Streaming data analytics: Frameworks

What is stream processing?

- Act of **continuously incorporating new data** to compute a result
- **Input data is unbounded** → no beginning and no end
- Series of events that arrive at the stream processing system
- The application will output multiple versions of the results as it runs or put them in a storage

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
  - ...

Advantages

- Vastly **higher throughput** in data processing
- **Low latency**: application respond quickly (e.g., in seconds)
  - It can keep states in memory
- **More efficient** in updating a result than repeated batch jobs, because it automatically incrementalizes the computation

Requirements and Challenges

- Scalable to large clusters
- Responding to events at low latency
- Simple programming model
- Processing each event exactly once despite machine failures - Efficient fault-tolerance in stateful computations

Requirements and Challenges

- Processing out-of-order data based on application timestamps (also called event time)
- Maintaining large amounts of state
- Handling load imbalance and stragglers
- Updating your application's business logic at runtime

Stream Processing Frameworks for Big Streaming Data Analytics

- Several frameworks have been proposed to process in real-time or in near real-time data streams
  - Apache Spark (Streaming component)
  - Apache Storm
  - Apache Flink
  - Apache Samza
  - Apache Apex
  - Apache Flume
  - Amazon Kinesis Streams
  - ...
- All these frameworks use a cluster of servers to scale horizontally with respect to the (big) amount of data to be analyzed

Stream Processing Frameworks for Big Streaming Data Analytics

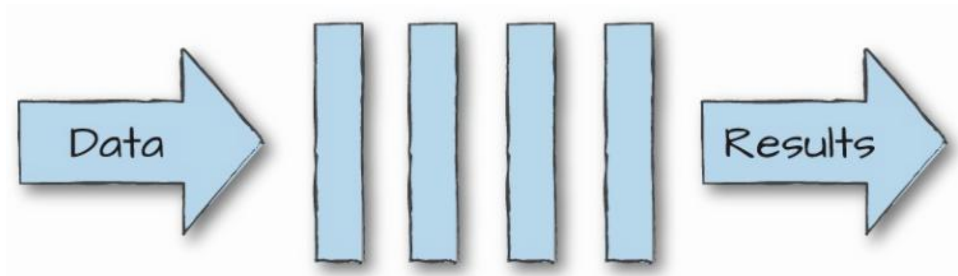
- Two main “solutions”
  - “Continuous” computation of data streams
    - Data are processed as soon as they arrive
      - Every time a new record arrives from the input stream, it is immediately processed and a result is emitted as soon as possible
    - Real-time processing

Stream Processing Frameworks for Big Streaming Data Analytics

- "Micro-batch" stream processing
  - Input data are collected in micro-batches
    - Each micro-batch contains all the data received in a time window (typically less than a few seconds of data)
  - One micro-batch a time is processed
    - Every time a micro-batch of data is ready, its entire content is processed and a result is emitted
  - Near real-time processing

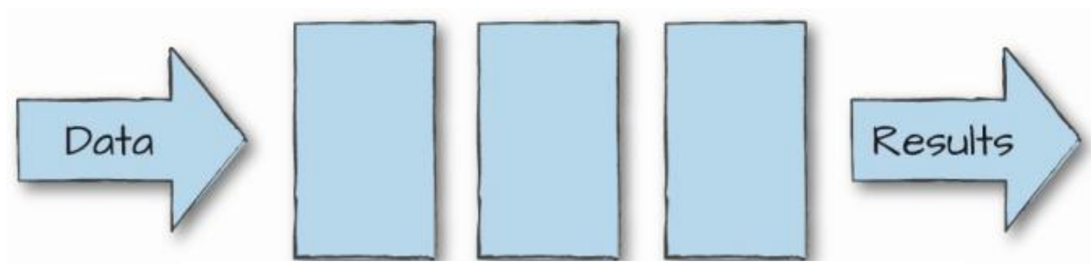
Continuous vs Micro-batch

- **Continuous computation**



One record at a time

- **Micro-batch computation**



One micro-batch at a time

Input data processing and Result guarantees

- At-most-once
  - Every input element of a stream is processed once or less
  - It is also called no guarantee
  - The result can be wrong/approximated
- At-least-once
  - Every input element of a stream is processed once or more
  - Input elements are replayed when there are failures
  - The result can be wrong/approximated

Input data processing and Result guarantees

- Exactly-once
  - Every input element of a stream is processed exactly once
  - Input elements are replayed when there are failures
  - If elements have been already processed they are not reprocessed
  - The result is always correct
  - Slower than the other processing approaches

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
  - **Micro-batch** streaming processing
  - **Exactly-once** guarantees
  - 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

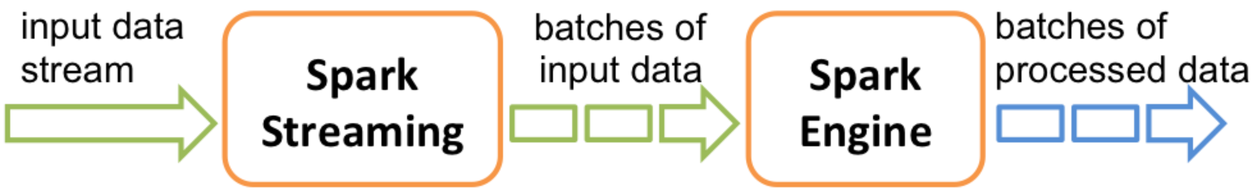
Spark Discretized Stream Processing

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**

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



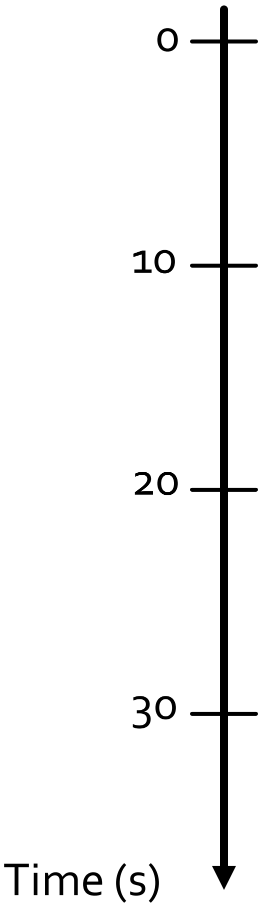
Word count – Spark Streaming version

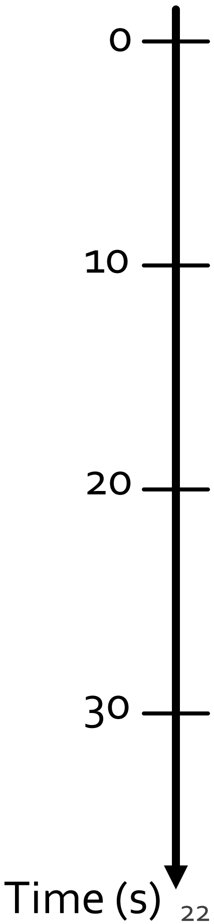
- Problem specification
  - Input: a stream of sentences
  - Split the input stream in batches of 10 seconds each and print on the standard output, for each batch, the occurrences of each word appearing in the batch
    - i.e., execute the word count application one time for each batch of 10 seconds

Word count – Spark Streaming version

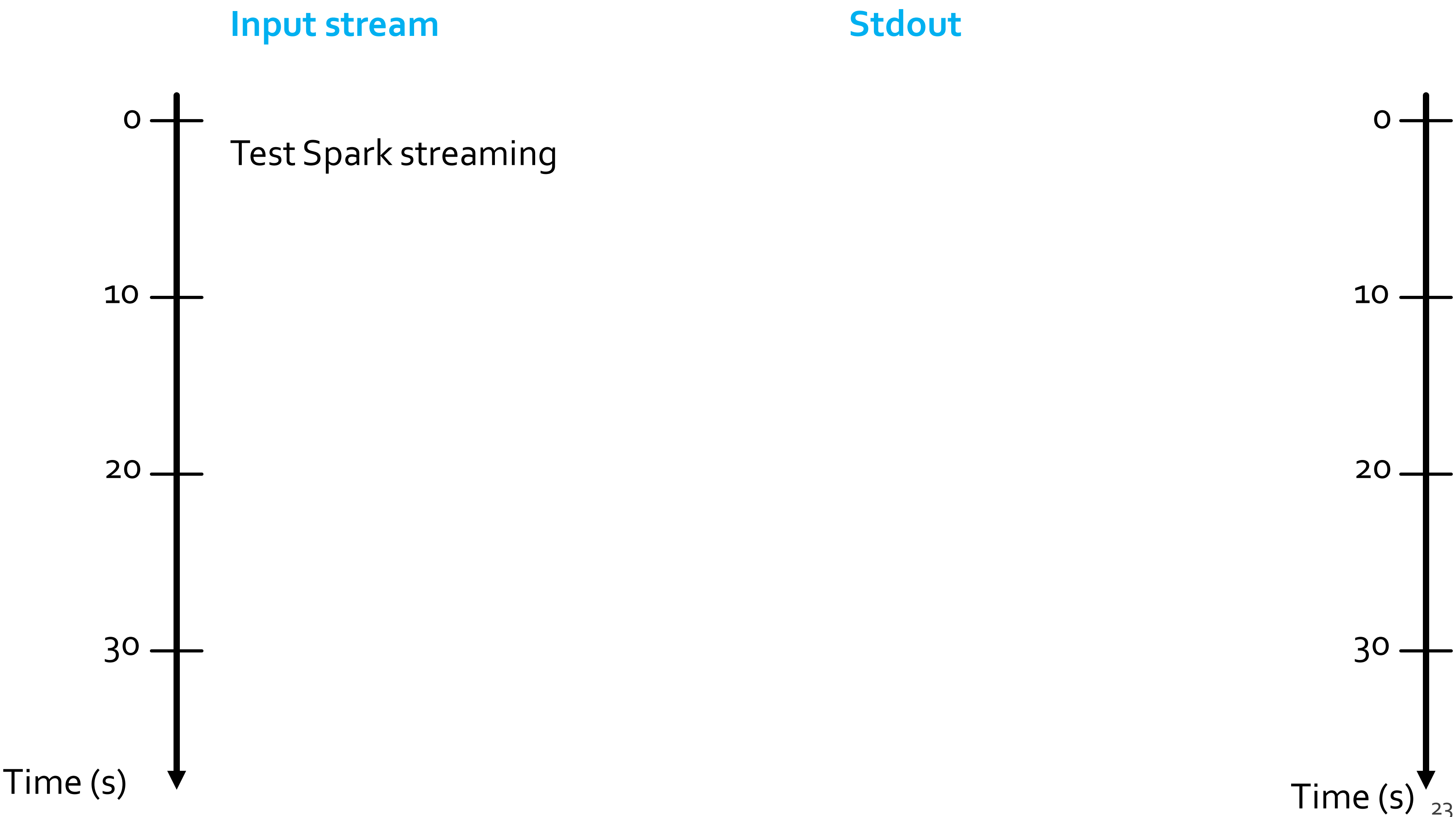
Input stream

Stdout



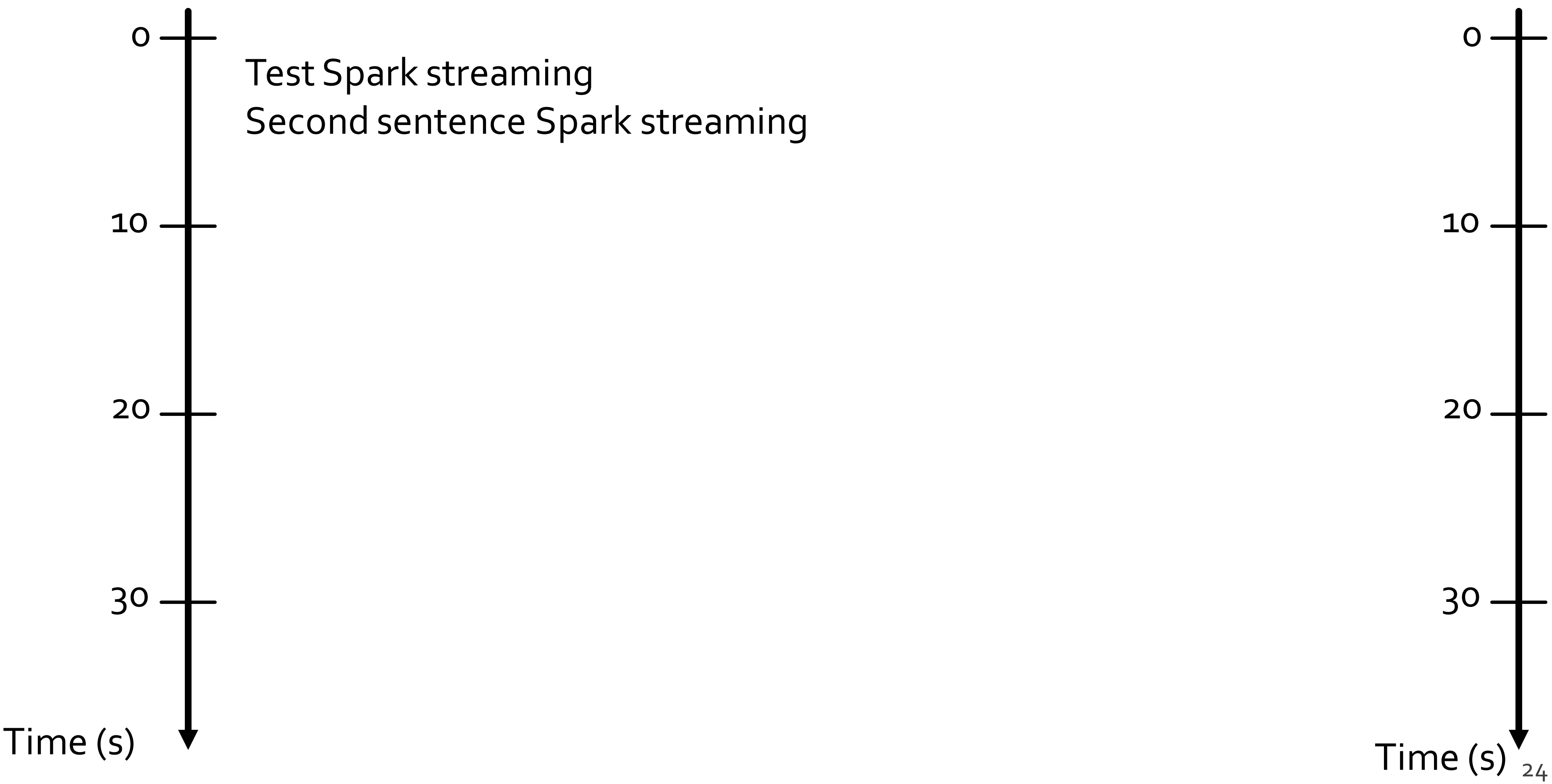


Word count – Spark Streaming version

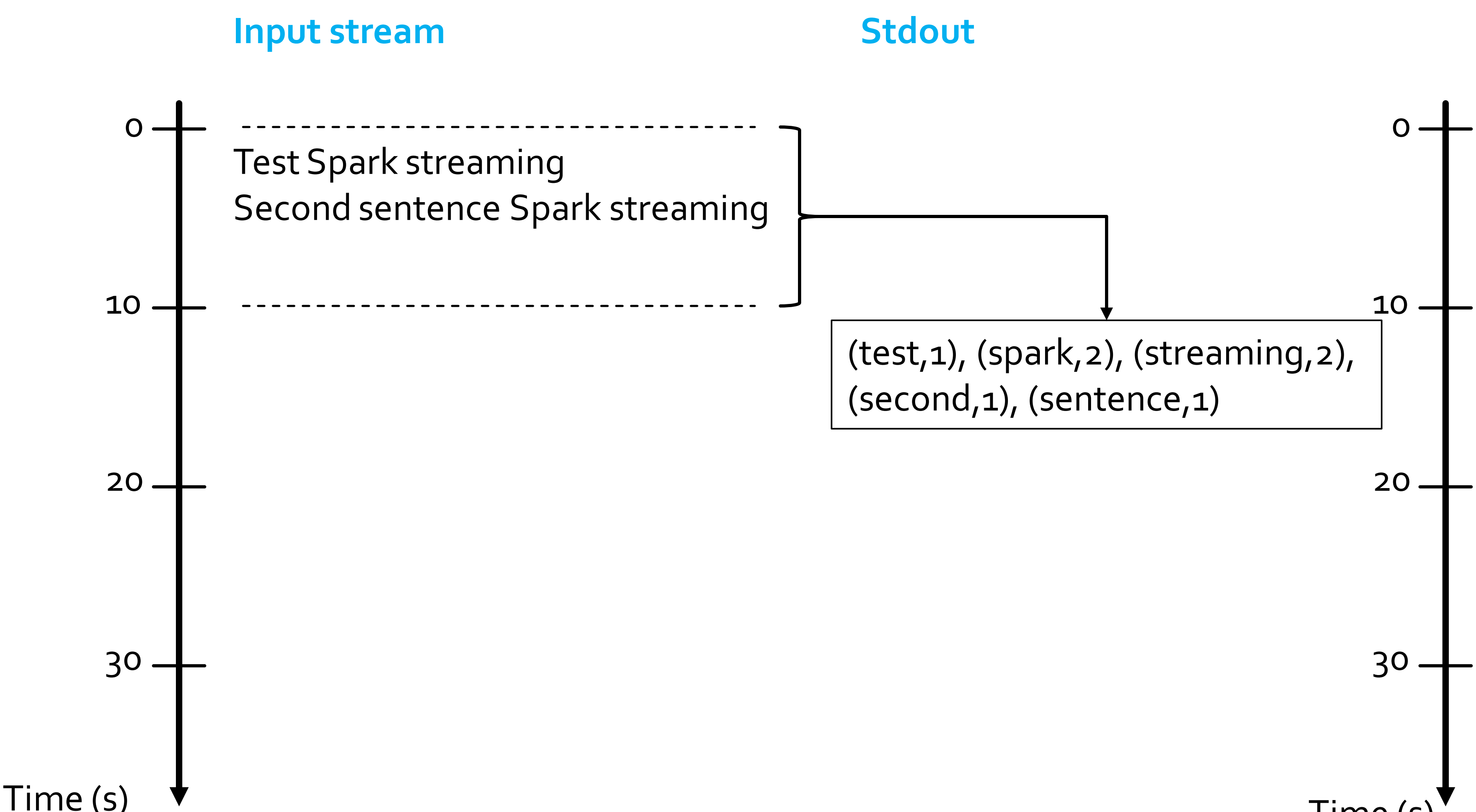


Word count – Spark Streaming version

Input stream **Stdout**



Word count – Spark Streaming version



Word count – Spark Streaming version

Input stream

Stdout

0

Test Spark streaming
Second sentence Spark streaming

10

Second Spark batch of 10 seconds
Batch size 10 seconds

(test,1), (spark,2), (streaming,2), (second,1), (sentence,1)

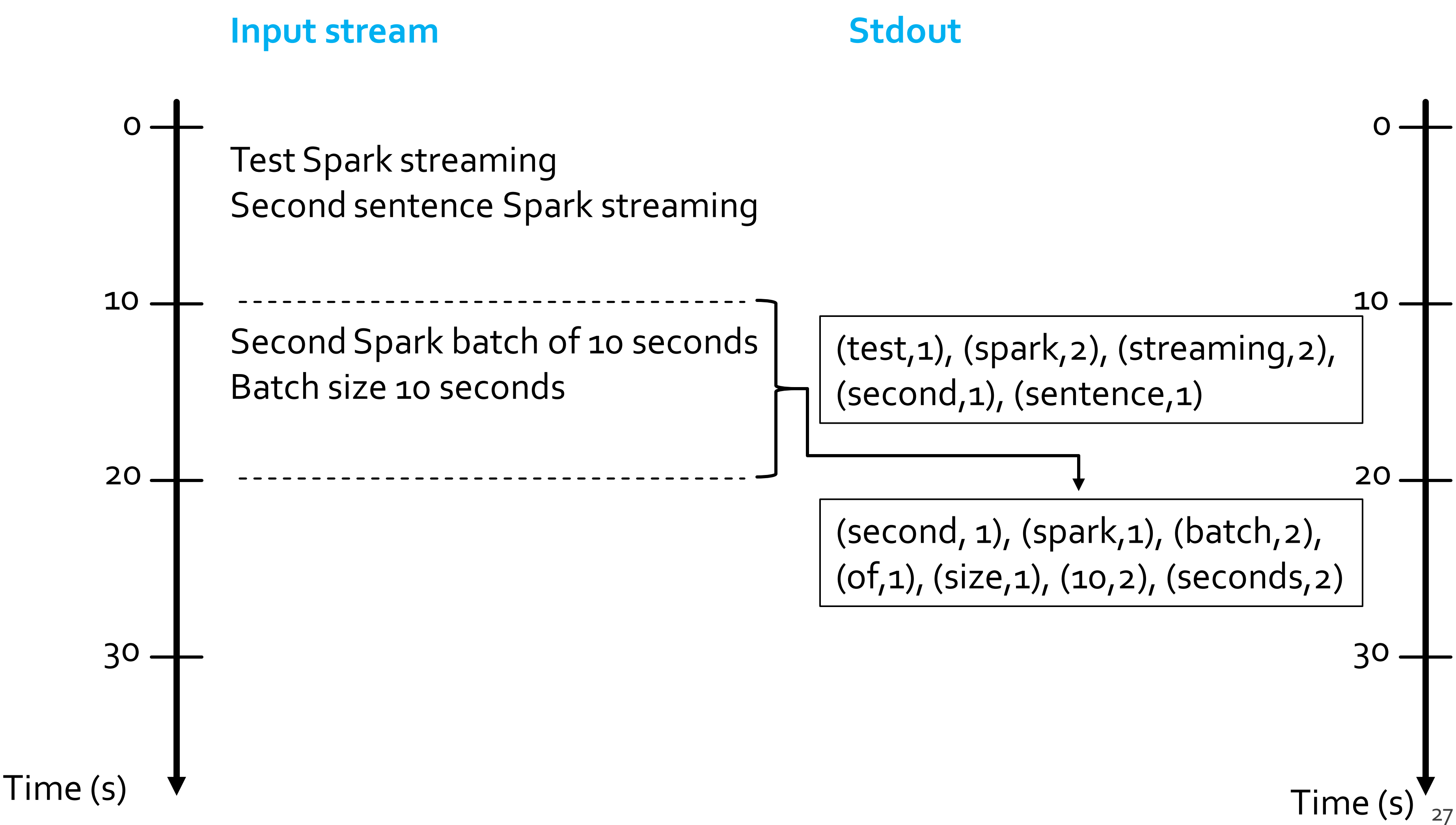
20

30

Time (s)

Time (s)

Word count – Spark Streaming version



Word count – Spark Streaming version

Input stream

0

Test Spark streaming
Second sentence Spark streaming

10

Second Spark batch of 10 seconds
Batch size 10 seconds

20

Third batch

30

Time (s)

Stdout

0

10

(test,1), (spark,2), (streaming,2), (second,1), (sentence,1)

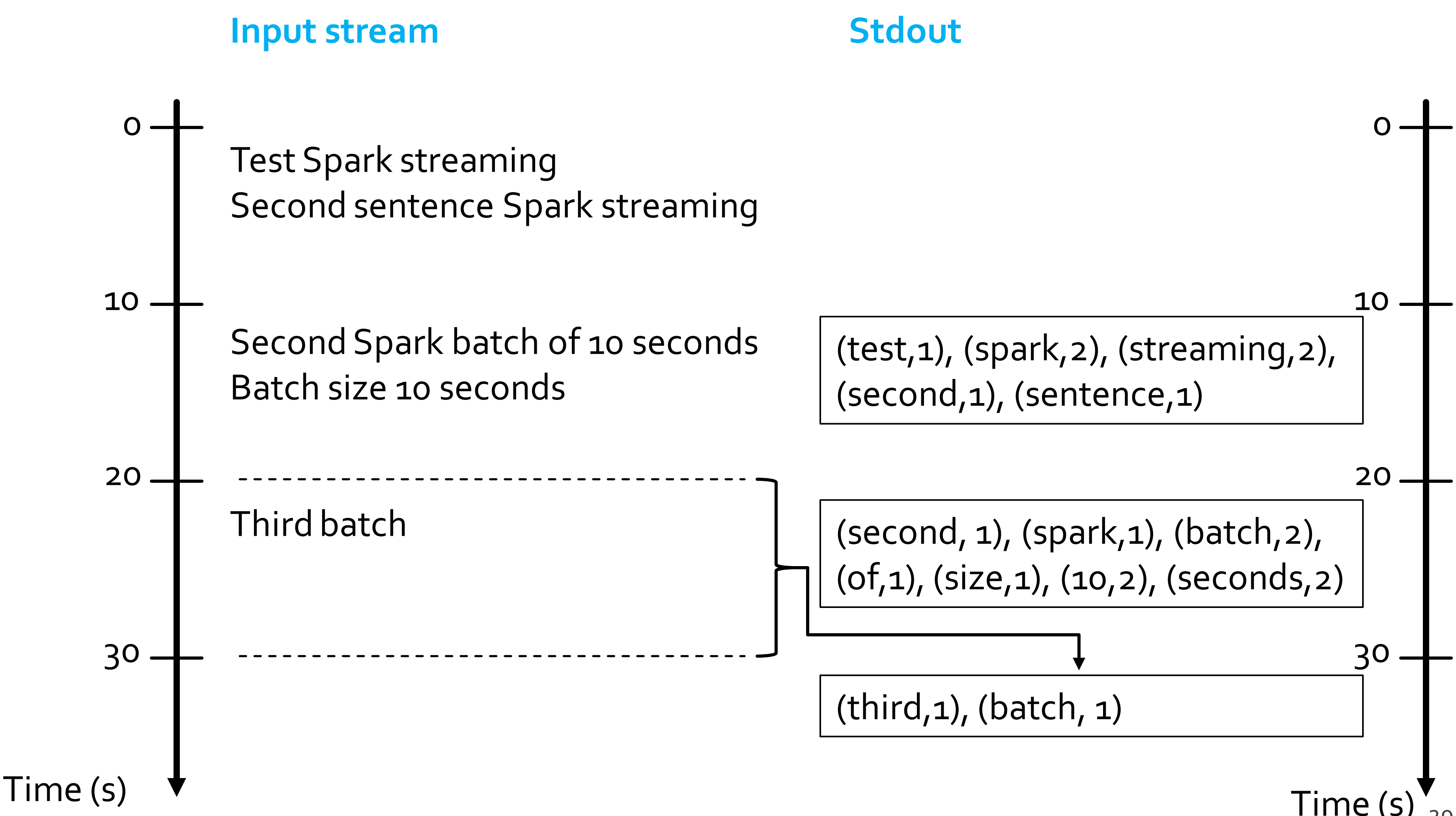
20

(second, 1), (spark,1), (batch,2), (of,1), (size,1), (10,2), (seconds,2)

30

Time (s)

Word count – Spark Streaming version



Key concepts

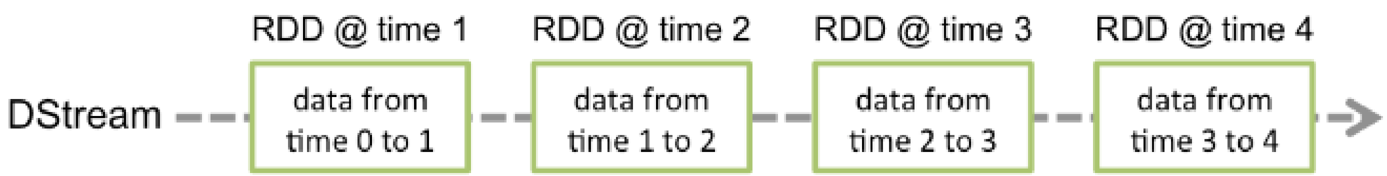
- DStream
  - Sequence of RDDs representing a discretized version of the input stream of data
    - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets, ..
  - One RDD for each batch of the input stream

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, …

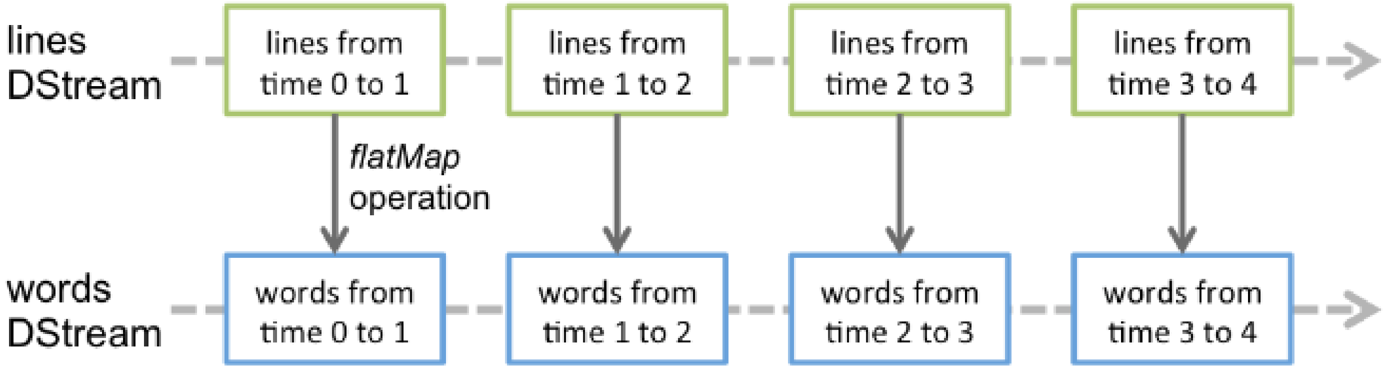
Word count – DStreams

- A DStream is represented by a continuous series of RDDs. Each RDD in a DStream contains data from a certain batch/interval



Word count – DStreams

- Any operation applied on a DStream translates to operations on the underlying RDDs
- These underlying RDD transformations are computed by the Spark engine



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

Spark Streaming Programs

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 **StreamingContext(SparkConf sparkC, Duration batchDuration)** constructor of the class **pyspark.streaming.StreamingContext**
- The **batchDuration** parameter specifies the "size" of the batches in seconds
- Example

```
from pyspark.streaming import StreamingContext
ssc = StreamingContext(sc, 10)
```

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

Spark Streaming Context

- After a context is defined, you have to do the following
  - Define the input sources by creating input Dstreams
  - Define the streaming computations by applying transformation and output operations to DStreams

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”
    - This option is usually used during the application development to perform a set of initial tests

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
  lines = ssc.socketTextStream("localhost", 9999)
  - "Store" the content emitted by localhost:9999 in the lines DStream

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
  - Pay attention that updating the content of a file does not trigger/change the content of the DStream
- **textFileStream(String folder)** is used to create a DStream based on the content of the input folder

Input Streams: (HDFS) folder

- Example

  lines = 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

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

Transformations

- Analogously to standard RDDs, also DStreams 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

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

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 must be associative and commutative so that it can be computed in parallel
  - Note that the **reduce** method of DStreams is a **transformation**

Basic Transformations on DStreams

- **reduceByKey(func)**
  - When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function
- **combineByKey(createCombiner, mergeValue, mergeCombiners)**
  - When called on a DStream of (K, V) pairs, returns a new DStream of (K, W) pairs where the values for each key are aggregated using the given combine functions

Basic Transformations on DStreams

- **groupByKey()**
  - When called on a DStream of (K, V) pairs, returns a new DStream of (K, Iterable<V>) pairs where the values for each key is the "concatenation" of all the values associated with key K
    - I.e., It returns a new DStream by applying groupByKey on one batch (one RDD) of the input stream at a time

Basic Transformations on DStreams

- **countByValue()**
  - When called on a DStream of elements of type K, returns a new DStream of (K, Long) pairs where the value of each key is its frequency in each batch of the source Dstream
  - Note that the **countByValue** method of DStreams is a **transformation**

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)
  - Note that the **count** method of DStreams is a **transformation**

Basic Transformations on DStreams

- **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 DStreams of (K, V) and (K, W) pairs, return a new DStream of (K, (V, W)) pairs with all pairs of elements for each key

Basic Transformations on DStreams

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

Basic Actions on DStreams

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

Basic 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]“
  - Example

```
Counts.saveAsTextFiles(outputPathPrefix, "")
```

Start and run the computation

- The **streamingContext.start()** method is used to start the application on the input stream(s)
- The **awaitTerminationOrTimeout(long millisecons)** method is used to specify how long the application will run
- The **awaitTermination()** method is used to **run** the application **forever**
  - Until the application is explicitly killed
  - The processing can be manually stopped using **streamingContext.stop()**

Start and run the computation

- Points to remember:
  - Once a context has been started, no new streaming computations can be set up or added to it
  - Once a context has been stopped, it cannot be restarted
  - Only one StreamingContext per application can be active at the same time
  - **stop()** on StreamingContext also stops the SparkContext
    - To stop only the StreamingContext, set the optional parameter of stop() called **stopSparkContext** to False

Example: Word count – Spark Streaming version

- Problem specification
  - Input: a stream of sentences retrieved from localhost:9999
  - Split the input stream in batches of 5 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 5 seconds
  - Store the results also in an HDFS folder

Example: Word count – Spark Streaming version

```
from pyspark.streaming import StreamingContext

# Set prefix of the output folders
outputPathPrefix="resSparkStreamingExamples"

#Create  a configuration  object and#set the name  of the applicationconf
SparkConf().setAppName("Streaming word count")

# Create  a Spark Context  object
sc = SparkContext(conf=conf)

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 5)

# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
```

Example: Word count – Spark Streaming version

```
# Apply a chain of  transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))

wordsCounts = wordsOnes.reduceByKey(lambda v1, v2: v1+v2)

# Print the result on the standard output
wordsCounts.pprint()

# Store the result in HDFS
wordsCounts.saveAsTextFiles(outputPathPrefix, "")
```

Example: Word count – Spark Streaming version

```
#Start the computation
ssc.start()

# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)

ssc.stop(stopSparkContext=False)
```

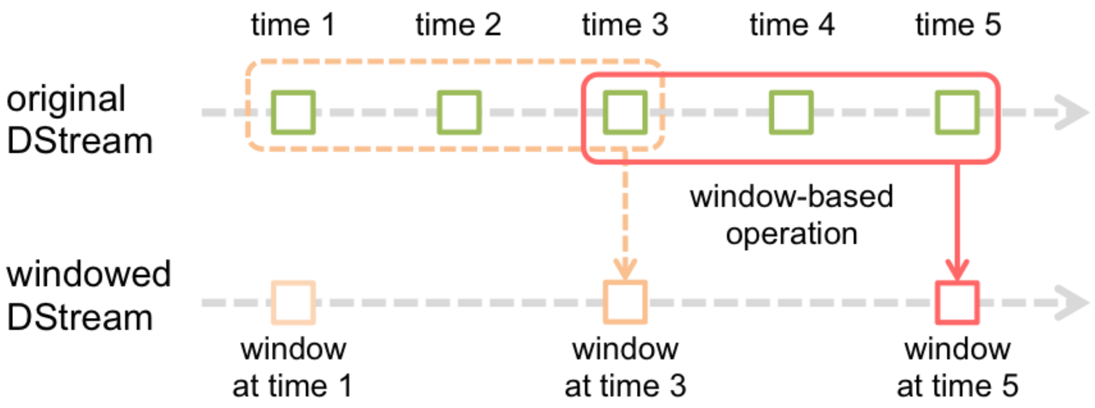
Windowed Computation

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

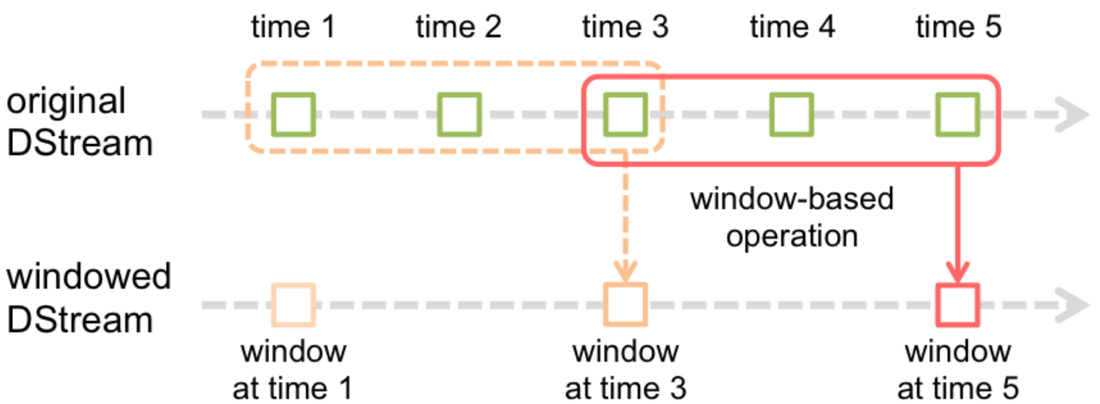
Window operation

- 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

Window operation



- In the graphical 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
  - slides by 2 time units

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

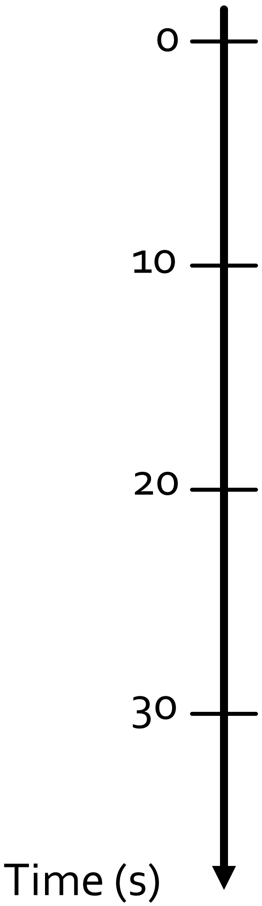
Word count and Window

- Problem specification
  - Input: a stream of sentences
  - Split the input stream in batches of 10 seconds
  - Define widows 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

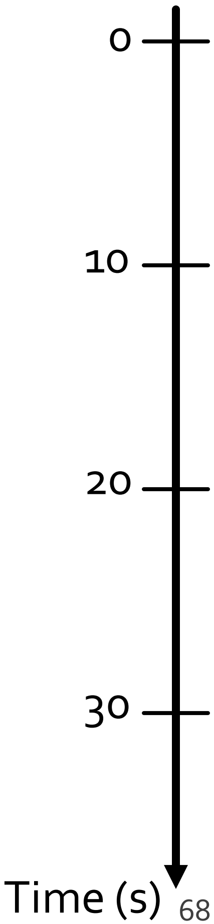
Word count and Window

Input stream

Stdout



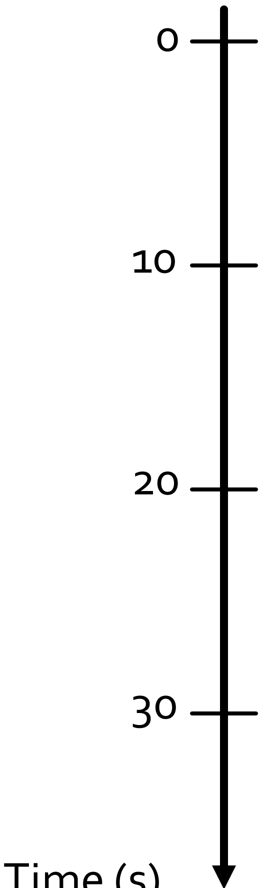
Test Spark streaming
Second sentence Spark streaming



Word count and Window

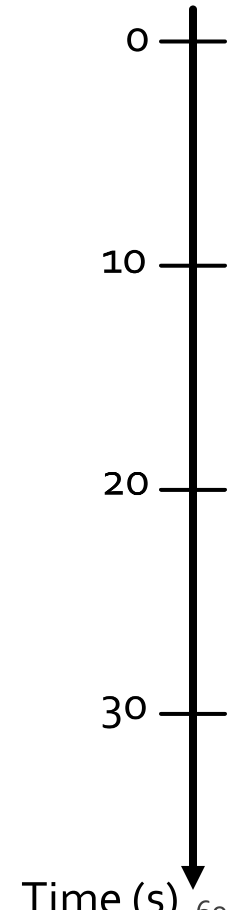
Input stream

Stdout

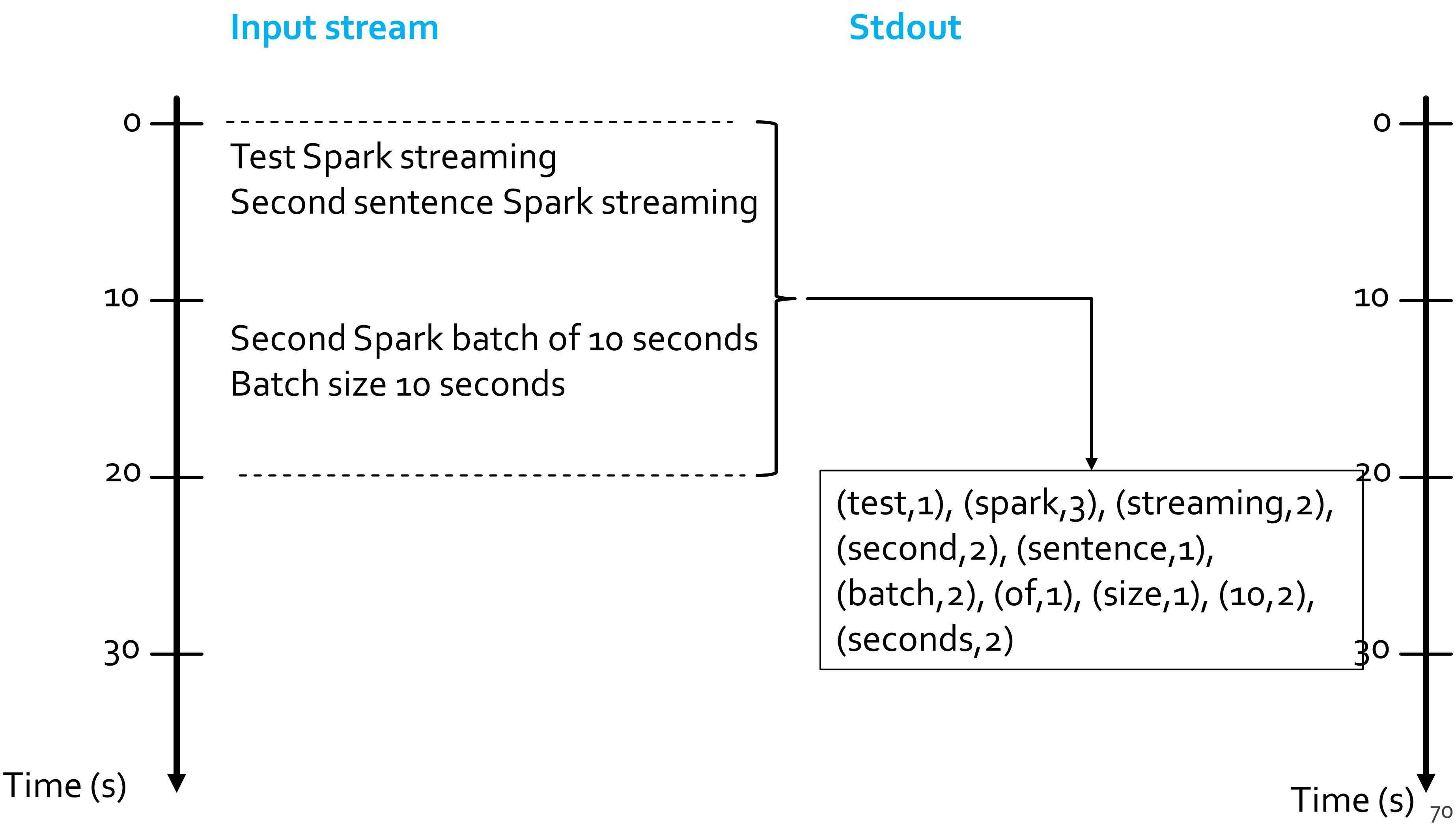


Test Spark streaming
Second sentence Spark streaming

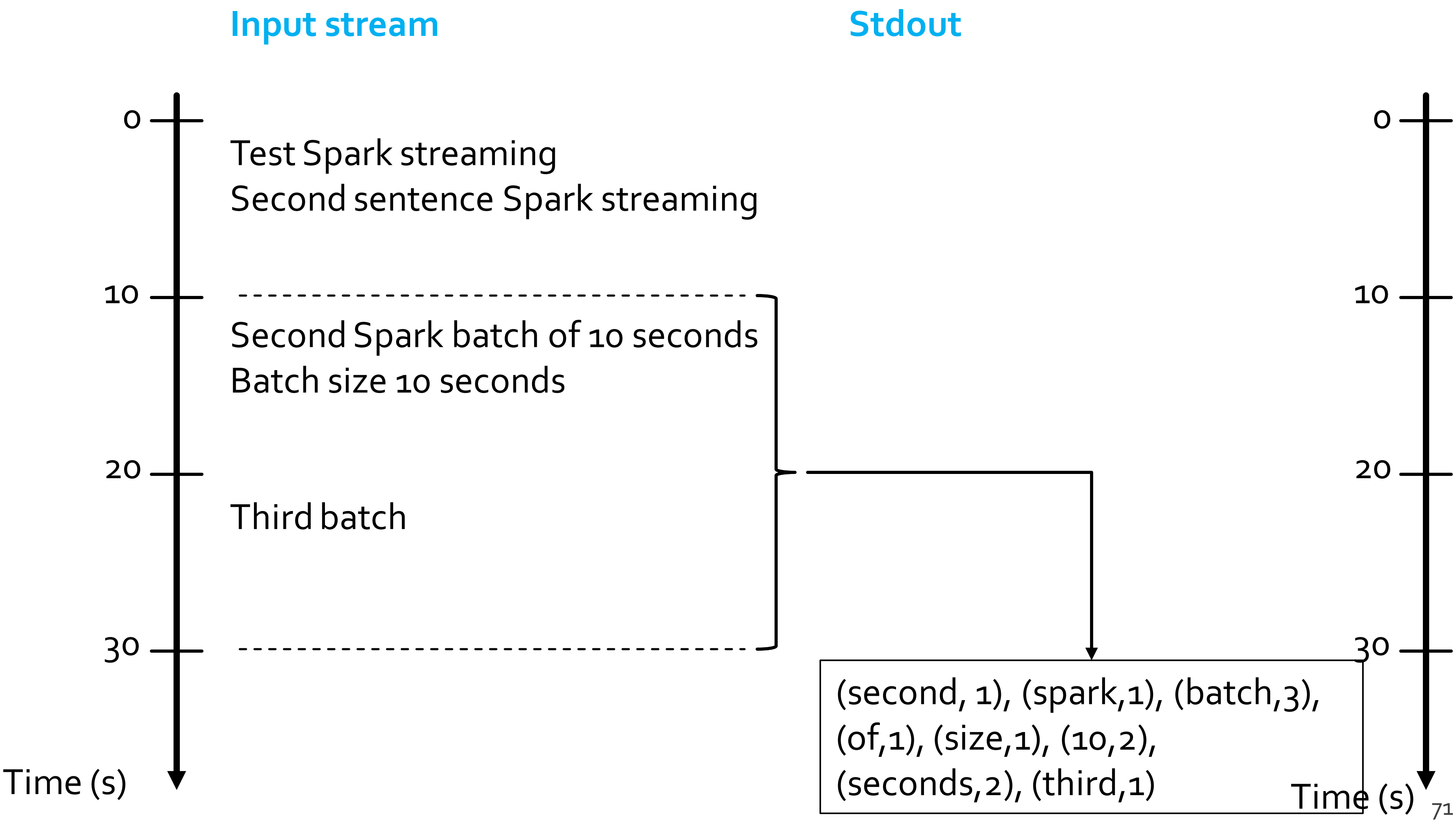
Second Spark batch of 10 seconds
Batch size 10 seconds



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 Dstream of Long objects. However, it contains only one value for each window (the number of elements of the last analyzed window)

Basic Window Transformations

- **reduceByWindow(reduceFunc, invReduceFunc, windowDuration, slideDuration)**
  - Returns a new single-element stream, created by aggregating elements in the stream over a sliding interval using **func**
    - The function must be associative and commutative so that it can be computed correctly in parallel
  - If **invReduceFunc** is not None, the reduction is done incrementally using the old window's reduced value

Basic Window Transformations

- **countByValueAndWindow(windowDuration , slideDuration)**
  - When called on a DStream of elements of type K, returns a new DStream of (K, Long) pairs where the value of each key K is its frequency in each window of the source DStream

Basic Window Transformations

- **reduceByKeyAndWindow(func, invFunc, windowDuration, slideDuration=None, numPartitions=None)**
  - When called on a DStream of (K, V) pairs, returns a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function **func over batches in a sliding window**
  - The window duration (length) is specified as a parameter of this invocation (**windowDuration**)

Basic Window Transformations

- If **slideDuration** is None, the batchDuration of the StreamingContext object is used
  - i.e., 1 batch sliding window
- If **invFunc** is provideved (is not None), the reduction is done incrementally using the old window's reduced values
  - i.e., invFunc is used to apply an inverse reduce operation by considering the old values that left the window (e.g., subtracting old counts)

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

Checkpoints

- Checkpointing is enabled by using the **checkpoint(String folder)** method of SparkStreamingContext
  - The parameter is the folder that is used to store temporary data
- Similar as for processing graphs with GraphFrames library
  - With GraphFrames, the checkpoint was the one of SparkContext

Example: Word count and Windows

- Problem specification
  - Input: a stream of sentences retrieved from localhost:9999
  - Split the input stream in batches of 5 seconds
  - Define widows with the following characteristics
    - Window length: 15 seconds (i.e., 3 batches)
    - Sliding interval: 5 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

Example: Word count and Windows

```
from pyspark.streaming import StreamingContext

# Set prefix of the output folders
outputPathPrefix="resSparkStreamingExamples"

#Create  a configuration  object and#set the name  of the applicationconf
SparkConf().setAppName("Streaming word count")

# Create  a Spark Context  object
sc = SparkContext(conf=conf)

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 5)

# Set the checkpoint folder (it is needed by some window transformations)
ssc.checkpoint("checkpointfolder")
```

Example: Word count and Windows

```
# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Apply a chain of  transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))

# reduceByKeyAndWindow is used instead of reduceByKey
# The durantion of the window is also specified
wordsCounts = wordsOnes\
.reduceByKeyAndWindow(lambda v1, v2: v1+v2, None, 15)
```

Example: Word count and Windows

```
# Print the num. of occurrences of each word of the current window
# (only 10 of them)
wordsCounts.pprint()

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

#Start the computation
ssc.start()

ssc.awaitTermination ()
```

Example: Word count and Windows – V2

```
from pyspark.streaming import StreamingContext

# Set prefix of the output folders
outputPathPrefix="resSparkStreamingExamples"

#Create  a configuration  object and#set the name  of the applicationconf
SparkConf().setAppName("Streaming word count")

# Create  a Spark Context  object
sc = SparkContext(conf=conf)

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 5)

# Set the checkpoint folder (it is needed by some window transformations)
ssc.checkpoint("checkpointfolder");
```

Example: Word count and Windows – V2

```
# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Apply a chain of  transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))

# reduceByKeyAndWindow is used instead of reduceByKey
# The durantion of the window is also specified
wordsCounts = wordsOnes\
.reduceByKeyAndWindow(lambda v1, v2: v1+v2, lambda vnow, vold: vnow-vold, 15)
```

Example: Word count and Windows – V2

```
# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Apply a chain of  transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(laml
```

In this solution the inverse function is also specified in order to compute the result incrementally

```
# reduceByKeyAndWindow is used instead of reduceByKey
# The durantion of the window is also specified
wordsCounts = wordsOnes\
.reduceByKeyAndWindow(lambda v1, v2: v1+v2, lambda vnow, vold: vnow-vold, 15)
```

Example: Word count and Windows – V2

```
# Print the num. of occurrences of each word of the current window
# (only 10 of them)
wordsCounts.pprint()

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

#Start the computation
ssc.start()

# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)

ssc.stop(stopSparkContext=False)
```

Stateful Computation

UpdateStateByKey Transformation

- The updateStateByKey transformation allows maintaining a “state” for each key
  - The value of the state of each key is continuously updated every time a new batch is analyzed

UpdateStateByKey Transformation

- The use of updateStateByKey is based on two steps
  - Define the state
    - The data type of the state associated with the keys can be an arbitrary data type
  - Define the state update function
    - Specify with a function how to update the state of a key using the previous state and the new values from an input stream associated with that key

UpdateStateByKey Transformation

- In every batch, Spark will apply the state update function for all existing keys
- For each key, the update function is used to update the value associated with a key by combining the former value and the new values associated with that key
  - For each key, the call method of the “function” is invoked on the list of new values and the former state value and returns the new aggregated value for the considered key

Word count and UpdateStateByKey Transformation

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

Example: Word count - stateful version

- Problem specification
  - Input: a stream of sentences retrieved from localhost:9999
  - Split the input stream in batches of 5 seconds
  - Print on the standard output, every 5 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

Example: Word count - stateful version

```
from pyspark.streaming import StreamingContext

# Set prefix of the output folders
outputPathPrefix="resSparkStreamingExamples"

#Create  a configuration  object and#set the name  of the applicationconf
SparkConf().setAppName("Streaming word count")

# Create  a Spark Context  object
sc = SparkContext(conf=conf)

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 5)

# Set the checkpoint folder (it is needed by some window transformations)
ssc.checkpoint("checkpointfolder")
```

Example: Word count - stateful version

```
# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Apply a chain of  transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))
```

Example: Word count - stateful version

```
# Define the function that is used to update the state of a key at a time
def updateFunction(newValues, currentCount):
    if currentCount is None:
        currentCount = 0

    # Sum the new values to the previous state for the current key
    return sum(newValues, currentCount)


# DStream made of cumulative counts for each key that get updated in every batch
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)
```

Example: Word count - stateful version

```
# Define the function that is used to update the state of a key at a time
def updateFunction(newValues, currentCount):
    if currentCount is None:
        currentCount = 0

    # Sum the new values to the previous state for the current key
    return sum(newValues, currentCount)
```

This function is invoked one time for each key

```
# DStream made of cumulative counts for each key that get updated in every batch
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)
```

Example: Word count - stateful version

```
# Define the function that is used to update the state of a key at a time
def updateFunction(newValues, currentCount):
    if currentCount is None:
        currentCount = 0

    # Sum the new values to the previous state for the current key
    return sum(newValues, currentCount)


# DStream made of cumulative counts for each key that get updated in every batch
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)
```

Current state/value for the current key

Example: Word count - stateful version

```
# Define the function that is used to update the state of a key at a time
def updateFunction(newValues, currentCount):
    if currentCount is None:
        currentCount = 0
```

List of new integer values for the current key

```
    # Sum the new values to the previous state for the current key
    return sum(newValues, currentCount)


# DStream made of cumulative counts for each key that get updated in every batch
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)
```

Example: Word count - stateful version

```
# Define the function that is used to update the state of a key at a time
def updateFunction(newValues, currentCount):
    if currentCount is None:
        currentCount = 0

    # Sum the new values to the previous state for the current key
    return sum(newValues, currentCount)
```

Combine current state and new values

```
# DStream made of cumulative counts for each key that get updated in every batch
totalWordsCounts = wordsOnes.updateStateByKey(updateFunction)
```

Example: Word count - stateful version

```
# Print the num. of occurrences of each word of the current window
# (only 10 of them)
totalWordsCounts.pprint()

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

#Start the computation
ssc.start()

# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)

ssc.stop(stopSparkContext=False)
```

Transform transformation

Transform transformation

- Some types of transformations are not available for DStreams
  - E.g., sortBy, sortByKey, distinct()
- Moreover, sometimes you need to combine DStreams and RDDs
  - 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
- The **transform()** transformation can be used in these situations

Transform transformation

- **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 apply arbitrary RDD operations on the DStream

Example: Word count – Use of transform

- Problem specification
  - Input: a stream of sentences retrieved from localhost:9999
  - Split the input stream in batches of 5 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 decreasing number of occurrences (per batch)**
  - Store the results also in an HDFS folder

Example: Word count – Use of transform

```
from pyspark.streaming import StreamingContext

# Set prefix of the output folders
outputPathPrefix="resSparkStreamingExamples"

#Create  a configuration  object and#set the name  of the applicationconf
SparkConf().setAppName("Streaming word count")

# Create  a Spark Context  object
sc = SparkContext(conf=conf)

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 5)

# Create a (Receiver) DStream that will connect to localhost:9999
lines = ssc.socketTextStream("localhost", 9999)
```

Example: Word count – Use of transform

```
# Apply a chain of  transformations to perform the word count task
# The returned RDDs are DStream RDDs
words = lines.flatMap(lambda line: line.split(" "))

wordsOnes = words.map(lambda word: (word, 1))

wordsCounts = wordsOnes.reduceByKey(lambda v1, v2: v1+v2)

# Sort the content/the pairs by decreasing value (# of occurrences)
wordsCountsSortByKey = wordsCounts\
.transform(lambda batchRDD: batchRDD.sortBy(lambda pair: -1*pair[1]))
```

Example: Word count – Use of transform

```
# Print the result on the standard output
wordsCountsSortByKey.pprint()

# Store the result in HDFS
wordsCountsSortByKey.saveAsTextFiles(outputPathPrefix, "")


#Start the computation
ssc.start()

# Run this application for 90 seconds
ssc.awaitTerminationOrTimeout(90)

ssc.stop(stopSparkContext=False)
```