Stream Processing

Lecture 2

2022/2023

Table of Contents

Spark Streaming

- Spark Streaming is an extension of the Spark batch processing system to enable scalable, high-throughput, fault-tolerant stream processing of live data streams.

Matei Zaharia, et. al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In Proc. SOSP'13.
http://people.csail.mit.edu/matei/papers/2013/sosp_spark_streaming.pdf

http://spark.apache.org/streaming/

Apache Spark

- Apache Spark provides in-memory, fault-tolerant distributed processing
- Spark programs can comprise multiple chained data transformation steps.
  - each step produces a RDD (Resilient Distributed Dataset)
    - RDD is the core abstraction

Apache Spark: Data Model

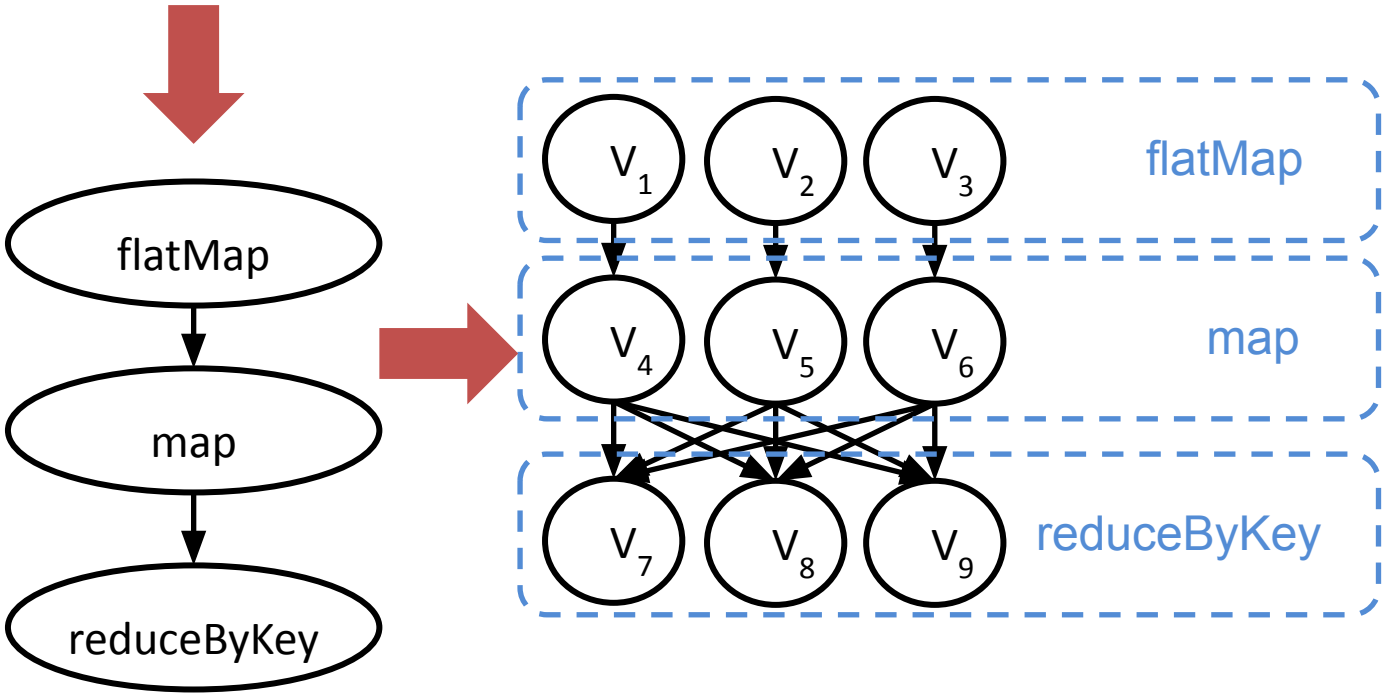
- RDDs are immutable data
  - logically a RDD is a set of data tuples;
  - physically distributed (partitioned) across many nodes;
  - upon a failure (or cascade of failures), RDDs can be recreated automatically and efficiently from the dependencies.

Apache Spark: Data Model

- Spark programs describe the flow of transformations that creates an RDD from another, usually in several steps.
- Spark programs, therefore, encode the dependencies among the various RDDs
  - this is known as the lineage graph

Apache Spark: an Example

```
wc = file.flatMap(lambda line: line.split(' '))
          .map(lambda vals: (vals[0], 1))
          .reduceByKey(lambda a,b: a+b)
```

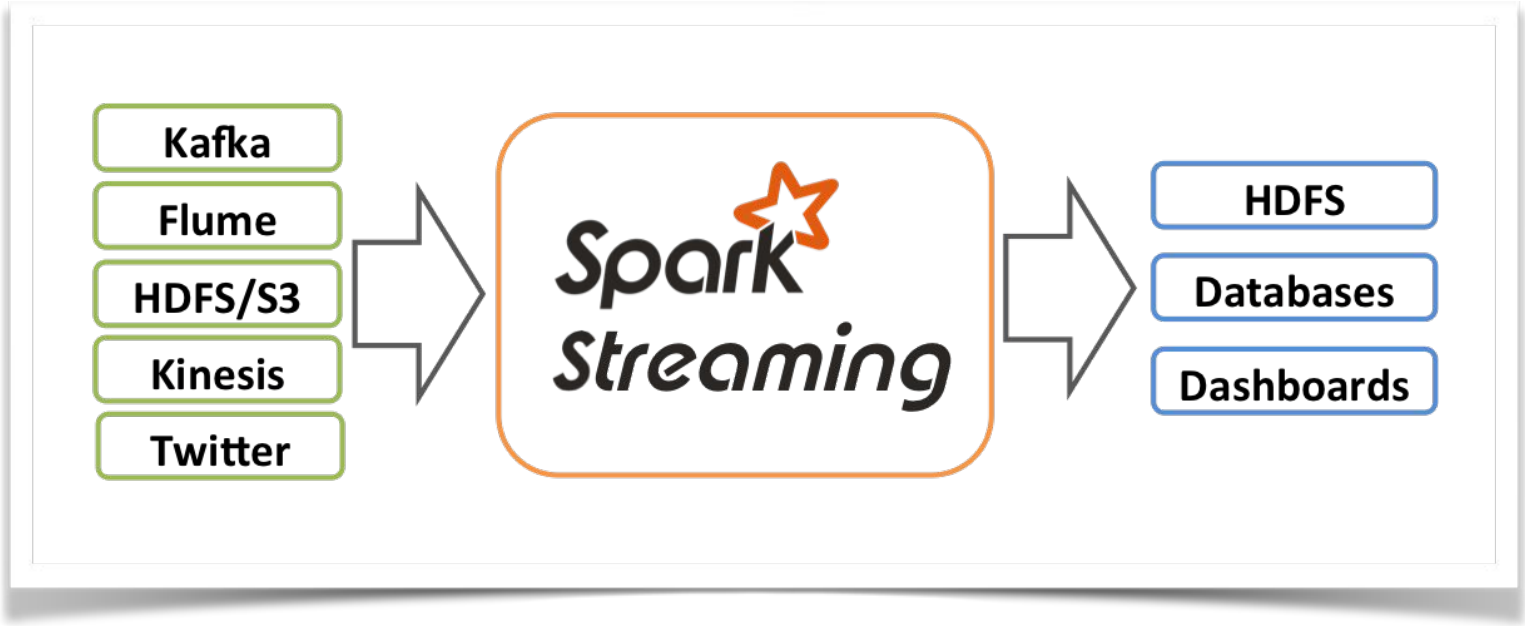


Spark Streaming: Core Idea

- Discretize a continuous stream of data into a sequence of small, discrete batches, each represented as a RDD
- Each of the resulting RDDs is processed in Spark to produce an output stream of RDDs

SparkStreaming : input data stream

- Spark already provides a number of standard data input streams.



Programming Model

- Incoming stream data is **split** and **batched** according **to fixed time intervals**, resulting in a new RDD for each interval period - a discrete stream of RDDs.

  (the size of the discrete RDDs varies as it depends on the ingress rate)
- RDDs are processed in a similar way as in Spark – by **applying transformations**.
- Output operators/actions provide several ways to **consume the processed stream.**

Running example

In the examples used, we will consider a stream with log entries from a web site. The entries are lines with the following values.

<date> <IP_source> <return_value> <operation> <URL> <time>

2016-12-06T08:58:35.318+0000 37.139.9.11 404 GET /codemove/TTCENCUFMH3C 0.026

First example

- List the top-3 IP sources with more accesses in the last 30 seconds. Update the list every 10 seconds.

Start Spark Streaming

```
import pyspark
from pyspark.streaming import StreamingContext

sc = pyspark.SparkContext('local[*]')
try :
    ssc = StreamingContext(sc, 10)

        ...

    ssc.start()
    ssc.awaitTermination()
except err:
    print( err)
    sc.stop()
    ssc.stop()
```

Start Spark Streaming

```
import pyspark
from pyspark.streaming import StreamingContext

sc = pyspark.SparkContext('local[*]')
try :
    ssc = StreamingContext(s

        ...

    ssc.start()
    ssc.awaitTermination()
except err:
    print( err)
    sc.stop()
    ssc.stop()
```

SparkContext(descr)

A SparkContext represents a connection to a Spark cluster. 'local' is used for running in local mode.

Start Spark Streaming

```
import pyspark
from pyspark.streaming import StreamingContext

sc = pyspark.SparkContext('local[*]')
try :
    ssc = StreamingContext(sc, 10)

        ...

    ssc.start()
    ssc.awaitTermination()
except err:
    print( err)
    sc.stop()
    ssc.stop()
```

StreamingContext(ctx, dur)
A StreamingContext represents a connection to a Spark cluster for executing streaming operations. **dur** is the duration of the batch (in seconds).
Why are we using 10 as the value of **duration**?

Start Spark Streaming

```
import pyspark
from pyspark.streaming import StreamingContext

sc = pyspark.SparkContext('local[*]')
try :
    ssc = StreamingContext(sc, 10)

        ...

    ssc.start()
    ssc.awaitTermination()
except err:
    print( err)
    sc.stop()
    ssc.stop()
```

scc.start()
Start processing the stream.

Start Spark Streaming

```
import pyspark
from pyspark.streaming import StreamingContext

sc = pyspark.SparkContext('local[*]')
try :
    ssc = StreamingContext(sc, 10)

        ...

    ssc.start()
    ssc.awaitTermination(120)
except err:
    print( err)
    sc.stop()
    ssc.stop()
```

scc.awaitTermination(timeout = NONE)
Wait the execution to stop, or end it after timeout in seconds.

First example

- **List the top-3 IP sources with more accesses in the last 30 seconds. Update the list every 10 seconds.**

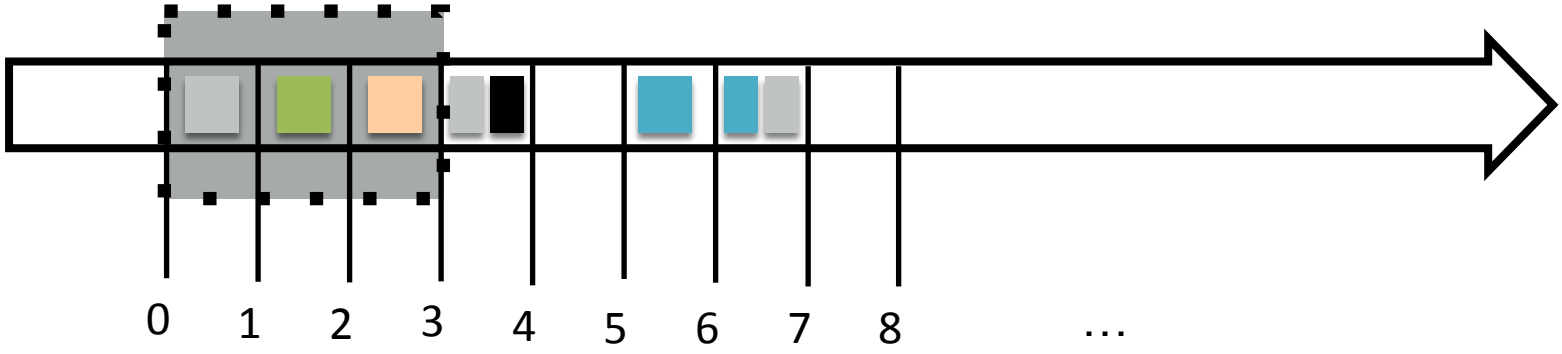
Connect to a stream

- **socketTextStream(host, port)**
- Connect to a stream source with IP address (host,port)

```
lines = ssc.socketTextStream("logsender", 7777)
```

Windowing

- In some cases, we want to process data over a larger interval — for example, process data of last 3 seconds, but produce results every second.
- Windowing groups the input stream RDDs that fall inside a larger time interval into one compound RDD. It advances automatically with passing time.
- SparkStream supports this by using a sliding window.
  - s.window("3s") would then output RDDs comprising the records in intervals: [0,3), [1,4), [2,5), …



Function: window

- **window**(*windowLength*, *slideInterval*)
- Define window of length *windowLength* at every *slideInterval* period

```
lines = ssc.socketTextStream("logsender", 7777)
result = lines.window(30,10)
```

In the example:
Every 10 seconds, the results of the last 30 seconds.

Function: filter

- **filter**(*func*)
- Return a new DStream by selecting only the records of the source DStream on which *func* returns true.

```
lines = ssc.socketTextStream("logsender", 7777)
result = lines.window(30,10) \
                .filter(lambda line : len(line) > 0 )
```

In the example:
Ignore empty lines.

Function: map

- **map**(*func*)
- Returns a new DStream by passing each element of the source DStream through a function *func*.

```
lines = ssc.socketTextStream("logsender", 7777)
result = lines.window(30,10) \
                    .filter(lambda line : len(line) > 0 ) \
                    .map(lambda line: line.split(' ')) \
                    .map(lambda vals: (vals[1],1))
```

In the example:
First, split a line into its components.
Second, return a pair (IP,1) as the basis for counting the number of occurrences.

- **map**(*func*)
- Returns a new DStream by passing each element of the source DStream through a function *func*.

```
lines = ssc.socketTextStream("logsender", 7777)
result = lines.window(30,10) \
                    .filter(lambda line : len(line) > 0 ) \
                    .map(lambda line: line.split(' ')) \
                    .map(lambda vals: (vals[1],1))
```

Function: reduceByKey

- **reduceByKey**(*func*, [*numTasks*])
- When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

```
lines = ssc.socketTextStream("logsender", 7777)
result = lines.window(30,10) \
                    .filter(lambda line : len(line) > 0 ) \
                    .map(lambda line: line.split(' ')) \
                    .map(lambda vals: (vals[1], 1)) \
                    .reduceByKey(lambda v1, v2: v1+v2)
```

In the example:
For each IP, count the number of occurrences.

Function: reduceByKey

- **reduceByKey**(*func*, [*numTasks*])
- When called on a DStream of (K, V) pairs, return a new DStream of (K, V) pairs where the values for each key are aggregated using the given reduce function.

```
lines = ssc.socketTextStream("logsender", 7777)
result = lines.window(30,10) \
          .filter(lambda line : len(line) > 0 ) \
          .map(lambda line: line.split(' ')) \
          .map(lambda vals: (vals[1],1)) \
          .reduceByKey(lambda v1, v2: v1+v2)
```

Function: transform

- **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.

```
lines = ssc.socketTextStream("logsender", 7777)
result = lines.window(30,10) \
        .filter(lambda line : len(line) > 0 ) \
        .map(lambda line: line.split(' ')) \
        .map(lambda vals: (vals[1],1)) \
        .reduceByKey(lambda v1, v2: v1+v2) \
                .transform(lambda rdd: rdd.sortBy(lambda x: x[1], \
                                                  ascending=False)) \
```

In the example:
Sort the list of occurrences…

Function: transform

- **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.

```
lines = ssc.socketTextStream("logsender", 7777)
result = lines.window(30,10) \
        .filter(lambda line : len(line) > 0 ) \
        .map(lambda line: line.split(' ')) \
        .map(lambda vals: (vals[1],1)) \
        .reduceByKey(lambda a, b: a+b) \
                .transform(lambda rdd: \
                    rdd.sortBy(lambda x: x[1], ascending=False) \
```

In the example:
Augment element with its index in the RDD. Filter out elements with index larger than 2. Remove index.

- **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.

```
lines = ssc.socketTextStream("logsender", 7777)
result = lines.window(30,10) \
        .filter(lambda line : len(line) > 0 ) \
        .map(lambda line: line.split(' ')) \
        .map(lambda vals: (vals[1],1)) \
                .reduceByKey(lambda a, b: a+b) \
                .transform(lambda rdd: \
                    rdd.sortBy(lambda x: x[1], ascending=False))\
                        .zipWithIndex() \
                        .filter( lambda v: v[1] < 3) \
                        .map( lambda v: v[0]))
```

Function: pprint

- **pprint()**
- Dump the results to the console.

```
result.pprint()
```

Function: pprint

- **pprint()**
- Dump the resu

```
result.pprint()
```

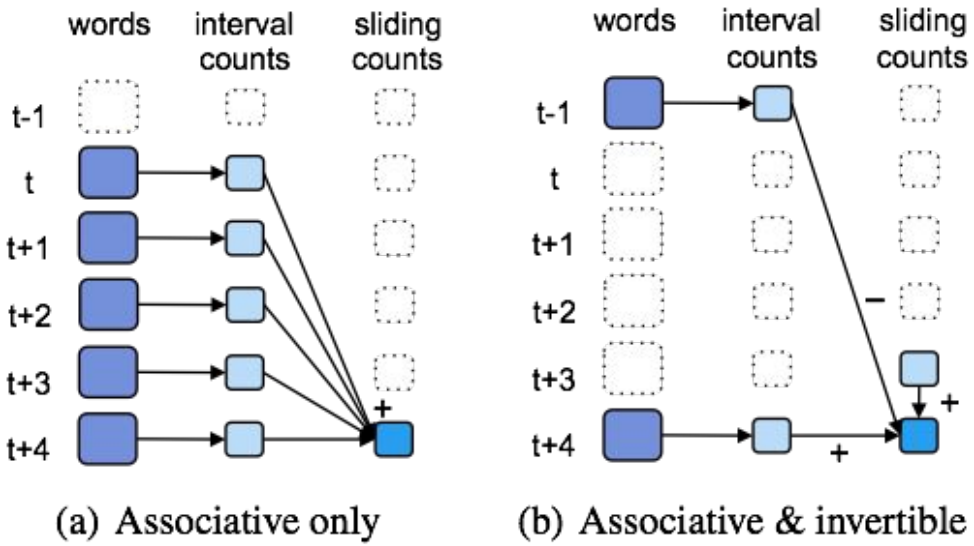
```
-------------------------------------------
Time: 2020-03-09 00:28:00
-------------------------------------------
('185.28.193.95', 91)
('120.52.73.97', 47)
('120.52.73.98', 35)

-------------------------------------------
Time: 2020-03-09 00:28:05
-------------------------------------------
('120.52.73.97', 240)
('120.52.73.98', 175)
('178.22.148.122', 135)

-------------------------------------------
Time: 2020-03-09 00:28:10
-------------------------------------------
('120.52.73.97', 532)
```

Incremental Aggregation (1)

- Incremental aggregation over a sliding window is possible using the ***reduceByWindow*** operator and an associative merge function:
  - **… dstream.reduceByWindow(lambda x, y: x + y, 30, 10)**
  - first aggregates data for each 10 second interval; then merge results into a single RDD result.
  - as the window slides, and one interval is dropped and another is added, it is not necessary to repeat the reduction of intervals already processed in the previous window, but still requires the aggregation of 3 intermediate results.

Incremental Aggregation (2)

- Optimized incremental aggregation is available for invertible merge functions:
  - **dstream.reduceByKeyAndWindow(lambda x, y: x + y, lambda x, y: x - y, 30, 10)**
  - as the window slides, the dropped interval is removed from the total and only the new interval that enters the window is added.



(a) Associative only (b) Associative & invertible

Function: join

- **join**(*otherStream*, [*numTasks*])
- 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 for each RDD generated in both streams.

```
otherCounts = ...

counts = lines.filter(lambda line : len(line) > 0 ) \
          .map(lambda line: line.split(' ')) \
          .map(lambda vals: (vals[1],1)) \
                    .join( otherCounts)
```

Second example

- List the top-3 IP sources with more accesses in the last 30 seconds. Update the list every 10 seconds.
- **Print the country of the URL, assuming there is a CSV file with the country for each IP.**

Join with RDD

- For joining with an RDD, it is necessary to use transform.

```
countries = sc.textFile( 'countries.csv') \
    .filter( lambda l: len(l) > 0) \
    .map( lambda l : l.split(',')) \
    .map( lambda l : (l[0],l[1]))

...
result =  result.transform( \
              lambda rdd: rdd.join(countries))
```

Third example

- List the top-3 IP sources with more accesses in the last 30 seconds. Update the list every 10 seconds.
- Print the country of the URL, assuming there is a CSV with the country for each IP.
- **Print also the total number of requests for each IP overtime.**

State tracking

- Computations can also carry arbitrary state across the whole discretized stream [and not just over a fixed size window]
- ***updateStateByKey****(func)*
- returns a new "state" Stream RDD, 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, for the whole stream.
  - Not all keys need to maintain state at all times.

Accumulate result

```
def updateFunction(newValues, runningCount):
    if runningCount is None:
        runningCount = 0
    return sum(newValues, runningCount)


...
runningCounts = result.updateStateByKey(updateFunction)
```

Join result with accumulated value

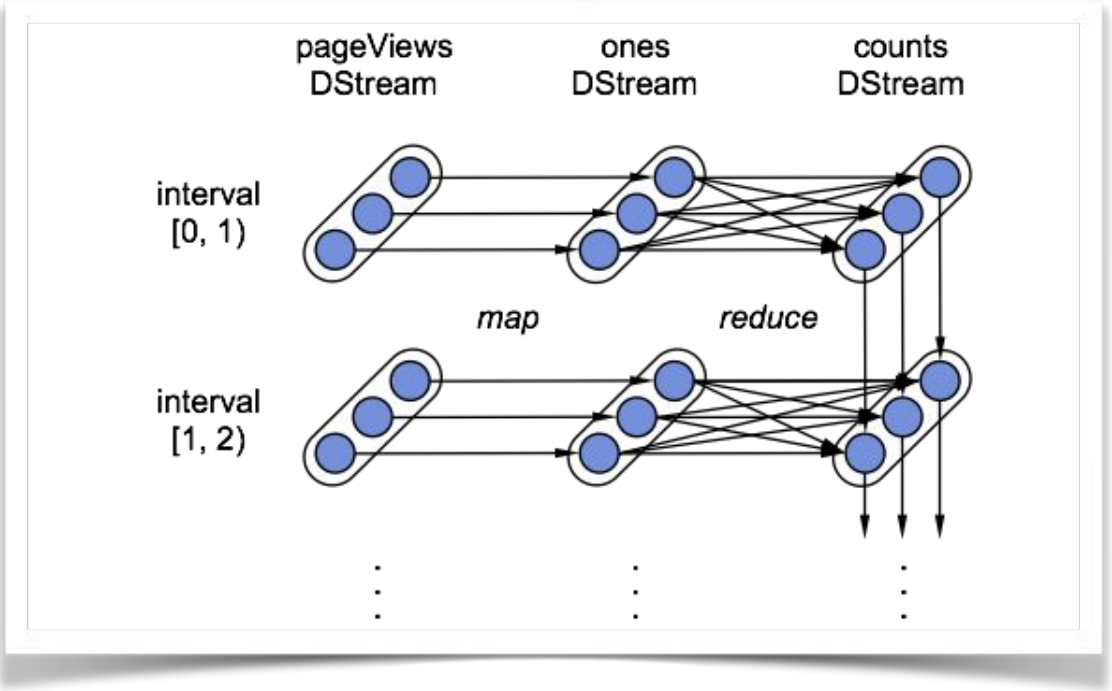
```
def updateFunction(newValues, runningCount):
    if runningCount is None:
        runningCount = 0
    return sum(newValues, runningCount)

result = lines.window(10,5) \
            .filter( lambda line : len(line) > 0 ) \
            .map(lambda line: (line.split(' ')[1],1)) \
            .reduceByKey(lambda a, b: a+b)

runningCounts = result.updateStateByKey(updateFunction)

result = result.transform( ... ) // top N

result = result.join(runningCounts)

result.pprint()
```

SparkStreaming : fault-tolerance

- SparkStreaming can recover from faults, by rebuilding missing RDDs across several nodes, in parallel.
  - Uses lineage information to allow minimal re-computation of the RDDs lost in a fault.
  - Independent sections of the lineage graph can be processed and recovered in parallel

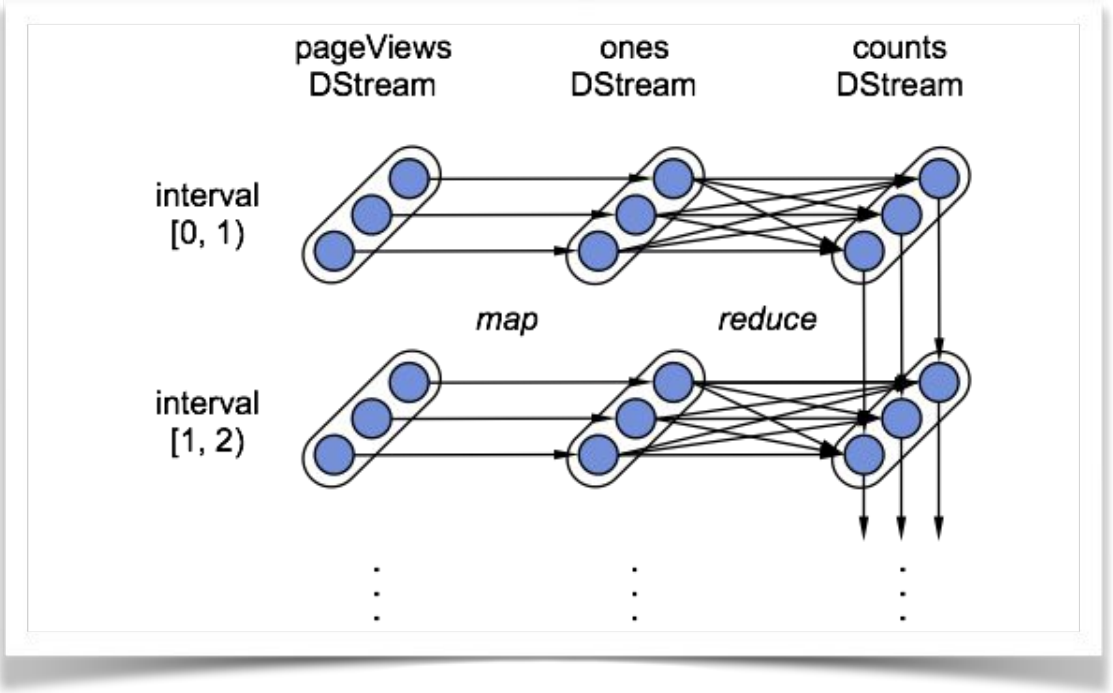
SparkStreaming : lineage graph

– The lineage graph in SparkStreaming is logically static but dynamic in physical terms.



Fault-tolerance

– Can recover from faults, by rebuilding missing RDDs
  - Maintains lineage information to minimize re-computation
  - Maintains checkpoints



SparkStreaming : checkpointing

- The lineage graph in SparkStreaming is logically static but dynamic in physical terms
  - ***updateStateByKey,*** creates a lineage graph that goes back to the first RDD of the input stream
  - Fault recovery would require persisting the entire stream.
  - Instead, SparkStreaming performs **periodic checkpointing**, by saving internal state to the filesystem
  - Upon a failure, RDDs are re-created from the last valid checkpoint.

SparkStreaming : limitations

- SparkStreaming operates over discrete portions of the stream that are created **based on time**, with a **fixed** minimum latency
  - Fixed duration time intervals do not generally translate into constant sized RDDs… it depends on the ingress rate
- SparkStreaming does not **naturally** support processing streams at a granularity that is not time oriented…
  - For instance, it is difficult to process each stream item independently and update the resulting stream as each item arrives

Bibliography

- Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13). ACM, New York, NY, USA, 423-438. DOI: https://doi.org/10.1145/2517349.2522737

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