Bloom filters & HyperLogLog

DAT470/DIT065 Computational techniques for large-scale data

2024-04-29

Matti Karppa

First problem of the day

- **Set membership:** Is the given object an element of the set (in the dataset)?
- Potential applications:
  - Is the given word in some forbidden list (known compromised passwords, banned terms in some application?)
  - Is the given object stored in the database?
  - Is the URL known to be malicious?
  - And many others
- We want a data structure that can represent the members of a **set** and that supports
  - **Inserting** new elements into the set
  - **Querying** whether an object is an element of the set
  - Support for regular set operations (union, intersection, difference, symmetric difference)

Python set

- Python sets support all these operations
- Very convenient and flexible data structure
- Implemented as a hash table
- Pros
  - Stores all values explicitly
  - We always get correct answers
  - Amortized constant query times
- Cons
  - Since all values are stored explicitly, we need a lot of space if the universe, or the elements, or the sets are large

```
A = { 1, 2 }
B = { 1, 2, 4, 6, 8 }
C = { 2, 3, 5, 7 }

A.add(3) # insertion

assert 1 in A # membership
assert 1 not in C

D = A | B # union
assert D == { 1, 2, 3, 4, 6, 8 }

E = A & B # intersection
assert E == { 1, 2 }

F = C - A # difference
assert E = { 5, 7 }

G = C ^ A # symmetric difference
assert G = { 1, 5, 7 }
```

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

Bit vectors

- Suppose we know in advance that our **universe** U consists of n elements
- We can then bijectively map each element to an integer in $[n]$, so $U = \{u_1, u_2, \dots, u_n\}$
- Any subset of the universe can thus be represented as a **bit vector** of length n
- Let $x \in \{0,1\}^n$ be such a vector corresponding to some subset $S \subseteq U$
- We then have that $x_i = 1$ if and only if $u_i \in S$
- For small values of n this is efficient: we can simply take sufficiently large integers and apply bitwise operations
- Suppose $x, y \in \{0,1\}^n$ correspond to sets $S, T \subseteq U$, and a and b are sufficiently long integers encoding x and y, then
  - a | b corresponds to $S \cup T$
  - a & b corresponds to $S \cap T$
  - a & ~b corresponds to $S \setminus T$
  - a ^ b corresponds to $S \triangle T = (S \setminus T) \cup (T \setminus S)$
- Unfortunately, this is undoable if $|U|$ is very large, or if we don't know $|U|$

Bit vector example

- Suppose $|U| = 8$, so any subset can be represented with 8 bits
- The set $S = \{2,3,6\}$ could then be represented as

0	0	1	0	0	1	1	0

- The set $T = \{1,2,4,5,7\}$ could be represented as

0	1	0	1	1	0	1	1

- These are often convenient to denote using **hexadecimal** notation, since each group of four binary digits corresponds to a single hex digit

Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

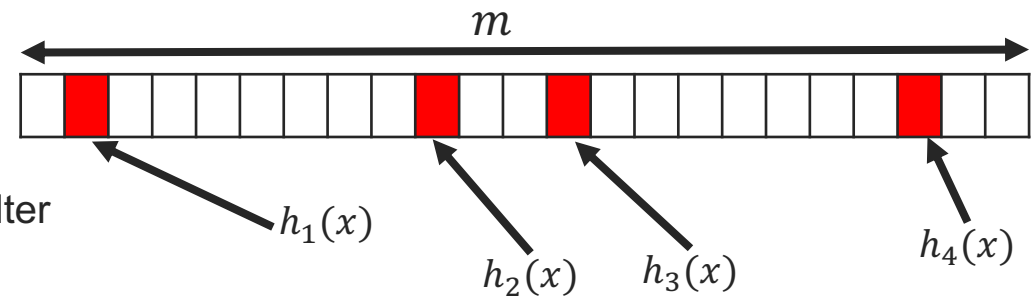
- The sets S and T could then be written as `0x26` and `0x5B`, respectively

Bit vector example

- $S = \{2,3,6\} \simeq$ `0x26`, $T = \{1,2,4,5,7\} \simeq$ `0x5B`
- **Idiom:** membership query becomes asking whether a bit is set
- `((x >> i) & 1) == 1` if and only if bit `i` is set (0-based indexing)
- $3 \in S$, so `((0x26 >> 2) & 1) == 1`, but $3 \notin T$, so `((0x5B >> 2) & 1) == 0`
- Adding an element to the set is equivalent to setting the corresponding bit: `x |= (1 << i)`
- Union: $S \cup T = \{1,2,3,4,5,6,7\}$, so `(0x26 | 0x5B) == 0x7F`
- Intersection: $S \cap T = \{2\}$, so `(0x26 & 0x5B) == 0x02`
- Difference: $S \setminus T = \{3,6\}$, so `(0x26 & ~0x5B) == 0x24`
- Symmetric difference: $S \triangle T = \{1,3,4,5,6,7\}$, so `(0x26 ^ 0x5B) == 0x7D`

Bloom filters

- A **Bloom filter** is a bit vector of specified length m that is filled with prespecified k hash functions $h_1, h_2, \dots, h_k : U \to [m]$ that can give **probabilistic answers** to the set membership problem



- Initially, all elements of the filter are set 0
- **Insertion:** inserting element $x \in U$ into the filter
  - Compute $j_1 \leftarrow h_1(x), j_2 \leftarrow h_2(x), \dots, j_k \leftarrow h_k(x)$
  - Set each bit $j_1, j_2, \dots, j_k$ to 1
- **Query:** asking whether element $x \in U$ is a member of the set
  - Compute $j_1 \leftarrow h_1(x), j_2 \leftarrow h_2(x), \dots, j_k \leftarrow h_k(x)$
  - Check if all bits $j_1, j_2, \dots, j_k$ are 1
  - If even one of the bits is 0, return **definitely not a member**
  - If all bits are 1, return **probably is a member**
- Note that the filter allows for **one-sided error**: it can return false positives, but it can never return false negatives

Bloom filter example

- Let us fix $m = 11$ and $k = 3$
- Initialize the filter to all zeros

0	0	0	0	0	0	0	0	0	0	0	0

- Adding the first element x: suppose we get the following values for the hash functions:

$$h_1(x) = 3, h_2(x) = 5, h_3(x) = 11$$

Bloom filter example

- Let us fix $m = 11$ and $k = 3$
- Initialize the filter to all zeros
- Adding the first element x: suppose we get the following values for the hash functions:

$$h_1(x) = 3, h_2(x) = 5, h_3(x) = 11$$

Bloom filter example

- Let us fix $m = 11$ and $k = 3$
- Initialize the filter to all zeros

0	0	1	0	1	0	0	0	0	0	1	0

- Adding the first element x: suppose we get the following values for the hash functions:

$$h_1(x) = 3, h_2(x) = 5, h_3(x) = 11$$

- Adding the second element y: suppose we get the following values for the hash functions:

$$h_1(y) = 5, h_2(y) = 6, h_3(y) = 9$$

Bloom filter example

- Let us fix $m = 11$ and $k = 3$
- Initialize the filter to all zeros

- Adding the first element x: suppose we get the following values for the hash functions:

$$h_1(x) = 3, h_2(x) = 5, h_3(x) = 11$$

- Adding the second element y: suppose we get the following values for the hash functions:

$$h_1(y) = 5, h_2(y) = 6, h_3(y) = 9$$

Bloom filter example

- Let us fix $m = 11$ and $k = 3$
- Initialize the filter to all zeros

0	0	1	0	1	1	0	0	1	0	1	0

- Adding the first element x: suppose we get the following values for the hash functions:

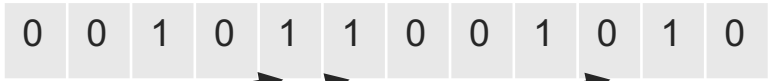
$$h_1(x) = 3, h_2(x) = 5, h_3(x) = 11$$

- Adding the second element y: suppose we get the following values for the hash functions:

$$h_1(y) = 5, h_2(y) = 6, h_3(y) = 9$$

- Since the bit 5 was already set, it the second update didn't affect it

Bloom filter example

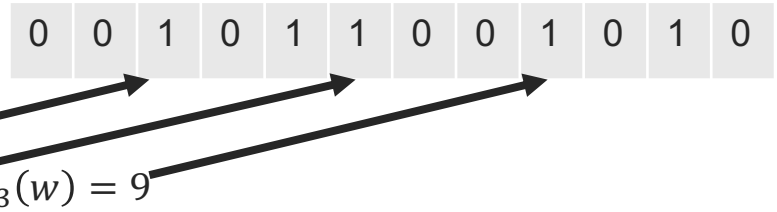


- Querying for element z: suppose we get the following values for the hash functions:

$$h_1(z) = 5, h_2(z) = 6, h_3(z) = 10$$

- We conclude that z is not in the set

Bloom filter example



- Querying for element w: suppose we get the following values for the hash functions:

$$h_1(w) = 3, h_2(w) = 6, h_3(w) = 9$$

- We conclude that w probably is in the set
- This may be a false positive
- In this case, this **is** a false positive since (3,6,9) does not match any of the elements we inserted because the hash functions are deterministic and always yield the same values for the same elements

Probability of false positives

- Bloom filters can be analyzed in the balls and bins model: inserting n elements into a Bloom filter of length m with k hash functions amounts to throwing nk balls into m bins (assuming random hash functions)
- By the previous balls and bins argument, the probability that any given bin is empty is

$$p = \left(1 - \frac{1}{m}\right)^{nk} \approx e^{-\frac{nk}{m}}$$

- Assuming (technically incorrectly, but sufficiently accurately in practice) that p fraction of the bits are 0 in the filter after all n elements have been hashed into the filter, we get for any hash function that it maps coincidentally to a 1 with probability $1 - \left(1 - \frac{1}{m}\right)^{nk}$
- Since the hash functions are independent, this gives the joint probability for **all** hash functions mapping to a 1 coincidentally

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right)^k \approx \left(1 - e^{-\frac{nk}{m}}\right)^k = (1-p)^k = f$$

Optimal choice for *k*

- Denote $g = k \ln\left(1 - e^{-\frac{nk}{m}}\right)$, so $e^g = f$
- Then $\frac{\mathrm{d}g}{\mathrm{d}k} = \ln\left(1 - e^{-\frac{nk}{m}}\right) + \frac{nk}{m} \cdot \frac{e^{-\frac{nk}{m}}}{\left(1 - e^{-\frac{nk}{m}}\right)}$
- Substituting $k = \ln 2 \cdot \frac{m}{n}$ yields 0 for the derivative, and this is in fact a global minimum
- The optimal probability for false positives is thus approximately

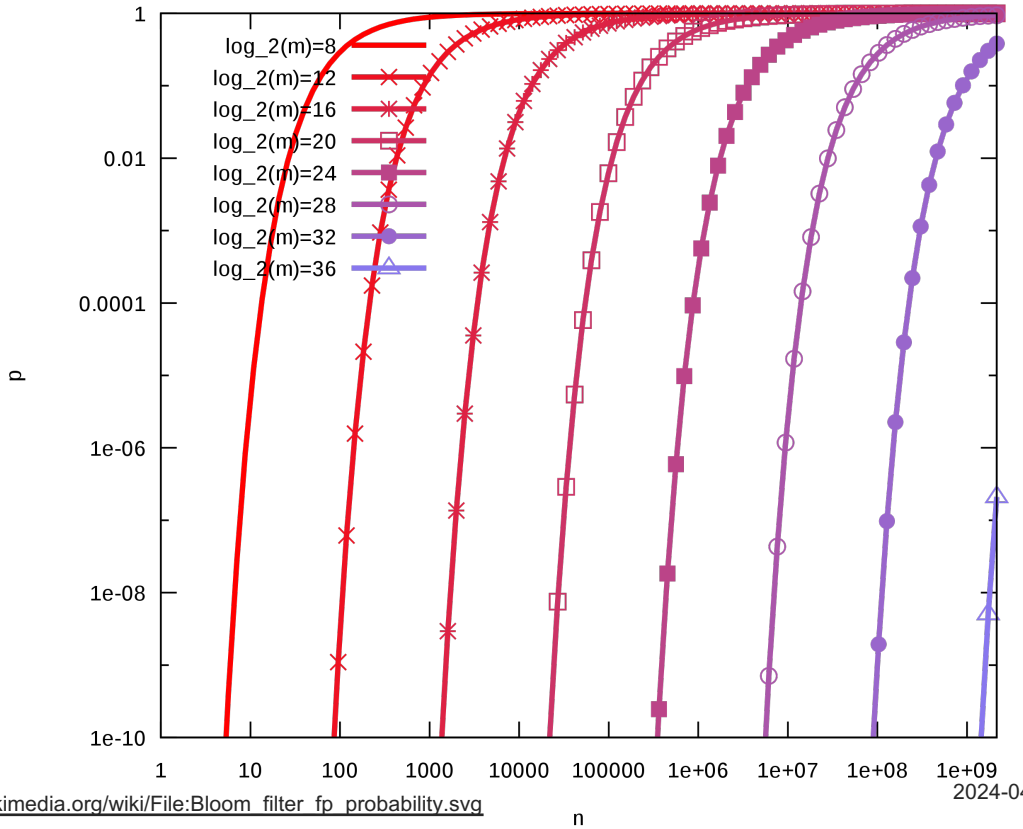
$$f = (1-p)^k = \left(1 - e^{-\frac{nk}{m}}\right)^k = \left(2^{-\ln 2}\right)^{\frac{m}{n}} \approx 0.6185^{\frac{m}{n}}$$

Optimal load for a Bloom filter

- Remember that we defined
$$p = e^{-\frac{nk}{m}}$$
- Solving this for k, we get
$$k = (-\ln p) \cdot \frac{m}{n}$$
- For the false positive probability, we defined
$$f = (1-p)^k$$
- Substituting k, we get
$$f = (1-p)^{(-\ln p)\cdot\frac{m}{n}} = \left(e^{-(\ln(1-p))(\ln p)}\right)^{\frac{m}{n}}$$
- This expression has an obvious minimum (by symmetry) at $p = \frac{1}{2}$
- This, in fact, minimizes f
- We can also verify this by substituting $k = \ln 2 \cdot \frac{m}{n}$ into the expression for p

Length of the Bloom filter

- Bloom filters work the best when they are sufficiently long: in the optimal case, the fraction of zero bits is $p = \frac{1}{2}$, so the filter looks like a random bit string
- If the filter is too short, that is, $p \to 0$, we always get a (false) positive answer to membership query
- To the right: the probability of false positives (denoted *p* here) as function of the number of elements *n* and filter size *m*



https://commons.wikimedia.org/wiki/File:Bloom_filter_fp_probability.svg

Properties of the Bloom filter

- Bloom filters of the same length and same hash functions can be merged: union is just a bitwise or, and intersection is just a bitwise and
- Union is **lossless**: the result of taking an or of two Bloom filters is guaranteed to yield the same Bloom filter as if constructed from the union of the sets
- Intersection is **not**: there may be extra ones, so a greater probability of false positives than when constructing the filter directly
  - Consider $S = \{1,2,3\}$ and $T = \{2,3,4\}$
  - Obviously $S \cap T = \{2,3\}$
  - However, what if 1 and 4 both hash to a bit i while 2 and 3 do not?
  - Taking the intersection of the bloom filters thus has bit i set even though it wouldn't be, had the filter been constructed directly from $\{2,3\}$
- The error can be bounded by the worse of the two filters, though
- These properties enable distributed computation of the filters

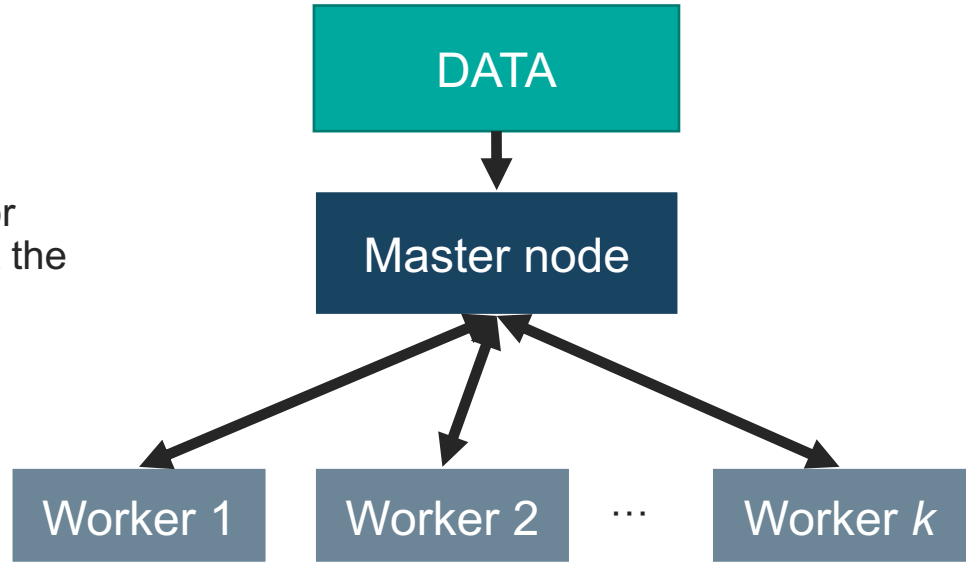
Finding duplicates in Python

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

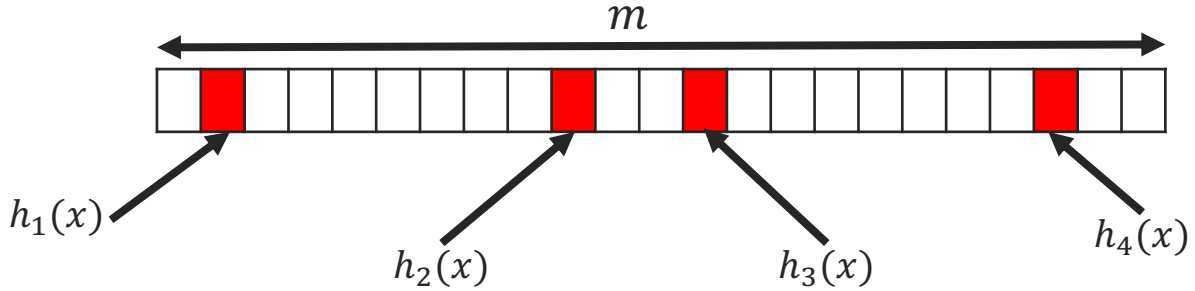
```
from pybloom_live import BloomFilter

f = BloomFilter(capacity=len(X), error_rate=p)
for x in X:
    if x in f:
        print(x, 'is duplicate')
    else:
        f.add(x)
```

Parallelizing Bloom filters

- Trivial to distribute data
- Nodes can compute filters themselves
- If we are interested in constructing the filter for the union, the workers just need to send back the filter (which is small), and the master node computes the union



Cache behavior of Bloom filters



- If m is sufficiently small such that the entire filter fits in cache, everything is good
- If m is very large, then each bit that needs to be set during insertion is probably going to be in a different cache line, both in time and space, so we expect $\Theta(k)$ cache misses for each insertion
- Assuming optimal length of the filter, the probability that a bit is 0 is $\frac{1}{2}$, so the expected number of cache misses for a query is $1 + \frac{1}{2} + \frac{1}{4} + \cdots = 2$ until we find a 0, at which we can abort the query
- Cache behavior of insertions can be improved by **blocking**: instead of one very long Bloom filter, store cache-line-length Bloom filters and use the first hash value to select the block and store bits then within the block; this requires only 1 cache miss per insertion, at the cost of requiring approximately 32% more space to achieve similar false positive rate (see Putze, Sanders, Singler: “Cache-, Hash- and Space-Efficient Bloom Filters” (2007))

Streaming algorithms

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Suppose our input comes in as a **stream**
  - We can inspect one element at a time, but after we decide what to do with it, we must discard it
  - We cannot seek the stream (we have no random access)
  - This corresponds to a big data environment where there is a constant supply of new data
- Once we discard an element, it is lost forever
- We only have a small amount of auxiliary memory that we can use use (in relation to the amount of data in the stream)
- This is the mental model for the next problem

Second problem of the day

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Suppose we are given a stream of data that may contain multiple **repetitions** of the same element

- We are tasked to determine the **number** of **distinct** objects, that is, the **cardinality** of the set of the objects (and we may treat the input sequence as a multiset)
- $n = |\{$  $\}| = 6$
- For the remainder of this lecture, we will refer to the cardinality of interest as n (unless otherwise noted)
- An unrelated riddle: what constant is encoded in the sequence of Lego bricks above?

Applications

Applications of cardinality estimation include:

- Network monitoring (anomaly detection: worms, port scan, DDoS)
- Distinct users in a network stream (content popularity)
- Distinct search queries on Google
- Distinct motifs in DNA sequence
- Distinct elements in a sensor network

Design considerations

A good algorithm and the associated data structure for the problem at hand needs to satisfy several properties, such as:

- **(Near-)linear runtime**: the sequences can originate from very long streams that cannot be stored or rewinded, and there is little time per element (ideally, we want to do a constant amount of work per element)
- **Strictly sublinear space**: the number of distinct elements may be prohibitively large to store all of them
- **Mergeability**: it may be necessary to run the algorithm on multiple (sub)streams in parallel and be able to **efficiently** merge the data structures to get an estimate on the overall cardinality (**sublinear** communication)
- **Idempotence**: we want the data structure to be the same regardless of the number of insertions of the element
- **Commutativity**: we want the data structure to be the same regardless of the order in which the elements are added

Naïve solution: hash table

1. Initialize a hash set $\mathcal{H}$
2. For each element x in the sequence, insert x into $\mathcal{H}$
3. Return $|\mathcal{H}|$, the size of the set

Naïve cardinality estimation in Python

```
H = set()
for x in X:
    H.add(x) # insertion

len(H) # compute (exact) value

H = H1 | H2 # merge two sets
H1 |= H2    # merge two sets in-place
```

Analysis of the naïve solution

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Running time is **linear**: we perform exactly one update per each with amortized constant amount of work per element
- Data structure size is **linear**: we need to store a copy of each element, which may be prohibitively expensive
- For n elements, we need $\Omega(\log n)$ bits to represent each element, so this amounts to a data structure of size $\Omega(n \log n)$
- Data structure is mergeable, but the amount of communication needed for merging is **linear**: we need to potentially communicate $O(n)$ elements in the set to different nodes and process all elements to produce the merged set
- Data structure is idempotent and commutative
- In short: the data structure is very flexible and provides an exact solution to the problem, but requires potentially a prohibitive amount of space

Sketching

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Since storing all elements is a bad idea, we need to resort to **sketching**
- A sketch is an **approximation** maintained by a streaming algorithm that can be **updated** when we see new elements, and that supports some kind of **query** operation
- The sketch can be substantially smaller than the full set while maintaining a reasonable approximation of the quantity we aim to approximate
- In this case, we'd like to update the sketch with elements we receive, and query for the **cardinality estimate**
- Sketches often make use of randomization
- We've already seen an example of a sketch: the **Bloom filter**

Number of elements in a Bloom filter

- The Bloom filter is a sketch for set membership
- Could we determine how many elements were inserted in the sketch?
- Remember that $p = \left(1 - \frac{1}{m}\right)^{nk}$ is the probability that a fixed element of the filter is zero after n distinct elements have been inserted into the filter
- The filter is rather difficult to analyze because all bits are **dependent**: In the balls and bins model, each insertion amounts to throwing k balls into m bins; knowing that a ball landed in a certain bin makes it less likely that a ball lands in one of the other bins (at extreme if $k = 1$: then we know for certain that no balls landed on other bins)

Number of elements in a Bloom filter

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- We can use **Poissonization** to make the setting independent: instead of throwing balls into bins, we assume that balls arrive at the bins at a rate λ (independently and uniformly at random)
- The number of balls in each bin is then an independent random variable that is Poisson distributed
- If we set $\lambda = \frac{n}{m}$, then the Poisson model provides a very good approximation for throwing n balls into m bins
- For all $j \in [m]$, define thus N_j as the number of balls that arrive (independently at random) in the bin j at rate $\lambda = \frac{nk}{m}$
- Thus, $N_j \sim \text{Poisson}(\lambda)$
- We have $\Pr[N_j = \ell] = \frac{\lambda^\ell e^{-\lambda}}{\ell!}$, so the probability that the element j is empty is thus $\Pr[N_j = 0] = e^{-\lambda} = e^{-\frac{nk}{m}}$, matching the approximation for p from earlier

Number of elements in a Bloom filter

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- So, for $j \in [m]$, let us define the indicator variable $X_j = [\![N_j = 0]\!]$, so X_j is 1 if and only if the corresponding element is zero (no balls hit that bin)
- Note: $X_j \sim \text{Bernoulli}(p)$
- Therefore, $\Pr[X_j = 1] = \Pr[N_j = 0] = e^{-\frac{nk}{m}}$
- Thus, $\mathrm{E}[X_j] = e^{-\frac{nk}{m}}$
- Let $X = \sum_{j=1}^{m} X_j$ be the total number of empty elements
- By linearity of expectation $\mathrm{E}[X] = \sum_{j=1}^{m} \mathrm{E}[X_j] = me^{-\frac{nk}{m}}$
- Denote the observed number of zeros in the filter by V_0 and **assume** we've hit the expected value, so solve for $\hat{n}$ as the estimate

$$V_0 = me^{-\frac{\hat{n}k}{m}}$$
$$\hat{n} = -\frac{m}{k} \ln \frac{V_0}{m}$$

Number of elements in a Bloom filter

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- Using the formula for the estimate $\hat{n}$, we can get a pretty reliable estimate for the number of elements inserted in a Bloom filter
- Suppose a and b are Bloom filters for sets A and B, and let $c = a|b$ (with the same hash functions)
- Then, c is the Bloom filter for $C = A \cup B$
- If $\hat{n}_a$, $\hat{n}_b$, and $\hat{n}_c$ are the cardinality estimates for a, b, and c, we can use the estimates together with the equation $|A \cup B| = |A| + |B| - |A \cap B|$ to derive an estimate for the intersection
- The estimates are good as long as $m \gg n$
- Note that we need $m = \Omega(nk)$ bits for the sketch
- If the sketch becomes too full, then we cannot determine the cardinality (in particular, $\hat{n} \to \infty$ as $V_0 \to 0$)

Linear counting

- We can reduce the number of bits required by setting $k = 1$
- A value of $k > 0$ is only useful if we care about the number of false positives for membership queries
- In the case of counting distinct elements, we only need to record that an element has been inserted into the sketch, without worrying about false positives
- By the same argument, the estimate becomes $\hat{n} = -m \ln \frac{V_0}{m}$
- This reduces the number of bits for the sketch to $m = \Omega(n)$
- In particular, if $m \gg n$, the estimate is essentially just the number of bits set in the bit array
- This method is known as **linear counting**

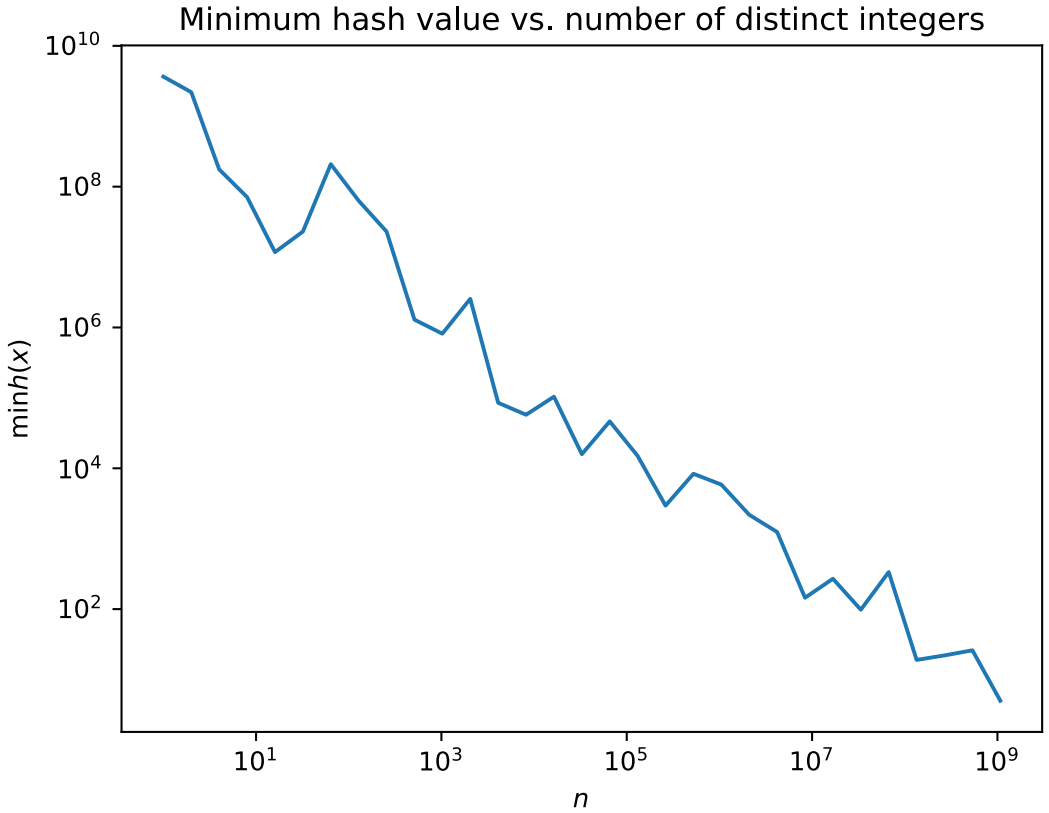
Analysis of linear counting

- Assuming hash the function can be evaluated in $O(1)$ time, runtime is **linear**
- Space usage is $\Omega(n)$ bits, **linear**, which can be prohibitively large for sets with very large cardinalities
- The sketches are mergeable with bitwise OR because they are Bloom filters
- Parallel processing requires the use of the same parameters and hash functions, so merging requires $\Omega(n)$ bits of communication (the entire sketch must be transferred), an improvement from $\Omega(n \log n)$ for hash tables
- The data structure is by its very nature idempotent and commutative

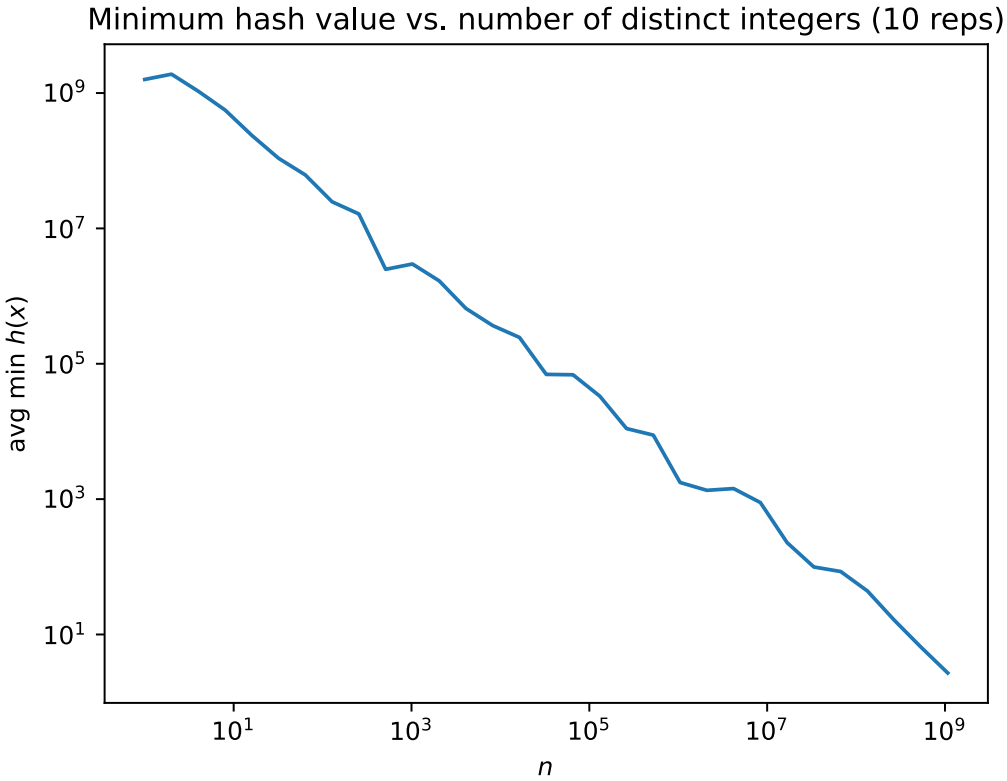
Idea: minimum hash value

- Suppose we hash all values in the dataset and track the **minimum** hash value
- What do we expect to see?

Minimum hash value

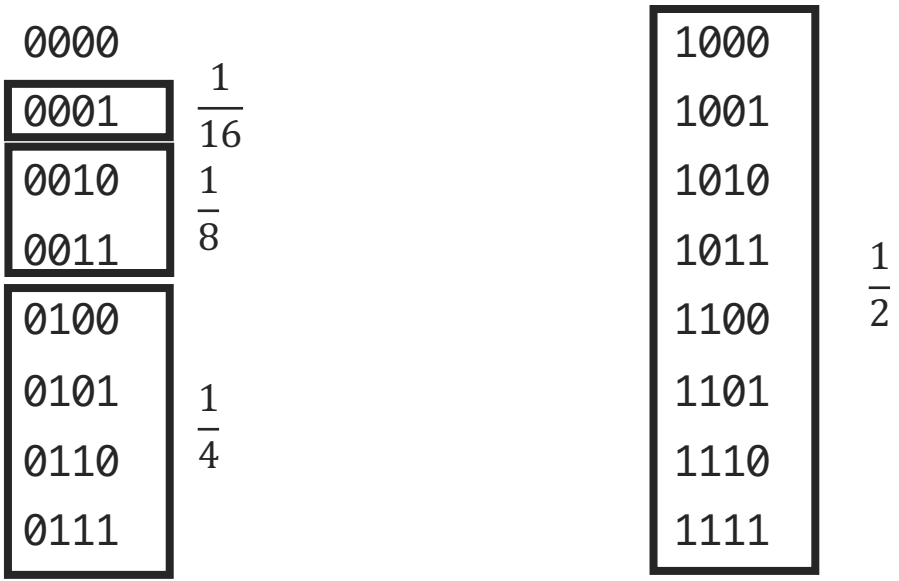


Minimum hash value



Hash values

- Consider the possible 4-bit hash values



The function ρ

- **Observation:** Recording the greatest number of zeros preceeding the first one in the random hash values of the elements in the sequence is an indication of cardinality!
- The probability of witnessing a bit pattern $\underbrace{00\ldots01}_{k-1}$ is 2^{-k}
- This is known as the **geometric distribution**
- We will indicate the position of the leftmost one-bit in a word by $\rho(x)$ such that $\rho(1\cdots) = 1$, $\rho(01\cdots) = 2$, $\rho(001\cdots) = 3$, and so on
- We will leave $\rho(0)$ undefined

HyperLogLog

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- The position of the leftmost 1-bit is a **weak** indication of cardinality
- In order to reduce variance, we will use **another hash function** to split the stream into multiple disjoint substreams, estimate the cardinality of each substream separately, and merge the sketches to form the total cardinality estimate
- The data structure is an array of m **registers** (denoted M)
- Initially, each register $M[j]$ is set to zero
- **Invariant:** $M[j]$ holds the largest value $\rho(x)$ seen among the elements of the jth substream

Adding a new element to the sketch

- On receiving a new element y, use a hash function $f : U \rightarrow [m]$ to select the index (and the matching register) for the substream $j \leftarrow f(y)$
- Map the element to a random hash value with a suitable hash function $h : U \rightarrow [2^{32}]$
- Compute the position of the leftmost one-bit in the hash value and update the register $M[j] \leftarrow \max\{M[j], \rho(h(y))\}$

Cardinality estimate

- The estimate for the cardinality of the elements seen thus far can be obtained by computing the normalized **harmonic mean** of the register values

$$\hat{n} \leftarrow \alpha_m m^2 \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

- Harmonic mean is more resistant to outliers than the arithmetic mean
- **Example**:
  - The arithmetic mean of 10, 10, 10, 10, 460 is 100
  - The harmonic mean of the same numbers is approximately 12.43
- $\alpha_m = \frac{0.7213}{1+\frac{1.079}{m}}$ is a constant for correcting the bias of the estimate
- The full algorithm includes special handling of very small and very large cardinalities that are strongly biased
- Cardinalities whose estimate is up to $\frac{5}{2}m$ are estimated using **linear counting**
- Cardinalities estimated above $\frac{1}{30}2^{32}$ are corrected by estimating

$$\hat{n} \leftarrow -2^{32} \ln\left(1 - \frac{\hat{n}}{2^{32}}\right)$$

Full algorithm

for $j \leftarrow [m]$
  $M[j] \leftarrow 0$
end for
for $y \in Y$ **do**
  $M[f(y)] \leftarrow \max\{M[f(y)], \rho(h(y))\}$
end do

$$\hat{n} \leftarrow \alpha_m m^2 \cdot \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$$

$$V_0 = |\{j : M[j] = 0\}|$$

if $\hat{n} \leq \frac{5}{2} m$ **and** $V_0 > 0$ **then**
  $\hat{n} \leftarrow m \ln\left(\frac{m}{V_0}\right)$
else if $\hat{n} > \frac{1}{30} 2^{32}$ **then**
  $\hat{n} \leftarrow -2^{32} \ln\left(1 - \frac{\hat{n}}{2^{32}}\right)$
end if
return $\hat{n}$

Analysis of the algorithm

- Storing elements naïvely in a hash map requires at least $\Omega(n \log n)$ bits because each element must be represented with $\Omega(\log n)$ bits
- $\Omega(\log n)$ bits are required to be able to distinguish n different elements from one another, so we need at least that many bits for the hash function in all cases
- Linear counting requires $\Theta(n)$ bits
- The m HyperLogLog registers store ρ-values
- Assuming each element is represented by $\Theta(\log n)$ bits, then $\rho(x) = O(\log n)$ for all x, so the registers only need $O(\log \log n)$ bits each, hence the name of the algorithm
- Total space usage is thus $O(m \log \log n)$

Error

- Standard error depends on the number of registers and is approximately $\sigma \approx \frac{1.04}{\sqrt{m}}$
- Concretely, for $m = 1024$, the error is less than 4% in 65% of cases, less than 7% in 95% of cases, and less than 10% in 99% of cases
- Thus, the total space usage for estimating cardinality of up to $n = 10^9$ elements at less than 10% error (with 99% probability) at $m = 1024$ takes $m \log \log n = 1024 \cdot 5 = 5120$ bits, or 640 bytes

Mergeability

- The max function can only increase the register value
- If the same hash functions are used, the same element yields at most the same value for the register in each substream if the stream is processed in parallel
- Given two sketches M_1 and M_2, setting $M[j] = \max\{M_1[j], M_2[j]\}$ for all $j \in [m]$ yields exactly the same sketch as if all elements were inserted directly into $M[j]$
- This means that HyperLogLog is very easy and efficient to parallelize

History and improvements

UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY

- HyperLogLog is a continuation of Flajolet & Martin (1985) probabilistic counting sketch
- In 2003, Flajolet & Durand presented the LogLog algorithm, together with its practical variant, the SuperLogLog, which is the direct predecessor of HyperLogLog
- In 2013, Heule, Nunkesser & Hall at Google presented HyperLogLog++ which provided better space usage & estimation, and was used in practice to estimate the number of distinct Google queries
- In 2017, Xiao, Zhou & Zhen presented the HLL-TailCut that uses even fewer bits
- In 2022, Karppa & Pagh presented HyperLogLogLog, a practical compressed variant of HyperLogLog that maintains mergeability and amortized constant time insertions, and reduces the theoretical space usage to $O\left(\log n + m \log\log\log\log m + m \cdot \frac{\log\log\log m}{\log\log m}\right)$ bits