# Big data: architectures and data analytics

# Spark Streaming

# 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

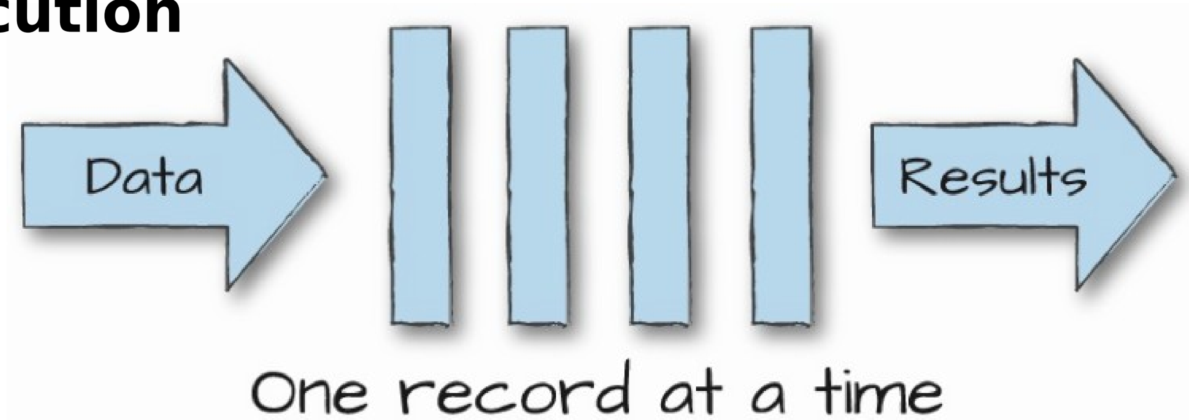
- Vastly **higher throughput** in data processing
- **Low latency**: application respond quickly (e.g., in seconds). It can keep states in memory
- **More efficient** in updating a result than repeated batch jobs, because it automatically incrementalizes the computation

# Requirements & Challenges

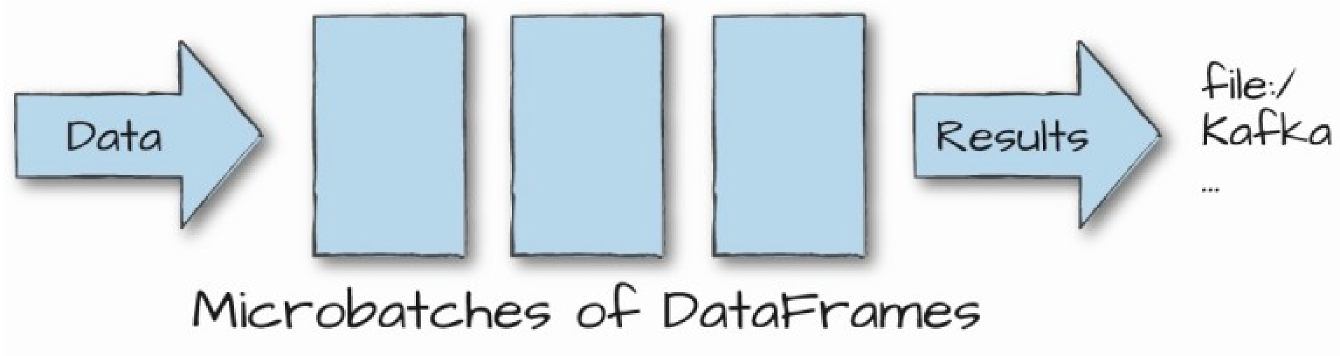
- Scalable to large clusters
- Maintaining large amounts of state
- Writing data transactionally to output systems
- Fault-tolerance in stateful computations
- Simple programming model

# Continuous vs Micro-batch execution

**Continuous execution**




**Micro-batch execution**



# Continuous vs Micro-batch execution

- Continuous processing offer the lowest possible latency, because each node responds immediately to a new message.
- Micro-batch execution has higher maximum throughput, because incur in less overhead (do not “reduce” per-record)

# Spark Streaming

- Spark Streaming is a framework for large scale stream processing
  - Scales to hundreds of nodes
  - Can achieve second scale latencies
  - Provides simple APIs for implementing complex algorithm
  - Can absorb live data streams from Kafka, Flume, ZeroMQ, Twitter, …

# Spark Streaming

# Spark Streaming APIs

- Spark includes two streaming APIs
  - The earlier **Discretized Stream API** is micro-batch oriented (Dstream), that works with **RDD**-like structures
  - The newer **Structured Streaming API** adds higher-level optimizations, event time, and support for continuous processing. It works with **DataFrame**-like structures

# 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**

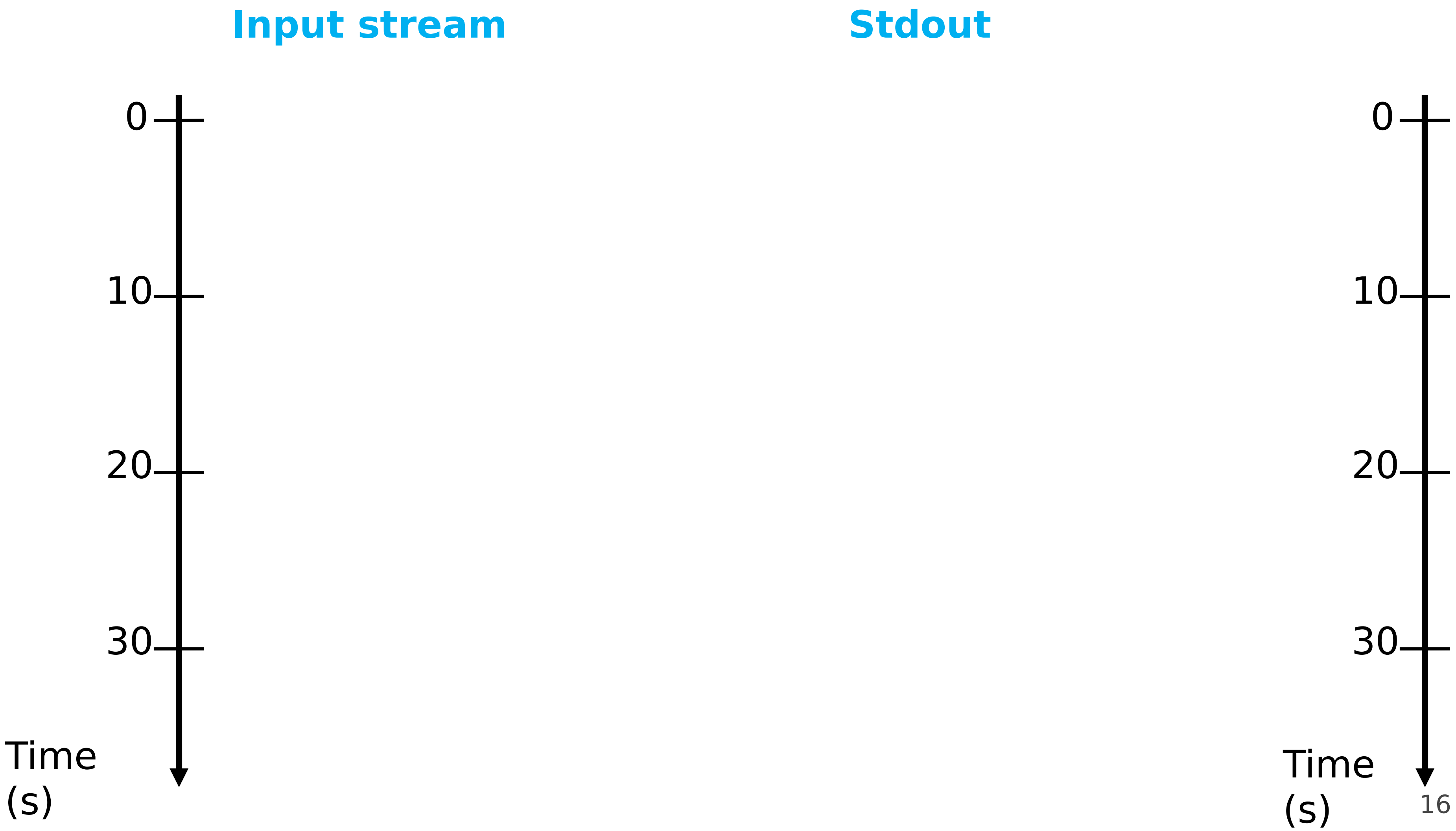
# Discretized Stream Processing

- 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 problem for each batch of 10 seconds

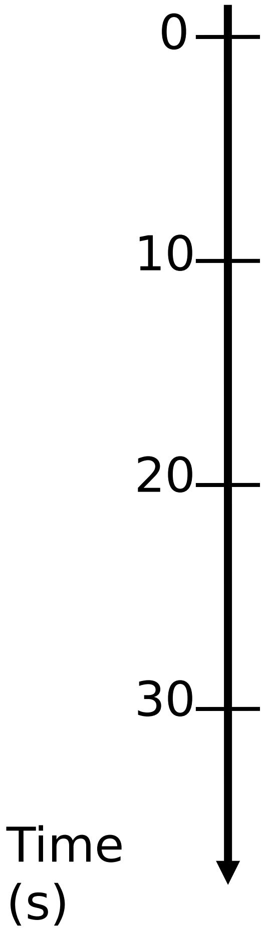
# Word count – Spark Streaming version



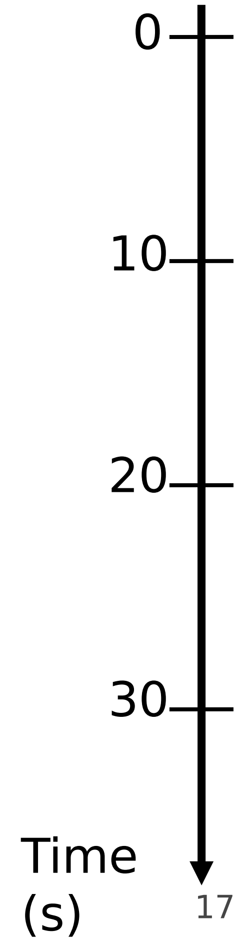
# Word count – Spark Streaming version

**Input stream**

**Stdout**




Test Spark streaming



# Word count – Spark Streaming version

**Input stream**

**Stdout**

0

Test Spark streaming
Second sentence Spark
streaming

10

20

30

Time
(s)

0

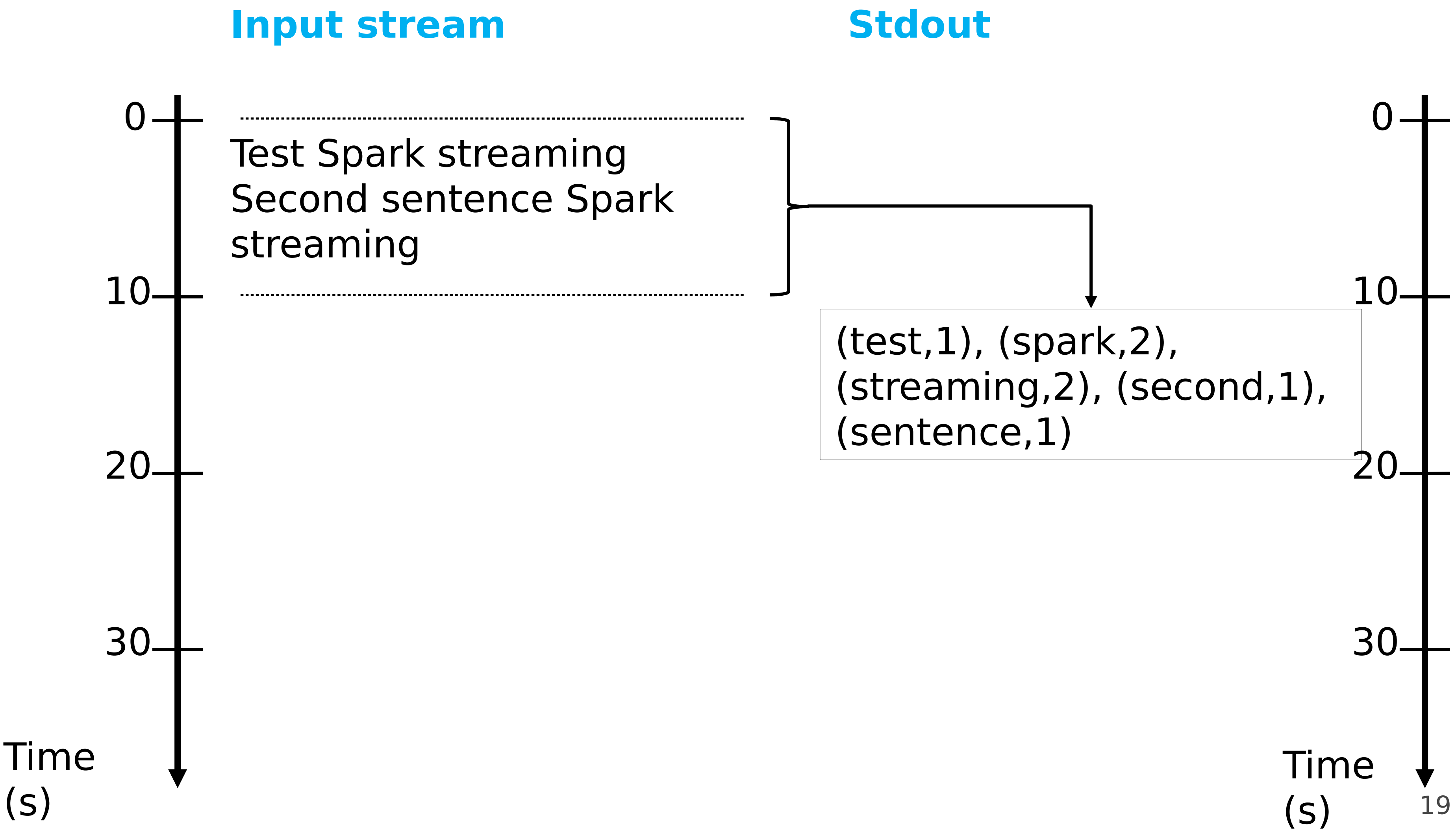
10

20

30

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

30

Time (s)

**Stdout**

0

10

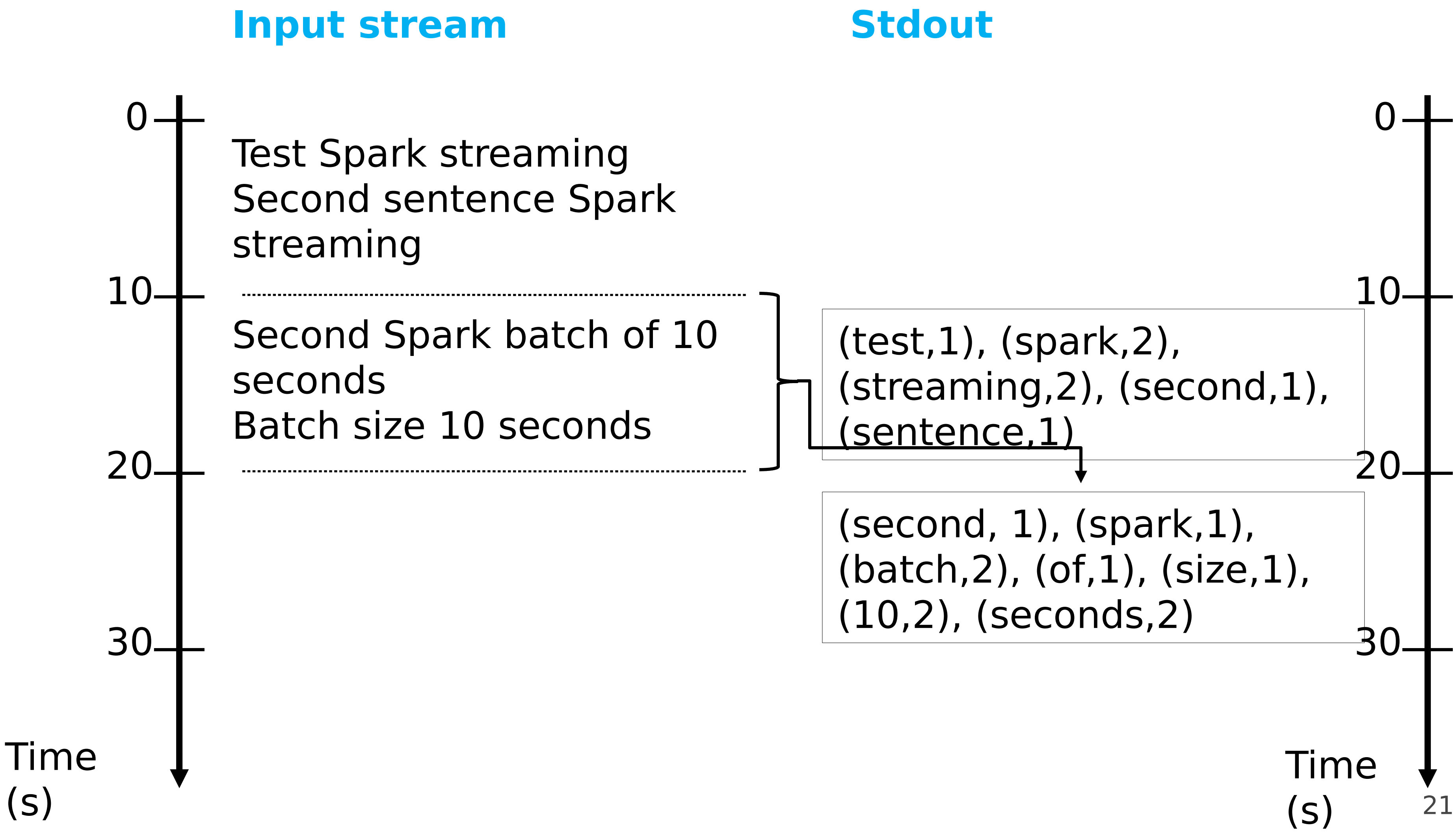
(test,1), (spark,2), (streaming,2), (second,1), (sentence,1)

20

30

Time (s)

# Word count – Spark Streaming version



# Word count – Spark Streaming version

| Time (s) | Input stream | Stdout |
|---|---|---|
| 0 | | |
| | Test Spark streaming<br>Second sentence Spark streaming | |
| 10 | | |
| | Second Spark batch of 10 seconds<br>Batch size 10 seconds | (test,1), (spark,2), (streaming,2), (second,1), (sentence,1) |
| 20 | | |
| | Third batch | (second, 1), (spark,1), (batch,2), (of,1), (size,1), (10,2), (seconds,2) |
| 30 | | |

# Word count – Spark Streaming version

| Time (s) | Input stream | Stdout |
|---|---|---|
| 0–10 | Test Spark streaming<br>Second sentence Spark streaming | |
| 10–20 | Second Spark batch of 10 seconds<br>Batch size 10 seconds | (test,1), (spark,2), (streaming,2), (second,1), (sentence,1) |
| 20–30 | Third batch | (second, 1), (spark,1), (batch,2), (of,1), (size,1), (10,2), (seconds,2) |
| 30– | | (third,1), (batch, 1) |

# 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

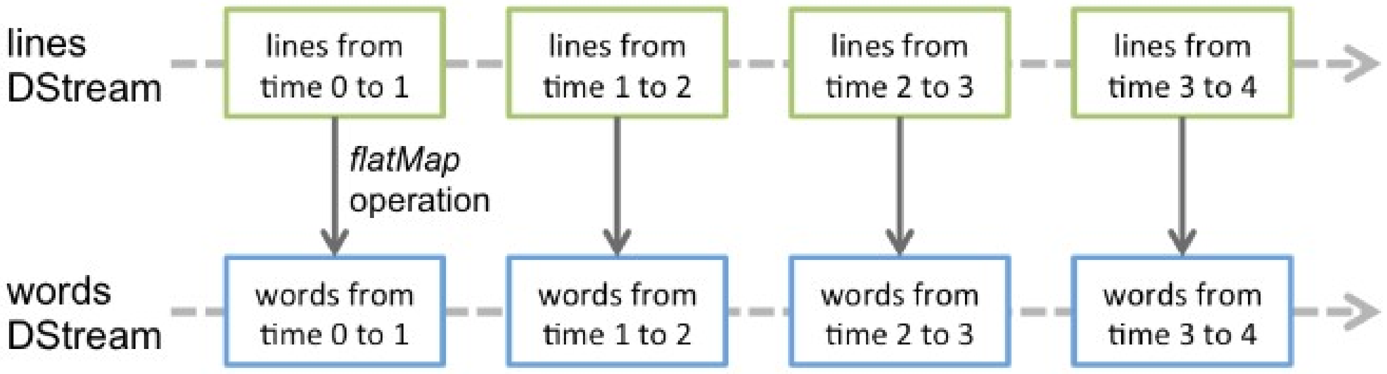
- **Transformations**
  - Modify data from one DStream to another
  - Standard RDD operations
    - map, countByValue, reduce, join, …
  - Window and Stateful operations
    - window, countByValueAndWindow, …
- **Output Operations (actions)**
  - Send data to external entity
    - saveAsHadoopFiles, saveAsTextFile, …

# Word count – DStreams

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

# 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 included in the module **pyspark.streaming**
- The **batchDuration** parameter specifies the “size” of the batches
- 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.

# 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 **awaitTerminationOrTimeout()** method is used to **run** the application **forever**
  - Until the application is explicitly killed
- The processing can be manually stopped using **streamingContext.stop()**.

# Spark Streaming Context

- 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 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

# 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: (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: 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: 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 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

# 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 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 **func**.
- **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

# 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.

# 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

# 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

# Basic Output Operations (actions) on DStreams

- **pprint()**
  - 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

# Basic Output Operations (actions) on DStreams

- **saveAsTextFiles(prefix, [suffix])**
  - Save each RDD in 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(“Prefix”, "txt");

# Example: Word count – Spark Streaming version

- Problem specification
  - Input: a stream of sentences retrieved from an hdfs folder
  - 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

# Example: Word count – Spark Streaming version

```
from pyspark.streaming import StreamingContext
import sys

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

#Create a (Receiver) DStream that will monitor the folder
lines = ssc.textFileStream("StreamingFolder/")
```

# Example: Word count – Spark Streaming version

```
#Apply the "standard" transformations to perform the word count task
#However, the "returned" RDDs are DStream/PairDStream RDDs
counts = lines.flatMap(lambda line: line.split(" ")) \
            .map(lambda word: (word, 1)) \
            .reduceByKey(lambda a, b: a+b)

counts.pprint()

counts.saveAsTextFiles("Output/Streaming", "txt");

ssc.start()
ssc.awaitTermination()
```

# Example: Word count – Spark Streaming version

**Output**

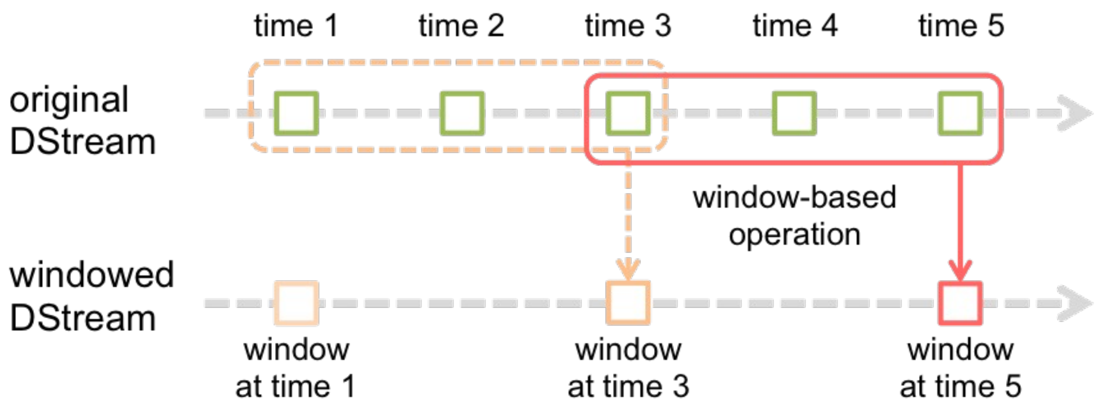
```
-------------------------------------------
Time: 2019-12-12 23:14:40
-------------------------------------------
('streaming', 2)
('Test', 1)
('Second', 1)
('Spark', 2)
('sentence', 1)

-------------------------------------------
Time: 2019-12-12 23:14:50
-------------------------------------------
('of', 1)
('batch', 1)
('Second', 1)
('10', 2)
('Batch', 1)
('seconds', 2)
('Spark', 1)
('size', 1)
-------------------------------------------
Time: 2019-12-12 23:15:00
-------------------------------------------
('', 2)
('Third', 1)
('batch', 1)
```

# Window operation

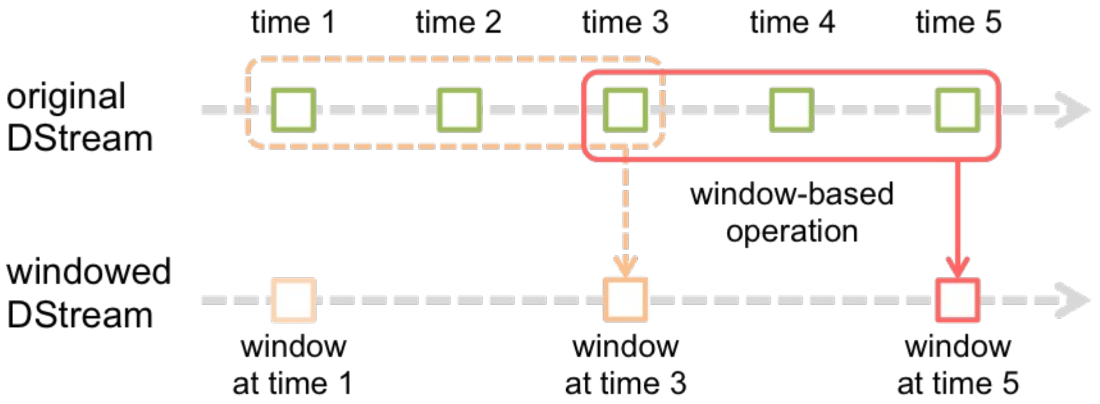
- Spark Streaming also provides windowed computations
  - It allows you to apply transformations over a sliding window of data
    - Each window contains a set of batches of the input stream
    - Windows can be overlapped
      - i.e., the same batch can be included in many consecutive windows

# Window operation




- 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 example above, 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

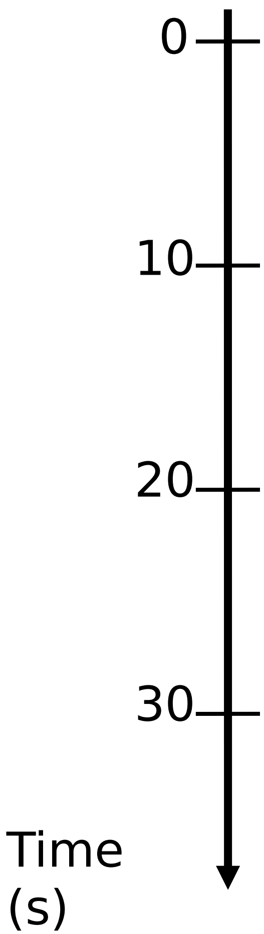
# Word count and Window

- 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

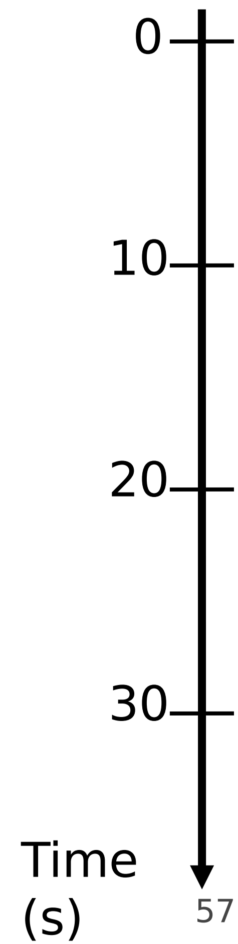
# Word count and Window

**Input stream**

**Stdout**




Test Spark streaming
Second sentence Spark
streaming



# Word count and Window

**Input stream**

**Stdout**

0

Test Spark streaming
Second sentence Spark streaming

10

Second Spark batch of 10 seconds
Batch size 10 seconds

20

30

Time (s)

0

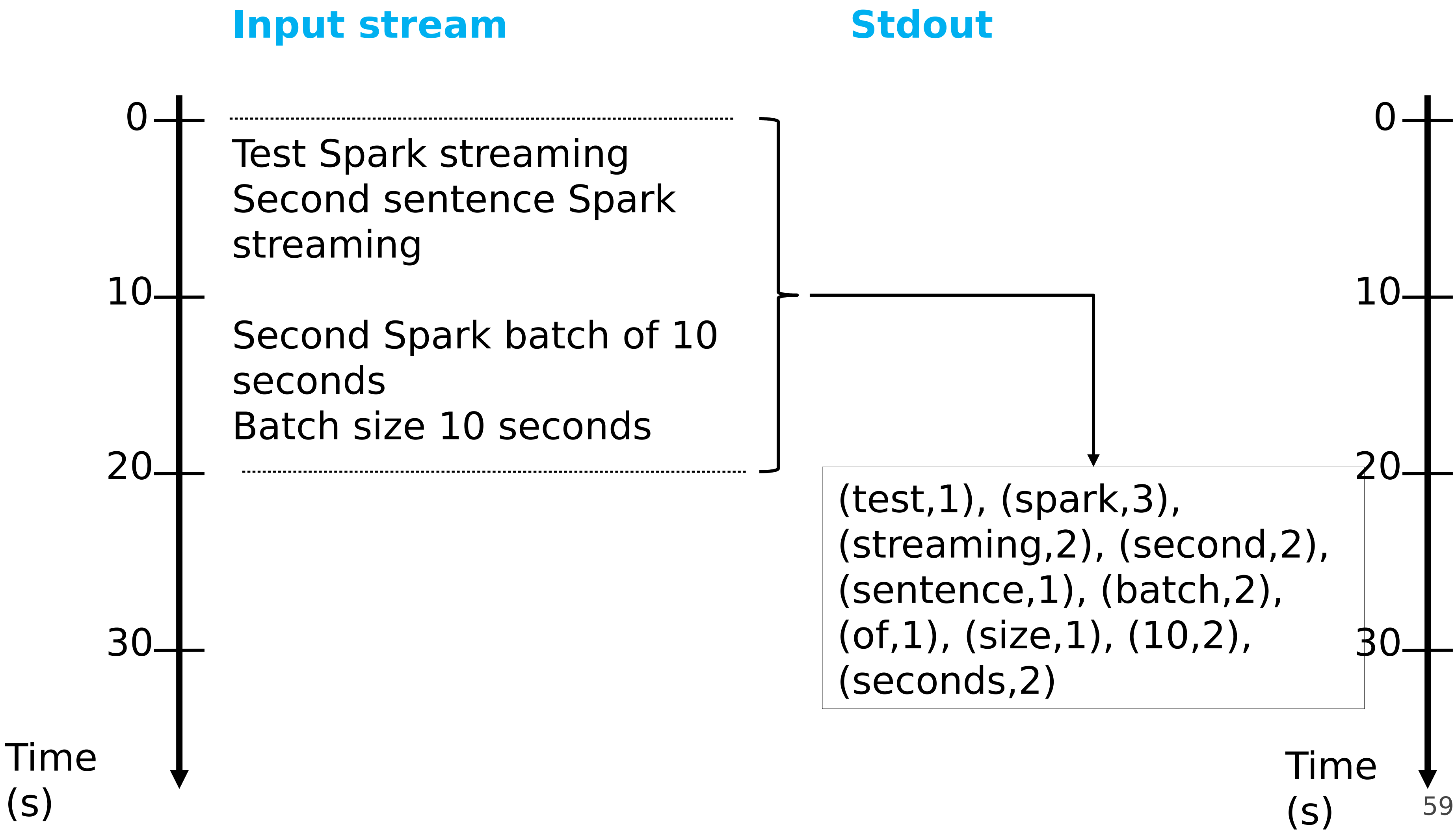
10

20

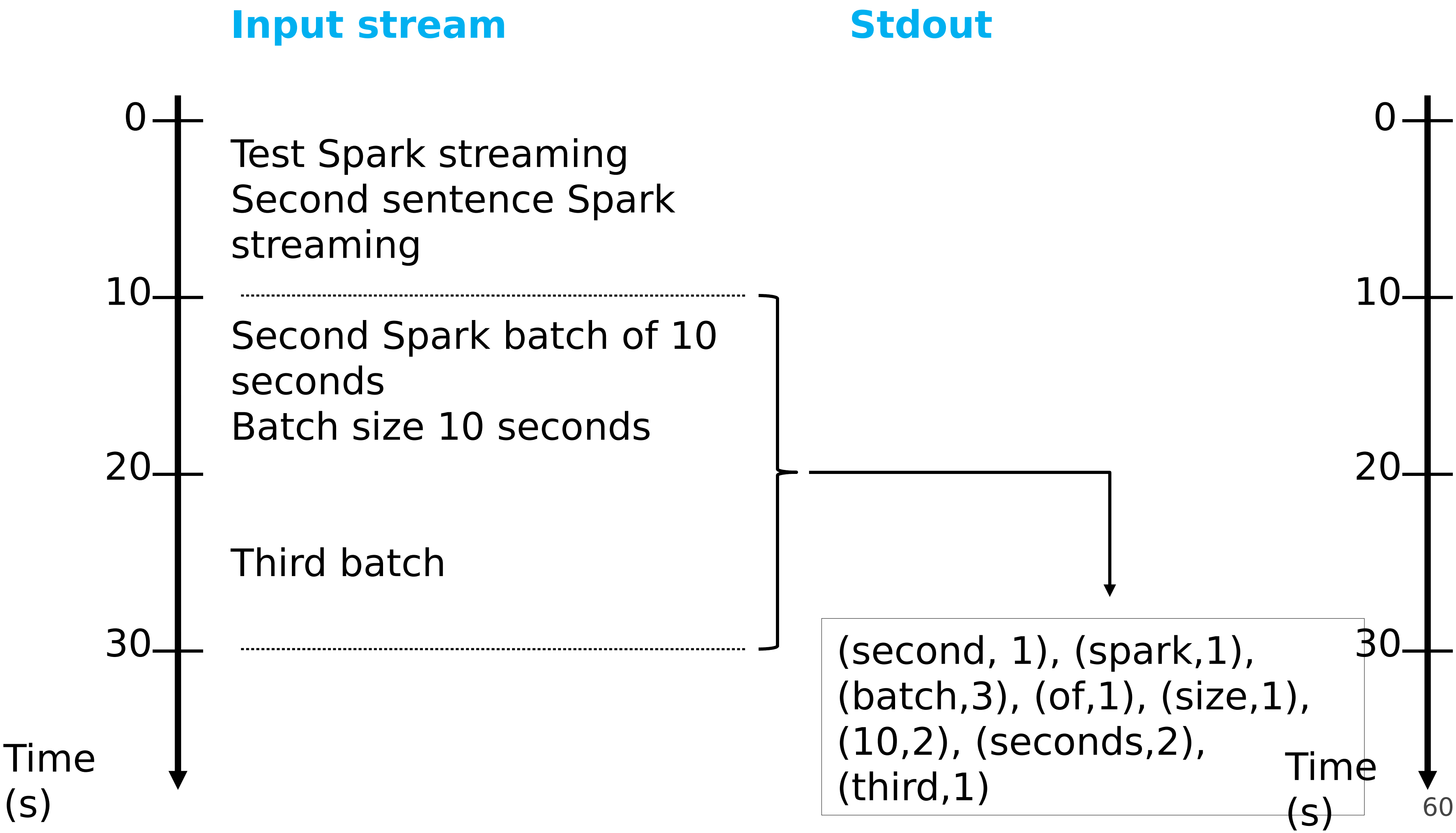
30

Time (s)

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

# Basic Window Transformations

- **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
- **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

# Basic Window Transformations

- **reduceByKeyAndWindow(func, windowLength, slideInterval)**
  - 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 **over batches in a sliding window**
    - The window length and the sliding window step are specified as parameters of this invocation

# Checkpoints

- A streaming application must operate 24/7 and hence must be resilient to failures unrelated to the application logic (e.g., system failures)
- Spark Streaming needs to checkpoint information to a fault- tolerant storage system such that it can recover from failures
- 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 an HDFS folder
  - Split the input stream in batches of 10 seconds
  - Define widows 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

# Example: Word count and Windows

```
from pyspark.streaming import StreamingContext
import sys

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

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

#Create a (Receiver) DStream that will monitor a HDFS folder
lines = ssc.textFileStream("InputStreamingFolder/")

#Apply the "standard" transformations to perform the word count task
#However, the "returned" RDDs are DStream/PairDStream RDDs
wordsOnes = lines.flatMap(lambda line: line.split(" ")).map(lambda word:
    (word, 1))
```

# Example: Word count and Windows

```
#reduceByKeyAndWindow is used instead of reduceByKey
# characteristics of the window is also specified
wordsCounts = wordsOnes.reduceByKeyAndWindow(lambda i1, i2: i1 +
    i2,30,10);

#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("/Streaming", "txt");

#Start the computation
ssc.start();
ssc.awaitTerminationOrTimeout(120000); #120 seconds
ssc.close();
```

# Example: Word count and Windows

**Output**

```
------------------------------------------
Time: 2019-12-12 23:14:40
------------------------------------------
(test,1)
(spark,3)
(streaming,2)
(second,2)
(sentence,1)
(batch,2)
(of,1)
(size,1)
(10,2)
(seconds,2)
------------------------------------------
Time: 2019-12-12 23:14:50
------------------------------------------
(second, 1)
(spark,1)
(batch,3)
(of,1)
(size,1)
(10,2)
(seconds,2)
(third,1)
------------------------------------------
Time: 2019-12-12 23:15:00
------------------------------------------
('Third', 1)
('batch', 1)
```

# UpdateStateByKey Transformation

- The updateStateByKey transformation allows maintaining a state
  - The value of the state 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 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

# 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 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

# Example: Word count (stateful version)

- Problem specification
  - Input: a stream of sentences retrieved from a folder
  - 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

# Example: Word count (stateful version)

```
from pyspark.streaming import StreamingContext
import sys

# Create a Spark Streaming Context object
ssc = StreamingContext(sc, 10)

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

#Create a (Receiver) DStream that will monitor a HDFS folder
lines = ssc.textFileStream("StreamingFolder/")

#Apply the "standard" transformations to perform the word count task
#However, the "returned" RDDs are DStream/PairDStream RDDs
wordsOnes = lines.flatMap(lambda line: line.split(" ")).map(lambda word:
    (word, 1))
```

# Example: Word count (stateful version)

```
# RDD with initial state (key, value) pairs
initialStateRDD = sc.parallelize([])

#Iterates over the new values and sum them to the previous state
def updateFunc(NewValues, state):
    if state is None:
        state=0
    return sum(NewValues,state)

#DStream made of get cumulative counts that get updated in every batch
#state.or(0)  returns the value of State or the default value 0 if state is not defined
totalWordsCounts = wordsOnes.updateStateByKey(updateFunc,
     initialRDD=initialStateRDD)

#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("fOutputStreamingFolder/Streaming", "txt");
```

# Example: Word count (stateful version)

```
# RDD with initial state (key, value) pairs
initialStateRDD = sc.parallelize([])

#Iterates over the new values and sum them to the previous state
def updateFunc(NewValues, state):
    if state is None:
        state=0
    return sum(NewValues,state)

#DStream made of get cumulative counts that get updated in every batch
#state.or(0)  returns the value of State or the default value 0 if state is not defined
totalWordsCounts = wordsOnes.updateStateByKey(updateFunc,
     initialRDD=initialStateRDD)

#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("fOutputStreamingFolder/Streaming", "txt");
```

It is invoked one time for each key

# Example: Word count (stateful version)

```
# RDD with initial state (key, value) pairs
initialStateR
```

List of new integer values (1) for the current key

```
#Iterates over the new values and sum them to the previous state
def updateFunc(NewValues, state):
    if state is None:
        state=0
    return sum(NewValues,state)

#DStream made of get cumulative counts that get updated in every batch
#state.or(0)  returns the value of State or the default value 0 if state is not defined
totalWordsCounts = wordsOnes.updateStateByKey(updateFunc,
     initialRDD=initialStateRDD)

#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("fOutputStreamingFolder/Streaming", "txt");
```

# Example: Word count (stateful version)

```
# RDD with initial state (key, value) pairs
initialStateRDD = sc.parallelize([])

#Iterates over the new values and sum them to the previous state
def updateFunc(NewValues, state):
    if state is None:
        state=0
    return sum(NewValues,state)
```

Current “state” of the current key,
i.e., number of occurrences in the previous part of the stream

```
totalWordsCounts = wordsOnes.updateStateByKey(updateFunc,
     initialRDD=initialStateRDD)

#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("fOutputStreamingFolder/Streaming", "txt");
```

# Example: Word count (stateful version)

```
#Start the computation
ssc.start();
ssc.awaitTerminationOrTimeout(120000);
ssc.close();
```

# Example: Word count (stateful version)

**Output**

```
-------------------------------------------
Time: 2019-12-13 00:10:40
-------------------------------------------
('streaming', 2)
('Test', 1)
('Second', 1)
('Spark', 2)
('sentence', 1)
-------------------------------------------
Time: 2019-12-13 00:10:50
-------------------------------------------
('', 1)
('of', 1)
('batch', 1)
('streaming', 2)
('Test', 1)
('Second', 2)
('10', 2)
('Batch', 1)
('seconds', 2)
('Spark', 3)
...
-------------------------------------------
Time: 2019-12-13 00:11:00
-------------------------------------------
('', 3)
('of', 1)
('batch', 2)
('Third', 1)
('streaming', 2)
('Test', 1)
('Second', 2)
('10', 2)
('Batch', 1)
('seconds', 2)
...
```

# MLib operations

- You can use machine learning algorithms provided by MLlib with Spark Streaming.
- There are specific streaming machine learning algorithms (e.g. Streaming Linear Regression, Streaming KMeans, etc.)
- They can simultaneously learn from the streaming data as well as apply the model on the streaming data.
- Beyond these, for all machine learning algorithms, you can learn a model offline (i.e. using historical data) and then apply the model online on streaming data

# Documentation

- Spark Streaming Programming Guide

  https://spark.apache.org/docs/latest/streaming-programming-guide.html

- Pyspark.streaming module

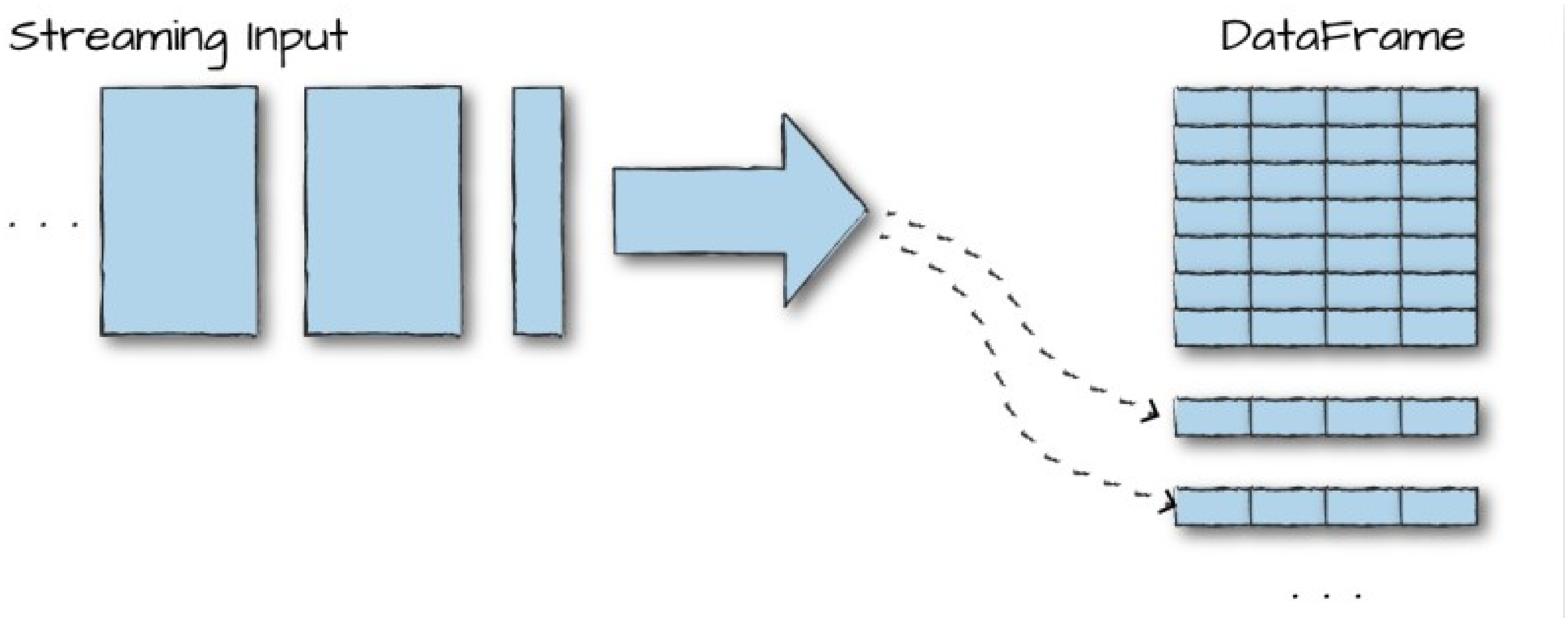
  https://spark.apache.org/docs/2.2.0/api/python/pyspark.streaming.html

# Structured Streaming

# Structured streaming

- Structured Streaming is a stream processing framework built on the **Spark SQL engine**
- Rather than introducing a separate API, Structured Streaming uses the existing structured APIs in Spark (DataFrames, Datasets, and SQL)
- Users express a streaming computation in the same way as batch computation on static data
- The Structured Streaming engine will take care of running your query **incrementally and continuously** as new data arrives into the system

# Structured streaming



# Structured streaming

- Treat a stream of data as a table to which data is continuously appended
- The job then periodically checks for new input data, process it, updates internal state and its result
- No need to change query code when doing batch or stream processing
- Specify only whether to run that query in a batch or streaming fashion
- It will be run in a fault-tolerant fashion

# Transformations and actions

- Structured Streaming maintains the same concept of transformations and actions of Dataframe
- Transformations are the same of Dataframes
- There is generally only one action available: starting a stream, which will then run continuously and output results.

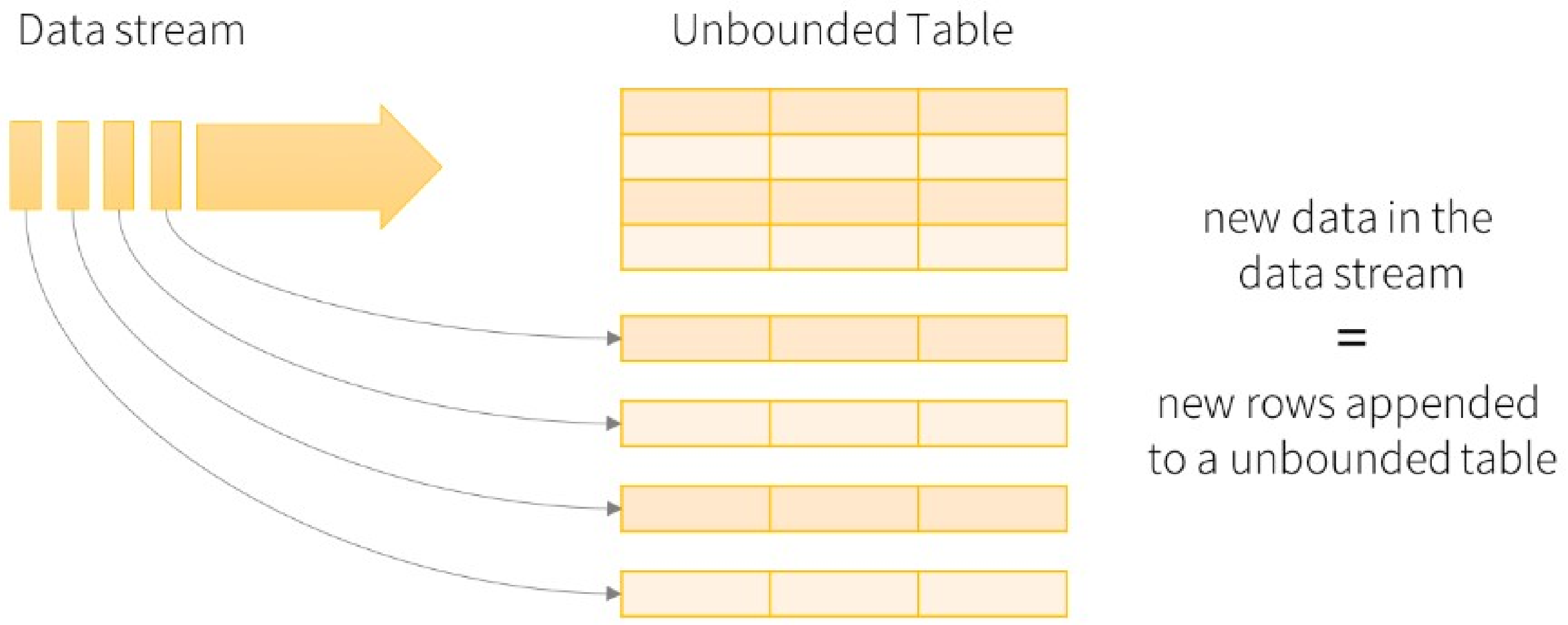
# Output modes

- Define how we want Spark to write data
- Output modes supported:
  - Append (only add new records to the output sink)
  - Update (update changed records in place)
  - Complete (rewrite the full output)

# Triggers

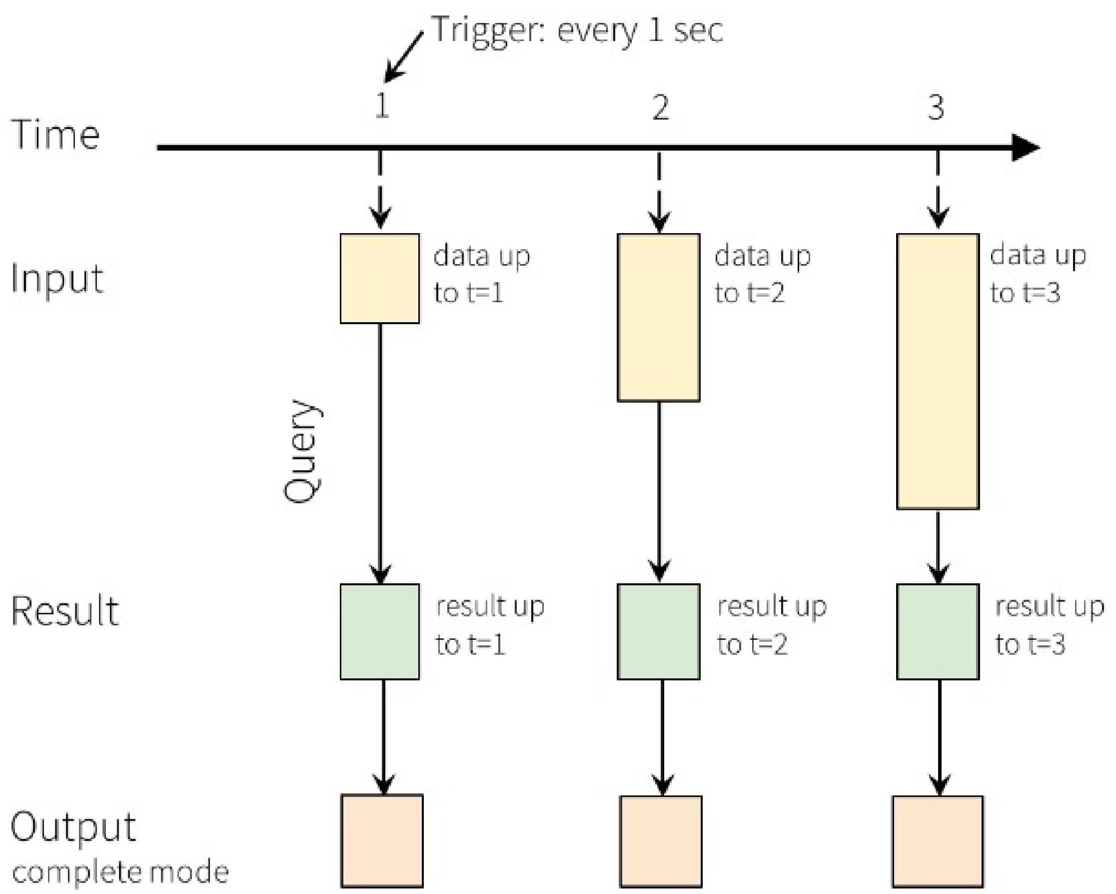
- Triggers define when data is output
- When Structured Streaming should check for new input data and update its result
- By default, as soon as it has finished processing the last group of input data
- Supports triggers based on processing time (only look for new data at a fixed interval)

# Key concepts




Data stream as an unbounded table

# Key concepts




Programming Model for Structured Streaming

# Example: Word count with structured streaming

- Problem specification
  - Input: a stream of sentences retrieved from a folder
  - Print on the standard output, **as soon as new data is available**, the occurrences of each word appearing in the stream (from time 0 to the current time)

# Example: Word count with structured streaming

```
from pyspark.sql.functions import explode
from pyspark.sql.functions import split


# Create DataFrame representing the stream of input lines
lines = spark \
    .readStream \
    .text("StreamingFolder/")

# Split the lines into words
words = lines.select(explode(split(lines.value, " ")).alias("word"))

# Generate running word count
wordCounts = words.groupBy("word").count()
```

# Example: Word count with structured streaming

```
 # Start running the query that prints the running counts to the console
print("Start")

    query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

# Example: Word count with structured streaming

**Output**

```
-------------------------------------------
Batch: 0
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    1|
| spark|    1|
+------+-----+
-------------------------------------------
Batch: 1
-------------------------------------------
+------+-----+
| value|count|
+------+-----+
|apache|    2|
| spark|    1|
|hadoop|    1|
+------+-----+
...
```

# Documentation

- Spark Structured Streaming Programming Guide

  https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

- Structured Streaming examples

  https://github.com/apache/spark/tree/v2.4.4/examples/src/main/python/sql/streaming