Mining Data Streams

Partially based on slides from:

http://infolab.stanford.edu/~ullman/mining/2009/index.html

Anand Rajaraman & Jeff Ullman

Contents

- Data Stream Model
  - Random Sampling of Data Streams
  - Filtering Data Streams (Bloom Filter)
  - Counting Distinct Elements
  - ~~Estimating Moments~~

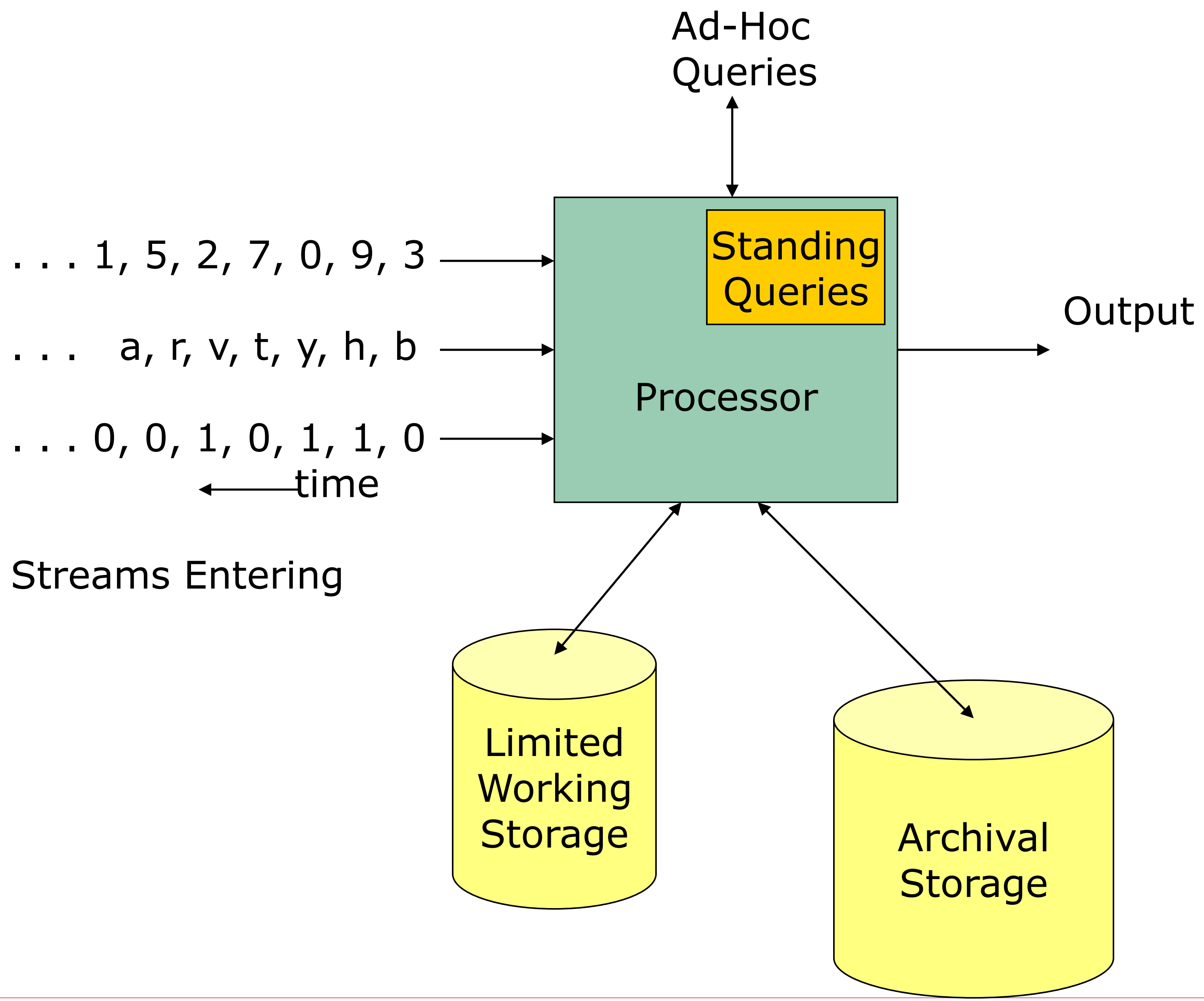
- ~~A sliding window model~~
  - ~~Counting 1's~~
  - ~~Summing integers~~

Conventional Data Mining

- All data stored on slow hard disks/archives.
- Slow access, slow processing.
- Off-line analysis of data.
- Instant action on results impossible.
- However, trained MODELS (rules, decision trees, neural networks, etc.) can be deployed to work in real time.

The Stream Model

- Data enter the system at a rapid rate,
  at one or more input ports
- The system cannot store the entire stream accessibly
- How do you make critical calculations about the stream using a limited amount of fast (RAM) memory?
- We might be interested in queries over :
  - “the last N records” (a sliding window model), or
  - “everything seen so far (or the last N days)”


Ad-Hoc Queries
. . . 1, 5, 2, 7, 0, 9, 3
. . . a, r, v, t, y, h, b
. . . 0, 0, 1, 0, 1, 1, 0
time
Streams Entering
Standing Queries
Processor
Output
Limited Working Storage
Archival Storage

Applications – (1)

- Mining Twitter's tweets
  - Detecting breaking news, disasters, scandals, …
- 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

Applications – (2)

- Sensors of all kinds need monitoring, especially when there are many sensors of the same type, feeding into a central controller.
  - Detecting/predicting traffic jams
  - Tsunami alerts

- Telephone call records are summarized into customer bills.

Applications – (3)

- IP packets can be monitored at a switch
  - Gather information for optimal routing
  - Detect denial-of-service attacks
  - Filter spam of phishing attacks

1. Sampling a data stream

- Suppose a bank would like to have (at any moment) a “representative” sample of 3% of all their data, so it would fit into RAM and could be queried interactively (quick response)
- Example query: *what is the ratio of clients with exactly 1 trx to clients which made exactly 2 trxs during the last weekend?*

Naïve Approaches

- Scan the stream of all transactions; the n-th transaction is put into the sample if
  $r=(n \bmod 100) < 3$
  (0, 1, 2, 100, 101, 102, …)

- Use a random selection with $p=3\%$

- **Will not work:**
  - Many clients with 2, 3, … transactions would be “viewed” as those which made a single transaction (see 4.2.1)

Better Approach

- Make, off-line, a list of all clients, take a 3% sample of them, put them into a hash table (or a binary search tree) and store only transactions from these selected clients.
- (If needed, use not only client_id as the key, but also other fields, like card type, terminal type, etc.)
- Problems:
  - Cost time and memory (to keep the hash table)
  - What about new clients (we will never see them!)?

An Even Better Approach

- - instead of a hash table, use a hash function to map each client to an integer 1, …, 100;
  - transactions of clients with number 1, 2, 3 enter the sample; others are ignored

- Advantage:
  - No need to create and keep any hash table!
  - New clients are also included in the sampling process

- Problem: what if the sample size exceeds available RAM?

Sampling a stream in limited RAM

- Instead of a pre-specified percentage of clients that we want to cover (e.g., 3%), specify the amount of RAM that can be used for storing the sample.

- Use a hash function with, say, **L=10.000** buckets (as before) and put all the records from buckets **1, 2, ..., L** to your sample. **L** is initially set to **10.000**.

- Whenever the size of your samples reaches the size of available RAM, **remove** all the records from **bucket L** and **set L to L-1.**

Reservoir Sampling

- Suppose your RAM can store ***s* records;** each record has the same size

- How could you sample an infinite stream of records, in such a way that at any moment your RAM keeps a **random, uniformly distributed, sample of *s* records**? (uniformly distributed: every record from the stream has the same chance of being kept in RAM, i.e.,

  ***s*/(#records seen so far)**

- **Initialization:** the first ***s* records are stored** in RAM.

- *How to proceed with further records, s+1, s+2, ... in such a way that at any moment, any element of the stream has the same chance of being in RAM?*

A Solution: Reservoir Sampling

- Initialization:
  Store the first s records of the stream in your RAM. At this moment $n=s$ and the probability of an element entering RAM the is s/n (accidentally, it's 1!)

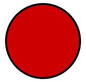
- Inductive Step:
  - When the $(n+1)^{th}$ element arrives, decide with probability $s/(n+1)$ to keep the record in RAM (otherwise, ignore it)
  - If you choose to keep it, throw one of the previously stored record out, selected with equal probability, and use the freed space for the new record.

- Prove by induction that at any moment all records enter RAM with probability s/n (Chapter 4.5.5, page 181?)

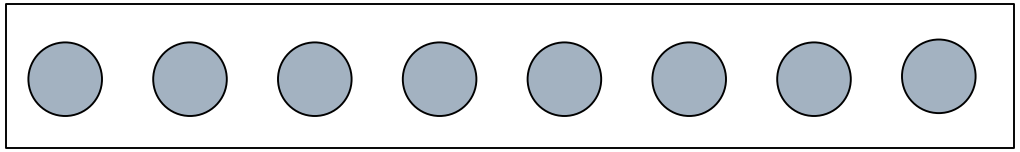
Outline of the proof

- When the $(n+1)^{th}$ element arrives it is chosen to be stored with probability $s/(n+1)$ - that's what we wanted!
- However, all previous positions were chosen with probability s/n (by induction hypothesis) - that's bad - we need $s/(n+1)$!
- Fortunately, our procedure will delete one of the stored elements to create space for the newly selected element. This "random deletion" will modify the probabilities of "surviving" from s/n to $s/(n+1)$ - exactly what we need!
- $$\left(1-\frac{s}{n+1}\right)\left(\frac{s}{n}\right)+\left(\frac{s}{n+1}\right)\left(\frac{s-1}{s}\right)\left(\frac{s}{n}\right)=\frac{s}{n+1}$$

Outline of the proof



n-th+1 element



s = buffer size

Every element has chance of ***s/n*** of being in buffer at moment ***n***; a new element arrives…

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

Don't keep

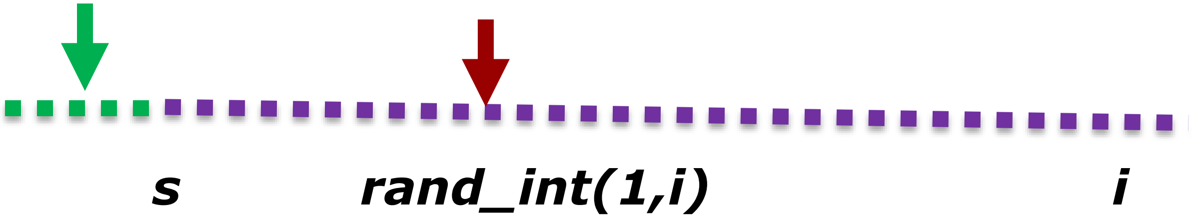
Keep

Survived

Simple implementation of Reservoir Sampling

```
(* S has items to sample, R will contain the result *)
ReservoirSample(S[1..n], R[1..s])
 // fill the reservoir array
 for i := 1 to s
   R[i] := S[i]

 // replace elements with gradually decreasing probability
 for i := s+1 to n
  (* randomInteger(a, b) generates a uniform integer from the inclusive
range {a, ..., b} *)
  j := randomInteger(1, i)
  if j <= s
    R[j] := S[i]
```



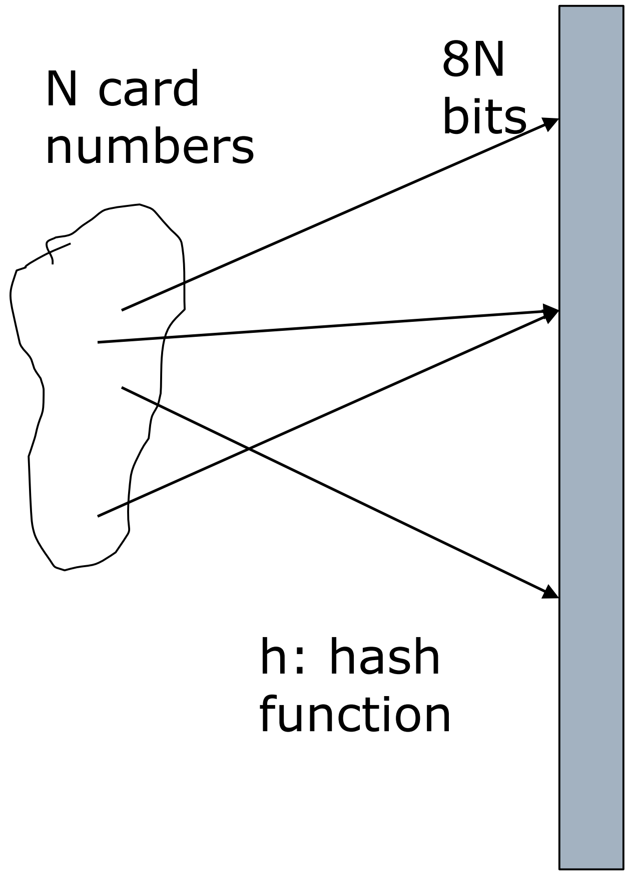
https://en.wikipedia.org/wiki/Reservoir_sampling

2. Fast filtering of “bad” records

- Sometimes we know which records are “legal” and want to filter out all (or almost all) “illegal” records from a stream
- Examples:
  - A bank wants to ignore card numbers of clients which are not in their database
  - A mailing system wants to block all incoming e-mails from “non-registered” e-mail addresses (spam?)
  - Amazon.com wants to allow access only to their registered clients
  - Scenario: 1 billion “legal” card numbers, each 28 chars long

Bloom Filter

- Scenario: N=1 billion “legal” card numbers, each 28 chars long; how could we quickly filter out ***almost all*** “illegal” numbers?
- A hash table? How big should it be? **56GB RAM.**
- **Bloom Filter (one hash function):**
  - Consider a vector of **8 billion bits** (just **1GB of RAM**!) initialized to 0's and a hash function **h(x)** with values in 1, …, 8*10^9.
  - For each “legal” card number **x**, **set bit h(x) to 1.**
  - An incoming **y** is considered to be legal if the **h(y) bit is 1.**
- What is the percentage of ***false positives***? (illegal cards that would pass through the filter)



Bloom Filter: accuracy

- How many bits are set to 1?
  - n=10^9 legal cards
  - N=8*10^9 buckets

- What is the chance that a randomly selected bit is **not set to 1**?

$$((N-1)/N)^n = [(1-1/N)^N]^{n/N} = e^{-1/8}$$

- Thus the chance randomly hitting 1 is 1-exp(-1/8)=**0.1175**, so the rate of false positive is about **11.75%**

- Suppose that instead of one hash function we would use **k hash functions, $h_1, \ldots, h_k$** and demand that y passes the filter when **all bits $h_1(y), \ldots, h_k(y)$ are set to 1** ...

Bloom Filter with *k* hash functions

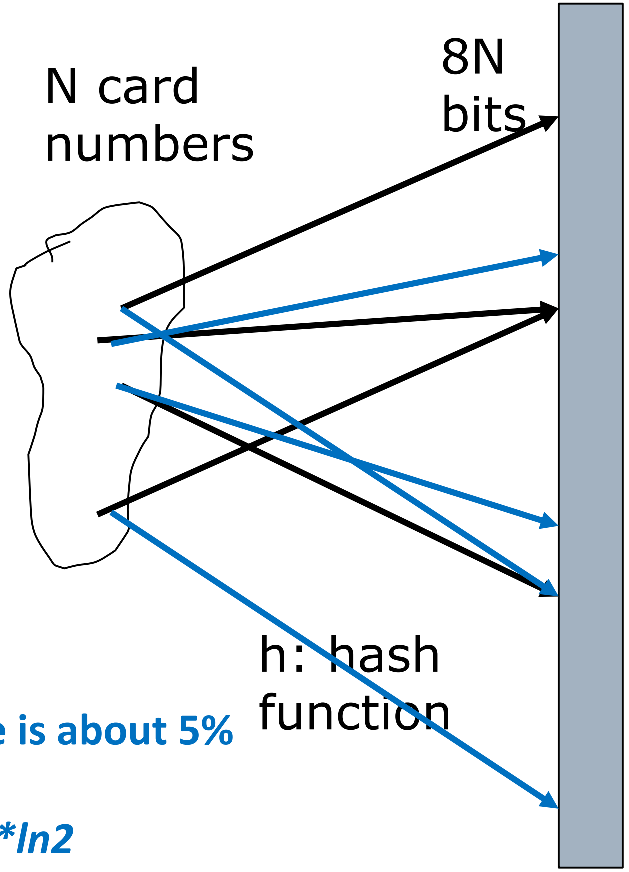
- How many bits are set to 1?
  - n=10^9 legal cards
  - N=8*10^9 buckets
  - k=2 the number of hash functions
- What is the chance that a randomly selected bit is **not set to 1**?

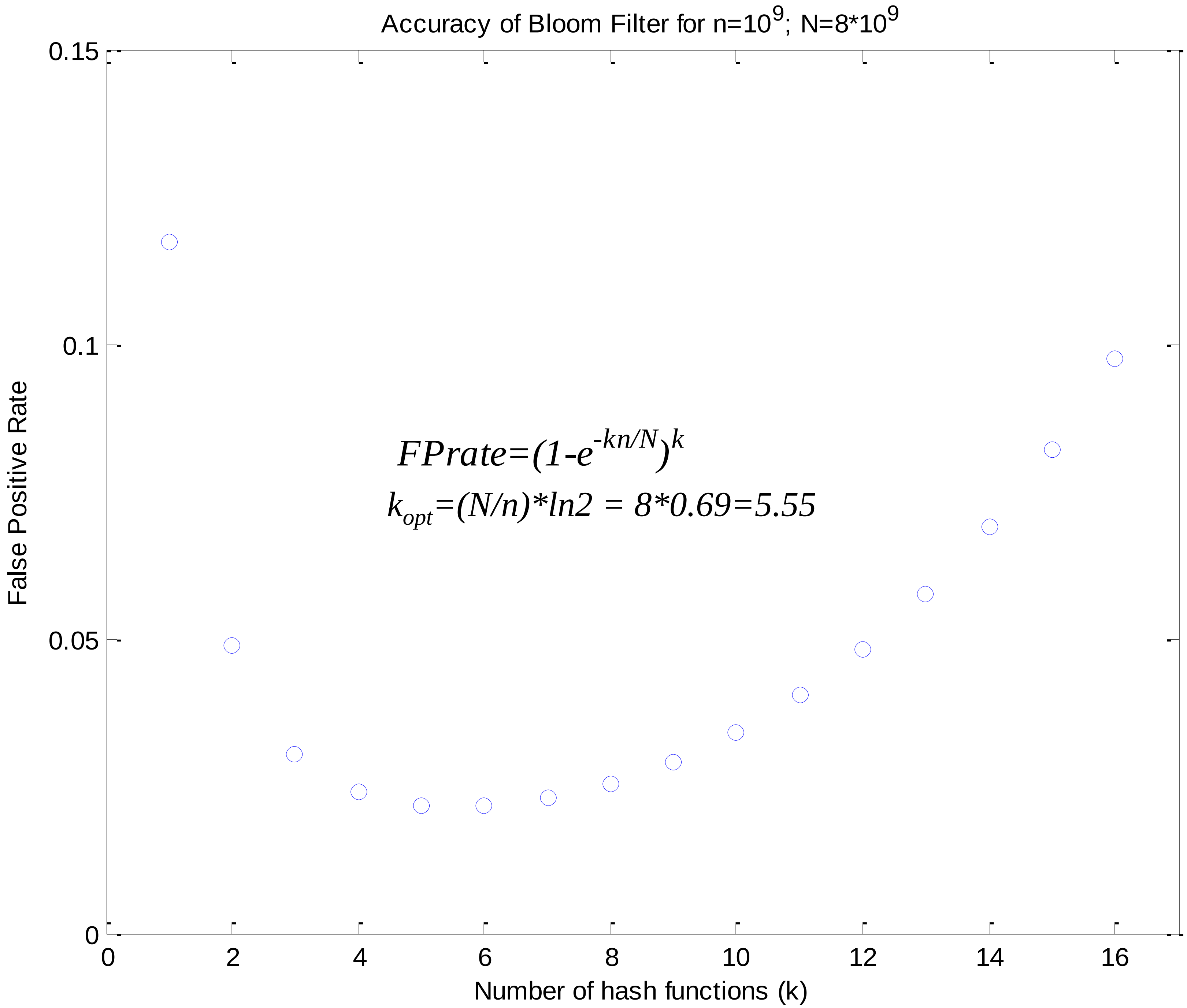
$$((N-1)/N)^{kn} = ((1-1/N)^{N})^{kn/N} \approx e^{-k\frac{n}{N}}$$

- Thus the **chance of hitting 1 by k independent hash functions** is:

$$(1-((1-1/N)^{N})^{kn/N})^{k} \approx (1-e^{-k\frac{n}{N}})^{k}$$

- k=2, n/N=1/8 => (1-exp(-1/4))**^2**=0.0493, so **the false positives rate is about 5%**
- What number of hash functions k is optimal? ***k=(#bits/#elements)*ln2***




Accuracy of Bloom Filter for n=10^9; N=8*10^9
False Positive Rate
Number of hash functions (k)
0.15
0.1
0.05
0
0
2
4
6
8
10
12
14
16
$FPrate=(1-e^{-kn/N})^k$
$k_{opt}=(N/n)*ln2 = 8*0.69=5.55$

Final Remarks on Bloom Filters

- **Cascading**: using two or more filters one after another reduces errors "exponentially fast" (e.g., 5%*5%=0.25%).
- **Inserting** new elements to a filter is **easy**.
- **Removal of elements from a filter is "almost impossible"** ***(Why?)***
- If two Bloom filters F_A and F_B represent sets A and B then the bitwise AND of F_A and F_B represents the intersection of A and B and the bitwise OR of F_A and F_B represents the union of A and B. *(Why? What about false positive rates for such constructions?)*

More applications of Bloom Filters:

- Google BigTable and Apache Cassandra use Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.
- The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result).
- The Squid Web Proxy Cache uses Bloom filters for cache digests.[10]
- Bitcoin uses Bloom filters to speed up wallet synchronization.[11][12]
- The Venti archival storage system uses Bloom filters to detect previously stored data.[13]

3. Counting Distinct Elements

- Problem: a data stream consists of elements chosen from a set of size ***n (n very big!)***.
  How to maintain the count of the number of distinct elements seen so far?
- Obvious approach: maintain the set of elements seen (costs O(n) memory!)
- *Use less memory (and accept loss of accuracy)*

Applications

- How many different URLs have we seen so far?

- How many different words are found among the Web pages being crawled at a site?
  - Unusually low or high numbers could indicate "artificial pages"

- How many different Web pages does each customer requests in a week?

- How many distinct elements in a column of a table?
  (optimization of the join operation of two tables)

- How many distinct <source, destination> pairs through a router?
  (detection of DoS attacks)

Using Small Storage

- Real Problem:
  what if we do not have space to store the complete set?

- Estimate the count in an unbiased way.

- Accept that the count may be in error, but limit the probability that the error is large.

A simple idea: MinTopK estimate

- *Hash incoming objects into doubles from the interval [0, 1] and count them shrinking the interval if needed.*
- Due to limited memory, maintain only the K biggest values (“TopK”), say, K=1000.
- Let **s** denote the minimum of our set (MinTopK)
- The number of distinct elements ≈ **K/(1-s)**
- *What about the accuracy? The number of bits?*