Lecture 10: Streaming (Count Min Sketch and HyperLogLog Counting)

Sam McCauley

October 8, 2024

Williams College

Admin

- Questions about Homework 3?
- No leaderboard for Homework 3 or 4 (will come back for Homework 5)
  - Interesting things to say about optimizing (say) filters, but for our use case does not noticeably impact running time
- Mountain day Friday?
- Homework 4 will be released around then
- Homework 4 is not too long, especially for the code; a good time to catch up!

Really Large Data (as of 2021)

- Netflix sends (so far as I can tell) about 500TB per minute on average to its customers
- Google's search index is over 100,000,000 GB
- Brazil Internet Exchange processes 7 trillion bits every second

Really Large Data

- Modern companies deal with extremely large data

Really Large Data

- Modern companies deal with extremely large data
- Can’t even store all of it sometimes!

Really Large Data

- Modern companies deal with extremely large data
- Can’t even store all of it sometimes!
- If is possible to store, can be very difficult to access particular pieces

A Shift in Focus (Streaming)

- Up until now: nice self-contained instances; might fit in L3 cache; might fit in RAM

A Shift in Focus (Streaming)

- Up until now: nice self-contained instances; might fit in L3 cache; might fit in RAM

- In some situations: the data is *too big* and you can't hope to do that

A Shift in Focus (Streaming)

- Up until now: nice self-contained instances; might fit in L3 cache; might fit in RAM
- In some situations: the data is *too big* and you can't hope to do that
- The data is like a stream that's constantly rushing past

A Shift in Focus (Streaming)

- Up until now: nice self-contained instances; might fit in L3 cache; might fit in RAM
- In some situations: the data is *too big* and you can't hope to do that
- The data is like a stream that's constantly rushing past
- All you can do is sample pieces as they pass by

Streaming Model

- You receive a *stream* of N items one by one

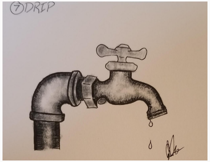
Streaming Model

- You receive a *stream* of N items one by one
- Stream is incredibly long; you can't store all of the items

Streaming Model

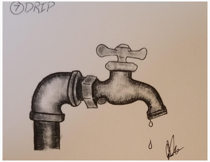
- You receive a *stream* of N items one by one
- Stream is incredibly long; you can't store all of the items
- Can't move forward or backward either; just come in one at a time

Streaming Model



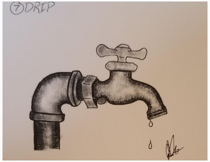
- Normally you're used to getting your data all at once, with the ability to store all of it, and access random pieces whenever you want.

Streaming Model



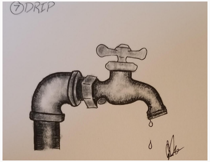
- Normally you're used to getting your data all at once, with the ability to store all of it, and access random pieces whenever you want.
- Now, a worst-case adversary is feeding you tiny pieces of information one-by-one, in whatever order they want

Streaming Model



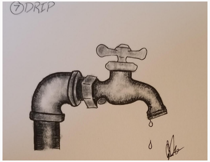
- Normally you're used to getting your data all at once, with the ability to store all of it, and access random pieces whenever you want.
- Now, a worst-case adversary is feeding you tiny pieces of information one-by-one, in whatever order they want
- You can only store $O(\log N)$ bytes of space, or maybe even $O(1)$

Streaming Model



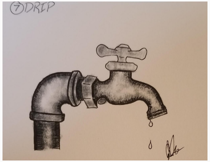
- Normally you're used to getting your data all at once, with the ability to store all of it, and access random pieces whenever you want.
- Now, a worst-case adversary is feeding you tiny pieces of information one-by-one, in whatever order they want
- You can only store $O(\log N)$ bytes of space, or maybe even $O(1)$
- What can we do in this situation?

Streaming Model



- Normally you're used to getting your data all at once, with the ability to store all of it, and access random pieces whenever you want.
- Now, a worst-case adversary is feeding you tiny pieces of information one-by-one, in whatever order they want
- You can only store $O(\log N)$ bytes of space, or maybe even $O(1)$
- What can we do in this situation?
- Note: very active area of research

Streaming Model



- Normally you're used to getting your data all at once, with the ability to store all of it, and access random pieces whenever you want.
- Now, a worst-case adversary is feeding you tiny pieces of information one-by-one, in whatever order they want
- You can only store $O(\log N)$ bytes of space, or maybe even $O(1)$
- What can we do in this situation?
- Note: very active area of research
- Today we'll look at two classic results

What We Really Want

- Much more extreme “compression” than a filter

What We Really Want

- Much more extreme “compression” than a filter
- (Filter used a constant number of bits per item; we can’t afford that)

What We Really Want

- Much more extreme "compression" than a filter
- (Filter used a constant number of bits per item; we can't afford that)
- Today: two data structures

What We Really Want

- Much more extreme “compression” than a filter
- (Filter used a constant number of bits per item; we can’t afford that)
- Today: two data structures
  - Count-min sketch: More aggressive than a filter. Good guarantees for counting how many times a given element occurred in a stream.

What We Really Want

- Much more extreme “compression” than a filter
- (Filter used a constant number of bits per item; we can’t afford that)
- Today: two data structures
  - Count-min sketch: More aggressive than a filter. Good guarantees for counting how many times a given element occurred in a stream.
  - HyperLogLog: Only uses a few bytes. Estimates how many unique items appeared in the stream.

When to Use Streaming Algorithms?

- Data streams: network traffic, user inputs, telephone traffic, etc.

When to Use Streaming Algorithms?

- Data streams: network traffic, user inputs, telephone traffic, etc.
- Cache-efficiency! Streaming algorithms only require you to *scan* the data once.

When to Use Streaming Algorithms?

- Data streams: network traffic, user inputs, telephone traffic, etc.
- Cache-efficiency! Streaming algorithms only require you to *scan* the data once.
  - N/B cache misses

Actual Applications

- DDOS attack: keep track of IP addresses that appear too often

Actual Applications

- DDOS attack: keep track of IP addresses that appear too often
- Keep track of popular passwords

Actual Applications

- DDOS attack: keep track of IP addresses that appear too often
- Keep track of popular passwords
- Google uses an improved HyperLogLog to speed up searches

Actual Applications

- DDOS attack: keep track of IP addresses that appear too often
- Keep track of popular passwords
- Google uses an improved HyperLogLog to speed up searches
- Reddit uses HyperLogLog to estimate views of a post

Actual Applications

- DDOS attack: keep track of IP addresses that appear too often
- Keep track of popular passwords
- Google uses an improved HyperLogLog to speed up searches
- Reddit uses HyperLogLog to estimate views of a post
- Facebook uses HyperLogLog to estimate number of unique visitors to site.

HyperLogLog at Facebook

“Doing this with a traditional SQL query on a data set as massive as the ones we use at Facebook would take days and terabytes of memory... With HLL, we can perform the same calculation in 12 hours with less than 1 MB of memory.”

Count-Min Sketch

Count-Min Sketch

Goal:

- Maintain a data structure on a *stream* of items

Count-Min Sketch

Goal:

- Maintain a data structure on a *stream* of items
    - See the items one at a time; you have no control over how they are given to you

Count-Min Sketch

Goal:

- Maintain a data structure on a *stream* of items
  - See the items one at a time; you have no control over how they are given to you
  - Want to be *extremely* space efficient

Count-Min Sketch

Goal:

- Maintain a data structure on a *stream* of items
  - See the items one at a time; you have no control over how they are given to you
  - Want to be *extremely* space efficient
- At any time, estimate how frequently a given item appeared

Example

You see the following items one by one:

adhesive

Example

You see the following items one by one:

flawless

Example

You see the following items one by one:

closed

Example

You see the following items one by one:

adhesive

Example

You see the following items one by one:

describe

Example

You see the following items one by one:

closed

Example

You see the following items one by one:

sea

Example

You see the following items one by one:

illustrious

Example

You see the following items one by one:

describe

Example

You see the following items one by one:

describe

Example

You see the following items one by one:

flawless

Example

You see the following items one by one:

street

Example

You see the following items one by one:

closed

Example

You see the following items one by one:

describe

Example

- Now, answer questions of the form: how many times did some item x_i occur in the stream?

Example

- Now, answer questions of the form: how many times did some item x_i occur in the stream?
- Example: how many times did adhesive appear? How about closed?

Example

- Now, answer questions of the form: how many times did some item x_i occur in the stream?

- Example: how many times did adhesive appear? How about closed?
  - (2 times and 3 times respectively)

Formally

- See a stream of elements $x_1, \ldots x_N$, each from a universe U[1]

[1] Like in the last lecture, this is just a requirement to make sure that we can hash them.

Formally

- See a stream of elements $x_1, \ldots x_N$, each from a universe U[1]
- For some element $q \in U$, estimate how many i exist with $x_i = q$?

[1] Like in the last lecture, this is just a requirement to make sure that we can hash them.

Formally

- See a stream of elements $x_1, \ldots x_N$, each from a universe U[1]
- For some element $q \in U$, estimate how many i exist with $x_i = q$?
- Today: pretty decent guess using $\left\lceil \frac{e}{\varepsilon} \right\rceil \lceil \ln(1/\delta) \rceil \lceil \log_2 N \rceil$ bits of space

[1] Like in the last lecture, this is just a requirement to make sure that we can hash them.

Formally

- See a stream of elements $x_1, \ldots x_N$, each from a universe U[1]
- For some element $q \in U$, estimate how many i exist with $x_i = q$?
- Today: pretty decent guess using $\lceil \frac{e}{\varepsilon} \rceil \lceil \ln(1/\delta) \rceil \lceil \log_2 N \rceil$ bits of space
    - ε and δ are parameters we can use to adjust the error

[1] Like in the last lecture, this is just a requirement to make sure that we can hash them.

Formally

- See a stream of elements $x_1, \ldots x_N$, each from a universe U[1]
- For some element $q \in U$, estimate how many i exist with $x_i = q$?
- Today: pretty decent guess using $\lceil \frac{e}{\varepsilon} \rceil \lceil \ln(1/\delta) \rceil \lceil \log_2 N \rceil$ bits of space
  - ε and δ are parameters we can use to adjust the error
  - Don't depend on N, or $|U|$

[1] Like in the last lecture, this is just a requirement to make sure that we can hash them.

How would you solve this problem with what you know right now?

- Let’s come up with a *space-inefficient* solution

How would you solve this problem with what you know right now?

- Let's come up with a *space-inefficient* solution
- Keep a hash table with all elements

How would you solve this problem with what you know right now?

- Let's come up with a *space-inefficient* solution
- Keep a hash table with all elements
- Increment a counter each time you see an element

How would you solve this problem with what you know right now?

- Let's come up with a *space-inefficient* solution
- Keep a hash table with all elements
- Increment a counter each time you see an element
- $O(N)$ space, $O(1)$ time per query

How would you solve this problem with what you know right now?

- Let's come up with a *space-inefficient* solution
- Keep a hash table with all elements
- Increment a counter each time you see an element
- $O(N)$ space, $O(1)$ time per query
- Pretty efficient! But we want way way less space.

Sketching: A first attempt

- Randomly sampling:

Sketching: A first attempt

- Randomly sampling:
  - Keep $N/100$ slots
  - For each item, with probability $1/100$, use the approach above

Sketching: A first attempt

- Randomly sampling:
  - Keep $N/100$ slots
  - For each item, with probability $1/100$, use the approach above
- If an item appears k times in the stream, we record it $k/100$ times in expectation.

Sketching: A first attempt

- If an item appears k times in the stream, we see it $k/100$ times in expectation.
- So, if we wrote an item down w times, we can estimate that it probably occurred $100w$ times in the stream.

Sketching: A first attempt

What are some downsides to this approach?

Sketching: A first attempt

What are some downsides to this approach?

- It's pretty loose. If our counter is just one off, that changes our guess by $+100$

Sketching: A first attempt

What are some downsides to this approach?

- It's pretty loose. If our counter is just one off, that changes our guess by $+100$
- Could have a fairly frequent item that we never write down.
- Can't guarantee much about our estimate

Second attempt: hash counts

- Maintain a hash table A with $1/\varepsilon$ entries, each of at least $\lceil \log N \rceil$ bits
    - Has enough room to store a number in $\{0, \ldots, N-1\}$.

Second attempt: hash counts

- Maintain a hash table *A* with $1/\varepsilon$ entries, each of at least $\lceil \log N \rceil$ bits
  - Has enough room to store a number in $\{0, \ldots, N-1\}$.
- Hash function *h* for *A*

Second attempt: hash counts

- Maintain a hash table A with $1/\varepsilon$ entries, each of at least $\lceil \log N \rceil$ bits
  - Has enough room to store a number in $\{0, \ldots, N-1\}$.
- Hash function h for A
- When we see an item x_i:

Second attempt: hash counts

- Maintain a hash table A with $1/\varepsilon$ entries, each of at least $\lceil \log N \rceil$ bits
  - Has enough room to store a number in $\{0, \ldots, N-1\}$.
- Hash function h for A
- When we see an item x_i:
  - Increment $A[h(x_i)]$

Second attempt: hash counts

- Maintain a hash table A with $1/\varepsilon$ entries, each of at least $\lceil \log N \rceil$ bits
  - Has enough room to store a number in $\{0, \ldots, N-1\}$.
- Hash function h for A
- When we see an item x_i:
  - Increment $A[h(x_i)]$

Counters of length $\lceil \log N \rceil$ so don't overflow

Second attempt: hash counts

- Maintain a hash table A with $1/\varepsilon$ entries, each of at least $\lceil \log N \rceil$ bits
  - Has enough room to store a number in $\{0, \ldots, N-1\}$.
- Hash function h for A
- When we see an item x_i:
  - Increment $A[h(x_i)]$
- How can we query?

Second attempt: hash counts

How can we query q?

Second attempt: hash counts

How can we query q?

- Return $A[h(q)]$

Second attempt: hash counts

How can we query q?

- Return $A[h(q)]$
- What guarantees does this give?

Second attempt: hash counts

How can we query q?

- Return $A[h(q)]$
- What guarantees does this give?
  - Always *overestimates* the number of occurrences

Second attempt: hash counts

How can we query q?

- Return $A[h(q)]$
- What guarantees does this give?
  - Always *overestimates* the number of occurrences

Since we always increase this counter when we see $x_i = q$

Second attempt: hash counts

How can we query q?

- Return $A[h(q)]$
- What guarantees does this give?
  - Always *overestimates* the number of occurrences

But, also increase it when $h(x_i) = h(q)$, but $x_i \neq q$

Second attempt: hash counts

How can we query q?

- Return $A[h(q)]$
- What guarantees does this give?
  - Always *overestimates* the number of occurrences
  - How much does it overestimate by?

Second attempt: hash counts

How can we query q?

- Return $A[h(q)]$
- What guarantees does this give?
  - Always *overestimates* the number of occurrences
  - How much does it overestimate by?
  - Each of N items hashes to same slot with probability ε, so $N\varepsilon$ in expectation

Second attempt: hash counts (Analysis)

Expectation is not that great!

Second attempt: hash counts (Analysis)

Expectation is not that great!

- Let's say we have only two items; A appears 100 times and B appears 900

Second attempt: hash counts (Analysis)

Expectation is not that great!

- Let's say we have only two items; A appears 100 times and B appears 900
- What are the possibilities for what happens when we query A?

Second attempt: hash counts (Analysis)

Expectation is not that great!

- Let's say we have only two items; A appears 100 times and B appears 900
- What are the possibilities for what happens when we query A?
- With probability $1 - \varepsilon$ we get 100; with probability ε we get 1000

What do we really want?

- To guarantee a high-quality answer, we want to say that the solution is *likely* to be close to correct.
    - We want concentration bounds!

What do we really want?

- To guarantee a high-quality answer, we want to say that the solution is *likely* to be close to correct.
  - We want concentration bounds!
- How can you increase the reliability of a random process?

What do we really want?

- To guarantee a high-quality answer, we want to say that the solution is *likely* to be close to correct.
  - We want concentration bounds!
- How can you increase the reliability of a random process?
- For example, let's say we're rolling a die. We want to be sure we see a 6 at least once. How can we do that?

What do we really want?

- To guarantee a high-quality answer, we want to say that the solution is *likely* to be close to correct.
  - We want concentration bounds!
- How can you increase the reliability of a random process?
- For example, let's say we're rolling a die. We want to be sure we see a 6 at least once. How can we do that?
- Of course: roll the die many times!

Repetitions

- Rather than having one hash table A, let's have a two-dimensional hash table T

Repetitions

- Rather than having one hash table A, let's have a two-dimensional hash table T
- T has $\lceil \ln(1/\delta) \rceil$ rows

Repetitions

- Rather than having one hash table A, let's have a two-dimensional hash table T

We'll come back to δ later.

- T has $\lceil \ln(1/\delta) \rceil$ rows

Repetitions

- Rather than having one hash table A, let's have a two-dimensional hash table T
- T has $\lceil \ln(1/\delta) \rceil$ rows
- Each row consists of $\lceil e/\varepsilon \rceil$ slots

Repetitions

- Rather than having one hash table A, let's have a two-dimensional hash table T
- T has $\lceil \ln(1/\delta) \rceil$ rows
- Each row consists of $\lceil e/\varepsilon \rceil$ slots

The e is important for the analysis.

Repetitions

- Rather than having one hash table A, let's have a two-dimensional hash table T
- T has $\lceil \ln(1/\delta) \rceil$ rows
- Each row consists of $\lceil e/\varepsilon \rceil$ slots
- Different hash function for each row

Inserts

To insert x_i:

- For $j = 0 \ldots \lceil \ln(1/\delta) \rceil - 1$:

Inserts

To insert x_i:

- For $j = 0 \ldots \lceil \ln(1/\delta) \rceil - 1$:
  - Increment $T[j][h_j(x_i)]$

Inserts

To insert x_i:

- For $j = 0 \ldots \lceil \ln(1/\delta) \rceil - 1$:
    - Increment $T[j][h_j(x_i)]$

We now have $\lceil \ln(1/\delta) \rceil$ counters for each item. How can we query?

Queries

Each entry is an *overestimate*.

Queries

Each entry is an *overestimate*.

- Find $\min_j T[j][h_j(x_i)]$.

Count-Min Sketch

- Table T with $\lceil \ln(1/\delta) \rceil$ rows, each with $\lceil e/\varepsilon \rceil$ columns. Cells of size $\lceil \log N \rceil$

Count-Min Sketch

- Table T with $\lceil \ln(1/\delta) \rceil$ rows, each with $\lceil e/\varepsilon \rceil$ columns. Cells of size $\lceil \log N \rceil$
- $\lceil \ln(1/\delta) \rceil$ hash functions; one for each row

Count-Min Sketch

- Table T with $\lceil \ln(1/\delta) \rceil$ rows, each with $\lceil e/\varepsilon \rceil$ columns. Cells of size $\lceil \log N \rceil$
- $\lceil \ln(1/\delta) \rceil$ hash functions; one for each row
- To insert x: increment $T[j][h_j(x)]$ for all $j = 0, \ldots \lceil \ln(1/\delta) \rceil - 1$

Count-Min Sketch

- Table T with $\lceil \ln(1/\delta) \rceil$ rows, each with $\lceil e/\varepsilon \rceil$ columns. Cells of size $\lceil \log N \rceil$
- $\lceil \ln(1/\delta) \rceil$ hash functions; one for each row
- To insert x: increment $T[j][h_j(x)]$ for all $j = 0, \ldots \lceil \ln(1/\delta) \rceil - 1$
- To query q: return $\min_{j \in \{0, \ldots, \lceil \ln(1/\delta) \rceil - 1\}} T[j][h_j(q)]$

Example Insert

x

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

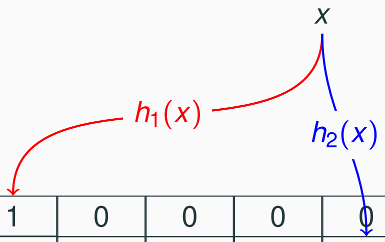
Example Insert

x

$h_1(x)$

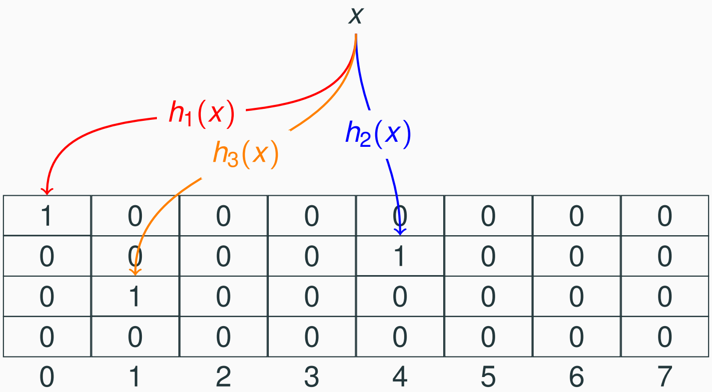
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

Example Insert

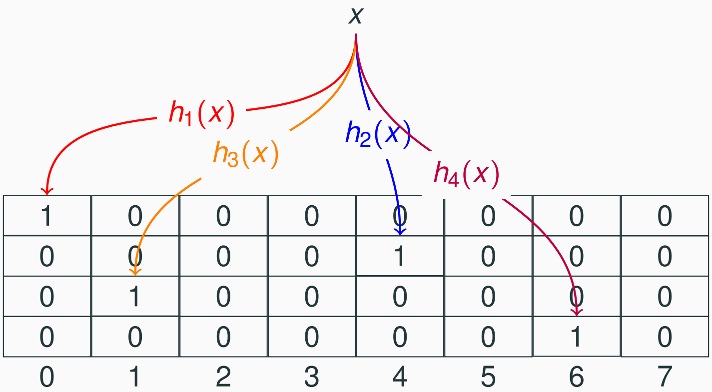


1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7

Example Insert



Example Insert



Example Insert

y

1	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0
0	1	2	3	4	5	6	7

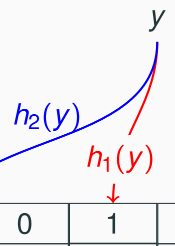
Example Insert

y

$h_1(y)$

1	0	0	1	0	0	0	0
0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0
0	1	2	3	4	5	6	7

Example Insert



1	0	0	1	0	0	0	0
0	1	0	0	1	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0
0	1	2	3	4	5	6	7

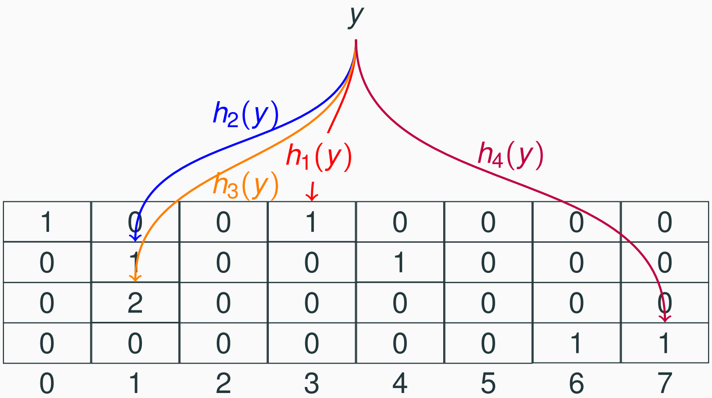
Example Insert

y

$h_2(y)$ $h_1(y)$ $h_3(y)$

1	0	0	1	0	0	0	0
0	1	0	0	1	0	0	0
0	2	0	0	0	0	0	0
0	0	0	0	0	0	1	0
0	1	2	3	4	5	6	7

Example Insert



1	0	0	1	0	0	0	0
0	1	0	0	1	0	0	0
0	2	0	0	0	0	0	0
0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7

Example Query

q

28	10	78	9	26	69	39	28
85	40	52	70	11	84	65	99
56	82	34	75	99	35	14	55
10	20	17	80	92	89	71	13
0	1	2	3	4	5	6	7

Example Query

q

$h_1(q)$

28	10	78	9	26	69	39	28
85	40	52	70	11	84	65	99
56	82	34	75	99	35	14	55
10	20	17	80	92	89	71	13
0	1	2	3	4	5	6	7

Example Query

q

$h_2(q)$ $h_1(q)$

28	10	78	9	26	69	39	28
85	40	52	70	11	84	65	99
56	82	34	75	99	35	14	55
10	20	17	80	92	89	71	13
0	1	2	3	4	5	6	7

Example Query

q

$h_2(q)$ $h_3(q)$ $h_1(q)$

0	1	2	3	4	5	6	7
28	10	78	9	26	69	39	28
85	40	52	70	11	84	65	99
56	82	34	75	99	35	14	55
10	20	17	80	92	89	71	13

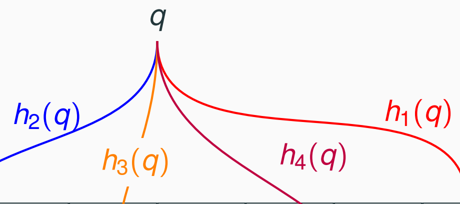
Example Query

q

$h_2(q)$ $h_3(q)$ $h_4(q)$ $h_1(q)$

28	10	78	9	26	69	39	28
85	40	52	70	11	84	65	99
56	82	34	75	99	35	14	55
10	20	17	80	92	89	71	13
0	1	2	3	4	5	6	7

Example Query



28	10	78	9	26	69	39	28
85	40	52	70	11	84	65	99
56	82	34	75	99	35	14	55
10	20	17	80	92	89	71	13
0	1	2	3	4	5	6	7

The estimated number of occurrences for q is 28.

Count-Min Sketch Guarantee: Lower bound

- On query q, let's say the filter returns that there were o_q occurrences
  - So $o_q = \min_j T[j][h_j(q)]$

Count-Min Sketch Guarantee: Lower bound

- On query q, let's say the filter returns that there were o_q occurrences
  - So $o_q = \min_j T[j][h_j(q)]$

- In reality, the correct answer is $\widehat{o_q}$ occurrences

Count-Min Sketch Guarantee: Lower bound

- On query q, let's say the filter returns that there were o_q occurrences
  - So $o_q = \min_j T[j][h_j(q)]$
- In reality, the correct answer is $\widehat{o_q}$ occurrences
- First: always have $\widehat{o_q} \leq o_q$.

Count-Min Sketch Guarantee: Upper bound

- On query q, let's say the filter returns that there were o_q occurrences; correct answer is $\widehat{o_q}$.

Count-Min Sketch Guarantee: Upper bound

- On query q, let's say the filter returns that there were o_q occurrences; correct answer is $\widehat{o_q}$.
- We know that for any j, $\mathrm{E}\left[T[j][h_j(q)]\right] \leq \widehat{o_q} + \frac{\varepsilon N}{e}$

Count-Min Sketch Guarantee: Upper bound

- On query q, let's say the filter returns that there were o_q occurrences; correct answer is $\widehat{o_q}$.
- We know that for any j, $\mathrm{E}\left[T[j][h_j(q)]\right] \leq \widehat{o_q} + \frac{\varepsilon N}{e}$
- That is to say: guess is off by $\frac{\varepsilon N}{e}$ in expectation

Count-Min Sketch Guarantee: Upper bound

- On query q, let's say the filter returns that there were o_q occurrences; correct answer is $\widehat{o_q}$.
- We know that for any j, $\mathrm{E}\left[T[j][h_j(q)]\right] \leq \widehat{o_q} + \frac{\varepsilon N}{e}$
- That is to say: guess is off by $\frac{\varepsilon N}{e}$ in expectation
- On Assignment 4, you'll prove that for any positive random variable X, $\Pr[X \geq e \cdot \mathrm{E}[X]] \leq 1/e$

Count-Min Sketch Guarantee: Upper bound

- On query q, let's say the filter returns that there were o_q occurrences; correct answer is $\widehat{o_q}$.
- We know that for any j, $\mathrm{E}\left[T[j][h_j(q)]\right] \leq \widehat{o_q} + \frac{\varepsilon N}{e}$
- That is to say: guess is off by $\frac{\varepsilon N}{e}$ in expectation
- On Assignment 4, you'll prove that for any positive random variable X, $\Pr[X \geq e \cdot \mathrm{E}[X]] \leq 1/e$
- So the probability that $T[j][h_j(q)] \geq \widehat{o_q} + \varepsilon N$ is at most $1/e$

Count-Min Sketch Guarantee: Upper bound

- For each row j, the probability that $T[j][h_j(q)] \geq \widehat{o_q} + \varepsilon N$ is at most $1/e$

Count-Min Sketch Guarantee: Upper bound

- For each row j, the probability that $T[j][h_j(q)] \geq \widehat{o_q} + \varepsilon N$ is at most $1/e$
- Are the rows independent?

Count-Min Sketch Guarantee: Upper bound

- For each row j, the probability that $T[j][h_j(q)] \geq \widehat{o_q} + \varepsilon N$ is at most $1/e$
- Are the rows independent?
  - Yes. (For each row, we select a new hash and start over)

Count-Min Sketch Guarantee: Upper bound

- For each row j, the probability that $T[j][h_j(q)] \geq \widehat{o_q} + \varepsilon N$ is at most $1/e$
- Are the rows independent?
  - Yes. (For each row, we select a new hash and start over)
- What is $\Pr\left[\min_j T[j][h_j(q)]\right] \geq \widehat{o_q} + \varepsilon N$?

Count-Min Sketch Guarantee: Upper bound

- For each row j, the probability that $T[j][h_j(q)] \geq \widehat{o_q} + \varepsilon N$ is at most $1/e$
- Are the rows independent?
  - Yes. (For each row, we select a new hash and start over)
- What is $\Pr\left[\min_j T[j][h_j(q)]\right] \geq \widehat{o_q} + \varepsilon N$?
- Only fails if cell is too big in *every* row! Occurs with probability

Count-Min Sketch Guarantee: Upper bound

- For each row j, the probability that $T[j][h_j(q)] \geq \widehat{o_q} + \varepsilon N$ is at most $1/e$
- Are the rows independent?
  - Yes. (For each row, we select a new hash and start over)
- What is $\Pr\left[\min_j T[j][h_j(q)]\right] \geq \widehat{o_q} + \varepsilon N$?
- Only fails if cell is too big in *every* row! Occurs with probability

$$\left(\frac{1}{e}\right)^{\#\text{ rows}} = \left(\frac{1}{e}\right)^{\lceil \ln 1/\delta \rceil} \leq \delta$$

Count-Min Sketch Bounds

- $\left\lceil \frac{e}{\varepsilon} \right\rceil \left\lceil \ln \frac{1}{\delta} \right\rceil \lceil \log_2 N \rceil$ bits of space

- For any query q, if the filter returns o_q and the actual number of occurrences is $\widehat{o_q}$, then with probability $1 - \delta$:

$$\widehat{o_q} \leq o_q \leq \widehat{o_q} + \varepsilon N.$$

Count-Min Sketch

- Small sketch (size based on error rate)

Count-Min Sketch

- Small sketch (size based on error rate)
- Always overestimates count

Count-Min Sketch

- Small sketch (size based on error rate)
- Always overestimates count
- Bound on overestimation is based on stream length

Parameters in Assignment CMS

- 300 entries in each row, 4 rows

Parameters in Assignment CMS

- 300 entries in each row, 4 rows
- 32-bit counters (a little wasteful!)

Parameters in Assignment CMS

- 300 entries in each row, 4 rows
- 32-bit counters (a little wasteful!)
- 7.3MB of data summarized in 4.8KB

Parameters in Assignment CMS

- 300 entries in each row, 4 rows
- 32-bit counters (a little wasteful!)
- 7.3MB of data summarized in 4.8KB
- Really accurate still: in 1.2 million word stream, can estimate num occurrences of each word within ± 1500

Parameters in Assignment CMS

- 300 entries in each row, 4 rows
- 32-bit counters (a little wasteful!)
- 7.3MB of data summarized in 4.8KB
- Really accurate still: in 1.2 million word stream, can estimate num occurrences of each word within ± 1500
- Often more accurate! Also: feel free to try 1000 or 10000 entries per row; it gets quite accurate

Hyper Log Log Counting

Setting up

- Count-min sketch takes up a lot of space!

Setting up

- Count-min sketch takes up a lot of space!
- OK not really. But, it stores a lot of information about the stream

Setting up

- Count-min sketch takes up a lot of space!
- OK not really. But, it stores a lot of information about the stream
- Common question: how many *unique* elements are there in the stream?

The problem we're trying to solve

- Stream of N elements

The problem we're trying to solve

- Stream of N elements
- Approximate number of unique elements

The problem we're trying to solve

- Stream of N elements
- Approximate number of unique elements
- (Compare to CMS: stores approximately how many there are of *each* element)

The problem we're trying to solve

- Stream of N elements
- Approximate number of unique elements
- To do this exactly: need dictionary of all elements we've already seen.

The problem we're trying to solve

- Stream of N elements
- Approximate number of unique elements
- To do this exactly: need dictionary of all elements we've already seen.
- How can you count unique elements approximately? Challenge: don't want to double-count when we see an element twice.

Cool way to solve this

- Let’s hash each item as it comes in

Cool way to solve this

- Let's hash each item as it comes in
- Then instead of a list of items, we get a list of random hashes

Cool way to solve this

- Let's hash each item as it comes in
- Then instead of a list of items, we get a list of random hashes
- Idea: let's look at a rare event in these hashes. The more often it happens, the more *distinct hashes* (and thus distinct items) we must be seeing!

Cool way to solve this

- Let's hash each item as it comes in
- Then instead of a list of items, we get a list of random hashes
- Idea: let's look at a rare event in these hashes. The more often it happens, the more *distinct hashes* (and thus distinct items) we must be seeing!
- In particular: how many 0s does each hash end with?

Hashes ending in 0s

- What is the probability that a hash ends in ten 0’s?

Hashes ending in 0s

- What is the probability that a hash ends in ten 0’s? Answer: 1/1024

Hashes ending in 0s

- What is the probability that a hash ends in ten 0's? Answer: 1/1024
- So if we have two distinct elements, it's very unlikely that the hash of either will end in 10 0's.

Hashes ending in 0s

- What is the probability that a hash ends in ten 0's? Answer: 1/1024
- So if we have two distinct elements, it's very unlikely that the hash of either will end in 10 0's.
- If we have $2^{10} = 1024$ distinct elements, it's pretty likely that the hash of one will end with 10 0's!

Hashes ending in 0s

- What is the probability that a hash ends in ten 0's? Answer: 1/1024
- So if we have two distinct elements, it's very unlikely that the hash of either will end in 10 0's.
- If we have $2^{10} = 1024$ distinct elements, it's pretty likely that the hash of one will end with 10 0's!
- Note "distinct!" All of this comes back to estimating how many *unique* elements there are. Unique elements give a new hash, and a new opportunity for many zeroes. Non-unique elements don't give a new hash.

Example

You see the following hashes one by one:

0010001010101001

Example

You see the following hashes one by one:

0010110010111101

Example

You see the following hashes one by one:

0001000111101111

Example

You see the following hashes one by one:

0000001011000011

Example

You see the following hashes one by one:

0110010010011100

Example

You see the following hashes one by one:

10001010111100001

Example

You see the following hashes one by one:

0110100100111101

Example

You see the following hashes one by one:

0011101001100010

Example

You see the following hashes one by one:

0110000000001110

Example

You see the following hashes one by one:

0011001110001111

Example

You see the following hashes one by one:

1111100010110000

Example

You see the following hashes one by one:

11111101010111100

Example

You see the following hashes one by one:

11000100011010011

Example

You see the following hashes one by one:

1101110101001100

How many unique items were there?

Example 2

You see the following hashes one by one:

0010001010101001

Example 2

You see the following hashes one by one:

0010110010111101

Example 2

You see the following hashes one by one:

0011101001100010

Example 2

You see the following hashes one by one:

0010001010101001

Example 2

You see the following hashes one by one:

0011101001100010

Example 2

You see the following hashes one by one:

0010110010111101

Example 2

You see the following hashes one by one:

0010110010111101

Example 2

You see the following hashes one by one:

0010001010101001

Example 2

You see the following hashes one by one:

0010110010111101

Example 2

You see the following hashes one by one:

0010001010101001

Example 2

You see the following hashes one by one:

0010110010111101

Example 2

You see the following hashes one by one:

0010001010101001

Example 2

You see the following hashes one by one:

0010110010111101

Example 2

You see the following hashes one by one:

0010001010101001

Example 2

You see the following hashes one by one:

0010110010111101

How many unique items were there? Was it more or less than the last one?

Which example had more unique items?

Which example had more unique items?

- Answer: 1st had 14 items, 2nd had 3

Which example had more unique items?

- Answer: 1st had 14 items, 2nd had 3
- Notice that only one hash in the second example ended with 0

Which example had more unique items?

- Answer: 1st had 14 items, 2nd had 3
- Notice that only one hash in the second example ended with 0
    - Extremely unlikely if there were 14 different elements!

Which example had more unique items?

- Answer: 1st had 14 items, 2nd had 3
- Notice that only one hash in the second example ended with 0
    - Extremely unlikely if there were 14 different elements!
- One of the items in the first example ended with 4 0’s

Which example had more unique items?

- Answer: 1st had 14 items, 2nd had 3
- Notice that only one hash in the second example ended with 0
  - Extremely unlikely if there were 14 different elements!
- One of the items in the first example ended with 4 0's
  - Unlikely if there were 3 elements!

Intuitive loglog counting

- Let’s say that the hash ending with the most 0s has k 0s at the end

Intuitive loglog counting

- Let's say that the hash ending with the most 0s has k 0s at the end
- Any given hash has k 0s with probability $1/2^k$

Intuitive loglog counting

- Let's say that the hash ending with the most 0s has k 0s at the end
- Any given hash has k 0s with probability $1/2^k$
- So it seems that, there are probably something like 2^k items

Intuitive loglog counting

- Let's say that the hash ending with the most 0s has k 0s at the end
- Any given hash has k 0s with probability $1/2^k$
- So it seems that, there are probably something like 2^k items
- But: if we're just off by 1 or 2 zeroes, that affects our answer by a lot! (We don't get good *concentration bounds*)

Improving reliability

- How do we improve the consistency of a random process?

Improving reliability

- How do we improve the consistency of a random process? Repeat!* (*in a particular way)

Improving reliability

- How do we improve the consistency of a random process? Repeat!* (*in a particular way)
- Hash each item *first* to one of several counters

Improving reliability

- How do we improve the consistency of a random process? Repeat!* (*in a particular way)
- Hash each item *first* to one of several counters
- For each counter, keep track of 1 + the maximum number of 0s at end of any item hashed to that counter

Improving reliability

- How do we improve the consistency of a random process? Repeat!* (*in a particular way)
- Hash each item *first* to one of several counters
- For each counter, keep track of 1 + the maximum number of 0s at end of any item hashed to that counter
- For CMS, we took the min. What do we do here to combine the estimates?

Improving reliability

- How do we improve the consistency of a random process? Repeat!* (*in a particular way)
- Hash each item *first* to one of several counters
- For each counter, keep track of 1 + the maximum number of 0s at end of any item hashed to that counter
- For CMS, we took the min. What do we do here to combine the estimates?
- Answer: It's complicated. (And the rationale is outside the scope of the course.)

HyperLogLog Counting

- Keep an array of m counters (m is a power of 2); let's call it M

HyperLogLog Counting

- Keep an array of m counters (m is a power of 2); let's call it M
- Hash each item as it comes in. Then:

HyperLogLog Counting

- Keep an array of m counters (m is a power of 2); let's call it M
- Hash each item as it comes in. Then:
  - Get an index i, consisting of the lowest $\log_2 m$ bits of $h(x)$. Then i will index into M. Shift off these bits.

HyperLogLog Counting

- Keep an array of m counters (m is a power of 2); let's call it M
- Hash each item as it comes in. Then:
  - Get an index i, consisting of the lowest $\log_2 m$ bits of $h(x)$. Then i will index into M. Shift off these bits.
  - Look at the remaining bits. Let z be the number of zeroes. If $z + 1 > M[i]$, set $M[i] = z + 1$

HyperLogLog Counting

- Keep an array of m counters (m is a power of 2); let's call it M
- Hash each item as it comes in. Then:
  - Get an index i, consisting of the lowest $\log_2 m$ bits of $h(x)$. Then i will index into M. Shift off these bits.
  - Look at the remaining bits. Let z be the number of zeroes. If $z + 1 > M[i]$, set $M[i] = z + 1$
- **Make sure to add 1 to your count of the number of zeroes**

Getting an Estimate

- At the end, we have an array M, each containing a count

[2] You have to look this constant up.

Getting an Estimate

- At the end, we have an array M, each containing a count
- Let
$$Z = \sum_{i=0}^{m-1} \left(\frac{1}{2}\right)^{M[i]}.$$

[2] You have to look this constant up.

Getting an Estimate

- At the end, we have an array M, each containing a count
- Let

$$Z = \sum_{i=0}^{m-1} \left(\frac{1}{2}\right)^{M[i]}.$$

- Let b be a bias constant.[2] For $m = 32$, $b = .697$.

[2] You have to look this constant up.

Getting an Estimate

- At the end, we have an array M, each containing a count
- Let
$$Z = \sum_{i=0}^{m-1} \left(\frac{1}{2}\right)^{M[i]}.$$
- Let b be a bias constant.[2] For $m = 32$, $b = .697$.
- Return bm^2/Z.

[2] You have to look this constant up.

Example (with $m = 8$; in practice m is higher)

$$x_1$$

Example (with $m = 8$; in practice m is higher)

$$x_1$$

$$h(x_1) = 010001000111110111111101010110$$

Example (with $m = 8$; in practice m is higher)

$$x_1$$

$$h(x_1) = 0100010001111101111111101010110$$

index $= 110$ Remaining: 0100010001111101111111101010

0	0	0	0	0	0	0	0
000	001	010	011	100	101	110	111

Example (with $m = 8$; in practice m is higher)

$$x_1$$

$$h(x_1) = 0100010001111101111111101010110$$

index $= 110$ Remaining: $0100010001111101111111101010$

000	001	010	011	100	101	110	111
0	0	0	0	0	0	0	0

Example (with $m = 8$; in practice m is higher)

$$x_1$$

$$h(x_1) = 01000100011111011111101010110$$

index $= 110$ Remaining: $01000100011111011111101010$

0	0	0	0	0	0	0	0
000	001	010	011	100	101	110	111

The remaining hash ends with 1 zero, so we want to store 2. The counter stores less than 2, so we store it.

Example (with $m = 8$; in practice m is higher)

$$x_1$$

$$h(x_1) = 01000100011111011111101010110$$

index $= 110$ Remaining: 01000100011111011111101010

0	0	0	0	0	0	2	0
000	001	010	011	100	101	110	111

The remaining hash ends with 1 zero, so we want to store 2. The counter stores less than 2, so we store it.

Example (with $m = 8$; in practice m is higher)

x_2

Example (with $m = 8$; in practice m is higher)

$$x_2$$

$$h(x_2) = 011110001100100001111010010110$$

Example (with $m = 8$; in practice m is higher)

x_2

$h(x_2) = 01111000110010000111101001011 0$

index $= 110$ Remaining: 01111000110010000111101001 0

0	0	0	0	0	0	2	0
000	001	010	011	100	101	110	111

Example (with $m = 8$; in practice m is higher)

x_2

$h(x_2) = 011110001100100001111010010110$

index $= 110$ Remaining: $011110001100100001111010010$

0	0	0	0	0	0	2	0
000	001	010	011	100	101	110	111

Example (with $m = 8$; in practice m is higher)

$$x_2$$

$$h(x_2) = 01111000110010000111101001011 0$$

index $= 110$ Remaining: 01111000110010000111101001 0

0	0	0	0	0	0	2	0
000	001	010	011	100	101	110	111

The remaining hash ends with 1 zero, so we want to store 2. The counter stores 2, so we keep it as-is.

Example (with $m = 8$; in practice m is higher)

x_3

Example (with $m = 8$; in practice m is higher)

$$x_3$$

$$h(x_3) = 110011011101100000011010000001$$

Example (with $m = 8$; in practice m is higher)

$$x_3$$

$$h(x_3) = 1100110111011000000011010000001$$

index $= 001$ Remaining: $1100110111011000000011010000$

0	0	0	0	0	0	2	0
000	001	010	011	100	101	110	111

Example (with $m = 8$; in practice m is higher)

$$x_3$$

$$h(x_3) = 11001101110110000000110100000001$$

index $= 001$ Remaining: $1100110111011000000011010000$

0	0	0	0	0	0	2	0
000	001	010	011	100	101	110	111

Example (with $m = 8$; in practice m is higher)

$$x_3$$

$$h(x_3) = 110011011101100000011010000001$$

index $= 001$ Remaining: 110011011101100000011010000

0	0	0	0	0	0	2	0
000	001	010	011	100	101	110	111

The remaining hash ends with 4 zeroes, so we want to store 5. The counter stores 0, so we store 5 in the slot.

Example (with $m = 8$; in practice m is higher)

$$x_3$$

$$h(x_3) = 110011011101100000011010000001$$

$$\text{index} = 001 \quad \text{Remaining: } 1100110111011000000011010000$$

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

The remaining hash ends with 4 zeroes, so we want to store 5. The counter stores 0, so we store 5 in the slot.

Example (with $m = 8$; in practice m is higher)

x_4

Example (with $m = 8$; in practice m is higher)

$$x_4$$

$$h(x_4) = 100010011101101110110110111001$$

Example (with $m = 8$; in practice m is higher)

$$x_4$$

$$h(x_4) = 10001001110110111011011011 1001$$

index $= 001$ Remaining: 10001001110110111011011011 1

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

The remaining hash ends with 0 zeroes, so we want to store 1. The counter stores 5, so we keep the slot as-is.

Example (with $m = 8$; in practice m is higher)

x_4

$h(x_4) = 100010011101101110110110111001$

index $= 001$ Remaining: $100010011101101110110110111$

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

Example (with $m = 8$; in practice m is higher)

$$x_4$$

$$h(x_4) = 100010011101101110110110111001$$

index $= 001$ Remaining: $100010011101101110110110111$

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

The remaining hash ends with 0 zeroes, so we want to store 1. The counter stores 5, so we keep the slot as-is.

Example (with $m = 8$; in practice m is higher)

x_2

Example (with $m = 8$; in practice m is higher)

$$x_2$$

$$h(x_2) = 011110001100100001111010010110$$

Example (with $m = 8$; in practice m is higher)

$$x_2$$

$$h(x_2) = 0111100011001000001111010010110$$

index $= 110$ Remaining: $0111100011001000001111010010110$

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

Example (with $m = 8$; in practice m is higher)

$$x_2$$

$$h(x_2) = 011110001100100001111010010110$$

index $= 110$ Remaining: $011110001100100001111010010110$

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

Example (with $m = 8$; in practice m is higher)

$$x_2$$

$$h(x_2) = 011110001100100001111010010110$$

index $= 110$ Remaining: $011110001100100001111010010110$

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

The remaining hash ends with 1 zero, so we want to store 2. The counter stores 2, so we keep it as-is.

At the end of the day

Have an array of counters:

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

At the end of the day

Have an array of counters:

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

- Sum up $(1/2)^{M[j]}$ across all $j = 0$ to $m - 1$; store in Z

At the end of the day

Have an array of counters:

0	5	0	0	0	0	2	0
000	001	010	011	100	101	110	111

- Sum up $(1/2)^{M[j]}$ across all $j = 0$ to $m - 1$; store in Z
- Return bm^2/Z. Here $m = 8$. We would have to look up the value of b for 8. (No one does HyperLogLog with 8)

Discussion

- How big do our counters need to be?

Discussion

- How big do our counters need to be?
- Need to be long enough to count the longest string of 0s in any hash

Discussion

- How big do our counters need to be?
- Need to be long enough to count the longest string of 0s in any hash
- Size $> \log\log(\text{number of distinct elements})$ (hence the *loglog* in the name)

Discussion

- How big do our counters need to be?
- Need to be long enough to count the longest string of 0s in any hash
- Size $> \log\log(\text{number of distinct elements})$ (hence the *loglog* in the name)
- 8-bit counters are good enough, so long as the number of elements in your stream is less than the number of particles in the universe

Discussion

- How big do our counters need to be?
- Need to be long enough to count the longest string of 0s in any hash
- Size $> \log\log(\text{number of distinct elements})$ (hence the *loglog* in the name)
- 8-bit counters are good enough, so long as the number of elements in your stream is less than the number of particles in the universe
- Note: one thing to be careful of is hash length. But 64 bit hashes should be good enough for any reasonable application (and 32 bits is usually fine)

HLL in the Assignment

- We'll use $m = 32$ counters
- Bias constant is .697

HLL Beyond the Assignment

- HLL does poorly when the number of distinct items is not much more than m

HLL Beyond the Assignment

- HLL does poorly when the number of distinct items is not much more than m
- Or is very very high

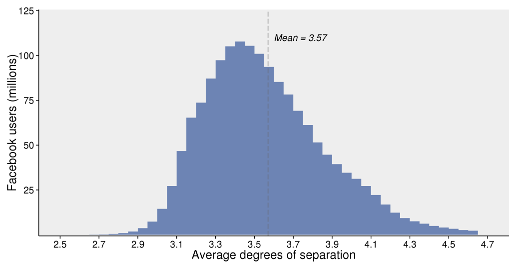
HLL Beyond the Assignment

- HLL does poorly when the number of distinct items is not much more than m
- Or is very very high
- Google developed HyperLogLog++ to help deal with these problems

HLL Beyond the Assignment

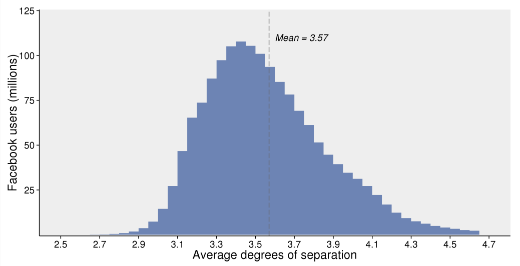
- HLL does poorly when the number of distinct items is not much more than m
- Or is very very high
- Google developed HyperLogLog++ to help deal with these problems
- Other known improvements as well

One More Cool Thing



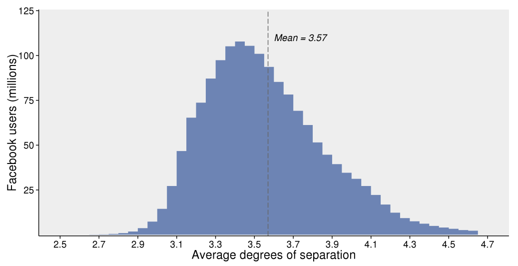
- Facebook developed an HLL-based algorithm to calculate the *diameter* of a graph

One More Cool Thing



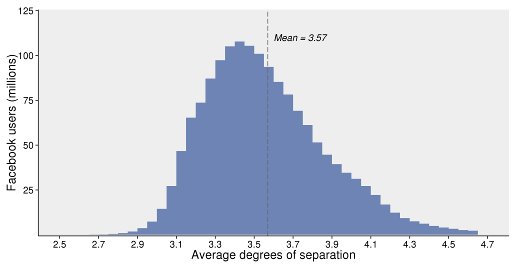
- Facebook developed an HLL-based algorithm to calculate the *diameter* of a graph
  - In terms of “friend jumps”, how far away are the furthest people in the Facebook graph?

One More Cool Thing



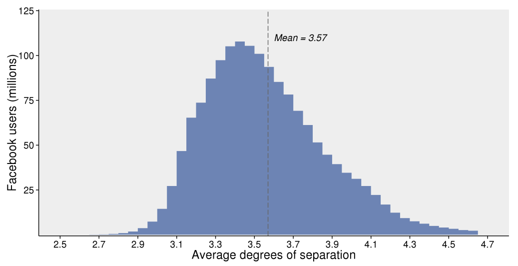
- Facebook developed an HLL-based algorithm to calculate the *diameter* of a graph
  - In terms of “friend jumps”, how far away are the furthest people in the Facebook graph?
  - How far away are two people on average?

One More Cool Thing



- Facebook developed an HLL-based algorithm to calculate the *diameter* of a graph
  - In terms of “friend jumps”, how far away are the furthest people in the Facebook graph?
  - How far away are two people on average?
- Usually takes $O(n^2)$ time!

One More Cool Thing



- Facebook developed an HLL-based algorithm to calculate the *diameter* of a graph
  - In terms of “friend jumps”, how far away are the furthest people in the Facebook graph?
  - How far away are two people on average?
- Usually takes $O(n^2)$ time!
- Theirs is essentially linear time, gives extremely accurate results

Hash Functions in Practice

What do we want out of a hash function?

Of course, we want consistency (each time we hash an item we get the same result back). What else might we want?

- Fast
- Low space requirements (i.e. may need to store a seed; don't want that to be too big)
- Good collision avoidance
- Bear in mind: different hashes work on different types of elements. We'll focus on integers and strings (especially strings)

Ideal Hash Functions (Independent, Uniform Hashing)

- Best possible collision avoidance

Ideal Hash Functions (Independent, Uniform Hashing)

- Best possible collision avoidance
- But: require extremely large space usage unless universe of possible elements is extremely small

Ideal Hash Functions (Independent, Uniform Hashing)

- Best possible collision avoidance
- But: require extremely large space usage unless universe of possible elements is extremely small
- You did use one of these...

Ideal Hash Functions (Independent, Uniform Hashing)

- Best possible collision avoidance
- But: require extremely large space usage unless universe of possible elements is extremely small
- You did use one of these...
  - For h on Assignment 3! Those values were all chosen independently, completely at random

Hashing in Java

- Anyone know how Java hashes a 64 bit `Long`?
- `return x ^ (x >> 32);`
- Advantages of this?

Is this good for:

- In cuckoo filter: h_1, h, f?
  - h_1 and f: might work if elements are fairly well-spread (we take mod)
  - h: probably won't work (output too small)
- CMS? HLL?
  - CMS might be OK; prob not (same as above)
  - HLL likely useless unless elements very uniformly spread

Multiply-Shift Hashing

```
uint64_t hash3(uint64_t value){
    return (uint64_t)(value * 0x765a3cc864bd9779) >> (64 - SHIFT);
}
```

- Hash from Assignment 1
- Seed is a large prime number to multiply by; can also add a large random prime
- Advantages?
    - Fast! (And easy.)

Multiply-Shift Hashing

```
uint64_t hash3(uint64_t value){
    return (uint64_t)(value * 0x765a3cc864bd9779) >> (64 -
        SHIFT);
}
```

- How good is it?
  - Pretty good! For any x, y, $\Pr[h(x) = h(y)] = 1/n$.
  - But unfortunately behavior doesn't extend to larger numbers of elements.
- Let's say we use this for a hash table with chaining (n items, n chains). What is the expected number of elements we find during a query q?
- $X_i = 1$ if $h(x_i) = h(q)$. Then $E[X_i] = 1/n$. By linearity of expectation, total number of items is $\sum_{i=1}^{n} 1/n = 1$.

Multiply-Shift hashing for other data structures

- Is this going to work well for a filter?

Multiply-Shift hashing for other data structures

- Is this going to work well for a filter?
    - Probably not. Would have to try it.

Multiply-Shift hashing for other data structures

- Is this going to work well for a filter?
  - Probably not. Would have to try it.
- Count-min sketch?

Multiply-Shift hashing for other data structures

- Is this going to work well for a filter?
  - Probably not. Would have to try it.
- Count-min sketch?
  - On paper should work pretty well! After all, our analysis only used the expectation

Multiply-Shift hashing for other data structures

- Is this going to work well for a filter?
  - Probably not. Would have to try it.
- Count-min sketch?
  - On paper should work pretty well! After all, our analysis only used the expectation
  - I’d guess it won’t work as well as with a better hash function

Multiply-Shift hashing for other data structures

- Is this going to work well for a filter?
    - Probably not. Would have to try it.
- Count-min sketch?
    - On paper should work pretty well! After all, our analysis only used the expectation
    - I'd guess it won't work as well as with a better hash function
- Hyperloglog?

Multiply-Shift hashing for other data structures

- Is this going to work well for a filter?
  - Probably not. Would have to try it.
- Count-min sketch?
  - On paper should work pretty well! After all, our analysis only used the expectation
  - I'd guess it won't work as well as with a better hash function
- Hyperloglog?
  - Would have to try but I would very much suspect it would not work well at all

Murmurhash

- Popular practical hash function

Murmurhash

- Popular practical hash function
- Uses repeated MUltiply and Rotate operations
  - Rotate is like shift, but bits that “fall off” are replaced on other side
  - Can be implemented with two shifts and an OR

Murmurhash

- Popular practical hash function
- Uses repeated MUltiply and Rotate operations
  - Rotate is like shift, but bits that “fall off” are replaced on other side
  - Can be implemented with two shifts and an OR
- Code isn’t exactly short; 50 operations to hash a number

Murmurhash Code

```
for(i = -nblocks; i; i++)
{
  uint32_t k1 = getblock(blocks,i*4+0);
  uint32_t k2 = getblock(blocks,i*4+1);
  uint32_t k3 = getblock(blocks,i*4+2);
  uint32_t k4 = getblock(blocks,i*4+3);

  k1 *= c1; k1  = ROTL32(k1,15); k1 *= c2; h1 ^= k1;

  h1 = ROTL32(h1,19); h1 += h2; h1 = h1*5+0x561ccd1b;

  k2 *= c2; k2  = ROTL32(k2,16); k2 *= c3; h2 ^= k2;

  h2 = ROTL32(h2,17); h2 += h3; h2 = h2*5+0x0bcaa747;

  k3 *= c3; k3  = ROTL32(k3,17); k3 *= c4; h3 ^= k3;

  h3 = ROTL32(h3,15); h3 += h4; h3 = h3*5+0x96cd1c35;

  k4 *= c4; k4  = ROTL32(k4,18); k4 *= c1; h4 ^= k4;

  h4 = ROTL32(h4,13); h4 += h1; h4 = h4*5+0x32ac3b17;
}
```

Murmurhash Code

```
switch(len & 15)
{
case 15: k4 ^= tail[14] << 16;
case 14: k4 ^= tail[13] << 8;
case 13: k4 ^= tail[12] << 0;
         k4 *= c4; k4  = ROTL32(k4,18); k4 *= c1; h4 ^= k4;

case 12: k3 ^= tail[11] << 24;
case 11: k3 ^= tail[10] << 16;
case 10: k3 ^= tail[ 9] << 8;
case  9: k3 ^= tail[ 8] << 0;
         k3 *= c3; k3  = ROTL32(k3,17); k3 *= c4; h3 ^= k3;
-- 12 lines: case  8: k2 ^= tail[ 7] << 24;-----------------------------
};
--  4 lines: -------------------------------------------------------------
h1 ^= len; h2 ^= len; h3 ^= len; h4 ^= len;

h1 += h2; h1 += h3; h1 += h4;
h2 += h1; h3 += h1; h4 += h1;

h1 = fmix32(h1);
h2 = fmix32(h2);
h3 = fmix32(h3);
h4 = fmix32(h4);

h1 += h2; h1 += h3; h1 += h4;
h2 += h1; h3 += h1; h4 += h1;
```

(The light grey lines skip pieces of code.)

Murmurhash3 Performance

- No known worst-case guarantees (not even $\Pr(h(x) = h(y)) = O(1/n)$)

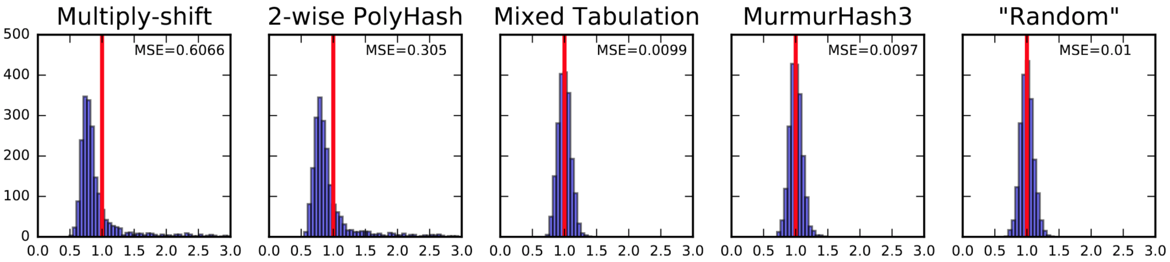
Murmurhash3 Performance

- No known worst-case guarantees (not even $\Pr(h(x) = h(y)) = O(1/n)$)
- Someday may discover: might not work well in some circumstances

Murmurhash3 Performance

- No known worst-case guarantees (not even $\Pr(h(x) = h(y)) = O(1/n)$)
- Someday may discover: might not work well in some circumstances
- This is what happened to Murmurhash2:
    - "Will this flaw cause your program to fail? Probably not - what this means in real-world terms is that if your keys contain repeated 4-byte values AND they differ only in those repeated values AND the repetitions fall on a 4-byte boundary, then your keys will collide with a probability of about 1 in $2^{27.4}$ instead of 2^{32}. Due to the birthday paradox, you should have a better than 50% chance of finding a collision in a group of 13115 bad keys instead of 65536."
    - `https://sites.google.com/site/murmurhash/murmurhash2flaw`

Murmurhash3 Performance



Average of square of bucket sizes. Data is an intentionally bad (albeit reasonable) case

From “Practical Hash Functions for Similarity Estimation and Dimensionality Reduction” by Dahlgaard, Knudsen, Thorup NeurIPS 2017

Murmurhash3 Performance in Practice

- Much more resilient than multiply-shift to more-difficult statistical tests (beyond average case)

Murmurhash3 Performance in Practice

- Much more resilient than multiply-shift to more-difficult statistical tests (beyond average case)
- Visual example: let's say we hash "number strings": "1", "2", ... "216553"

Murmurhash3 Performance in Practice

- Much more resilient than multiply-shift to more-difficult statistical tests (beyond average case)
- Visual example: let's say we hash "number strings": "1", "2", ... "216553"
- Cool experiment from `https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed`

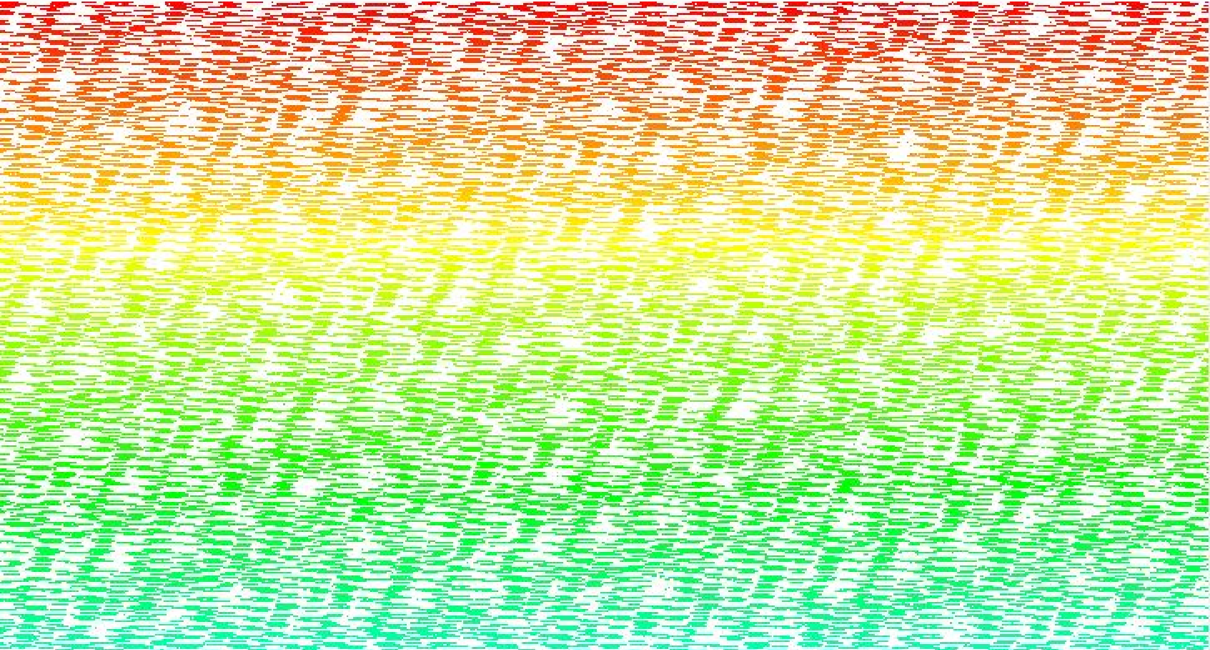
Murmurhash3 Performance in Practice

- Much more resilient than multiply-shift to more-difficult statistical tests (beyond average case)
- Visual example: let's say we hash "number strings": "1", "2", … "216553"
- Cool experiment from `https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed`
- I wouldn't normally cite stackexchange but this is really cool

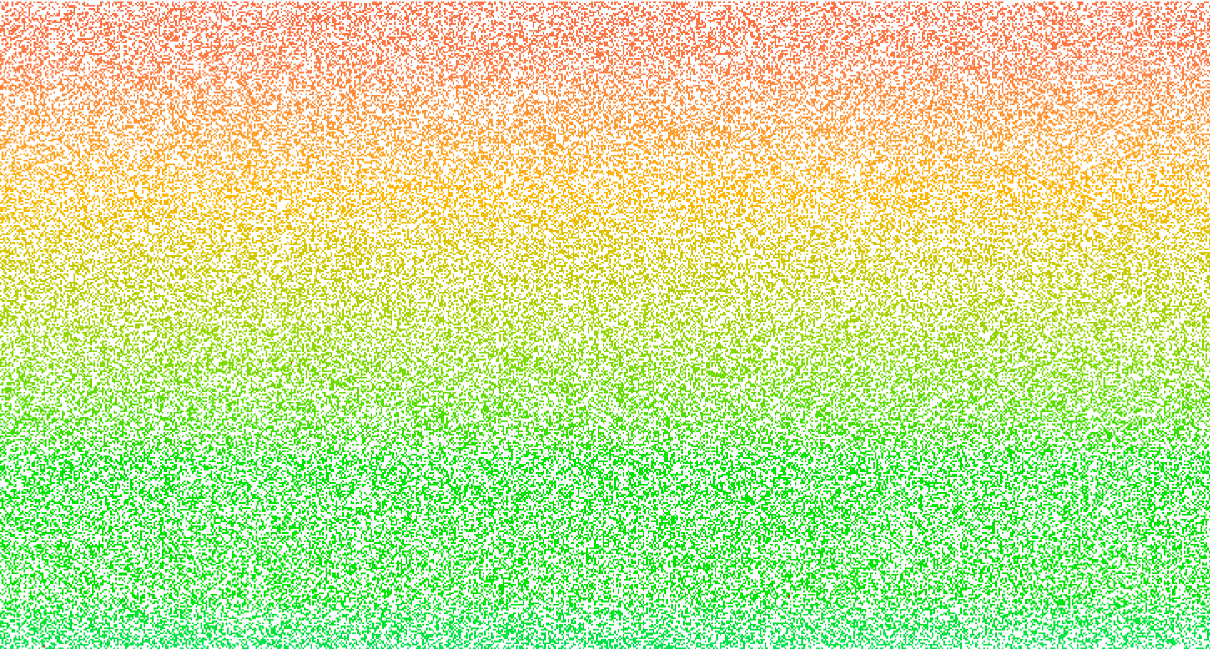
Murmurhash3 Performance in Practice

- Much more resilient than multiply-shift to more-difficult statistical tests (beyond average case)
- Visual example: let's say we hash "number strings": "1", "2", ... "216553"
- Cool experiment from `https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed`
- I wouldn't normally cite stackexchange but this is really cool
- Compare SDBM (another popular hash) with Murmurhash2; fill in pixel if corresponding table entry is hashed to

SDBM (lots of chunks of full cells!)



Murmurhash2 (visually: random)



One last murmurhash question

- Murmurhash really just does a bunch of arbitrary multiplies and rotates

One last murmurhash question

- Murmurhash really just does a bunch of arbitrary multiplies and rotates
- Is there anything special about this specific sequence, or will any such set work pretty well?

One last murmurhash question

- Murmurhash really just does a bunch of arbitrary multiplies and rotates
- Is there anything special about this specific sequence, or will any such set work pretty well?
- Answer: others might not work. Example: “SuperFastHash” also uses multiplies and rotates

Hash comparison

```
Hash            Lowercase       Random UUID  Numbers
=============   =============   ===========  =============
Murmur             145 ns       259 ns          92 ns
                     6 collis     5 collis       0 collis
SDBM               148 ns       484 ns          90 ns
                     4 collis     6 collis       0 collis
SuperFastHash      164 ns       344 ns         118 ns
                    85 collis     4 collis   18742 collis
```

SuperFastHash has bad performance on lowercase English words, and horrendous performance on numbers-as-strings.

(Also from `https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed`)

Unease with our options

- Murmurhash seems to do well (and is fast), but has few guarantees.

Unease with our options

- Murmurhash seems to do well (and is fast), but has few guarantees.
- What do we do if we’re OK with a slightly slower hash, but we REALLY want to be sure it does well?

Unease with our options

- Murmurhash seems to do well (and is fast), but has few guarantees.
- What do we do if we’re OK with a slightly slower hash, but we REALLY want to be sure it does well?
- Answer: cryptographic hashes! Secure even for cryptographic applications; no known statistical weaknesses

Unease with our options

- Murmurhash seems to do well (and is fast), but has few guarantees.
- What do we do if we’re OK with a slightly slower hash, but we REALLY want to be sure it does well?
- Answer: cryptographic hashes! Secure even for cryptographic applications; no known statistical weaknesses
- Examples: SHA-3, BLAKE2, many others

Unease with our options

- Murmurhash seems to do well (and is fast), but has few guarantees.
- What do we do if we're OK with a slightly slower hash, but we REALLY want to be sure it does well?
- Answer: cryptographic hashes! Secure even for cryptographic applications; no known statistical weaknesses
- Examples: SHA-3, BLAKE2, many others
- Broken: MD5, SHA-1, many others

(Source: `https://shattered.io`)