

Lecture 20: Bloom Filters

*Prof. Eric Price**Scribe: Shivam Gupta, Yan Zheng***NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS**

1 Hash Tables For Sets

As we know from the previous lecture, hash tables give mappings from keys to values. But often we want the mapping of only sets? How to do this more efficiently?

We already know that the hash table implementation for sets requires:

- Already $\mathcal{O}(1)$ look up time
- Randomized, but that's the class
- $\mathcal{O}(n)$ words of space (cannot be constant time).

Note that the word size $w = \log_2 u > \log_2 n$. If we use bit vector to implement sets, it is obvious that u bits are required. But the hash-table implementation for sets only requires $n \log_2 u$ bits. The following question will lead us to prove this.

Suppose we have a set of n items and u items in universe. We can know that

$$\begin{aligned}
 & \binom{u}{n} \text{ possible sets} \\
 & \Rightarrow \text{you need at least } \Omega\left(\log \binom{u}{n}\right) \text{ bits} \\
 & \approx n \log u \text{ bits} \\
 & = n \text{ words}
 \end{aligned}$$

which also implies that hash tables are the optimal implementation for sets, under deterministic circumstances.

2 Bloom Filters

The main purpose of Bloom filters is to build a space-efficient data structure for set membership. We try to think about an approximate set data structure.

- Insertions query(x)

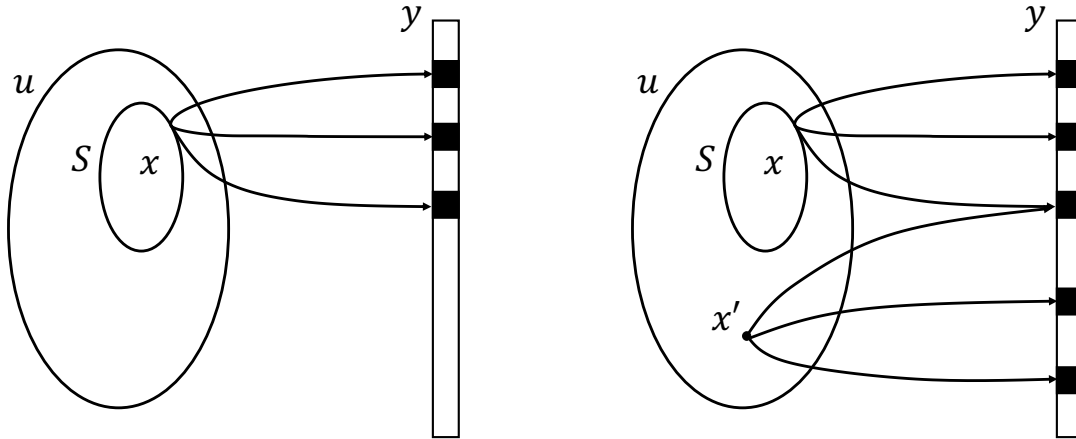


Figure 1: An example of the Bloom Filters

- if $x \in \text{set}$, then answer Yes always. if $x \notin \text{set}$, then answer No w.p. $1 - \delta$.

To control the false positive rate δ , we hope to achieve the $\Theta(n \log \frac{1}{\delta})$ bits.

2.1 Why is this useful

- Chrome list of bad URLs: The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result)
- Database list of keys: The Cascading analytics framework uses Bloom filters to speed up asymmetric joins, where one of the joined data sets is significantly larger than the other (often called Bloom join in the database literature)
- Bitcoin, wallet IDs: Bitcoin uses Bloom filters to speed up wallet synchronization.

2.2 How does it work?

Bloom filter works as follows:

Suppose we have k independent hash functions $h_1(x), h_2(x), \dots, h_k(x)$, $x \in U$, $h_i(x) \in [m]$. (Note that we assume, given each h_i , for any $x \in U$, $h_i(x)$ falls in $[m]$ uniformly.) And initially define $(y_1, y_2, \dots, y_m) = 0^m$. We are going to hash all the n items in set $S \subset U$.

- Start from $y = 0^m$.
- Insert $x \in S$: set $y_{h_i(x)} = 1, \forall i \in [k]$
- Query $x \in U$: If $y_{h_i(x)} = 1, \forall i \in [k]$, then answer Yes. Otherwise answer No.

2.3 How to analyze?

In this algorithm we have n items, m size array, k hash tables. How to set parameters to control the false probability error to be small, and trade off with the running time (corresponds to k) and memory size (corresponds to m).

Suppose we are given m and n , how to optimize k to make fair false positive rate f .

Because h_i are independent, then with n items into filter we have

$$\begin{aligned} Pr[y_i = 1] &= 1 - \left(1 - \frac{1}{m}\right)^{kn} // \text{each of } kn \text{ bits hits each position } i \text{ with equal probability} \\ &\approx 1 - e^{-\frac{kn}{m}} // \text{w.h.p } e^{-\frac{kn}{m}} \pm O\left(\sqrt{\frac{\log n}{n}}\right) \approx e^{-\frac{kn}{m}} \end{aligned}$$

The probability of a false positive is the probability that for $x \notin S$, $y_{h_i(x)} = 1, \forall i \in [k]$, that is

$$\text{false positive rate } f = \left(1 - e^{-\frac{kn}{m}}\right)^k$$

$$\begin{aligned} \arg \min_k \left(1 - e^{-\frac{kn}{m}}\right)^k &= \arg \min_k \log\left(\left(1 - e^{-\frac{kn}{m}}\right)^k\right) = \arg \min_k k \cdot \log\left(1 - e^{-\frac{kn}{m}}\right) \\ \frac{d}{dk} k \cdot \log\left(1 - e^{-\frac{kn}{m}}\right) &\stackrel{z=\frac{kn}{m}}{=} k \cdot \left(\log(1 - e^z) + \frac{z \cdot e^{-z}}{1 - e^{-z}}\right) \end{aligned}$$

when the derivative is 0, we find the optimal $k = \left(\frac{m}{n} \cdot \ln 2\right)$, then the false positive rate is $f = \frac{1}{2^k} = \left(\frac{1}{2}\right)^{\ln 2 \cdot \frac{m}{n}} \approx (0.618)^{\frac{m}{n}}$

So that means for instance, now if we want to store n items, and if we store $m = 8n$ bits, we can get $f \approx (0.618)^8 \approx 0.02$ false positive rate. And this time $k = 8 \cdot \ln 2 \approx 5.5$, we will pick integer 6.

2.4 Compare to hash set

Now let us compare to hash set, take Python implementation for instance. The hash table utilization typically is kept in the range of $\left[\frac{1}{6}, \frac{2}{3}\right]$. And for every entry you store, you need both the key value, key pointer and the list pointer. It mean you need to check if the hash value and the actual value of the item is really the same. For example, to store a string, you have the hash of a string and hash value of a string and then the link list, so this is 3 word/entry.

That means if you get up to $12n$ words and 1 word is 64 bits in machine, we need $12 \cdot 64 = 768$ bits.

3 Supporting Deletions – Counting Bloom Filter

To support deletions, we introduce a variant of the Bloom Filter called the *Counting Bloom Filter*. Instead of storing a single bit at each entry of our hash table, we will store a counter. Then, we will support the following operations:

1. Insert(x):
 - Increment $y_{h_i(x)}$ for every $i \in [k]$
2. Delete(x):
 - Decrement $y_{h_i(x)}$ for every $i \in [k]$
3. Query(x):
 - Return YES if $y_{h_i(x)} \geq 1$ for every $i \in [k]$
 - Return NO otherwise

Now, we will show that it is sufficient to store just 4 bits per entry of the hash table to ensure that the Counting Bloom Filter does not fail for realistic values of m . That is, we will show that $y_j \geq 16$ with small probability for all j .

$$Pr[\text{any given } y_j \geq t] \leq \binom{nk}{t} \frac{1}{m} \leq \left(\frac{enk}{mt}\right)^t = \left(\frac{e \log 2}{t}\right)^t$$

since we chose $k = \frac{m}{n} \log 2$.

Plugging in $t = 16$, we have

$$Pr[\text{any given } y_j \geq 16] \leq 1.4 \times 10^{-15}$$

Thus, by union bound,

$$Pr[\text{any } y_j \text{ is at least } 16] \leq 1.4 \times 10^{-15} \times m$$

In realistic scenarios, $m \leq 10^{10}$, and this gives very low probability of failure.

4 Reviewing Count-Min Sketch

Count-Min Sketch is a data structure for estimating a vector that is updated in a streaming manner. Let x be the vector of length n that we are trying to retrieve information about. At time t , we will receive an update of the form $x_{i_t} \leftarrow x_{i_t} + a_t$. We want to estimate x_i for every i .

We will maintain k hash tables y_1, \dots, y_k with m entries each, and we will hash each update we see once to each hash table (instead of hashing k times to a single hash table as in Bloom Filters). Let $h_j : [n] \rightarrow [m]$ be hash functions for $j \in [k]$. For update at time t of the form $x_{i_t} \leftarrow x_{i_t} + a_t$, we will let $y_{j, h_j(i_t)} \leftarrow y_{j, h_j(i_t)} + a_t$ for every $j \in [k]$.

At the end of the stream, note that for every $i : h_j(i) = u$,

$$y_{j,u} = \sum_{i': h_j(i')=u} x_{i'} \geq x_i$$

So, we will let our estimate of x_i be

$$\hat{x}_i := \min_{j \in [k]} y_{j, h_j(i)}$$

Note that $\hat{x}_i \geq x_i$ by above. So, we just need to show that it is not much larger than x_i with good probability. So, now for any j ,

$$E[y_{j,h_j(i)} - x_i] = \sum_{i' \neq i} \frac{x_{i'}}{m}$$

Thus, by Markov's inequality,

$$Pr[y_{j,h_j(i)} \geq x_i + 2 \sum_{i' \neq i} \frac{x_{i'}}{m}] \leq \frac{1}{2}$$

Thus, if h_j 's are $O(\log n)$ -wise independent,

$$Pr[\hat{x}_i \geq x_i + 2 \sum_{i' \neq i} \frac{x_{i'}}{m}] = Pr[\min_{j \in [k]} y_{j,h_j(i)} \geq x_i + 2 \sum_{i' \neq i} \frac{x_{i'}}{m}] \leq \frac{1}{2^k} \leq \frac{1}{n^2}$$

for $k = 2 \log n$.

5 Count Sketch

The setting is the same as Count Min Sketch. Let $h_j : [n] \rightarrow [m]$ and $s_j : [n] \rightarrow \{-1, +1\}$ be hash functions for each $j \in [k]$. Again, we will maintain k hash tables y_1, \dots, y_k with m entries each. Let h_j be $O(\log n)$ -wise independent and s_j be 4-wise independent. For each update $x_{i_t} \leftarrow x_{i_t} + a_t$, we will let $y_{j,h_j(i_t)} \leftarrow y_{j,h_j(i_t)} + s_j(i_t)a_t$. Now, at the end of the stream, for a fixed $i \in [n]$,

$$E[s_j(i)y_{j,h_j(i)}] = E[s_j(i) \sum_{i': h_j(i')=h_j(i)} s_j(i')x_{i'}] = E[x_i + s_j(i) \sum_{i' \neq i: h_j(i')=h_j(i)} s_j(i')x_{i'}] = x_i$$

since $E[s_j(i')] = 0$. Also,

$$E[s_j(i)^2 y_{j,h_j(i)}^2] = E[y_{j,h_j(i)}^2] = E[\sum_{i': h_j(i')=h_j(i)} x_{i'}^2] = x_i^2 + \sum_{i' \neq i} \frac{x_{i'}^2}{m}$$

Thus,

$$E[(s_j(i)y_{j,h_j(i)} - x_i)^2] = \sum_{i' \neq i} \frac{x_{i'}^2}{m}$$

So, letting our estimator \hat{x}_i be the median $s_j(i)y_{j,h_j(i)}$ gives us a good estimate of x_i in l_2 sense.

6 Power Law Distribution

Realistic data often follows or approximates the power law distribution, where $x_i = i^{-\alpha}$ for some $\alpha \in [\frac{1}{2}, \frac{3}{2}]$. In this case, a few entries of x are large and the remaining are relatively small. We can often obtain better concentration bounds in terms of just the small elements if we assume that the data follows a power law distribution.

References

- [B70] Bloom, Burton H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*. 13 (7): 422–426
- [MR95] Rajeev Motwani; Prabhakar Raghavan. Randomized Algorithm. *Cambridge University Press*. 0-521-47465-5, 1995.
- [M01] Michael Mitzenmacher. Compressed Bloom Filters. *Proceedings of ACM PODC*. 2001.
- [GM05] Cormode, Graham; S. Muthukrishnan. An Improved Data Stream Summary: The Count-Min Sketch and its Applications. *J. Algorithms*. 55: 29–38.
- [CCC04] Moses Charikar, Kevin C. Chen, Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.* 312(1): 3-15 (2004)