# Frequency and Cardinality Estimation using Sketching

Matthias Petri

Fri 20/5/16

# What we'll learn today

Practical, memory efficient methods to

- ▶ Estimate the **Cardinality** of a Set.
- **E**stimate the **Frequency** of an item in a stream.

### Set Cardinality

#### Problem

Given a stream of items from a universe U with |U|=n, keep track of the size m of the set S containing all unique items that have appeared so far.

### Example

Count the number of unique English words in Wikipedia

What is U in this case? What is S?

### Set Cardinality - Simple solutions

Keep track of the items that have appeared so far in:

- ► Hash table
- Binary search tree
- ► Array of size *n*

### Set Cardinality - Space Usage

#### Problem

Space usage at least linear to the number of items in the set S.

Many large (big data!) problems exist where this is not acceptable:

- How many unique IP addresses click on different links on a website?
- ► How many unique viewers per country does my video youtube have?

What is U in these examples? What is S?

# Set Cardinality - Estimation instead of exact counting

#### Idea

A good **estimate**  $\hat{m}$  of the actual cardinality is sufficient in most cases

What is a good estimate?

- ▶ The estimation error should be low
- Space and Runtime efficient
- ► Theoretical guarantees

# Set Cardinality - Algorithmic Idea

### Coin Flip Game

- Start flipping a coin.
- ▶ Keep track of results of each flip on a piece of paper.
- ▶ After some time you stop flipping the coin.
- ▶ Count the largest run of "heads" in your result table.

### Length of the largest run of heads?

- ▶ Largest run is 3 or less? You just flipped a few times.
- ► Largest run ins 15? Any guesses?

# Set Cardinality - Coinflip Math

### Probability

- $\,\blacktriangleright\,$  The probability of flipping heads for one coin toss is p=0.5
- ▶ Intuitively, to observe *k* consecutive heads, the number of tosses *n* is expected to be high.

Probability of k = 15?

### Set Cardinality - Coinflip Math

### Probability

- ▶ The probability of flipping heads for one coin toss is p = 0.5
- ▶ Intuitively, to observe *k* consecutive heads, the number of tosses *n* is expected to be high.

### Probability of k = 15?

So, k = 15 consecutive heads **suggests** lots of coinflips.

### From Coinflips to Counting items

A "good" hash function h mapping from U to [0,p] uniformly distributes hash values within [0,p].

### Example

A hash function h produces a 32 bit hash value, thus the hash value is a number in the range  $[0,2^{32}-1].$ 

Lets say the hash value v is  $3{,}037{,}935{,}517$  and this is binary representation of v:

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Each of those bits can be seen as a coinflip.

# Set Cardinality - Counting items

#### Idea

Keep track of largest number k of trailing 0s in bitpattern of the hash value:

### Example k=4

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

### Experimentally

Bit Pattern	Probability
P(10)	?
P(100)	?
P(1000)	?
P(10000)	?
P(100000)	?

Bit Pattern	Probability
P(10)	$\frac{1}{2}$
P(100)	?
P(1000)	?
P(10000)	?
P(100000)	?

Bit Pattern	Probability
P(10)	$\frac{1}{2}$
P(100)	$\frac{1}{4}$
P(1000)	?
P(10000)	?
P(100000)	?

Bit Pattern	Probability
P(10)	$\frac{1}{2}$
P(100)	$\frac{1}{4}$
P(1000)	$\frac{1}{8}$
P(10000)	?
P(100000)	?

Bit Pattern	Probability
P(10)	$\frac{1}{2}$
P(100)	$\frac{1}{4}$
P(1000)	$\frac{1}{8}$
P(10000)	$\frac{1}{16}$
P(100000)	$\frac{1}{32}$

In **expectation** we have to hash  $2^l$  values to encounter a hash value with l trailing zeros.

Let k be the largest number of trailing 0s we encounter.

So, we could estimate the set cardinality  $\hat{m}\approx 2^k$  ? What if we are "unlucky"?

# Set Cardinality - The Unlucky Case

#### Problem

We could come across a hash values with  $15\ \rm zeros$  "earlier" than expected.

### Stochastic Averaging

- Run q estimators at the same time.
- lacktriangle Divide the hash range [0, q-1] into sub ranges.
- Instead of storing only one k, store q counters in an array D[0,q-1] and measure the average  $\hat{k}$
- $\blacktriangleright$  Cardinality estimate  $\hat{m}=cq2^{\hat{k}}$  where c=0.39701 is a "magical" constant.
- Estimate more "resilient" to outliers.

- Store  $q=2^l$  counters of size  $\log \log p$  (where p is the size of the hash range. For example:  $[0..2^{32}-1]$ )
- Use bottom l bits to pick counter to update
- ▶ Use bits [32..l] to determine number of trailing 0s k

Example 
$$l = 4$$
,  $q = 2^4 = 8$ 

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 1 0 1 1 0 1 0 1 0 1 0 0 0 1 0 0 1 1 0 0 1 1 0 1 1 1 1 0 0 0 0 1 0 1

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

- ▶ Store  $q = 2^l$  counters of size  $\log \log p$  (where p is the size of the hash range. For example:  $[0..2^{32} 1]$ )
- ▶ Use bottom *l* bits to pick counter to update
- ▶ Use bits [32..l] to determine number of trailing 0s k

# Example l = 4, $q = 2^4 = 8$

- ▶ Store  $q = 2^l$  counters of size  $\log \log p$  (where p is the size of the hash range. For example:  $[0..2^{32} 1]$ )
- ▶ Use bottom *l* bits to pick counter to update
- ▶ Use bits [32..l] to determine number of trailing 0s k

### Example l = 4, $q = 2^4 = 8$

#### How good is it?

- Low relative estimation error of  $\sigma=1.30/\sqrt{q}$
- ▶ Example: q = 2048. Relative error  $\sigma = 1.30/\sqrt{2014} = 2.87\%$
- ▶ The estimate is within  $\sigma$ ,  $2\sigma$ , and  $3\sigma$  of the exact value of the cardinality n in respectively 65%, 95%, and 99% of the cases
- ▶ Small space. Only  $\log \log p = 5 6$  bits per bucket.
- Estimating the cardinality of a stream of 100 millions items, with q=2048 buckets to an accuracy of 2% requires only 2 kb memory!

### Intermission: Bitcoin

Similar concepts are also used in the bitcoin framework as a "Proof of Work"

- ➤ To "mine" a bitcoin you have to proof that you have performed a certain amount of work.
- ▶ A small string *Q* is generated based on the existing transactions in the bitcoin network.
- ► Task: Find a string X which has Q as a suffix, which produces a hash containing a certain amount of trailing 0s.
- ► First person to "find" such a string gets awarded the next bitcoin.

### Frequency Estimation

#### Problem

Given a stream of items from a universe U with  $\lvert U \rvert = n$ , keep track of the frequency  $f_i$  of each item i in the stream.

### Example

Count the word frequencies in Wikipedia

#### **Data Structures**

Hash Table, BST, Array of size n

# Space requirements for Frequency Statistics

### Space Usage

Simple solutions such as Hash Tables require space  ${\cal O}(m)$ 

### Large scale problems

- Network traffic analysis (Which hosts on the internet are accessed the most from the Unimelb network?)
- Website usage analysis (How many users from Melbourne have clicked on this image?)
  - 4.2 billion IPv4 addresses on the Internet. 64 bits for each ip requires 32 GB RAM!

### Frequency Estimation instead of exact counts

### Concept

In most cases, a "good" **estimate**  $\hat{f}_i$  of the true frequency  $f_i$  of item i is sufficient

What is a **good** estimate?

- ullet  $\hat{f}_i$  should be very close to  $f_i$
- lacktriangle With high probability,  $\hat{f}_i pprox f_i$  in most cases

# Review Hashing

- ▶ A hash function h maps from a universe U of size n to p bins [0, p-1] in O(1) time
- As n>>p, there can be **collisions** such that  $x\neq y$  and h(x)=h(y)
- ▶ A "good" hash function h guarantees, that for all  $x \neq y \in U$ , we have Pr[h(x) = h(y)] = 1/p

# Review Universal Hashing

A universal hash family H allows generating hash functions  $h \in H$  such that Pr[h(x) = h(y)] = 1/p

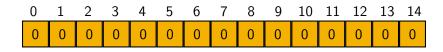
One popular universal hash family is the Linear congruential generator. Let  $h \in H$  be defined as

$$h(x) = ((ax + b) \mod q) \mod p$$

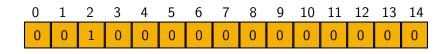
with  $a \neq 0$  and q being a prime number larger than p and a,b are **random** integers mod q, then H is an universal hash family

- 1  $h(x) = ((13698x + 13060) \mod 2147483647) \mod p$
- 2  $h(x) = ((48271x + 8943) \mod 2147483647) \mod p$
- 3  $h(x) = ((458x + 45322) \mod 2147483647) \mod p$

3532 521 31 3532 75 2 542 323 6436463 6545 56 4...

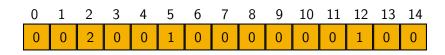


3532 521 31 3532 75 2 542 323 6436463 6545 56 4... 
$$h(3532) = 2$$

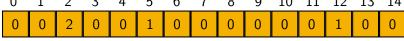


3532 521 31 3532 75 2 542 323 6436463 6545 56 4... 
$$h(31) = 12$$

3532 521 31 3532 75 2 542 323 6436463 6545 56 4... 
$$h(3532) = 2$$

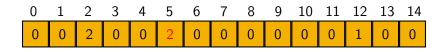


3532 521 31 3532 **75** 2 542 323 6436463 6545 56 4... 
$$h(75) = \mathbf{5}$$



3532 521 31 3532 75 2 542 323 6436463 6545 56 4... 
$$h(75) = 5$$
 
$$h(521) = 5$$

$$h(75) = 5$$
  
 $h(521) = 5$ 



Both frequency estimates for 75 and 521 are now 2 even though we only saw them once!

#### Problem

Hash collisions cause frequency counters for multiple items to "overlap" and return incorrect results.

How often does this happen?

What can we do about it?

#### Problem

Hash collisions cause frequency counters for multiple items to "overlap" and return incorrect results.

How often does this happen?

With probability 1/p

What can we do about it?

#### Problem

Hash collisions cause frequency counters for multiple items to "overlap" and return incorrect results.

How often does this happen?

#### What can we do about it?

 Make the hash table larger to decrease the collision probability will give better frequency estimates. (Smaller error)

#### **Problem**

Hash collisions cause frequency counters for multiple items to "overlap" and return incorrect results.

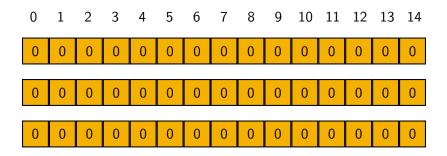
How often does this happen?

#### What can we do about it?

- Make the hash table larger to decrease the collision probability will give better frequency estimates. (Smaller error)
- Use multiple hash tables and hash functions to improve confidence in the estimate.

### Count-Min Sketch

Let  $p=\lceil e/\epsilon \rceil$  and  $d=\lceil \log_e \frac{1}{\delta} \rceil$ , generate d hash functions and hash tables of length p. For example, d=3 and p=15:



Hash items into d hash tables  $D_0$ ,  $D_1$ ,  $D_2$ , Then the frequency estimate of item i is  $\hat{f}_i = \min\{D_0[h_0(i)], D_1[h_1(i)], D_2[h_2(i)]\}.$ 

### Count-Min Sketch

Given  $p = \lceil e/\epsilon \rceil$  and  $d = \lceil \log_e \frac{1}{\delta} \rceil$ , it is guaranteed that after seeing N items with probability  $1 - \delta$ :

$$f_i \le \hat{f}_i \le f_i + \epsilon N$$

### Example

With  $\epsilon=1/10$  Million,  $\delta=0.05$ , then  $p=\lceil e/\epsilon \rceil=27.182.818$  and  $d=\lceil \log_e \frac{1}{\delta} \rceil=3$ , after seeing N=1 billion items, the estimate  $\hat{f}_i$  is within  $\epsilon N=100$  of  $f_i$  with probability 0.95.

Space usage:  $m \times d \times \log_2 n$  bits  $\approx 300$  MB.

### Runtime and Space Complexity

#### Cardinality Estimation

- ▶ Update Time: Compute one hash and update one bucket  $\rightarrow O(1)$  time.
- ▶ Estimation Time: Average over all q buckets  $\rightarrow O(q)$  time.
- ▶ Space:  $q \log \log p$  bits.

#### Frequency estimation

- ▶ Update Time: Compute d hashes and update d buckets  $\rightarrow O(d)$  time.
- ▶ Estimation Time: Take minimum of d buckets  $\rightarrow O(d)$  time.
- ▶ Space:  $p \times d \times \log_2 n$  bits.

### Summary

#### Sketches...

- allow space efficient processing of large data sets by utilizing summarization
- provide theoretical guaranteed estimates of cardinality of a set and item frequencies
- are practical and widely used in industry
- very simple to implement ( $\approx 100$  lines of code)

Hashing is a powerful tool with many applications.