Performance Analysis of Bloom Filter with Various Hash Functions on Spell Checker

Fauzan Hilmi Ramadhian 13512003
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
fauzan_hilmi@hotmail.com

***Abstract*— One of not-too-widely-known data structure is bloom filter. Bloom filter is a data structure that utilises hash functions to support its two core operations; element addition and element existence checking. Even though it doesn't earn enough fame, bloom filter still widely used in computer industry. The example of application that uses bloom filter is spell checker. An interesting point about bloom filter worth talking about is its dependency to hash functions. In this paper, there will be discussion about the peformance comparison of 8 hash functions on its duty as bloom filter's hash. The bloom filter will be tested on a simple spell checker. In the end, the recommended hash functions will be concluded.**

***Index Terms*—bloom filter, data structure, hash function, spell checker**

I. Introduction

Nowadays, computer technlogy has become a central support of modern life. Hundreds and thousands human activities will not possible if computer didn't exist. Computer is such well logic and complex designed that makes it possible. One of the key element behind the working computers is data structure.

Data structure is a particular way to store and organize a large amount of datas in computer. Why they becomes the key element of computers? Because each computer needs a way to store its gigabytes of data effectively and efficiently, and data structure provides that.

There are so many types of data structure out there. Data structures may exist as simple or complex structure, and they may be available in programming language libraries or not. One of not too-widely-known example of data structure is bloom filter.

Bloom filter is a data structure that supports two basic operations; element addition and element existence checking. In order to perform those operations, bloom filter utilises hash functions. Hash function is a type of function that takes a data input, usually in arbitrary size, and turns it into a completely different value, usually in different length from the input. Hashes are commonly used in computer technology. They serve as hash tables, caches, similarity checking, encryption, and in this case, a support technlogy for bloom filters.

Bloom filter uses hash functions to map the data input into its array upon both the element addition and element existence checking. It can uses any hash functions. Even then, bloom filter may use 1 or more hashes. So, different combinations of hash functions used are possible. Now, an interesting question came to mind; what is the most suitable hash function to be used by the bloom filters? This is the central discussion point of this paper. In this paper, there will be discussion about 8 hash functions that are going to be reviewed in bloom filter. Then, their performance will be tested. The performance here refers to the hashes' average running time and false positive rate. Each one of the hashes functions will be tested on a bloom filter implemented as a simple spell checker. The experiment result will be discussed to conclude hash functions that are recommended and that are not to be used in bloom filters.

II. Theory

A. Hash Function

Hash function is a type of function that takes a group of characters (called key) and maps it to a value of certain length (called hash value, hash codes, hash sums, or simply hash). Hashes are used in various sections of computer world. Primary use of hashes are hash tables which are used to quickly locate a data record by its query key. The hash is also useful for mapping the key to an index; the index locates the place in the hash table where the corresponding record should be stored. Besides that, hashes are benefical in caches, data security, data comparison, and cryptography. There is even a type of hash called cryptographic hash function that is used primarily to convert data keys into irreversible hash values. Last but not least, bloom filters use hash functions too.

There are so many hash functions out there. Some of them will be discussed here. Examples of hash function that are involved in this paper are as follows.

1. MD5

   MD5 is famous cryptographic hash function that maps arbitrary sized data input into 128-bit hash value. It was established by Ronald Rivest in 1991. MD5 has been widely used by people around the world until its security flaw was found not far too long ago.

2. SHA-1

Like MD5, SHA-1 is common used cryptographic hash function that produces 160-byte hash value from arbitrary sized key value. It was designed by United States National Security Agency (NSA) and is a U.S. Federal Information Processing Standard as stated by the United States NIST. This function is mostyly used on security applications like TLS and SSL.

3. CRC-32

CRC (Cyclic Redundancy Check) is hash function that is mainly used to detecting errors caused by accidental raw data changes in digital networks and storage devices. It was invented by W. Wesley Peterson in 1961. Unlike the cryptographic hashes, CRC is reversible. An implementation of CRC is CRC-32 which is widely implemented on ethernet and other standards.

4. Java hashCode

Java hashCode() is a hash function developed and implemented exclusively by and on Java. The hashCode() method is only available for String object. This function takes the product sum algorithm over the entire text of the string to produce the output value.

5. FNV

Fowler-Noll-Vo (FNV) is a non-cryptographic hash function developed by Glenn Fowler, Landon Curt Noll, and Phong Vo. FNV is famous because of its easiness design and thus easy implementation. This function was designed for fast hash table and checksum table.

6. MurmurHash

Not too differenr from FNV, MurmurHash is a non-cryptographic hash function created by Austin Aplleby in 2008. The current version of MurmurHash supports 32-bit and 128-bit hash values. This hash is built primarily for efficent hash-based lookup.

7. Jenkins Hash

Jenkins Hash functions are a collection of non-cryptographic hash functions for multi-byte keys invented by Bob Jenkins. The functions are widely used in checksums to detect accidental data corruption and in database in which the functions determine whether two data records are similar or not. The first Jenkins hash functions, one-at-atime, was published in 1997. Then, Jenkins released new improved functions such as lookup2, lookup3, and the newest one, SpookyHash. The function that is refered as Jenkins hash on this page is lookup3.

8. XXHash

XXHash is a not-yet-widely-known non-cryptographic hash function developed by Pike and Alakuijala. It is commonly used to detecting error in LZ4 Decoder. Besides that, XXHash can be useful in databases, games, and security world.

B. Bloom Filter

Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set or not. The term “probabilistic data structure” means that it can tells that an element is *definitely* not in the set. However, it cannot tells whether an element is *definitely* in the set; It can only tells that an element is *probably* in the set. Thus, false positives are possible but false negatives are not. Bloom filter was invented by Burton Howard Bloom in 1970. The data structure supports two basic operations; add element to the set and check whether the element is in the set or not. Some operations like element deletion, false positive handling, etc can be implemented along with modification of the data structure.

Bloom filter uses bit array or bit vector of m bits as its main base data structure. Also, k different hash functions are used to map some set elements to one of the m array positions with uniform random distribution. First, the bit array is initialized with value 0 on all of its elements. Then, addition operation is conducted by feeding the element to each of k hash functions to get k array positions. These k bits are all set to 1. Meanwhile checking operation is done by feeding the element to each of k hash functions to get k array positions. Then, check whether these k bits are all set to 1 or not. If they are all set, then the element is probably in the set. But if not, then the element is definitely not in the set.

Here are some pictures to describe the operations better. An example of element addition is as follows.

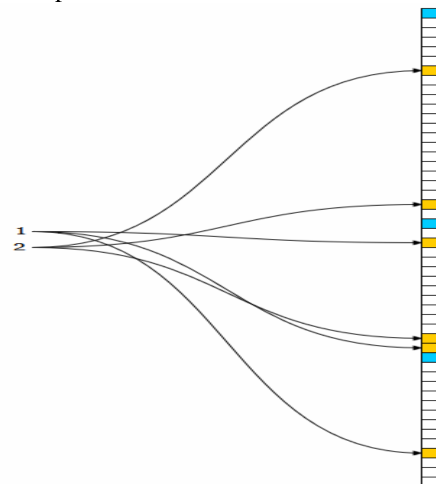


Fig 2.1 Two elements (1 and 2) are added to the empty bloom filter

Now the checking operation example will be presented. First, the positive. There are two possibilities when this happened. Either the element is really exists in the filter, or it isn’t. In that case, it is a case of false positive. An example of true positive result is as follows.

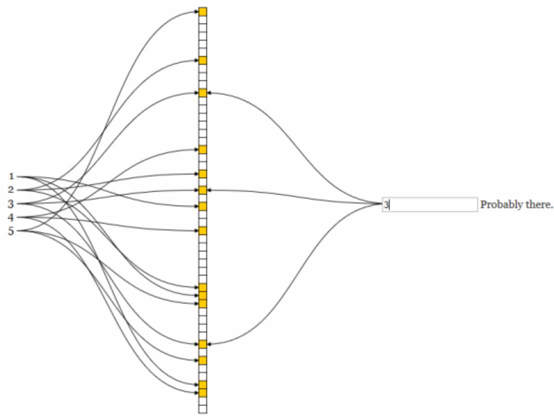


Fig 2.2 The element 3 is really exists and thus reported as true positive

Meanwhile, a case of false positive is given below.

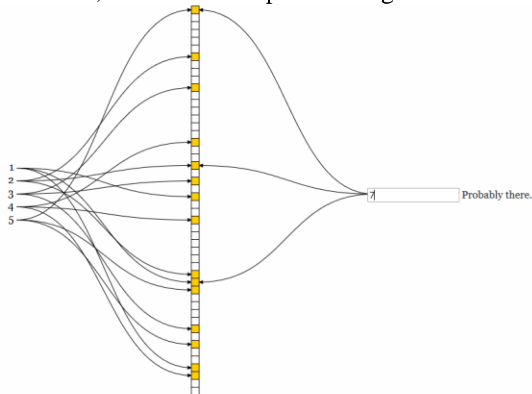


Fig 2.3 The element 7 is actually not exists but reported as positive

Now the negative result will be discussed. There is only one possibility for this result, true negative. Thus, a false negative will never happened.

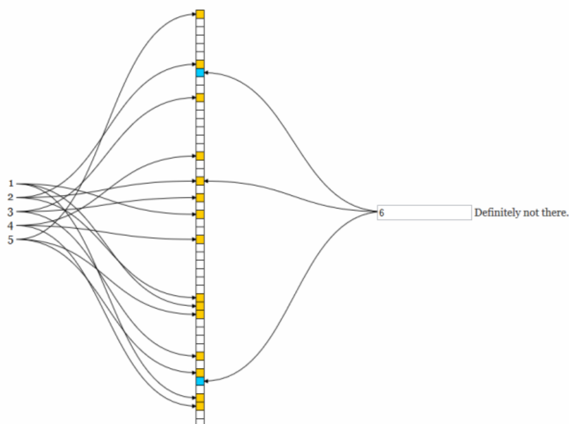


Fig 2.4 Element 6 is checked and it is definitely not in filter.

Compared to its data structure competitors for representing sets such as binary search tree, trie, hash table, etc, Bloom filter have strong advantage. Unlike most of the competitors, bloom filter doesn't store the data on its array. It only stores the hashed bits of elements. Thus, bloom filter is generally faster and needs less storage space. The time needed either to add elements or checking elements is a fixed constant *O(k)*.

The biggest talking point of bloom filter is the false positive. False positives are the major drawback of this data structure. However, the false positives rate can be adjusted as required. The rate can be calculated by the formula below.

$$\left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}.$$

Fig 2.5 The formula for finding *p*

k represented the number of hash functions, *m* is the size of bit array, and *n* is the expected number of elements. Besides this, there are formulas to determine the value of *m* and *k* too. Here are them.

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

Fig 2.6 The formula to determine *m* value

$$k = -\frac{m}{n} \ln 2$$

Fig 2.7 The formula to determine *k* value

Because of its frequently used operations in real world (additon and checking) and its fast, efficient performance, bloom filter is used in many applications. Apart from spell checker, bloom filter is also used in Squid Web Proxy Cache for cache digests, Google Chrome for filtering malicious URLs, Google BigTable and Apache Cassandra for reducing disk lookups, Bitcoin for speeding up wallet synchronization, and many other applications.

C. Spell Checker

Spell checker is an application program that checks the correct spelling of words. Spell checker may be found on its own application or embedded on another applications. Some examples of application that uses spell checker are word processor, email client, digital dictionary, and search engine.

Basically, spell checker do its job as follows. First, the spell cheker scans the text and extracts the words from it. Then, for each word, the checker will performing the lookup operation to its dictionary to check whether the word is there or not. If the word is found, the spell checker will do nothing. But if it isn't found, the checker will show the mistaken word message to the user. Additionally, the checker may find the closest right word and suggest it to the user.

III. IMPLEMENTATION

A. Experiment Setup

All implementation is done on Java. The spell checker implemented here is a simple type one. It basically loads the dictionary from external text file to internal memory. The dictionary is downloaded from http://www-01.sil.org/linguistics/wordlists/english/. Then, the checking operations from the bloom filter are performed.

The bloom filter itself is constructed of an array of byte. The bytes act as bit and thus can only be valued 0 or 1. Then, the bloom filter parameters are as follows. The number of items in the filter (*n)* is the number of words in dictionary, that is 109.583. Each hash function is reviewed once at a time and thus the number of hash function (*k*) is 1. It also concludes that there are going to be 8 bloom filters to be reviewed. Then, using the bloom filter formulas discussed before, the number of bits in filter (*n)* is 208.991. The final value, the false positive rate (*p*) is computed as around 0,4.

The experiment plot is as follows. There are two hashes' performance that will be measured; average running time and false positives rate. The average running time is going to be extracted by adding each word to the bloom filter. Each addition time is measured. Then, all the times are summed and then divided by the number of words to get the average additon time.

Next the false positive rate of each bloom filter is measured. This is performed by checking whether a non-exist word will gives positive result or not. Each "wrong word" are set to be as similar as possible to each word in the dictionary (by adding a charater or removing a character). Each bloom filter is tested by 1.154.336 false words.

B. Source Code

Here wil be written the source code of the bloom filter which is implemented in Java. The source code of the spell checker will not be written as its logic has been discussed on previous section.

The overall code of the bloom filter is as follows.

```
public class BloomFilter {
   private byte[] set;
   private int setsize = 208991; //from Bloom Filter formula
   private String hash;


   public BloomFilter() {

   }

   public BloomFilter(String _hash) {
       set = new byte[1+setsize];
       for(int i=0; i<set.length; i++) {
           set[i] = 0;
       }
       hash = _hash;
   }
```

```
public void Add(String s) throws UnsupportedEncodingException,
NoSuchAlgorithmException, IOException {
       int pos = 0;
       switch(hash) {
          case "MD5" : {
             pos = MD5(s);
             break;
          }
          case "SHA1" : {
             pos = SHA1(s);
             break;
          }
          case "CRC" : {
             pos = CRC(s);
             break;
          }
          case "hashCode" : {
             pos = hashCode(s);
             break;
          }
          case "FNV" : {
             pos = FNV(s);
             break;
          }
          case "Murmur" : {
             pos = Murmur(s);
             break;
          }
          case "Jenkins" : {
             pos = Jenkins(s);
             break;
          }
          case "XXHash" : {
             pos = XXHash(s);
             break;
          }
       }
       set[pos] = 1;
   }

   public boolean check(String s) throws IOException,
UnsupportedEncodingException, NoSuchAlgorithmException {
       int pos = 0;
       switch(hash) {
          case "MD5" : {
             pos = MD5(s);
             break;
          }
          case "SHA1" : {
             pos = SHA1(s);
             break;
          }
          case "CRC" : {
             pos = CRC(s);
             break;
          }
          case "hashCode" : {
             pos = hashCode(s);
             break;
          }
          case "FNV" : {
             pos = FNV(s);
             break;
          }
          case "Murmur" : {
             pos = Murmur(s);
             break;
          }
          case "Jenkins" : {
             pos = Jenkins(s);
             break;
          }
          case "XXHash" : {
             pos = XXHash(s);
             break;
          }
       }
       return (set[pos]==1);
   }
```

IV. RESULT AND ANALYSIS

The result table for average running time experiment is as follows.

Hash	Average Running Time (ns)
MD5	3604
SHA-1	2669
CRC-32	841
Java hashCode	790
FNV	599
MurmurHash	775
Jenkin Hash	832
XXHash	4665

Table 4.1 Average running time experiment results

It can be seen that MD5 and SHA-1 are quite slow compared to other functions. It is because they are crytptographic hash functions; they are not optimized to works fastly as their primary job is to ensure no collissions happened and the key values are not trackable. However, it is a bit surprising to find that XXHash is categorized as slow too. Even it is the slowest one! The possible cause lies on the implementation of the function; the bottleneck is on the XXHash's library chosen as it is execution required several additional steps. The other 4 functions, hashCode, FNV, MurmurHash, and Jenkins Hash performs reasonably. They executes fastest with not much time difference between them. It is very understandable as their function is to computes hash values as fast as possible.

Now, the false positive rate analysis will be discussed. Here is the result of the false positive rate experiment.

Hash	False Positive Rate (%)
MD5	0.3850638
SHA-1	0.3864256
CRC-32	0.3854293
Java hashCode	0.3858365
FNV	0.3836803
MurmurHash	0.3860427
Jenkin Hash	0.3853652
XXHash	0.3864536

Table 4.2 False positive rate experiment results

It is a bit surprising that the performance of cryptographic hash functions (MD5 and SHA-1) is not far off behind the others. Even MD5 was managed to be the second least for the number of false positives! In theory, it should not happened as cryptographic hash functions do not well distributed the hash values. They only concerned on how the hash values are not reversible. There are several possibilites on why this could happen. Maybe it's just a plain of luck that the hash values are distributed uniformly, though it is unlikely. Another possibility lies on maybe broken implementation of the bloom filter. It is not easy to track the causes. Afterall, this experiment shows that cryptographic hash functions can be good bloom filter hash functions too.

Besides the tale of the first two function, it can be seen that FNV is the king of the least false positives. It is quite leaving its competitors quite far off in the front. Then, the CRC, hashCode, and Jenkins Hash performs reasonably. However, MurmurHash and XXHash' bad performance again raises red flag that they should be avoided to be used in bloom filters.

V. CONCLUSION

Bloom filter is a very useful though not very well known data structure. Basically, it supports two basic operations. The operations are element addition and element existence checking. Bloom filter is used in Squid Web Proxy Cache for cache digests, Google Chrome for filtering malicious URLs, Google BigTable and Apache Cassandra for reducing disk lookups, Bitcoin for speeding up wallet synchronization, and many other applications. In this case, bloom filter can be used in spell checker too. The words addition and correct words checking can be done by the filter.

Bloom filter uses hash functions on its implementation. The hashes are used to maps elements to the array. An interesting problem to be discussed is which hash function best suited to bloom filter? Then, an experiment is conducted with 8 hash functions to determine which has the best and worst performance. The performance here refers to the average running time and false positive rate.

The result is as follows. In terms of average running time, MD5 and SHA-1 is the least performing hashes. It is an understandable result as their job as cryptographic hash function makes them not as fast as others. The XXHash result is an anomaly as its implementation is broken in execution time. Meanwhile, the false positive rate results are more surprising. Cryptographic hash functions manage to get a good result relatively to the non-cryptographic. This may be occured because of broken implementation or they just a bit "lucky" the results are distributed uniformly. However, the winner here is FNV as it gets the least false positive rate among the others.

In conclusion, cryptographic hash functions like MD5 and SHA-1 is not good choice for bloom filters. Although they can get good false positive rate in the experiment, their running time is far too long. Looking to the two tables above, the most recommended hash function is FNV. It has the least false positive rate, and its average running time is among the fastest ones.

VII. ACKNOWLEDGMENT

First of all, Author would say thank you to Almighty God because of His mercy and grace Author can finish this paper. Then, Authors also wants to express his thanks to Dr. Ir. Rinaldi Munir, M.T. whose give helpful advices and assistances. Finally, Author want to say thank you to his parents and beloved friends who are always give Author strengths and spirits to pass the struggles during the writing of this paper.

REFERENCES

[1] Burstein, Max. *Creating a Simple Bloom Filter.* *http://www.maxburstein.com/blog/creating-a-simple-bloom-filter/. Accessed on 10 May 2015, 19.58*

[2] Hurst, Thomas. *Bloom Filter Calculator.* http://hur.st/bloomfilter?n=4&p=1.0E-20. Accessed on 10 May 2015, 13.23

[3] Mill, Bill. *Bloom Filters by Example. billmill.org/bloomfilter-tutorial/.* Accessed on 10 May 2015, 13.09

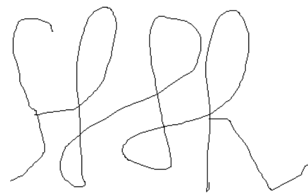
[4] Technopedia, *Hash Function.* http://www.techopedia.com/definition/19744/hash-function. Accessed on 10 May 2015, 12.30

[5] Zhen, Jian. *Benchmarking Bloom Filters and Hash Functions and Go.* http://zhen.org/blog/benchmarking-bloom-filters-and-hash-functions-in-go/. Accessed on 10 May 2015, 14.27

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Mei 2015



Fauzan Hilmi Ramadhian 13512003