International Journal of Networked and Distributed Computing, Vol. 3, No. 1 (January 2015), 60-68

# Comparison of Hash Table Performance with Open Addressing and Closed Addressing: An Empirical Study

**Dapeng Liu**
*Gradient X, 3100 Donald Douglas Loop North*
*Santa Monica, CA 90405, USA*
*Liodapeng@gmail.com*

**Shaochun Xu**
*Department of Computer Science, Algoma University, 1520 Queen Street East*
*Sault Ste Marie, ON, P6A2G4, Canada*
*simon.xu@algomau.ca*

## *Abstract*

In this paper, we conducted empirical experiments to study the performance of hashing with a large set of data and compared the results of different collision approaches. The experiment results leaned more to closed addressing than to open addressing and deemed linear probing impractical due to its low performance. Moreover, when items are randomly distributed with keys in a large space, different hash algorithms might produce similar performance. Increasing randomness in keys does not help hash table performance either and it seems that the load factor solely determines possibility of collision. These new discoveries might help programmers to design software products using hash tables.

*Keywords:* hash table, open addressing, closed addressing, nosql, online advertising.

## 1. Introduction

Hash table [1] is a critical data structure which is used to store a large amount of data and provides fast amortized access. When two items with same hashing value, there is a collision. There are different implementations to solve collisions and reduce the possibility of collisions, such as open addressing and closed addressing. Some hash variants, such as Cuckoo [2], worked to improve access time even more predictable. A few approaches [3] tried to reduce collisions by using a family of multiple hash functions instead of one. Some work tried to find out better hash functions, such as SHA-1 [4]; while most of such effort is used in cryptography. Most previous researches mainly focused on the amortized access time, instead of understanding the worst cases or finding quantitative measurements. It seems necessary to study the performance of hash functions so that we can make good choice for the hash tables. On the other hand, most of existing research work on hashing did not focus on a large amount of data. Nowadays, it is not rare to work with multiple billion data sets and large data sets are often partitioned into multiple hash tables; which are not necessarily distributed hash tables.

A lot of computer applications are expected to respond fast while processing a large amount of requests which may be continuous. Such applications are typical online ones, including online search and online advertising.

Published by Atlantis Press

In this paper, we would like to choose on-line applications to conduct some experiments with different hashing approaches to make comparison. Such experiments could help us to understand further the performance of open addressing and closed addressing with a large set of data. The experiment results might be useful for programmers in making design decisions.

The research questions of this work include:

- Would it be the case that open addressing and closed addressing perform similarly when a large set of data are involved?
- If open addressing and closed addressing have different performance, when will they diverge along with the increase of data?
- Would a better hash algorithm increase amortized hash table access time? Or, in another word, what factors might affect the hash table performance?
- For closed addressing, how does the size of the key space affect the performance of hashing?

This paper is organized as follows: Section 2 introduces related work. The experiment motivation and setting are given in Section 3. Section 4 describes experiment results and provides analysis. Section 5 concludes the paper and lists future work.

## 2. Related Work

Hash table [5] is an associative array that maps keys to hash values and uses the hash values as the indexes of the array to store the data. The core is a hash function that computes an index of buckets in an array, from which the item can be found. It has been believed that in many situations, hash tables turn out to be more efficient than search trees or any other lookup structures.

There have been many researches on hashing. Robin Hood Hashing [6] tied to equalize the length of probe sequences by leaving the item with the longer probe sequence in current probing position and move forward the other one with shorter probe sequence in case of collision. Cuckoo Hashing [7][8][9] used two hash tables and two different hash functions respectively; each item, is either in the first hash table or in the other one. If a new item has a collision in the first hash table, the collided item is moved to the second hash table using the second hash function. Items are moved back and forth until the last collided item moves to an empty location or a limit is reached. If the limit is reached, new hash functions are chosen to rehash the tables. It was reported that for tables that are a bit less than half full and with carefully chosen universal hashing functions, performance would be good.

There are also some researches focusing on the quality of hash function. A basic requirement is that the function should provide a uniform distribution of hash values, which may be evaluated empirically using statistical tests, e.g. a Pearson's chi-squared test for discrete uniform distributions [10][11]. Cryptographic hash functions, such Message Digest Algorithm (MD, well-known example MD5) [12] and Secure Hash Algorithm (SHA, representative SHA-1) [13], are believed to provide good hash functions for any table sizes.

Hash table has recently been widely used in many NoSQL products such as Memcached [14], Cassandra [15], Redis [16], and Riak [17]. One common characteristic of NoSQL products is that they can handle large data volume, such as multiple billion rows, given enough hardware capability. The popularity of these NoSQL products has validated the effectiveness of hash table.

With the growth of the key size of the items, a hash table often needs to be rehashed with distribute keys into a bigger space and some effort was taken to reduce the impact on performance [18]. Due to limited system resources, large data might have to be saved into multiple hash tables on different computers. One specific such variant is distributed hash table [19][20][21] which especially cares about the distributed and unreliable environments.

Although there are a lot of researches and applications on hash table, further work on some directions are still needed. For example, it is commonly believed that closing addressing provides better predictability than open addressing, but there are few works measuring their practical performances. In addition, a lot of existing work did not test their algorithms or approaches with large hash tables with multiple million rows.

## 3. Experiment Design

It has been well known that there are two approaches handling hash collision: open addressing and closed addressing and the load factor is better to be controlled under 0.7. However, the preference has been rarely given to any addressing and how hash table performance degrades along with the increase of the load factor. In reality, better hash functions are often sought to improve the worst case performance of hash tables. Given the limited hash table size and enormous value space of hash functions, it seems that a better hash function might not be able to increase the worst case performance of hash tables. Thereafter, we

designed a set of experiments to investigate the impact of various factors on hash table performance.

The purpose of the experiment is to compare the performance of open addressing and closed addressing with large data size along with various load factors and to see whether there is any impact of the size of key space on the performance of hashing. We also try to evaluate impact of partitioning in hashing in which data are divided into multiple hash tables.

Since there are a large variety of software programs that are using hashing in different ways to handle data of different patterns, at first we want to summarize characteristics of the applications that affect our choice of data access and storage solutions. In this experiment, we focused on online advertising application which has the following characteristics:

- Fast response and vast throughput; each data access has to completed in about 10 ms and there is no pause time of the system to reorganize any data; this means hashing has to be extremely reliable;
- There is no rest time so that hashing has to be continuously reliable;
- Huge data volume, so that it is critical to be memory-efficient;
- User ids are normally randomly created since there is no precise way to keep track users; GUID is often used; as a result, the key space is huge, keys are sparse in the space, and there is little chance of key collision;
- Nosql, no join, just discrete data; there is no interrelationship between different user ids; so the program performance is closely related to hashing performance and data can be easily partitioned;

We think such application scenarios most likely would give challenges to hashing by calling for reliable and consistent high performance. Choosing such applications for the experiment might give insight on the performance of hashing.

### 3.1. *Hash algorithms*

Hash algorithm is the core of hash tables and it is a common effort to look for a better hash algorithm that could reduce the collision possibility.

In this experiment, we chose three representative hash algorithms: SHA-1, which represents complex and best-quality hash functions, X31, a fast and simple one, Murmur, a popular and well-received tradeoff of the previous two.

The result of Murmur hash is an eight-byte (64-bit) long and its calculation is shown in Figure 1:

```
Public static long murmurHash (String key) {
        byte[] data = key.getBytes ();
        ByteBuffer bb = ByteBuffer.wrap (data);
        LongBuffer lb = bb.asLongBuffer ();

        int len = key.length ();
        int seed = 42;
        long m = 0x00c6a4a7935bd1e995L;
        int r = 47;
        long h = seed ^ (len * m);
        for (int i=0; i < len / 8; i ++){
                long k = lb.get (i);
                k *= m;
                k ^= k >> r;
                k *= m;
                h ^= k;
                h *= m;
        }
        if (len % 8 != 0) {
                byte[] data2 = null;
                data2 = Arrays.copyOfRange(data, (len / 8) * 8,
len);
                switch(len & 7){
                        case 7: h ^= ((long)data2[6]) << 48;
                        case 6: h ^= ((long)data2[5]) << 40;
                        case 5: h ^= ((long)data2[4]) << 32;
                        case 4: h ^= ((long)data2[3]) << 24;
                        case 3: h ^= ((long)data2[2]) << 16;
                        case 2: h ^= ((long)data2[1]) << 8;
                        case 1: h ^= ((long)data2[0]);
                h *= m;
                };
        }
        h ^= h >> r;
        h *= m;
        h ^= h >> r;

        return h;
}
```

Fig. 1. Murmur hash in Java code

Compared to more complex calculation of Murmur hash, X31 is much simpler although it still returns a 64-bit value (see Figure 2). Simple hash functions like X31 have been commonly used in non-critical situations where the high calculation speed is preferred. In reality, simple hash functions often provide satisfactory performance.

```
public static long stringHashCode(String str) {
        char [] val = str.toCharArray ();

        int h = 0;
        for (char c : val) {
                h = 31*h + c;
        }
        return 0x00FFFFFFFL & h;
        }
```

Fig. 2. X31 hash in Java code

SHA-1 is a currently popular cryptographic hash function which returns 160-bit value. It is so popular and CPU-expensive that Intel supports its calculation with hardware [22]. In the situations where hash collisions are all to be avoided, SHA-1 is often used. Due to its complex, the implementation is not shown here.

### 3.2. *Hash table size*

All hash tables have limited size and in fact, many are not even large unless they have to be. Data could be rehashed if the volume is increased and a bigger hash table size is needed.

We use prime numbers for the table sizes which allow easy probing in open addressing, because any step length is a relative prime of the table size so that the open addressing can surely find an empty spot only if the number of items is smaller than or equal to the table size.

In reality the hash table size could exceed multiple trillion slots sometimes. However, no matter how large the hash table is, its size is still trivial compared to the value space of hash algorithms. X31 and Murmur return values of 64 bits, while SHA-1 returns 160 bits. In order to map from a much bigger value space into a smaller one, the randomness might need to be sacrificed, which is the motivation of us to investigate the performance of different hash algorithms.

### 3.3. *Key space*

Keys play an important role in the performance of hash tables as well. If a hash algorithm could not always scatter hash values for adjacent key values, in the situations where keys are sequentially created, collisions might have large chance to happen.

In this study, we focus on random keys, which are often adopted in web applications. A typical key generation algorithm in practice is UUID [23]. As UUID has a tremendous value space of fixed size, in this experiment we generate random keys by ourselves so that we can control its value space and thereafter the chance of clustering. A primitive randomization utility (Apache Commons, [24]) is used.

### 3.4. *Collision solutions*

Both closed addressing and open addressing are tested. In open addressing, linear probing and a long-jump strategy are tested. The formula to calculate probing step length in the long-jump strategy is as below:

$$probingStepLength = hashValue \% (hasTableSize - 1) + 1 \quad (1)$$

This formula creates a probing step of dynamic length and has been seen in concrete hash table implementation. The trick is to only use prime table size so that any step length will guarantee that it is able to probe all slots in the table. Table sizes that have been used in our experiments are listed in **Table** , from which we can see that using prime table sizes still enable us to roughly double table size along with data growth.

### 3.5. *Critiria: number of memory accesses*

We do not measure hash table building time or lookup time in this experiment. Our random key creation is CPU intensive and may take remarkably long time compared to memory lookup. Since we test keys of different lengths, the portion of time dedicated to memory search is not a constant value so that it is unfair for us to directly compare them. Meanwhile, while we emulate hash tables of different sizes, the effect of high-speed cache improve memory search to different extents. Moreover, modern multi-thread operating systems bring fluctuation into execution times. Due to those considerations, we do not measure access time of hash table; instead, we chose a more objective and hardware-independent metric: how many memory (cache included) accesses for one read or write.

### 3.6. *Process*

For different hash table sizes, keys of different lengths are randomly created, different collision solutions are used, and different numbers of items are filled into the hash table.

All test settings are listed in **Table** : cross multiplication of all columns represents experiment configurations. Moreover, we run such configuration for keys of length 3~15, exception for linear probing when table size is bigger than 3145739 when we found out the testing took many

minutes so that in reality nobody would consider it as a practical implementation.

We wrote a Java program to simulate hash operations with such configurations and the source code is available at the personal website of one author (http://cicadida.com/HashTesting.java). After we run our emulation program for a few days to fill into our database with data, we run SQL queries such as the sample shown in Fig. 3 to get statistical result.

Table 1. Combinations of Test paramters

| Hash Algorithm | Key length | Collision Solution | Table Size | Load factor | Read/ write |
|---|---|---|---|---|---|
| SHA-1<br>Murmur<br>X31 | 3<br>4<br>5<br>6<br>7<br>8<br>9<br>10<br>11<br>12<br>13<br>14<br>15 | Linked<br>Linear probling<br>Long jump | 3079<br>6151<br>12289<br>24593<br>49157<br>98317<br>196613<br>393241<br>786433<br>1572869<br>3145739<br>6291469<br>12582917<br>25165843 | 5%<br>10%<br>15%<br>…<br>…<br>…<br>70%<br>75%<br>80%<br>85%<br>90%<br>95%<br>100% | Read<br>write |

```
Select percentage, max(maxConflict), avg(maxConflict), avg(ratio)
from hashTest
where hashAlgorithm='murmur'
        and conflictSolution='Linked'
        and stringLength=13
        and size=786433
group by percentage
order by ratio;
```

Fig. 3. Sample SQL statements executed to get statistics

## 4. Experiment Result and Analysis

Since the complete statistical table is huge, we could only present a representative part here which best shows our concern. Specifically, without annotation, we will use data collected for Murmur as the representative for our analysis.

### 4.1. *Space utilization*

Space utilization for open addressing is simply the number of entries; but for separate chaining, while 95%*size entries are saved, there is still 39% free space left. That means we waste 39%*size memory with head records in the buckets implementation. We tried 100%*size entries and there are still about 37% empty buckets. Trying 200%*size entries still left us about 13.5% free space; trying 300%*size entries still left 5% free space. As fewer slots are used, some slots are occupied by multiple keys and therefore collision is unavoidable. While unused space reduces along with growth of keys count, the number of collisions is more linearly proportional. Space utilization of closed addressing is shown in Fig. 4.

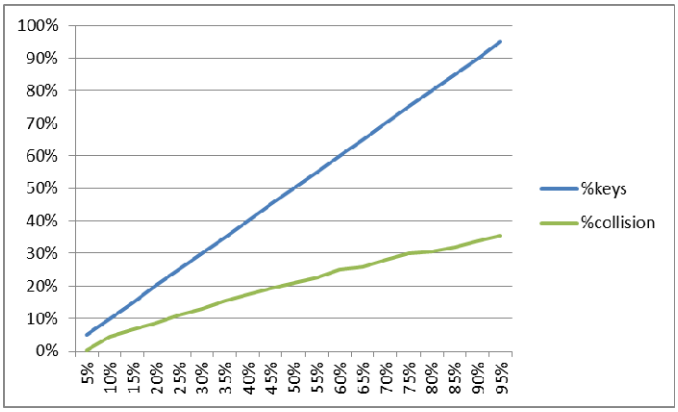


Fig. 4. %Collisions is roughly portioned to %keys

This observation helps us to understand the normal belief that a hash table should be enlarged when the load factor is 0.7, when only about 60% of slots are occupied. In other words, roughly (7-6)/7=14.3% keys have collisions.

We observed that the space wasting may be slightly higher when hash table size is bigger, the difference is around 1%; nonetheless, we did not do statistical analysis since we just ran emulation a couple of times and 1% is not statistically significant.

It seems that closed addressing would waste 37% space when the number of items is equal to the hash table size. This strongly suggests the use of separate chaining instead of separate chaining with list heads. If open addressing is used, high space utilization will increase collision possibility, especially when the number of items is near to the table size. Linear probing should be generally avoided and might only be considered for the favor of cache locality.

### 4.2. *Collisions*

For closed addressing, collisions are related to space utilization in that a collision would miss an empty spot; it provides more information in that we can see the worst performance of the hash table.

For open addressing, for random jump probing, if we try 95% and 100% size of entries when size=49157, from **Table 2** we can see clearly that when the amortized performance for hash table degraded 5 times, the worst performance deteriorated quickly: 283.7 times. This analysis is theoretical and shows clearly that a fully loaded open addressing hash table is barely useful.

Table 2: size=49157, Murmur, Linear probling, str Len=13

| #keys | Amortized collisions | Max collision |
|---|---|---|
| 95% | 2.17838 | 155 |
| 100% | 10.60608 | 43973 |

It is not a surprise along the big jump in the last 5% of writes because while the table is almost full some unlucky writes have to probe many times. Linear probing deteriorates more quickly when hash table becomes full.

As to collisions, linked hash table provides the most predictable worst cases, linear probing is intolerable, especially when the table gets full, and random jump probing performs average.

### 4.3. *Max collisions become worse as hash table sizes grow*

From Fig. 5, Fig. 6, and Fig. 7, we can see clearly that worst cases (max collisions) are getting worse along with the growth of hash tables. This coincides with the fact there are always access time outliers in presence of large amount of data.

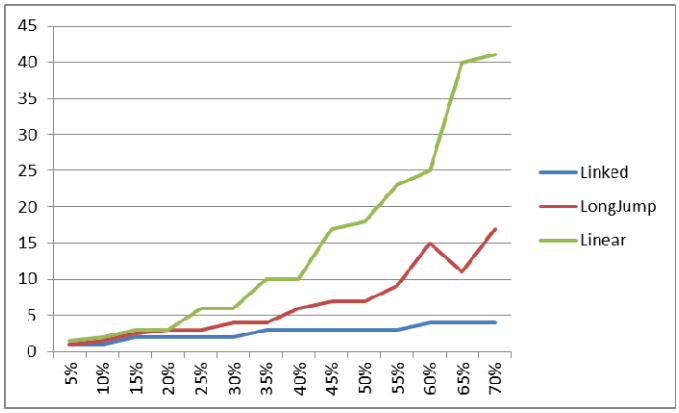


Fig. 5. Average max conflict for Murmur, size=6,151[1]

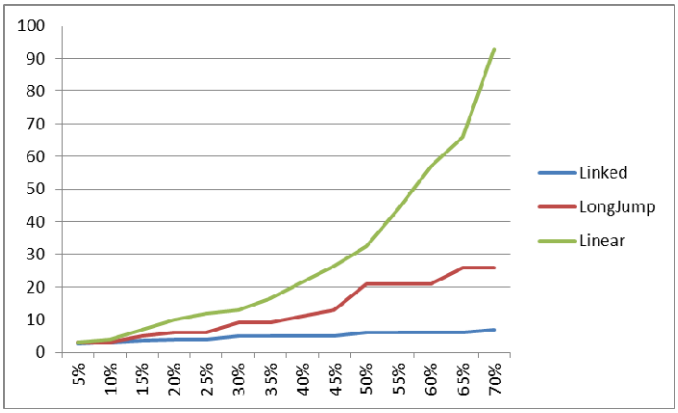


Fig. 6. Average max conflict for Murmur, size=786,433

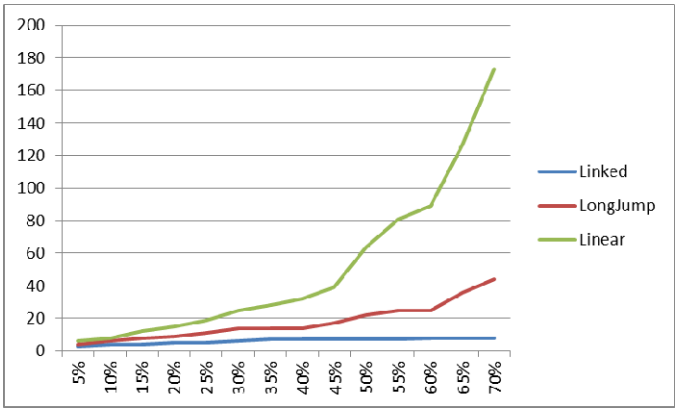


Fig. 7. Average max conflict for Murmur, size=100,663,319

Based on the experiment result, it seems that linear probing is not a good choice when hash table becomes crowded and the normal load factor 0.7 of hash table is too large for open addressing for fast response with large amount of saved data.

### 4.4. *Max collision of open addressing*

In this section, collision is also used to depict the number of checks there are before the sought key is found. In other words, how many times of rehashing there would be before the target is reached.

For open addressing, in general, we can see the worst cases get worse along with the growth of hash table size, which is reasonable because rare cases have bigger chance to happen.

The other discovery is the consistent amortized collision ratios.

[1] Long jump curve does not monotonically increase because we ran separate emulations for each percentage; later we collected data along with one round of execution so that all later curves are monotonic.

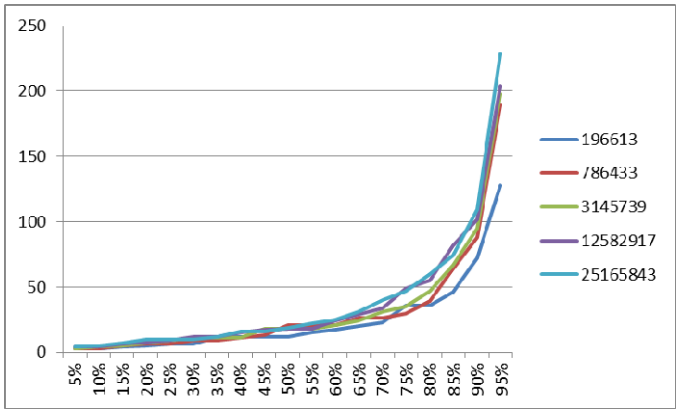


Fig. 8. Max collisions for Murmur, long jump, string length = 13

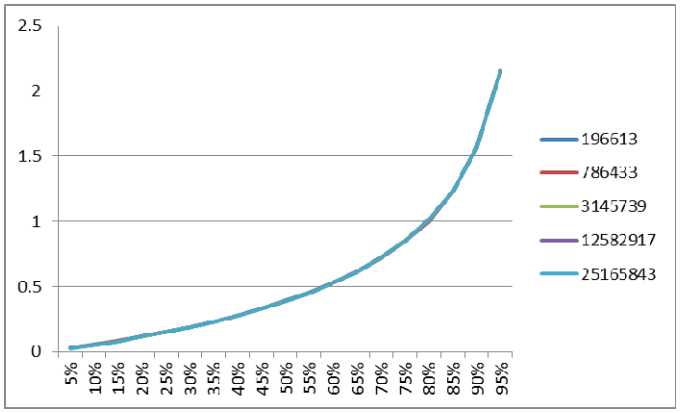


Fig. 9. Amortized collision, Murmur, long jump, string length=13
* all lines are overlapped

We can see at load ratio 0.7 the worst case of collision is about 40 times, which may be fatal for real-time applications because we are very sure cache misses happen most of the time and 40 accesses of main memory will put the host thread in an inferior situation when competing with other threads that only access main memory a couple of times.

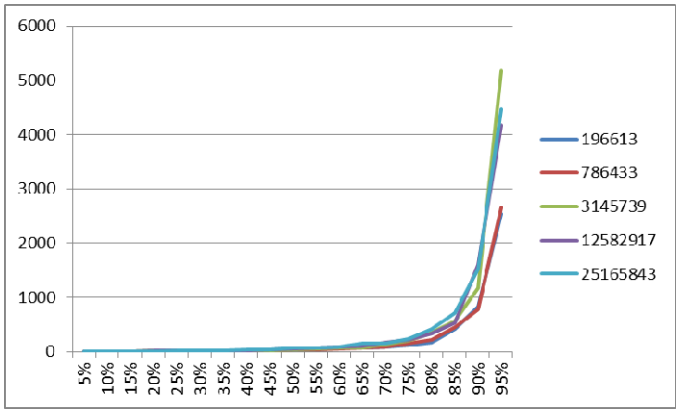


Fig. 10. Max collisions for Murmur, linear probing, string length=13

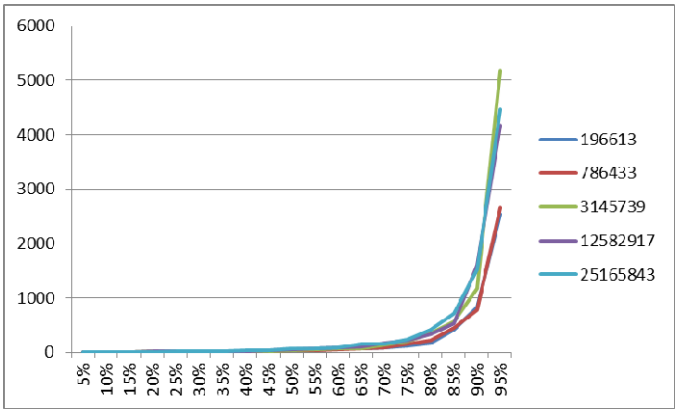


Fig. 11. Amortized collision, Murmur, linear probing, string length=13

From figures 10-11, we can see that linear probing clearly demonstrates its fast deterioration when the hash table is filled up. **Table 3**3 shows that worst case of write can be large. **Table**4 shows that 5% additional data will increase the amortized write count by 0.23 if load ratio is already 0.7. Thereafter the amortized collision ratio of this 5% data is:

(1.1688 * 0.7 – 0.9299 * 0.65) / 0.05 = 4.2745

While collision 4.3 times is negligible, the max collision times can be up to 156, which may be unacceptable.

Table 3. Max collision, Murur, linear probing, string length=13

| | **196613** | **786433** | **3145739** | **12582917** | **25165843** |
|---|---|---|---|---|---|
| **65%** | 71 | 66 | 77 | 110 | 156 |
| **70%** | 84.5 | 93 | 133 | 171 | 156 |
| **75%** | 124.5 | 153.5 | 202 | 222 | 242 |

Table 4: Amortized #write, Murur, linear probing, str length=13

| | **196613** | **786433** | **3145739** | **12582917** | **25165843** |
|---|---|---|---|---|---|
| **60%** | 0.74 | 0.7438 | 0.75 | 0.747 | 0.75 |
| **65%** | 0.92 | 0.9248 | 0.928 | 0.92587 | 0.9299 |
| **70%** | 1.15 | 1.15957 | 1.1679 | 1.163 | 1.1688 |

Such discoveries prompt partitioning a large hash table into a few smaller ones to avoid worse worst case.

### 4.5. *Hash table predictability*

From all of our above discussions, we can see that amortized access time for hash table is largely independent from table size; however, the worst cases get worse along with the growth of table size. Such discovery has not been mentioned in existing literature.

### 4.6. *Key length*

If key length is too small, there are surely duplicated randomly generated keys so that hash performance is lower than expected. From Fig. 12, we can see that merely when key length reaches 5, the hash table performance would become stable. Apache Commons is used to create random strings; under the hood a primitive randomization algorithm is used.

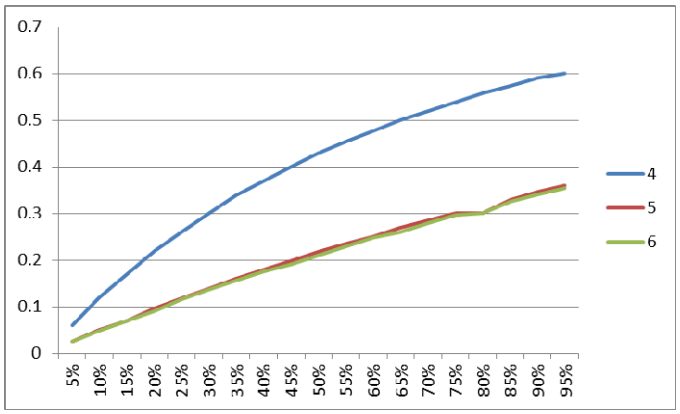


Fig. 12. Collision ratio per string length, Murmur, linked, size=25165843

We also observed that while key length is above or equal to 5, all three hash algorithms perform similarly. It seems that if a key is long enough and thereafter randomness is plenty, there is no need to increase the length anymore.

### 4.7. *Better hash functions bring fewer collisions?*

From Fig. 13, we can see that a better hash function would reduce collisions only when keys themselves are neither too dense nor too sparse in the key space. When key length is 3, there are only $62^3$ distinct keys and it is doomed to be too many same keys in the experiment so that all hash function result in 0.99 collision ratio; (actually with load factor 0.95 all hash functions performed similarly again for the same reason; constrained by limited space, no) when key is equal to or larger than 7, keys are so sparse that no hash function can further increase the randomness.

We know $\log_{62}(100663319)=4.464912$, however, due to birthday effect, keys become really sparse only when their length reaches 7. This observation shows the better hash function would help reduce collisions only when randomness of keys is in a middle range. If there are many duplicated keys, no hash algorithm can help to distinguish them. On the other side, if keys are already scattered into a large space, mathematical quality of hash algorithms do not matter much as well.

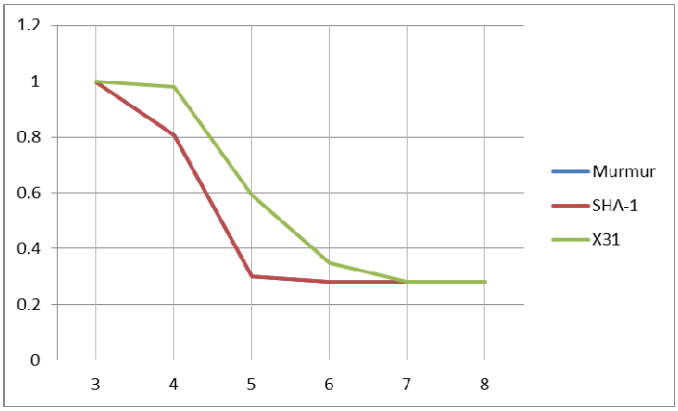


Fig. 13. Load=0.7, size=100663319 (*Murmur and SHA-1 are overlapped)

### 4.8. *Summary*

Based on the experiment, it seems that close addressing has better stability than open addressing when they are used in an on-line application with a large set of data. For close addressing, when the size of key space reach a certain level, bigger key space does not improve the performance of hashing.

Other conclusions which could be made include:

- Linear probing is not a really feasible variant in general.
- One third of in-list space is wasted in close addressing when the load factor is 1.0 so that separate chains with head in list would waster 30% memory.
- Better hash function works better for small key space where keys are not far from each so that avalanche effect can help distribute their index more evenly.
- When key space is big enough and keys are random, there is no difference between different hash algorithms; although better hash algorithm can distribute hash value more evenly, folding it into a much smaller index space damages its performance.
- Hash table performance is related to its size. The bigger the size is, the larger the max memory access time is to solve a collision.
- Space efficient 0.7 may be too large for time-critical applications, especially when the data size is large, when close addressing is used.

## 5. Conclusion and Future Work

In this paper we have conducted an empirical study of hash table performances for a particular kind of software applications. The result demonstrates that closed addressing has better stability than open addressing when they are used in an on-line application with a large set of data. For closed addressing, when the size of key space reaches a certain level, bigger key space does not improve the performance of hashing. It seems that hash table performance is related to its size, and the bigger the size is, the larger the max memory access time is to solve a collision.

This work would help programmers better understand why their system cannot achieve 100% consistency and select hash table products wisely even before they do benchmark tests.

Our future work is to find a way to improve predictability of hash table having a large amount of data.

## 6. Acknowledgments

Shaochun Xu, a faculty member of Algoma University, would like to acknowledge the support of Algoma University Travel and Research Fund, and the Oversea Expert Grant provided by Changsha University, China.

## 7. References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Third edition. The MIT Press, 2009.

[2] R. Pagh, N. M. Bldg, D.-A. C, and F. F. Rodler, *Cuckoo Hashing*. 2001.

[3] M. Thorup, "Even strongly universal hashing is pretty fast," in *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, Philadelphia, PA, USA, 2000, pp. 496–497.

[4] H. Feistel, "Cryptography and Computer Privacy," *Scientific American*, vol. 228, no. 5, pp. 15–23, May 1973.

[5] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *Data Structures and Algorithms*, 1st ed. Addison-Wesley, 1983.

[6] P. Celis, P.-A. Larson, and J. I. Munro, "Robin hood hashing," in *, 26th Annual Symposium on Foundations of Computer Science, 1985*, 1985, pp. 281–288.

[7] R. Pagh and F. F. Rodler, "Cuckoo hashing," in *Journal of Algorithms*, 2001, p. 2004.

[8] M. Drmota and R. Kutzelnigg, "A precise analysis of Cuckoo hashing," *ACM Trans. Algorithms*, vol. 8, no. 2, pp. 11:1–11:36, Apr. 2012.

[9] L. Devroye and P. Morin, "Cuckoo Hashing: Further Analysis," *Information Processing Letters*, vol. 86, p. 2003, 2001.

[10] K. Pearson, *On the Criterion that a Given System of Deviations from the Probable in the Case of a Correlated System of Variables is Such that it Can be Reasonably Supposed to Have Arisen from Random Sampling*. 1900.

[11] R. L. Plackett, "Karl Pearson and the Chi-Squared Test," *International Statistical Review / Revue Internationale de Statistique*, vol. 51, no. 1, p. 59, Apr. 1983.

[12] R. Rivest, "The MD5 Message-Digest Algorithm." [Online]. Available: http://tools.ietf.org/html/rfc1321. [Accessed: 17-Apr-2013].

[13] Federal Information Processing Standards Publication 180-1, *SECURE HASH STANDARD*. 1995.

[14] "Memcached," *Wikipedia, the free encyclopedia*. 13-Apr-2013.

[15] "Apache Cassandra," *Wikipedia, the free encyclopedia*. 16-Apr-2013.

[16] "Redis," *Wikipedia, the free encyclopedia*. 15-Apr-2013.

[17] "Riak." [Online]. Available: http://basho.com/riak/. [Accessed: 17-Apr-2013].

[18] Q. Ye, D. Parson, and L. Cheng, "Hybrid open hash tables for network processors," in *2005 Workshop on High Performance Switching and Routing, 2005. HPSR*, 2005, pp. 113–117.

[19] J. Li, J. Stribling, T. Gil, R. Morris, and F. Kaashoek, "Comparing the performance of distributed hash tables under churn," presented at the Proc. of the 3rd International Workshop on Peer-to-Peer Systems, 2004.

[20] M. Naor and U. Wieder, "A Simple Fault Tolerant Distributed Hash Table," in *In Second International Workshop on Peer-to-Peer Systems*, 2003, pp. 88–97.

[21] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood, "Fast hash table lookup using extended Bloom filter: an aid to network processing," in *In ACM SIGCOMM*, 2005, pp. 181–192.

[22] "Intel® SHA Extensions." Accessed May 18, 2014. https://software.intel.com/en-us/articles/intel-sha-extensions.

[23] Leach, Paul J., Michael Mealling, and Rich Salz. "A Universally Unique IDentifier (UUID) URN Namespace." Accessed May 19, 2014. http://tools.ietf.org/html/rfc4122.

[24] "Apache Commons - Apache Commons." Accessed June 8, 2014. http://commons.apache.org/.