CMPSCI 590s Systems for Data Science **Fall 2017**

Lecture 13

Lecturer: Emery Berger *Scribe: Ananya Ganesh, Nikhil George Titus*

13.1 Bloom Filters

Bloom filters are data structures that can handle a large number or items efficiently in less space. Bloom filters can represent n elements in a set with m bits where m ¡¡ n. They answer inexact set membership queries in the following way:

ADD(item, S)
ISMEMBER(x,S)
  if $x \in S$ then report TRUE
  but may also return MAYBE

That is, it might return true positives as well as false positives, but will never return false negatives.

13.1.1 1-bit Bloom filters

This is a simple implementation where the filter has one bit b which can be 0 or 1. A function h(x) returns 0 if x is even and 1 if x is odd. When an element is added through ADD(x,S), b is set to h(x) or b. When a membership query ISMEMBER(x,S) is made, the filter returns MAYBE if the bits are the same.

For example, let the bit b initially be 0. The following queries are made:

```
ADD(20, S)
ADD(40, S)
ISMEMBER(20, S) --- returns MAYBE, true positive
ISMEMBER(60, S) --- returns MAYBE, false positive
ISMEMBER(51, S) --- returns NO, true negative.
```

13.1.2 16-bit Bloom filters

The hash function is defined as h(x)=x%16. So for the query ADD(51, S), bit 3 if flipped to 1. Similarly, ADD(40, S) flips bit 8 and ADD(20, S) flips bit 4. Now, ISMEMBER(17) returns false as bit 1 is 0, but ISMEMBER(35) returns MAYBE, which is a false positive.

However, these provide good performance in practice, as with k good hash functions, the false positive rate is only $(1 - e^{-kn/m})^k$. With 2 hash functions and a 1000 elements, this is a rate of 0.1%.

13.1.3 Variations

Bloom filters do well with insertions and lookups, but can't handle deletions very well. One strategy is to use Counting bloom filters, which consists of n bit counts instead of a single bit. The count is incremented

whenever an element is added and decremented when an element is deleted (covered with an example in the next lecture).

Another related problem is answering set cardinality queries: finding how many items of a particular type exist in a set, like counting the number of unique visitors to a website. This is utilized in applications such as click fraud detection.

13.2 Hyperloglog

13.2.1 Introduction

Hyperloglog is a probabilistic data structure that is used to determine the cardinality of a set. We sacrifice some accuracy for reduced storage. It is used in big data systems like Redis. The cardinality of a set is denoted by $|S|$.

We can use hyperloglog to find information like the number of unique visitors to a site. We can also use multiple hyperloglogs to find more complex join queries using set property:

$|A \cap B| = |A| + |B| - |A \cup B|$

This can be used to answer queries like the number of unique android devices visiting a site.

13.2.2 Probabilistic counting

The main idea behind hyperloglog is based on the paper Probabilistic Counting Algorithms for Data Base Applications by Flajolet and Martin. We assume that we have a hash function that uniquely distributed data between 0 to 2^{k-1}.

Now, If we begin counting the largest sequence of 0 bits at the start of the hashed value. We can observe the following:

- The probability of getting a 0 in the first place is 1/2.
- The probability of getting 00 in the first place is 1/4.
- Similarly if we see a sequence of k zeros we can assume that we have seen 2^k elements.

13.2.3 LogLog and HyperLogLog improvements

We can observe that the above approach has high variance and this can be further improved. Hyperloglog uses bucketing where the initial bits of the hash function is used to choose a bucket and the other bits to count the number of leading zeros. Then the count across all the buckets is averaged. This will give us k and we can conclude that there are 2^k elements.

Further improvements to hyperloglog was made by throwing out 30% of buckets with larger values and averaging out the remaining 70% of the buckets. The error was reduced to $1.04/\sqrt{m}$, where m is the number of buckets.

13.3 Mongo DB

Mongo DB is a NoSQL database which is fast. It sacrifices consistency guarantees to achieve high speeds. It keeps data in memory. Mongo DB uses JSON to define schemas. Example of JSON:

```
{"key":"value","key2":{"key":"value"}}
```

MongoDB uses binary encoded format of JSON called BSON. It also inserts unique ids to JSON fields to make searching possible. We can say that MongoDB is collection oriented.

Sample API: If we have an entity called posts. We can insert using: posts.insert(),posts.insertMany().

Mongo DB is partially lazy in loading data and it scales well.