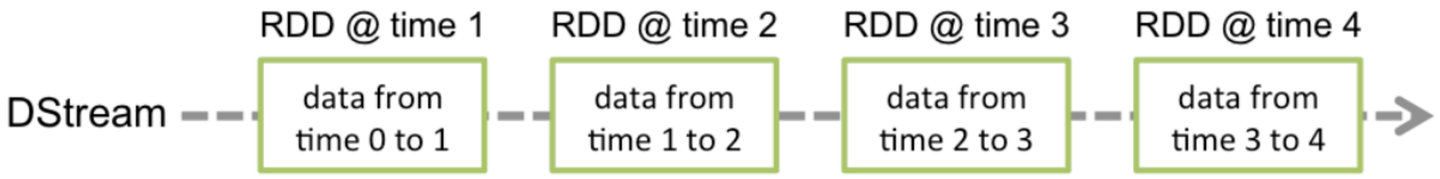
Streaming Spark

Spark Streaming provides a high-level abstraction called *discretized stream* or *DStream*, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.



Socket e DStream

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

# Create a local StreamingContext with two working thread and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)
```

Using this context, we can create a DStream that represents streaming data from a TCP source, specified as hostname (e.g. `localhost`) and port (e.g. `9999`).

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

This `lines` DStream represents the stream of data that will be received from the data server. Each record in this DStream is a line of text. Next, we want to split the lines by space into words.

TCP Producer

Waits for a producer

Opens a big1.txt

Sends each line on the “conn” socket

```
import socket
from time import sleep

host = 'localhost'
port = 9999

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)
while True:
    print('\nListening for a client at',host , port)
    conn, addr = s.accept()
    print('\nConnected by', addr)
    try:
        print('\nReading file...\n')
        with open('big1.txt') as f:
            for line in f:
                out = line.encode('utf-8')
                print('SENT: ',line)
                conn.send(out)
                sleep(2)
            print('End Of Stream.')
    except socket.error:
        print ('Error Occured.\n\nClient disconnected.\n')
conn.close()
```

Operations on DStreams

```
# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))
```

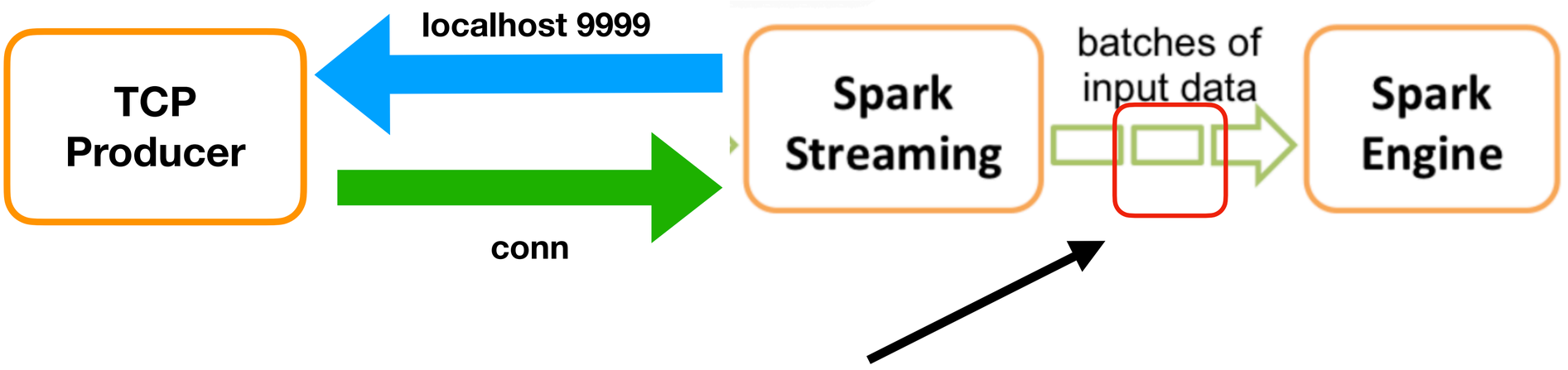
`flatMap` is a one-to-many DStream operation that creates a new DStream by generating multiple new records from each record in the source DStream. In this case, each line will be split into multiple words and the stream of words is represented as the `words` DStream. Next, we want to count these words.

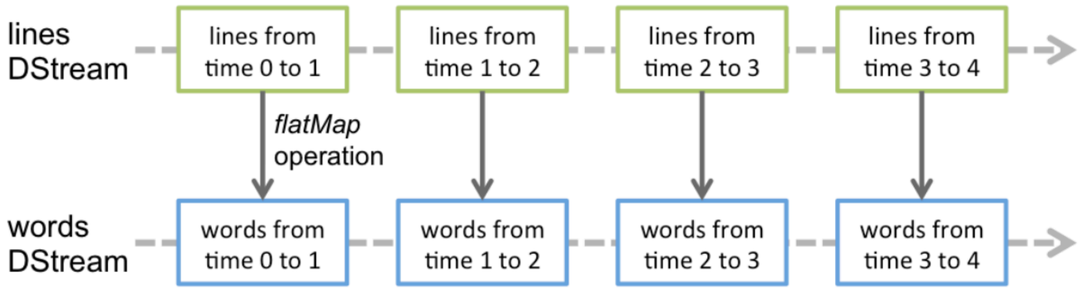
```
# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# Print the first ten elements of each RDD generated in this DStream to the console
wordCounts.pprint()
```

Producer and Consumer

Note that when these lines are executed, Spark Streaming only sets up the computation it will perform when it is started, and no real processing has started yet. To start the processing after all the transformations have been setup, we finally call

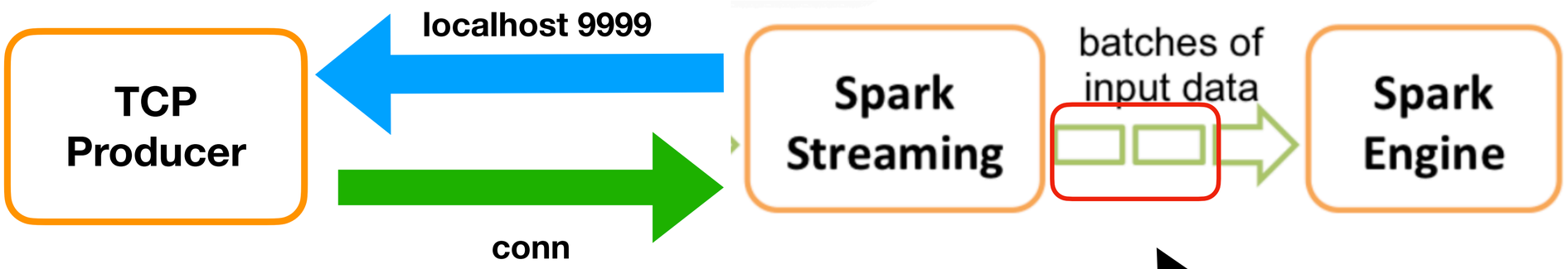
```
ssc.start()             # Start the computation
ssc.awaitTermination()  # Wait for the computation to terminate
```





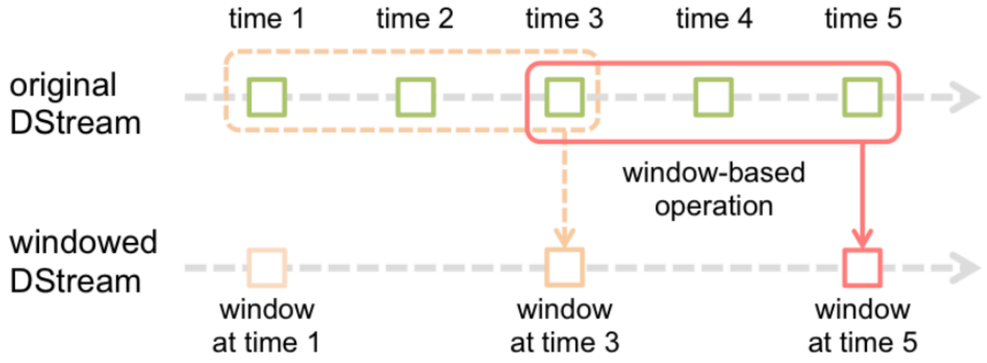
Windows

```
# Reduce last 30 seconds of data, every 10 seconds
windowedWordCounts = pairs.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 10)
```



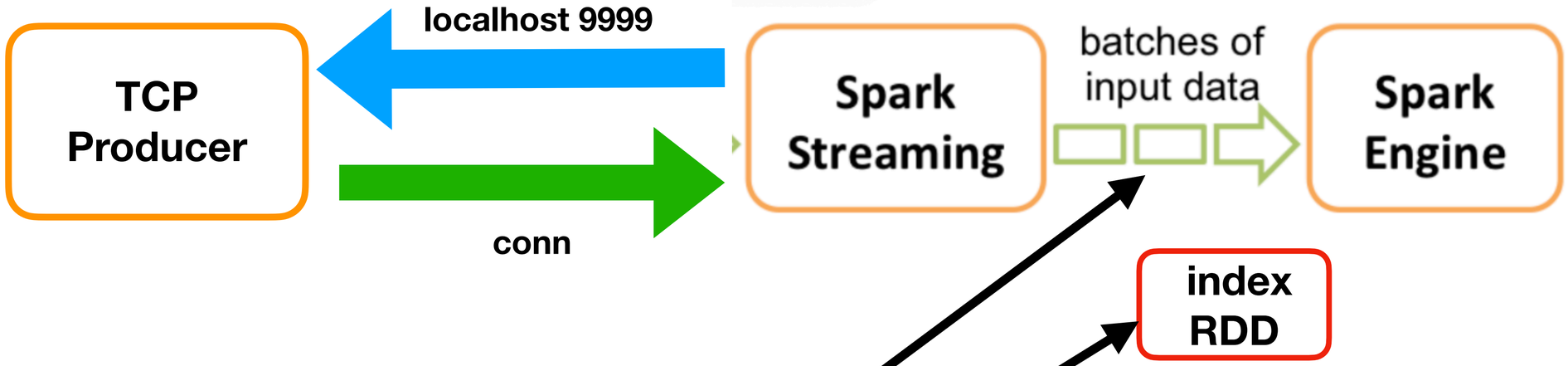
- *window length* - The duration of the window (3 in the figure).
- *sliding interval* - The interval at which the window operation is performed (2 in the figure).

These two parameters must be multiples of the batch interval of the source DStream (1 in the figure)



Transform (e.g. join with a fixed RDD)

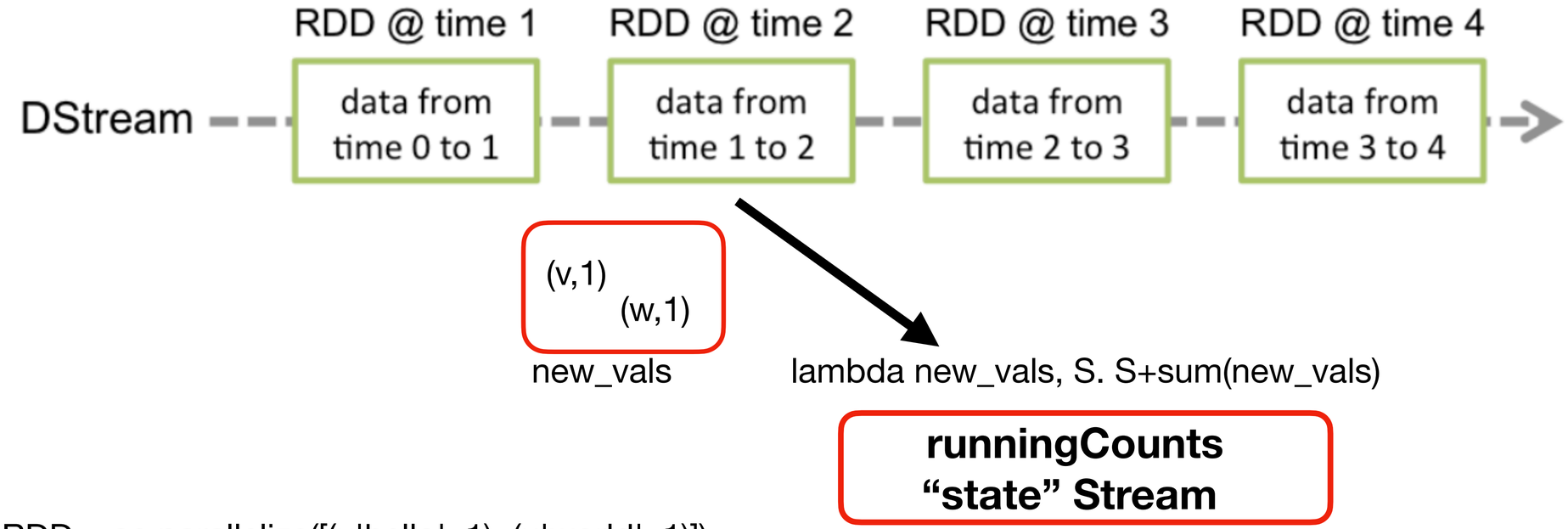
transform(*func*)	Return a new DStream by applying a RDD-to-RDD function to every RDD of the source DStream. This can be used to do arbitrary RDD operations on the DStream.




```
happiest_words = word_counts

  .transform(lambda rdd: index.join(rdd))
  .map(lambda x: (x[0], round(float(x[1][0]) * x[1][1],2)))
  .map(lambda w: (w[1], w[0]))
  .transform(lambda rdd: rdd.sortByKey(False))
```

Stateful Computation

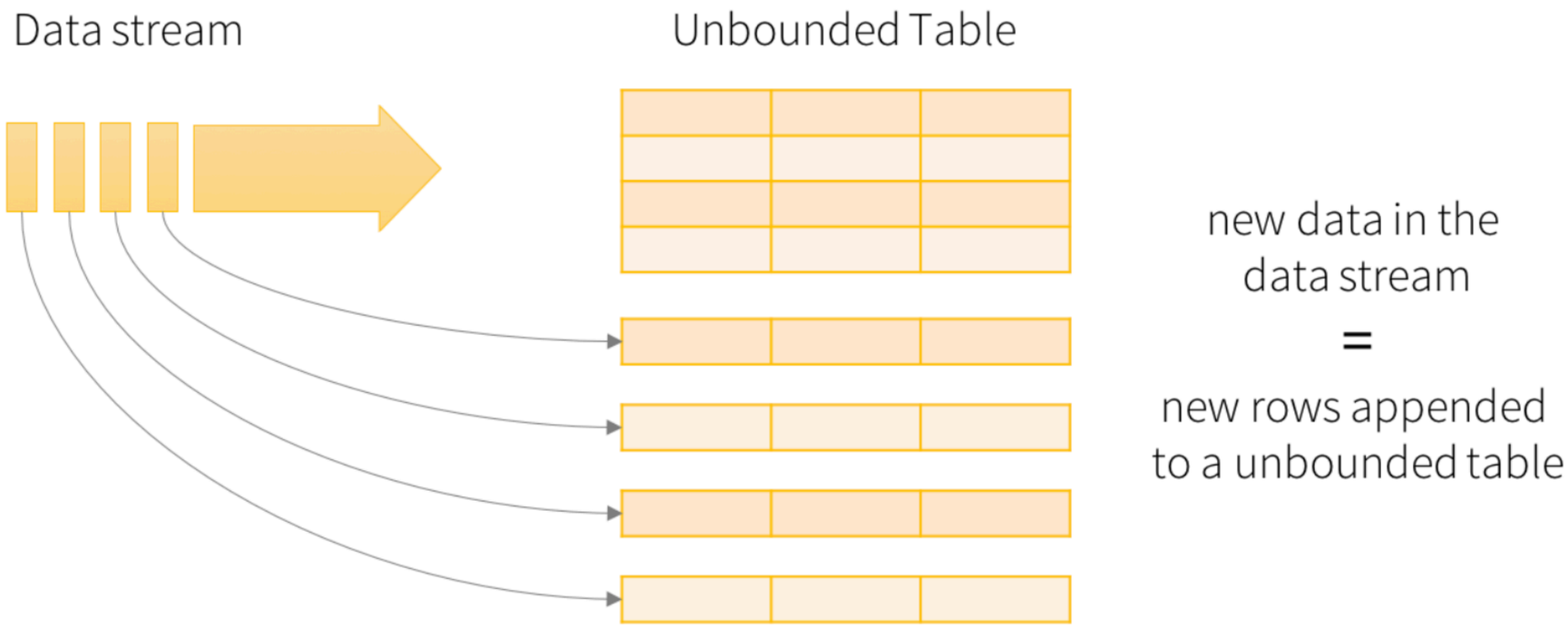
updateStateByKey(*func*)	Return a new "state" DStream where the state for each key is updated by applying the given function on the previous state of the key and the new values for the key. This can be used to maintain arbitrary state data for each key.




```
initialStateRDD = sc.parallelize([(u'hello', 1), (u'world', 1)])

def updateFunc(new_values: Iterable[int], last_sum: Optional[int]) -> int:
  return sum(new_values) + (last_sum or 0)

lines = ssc.socketTextStream("localhost", 9999)
running_counts = lines.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).updateStateByKey(updateFunc,
initialRDD=initialStateRDD)
```

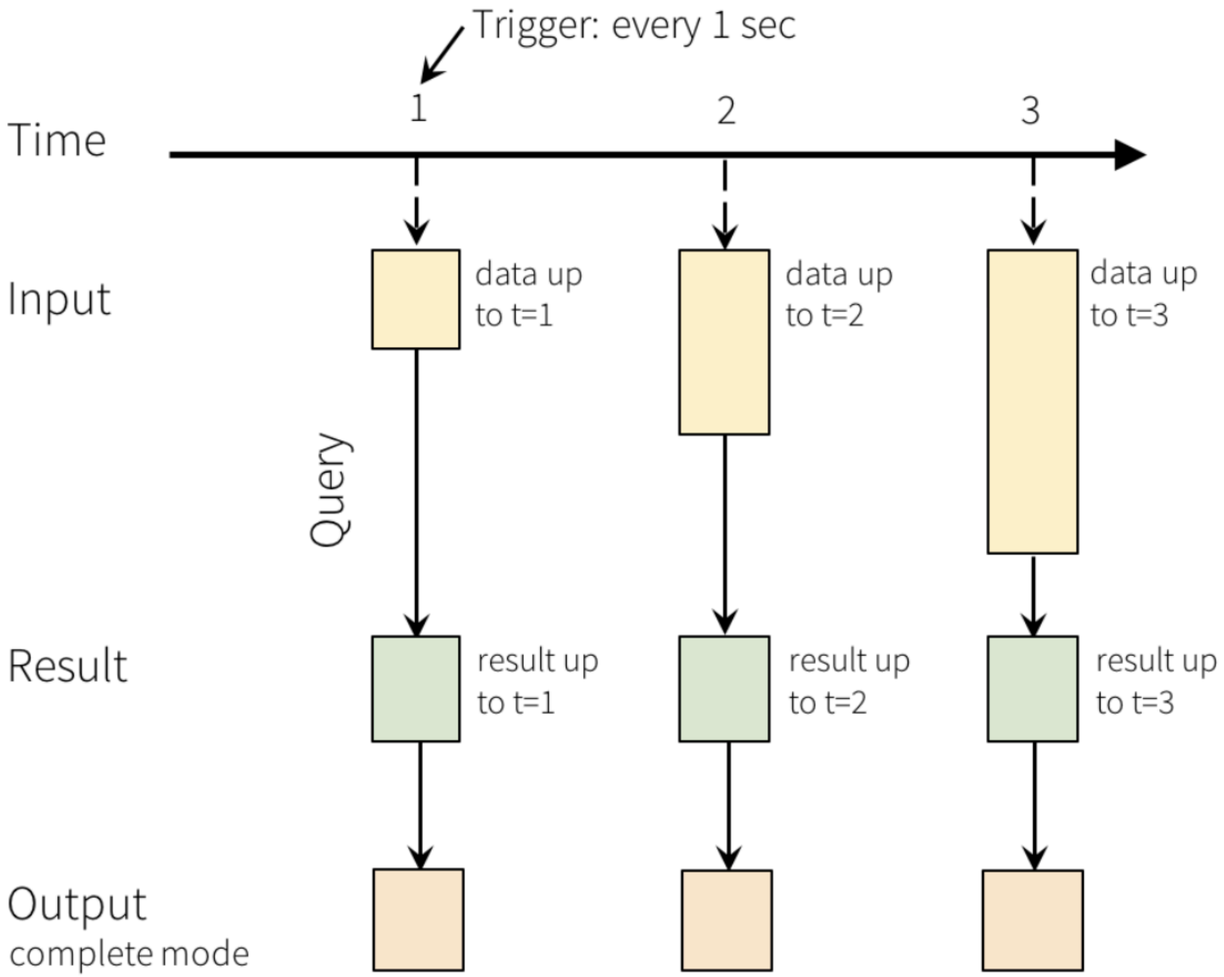
Structured Streaming

Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. The Spark SQL engine takes care of running a program incrementally and continuously and updating the final result as streaming data continues to arrive.
It is possible to use the Dataframes instead of RDDs



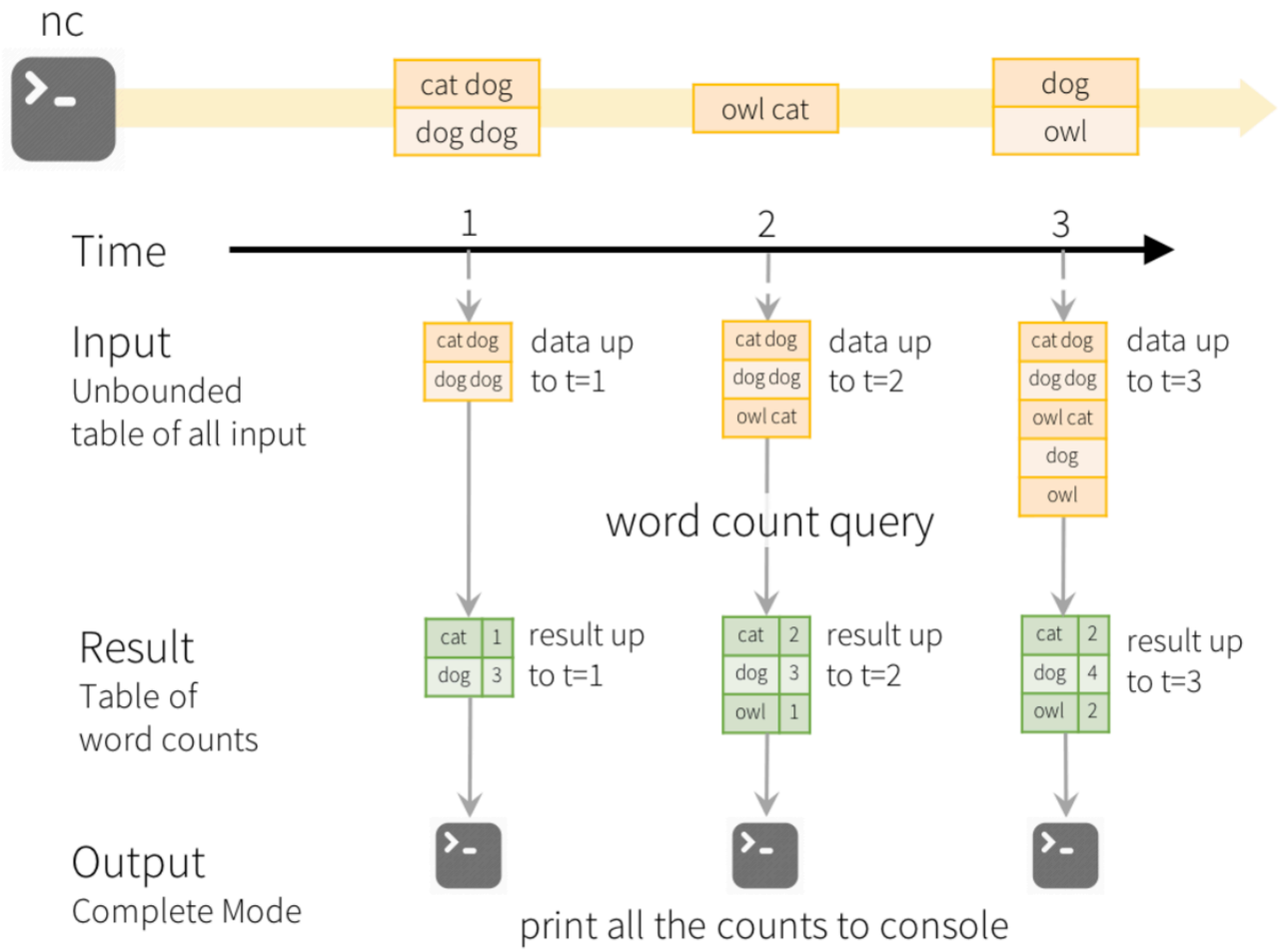
Data stream as an unbounded table

Programming Model



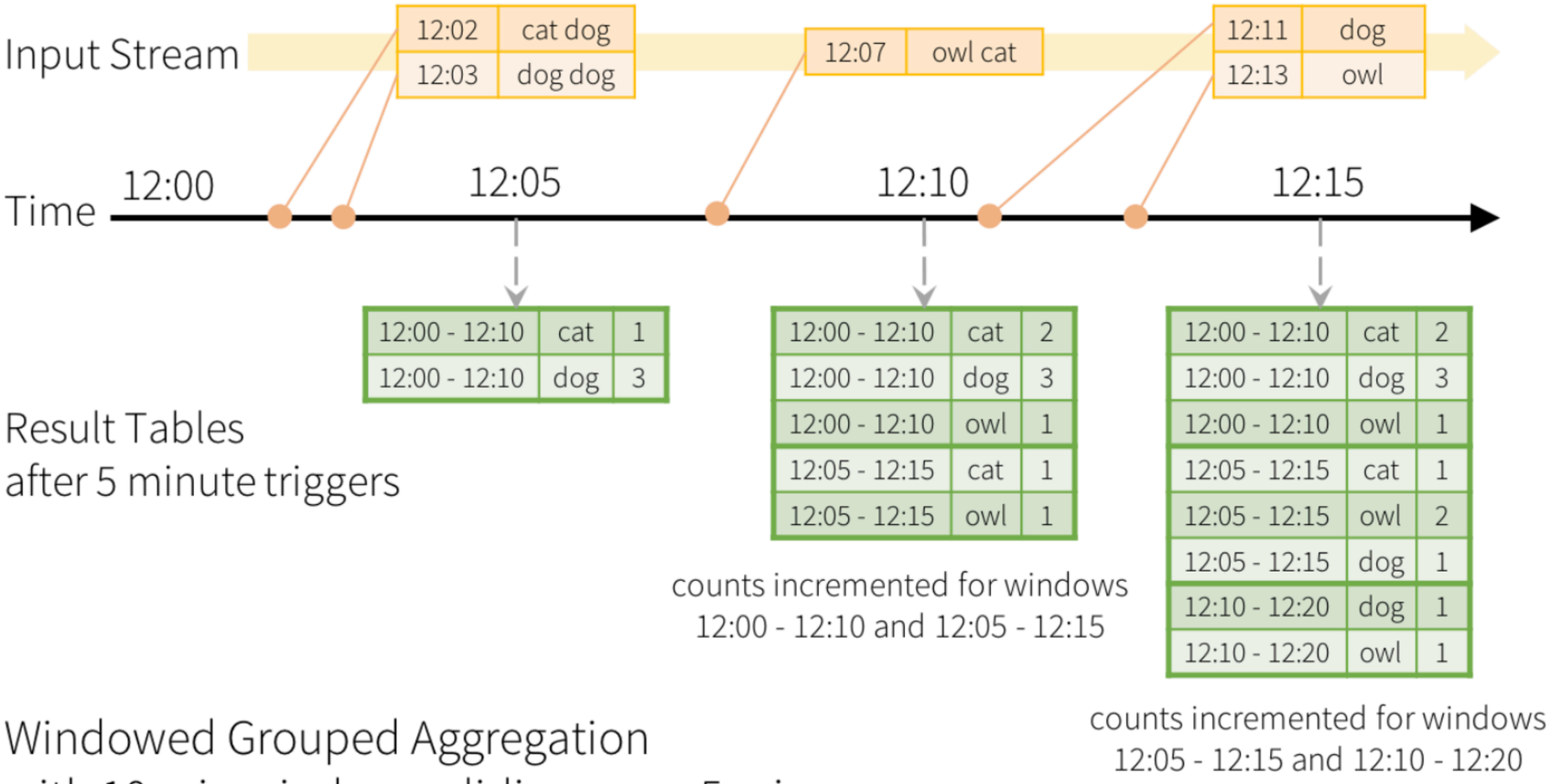
Programming Model for Structured Streaming

Incremental word counting




```
# Split the lines into words, retaining timestamps
    # split() splits each line into an array, and
    #  explode() turns the array into multiple rows
    words = lines.select(
        explode(split(lines.value, ' ')).alias('word'),
        lines.timestamp
    )


    # Group the data by window and word and compute
    # the count of each group
    windowedCounts = words.groupBy(
        window(words.timestamp, windowDuration, slideDuration),
        words.word
    ).count().orderBy('window')
```

Window and slide duration



Windowed Grouped Aggregation
with 10 min windows, sliding every 5 mins

Incremental word counting