Large Scale Distributed Data Processing
Big Streaming Data, Spark Streaming

Silviu Maniu

March 23rd, 2023

Table of contents

Streaming Data

Much of data in Big Data is received *sequentially* – streaming data

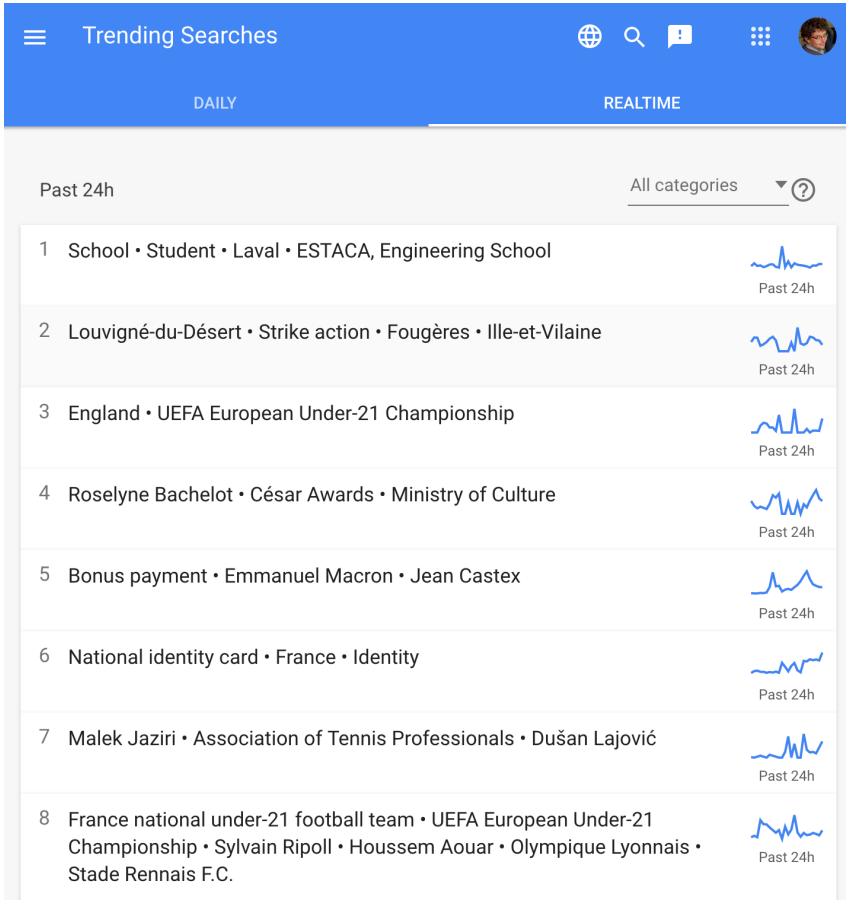
Examples:

- detecting trending hashtags in Twitter
- modeling users who visit new pages
- monitoring logs to detect faults
- measuring sensor data and making decisions based on it

Example – Web Searches

Amount: millions of queries per day

- what are the top-10 queries in a country?
- which are the popular and trending terms?
- what ads should be served for this user and query?



Streams – Challenges

Streaming data – different algorithmic challenges than offline data

Data characteristics

- data *sequence* is potentially infinite
- data *velocity* can be very high
- ephemereal data – once an instance arrives and is processed it then is discarded

Algorithmic challenges

- *data size* – sublinear space
- *data speed* – sublinear time per instance

The Main Flow

Basic operations on streams:

- mapping data – transforming the data depending on some rules
- filtering data – keeping only some data from the stream

Depending on space/time constaints – even basic algorithms (aggregates, filters, etc.) can become costly

Algorithm Example 1 – Missing Numbers

Missing Numbers Problem
Let π be a permutation of the numbers $\{1, \ldots, n\}$ (n known), and π_{-1} be π with one element missing; assume also that $\pi_{-1}[i]$ arrives in order ($\pi_{-1}[1]$ is the first element in the permutations, $\pi_{-1}[2]$ is the second, ...).

Find the missing number.

Solutions

- basic solution: keep a bit vector of n size, space complexity $\mathcal{O}(n)$ (*linear*);
- better solution in $\mathcal{O}(\log n)$: store only

$$\frac{n(n+1)}{2} - \sum_{j \leq i} \pi_{-1}[j].$$

Algorithm Example 2 – Counting

Counting Number of Elements
Easy to count exactly – keep a register of $\log N$ bits.

Morris' algorithm [Morris, 1978] – allows to keep approximate counts in $\log \log N$ space

- precursor to streaming algorithmics
- *main idea*: "skip" the counting of some elements, return an estimation
- trade-off: space for approximate solution, with variance in estimation

Algorithm Example 2 – Counting

Morris counting algorithm

initialize counter $c \leftarrow 0$

for *every element in the stream* **do**

  draw random number $d \in [0, 1]$

  if $d < p$ **then**

    $c \leftarrow c + 1$

  end

end

Performance depends on how p is set:

p	estimation	variance	bits
1	$\mathrm{E}[c] = n$	0	$\log n$
$1/2$	$\mathrm{E}[2c] = n$	$n/4$	$\log n - 1$
2^{-c}	$\mathrm{E}[2^c] = n + 2$	$\frac{n(n+1)}{2}$	$\log \log n$

Algorithm Example 2 – Counting

There are ways to reduce the variance:

1. use a basis $b < 2$, so $p = 2^{-b}$, estimator becomes b^c and variance reduces to $n\sqrt{(b-1)/2}$ – more space: $\log\log n - \log\log b$
2. run r estimators in parallel (so running time increases r-fold), average the estimators, variance reduces to $n/\sqrt{2r}$ – more space: $r \log\log n$

Algorithm Example 3 – CountMin Sketch

Counting Frequencies
Assume one has a stream of items drawn from an universe $\mathcal{U}$. Maintain a count of *frequencies* of distinct items.

Naïve solution – keep a $\mathcal{O}(|\mathcal{U}|)$ hash table linear cost

Efficient solution – sacrifice exact computation for space concerns

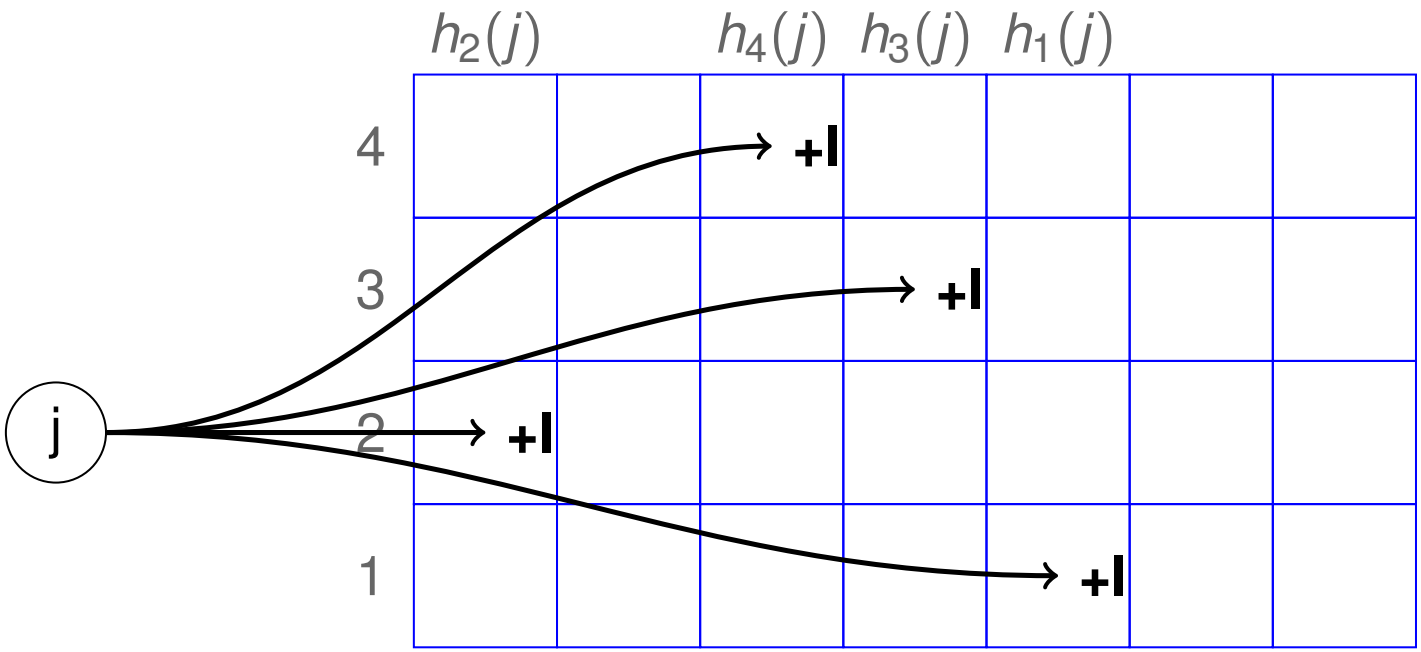
- (δ, ϵ)-approximate solutions for X – it outputs an approximation $\tilde{X}$ for which:

$$\Pr[|\tilde{X} - X| > \epsilon F] < \delta.$$

Algorithm Example 3 – CountMin Sketch

CountMin sketch [Cormode and Muthukrishnan, 2005] – data structure containing $d \times w$ cells:

- each of the d rows is a pairwise-independent hash function mapping to w cells



Algorithm Example 3 – CountMin Sketch

Operations

- UPDATE(i) – for every row j in $1, \ldots, d$ increment the corresponding cell $h_j(i)$
- QUERY(i) – return $\min_{j=1}^{d} h_j(i)$ (the smallest value in the cells corresponding to i)

CountMin sketch property

For a CountMin sketch having width $w = \mathrm{e}/\epsilon$ and depth $d = \ln 1/\delta$, QUERY returns an (ϵ, δ) approximation of the real frequencies of the items in $\mathcal{U}$.

- space complexity $\mathcal{O}(wd) = \mathcal{O}(\frac{\mathrm{e}}{\epsilon} \ln \frac{1}{\delta})$
- update/query complexity $\mathcal{O}(d) = \mathcal{O}(\ln \frac{1}{\delta})$

Data Stream Mining Algorithms

Very rich algorithms research subject [Aggarwal, 2007]:

- sampling items from a stream
- counting distinct items
- keeping statistics over sliding windows
- testing membership (Bloom filters)

What about systems?

Table of contents

Streaming Data Systems

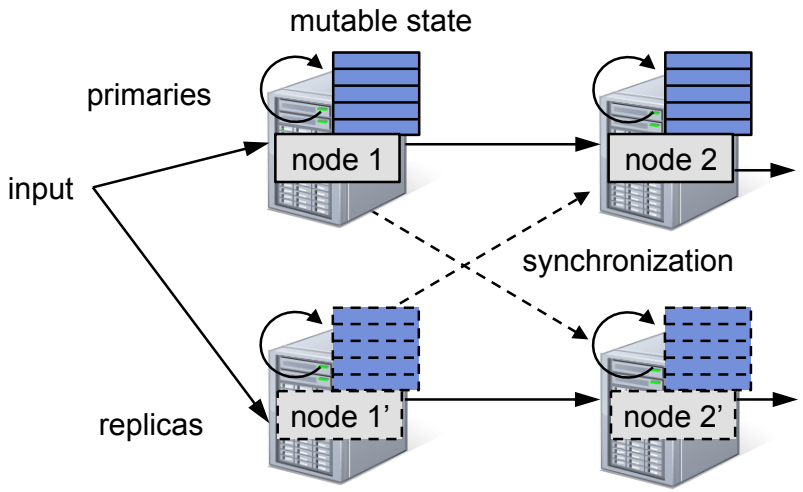
Desiderata:

- scale to hundreds of nodes
- low latency – to process new instances quickly
- efficient recovery from failures
- **exactly once**: each record should be processed only once

Main approaches

- interactive processive – every instance is processed as it comes, low-latency approach
- batch processing – instances are grouped in *batches* which are then processed

Interactive Processing Systems

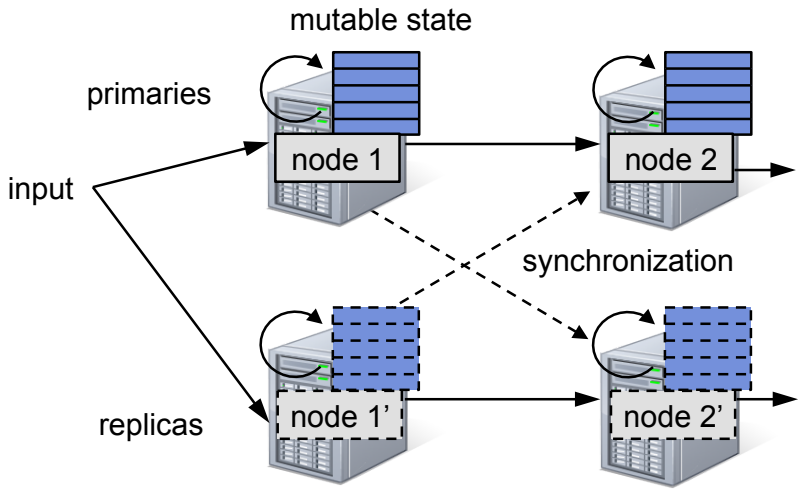


Also called continuous operator models:

- a flow of stateful operators, each node processed instances and updates their state
- depending on state, send new instances in response

Main problem – fault tolerance: state is lost if node fails

Fault Tolerance in Streaming



Replication vs. Upstream backup

1. replication: several copies are kept, and record are duplicated – it needs a *synchronization protocol* (Flux, DPC); costly, but quick
2. upstream backup: each node keeps copy of the messages; when failure, nodes replay the messages – high recovery times

Neither can handle stragglers eficiently – nodes which are slow in processing

Data Streaming Systems

Storm 

- replays record if not processed
- processes each record at least once
- may update state more than one time
- states can be lost in failures

Trident (on top of Storm)

- use transactions to update state
- each record is processed exactly one

Flink 

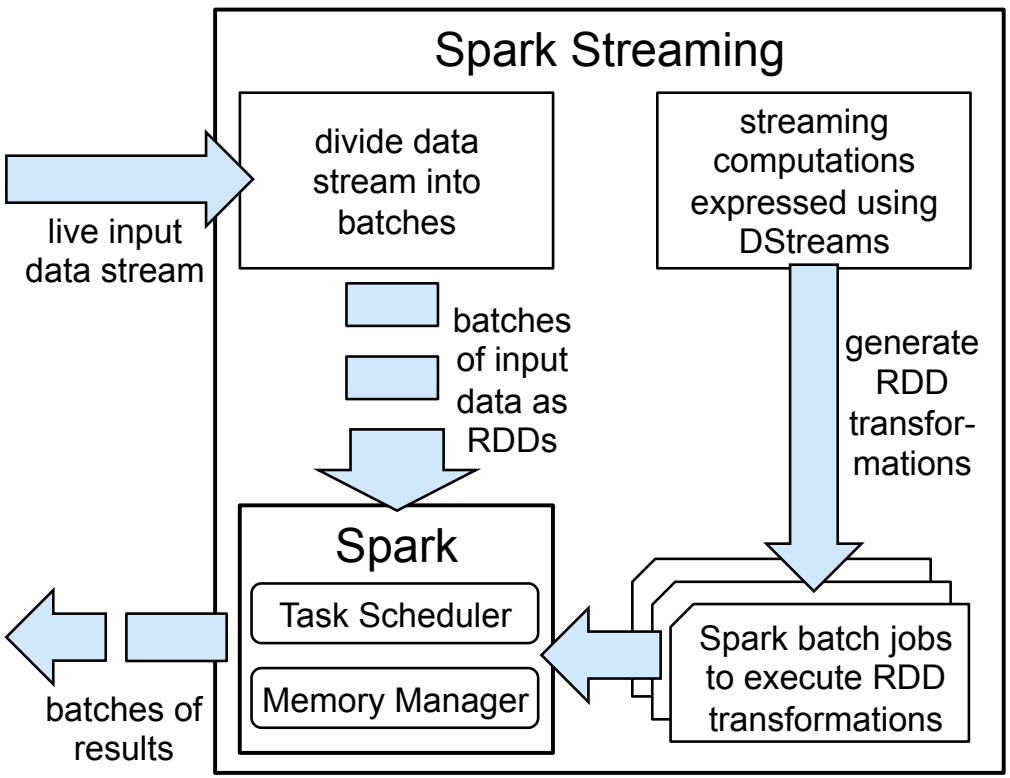
- lightweight fault tolerance: distributed checkpoints (exactly-once)
- also allows manual checkpointing

Table of contents

Spark Streaming

Batch processing on top of Spark:

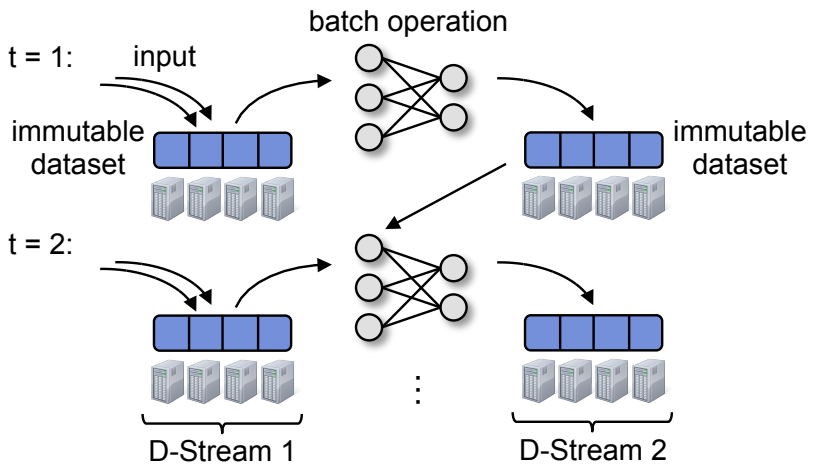
- data stream divided into *batches* of a few seconds
- Spark treats each batch as an RDD (so RDD operations apply)
- results are also pushed in batches



Discretized Streams

Spark Streaming uses a computation model called Discretized Streams [Zaharia et al., 2013]

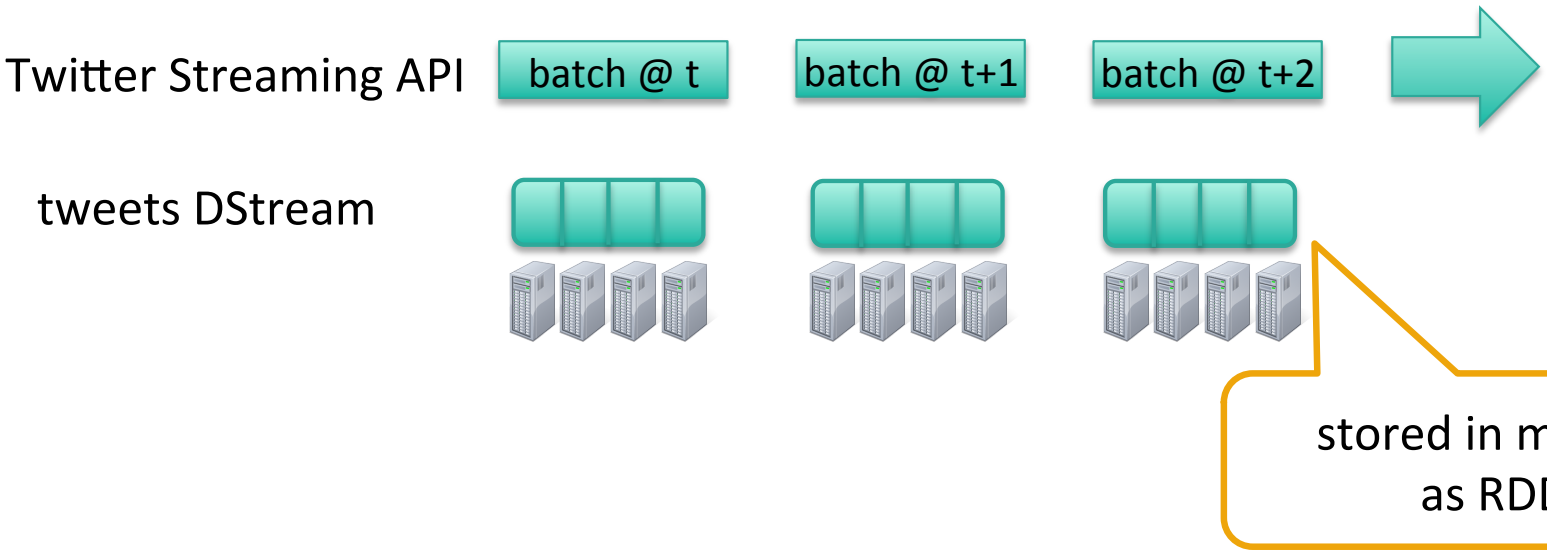
- data received in each time interval is stored on the cluster and is a dataset
- once the time interval is over, the dataset is processed via parallel operations
- *programming*: manipulating objects called DStreams, using Spark transformations on them, output also as DStreams



Example – Hashtags in Twitter

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

Input DStream



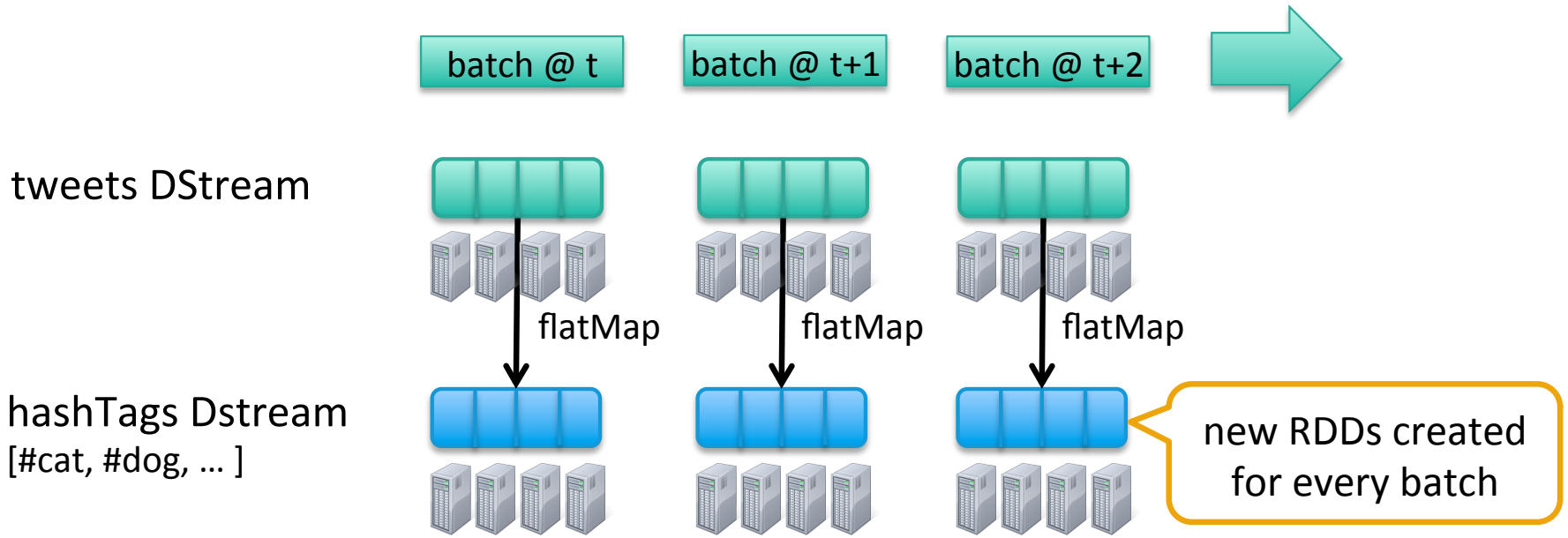
taken from `https://stanford.edu/~rezab/sparkclass/slides/td_streaming.pdf`

Example – Hashtags in Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
```

transformed DStream

transformation: modify data in one DStream to create another DStream

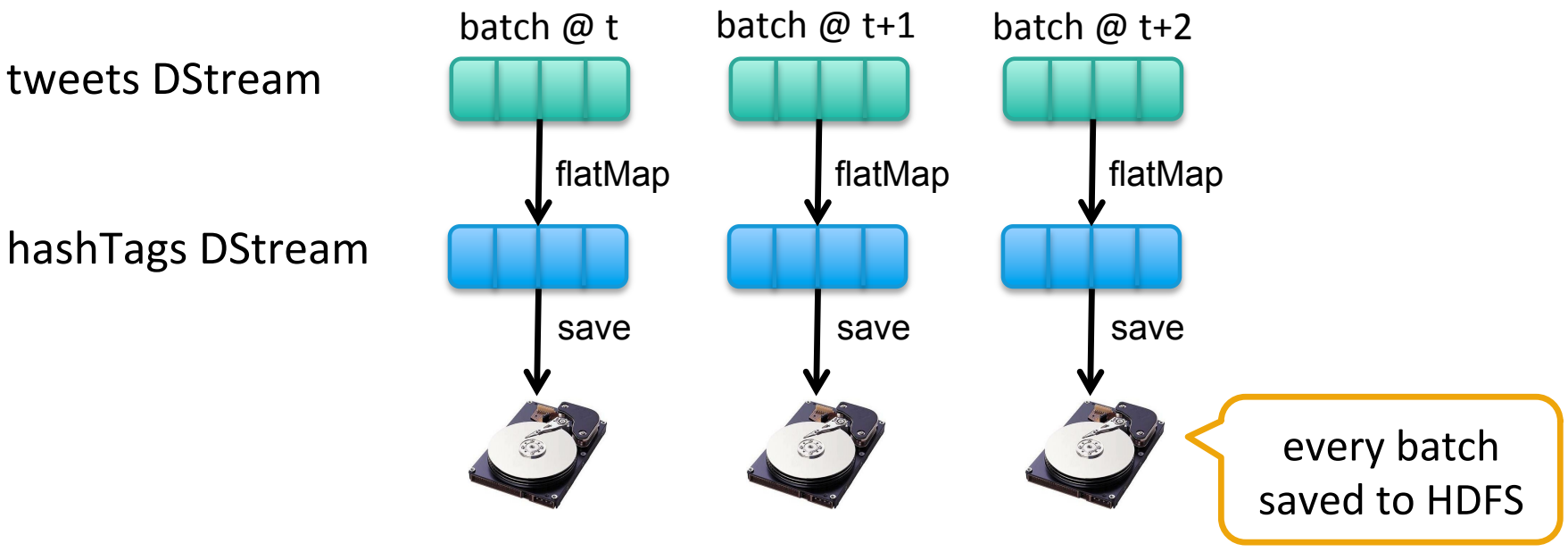


taken from https://stanford.edu/~rezab/sparkclass/slides/td_streaming.pdf

Example – Hashtags in Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

output operation: to push data to external storage

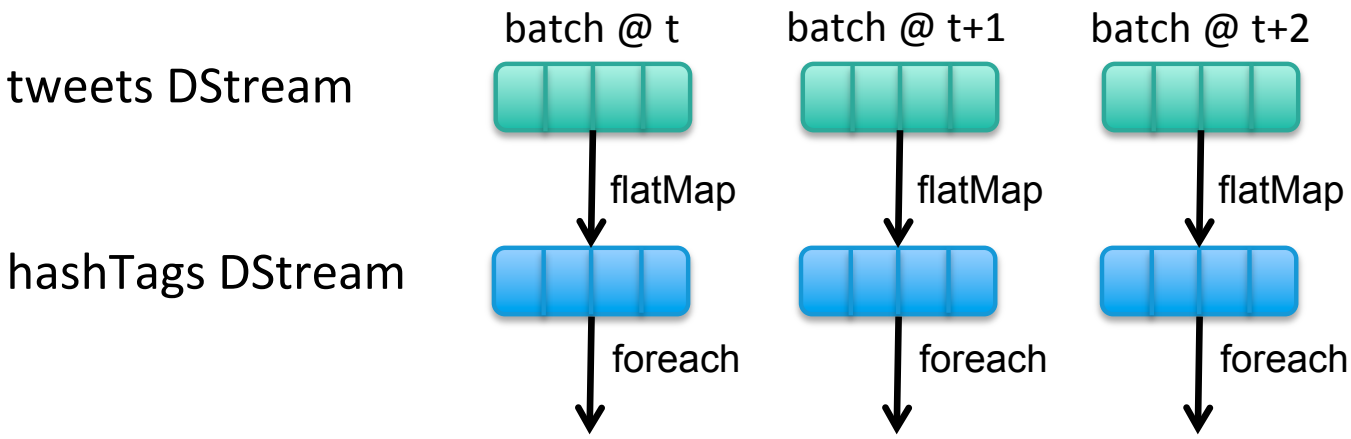


taken from https://stanford.edu/~rezab/sparkclass/slides/td_streaming.pdf

Example – Hashtags in Twitter

```
val tweets = TwitterUtils.createStream(ssc, None)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.foreachRDD(hashTagRDD => { ... })
```

foreach: do whatever you want with the processed data



Write to a database, update analytics UI, do whatever you want

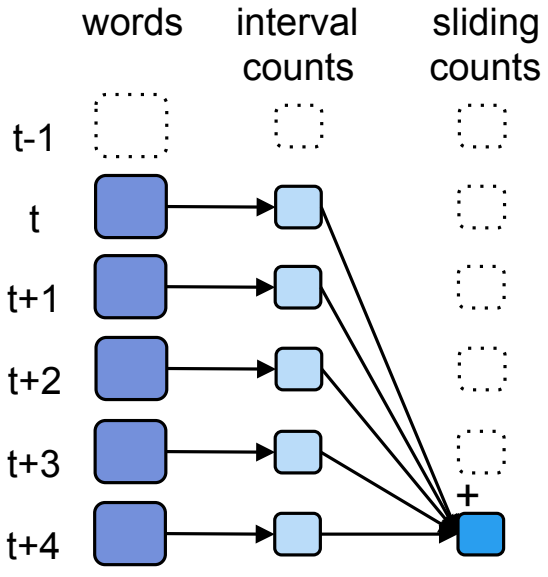
taken from https://stanford.edu/~rezab/sparkclass/slides/td_streaming.pdf

Window Operations

```
pairs = words.map(w => (w,1))
counts = pairs.reduceByKey((a, b) => a + b)
```

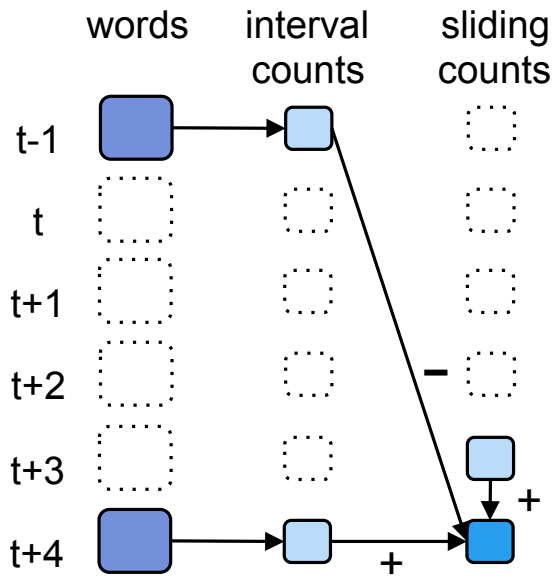
associative merge

```
pairs.reduceByWindow("5s", (a,b)
=> a+b, (a,b) => a-b)
```



associative and invertible merge

```
pairs.reduceByWindow("5s", (a,b)
=> a+b, (a,b) => a-b)
```

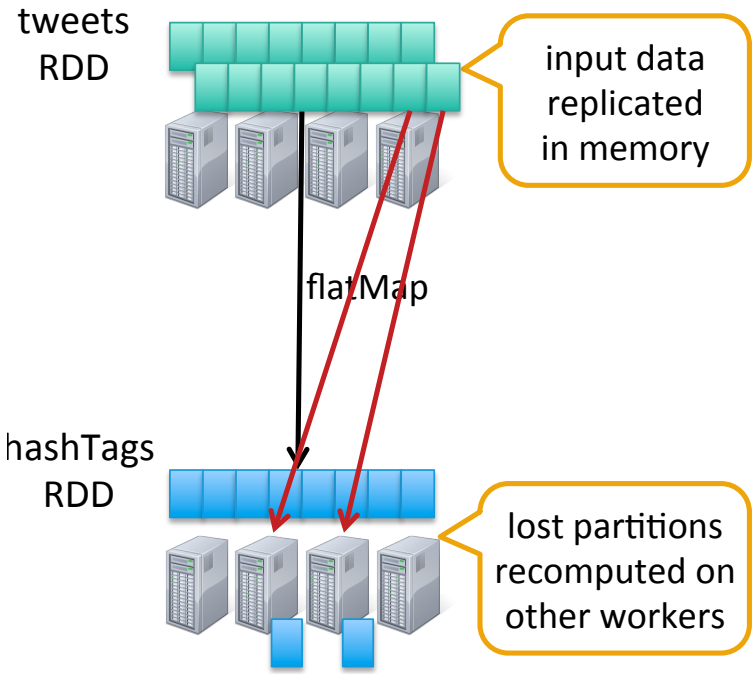


Fault Tolerance in Spark Streaming

batches of data are replicated in memory

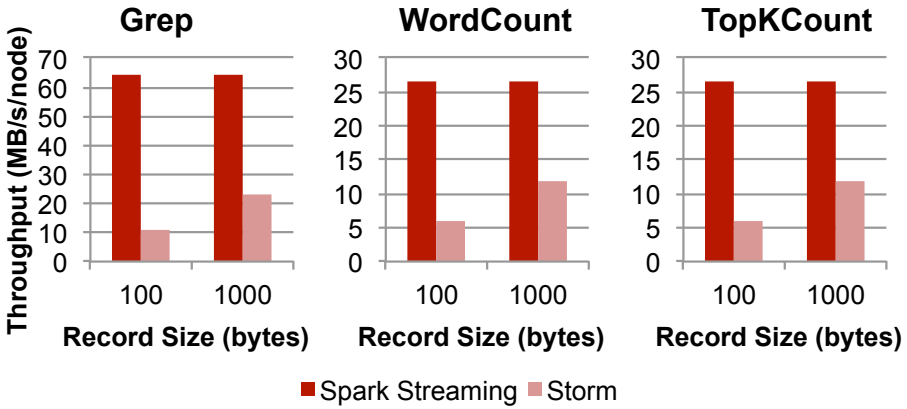
data lost is recomputed from replicated data – also helps with stragglers

all transformations are exactly-once

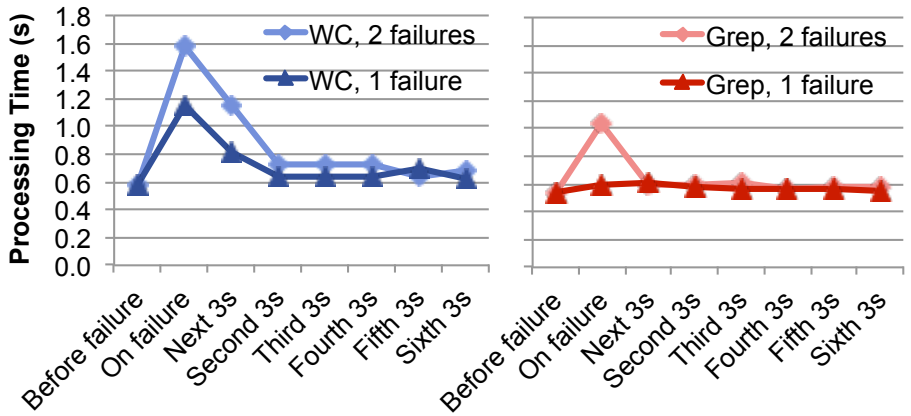


Spark Streaming Performance

throughtput vs. Storm



performance under failures



Further Reading

For in-depth details:

1. read the original paper on discretized streams [Zaharia et al., 2013]
2. read the Spark Streaming programming guide `https://spark.apache.org/docs/latest/streaming-programming-guide.html`

Sources

The contents is partly inspired by the following lectures:

- Albert Bifet's *Big Data Streaming* course `https://albertbifet.com/dk-iot-big-data-stream-mining-2018-2019/`
- Ricard Gavaldà's *Seminar on Data Streams* `http://datastreamsspring2015.blogspot.com/`
- Tathagata Das' presentation on Spark Streaming `https://stanford.edu/~rezab/sparkclass/`

Some figures are also taken from the references below.

References i


Aggarwal, C. C., editor (2007).
***Data Streams - Models and Algorithms*, volume 31 of *Advances in Database Systems*.**
Springer.


Cormode, G. and Muthukrishnan, S. (2005).
An improved data stream summary: the count-min sketch and its applications.
J. Algorithms, 55(1):58–75.


Morris, R. (1978).
Counting large numbers of events in small registers.
Commun. ACM, 21(10):840–842.

References ii

Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013).
Discretized streams: Fault-tolerant streaming computation at scale.
In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, page 423–438.