can recover from failures
- This result is achieved by means of checkpoints
  - Operations that store the data and metadata needed to restart the computation if failures happen
- Checkpointing is necessary even for some window transformations and stateful transformations

56

## Checkpoints

- Checkpointing is enabled by using the `checkpoint(String folder)` method of `JavaSparkStreamingContext`
  - The parameter is the folder that is used to store temporary data

57

## Example: Word count and Windows

- Problem specification
  - Input: a stream of sentences retrieved from `localhost:9999`
  - Split the input stream in batches of 10 seconds
  - Define windows with the following characteristics
    - Window length: 30 seconds (i.e., 3 batches)
    - Sliding interval: 10 seconds (i.e., 1 batch)
  - Print on the standard output, for each window, the occurrences of each word appearing in the window
    - i.e., execute the word count problem for each window
  - Store the results also in an HDFS folder

58

## Example: Word count and Windows

```
package it.polito.bigdata.spark.StreamingWordCount;

import .....

public class SparkDriver {
    public static void main(String[] args) {

        String outputPathPrefix=args[0];

        // Create a configuration object and set the name of the application
        SparkConf conf=new SparkConf().setAppName("Spark Streaming word count");

        // Create a Spark Streaming Context object
        JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(10));

        // Set the checkpoint folder (it is needed by some window transformations)
        jssc.checkpoint("checkpointfolder");

        // Create a (Receiver) DStream that will connect to localhost:9999
        JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost", 9999);
```

59

## Example: Word count and Windows

```
// Apply the "standard" transformations to perform the word count task
// However, the "returned" RDDs are DStream/PairDStream RDDs
JavaDStream<String> words = lines.flatMap(new Split());

// Count the occurrence of each word in the current window
JavaPairDStream<String,Integer> wordsOnes = words.mapToPair(new WordOne());

// reduceByKeyAndWindow is used instead of reduceByKey
// The characteristics of the window is also specified
JavaPairDStream<String,Integer> wordsCounts =
    wordsOnes.reduceByKeyAndWindow(new Sum(),
                                    Durations.seconds(30),
                                    Durations.seconds(10));
```

60

## Example: Word count and Windows

```
// Print the num. of occurrences of each word of the current window (only 10 of them)
wordsCounts.print();

// Store the output of the computation in the folders with prefix outputPathPrefix
wordsCounts.dstream().saveAsTextFiles(outputPathPrefix, "");

// Start the computation
jssc.start();

jssc.awaitTerminationOrTimeout(120000);

jssc.close();

    }
}
```

61

## UpdateStateByKey Transformation

- The updateStateByKey transformation allows maintaining arbitrary state while continuously updating it with new information
  - It is based on two steps
    - Define the state
      - The state can be an arbitrary data type
    - Define the state update function
      - Specify with a function how to update the state using the previous state and the new values from an input stream

62

## UpdateStateByKey Transformation

- In every batch, Spark will apply the state update function for all existing keys, regardless of whether they have new data in a batch or not
  - If the update function returns None then the key-value pair will be eliminated
- The function is used to update the value associated with a key by combining the former value and the new values associated with the key
  - The call method of the "function" is invoked on the list of values associated with on one key at a time and return the new aggregated value for the considered key

63

## Word count example (Stateful version)

- By using the UpdateStateByKey, the application can continuously update the number of occurrences of each word
  - The number of occurrences stored in the PairDStream returned by this transformation is computed over the union of all the batches (for the first one to current one)
    - For efficiency reasons, the new value is computed by combining the last value with the values of the current batch

64



## Example: Word count (stateful version)

- Problem specification
  - Input: a stream of sentences retrieved from localhost:9999
  - Split the input stream in batches of 10 seconds
  - Print on the standard output, every 10 seconds, the occurrences of each word appearing in the stream (from time 0 to the current time)
    - i.e., execute the word count problem from the beginning of the stream to current time
  - Store the results also in an HDFS folder

65

## Example: Word count (stateful version)

```
package it.polito.bigdata.spark.StreamingWordCount;

import .....

public class SparkDriver {
    public static void main(String[] args) {

        String outputPathPrefix=args[0];

        // Create a configuration object and set the name of the application
        SparkConf conf=new SparkConf().setAppName("SparkStreaming word count");

        // Create a Spark Streaming Context object
        JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(10));

        // Create a (Receiver) DStream that will connect to localhost:9999
        JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost", 9999);
```

66

## Example: Word count (stateful version)

```
// Set the checkpoint folder (it is needed by the stateful transformation)
jssc.checkpoint("checkpointfolder");

// Apply the "standard" transformations to perform the word count task
// However, the "returned" RDDs are DStream/PairDStream RDDs
JavaDStream<String> words = lines.flatMap(new Split());

JavaPairDStream<String,Integer> wordsOnes = words.mapToPair(new WordOne());

JavaPairDStream<String,Integer> wordsCounts = wordsOnes.reduceByKey(new Sum());

// Update the number of occurrences of each word
JavaPairDStream<String,Integer> totalWordsCounts =
    wordsCounts.updateStateByKey(new Update());
```

67

## Example: Word count (stateful version)

```
// Print the result on the standard output
totalWordsCounts.print();

// Store the output of the computation in the folders with prefix outputPathPrefix
totalWordsCounts.dstream().saveAsTextFiles(outputPathPrefix, "");

// Start the computation
jssc.start();

jssc.awaitTerminationOrTimeout(120000);

jssc.close();

    }
}
```

68

## Example: Word count (stateful version)

```
// This class contains the code that is used to update the "state" after each batch analysis
// It updates the frequency of each occurring word
package it.polito.bigdata.spark.StreamingWordCount;

import java.util.List;

import org.apache.spark.api.java.function.Function2;

import com.google.common.base.Optional;
```

69

## Example: Word count (stateful version)

```
@SuppressWarnings("serial")
public class Update implements Function2<List<Integer>, Optional<Integer>, Optional<Integer>> {

    @Override
    public Optional<Integer> call(List<Integer> newValues, Optional<Integer> state) throws
        Exception {
        // state.or(o) returns the value of State or the default value o if state is not defined
        Integer newSum = state.or(0);

        // Iterates over the new values and sum them to the previous value
        for (Integer value : newValues) {
            newSum += value;
        }

        return Optional.of(newSum);
    }
}
```

70

## Example: Word count – use of transformPair

- Problem specification
  - Input: a stream of sentences retrieved from localhost:9999
  - Split the input stream in batches of 10 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch
    - The pairs must be returned/displayed sorted by key
  - Store the results also in an HDFS folder

71

## Example: Word count – Spark Streaming version

```
package it.polito.bigdata.spark.StreamingWordCount;
import .....
public class SparkDriver{

    public static void main(String[] args) {
        String outputPathPrefix;
        outputPathPrefix=args[0];

        // Create a configuration object and set the name of the application
        SparkConf conf=new SparkConf().setAppName("SparkStreaming word count");

        // Create a Spark Streaming Context object
        JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.seconds(10));

        // Create a (Receiver) DStream that will connect to localhost:9999
        JavaReceiverInputDStream<String> lines = jssc.socketTextStream("localhost", 9999);
```

72

## Example: Word count – Spark Streaming version

```
// Apply the "standard" transformations to perform the word count task
// However, the "returned" RDDs are DStream/PairDStreamRDDs
JavaDStream<String> words = lines.flatMap(new Split());

JavaPairDStream<String, Integer> wordsOnes = words.mapToPair(new WordOne());

JavaPairDStream<String, Integer> wordsCounts = wordsOnes.reduceByKey(new Sum());

// Sort the pairs word, frequency by key
JavaPairDStream<String, Integer> wordsCountsSortByKey =
    wordsCounts.transformToPair(new Sort());

// Print on the stdout 10 pairs of the wordsCountsPairDStream
wordsCountsSortByKey.print();
```

73

## Example: Word count – Spark Streaming version

```
// Store the result in the output folders with prefix outputPathPrefix
wordsCountsSortByKey.dstream().saveAsTextFiles(outputPathPrefix, "");

// Start the computation
jssc.start();

// Await until termination or timeout
jssc.awaitTerminationOrTimeout(120000);

jssc.close();
}
}
```

74

## Example: Word count – Spark Streaming version

```
// This is the class that is used to sort the content of the PairRDD associated with each batch
package it.polito.bigdata.spark.StreamingWordCount;

import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.function.Function;

@SuppressWarnings("serial")
public class Sort implements Function<JavaPairRDD<String, Integer>, JavaPairRDD<String,
    Integer>> {

    @Override
    public JavaPairRDD<String, Integer> call(JavaPairRDD<String, Integer> rdd) throws Exception {
        // Sort the content of the "Standard" RDD by key and return the new one
        return rdd.sortByKey();
    }

}
```

75