Bloom Filters

Joaquim Madeira

Version 0.1 – November 2017

Overview

- Motivation
- Hash Tables – A quick review
- Hash Functions – A quick review
- Bloom Filters
- Counting Bloom Filters

Set Membership

- Given an arbitrary sized string s and a set S
- Does s belong to S ?
- Easy answer for small sets !
  - Complexity ?
- BUT “difficult” answer for huge sets !
  - E.g., Big-Data applications

Hash Tables

- Data structure for storing key-value pairs
- No ordering !!
- BUT, fast access !!
- No duplicate keys !!

Hash Tables

- Two main operations :

- Insert (put) a key-value pair into the table
  - If key already exists, update the value

- Search for (get) the value associated with a given key

Hash Tables

- Additional operations :

- contains(key)
- delete(key)
- is_empty()
- Keys iterator
- …

Hashing

- To reference key-value pairs stored in a table
- Perform arithmetic operations that transform search keys into array indices
  - FAST !!
- Ideally, different keys would map to different indices
- BUT, collisions do occur !!

Hash Tables – Toy Example

- Download the file hash_table_V_1.py

- Identify the available operations
- Create a table and insert several key-value pairs

- What kind of keys can be used ?
- How are collisions resolved ?
- What operations are missing ?

HashTables – Time complexity

- The time complexity of searches by hashing can be as low as O(1) or as high as O(N)

- Worst-case ?
- Distinct keys $K_i \neq Kj$ collide: $h(K_i) = h(K_j)$
- The entire table must be searched to find the correct entry
- Or to conclude it is not there !

Hash Functions

- Pseudo-random mathematical functions used to compute indices for table look-up
  - Keys are mapped to small integers
- Indices should be evenly distributed
  - Even if there are regularities in the data
- There are many hash functions
  - With different degrees of complexity
  - And with differences in performance
  - For different applications

Simple Hash Functions

- Division method
  - Choose a prime m that isn't close to a power of 2
  - $h(k) = k \bmod m$
  - Works badly for many types of patterns in the input data
- Knuth's variant
  - $h(k) = k(k+3) \bmod m$
  - Supposedly works much better than the raw division method

Simple Hash Functions

```
def hash(astring, tablesize):
    sum = 0
    for pos in range(len(astring)):
        sum = sum + ord(astring[pos])

    return sum%tablesize
```

- Anagrams will be given the same values…

Hash Functions – DJB31MA

```
uint hash(const uchar* s, int len, uint seed)
{
   uint h = seed;
   for (int i=0; i < len; ++i)
       h = 31 * h + s[i];
   return h;
}
```

Non-cryptographic Hash Functions

- Suitable for hash table lookup but not for crytography / secure uses
- Fast computation

- FNV – Fowler-Noll-Vo hash function
- Murmur Hash
  - Multiply and rotate
- …

Universal Hashing

- Issue
  - There always exist keys that are mapped to the same integer / index

- Consider a set of hash functions H
- H is universal (good), if
  - For all keys $0 \leq i \leq j < M$
  - Probability $(h(i) = h(j)) \leq 1 / M$, for h randomly selected from H

Approximate Membership Queries

- Given a set $S = \{x_1, x_2, \dots, xn\}$
- Answer queries of the form: $Is\ y\ in\ S\ ?$

- Data structure should be FAST and SMALL
  - Faster than searching through S
  - Smaller than explicit representation

Approximate Membership Queries

- How to get speed and size improvements ?
- Allow some probability of error !!
- False positives
  - $y \notin S$ but reporting $y \in S$
- False negatives
  - $y \in S$ but reporting $y \notin S$

Bloom Filters

- B. H. Bloom, 1970
- Use hash functions to determine approximate set membership
- Allow for fast set membership tests on very large data sets
- Applications
  - Spell-Checking / Text Analysis
  - Network monitoring
  - …

Application – Spell-Checkers

- Determine if candidate words are members of the set of words in a dictionary
- The Bloom filter should be large enough to allow the inclusion of additional words by the user

Application – Text Analysis

- Find related passages in different reports
- Constructing a Bloom Filter of all the words in each passage
- Computing the normalized dot product of all Bloom filter pairs
- The result of every dot product is a similarity measure

Application – Web-Caching

- Bloom filters are used in WWW caching proxy servers
- Proxy servers intercept requests from clients and either fulfill the requests themselves or re-issue them to servers

Application – Email Spam

- We know 1 billion “good” email addresses
- If an email comes from one of these, it is **NOT** spam
- How check for spam in a FAST way ?

Bloom Filters

- ***Is y in S*** ?

- A Bloom filter
  - Provides an answer in constant time
    - Time to hash
- Uses a small amount of memory space
- BUT, with some small probability of being wrong !

1^{st} – Register the elements of set S

Start with an m bit array, filled with 0s.

B	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash each item x_j in S k times. If $H_i(x_j) = a$, set $B[a] = 1$.

B	0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

[Mitzenmacher]

2nd – Process the queries

To check if y is in S, check B at $H_i(y)$. All k values must be 1.

B																
	0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

Possible to have a false positive; all k values are 1, but y is not in S.

B																
	0	1	0	0	1	0	1	0	0	1	1	1	0	1	1	0

[Mitzenmacher]

Basic operations

- Initialization
  - Clear all cells
- Insertion
  - Compute the values of k hash functions
  - Set the corresponding cells, if needed
  - It takes constant time, but proportional to k

Basic operations

- Membership test
  - Compute the values of k hash functions
  - Check if the corresponding cells have been set
  - If any such cell is not set, the searched element is not a member of the set
- Worst-case ?
- Checking all k cells !
  - Set elements and false positives

Bloom Filter – Simple Demos

- Bloom Filters by Example
  - http://billmill.org/bloomfilter-tutorial/
- Bloom Filters
  - https://www.jasondavies.com/bloomfilter/

Bloom Filters – Toy Example

- Download the file bloom_filter_V_1.py
- Identify the available operations
- Create a Bloom filter and insert several items
- Perform membership tests for various items
  - Belonging and not belonging to the set

Bloom Filters – Behaviour

- Deterministic hash functions !
- No attempt to solve hashing collisions !
- Can we get false negatives ?
- Probability of false positives ?
- How to minimize ?

Bloom Filter – Parameters

- The behaviour of a Bloom filter is determined by four parameters

- n set elements registered in B
- $m = c \times n$ cells in B (i.e., bits)
- k independent, random hash functions
- f is the fraction of cells set to 1

Bloom Filter – Parameters

- How to choose m, the size of the filter ?
- How to choose k, the number of hash functions ?
- How do we choose the best k value ?

Probabilities – After 1 insertion

- Initially all bits are set to zero
- Inserting one element

- What is the probability of $b_i = 1$, after using the first hash function ?
  - Equal probability for any cell

$$P(b_i = 1) = \frac{1}{m}$$

$$P(b_i = 0) = 1 - \frac{1}{m}$$

Probabilities – After 1 insertion

- After computing the k hash functions and setting k cells

$$P(b_i = 0) = \left(1 - \frac{1}{m}\right)^{k}$$

Probabilities – After n insertions

- After inserting all n set elements, by computing each time k hash values
  - Assuming independence

$$P(b_i = 0) = \left(1 - \frac{1}{m}\right)^{k \times n}$$

Probabilities – After n insertions

$$P(b_i = 0) = \left(1 - \frac{1}{m}\right)^{k \times n}$$

$$P(b_i = 1) = 1 - a^k, \qquad a = \left(1 - \frac{1}{m}\right)^n$$

Probability of a false positive

- Testing the membership of an item not in S entails a positive answer
  - Corresponding k bits are set to 1

- The probability of that happening is

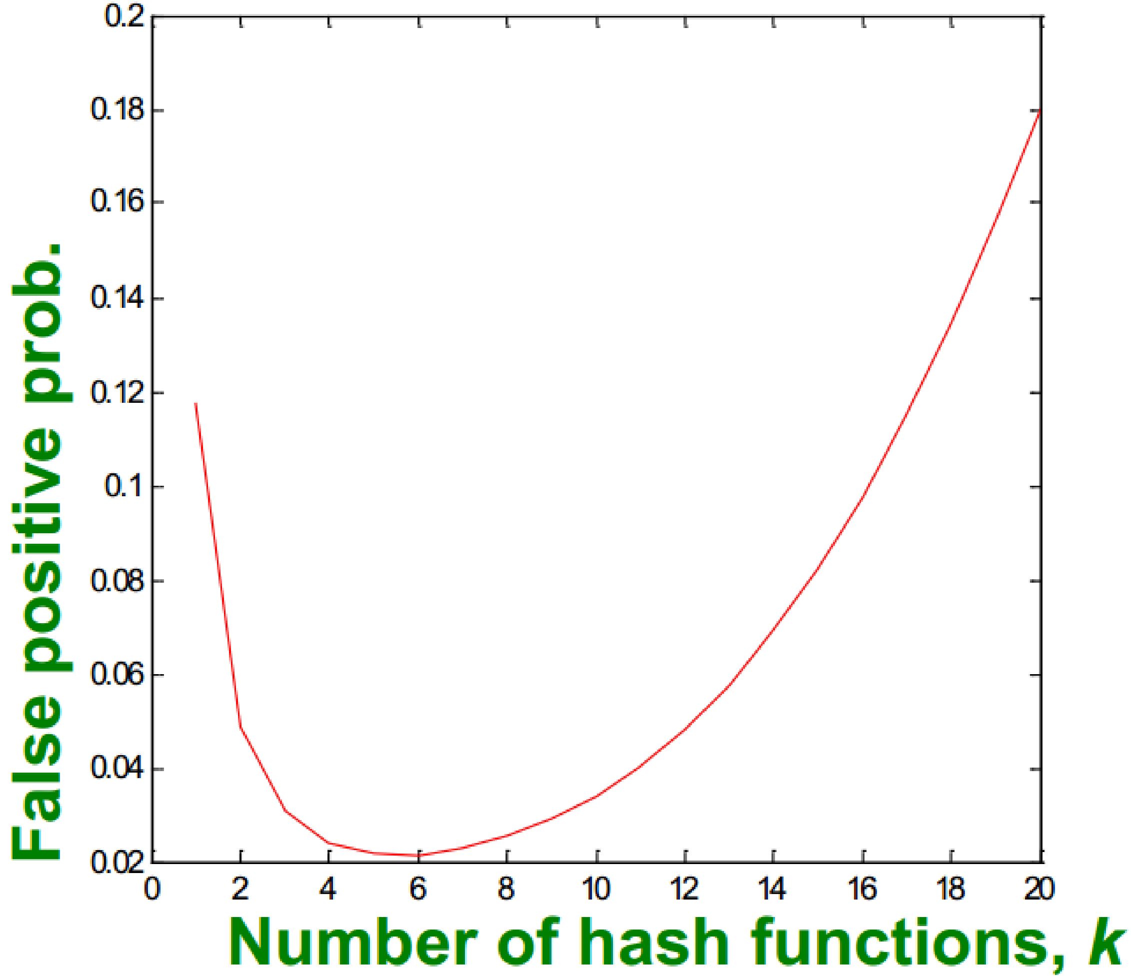
$$p = \left(1 - a^k\right)^k$$

$$p \approx \left(1 - e^{-kn/m}\right)^k$$

Example

- n = 1 billion items, m = 8 billion bits
- k = 1 : $p \approx \left(1 - e^{-1/8}\right) = 0.1175$
- k = 2 : $p \approx \left(1 - e^{-2/8}\right)^2 = 0.0493$
- What happens as we keep increasing k ?

Optimal value of k



Optimal value of k

- To determine the value of k that minimizes p we minimize log p, which is more tractable
- And get

$$k_{opt} \approx \frac{m}{n} \times \ln 2 \approx 0.693 \times \frac{m}{n}$$

- Use the closest integer to k_{opt}
- For the previous example : $k_{opt} \approx 5{,}54 \approx 6$

Which Hash Functions ?

- No need to use cryptographic hash functions !
- You can simulate k hash functions by simply combining two hash functions
  - Kirsch and Mitzenmacher (2006)
- Compute one base hash function on unsigned 64-bit numbers
- Take the upper half and the lower half of that value and return them as *two* 32 bit numbers

Bloom Filters – Toy Example – Tasks

- Carry out computational experiments with different filter parameters (m, n, k)
- Generate a random set of keys and insert pairs key-value
- Perform membership tests
- Analyze the percentage of false positives

Bloom Filters – Wrap-up

- No false negatives and limited memory usage
  - Great for pre-processing before more expensive checks
- Suitable for hardware implementation
  - Hash computations can be parallelized
- Error rate can be decreased by increasing the number of hash functions and allocated memory space

Bloom Filters – Wrap-up

- Useful for applications where an imperfect set membership test can be helpfully applied to a large data set of unknown composition

- Advantage over hash tables is Bloom filter speed and error rate

Bloom Filters – Pending Issues

- Cannot represent multi-sets
  - I.e., sets with repeated elements
- Cannot query the multiplicity of an item
- Deleting an item is not possible !

Counting Bloom Filters

- Multi-set representation
- Now, each filter cell is a w-bit counter
    - w = 4 seems to be enough for most applications

Counting Bloom Filters

- To insert an element, increase the value of each corresponding cell
- Test membership checks if each of the required cells is non-zero

Counting Bloom Filters

- To delete an element, decrease the value of each corresponding cell
- Deletions necessarily introduce false negative errors !!
  - How ?

Counting Bloom Filters

- To retrieve the count of an element :
- Compute its set of counters
- And return the minimum value as a frequency estimate

Counting Bloom Filters

Start with an m bit array, filled with 0s.

B																
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash each item x_j in S k times. If $H_i(x_j) = a$, add 1 to $B[a]$.

B																
	0	3	0	0	1	0	2	0	0	3	2	1	0	2	1	0

[Mitzenmacher]

Counting Bloom Filters

To delete x_j decrement the corresponding counters.

B	0	2	0	0	0	0	2	0	0	3	2	1	0	1	1	0

Can obtain a corresponding Bloom filter by reducing to 0/1.

B	0	1	0	0	1	0	1	0	0	1	1	1	0	0	1	0

[Mitzenmacher]

Counting Bloom Filters – Issues

- Counter overflow
  - No more increments after reaching $2^w - 1$
  - BUT, now we have undercounts !!
- Choice of counter width w
  - A large w diminishes space savings and introduces unused space (many zeros)
  - A small w quickly leads to maximum values
  - Trade-off…

Counting Bloomm Filters in Practice

- If insertions/deletions are rare compared to look-ups
  - Keep a CBF in “off-chip memory”
  - Keep a BF in “on-chip memory”
  - Update the BF when the CBF changes

- Keep space savings of a Bloom filter
- But can deal with deletions
- Popular design for network devices

References

- J. Leskovec, A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*, 2014 – Chapter 4
- B. H. Bloom, Space/Time Trade-offs in Hash Coding with Allowable Errors, *Commun. ACM*, July 1970
- J. Blustein and A. El-Maazaw, Bloom Filters – A Tutorial, Analysis, and Survey, TR CS 2002-10, Dalhousie University, Halifax, NS, Canada, December 2002
- A. Broder and M. Mitzenmacher, Network Applications of Bloom Filters: A Survey, *Internet Mathematics*, Vol. 1, N. 4, 2004

Acknowledgments

- An earlier version of some of these slides was developed by Professor Carlos Bastos

- Part of the slides adapted from original slides of
  - J. Leskovec, A Rajaraman and J. Ullman – Mining of Massive Datasets – www.mmds.org
  - M. Mitzenmacher, Bloom Filters and Such – 2014 Summer School on Hashing, Copenhagen, DK