MapReduce III / Mining Data Streams I

Alexander Schönhuth

UNIVERSITÄT
BIELEFELD
Faculty of Technology

Bielefeld University
June 4, 2020

TODAY

Overview

- MapReduce III
  - Reducer Size
  - Replication Rate
  - Graph Model
  - Mapping Schema
  - Lower Bounds on Replication Rate
- Mining Data Streams I
  - Intro: A Data Stream Management Model
  - Sampling Data in a Stream
  - Filtering Streams: Bloom Filters
  - Counting Distinct Elements: Flajolet-Martin algorithm
  - Estimating Moments: Alon-Matias-Szegedy algorithm
  - Counting Ones in a Window: Datar-Gionis-Indyk-Motwani algorithm
  - Decaying Windows

Learning Goals: Understand these topics and get familiarized

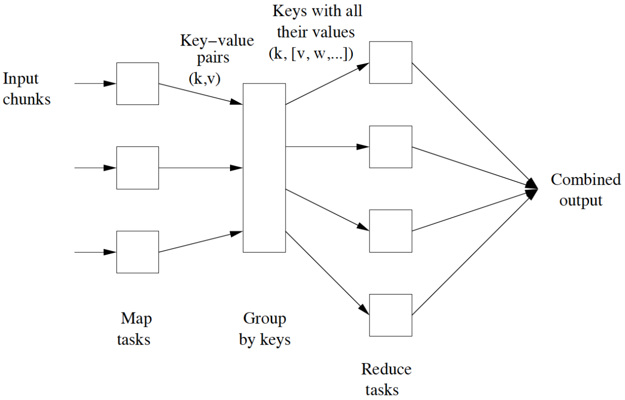
Complexity Theory for MapReduce

MapReduce: Complexity Theory

Idea

- *Reminder:* A "reducer" is the execution of a Reduce task on a single key and the associated value list
- *Important considerations:*
  - Keep communication cost low
  - Keep wall-clock time low
  - Execute each reducer in main memory
- *Intuition:*
  - The less communication, the less parallelism, so
  - the more wall-clock time
  - the more main memory needed
- *Goal:* Develop encompassing complexity theory

REDUCER SIZE: INFORMAL EXPLANATION



Reducer size: maximum length of list [v,w,...] after grouping keys

Adopted from `mmds.org`

REDUCER SIZE

DEFINITION [REDUCER SIZE]:

The *reducer size* q is the upper bound on the number of values to appear in the list of a single key.

Motivation

- Small reducer size forces to have many reducers
- Further creating many Reduce tasks implies high parallelism, hence small wall-clock time
- Sufficiently small reducer size allows to have all data in main memory

REPLICATION RATE

DEFINITION [REPLICATION RATE]:

The *replication rate* r is the number of all key-value pairs generated by Map tasks, divided by the number of inputs.

Motivating Example

- One-pass matrix multiplication algorithm:
    - Matrices involved are $n \times n$
    - *Reminder:* Key-value pairs for MN are $((i,k),(M,j,m_{ij})), j = 1, ..., n$ and $((i,k),(N,j,n_{jk})), j = 1, ..., n$
- Replication rate r is equal to n:
    - Inputs are all m_{ij} and n_{jk}
    - For each m_{ij}, one generates key-value pairs for $(i,k), k = 1, ..., n$
    - For each n_{jk}, one generates key-value pairs for $(i,k), i = 1, ..., n$
- Reducer size is $2n$: for each key (i,k) there are n values from each m_{ij} and n values from each n_{jk}

EXAMPLE: SIMILARITY JOIN

Situation

- Given large set X of elements
- Given similarity measure $s(x, y)$ for measuring similarity between $x, y \in X$
- Measure is symmetric: $s(x, y) = s(y, x)$
- *Output* of the algorithm: all pairs x, y where $s(x, y) \geq t$ for threshold t
- *Exemplary input:* 1 million images (i, P_i) where
  - i is ID of image
  - P_i is picture itself
  - Each picture is 1MB

EXAMPLE: SIMILARITY JOIN

MapReduce: Bad Idea

- Use keys (i, j) for pair of pictures $(i, P_i), (j, P_j)$
- *Map*: generates $((i, j), [P_i, P_j])$ as input for
- *Reduce*: computes $s(P_i, P_j)$ and decides whether $s(P_i, P_j) \geq t$
- Reducer size q is small: 2 MB; expected to fit in main memory
- *However,* each picture makes part of 999 999 key-value pairs, so

$$r = 999\,999$$

- Hence, number of bytes communicated from Map to Reduce is

$$10^6 \times 999\,999 \times 10^6 = 10^{18}$$

that is one exabyte

👎

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

- Group images into g groups, each of $10^6/g$ pictures
- *Map:* For each (i, P_i) generate $g-1$ key-value pairs
  - Let u be group of P_i
  - Let v be one of the other groups
  - Keys are sets $\{u, v\}$ (set, so no order!)
  - Value is (i, P_i)
  - Overall: $(\{u, v\}, (i, P_i))$ as key-value pair
- *Reduce:* Consider key $\{u, v\}$
  - Associated value list has $2 \times \frac{10^6}{g}$ values
  - Consider (i, P_i) and (j, P_j) when i, j are from different groups
  - Compute $s(P_i, P_j)$
  - Compute $s(P_i, P_j)$ for P_i, P_j from same group on processing keys $\{u, u+1\}$

EXAMPLE: SIMILARITY JOIN

MapReduce: Better Idea

- *Replication rate* is $g - 1$
  - Each input element (i, P_i) is turned into $g - 1$ key-value pairs
- *Reducer size* is $2 \times \frac{10^6}{g}$
  - Number of values on list for reducer
  - Each value is about 1 MB yields $2 \times \frac{10^{12}}{g}$ stored at Reducer node
- *Example* $g = 1000$*:*
  - Input is 2 GB, fits into main memory
  - Total number of bytes communicated: $10^6 \times 999 \times 10^6 \approx 10^{15}$
  - 1000 times less than brute-force
  - Half a million reducers: maximum parallelism at Reduce nodes
- *Computation cost* is independent of g
  - Always all-vs-all comparison of pictures
  - Computing $s(P_i, P_j)$ for all i, j

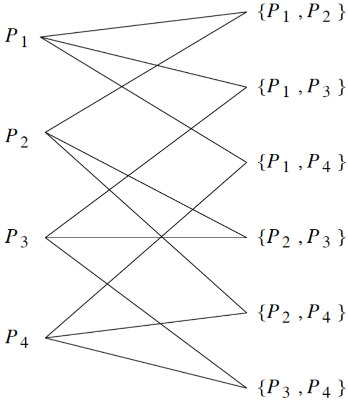
MapReduce: Graph Model

Goal: Proving lower bounds on replication rate as function of reducer size, for many problems. Therefore:

Graph Model:

- Graph describes how outputs depend on inputs
- Reducers operate independently: each output has one reducer that receives all input required to compute output
- *Model foundation:*
  - There is a set of inputs
  - There is a set of outputs
  - Outputs depends on inputs: many-to-many relationship

MAPREDUCE: GRAPH MODEL EXAMPLE



Graph for similarity join with four pictures

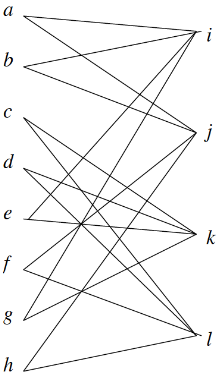
Adopted from `mmds.org`

MAPREDUCE: GRAPH MODEL MATRIX MULTIPLICATION

Graph Model Matrix Multiplication

- Multiplying $n \times n$ matrices M and N makes
    - $2n^2$ inputs $m_{ij}, n_{jk}, 1 \leq i, j, k \leq n$
    - n^2 outputs $p_{ik} := (MN)_{ik}, 1 \leq i, k \leq n$
- Each output p_{ik} needs $2n$ inputs $m_{i1}, m_{i2}, ..., m_{in}$ and $n_{1k}, n_{2k}, ..., n_{nk}$
- Each input relates to n outputs: e.g. m_{ij} to $p_{i1}, p_{i2}, ..., p_{in}$

MAPREDUCE: GRAPH MODEL MATRIX MULTIPLICATION II



$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} i & j \\ k & l \end{bmatrix}$$

Input-output relationship graph for multiplying 2x2 matrices

Adopted from `mmds.org`

MAPREDUCE: MAPPING SCHEMAS

A *mapping schema* with a given reducer size q is an assignment of inputs to reducers such that

- No reducer receives more than q inputs
- For every output, there is a reducer that receives all inputs required to generate the output

Consideration: The existence of a mapping schema for a given q distinguishes problems that can be solved in a *single* MapReduce job from those that cannot.

MAPPING SCHEMA: EXAMPLE

Consider computing similarity of p pictures, divided into g groups.

- Number of outputs: $\binom{p}{2} = \frac{p(p-1)}{2} \approx \frac{p^2}{2}$
- Reducer receives $2p/g$ inputs
  ☞ necessary reducer size is $q = 2p/g$
- Replication rate is $r = g - 1 \approx g$:

$$r = 2p/q$$

  ☞ r inversely proportional to q which is common

- In a mapping schema for reducer size q:
  - Each reducer is assigned exactly $2p/g$ inputs
  - In all cases, every output is covered by some reducer

MAPPING SCHEMAS: NOT ALL INPUTS PRESENT

Example: Natural Join $R(A, B) \bowtie S(B, C)$, where many possible tuples $R(a, b), S(b, c)$ are missing.

- Theoretically $q = |A| \cdot |C|$ (keys were $b \in B$)
- But in practice many tuples $(a, b), (b, c)$ are missing for each b, so q possibly much smaller than $|A| \cdot |C|$

Main Consideration: One can increase q because of the missing inputs, without that inputs do no longer fit into main memory in practice

MAPPING SCHEMAS: LOWER BOUNDS ON REPLICATION RATE

Technique for proving lower bounds on replication rates

1. Prove upper bound $g(q)$ on how many outputs a reducer with q inputs can cover
   ☞ This may be difficult in some cases
2. Determine total number of outputs O
3. Let there be k reducers with $q_i < q, i = 1, ..., k$ inputs
   ☞ observe that $\sum_{i=1}^{k} g(q_i)$ needs to be no less than O
4. Manipulate the inequality $\sum_{i=1}^{k} g(q_i) \geq O$ to get a lower bound on $\sum_{i=1}^{k} q_i$
5. Dividing the lower bound on $\sum_{i=1}^{k} q_i$ by number of inputs is lower bound on replication rate

LOWER BOUNDS: EXAMPLE ALL-PAIRS PROBLEM

- Recall that $r \leq 2p/q$ was upper bound on replication rate for all-pairs problem
- *Here:* Lower bound on r that is half the upper bound

LOWER BOUNDS: EXAMPLE ALL-PAIRS PROBLEM

- *Steps from slide before:*
  - Step 1: reducer with q inputs cannot cover more than $\binom{q}{2} \approx q^2/2$ outputs
  - Step 2: overall $\binom{p}{2} \approx p^2/2$ outputs must be covered
  - Step 3: So, the inequality approximately evaluates as

$$\sum_{i=1}^{k} q_i^2/2 \geq p^2/2 \qquad \Longleftrightarrow \qquad \sum_{i=1}^{k} q_i^2 \geq p^2$$

  - Step 4: From $q \geq q_i$, we obtain

$$q \sum_{i=1}^{k} q_i \geq p^2 \qquad \Longleftrightarrow \qquad \sum_{i=1}^{k} q_i \geq \frac{p^2}{q}$$

  - Step 5: Noting that $r = (\sum_{i=1}^{k} q_i)/p$, we obtain

$$r \geq \frac{p}{q}$$

which is half the size of upper bound

Mining Data Streams: Introduction

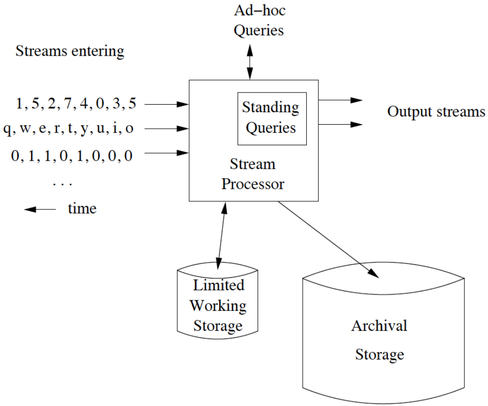
MINING DATA STREAMS: INTRODUCTION I

- *Situation:* Data arrives in a stream (or several streams)
    - Too much to be put in active storage (main memory, disk, database)
    - If not processed immediately or stored (in inaccesible archives), lost forever
- *Algorithms* involve some summarization of stream(s); e.g.
    - create useful samples of stream(s)
    - filter the stream(s)
    - focus on windows of appropriate length (last n elements)

DATA STREAMS: EXAMPLES

- Sensor data:
    - Ocean data (temperature, height): terabytes per day
    - Tracking cars (location, speed)
- Image data from satellites
- Internet/web traffic
    - Switches that route data also decide on denial of service
    - Tracking trends via analyzing clicks

DATA STREAM MANAGEMENT SYSTEM



A data stream management system

Adopted from `mmds.org`

DATA STREAM QUERIES

- *Standing queries*
  - need to be answered throughout time
  - Answers need to be updated when they change
  - Example: current or maximum ocean temperature
- *Ad-hoc queries*
  - ask immediate questions
  - *Example:* number of unique users of a web site in the last 4 weeks
  - Not all data can be stored/processed
    ☞ Only certain questions feasible
  - Need to prepare for queries
    ☞ For example, store data from sliding windows

DATA STREAM QUERIES

Issues

- Streams deliver elements rapidly: need to act quickly
- Thus, data to work on should fit in main memory
- New techniques required:

☞ Compute approximate, not exact answers

☞ Hashing is a useful technique

Sampling Elements from a Stream

SAMPLING ELEMENTS

- *Situation:*
  - Select subsample from stream to store
  - Subsample should be representative of stream as a whole
- *Running Example:*
  - Search engine processes stream of search queries
  - Stream consists of tuples (user,query,time)
  - Can store only 1/10-th of data
  - *Stream Query:* Fraction of repeated search queries?

RUNNING EXAMPLE: PITFALL

- *Running Example:*
    - *Stream Query:* Fraction of repeated search queries?

Naive and bad approach

- For each query, generate random integer from $[0, 9]$
- Keep only queries if 0 was generated
- *Scenario:* Suppose a user has issued
    - s queries one time
    - d queries two times
    - no queries more than two times
- *Correct answer* is $\frac{d}{d+s}$

RUNNING EXAMPLE: PITFALL

- *Running Example:*
  - *Stream Query:* Fraction of repeated search queries?

Naive and bad approach

- *Correct answer* is $\frac{d}{d+s}$
- But on randomly selected queries, we see that
  - Of one-time queries, $s/10$ appear to show once
  - Of two-time queries, $d/10 \times d/10$ appear to show twice
  - Of two-time queries, $d(1/10 \times 9/10) \times 2$ appear to show once
  - Resulting in *estimate*

$$\frac{0.01d}{0.1s + 0.18d} = \frac{d}{10s + 19d}$$

for unique queries, which is wrong for positive s, d

RUNNING EXAMPLE: PITFALL

- *Running Example:*
    - *Stream Query:* Fraction of repeated search queries?

Better approach

- For each user (not query!), generate random integer from $[0, 9]$
- Keep 1/10th of users, e.g. if 0 was generated
- Implement randomness by hashing users to 10 buckets
    - ☞ avoids storing for each user whether he was in or out
- For maintaining sample for a/b-th of data, use b buckets, and keep users in buckets 0 to $a - 1$

RUNNING EXAMPLE: PITFALL

Better approach

- *General Sampling Problem:* Generalize from one-valued key to arbitrary-valued keys, keep a/b-th of (multi-valued) keys by the same technique
- *Reducing sample size:* On increasing amounts of data, ratio of data used for sample to be lowered
  - When lowering is necessary, decrease a by 1, so 0 to $a - 2$ are still accepted
  - Remove all elements with keys hashing to $a - 1$

Filtering Streams

FILTERING STREAMS: MOTIVATING EXAMPLE

- *Problem:* Filter for data for which certain conditions apply
- Can be easy: data are numbers, select numbers of at most 10
- *Challenge:*
  - There is a set S that is too large to fit in main memory
  - Condition is too check whether stream elements belong to S
- *Motivating Example: Email Spam*
  - Streamed data: pairs (email address, email text)
  - Set S is one billion (10^9) *approved (no spam!) addresses*
  - Only process emails from these addresses
    ☞ need to determine whether arbitrary address belongs to them
  - *But*, addresses cannot be stored in main memory
  - *Option 1:* make use of disk accesses
  - *Option 2 (preferrable):* Devise method without disk accesses, and determines set membership right in (vast) majority of cases
- *Solution:* "Bloom Filtering"

BLOOM FILTERING: RUNNING EXAMPLE

- Assume that main memory is 1GB
- Bloom filtering: use main memory as bit array (of eight billion bits)
- Devise hash function *h* that hashes email addresses to eight billion buckets
- Hash each member of *S* (allowed email addresses) to one of the buckets
- Set bits of hashed-to buckets to 1, leave other bits 0
- About 1/8-th of bits are 1

BLOOM FILTERING: RUNNING EXAMPLE

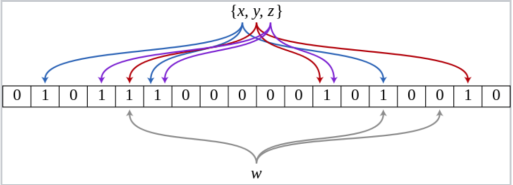
- Hash any new email address:
  - If hashed-to bit is 1, classify address as no spam
  - If hashed-to bit is 0, classify address as spam
- Each address hashed to 0 is indeed spam
- *But:* About 1/8-th of spam emails hash to 1
- So, not each address hashed to 1 is no spam
- 80% of emails are spam: filtering out 7/8-th is a big deal
- Filter cascade: filter out 7/8-th of (remaining) spam in each step

BLOOM FILTER: DEFINITION

DEFINITION [BLOOM FILTER]

A *Bloom filter* consists of

- A bit array B of n bits, initially all zero
- A set S of m key values
- Hash functions $h_1, \ldots, h_k$ hashing key values to bits of B
  - ☞ Number of buckets is n



A Bloom filter for set $S = \{x, y, z\}$ using three hash functions

From Wikipedia, by David Eppstein

BLOOM FILTER: DEFINITION

DEFINITION [BLOOM FILTER]

A *Bloom filter* consists of

- A bit array B of n bits, initially all zero
- A set S of m key values
- Hash functions $h_1, ..., h_k$ hashing key values to bits of B
  - ☞ Number of buckets is n

Bloom Filter Workflow

- *Initialization*
  - Take each key value $K \in S$
  - Set all bits $h_1(K), ..., h_k(K)$ to one
- *Testing keys:*
  - Take key K to be tested
  - Declare K to be a member of S if all $h_1(K), ..., h_k(K)$ are one

BLOOM FILTERING: ANALYSIS

- If $K \in S$, all $h_1(K), ..., h_k(K)$ are one, so K passes
- If $K \notin S$, all $h_1(K), ..., h_k(K)$ could be one, so K mistakenly passes
  ☞ *False positive!*
- *Goal:* Calculate probability of false positives
- *For that*, calculate probability that bit is zero after initialization
- *Relates to* throwing y darts at x targets, where
  - Targets are bits in array, so $x = n$
  - Darts are members in S $(= m)$ times hash functions $(= k)$, which makes $y = km$

☞ What is the probability that target is not hit by any dart?

BLOOM FILTERING: ANALYSIS

Throwing y darts at x targets:

- Probability that a given dart will not hit a given target is $(x-1)/x$
- Probability that none of the y darts will hit a given target is

$$(\frac{x-1}{x})^{y} = (1-\frac{1}{x})^{x\frac{y}{x}} \tag{1}$$

- By $(1-\epsilon)^{1/\epsilon} = 1/e$ for small ϵ, we obtain that (1) is $e^{-y/x}$
- $x = n, y = km$: probability that a bit remains 0 is $e^{-km/n}$
- Would like to have fraction of 0 bits fairly large
- If k is about n/m, then probability of a 0 is e^{-1} (about 37%)
- In general, probability of false positive is k 1 bits, which evaluates as

$$(1-e^{-\frac{km}{n}})^{k} \tag{2}$$

Counting Distinct Elements

The Flajolet-Martin Algorithm

COUNTING DISTINCT ELEMENTS: PROBLEM

- *Situation:* Stream elements chosen from universal set
- How many different elements have appeared in stream?
- Consider stream as a subset: determine cardinality (size) of subset
- *Example: Unique users of website*
    - Amazon: determine number of users from user logins
    - Google: determine number of users from search queries

COUNTING DISTINCT ELEMENTS: PROBLEM

- *Situation:* Stream elements chosen from universal set
- How many different elements have appeared in stream?
- *Obvious, but expensive:*
  - Keep stream elements in main memory
  - Store them in efficient search structure (hash table, search tree)
  - Works for sufficiently small amounts of distinct elements
- *If too many distinct elements, or too many streams:*
  - Use more machines ☞ Ok if affordable
  - Use secondary memory (disk) ☞ slow
  - *Here:* Estimate number of distinct elements using much less main memory than needed for storing all distinct elements
  - The *Flajolet-Martin algorithm* does this job

THE FLAJOLET-MARTIN ALGORITHM

- *Central idea:* Hash elements to bit strings of sufficient length
  - For example, to hash URL’s, 64-bit strings are sufficiently long
- *Intuition:*
  - The more different elements, the more different bit strings
  - The more different bit strings, the more “unusual” bit strings
  - Unusual here = bit string ends in many zeroes

DEFINITION [TAIL LENGTH]

- Let h be the hash function that hashes stream elements a to bit strings $h(a)$
- The *tail length* of $h(a)$ is the number of zeroes in which it ends

THE FLAJOLET-MARTIN ALGORITHM

DEFINITION [TAIL LENGTH]

- Let h be the hash function that hashes stream elements a to bit strings $h(a)$
- The *tail length* of $h(a)$ is the number of zeroes in which it ends

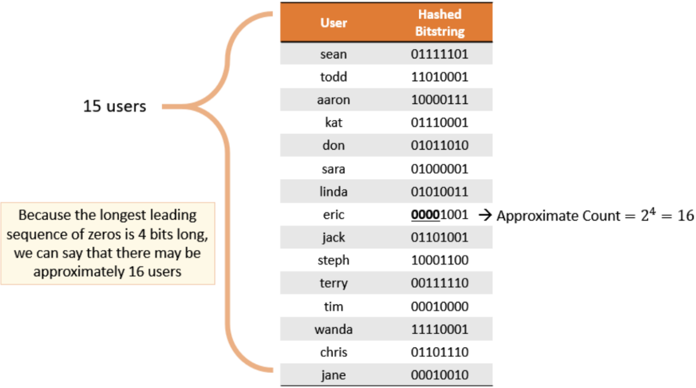
FLAJOLET ALGORITHM

- Let A be the set of stream elements
- Let

$$R := \max_{a \in A} h(a) \tag{3}$$

  be the maximum tail length observed among stream elements
- *Estimate* 2^R for the number of distinct elements in the stream

FLAJOLET-MARTIN ALGORITHM: EXAMPLE



15 users

User	Hashed Bitstring
sean	01111101
todd	11010001
aaron	10000111
kat	01110001
don	01011010
sara	01000001
linda	01010011
eric	**0000**1001
jack	01101001
steph	10001100
terry	00111110
tim	00010000
wanda	11110001
chris	01101110
jane	00010010

→ Approximate Count = $2^4 = 16$

Because the longest leading sequence of zeros is 4 bits long, we can say that there may be approximately 16 users

Hashing user names to 8-bit strings

From `towardsdatascience.com`

FLAJOLET-MARTIN ALGORITHM: EXPLANATION

- Probability that bit string $h(a)$ ends in r zeroes is 2^{-r}
- Probability that none of m distinct elements has tail length at least r is

$$(1-2^{-r})^m = ((1-2^{-r})^{2^r})^{m2^{-r}} \overset{(1-\epsilon)^{1/\epsilon} \approx 1/e}{=} e^{-m2^{-r}} \tag{4}$$

- Let $P_{m,r} := 1-(1-2^{-r})^m \approx 1-e^{-m2^{-r}}$ be the probability that for m stream elements, the maximum tail length R observed is at least r.
- Conclude:
  - For $m >> 2^r$, it holds that $P_{m,r}$ approaches 1
  - For $m << 2^r$, it holds that $P_{m,r}$ approaches 0
  - So, 2^R is unlikely to be much larger or much smaller than m

FLAJOLET-MARTIN ALGORITHM: COMBINING ESTIMATES

- *Idea:* Use several hash functions $h_k, k = 1, ..., K$
- Combine their estimates $X_k, k = 1, ..., K$
- *Pitfall 1: Averaging*
  - Let p_r be the probability that the maximum tail length of h_k is r
  - One can compute that

$$p_r \geq 2p_{r-1} \geq ... \geq 2^{-r+1}p_1 \geq 2^{-r}p_0$$

  - So $E(X_k)$, the expected value of X_k computes as

$$E(X_k) = \sum_{r \geq 0} p_r 2^r \geq p_o \sum_{r \geq 0} 2^{-r}2^r = p_0 \sum_{r \geq 0} 1 = \infty$$

  - Therefore $\frac{1}{K}\sum_{k=1}^{K} E(X_k)$ the expected value of the average of the X_k turns out to be infinite as well
  - *Conclusion:* Overestimates spoil averaging

FLAJOLET-MARTIN ALGORITHM: COMBINING ESTIMATES

- *Idea:* Use several hash functions $h_k, k = 1, ..., K$
- Combine their estimates $X_k, k = 1, ..., K$
- *Pitfall 2: Computing Medians*
  - The median is always a power of two
    ☞ makes only very limited sense
- *Solution:*
  - Group the hash functions into small groups and take averages within groups
  - Estimate m as median of group averages
  - Groups should be of size $C \log_2 m$ for some small C
- *Space Requirements:* Need to store only value of X_k, requiring little space as a maximum

MATERIALS / OUTLOOK

- See *Mining of Massive Datasets*: section 2.6; sections 4.1–4.4
- As usual, see `http://www.mmds.org/` in general for further resources
- For deepening your understanding, consider voluntary *homework*: read 2.6.7 and try to make sense of this
- Next lecture: “Mining Data Streams II / PageRank I”
    - See *Mining of Massive Datasets* 4.5–4.7; 5.1–5.2