Analysis of Hash Algorithms and Performance Comparison

Tong Wu

Department of Electrical and Computer Engineering
Rutgers, the State University of New Jersey
tong.wu96@rutgers.edu

***Abstract*— The paper gives an analysis of hash algorithms and a guidance on how to choose a best suitable hash algorithms for different problems.**

Hash algorithms are widely used in different data structures and application problems. However, there are too many kinds of hash functions. Some of them are designed a long time ago and some of them are new created. They put particular emphasis on different areas. It's hard to choose a proper hash algorithms when we need to use them. In this paper, I will describe the definition of hash algorithm and its data structure. I will also go into the classification and uses of hash algorithms. Last, I will base on the properties of a good hash algorithm, test some of them and compare their performance.

I. INTRODUCTION

Hash algorithm is defined as a correspondence between the recorded key and the storage address of the record. It takes a group of keywords and maps it to a value of a certain length (called a hash value) which presents as the storage address. Hashing is used to locate and retrieve items in a database because its really do the work in a faster manner like to find the item using the shorter hashed key than to find it using the original value.

A. Mathematical Definition

$$Addr = H(key) \tag{1}$$

Key is the keyword which is going to map. Addr is the address corresponding to the key. Ideally, a key is mapped to one single address.

A more detailed equation is:

$$Addr = H(key, structure\_size) \tag{2}$$

The equation presents that the hash function is also related to the size of structure. In order to achieve fast search and lower space complexity, the address size is much smaller than input data size.

Hash function gives us a way to find the address and retrieve the item by a specific way in a quite short time. This way seems jumbled and randomized. But computers know that it actually has a special law. In other words, it is like a opposite of sorting for some simple hash functions. As we all know, sorting makes the records suit a specific order. Hashing just scatters the records in the data structure like hash tables.

B. Hash table

Hash table is a data structure based on hash functions. As we know, array location is the fastest in all of linear data locations. As long as I get the index, I could get the value in $O(1)$ time. Therefore, hash table is based on this idea. It uses hash function to generate the hash code of key. Then divide it by size of array to generate the index. Finally, it uses index to get the value.

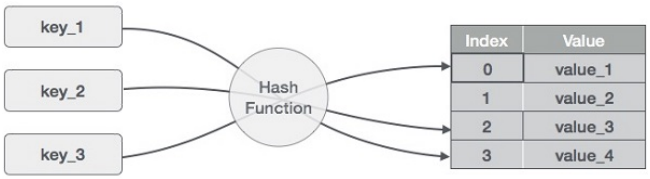

Fig. 1. Hash table

Hash table operations include:

- Insertion
- Deletion
- Search

If there is at most one record in a bucket, all operations above need $O(1)$ time complexity.

C. Collisions

However, things talked before in the paper is all based on an ideal case. Due to size of table is much less than input data, number of inputs is larger than number of buckets in table. In most cases, it is not an one-to-one mapping. When two keys hash to the same address, here comes a collision.

$$H(key1) = H(key2)\ ,\ key1 \neq key2, \tag{3}$$

Due to size of keys group is normally large, it is impossible to create a same-sized group of addresses. Therefore, collisions are unavoidable.

Considering collisions, time complexity of insertion, deletion and search in hash table becomes $O(n/m)$ where n is the number of keys, m is the number of buckets in the table.

Though collision is unavoidable, we can try to decrease the possibility of collision by using following methods:

1) Separate Chaining: Each bucket is independent, and has some sort of list of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation.

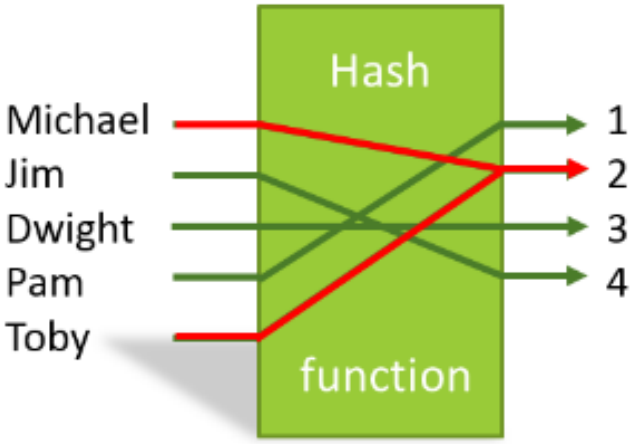

Fig. 2. Collision: Michael and Toby have same address.

2) Open addressing: When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. Some well-known methods include:

- Linear probing
- Quadratic probing
- Double hashing

A good collision resolution can resolve the collision in an efficient way. It is critical to the performance of hash function.

II. Classification

This section introduces some of fundamental hashing methods. Let's see how various hash functions stack up.

A. Additive hash

The idea is to add all of keys to get hash result.

```
static int additiveHash(String key, int prime){
  int hash, i;
  for (hash = key.length(), i = 0; i < key.length()
      ; i++)
      hash += key.charAt(i);
  return (hash % prime);
 }
```

$prime$ is an arbitrary prime number. The hash result is in $[0, prime-1]$. This function takes $5n+3$ instructions and the length of table should be prime.

B. Rotating hash

The rotating hash function using a series of bitwise operations to shuffle input values.

```
static int rotatingHash(String key, int prime)
 {
   int hash, i;
   for (hash=key.length(), i=0; i<key.length(); ++i
       ){
      hash = (hash<<4)^(hash>>28)^key.charAt(i);}
   return (hash % prime);
 }
```

It takes $8n+3$ instructions in this function. The function can also be transformed to:

```
hash = (hash<<5)^(hash>>27)^key.charAt(i);
```

and:

```
hash += key.charAt(i);
        hash += (hash << 10);
        hash ^= (hash >> 6);
```

C. Multiplication hash

The idea is to multiply all of the individual digits of keys and then divide it by size of table.

$$H(key) = (a*b*c*d*....)\%m \tag{4}$$

a,b,c,d are digits of keys, $m$is the size of hash table.

```
static int bernstein(String key)
{
  int hash = 0;
  int i;
  for (i=0; i<key.length(); ++i)
     hash = 33*hash + key.charAt(i);
  return hash;
}
```

The parameter 33 can be replaced by 31, 131 or 1313 and so on.

D. Division hash

$$H(x) = key\%m + 1 \tag{5}$$

In this equation, $+1$ could be deleted if table starts at index 0.

III. Use cases

Hash functions are commonly used in 2 application areas: Searching and Cryptography.

A. Cryptographic hash algorithms

The hash algorithms play an important part on cryptography. Because hash functions can scatter the keys, they are able to cover up passwords, protect secret messages and detect tampering. They all use iterations of some specific hash functions to generate a message digest in a given length. The larger in length, the higher of safety factor.

The process in cryptography contains:

1) Preprocessing: Splitting messages into data blocks and setting default values for hashing.

2) Hash Computation: By using preprocessed data blocks, hash functions and relevant parameters, generate the hash values.

Main applications in cryptography can be divided into:

- File checksums
- Digital signatures
- Authentication protocol

Some well-known hash algorithms used in cryptography include:

MD5: Message-Digest Algorithm 5, which is used to ensure complete and consistent information. The inputs of function are messages in different lengths. Output is a 128-bits digest.

SHA-1: Secure Hash Algorithm is a widely used cryptographic hash algorithms on many security protocols including TLS, SSL, PGP and SSH. Compared to MD5, it uses a larger digest(16-bits). It was designed by NSA in 1995 and thought to be the replacement of MD5. Due to its old generation, it can be decoded easily now. Thus, it is not used in high security areas. But it still plays an important role in public uses like checking the complication of downloaded files.

SHA-2: It is a series algorithms which has same algorithm idea like SHA-1. Instead of being well known by the public, it is more confidential and there is not any successful attacks on it.

- SHA-256: 32-bytes digest
- SHA-384: 48-bytes digest
- SHA-512: 64-bytes digest

B. Searching hash algorithms

Searching hash is a way to find out the storage address of data elements. It has $O(1)$ time complexity which is really attractive. The overall progress includes:

1) Lookup3: The algorithm was created by Bob Jenkins in 2006. It achieves good evenly hash distribution. The two special properties of $lookup3$ are temper-proof and reversibility. However, it takes a little more time than other algorithms.

2) Murmur3: It is a non-encrypted hash algorithm which was created by Austin Appleby in 2008. It is now widely used in many distributed frameworks such as Hadoop. The algorithm is fast and efficient. But it is not seriously even.

3) xxhash: xxHash is used in Boom Filter. It is almost capable of twice the throughput than the default Java implementation on Ubuntu 16.04.

4) FNV-1a: FNV-1a is a open source algorithm which currently comes in 32-, 64-, 128-, 256-, 512-, and 1024-bit flavors. It was designed for fast hash table and checksum use.

IV. What is a good hash function

Please not that those are some of representative and common used hash algorithms. There are much more hash algorithms. In order to test the performance of hash algorithms, we should figure out what is a good hash function. A good hash function should have following properties:

1) Fast Forward: Given the clear text and hash algorithm, hash values can be calculated within finite time and limited resources.

2) Difficult Reverse: Given hash values, clear text are really difficult(almost impossible) to be calculated.

3) Sensitive Input: When input changes a little, the hash values should be changed wildly.

4) Conflict Avoidance: The possibility that a collision happens should be small.

However, these are overall properties to judge a hash algorithm. There is not a perfect hash algorithm which is suitable to all applications. We should put particular emphasis on some of them according to different specific problems.

V. Test Environment

To test the performance, I used Intel Core $i5-5257U$ with 8GB memory and 64 bit build Late 2015 MacBook Pro. The inversion of Java is $1.8.0_144 - b01$.

VI. Benchmarks

A. Performance of Java hash functions

In this implementation, I used both cryptographic hash algorithms and searching algorithms to test the total time on different size of data blocks. The algorithm codes comes from $github$ and other open sources.

Count of 1K blocks	FNV1a	FNV1a-64	Murmur3A	Murmur3F	MD5	SHA-1
	Times[ms]					
100000	131	123	52	37	342	506
1000000	1307	1232	465	269	2880	4782
10000000	13136	12306	4587	2609	28513	47749
100000000	132487	124944	45866	26879	276253	497374
	MByte / second					
Average	743	792	2097	3543	315	198

Fig. 3. Performance of hash algorithms on total execution time

The average time in graph:

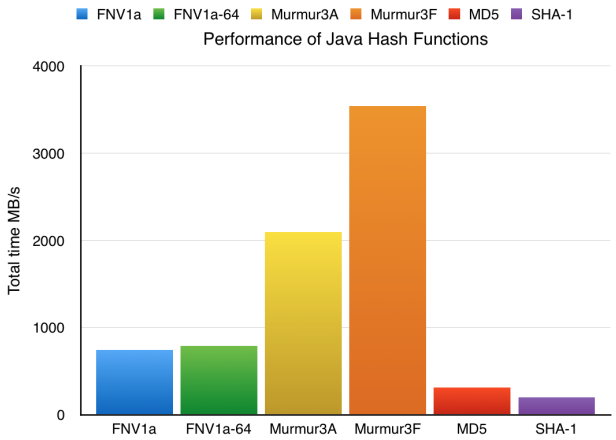

Fig. 4. Performance of hash algorithms on total execution time. We can see that Murmur3F costs the most time and SHA-1 algorithms uses the least time in this implementation. The Cryptography hashes have better performance than Searching hashes.

B. Throughoutput of searching hash algorithms

In order to test the performance of efficiency of different hash algorithms to look up, I used searching hash functions above. In order to compare, some other famous hash algorithms are added in the test.

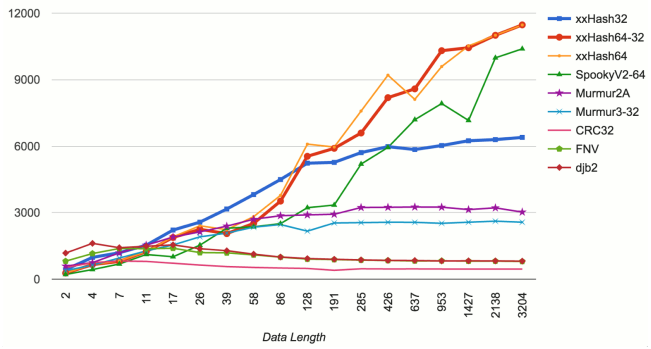

Fig. 5. Performance of hash algorithms on Throughoutput MB/s of large size of data. We can see that in terms of thourghoutput, the xxHash is the king on not too small size of data.

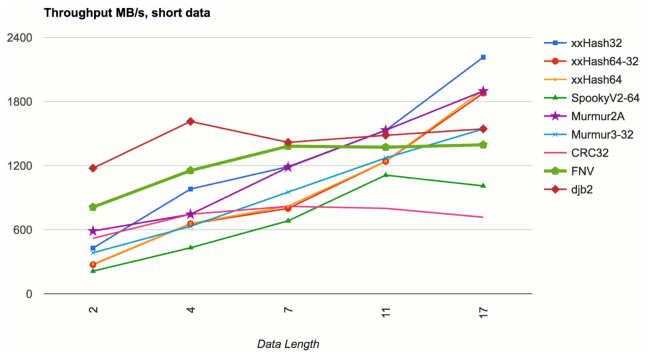

Fig. 6. Performance of hash algorithms on Throughoutput MB/s of small size of data. In terms of small size of data, distances among each hash algorithms are not obvious.

C. Performance on collision avoidance

As we talked about before, a good hash algorithm not only has better performance on time and efficiency, we need to consider the collisions of hash algorithms.

In this test, I used 3 datasets:

- Words: Made up with many English words with 2.3MB total size.
- Filepaths: Made up with many paths of system like "/Users/wt/java/bin". The total is 4.2MB.

Let's see how the collision performance on each hash algorithms:

Hash	Words	Filepaths
	collisions	
xxHash32	6	0
xxHash64	0	0
Murmur2A	11	0
Murmur3	3	1
FNV	5	1

Fig. 7. Performance of hash algorithms on number of collisions. It is obvious xxHash64 has a better performance on 1 to 10 MB size of data.

VII. Future work

In this paper, I tested a limited number of hash algorithms. And the datasets are still small (no larger than 10MB). There are still lots of hash algorithms worth to test. Therefore, in the future, I'm going to test them. If possible, I'd like to expend the size of datasets to GB or even larger, which requires cloud computing technologies and distributed systems.

VIII. Conclusion

In this paper, I introduced the hash algorithms and its data structure–Hash table. From the basic types of hashing methods, I showed some main uses of hash algorithms: Cryptography and Searching. I presented what is a good hash algorithm and what's the properties of a good hash algorithm. In order to find a strategy on how to choose a suitable hash algorithm according to different problems, I did some tests.

The first implementation tested 6 hash algorithms. I found that searching hash algorithms have better performance than cryptographic hashes on total execution time. And SHA-1 is the king.

In terms of performance on throughtoutput, xxHash performs better than others on large size of data.

As for the quality of hash functions–collision avoidance, the xxHash64 won again.

Based on the performance, we can conclude that searching hash algorithms is faster than cryptographic hashes. And xxHash is the best one which has good performance on efficiency and quality. Because of this, it is used in many open source frameworks such as Boom Filter.

References

[1] Wikipedia, Hash Table. From Wikipedia, the free encyclopedia.
[2] Introduction to Algorithms,2nd edition by Thomas H. Coremen, Charles E. Leiserson ,Ronald L. Stein,.PHI,Chapter 11.
[3] J.Zobel,S.Heinz,andH.E.Williams.In- memory Hash Tables for Accumulating Text Vocabularies.
[4] Carter, J. L., And Wegmn, M. N. 1979. Universal classes of hash functions.J.Comput. Syst.Sci.28,2(Apr.),143-154.
[5] Shlomi Dolev, Limor Lahiani, Yinnon Haviv, "Unique permutation hashing", Theoretical Computer Science Volume 475, 4 March 2013, Pages 5965.
[6] Broder, A. Z. (1993). "Some applications of Rabin's fingerprinting method". Sequences II: Methods in Communications, Security, and Computer Science. Springer-Verlag. pp. 143152.
[7] Knuth. "The Art of Computer Programming". Volume 3: "Sorting and Searching". Section "6.4. Hashing".
[8] Open Data Structures. "Multiplicative Hashing"
[9] Bret Mulvey, Evaluation of SHA-1 for Hash Tables, in Hash Functions. Accessed April 10, 2009.
[10] http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.7520 Performance in Practice of String Hashing Functions
[11] Menezes, Alfred J.; van Oorschot, Paul C.; Vanstone, Scott A (1996). Handbook of Applied Cryptography. CRC Press. ISBN 0849385237.
[12] Richard J. Cichelli. Minimal Perfect Hash Functions Made Simple, Communications of the ACM, Vol. 23, Number 1, January 1980.
[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Third Edition. MIT Press, 2009. ISBN 978-0262033848. Section 11.5: Perfect hashing, pp.267, 277282.
[14] Fabiano C. Botelho and Nivio Ziviani. "External perfect hashing for very large key sets". 16th ACM Conference on Information and Knowledge Management (CIKM07), Lisbon, Portugal, November 2007.

[15] Schneier, Bruce. ”Cryptanalysis of MD5 and SHA: Time for a New Standard”. Computerworld. Retrieved 2016-04-20. Much more than encryption algorithms, one-way hash functions are the workhorses of modern cryptography.
[16] Finney, Hal (August 20, 2004). ”More Problems with Hash Functions”. The Cryptography Mailing List. Retrieved May 25, 2016.
[17] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, Finding Collisions in the Full SHA-1
[18] Bruce Schneier, Cryptanalysis of SHA-1 (summarizes Wang et al. results and their implications)