Mining Massive Datasets

Lecture 12

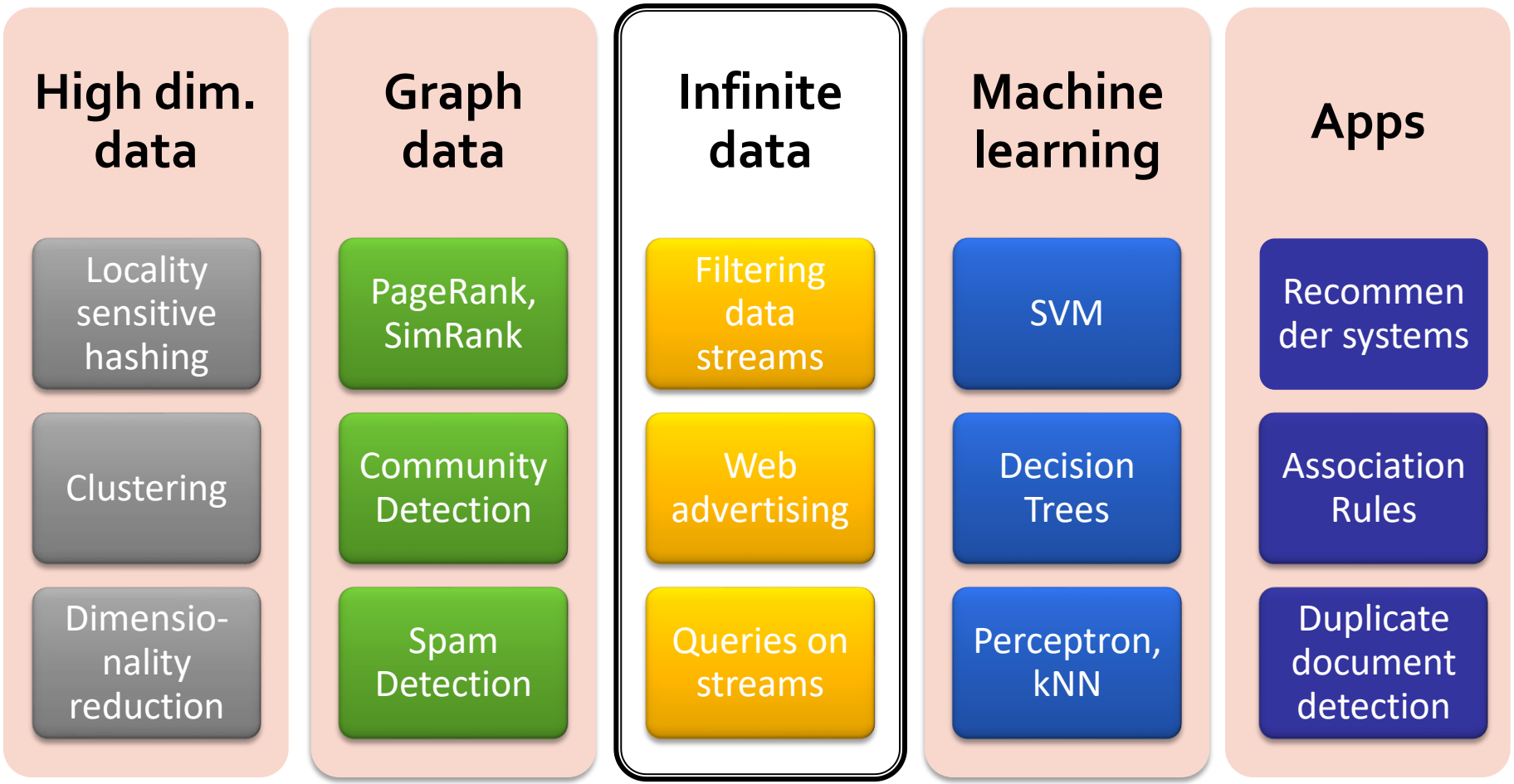
Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de

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

Mining Data Streams

Data Streams

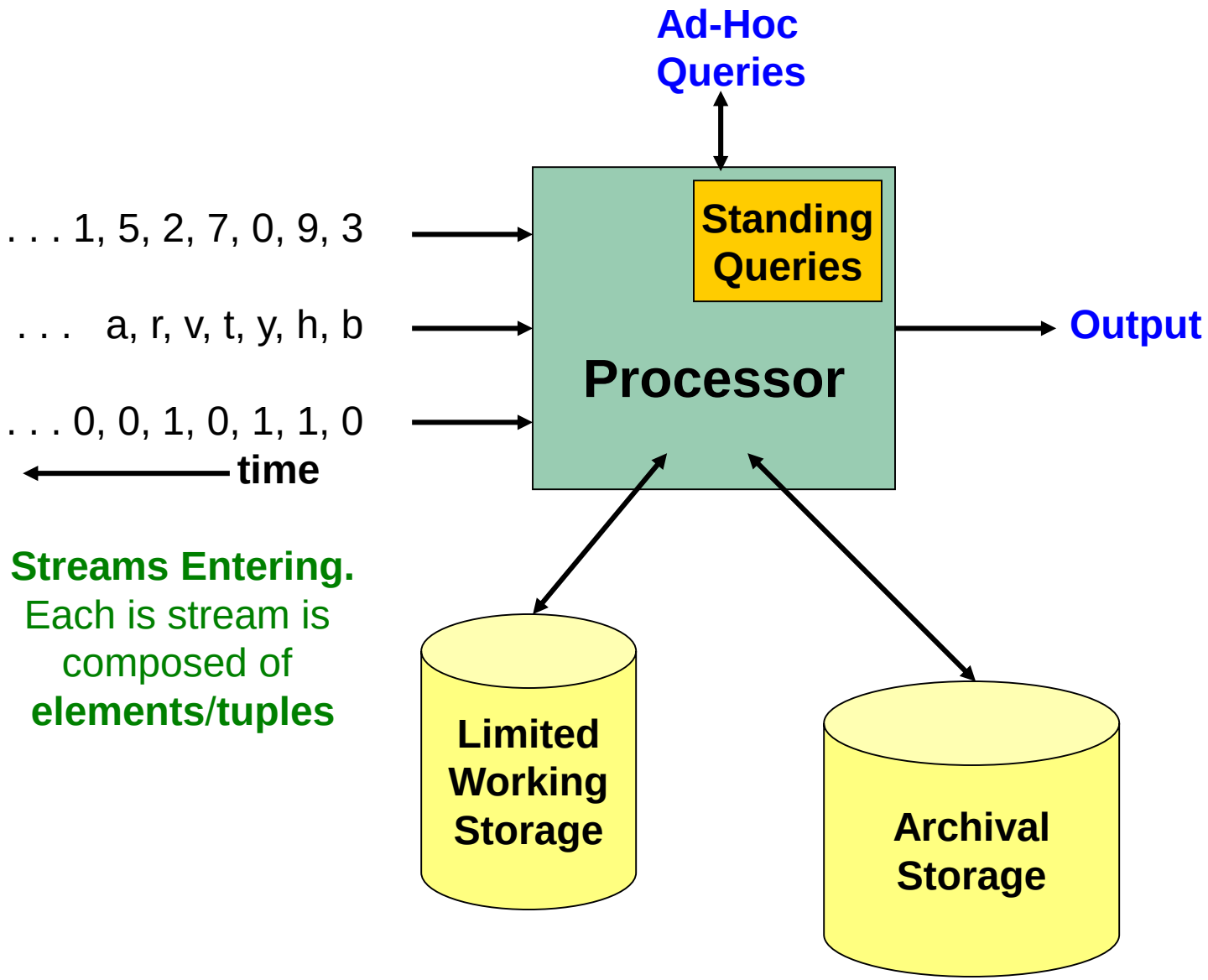
- In many data mining situations, we do not know the entire data set in advance

- **Stream Management** is important when the input rate is controlled **externally:**
  - Google queries
  - Twitter or Facebook status updates
- We can think of the **data** as **infinite** and **non-stationary** (the distribution changes over time)

The Stream Model

- Input **elements** enter at a rapid rate, at one or more input ports (i.e., **streams**)
  - We call elements of the stream **tuples**
- The system cannot store the entire stream accessibly
- **Q: How do you make critical calculations about the stream using a limited amount of (secondary) memory?**

General Stream Processing Model



Streams Entering.
Each is stream is composed of **elements/tuples**

Problems on Data Streams /1

- **Types of queries one wants on answer on a data stream:** (simpler)
  - **Sampling data from a stream**
    - Construct a random sample
  - **Queries over sliding windows**
    - Number of items of type ***x*** in the last ***k*** elements of the stream
    - Or average, maximum, minimum, …

Problems on Data Streams /2

- **Types of queries one wants on answer on a data stream:** (complex)
  - **Filtering a data stream**
    - Select elements with property ***x*** from the stream
  - **Counting distinct elements**
    - Number of distinct elements in the last ***k*** elements of the stream
  - **Estimating moments**
    - Estimate avg./std. dev. of last ***k*** elements
  - **Finding frequent elements**

Applications /1

- **Mining query streams**
  - Google wants to know what queries are more frequent today than yesterday
- **Mining click streams**
  - Yahoo wants to know which of its pages are getting an unusual number of hits in the past hour
- **Mining social network news feeds**
  - E.g., look for trending topics on Twitter, Facebook

Applications /2

- **Sensor Networks**
  - Many sensors feeding into a central controller
- **Telephone call records**
  - Data feeds into customer bills as well as settlements between telephone companies
- **IP packets monitored at a switch**
  - Gather information for optimal routing
  - Detect denial-of-service attacks

Stream Processing - Contents

- **Today**
  - **Stream filtering**
  - **Sampling a fixed proportion of a stream**
    - Sample size grows as the stream grows
  - **Sampling a fixed-size sample**
    - Reservoir sampling
  - Maybe: Spark Streaming

Filtering Data Streams

Filtering Data Streams

- **Each element of data stream is a tuple**
- Given a list of keys **S, determine which tuples of the stream are in *S***

- **Obvious solution: Hash table**
  - But suppose we **do not have enough memory** to store all of ***S*** in a hash table
    - E.g., we might be processing millions of filters on the same stream

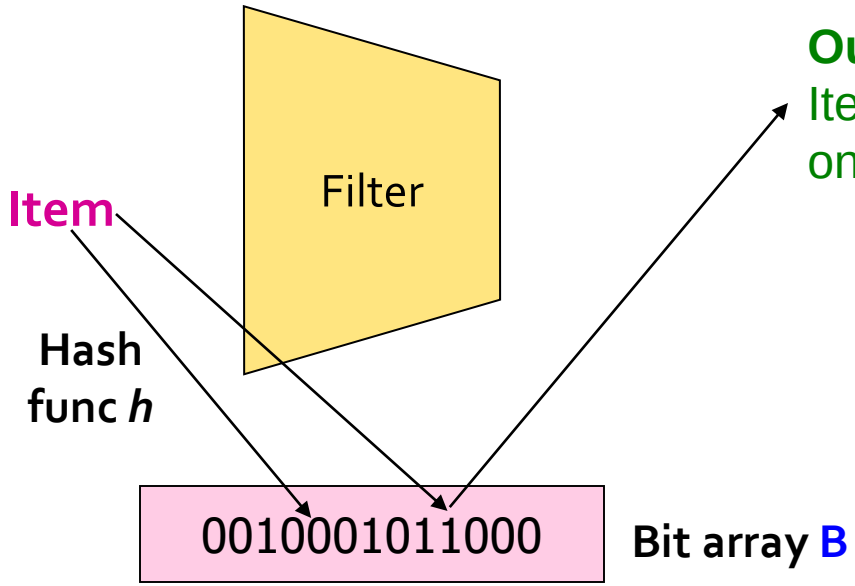
Applications

- **Example: Email spam filtering**
  - We know 1 billion "good" email addresses
  - If an email comes from one of these, it is **NOT** spam
- **Publish-subscribe systems**
  - You are collecting lots of messages (news articles)
  - People express interest in certain sets of keywords
  - Determine whether each message matches user's interest

First Cut Solution (1)

- Given a set of keys *S* that we want to filter …
- Create a **bit array *B*** of ***n*** bits, initially all ***0*s**
- Choose a **hash function *h*** with range **[*0*,*n*)**
- Hash each member of ***s* ∈ *S*** to one of ***n*** buckets, and set that bit to **1**, i.e., ***B[h(s)]=1***
- Hash each element ***a*** of the stream and output only those that hash to bit that was set to **1**
  - **Output *a* if B[h(a)] == 1**

First Cut Solution (2)



Output the item since it may be in *S*.
Item hashes to a bucket that at least one of the items in **S** hashed to.

Drop the item.
It hashes to a bucket set to **0** so it is surely not in ***S***.

- **Creates false positives but no false negatives**
  - If the item is in ***S*** we surely output it,
  - If not in ***S***: we may still output it (erroneously)

First Cut Solution (3)

- **|S| = 1 billion email addresses**
  |B|= 1GB = 8 billion bits
- If the email address is in ***S***, then it surely hashes to a bucket that has the big set to **1**, so it always gets through (***no false negatives***)
- Approximately **1/8** of the bits are set to **1**, so about **1/8th** of the addresses not in ***S*** get through to the output (***false positives***)
  - Actually, less than **1/8th**, because more than one address might hash to the same bit

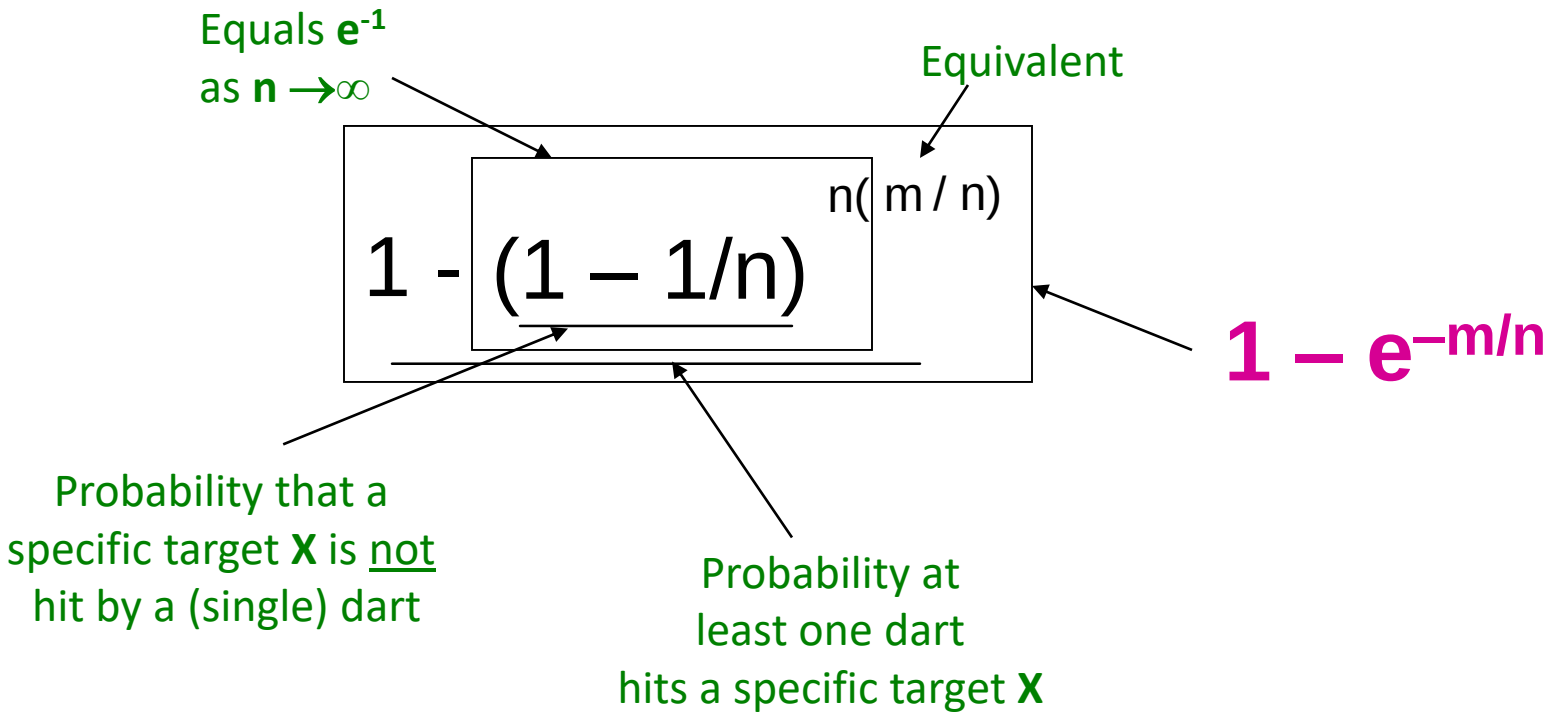
Analysis: Throwing Darts (1)

- **More accurate analysis for the number of false positives**
- **Consider:** If we throw ***m*** darts into ***n*** equally likely targets, what is the probability that a target gets at least one dart?
  - Every dart hits (but don't know which target)
- **In our case:**
  - **Targets** = bits/buckets
  - **Darts** = hash values of items

We want to obtain prob. that a "random" bit of array B is set to 1 (by one of the addresses in S) = fraction of 1s in B

Analysis: Throwing Darts (2)

- We have ***m*** darts, ***n*** targets (each dart hits)
- What is the probability that a specific target X gets at least one dart?



Analysis: Throwing Darts (3)

- **Fraction of 1s in the array B =**
  = probability of false positive = $1 - e^{-m/n}$

- **Example:** m=10^9 darts, n=$8 \cdot 10^9$ targets
  - Fraction of **1s** in **B** = **$1 - e^{-1/8} = 0.1175$**
    - Compare with our earlier estimate: **1/8 = 0.125**

Bloom Filter: Algorithm

- Consider: **|S| = *m*,** bitset **B, |B| = *n***
- Use ***k*** independent hash functions $h_1, \ldots, h_k$
- **Initialization:**
  - Set **B** to **0s**
  - Hash each element $s \in S$ using each hash function h_i, set $B[h_i(s)] = 1$ (for each $i = 1, .., k$)

(**note:** we have a single bitset B!)

- **Run-time:**
  - When a stream element with key ***x*** arrives
    - If $B[h_i(x)] = 1$ **for all** $i = 1, \ldots, k$ then declare that ***x*** is in ***S***
      - That is, ***x*** hashes to a bucket set to **1** for every hash function $h_i(x)$
    - Otherwise discard the element ***x***

Independent Hash Functions

- Input: a binary string x, e.g. integer
- Possible h1:
  - Take all **odd**-numbered positions of x
    - I.e. positions 1, 3, 5, …
  - Treat them as a number, then compute modulo 11
  - … 11 or some other prime number
- Possible h2:
  - Take all **even**-numbered positions of x
    - i.e. positions 0, 2, 4, …
  - Treat them as a number, then compute modulo 11

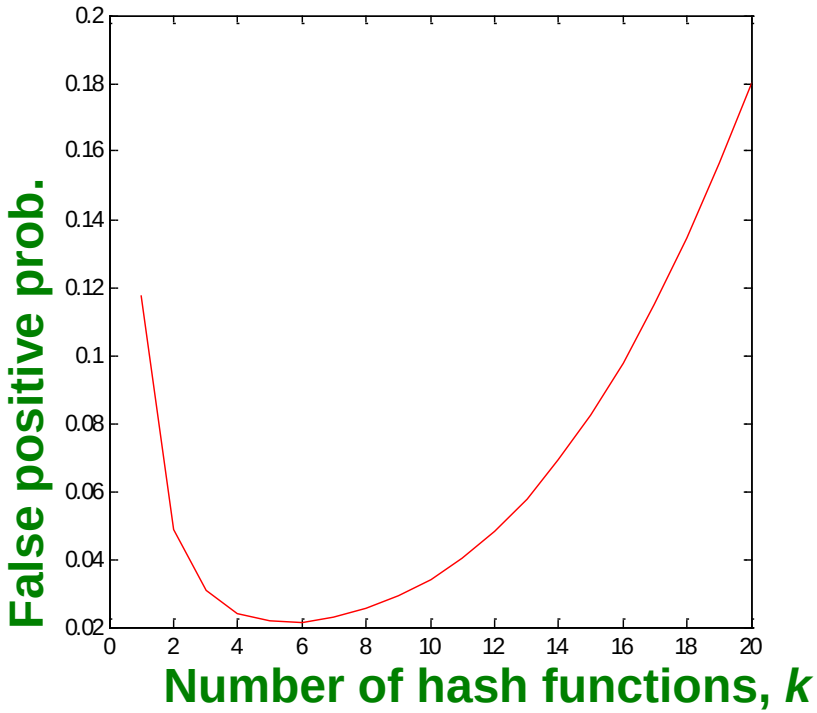
Bloom Filter -- Analysis

- **What fraction of the bit vector B are 1s?**
  - Throwing $k \cdot m$ darts at n targets
  - So fraction of **1**s is $(1 - e^{-km/n})$
- But we have k independent hash functions and we only let the element x through **if all** k hash element x to a bucket of value **1**
- So, **false positive probability** = $(1 - e^{-km/n})^k$

Bloom Filter – Analysis (2)

$(1 - e^{-km/n})^k$

- ***m* = 1 billion, *n* = 8 billion**
  - **k = 1**: $(1 - e^{-1/8})$ = **0.1175**
  - **k = 2**: $(1 - e^{-1/4})^2$ = **0.0493**



- **What happens as we keep increasing *k*?**
- "Optimal" value of ***k***: ***n/m* ln(2)**
  - **In our case:** Optimal **k = 8 ln(2) = 5.54 ≈ 6**
    - **Error at k = 6**: $(1 - e^{-1/6})^2$ = **0.0235**

Bloom Filter: Wrap-up

- **Bloom filters guarantee no false negatives, and use limited memory**
  - Great for pre-processing before more expensive checks
- **Suitable for hardware implementation**
  - Hash function computations can be parallelized

- Is it better to have **1** big **B** or ***k*** small **B**s?
  - **It is the same:** $(1 - e^{-km/n})^k$ vs. $(1 - e^{-m/(n/k)})^k$
  - But keeping **1 big B** is simpler

Sampling from a Data Stream:

Sampling a fixed proportion

Sampling from a Data Stream

- Since **we can not store the entire stream**, one obvious approach is to store a **sample**
- **Two different problems:**
  - **(1)** Sample a **fixed proportion** of elements in the stream (say 1 in 10) (=> storage grows!)
  - **(2)** Maintain a random sample **of fixed size** over a potentially infinite stream
    - At any "time" k we would like a random sample of s elements
    - **What is the property of the sample we want to maintain?**
      For all time steps ***k***, each of ***k*** elements seen so far has equal prob. of being sampled

Sampling a Fixed Proportion

- **Problem 1: Sampling fixed proportion**
- **Scenario:** Search engine query stream
  - **Stream of tuples:** (user, query, time)
  - **Answer questions such as: How often did a user run the same query in a single day?**
  - Have space to store **1/10th** of query stream
- **Naïve solution:**
  - Generate a random integer in **[0..9]** for each query
  - Store the query if the integer is **0**, otherwise discard

Problem with Naïve Approach

- **Simple question: What fraction of queries by an average search engine user are duplicates?**
  - Suppose each user issues ***x*** queries once and ***d*** queries twice => user issues ***x*+2*d*** "atomic" queries
  - => **Correct answer: *d*/(*x*+*d*)**
- **Proposed solution: We keep 10% of the queries**
  - Sample will contain ***x*/10** of the singleton queries and **2*d*/10** of the duplicate queries at least once
  - But sample contains only ***d*/100** pairs of duplicates
    - **d/100 = 1/10 · 1/10 · d**
  - Of ***d*** duplicates, ***18d/100*** appear exactly once in the sample
    - **18d/100 = ((1/10 · 9/10)+(9/10 · 1/10)) · d**

d/100 appear twice: 1st query gets sampled with prob. *1/10*, 2nd also with *1/10*, there are d such "query pairs" => *d/100*

Prob. 1/10 for first to get selected and 9/10 for the second to not get selected; and the other way around, so in total 18d/100

Very different from ***d*/(*x*+*d*)**!

- **So the sample-based answer is:** $\frac{\frac{d}{100}}{\frac{x}{10}+\frac{d}{100}+\frac{18d}{100}} = \frac{d}{10x+19d}$

Sampling by Value

- Our mistake: we sampled based on the position in the stream, rather than the value of the stream element
- **Solution:** Pick **1/10th** of **users** and take all their searches in the sample
  - Use a hash function that hashes the user name or user id uniformly into 10 buckets
- All or none of the query instances of a user are selected
  - Therefore the fraction of his duplicate queries in the sample is the same as for the stream as a whole

Generalized Solution

- **Stream of tuples with keys:**
  - Key is some subset of each tuple's components
    - e.g., tuple is (user, search, time); key is **user**
  - Choice of key depends on application

- **To get a sample of *a/b* fraction of the stream:**
  - Hash each tuple's key uniformly into ***b*** buckets
  - Pick the tuple if its hash value is $\leq$ ***a***

Hash table with **b** buckets, pick the tuple if its hash value is $\leq$ **a.**

How to generate a 30% sample?

Hash into b=10 buckets, take the tuple if it hashes to one of the first 3 buckets

Sampling from a Data Stream:

Sampling a fixed-size sample

Maintaining a fixed-size sample

- **Problem 2: Fixed-size sample**
- Suppose we need to maintain a random sample S of **size exactly *s* tuples**
  - Why? Don't know length of stream in advance
- Suppose at time n we have seen n items
  - Each item is in the sample S with equal prob. s/n

How to think about the problem: say s = 2

Stream: a x c y z k c d e g…

At **n= 5,** each of the first 5 tuples is included in the sample **S** with equal prob.

At **n= 7,** each of the first 7 tuples is included in the sample **S** with equal prob.

Impractical solution would be to store all the *n* tuples seen so far and out of them pick *s* at random

Solution: Fixed Size Sample

- **Algorithm (a.k.a. Reservoir Sampling)**
  - Store all the first ***s*** elements of the stream to ***S***
  - Suppose we have seen ***n-1*** elements, and now the ***n^{th}*** element arrives (***n > s***)
    - With probability ***s/n***, keep the ***n^{th}*** element, else discard it
    - If we picked the ***n^{th}*** element, then it replaces one of the ***s*** elements in the sample ***S***, picked uniformly at random
- **Claim:** This algorithm maintains a sample ***S*** with the desired property:
  - After ***n*** elements, the sample contains each element seen so far with probability ***s/n***

Proof: By Induction

- **We prove this by induction:**
  - Assume that after ***n*** elements, the sample contains each element seen so far with probability ***s/n***
  - We need to show that after seeing element ***n+1*** the sample maintains the property
    - Sample contains each element seen so far with probability ***s/(n+1)***
- **Base case:**
  - After we see **n=s** elements the sample **S** has the desired property
    - Each out of **n=s** elements is in the sample with probability ***s/s = 1***

Proof: By Induction

- **Inductive hypothesis:** After ***n*** elements, the sample ***S*** contains each element seen so far with prob. ***s/n***
- **Now element *n+1* arrives**
- **Inductive step:** For elements already in ***S***, probability that the algorithm keeps it in ***S*** is:

$$\left(1-\frac{s}{n+1}\right)+\left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right)=\frac{n}{n+1}$$

Element **n+1** discarded

Element **n+1** not discarded

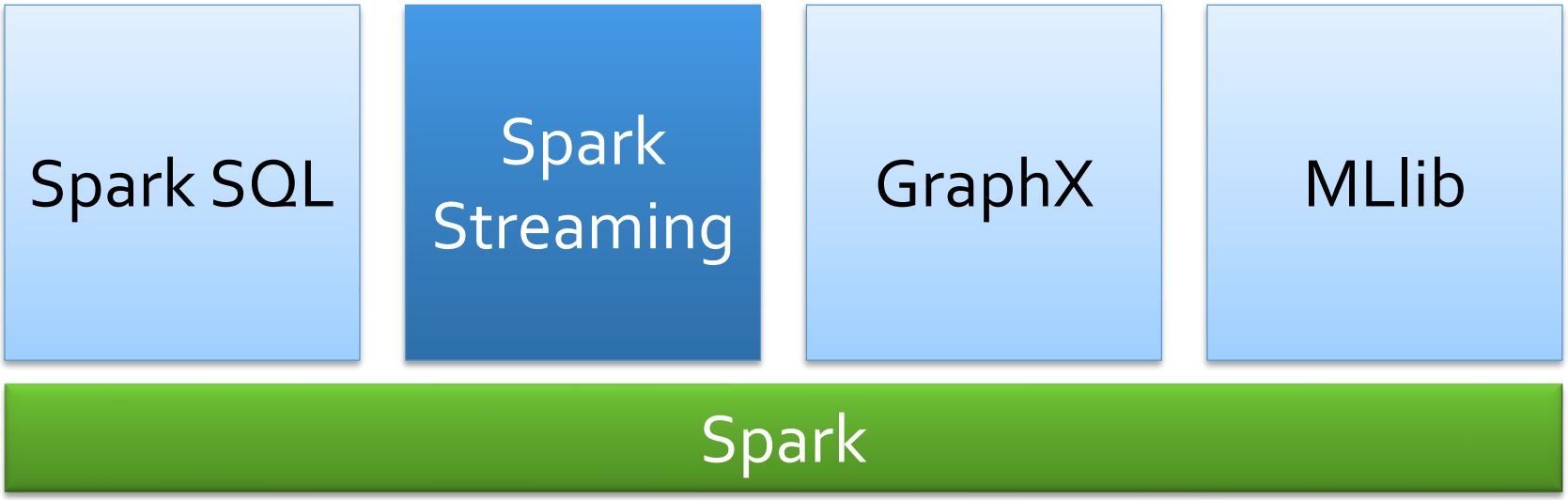
Element in the sample not picked

- So, at time ***n,*** a tuple in ***S*** was there with prob. **s/n**
- Time ***n→n+1,*** tuple stayed in ***S*** with prob. **n/(n+1)**
- So prob. tuple is in ***S*** at time ***n+1*** = $\frac{s}{n}\cdot\frac{n}{n+1}=\frac{s}{n+1}$

Spark Streaming

Overview

What is Spark Streaming?



- Spark library / module: extends Spark for (large-scale) distributed data stream processing
- Started in 2012, included in 2014 in Spark 0.9
- Bindings in Spark version 1.2+
  - Scala, Java, Python (partial)

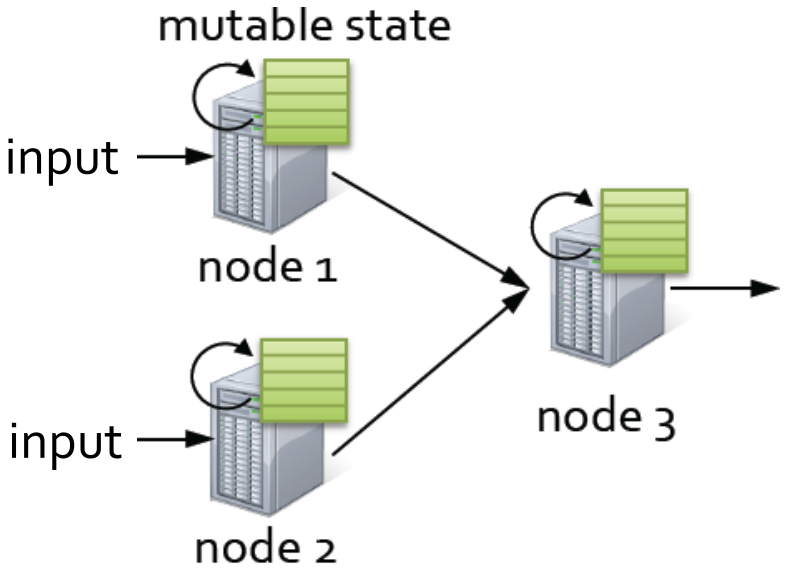
Problem 1: Stream vs. Batch Processing

- Many apps require processing the same data in live streaming as well as in batches
  - E.g. finance: trading robots / high freq. trading
    - Batch: testing and evaluating trading systems (backtests)
    - Stream: live trading using prepared systems
  - Detecting DoS attacks
    - Batch: understand patterns of DoS, tune algorithms
    - Stream: apply prepared algorithms to live data
- => Need for two separate programming models
  - Doubled effort, inconsistency, hard to debug

Problem 2: Fault-tolerance

- Traditional processing model:
  - Pipeline of nodes
  - Each node maintain a **mutable state**
  - Each input record updates the state and new records are sent out

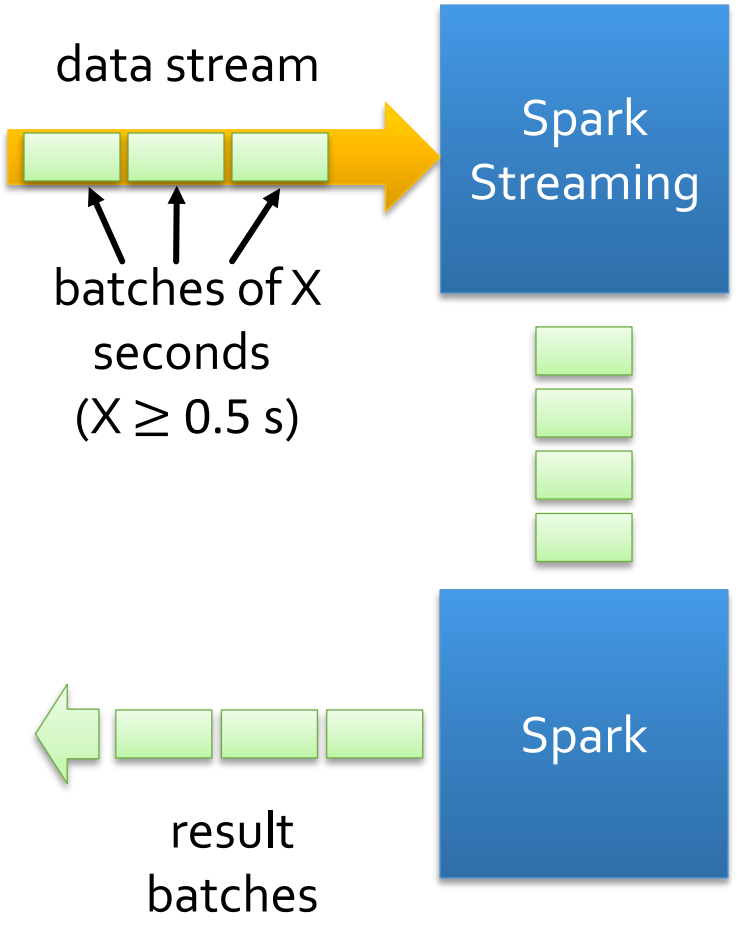
- => Mutable state is lost if node fails
- => Making stateful stream processing **fault-tolerant** is challenging



From: Tathagata Das, Spark Streaming, Spark Summit 2014

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



Data Sources and Sinks

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

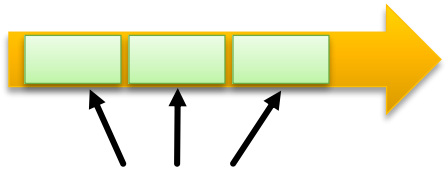
- Input data streams can come from many sources, e.g.
  - HDFS/S3 (files), TCP sockets, Kafka, Flume, Twitter, …
- Output data can be pushed out to …
  - File systems, databases, live dashboards

Spark Streaming

Basic Programming

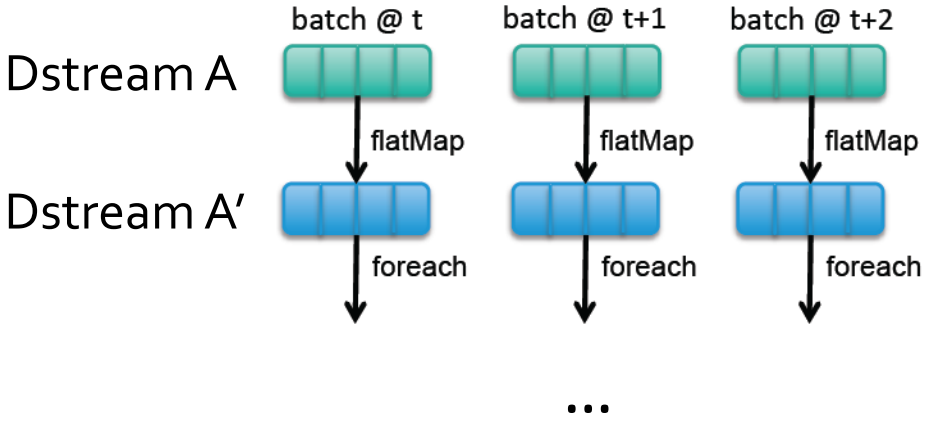
Programming Model - DStream

- **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)
- …

Thank you.

Questions?

Additional Slides