Mining Massive Datasets

Lecture 13

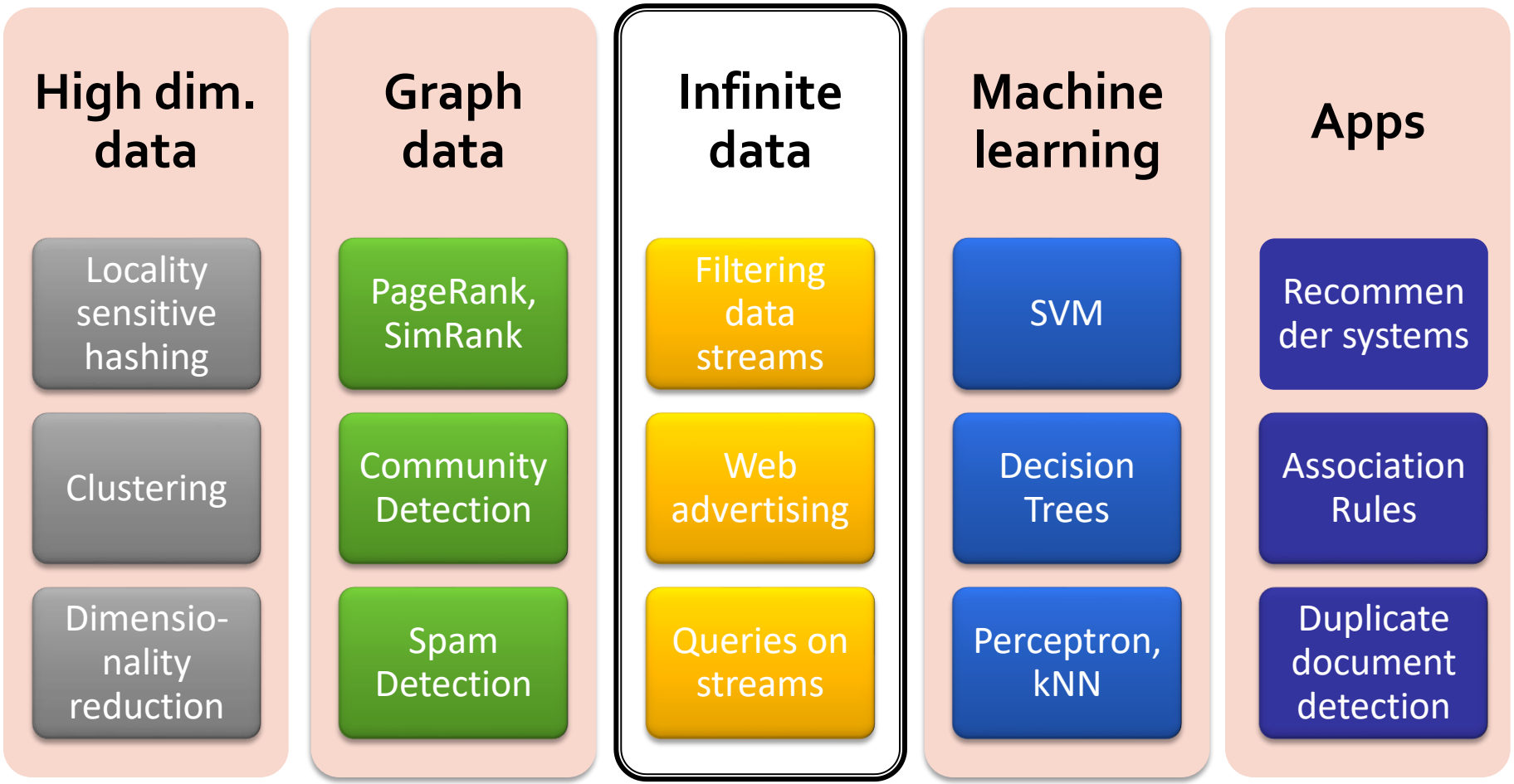
Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

Note on Slides

A substantial part of these slides come (either verbatim or in a modified form) from the book Mining of Massive Datasets
by Jure Leskovec, Anand Rajaraman, Jeff Ullman (Stanford University).
For more information, see the website accompanying the book: http://www.mmds.org.

Infinite Data



Programming in Spark & MapReduce

Counting Distinct Elements in a Stream

Counting Distinct Elements

- **Problem:**
  - Data stream consists of a universe of elements chosen from a set of size ***N***
  - Maintain a count of the number of distinct elements seen so far
- **Obvious approach:**

  Maintain the set of elements seen so far
  - That is, keep a hash table of all the distinct elements seen so far

Applications

- **How many different words are found among the Web pages being crawled at a site?**
  - Unusually low or high numbers could indicate artificial pages (spam?)
- **How many different Web pages does each customer request in a week?**
- **How many distinct products have we sold in the last week?**

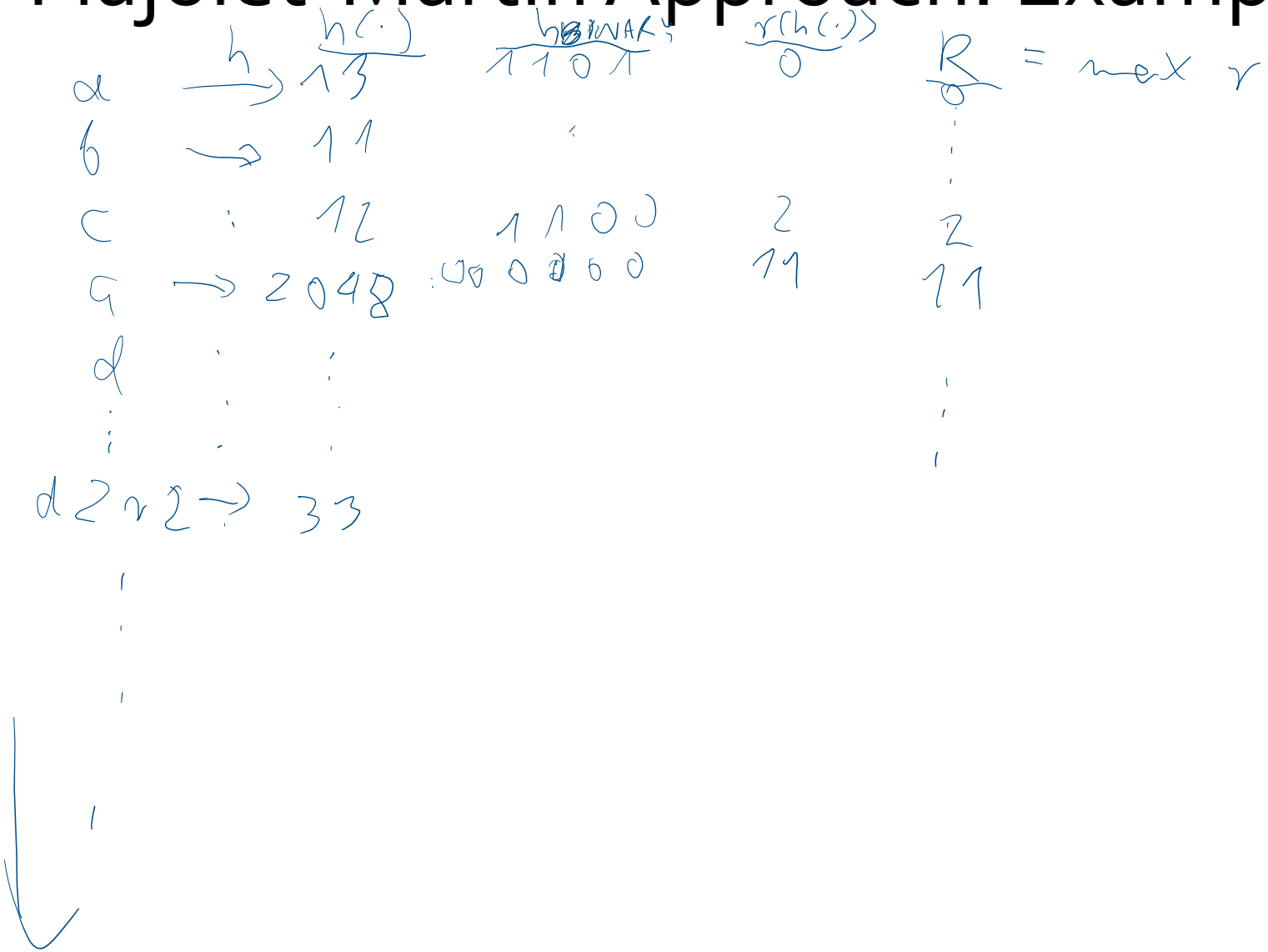
Using Small Storage

- Real problem: What if we do not have space to maintain the set of elements seen so far?
- Estimate the count in an unbiased way
- **Accept that the count may have a little error, but limit the probability that the error is large**

Flajolet-Martin Approach

- Pick a hash function ***h*** that maps each of the ***N*** elements to at least $\log_2 N$ bits
- For each stream element ***a***, let ***r*(*a*)** be the number of trailing **0s** in ***h*(*a*)**
  - **r(a)** = position of first 1 counting from the right
    - E.g., say ***h(a) = 12***, then ***12*** is ***1100*** in binary, so ***r(a) = 2***
- Record ***R* = the maximum *r*(*a*) seen**
  - $\mathbf{R = max_a\ r(a)}$, over all the items ***a*** seen so far
- **Estimated number of distinct elements = 2^R**

Flajolet-Martin Approach: Example



Why It Works: Intuition

- Very very rough and heuristic intuition why Flajolet-Martin works:
  - ***h(a)*** hashes ***a*** with **equal prob.** to any of ***N*** values
  - Then ***h(a)*** is a sequence of $\mathbf{log_2 N}$ bits, where $\boldsymbol{2^{-r}}$ fraction of all ***a***s have a tail of ***r*** zeros
    - About 50% of ***a***s hash to ***0**
    - About 25% of ***a***s hash to ****00**
    - So, if we saw the longest tail of ***r=2*** (i.e., item hash ending ***100**) then we have probably seen **about *4*** distinct items so far
  - So, it takes to hash about 2^r items before we see one with zero-suffix of length *r*

Why It Works: More formally

- **Now we show why Flajolet-Martin works**
- **Formally, we will show that probability of finding a tail of *r* zeros:**
  - **Goes to 1 if $m \gg 2^r$**
  - **Goes to 0 if $m \ll 2^r$**

  where m is the number of distinct elements seen so far in the stream
- **Thus, 2^R will almost always be around *m!***
- Note: Compare to **proof-of-work** in Bitcoin ☺

Why It Works: More formally

- **What is the probability that a given *h*(*a*) ends in at least *r* zeros is 2^{-r}**
  - **h(a)** hashes elements uniformly at random
  - Probability that a random number ends in at least ***r*** zeros is $\mathbf{2^{-r}}$
- Then, the probability of **NOT** seeing a tail of length ***r*** among ***m*** elements:

$$\left(1 - 2^{-r}\right)^{m}$$

Prob. all end in fewer than ***r*** zeros.

Prob. that given **h(a)** ends in fewer than ***r*** zeros

Why It Works: More formally

- **Note:** $(1-2^{-r})^{m} = (1-2^{-r})^{2^{r}(m2^{-r})} \approx e^{-m2^{-r}}$
- **Prob. of NOT finding a tail of length *r* is:**
  - If ***m* << *2*^{r}**, then prob. tends to **1**
    - $(1-2^{-r})^{m} \approx e^{-m2^{-r}} = 1$ as **m/2^{r} → 0**
    - So, the probability of finding a tail of length ***r*** tends to **0**
  - If ***m* >> *2*^{r}**, then prob. tends to **0**
    - $(1-2^{-r})^{m} \approx e^{-m2^{-r}} = 0$ as **m/2^{r} → ∞**
    - So, the probability of finding a tail of length ***r*** tends to **1**
- **Thus, 2^{R} will almost always be around *m!***

Why It Doesn't Work

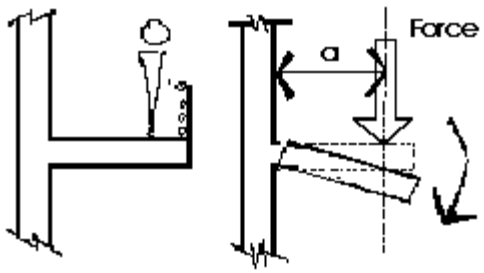
- **E[2^R] is actually infinite; Why?**
- Recall that $E[X] = \sum_{X=x} P[X = x] \cdot x$
- Here $x = 2^1, \ldots, 2^R, \ldots$
- Probabilities $P[X = x]$ are:
  - Let $\boldsymbol{r}$ be such that $\boldsymbol{2^r \gg m}$ ($\boldsymbol{m}$ is #elements seen so far)
  - Let $\boldsymbol{p > 0}$ be probability that r was the largest number of 0's at the end of the hash value for any of the $\boldsymbol{m}$ elements
  - Then the probability of finding $\boldsymbol{r + 1}$ to be the largest number of 0's instead is at least $\boldsymbol{p/2}$

- So: Probability $P[X = 2^{R+1}]$ halves when ***R → R+1***, but value 2^{R+1} doubles
- => The summands in $E[X]$ remain ~constant
- => We get an infinite sum as limes

Why It Doesn't Work

- $E[2^R]$ is actually infinite
- Workaround involves using many hash functions h_i and getting many samples of R_i
- How are samples R_i combined?
  - **Average?** What if we get one very large value 2^{R_i}?
  - **Median?** All estimates are a power of **2**
  - **Solution:**
    - Partition your samples into small groups
    - Take the median of groups
    - Then take the average of the medians

Computing Moments



$$\sum_{i \in A} (m_i)^k$$

Generalization: Moments

- Suppose a stream has elements chosen from a set A of N values
- Let $\boldsymbol{m_i}$ be the number of times value $\boldsymbol{i}$ occurs in the stream
- Example stream: a, b, c, c, a, a, b, …
- => $m_a = 3, m_b = 2, m_c = 2$
- The ***k*th *moment*** is defined as:

$$\sum_{i \in A} (m_i)^k$$

Special Cases

$$\sum_{i \in A} (m_i)^k$$

- **0th moment =** number of distinct elements
  - The problem just considered
- **1st moment =** count of the numbers of elements = length of the stream
  - Easy to compute
- **2nd moment =** ***surprise number S*** **=** a measure of how uneven the distribution is

Example: Surprise Number

- Stream of length 100 with 11 distinct values
- Item counts (m_i's): **10, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9**
- **Surprise S = 910**
- Item counts (m_i's): **90, 1, 1, 1, 1, 1, 1, 1 ,1, 1, 1**
- **Surprise S = 8110**

- Q: Why is $\sum_{i \in A} (m_i)^2$ a good estimate of an uneven distribution?
- A: Because $\sum_{i \in A} (m_i)^2$ is minimized if all m_i's are (roughly) equal

Alon-Matias-Szegedy (AMS) Method

- AMS method works for all moments
  - Useful if don't have enough RAM to keep all m_i's
- We just consider the 2nd moment $S = \sum_i m_i^2$
- We pick and keep track of "approx" **variables *X***
- For each variable ***X*** we store ***X.el*** and ***X.val***
  - ***X.el*** corresponds to the (value or ID of) item ***i***
  - ***X.val*** corresponds to the **count** of item ***i***
- Each ***X.el*** and ***X.val*** needs RAM => #***X*** is bounded
  - But: The larger #***X***, the higher the accuracy of AMS

First: Only One Random Variable **X**

- How to set ***X.val*** and ***X.el***?
  - Assume stream has length ***n*** (we relax this later)
  - Pick some random time ***t*** (***t < n***) to start, so that any time is equally likely
  - Let at time ***t*** the stream have item ***i***: We set ***X.el = i***
  - Then we maintain count ***c*** (***X.val = c***) of the number of ***is*** in the stream starting from the chosen time ***t***
- Then the estimate of the 2nd moment ($\sum_i \boldsymbol{m_i^2}$) is:

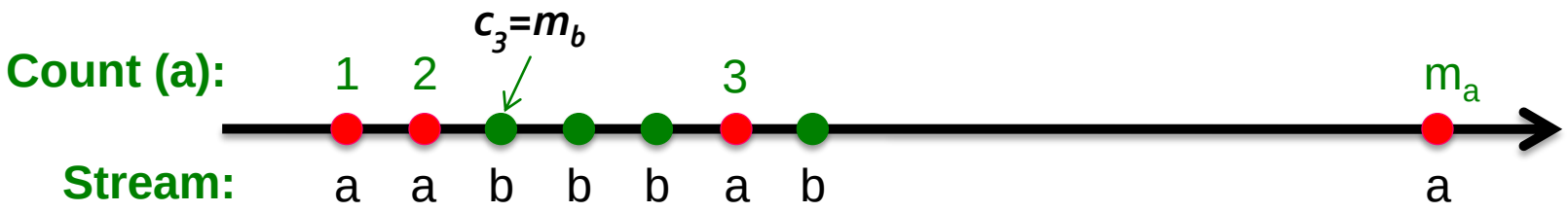
$$\boldsymbol{S = f(X) = n\,(2 \cdot c - 1)}$$

  - Note: We use multiple **X**s, (X_1, X_2,… X_k) and our final estimate will be the average $S = 1/k \sum_j^k f(X_j)$

Example: "Approx" Variables

- Stream: a, b, c, b, d, a, c, d, a, b, d, c, a, a, b
  - $n = 15, m_a = 5, m_b = 4, m_c = 3, m_d = 3$
  - 2nd moment $\sum_{i \in A} (m_i)^2 = 5^2 + 4^2 + 2 \cdot 3^2 = 59$
- Assume we use 3 vars X_1, X_2, X_3
  - To define them, we pick random positions 3, 8, 13:
  - a, b, c, b, d, a, c, d, a, b, d, c, a, a, b
- => $X_1.\text{el} = \text{c}, X_1.\text{val} = 3; X_2.\text{val} = 2, X_3.\text{val} = 2$
- The estimate $\boldsymbol{S}$ of 2nd moment is then:
  - $f(X_1) = n(2 \cdot c - 1)$=15·(2·3−3)=**75**; $f(X_2) = f(X_3)$ = **45**
  - Final estimate = average of X_i's: $\boldsymbol{S} = 1/k \sum_j^k f(X_j)$ = **55**

Expectation Analysis (One **X**)



- **2nd** moment is $\boldsymbol{S = \sum_i m_i^2}$
- $\boldsymbol{c_t}$ … number of times item at time $\boldsymbol{t}$ appears from time **t** onwards ($\boldsymbol{c_1=m_a}$, $\boldsymbol{c_2=m_a-1}$, $\boldsymbol{c_3=m_b}$)
- $\boldsymbol{E[f(X)] = \frac{1}{n}\sum_{t=1}^{n} n(2c_t - 1)}$

$$\boldsymbol{= \frac{1}{n}\sum_i n\,(1 + 3 + 5 + \cdots + 2m_i - 1)}$$

$\boldsymbol{m_i}$ … total count of item $\boldsymbol{i}$ in the stream (we are assuming stream has length **n**)

(A) Group times by the stream el. value $\boldsymbol{i}$ (item)

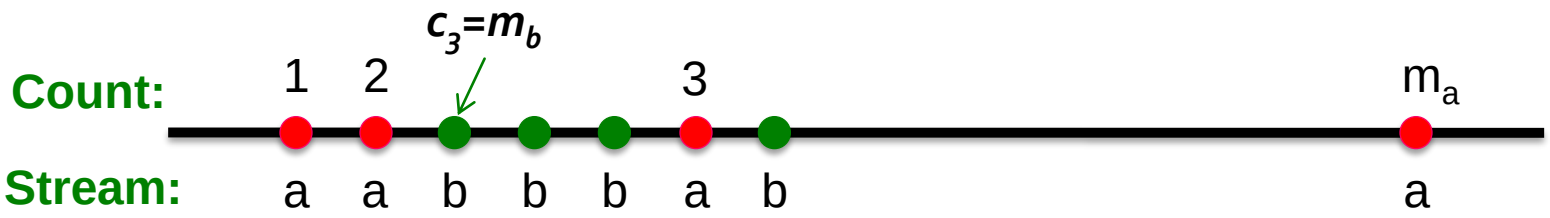
Time t when the last $\boldsymbol{i}$ is seen ($\boldsymbol{c_t=1}$)

Time $\boldsymbol{t}$ when the penultimate $\boldsymbol{i}$ is seen ($\boldsymbol{c_t=2}$)

Time $\boldsymbol{t}$ when the first $\boldsymbol{i}$ is seen ($\boldsymbol{c_t=m_i}$)

Due to (A), the content of the parentheses corresponds to a specific item $\boldsymbol{i}$

Expectation Analysis



- $E[f(X)] = \frac{1}{n}\sum_i n\,(1+3+5+\cdots+2m_i-1)$
  - Little side calculation: $(1+3+5+\cdots+2m_i-1) = \sum_{i=1}^{m_i}(2i-1) = 2\frac{m_i(m_i+1)}{2} - m_i = (m_i\ )^2$
- Then $\boldsymbol{E[f(X)] = \frac{1}{n}\sum_i n\,(m_i)^2}$
- **So, $\mathbf{E[f(X)] = \sum_i (m_i)^2 = S}$**
- We have the second moment (in expectation)!

Higher-Order Moments

- For estimating kth moment we essentially use the same algorithm but change the estimate:
  - For **k=2** we used ***n* (2·c – 1)**
  - For **k=3** we use: ***n* (3·c^2 – 3c + 1)** (where **c=X.val**)
- **Why?**
  - **For k=2:** Remember we had $(1 + 3 + 5 + \cdots + 2m_i - 1)$ and we showed terms ***2c-1*** (for **c=1,…,m**) sum to ***m^2***
    - $\sum_{c=1}^{m} 2c - 1 = \sum_{c=1}^{m} c^2 - \sum_{c=1}^{m} (c-1)^2 = m^2$
    - **So:** $\boldsymbol{2c - 1 = c^2 - (c-1)^2}$
- **For k=3: c^3 - (c-1)^3 = 3c^2 - 3c + 1**
- **Generally:** Estimate of kth mom.: $\boldsymbol{n(c^k - (c-1)^k)}$

Combining Samples

- In practice:
  - Compute $f(X) = n(2^c - 1)$ for as many variables X as you can fit in memory
  - Average them in groups
  - Take median of averages
- Problem: Streams never end
  - We assumed there was a number n, the number of positions in the stream
  - But real streams go on forever, so n is a variable – the number of inputs seen so far

Streams Never End: Fixups

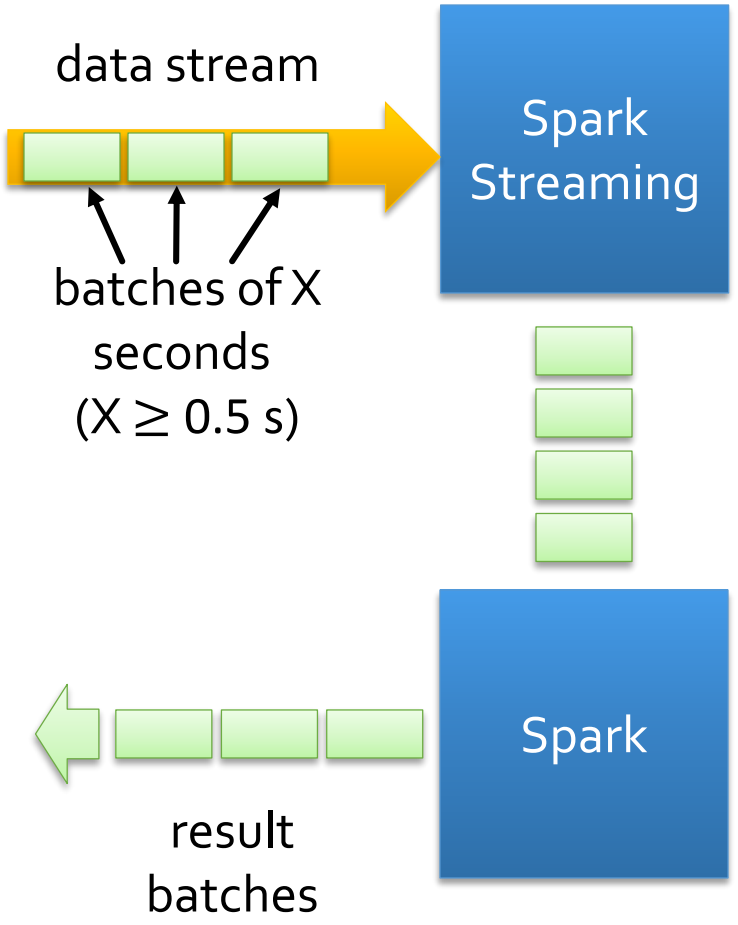
- (1) The variables ***X*** have ***n*** as a factor – keep ***n*** separately; just hold the count in ***X***
- (2) Suppose we can only store ***k*** counts; We must throw some ***X***s out as time goes on:
  - **Objective:** Each starting time ***t*** is selected with probability ***k/n***
  - **Solution: (fixed-size sampling!)**
    - Choose the first ***k*** times for ***k*** variables
    - When the n^{th} element arrives (***n*** > ***k***), choose it with probability ***k/n***
    - If you choose it, throw one of the previously stored variables **X** out, with equal probability

Spark Streaming

Basic Programming: Repetition

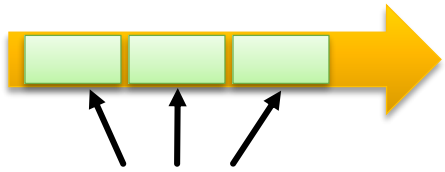
Spark Streaming: Concept

- Process stream as a **series of small batch jobs**
  - Chop up the live stream into batches of X seconds
  - Spark treats each batch of data as an RDD and processes them using (normal) RDD operations
  - The results of the RDD operations are returned in batches



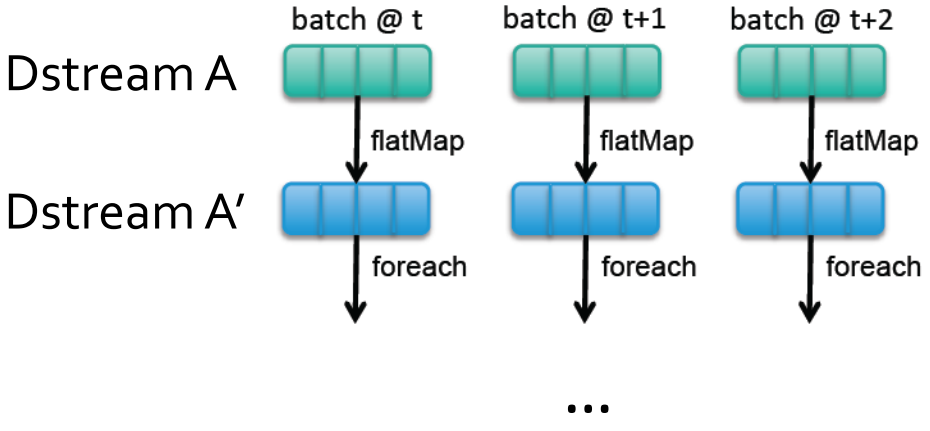
Programming Model - DStream

- **DStream** = Discretized Stream
  - "Container" for a stream
  - Implemented as a **sequence of RDDs**



Each chunk = Resilient Distributed Dataset (RDD)

- DStreams can be …
  - Created from "raw" input streams
  - Obtained by transforming existing DStreams



From: Tathagata Das, Spark Streaming, Spark Summit 2014

Example: (Stream) Word Count

- Goal: We want to count the occurrences of each word in each batch a text stream
  - Data received from a TCP socket 9999, each "event" (= record) is a line of text
  - Stream is split into RDDs, each 1 second "length"
    - Each RDD can have 0 or more records!
  - Output: first ten elements of each RDD
- Program structure
  - 1. Set up the processing "pipeline"
  - 2. Start the computation and specify termination

Word Count: Pipeline Setup /1

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

sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)

lines = ssc.socketTextStream("localhost", 9999)
```

Use two threads: 1 for source feed, 1 for processing

Set batch interval to 1 second

Create a DStream that will connect to hostname:port, like localhost:9999

Word Count: Pipeline Setup /2

- Since each "event" in a DStream is a "normal" RDD-record, we can process it with Spark operations
  - Here: each record is a line of text

New DStream (and new RDD for each batch)

Split each line into words

```
words = lines.flatMap( lambda line: line.split(" ") )
pairs = words.map( lambda word: (word, 1) )
wordCounts = pairs.reduceByKey( lambda x, y: x + y )
```

Count each word in each batch

```
wordCounts.pprint()
```

Print the first ten elements of each RDD generated in this DStream to the console

Word Count: Start & End

Start the computation

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

Wait to terminate

Terminal 1

- Netcat (link) utility can redirect std input to a TCP port (here: 9999)
- **nc -lk 9999**

- <type anything…>
- Hello IMMD

Terminal 2

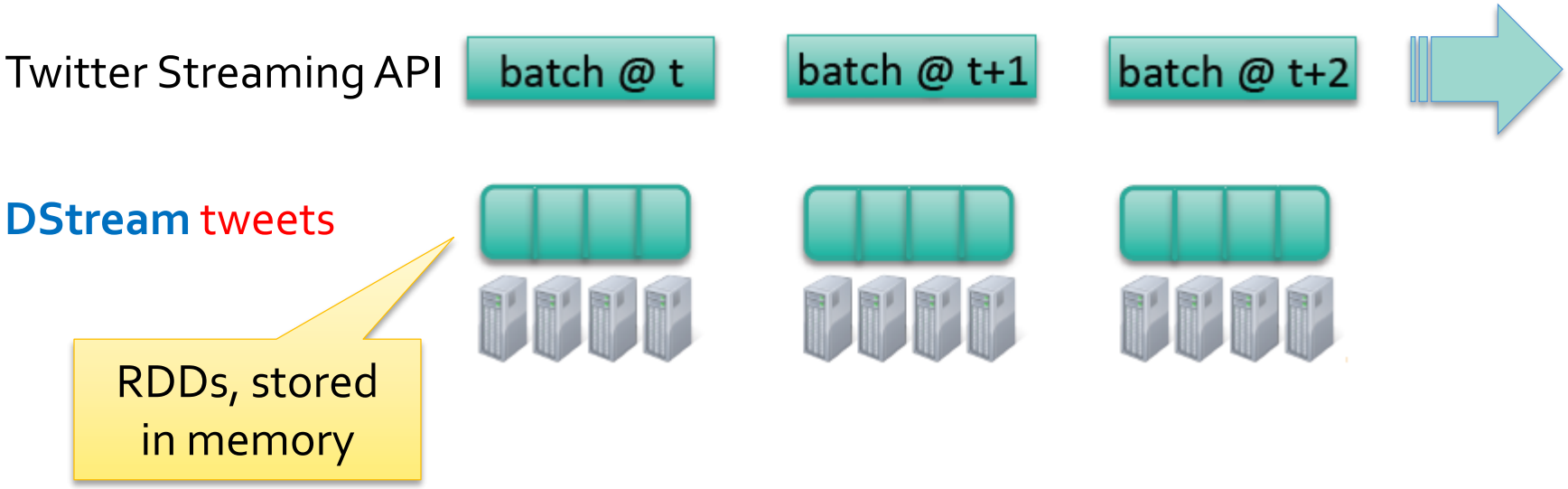
- ./bin/spark-submit network_wordcount.py localhost 9999

- ---
- Time: 2015-01-08 13:22:51
- ---
- (hello,1)
- (IMMD,1)
- …

Example – Get hashtags from Twitter

Example in **Scala** because Twitter source was not yet supported in Python (as of Spark 1.2)

```
val ssc = new StreamingContext (sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream (ssc, auth)
```



From: Tathagata Das, Spark Streaming, Spark Summit 2014

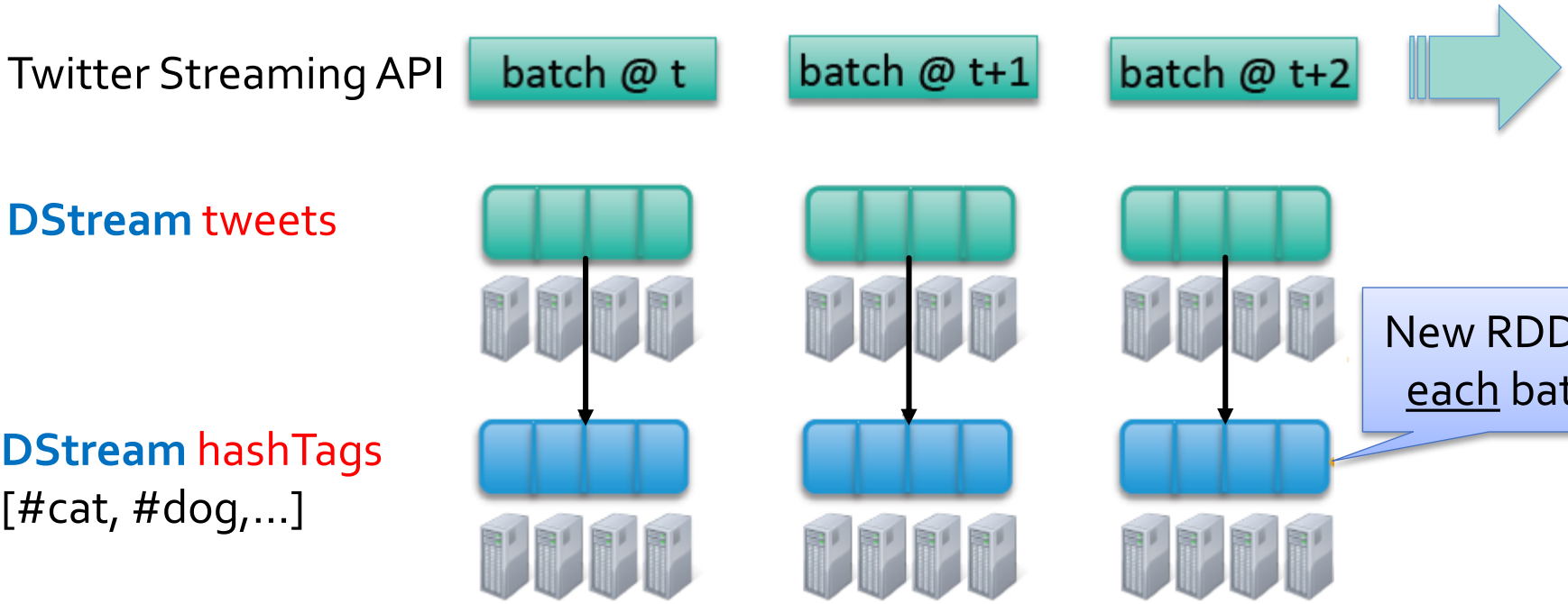
Get hashtags from Twitter /2

```
val ssc = new StreamingContext (sparkContext, Seconds(1))
val tweets = TwitterUtils.createStream (ssc, auth)
val hashTags = tweets.flatMap(status => getTags(status))
```

Transformed DStream

Spark flatMap

In Python:
lambda status: getTags(status)

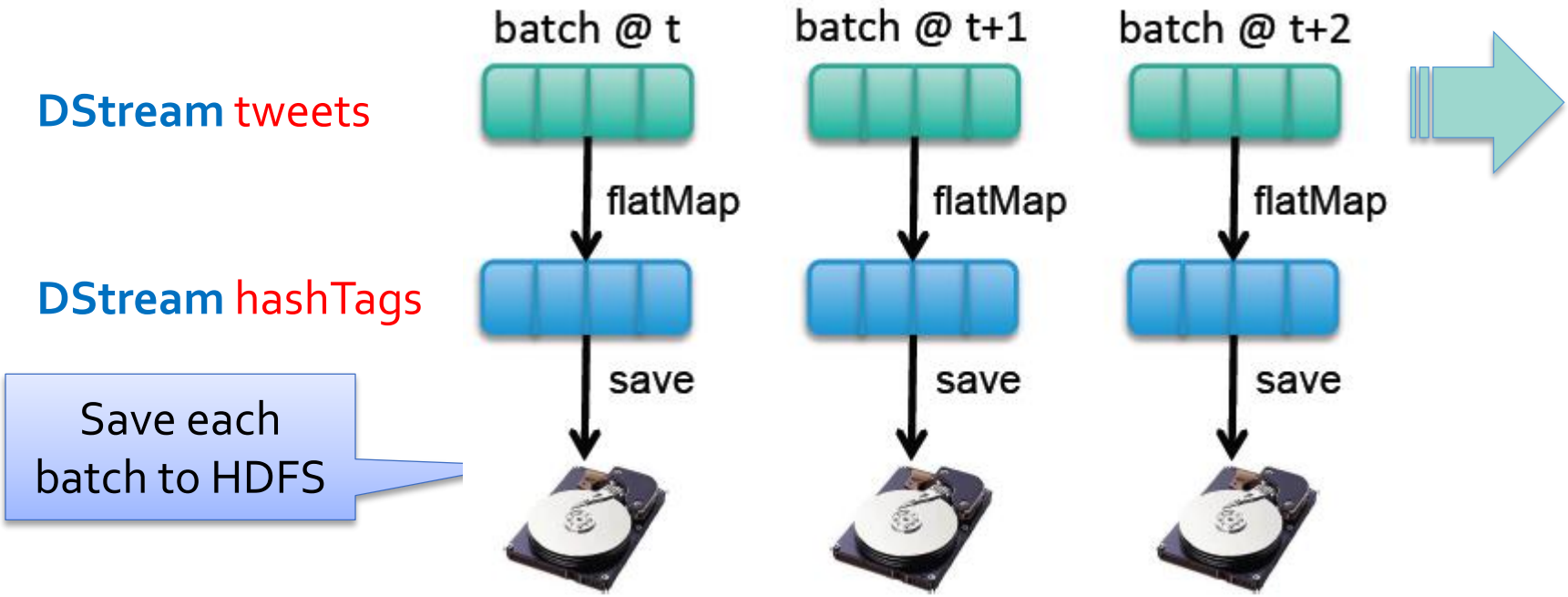


From: Tathagata Das, Spark Streaming, Spark Summit 2014

Get hashtags from Twitter /3

```
...
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

Output: write to external storage



From: Tathagata Das, Spark Streaming, Spark Summit 2014

Spark Streaming

Advanced Programming

Transformations on DStreams

- Many “normal” Spark transformations are available, and some additional ones

map	flatMap	filter
repartition	union	count
reduce	count	countByValue
reduceByKey	join	cogroup
transform	**updateStateByKey**	

Operation "transform"

- The **transform** operation applies an arbitrary RDD-to-RDD function (i.e. a transformation) on each RDD in a DStream
  - Allows using any RDD operation not in the DStream API
- Example: join each RDD in a DStream with additional (precomputed) information

```
# "Normal" RDD containing spam information
spamInfoRDD = sc.pickleFile(<loadPath>)
# join data stream with spamInfoRDD
cleanDStream = wordCounts.transform( lambda
                    rdd: rdd.join(spamInfoRDD).filter(...) )
```

Operation "updateStateByKey"

- Allows to maintain arbitrary state and update it with new data from a stream
  - Input DStream contains (key, value)-pairs
- Two components:
  - **State**: any datatype (e.g. primitive, object, list,…)
  - **Update function f** of the form:
    newState = **f** (newValues, oldState)
- **f** will be called for each key k separately; newValues is a sequence of new values (for k)
- Usage:
  - statesDStream = inputDStream.updateStateByKey(**f**)

The updateStateByKey method returns a new DStream which contains RDDs that has the state data of each key

Example for updateStateByKey

- We want to count number of occurrences of each word since the start of the stream
- The input DStream contains pairs (<word>, 1)
- We need a separate state for each encountered word w, namely a count of w (= integer)
- The update function in Python:

Different for each <word>!

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

Python's sum(iterable[, start])
(https://docs.python.org/2/library/functions.html#sum)

This is a sequence of 1's for a current word w since last call of the updateFunction (for w)

Excerpt from stateful_network_wordcount.py (link)

Call: stateful_network_wordcount.py localhost 9999

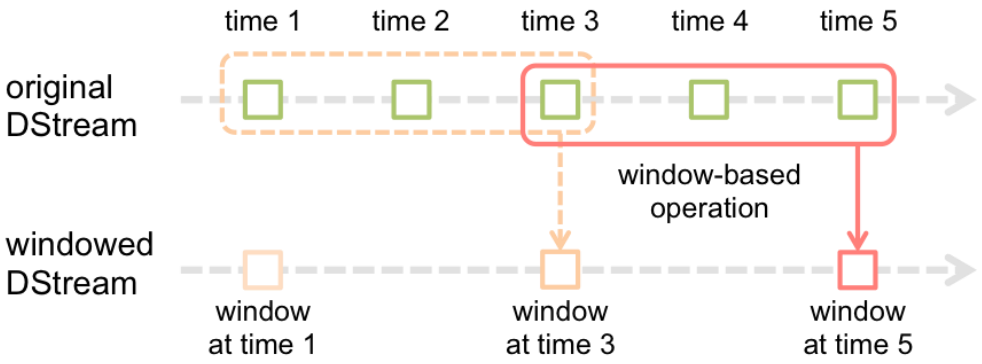
```
…
def updateFunc(new_values, last_sum):
        return sum(new_values) + (last_sum or 0)

lines = ssc.socketTextStream(sys.argv[1], int(sys.argv[2]))
running_counts = lines.flatMap(lambda line: line.split(" "))\
                      .map(lambda word: (word, 1))\
                      .updateStateByKey(updateFunc)

running_counts.pprint()

# Start the processing pipeline
ssc.start()
ssc.awaitTermination()
```

Spark Streaming

Window Operations

Window Operations

- Transformations over a **sliding window** of data, i.e. most recent stream fragment of fixed length



Spark Streaming Programming Guide v1.2, http://spark.apache.org/docs/latest/streaming-programming-guide.html

- Here, a **windowed operation** is performed every 2 sec. (= **sliding interval**) over the last 3 sec. of data (= **window length**)
- Example: create a new DStream via

  window (windowLength, slideInterval)

Overview: Window Operations

- **window** (*windowLength*, *slideInterval*)
  - Construct a new Dstream (Example 1)
- **reduceByKeyAndWindow** (*func[, invFunc]*, *windowLength*, *slideInterval*, [*numTasks*])
  - Example 2
- **countByWindow** (*windowLength*, *slideInterval*)
  - Sliding window count of elements in the stream
- **reduceByWindow** (*func*, *windowLength*, *slideInterval*)
  - A new single-element stream from aggregating elements over a sliding interval using func

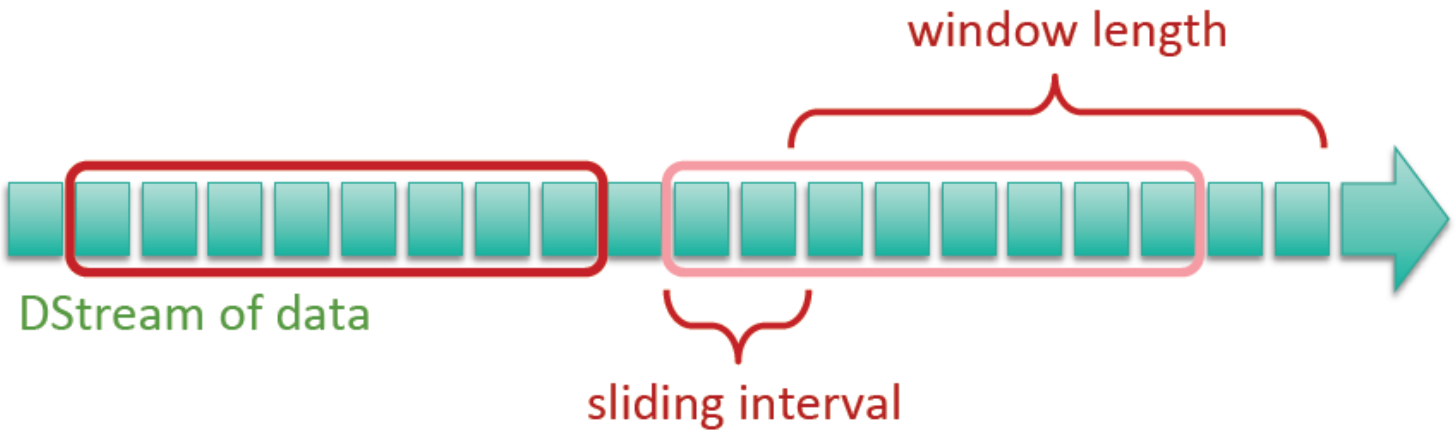
Window Op Example 1 (Scala)

```
val ssc = new StreamingContext (sparkContext, Seconds(1))
val hashTags = tweets.flatMap(status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
```

Sliding window operation

Window **length**

Sliding **interval**



From: Tathagata Das, Spark Streaming, Spark Summit 2014

Word Count in a Window (Ex. 2)

- We do count of word occurrences every 10 sec. over the last 30 sec. of stream data
  - Again, input stream contains pairs (<word>, 1)

```
windowedWordCounts =
pairs.reduceByKeyAndWindow( lambda x, y: x + y,
lambda x, y: x - y, 30, 10)
```

- Here: reduceByKeyAndWindow (func, invFunc, windowLength, slideInterval, [numTasks])
- func is clear (= adding up counts), but why invFunc?
- invFunc "substracts" values which leave the window
  - => Efficient handling by reusing the result of previous window

Thank you.

Questions?

Additional Slides