CS 388R: Randomized Algorithms Fall 2015

Lecture 11 — October 7, 2015

Prof. Eric Price *Scribe: Jimmy Xin Lin*

1 Overview

In the last lecture, we talk about perfect hashing and mechanism of constructing minimal pefect hash function (MPHF).

In this lecture, we introduce Bloom Filters with its characterized Approximate Membership Queries. Then we describe a variant of bloom filter, the Counting Bloom Filter, as a deletion-support filter. We also take an in-depth investigation to Count-Min Sketch algorithm, as well as Count Sketch algorithm.

2 Hash Tables For Sets

As we know from the previous lecture, hash tables give mappings from a set of keys to a collection of values. But can we use hash tables for sets? To implement sets, the operations we need include insert/delete/build, membership query (is x in the set S) and cardinality query ($|S|$). The hash table implementation for sets requires:

- $\mathcal{O}(1)$ per operation time.
- $\mathcal{O}(n)$ 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.

Question: How many bits are necessary to store a set of n items in $[u]$? Hint: how many sets of size n are there?

$$\begin{aligned}
\log \binom{u}{n} &\geq \log(\frac{u}{n})^n \\
&= n \log(\frac{u}{n}) \\
&= n \log u - n \log n \\
&\gtrsim n \log u
\end{aligned}$$

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

3 Bloom Filters

Now we start to discuss whether we can further improve the space cost by introducing randomness.

3.1 How does it work?

A Bloom filter is a space-efficient probabilistic data structure, which employ a bit vector to keep track of the memebership of a set. The construction of a (regular) bloom filter with k different independent hash functions is as follows: for each element x of a given set S, it is hashed into k positions, and the bit vector of these corresponding positions flip to 1. Formally:

$$\begin{cases} \forall i,\ h_i(x) = 1, & \text{if } x \in S \\ \forall i,\ h_i(x) \text{ keep unchanged.} & \text{if } x \notin S \end{cases}$$

An arbitrary element y is said to be a member of the set S if and only if the responses of all hash function are positive, that is,

$$\forall i \in [k],\ h_i(y) = 1 \implies y \in S$$

However, this membership rule can be faulty if all hashed positions of an element $w \notin S$ happen to be flipped to 1s by members of S. Also note that removal of an element is not possible for a regular bloom filter.

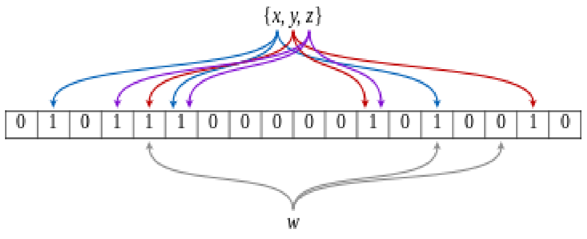

Figure 1: A Simple Bloom Filter

Queries to bloom filters can be characterized as Approximate Membership Queries.

Definition 1 (Approximate Membership Queries)**.** *Given a query x and a set S, the approximate memebership query returns as follows:*

If $x \in S$, answer YES.
If $x \notin S$, usually answer NO.

With the undeterministic membership queries, there could exist some false positive cases: *an queried element x is reported to be in the set S but in reality it is not.* Bloom Filters only employ $\mathcal{O}(n \log \frac{1}{f})$ bits. If we set the false positive rate as $f = 2\%$, it will cause approximately 10% improvement, comparing to the deterministic hash-table implementation.

3.2 Examples

Below is a list of examples to show the usage of bloom filters in real practices.

Example 1 (Chrome Browser [BFChrome])**.** 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.

Example 2 (Databases)**.** Bloom filters are used to reduce the disk lookups for non-existent rows or columns. Keys of bloom filters are stored in PAM, full content is restored on disk.

Example 3 (BitCoins [BFBitcoins])**.** Bitcoins implementation uses Bloom filters to speed up wallet synchronization, which updates a list of wallets and transaction IDs users care about.

3.3 Analysis

Now we turn to analytics over the false positive rate f.

Let n be the number of items, m be the number of buckets, k be the number of hash functions, p be the fraction of 0s in an array, and f be the false positive rate. Then we have

$$\mathbb{E}[p] = (1 - \frac{1}{m})^{kn} \approx e^{\frac{-kn}{m}}$$

Although we derive $\mathbb{E}[p]$, p may vary. So the question is how much does it vary.

$$\begin{aligned} \mathbb{E}\,\# &= P_n \text{ with } Pr = 1 - \delta \\ \# &= P_n \pm \sqrt{n} \log \frac{1}{\delta} \\ &= (1 \pm \mathcal{O}(1)) P_n \end{aligned}$$

With the $E[p]$, we have the false positive rate to be

$$\begin{aligned} f &= (1 - p)^k \\ &\approx (1 - e^{-\frac{kn}{m}}) \end{aligned}$$

Now our objective is to choose k to minimize f.

$$\begin{aligned} &\arg\min_k \left(1 - e^{\frac{-kn}{m}}\right)^k \\ =&\arg\min_k \left(1 - e^{\frac{-kn}{m}}\right)^{\frac{-kn}{m}} \\ =&\frac{m}{n} \arg\min_z \left(1 - e^{-z}\right)^z \end{aligned}$$

where $z = \frac{kn}{m}$. Apply logarithm and then derivative to the above objective,

$$\begin{aligned} g(z) &= z \ln(1 - e^{-z}) \\ g'(z) &= \ln(1 - e^{-z}) + z \frac{e^{-z}}{(1 - e^{-z})} \end{aligned}$$

Set $g'(z) = 0$, we have $z = \ln 2$, and then $k = \frac{m}{n} \ln 2$. Thus,

$$p = e^{\frac{-kn}{m}} = \frac{1}{2} \quad \longrightarrow \text{ bloom filter is balanced}$$
$$f = (1-p)^k = \big(\frac{1}{2}\big)^{\frac{m}{n}\ln 2} \approx (0.618)^{\frac{m}{n}}$$

For example, if $m = 8n$ bits, then $f \approx 0.02$.

3.4 Counting Bloom Filters

Now we discuss one particular variant of the bloom filter, named *Counting Bloom Filters*. In a Counting Bloom Filter, the removal of an element is plausible.

Two operations need to be explicitly specified: insert(x) and membership(x).

Algorithm 1 Counting Bloom Filters

```
1: procedure INSERT($x$)
2:     for $i \in [k]$ do
3:         $y_{h_i(x)}$ increments by 1.

1: procedure MEMBERSHIP($x$)
2:     if $y_{h_i(x)} = 1$, $\forall i \in [k]$ then
3:         return YES.
4:     else optimize $m$, $k$ such that $Pr(\text{YES}) = f$ is small.
```

What is the chance that the given counting bloom filter does not work? Let y_j be the number of items X, $i \in [k]$, s.t. $h_i(x) = j$. Now we throw nk balls into m bins.

$$\begin{aligned} Pr[y_j \geq t] &\leq \binom{nk}{t} \cdot \frac{1}{m^t} \\ &\leq \big(\frac{enk}{mt}\big)^t \\ &\leq \big(\frac{e \ln 2}{t}\big)^t \end{aligned}$$

If we employ 4 bits for each entry (set $t = 16$), then $Pr[y_j \geq t] = 1.4 \times 10^{-15}$.

In other words, $Pr[\text{any } y_j \text{ overflows given 4 bits}] \leq m \times 1.4 \times 10^{-15}$.

4 Count-Min Sketch

Now we introduce a new sublinear space data structure – the Count-Min Sketch (CMS) [CG05], for dealing with the scenario that an given element could appear multiple times. So for CMS, we have streams of insertions, deletions, and queries of how many times a element could have appeared. If the number is always positive, it is called Turnstile Model. For example, in a music party, you will see lots of people come in and leave, and you want to know what happens inside. But you do not want to store every thing happened inside, you want to store it more efficiently.

One application of CMS might be you scanning over a corpus of a lib. There are a bunch of URLs you have seen. There are huge number of URLs. You cannot remember all URLs you see. But you want to estimate the query about how many times you saw the same URLs. What we can do is to store a set of counting bloom filters. Because a URL can appear multiple times, how would you estimate the query given the set of counting bloom filter?

We can take the minimal of all hashed counters to estimate the occurrence of a particular URL. Specifically:

$$\text{query}(x) = \min_i y_{h_i(x)}$$

Note that the previous analysis about the overflow of counting bloom filters does work.

Then there is a question of how accurate the query is? Let $C(x)$ be the real count of an individual item x. One simple bound of accuracy can be

$$\begin{aligned}\forall i, \mathbb{E}[y_{h_i(x)} - C(x)] &= \frac{\big(||C||_1 - C(x)\big) \cdot k}{m} \\ &\leq \frac{||C||_2 \ln 2}{n}\end{aligned}$$

which tells us the average error for all single hashed places with regard to the real occurrence. So we know that it is always overestimated.

For the total number of items $\sum_x C(x)$, we have ($C(x)$ is non-negative)

$$\sum_x C(x) = ||C||_1$$

where, in general, $\forall z \in \mathbb{R}^d$, $||z||_1 = \sum_{i=1}^d |z_i|$, $||z||_2 = \sum_{i=1}^d z_i^2$, $||z||_\infty = \max_{i\in[1,d]} |z_i|$.

Note that you do not even need m to be larger than n. If you have a huge number of items, you can choose m to be very small (m can be millions for billions of URLs).

Now we have bound of occurrence estimation for each individual i in expectation. However, what we really need to concern is the query result. We know that

$$\forall i, k,\ y_{h_i(x)} - C(x) \leq 2||C||_1 \frac{k}{m} \quad \text{w.p. } \frac{1}{2}$$

And now if I choose $k = \log \frac{1}{\delta}$,

$$\min y_{h_i(x)} - C(x) \leq 2||C||_1 \frac{k}{m} \quad \text{w.p. } 1 - \delta$$

If $C(x)$ is concentrated in a few elements, the t^{th} largest is proportional to roughly $\frac{1}{t^\alpha}$ with the power law distribution. So if we choose m to be small, then you can estimate the top URLs pretty well.

$$\sum C(x) = \frac{m^2}{k} = \mathcal{O}(1)$$

In fact, you can show a better result for CMS, which is rather than having your norm depend on the 1-norm. There could be a few elements having all of occurrences. For example, Google.com has a lot of people visiting it. The top few URLs have almost all the occurrences. Then probabaly for a given URL, it might collide some of them in some of the time. But probably one of them is not gonna collide, and probably most of them are going to collide. So you can actually get in terms of l-1 norm but in terms of l-1 after dropping the top k elements. So given billions of URLs, you can drop the top ones and get l-1 norm for the residual URLs.

$$\sum_{\text{non-top } k} C(x) \approx \frac{1}{m/k} = \frac{k}{m}$$

5 Count Sketch

There is another sketch algorithm, which is called Count Sketch [CCF02]. It is basically like CMS, except that when you do hashing, you also associate the sum with each hash function h. Formally,

$$y_j = \sum_{(i,x):h_i(x)=j} C(x)S_i(x)$$

Then the query can be defined as

$$\text{query}(x) = \text{median } S_i(x)y_{h_i(x)}$$

The error can be converted from l-1 norm to l-2 norm.

$$error^2 \lesssim \frac{||C_{\frac{m}{k}}||_2^2}{\frac{m}{k}}$$

On top of that, suppose everything else is 0, then $y_{h_i(x)} \approx S_i(x)C(x)$. So we will have

$$\text{query}(x) \approx \text{median } \big(S_i(x)\big)^2 C(x)$$

Then if there is nothing special going on, the query result would be $C(x)$.

References

[MR] Rajeev Motwani, Prabhakar Raghavan Randomized Algorithms. *Cambridge University Press*, 0-521-47465-5, 1995.

[BFChrome] Yakunin, Alex. Alex Yakunin's blog: Nice Bloom filter application. *http://blog.alexyakunin.com/2010/03/nice-bloom-filter-application.html.*

[BFBitcoins] Bitcoins Developer Guide: Bloom Filters. *https://bitcoin.org/en/developer-guidebloom-filters.*

[CG05] Cormode, Graham, and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1), 58-75, 2005.

[CCF02] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding Frequent Items in Data Streams. *ICALP*, 55(1), 2002.