ISSN 2277-3061
Volume 15 Number 9
International Journal of Computers & Technology

EFFICIENT DESIGN OF STORAGE AND RETRIEVAL METHODS

Abid Thyab Al-Ajeeli
Higher Education Committee,
C.O.R, IRAQ
Email: abidtj@yahoo.com

ABSTRACT

The paper demonstrates the viability of a number of storage techniques such as hashing using multilingual texts. The empirical studies of hashing techniques based on multilingual provide researchers with insight information into the structure of the methods and their applicability. This may lead to more theoretical findings to enhancing performance of storage and retrieval algorithms.

A number of experiments were performed using real and random texts from English and Arabic languages with numerous sets of data items and length sizes. The English language is used in the experiments as measurement tools as it is the international language of the world and it has been received the best attention from researchers all over the world. The study shows that chaining techniques consistently generates less number of collisions than open addressing techniques for texts from any language.

The motivation behind conducting this study was the lack of research on the performance of hashing algorithms using data items that have dependencies and structures from other than English language. There are also no comprehensive studies performed on Arabic strings. The new finding probably leads to design more efficient hashing algorithms for storing, deleting and retrieving items of information.

Keywords

Hash Techniques, chaining Techniques, Loading Factors, Multilingual, Arabic texts.

INTRODUCTION

Hashing methods are used in many different applications of computer science disciplines. These applications include spell checkers, database management applications, information retrieval, symbol tables, loaders, assemblers, compilers, …, and thesaurus. The symbol table makes heavy use of hash techniques to perform the basic operations including searching, insertion, …, and deletion. Hashing techniques are used in numerous applications because of its performance.

Hashes are also used as controls guarding against violation of security such as integrity of files and documents. They are also used in digital signature algorithms. Hashing processes enhance security by detecting any slight change in data through the comparison of old hash with the current one. This is much more convenient for many encrypted applications since a reference (hash) to a document is encrypted which is much faster than encrypting the whole document.

In a normal data processing, one is given a key of an object (record, image… etc.) and is required to finding the data associated with that key. The problem, in these cases, is to find a way converting the key values into hash table addresses so that the items of data can be found at those addresses more quickly and efficiently.

Hash tables, sometimes known as associative arrays, or scatter tables, are well-designed implementation with a set of operations. The associated set of operations have a time complexity of *O (1)* while other data structures and the operation associated with them requires on average *O(n)* time complexity.

The characteristics of the names and attributes of hash tables vary according to the underlying applications. For example, in a thesaurus, the name is a word, and the attribute is a list of synonyms of the word; in a compiler symbol table, the name is an identifier, and the attribute is a list of values.

Hashing function is a one-way mathematical function that be used to compute a small and fixed length message also called fingerprint, or message abstract to ensure faster signing [28]. A hash function takes a key and maps it to some index in the data structure as shown in figure (1).

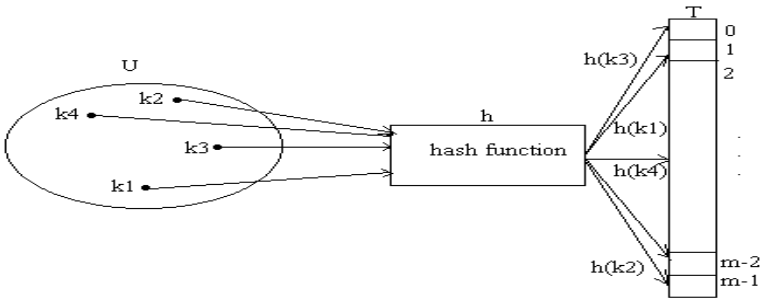

Figure (1): Basic concept of mapping from a key space to a table space

Figure (1) shows that we need to find a function “h” which translates a key “k” from the key space U into an index of table and for storing items of the same type as k. The key can be a string, number, record, etc. [1, 24]. A general approach is to design an abstract data type (ADT) hash table to perform a number of operations as described below:

```
ABSTRACTDATATYPE      HASHTABLE
{
        Instances:    collection of items with generated distinct keys
        Operations:
                GenerateKey (x, k): generate a distinct key for item x using a
                                    hash function f(k) where
                                    f(k) = generate( x, k).

                Insert (x, k):      insert the item x whose key is k into the
                                    HASHTABLE.

                GetItem (x, k):     return the item x at position k.
}
```

Studying hashing techniques using multilingual texts provides, I feel, researchers with insight into the structure of the hash methods and their applicability. This may lead to more theoretical studies to enhancing the performance of hashing algorithms.

BACKGROUND

In order to furnish the reader with some theoretical background, Let $U = \{0, 1, 2, \ldots, u-1\}$ to be a universe, and let S be a subset of such a universe with number of keys equal to n. The keys belong to the alphabet set Σ For example, the English language alphabet is the $\{A. . Z, a. z\} \varepsilon \Sigma$. The Arabic language alphabet is set $\{ا \ldots ي\}$ also $\varepsilon \Sigma$. The largest set of Σ is the ASCII set which includes up to 256 symbols.

Since S is a subset of U, then a hash function $h: U \rightarrow M$ maps the set of keys S into some given interval of some given type M (let it be integer), say $[0, m-1]$, $m > 0$. This hash function computes an address that is an integer from M and it is to be used for the storage or retrieval of that item [2].

A hashing method, as indicated above, is a scheme that computes an address directly from a key, which ideally aims to avoid collisions. In general, a limited amount of searching is unavoidable. The hashing schemes simply perform an identifier transformation through the use of a hash function h(x), which is used to perform an identifier transformation on x; it maps the set of possible identifiers onto the range [0, m-1] locations. It is desirable to choose a hash function h(x) that is easy to compute and capable to producing only few collisions. Therefore, a mechanism to handle overflow is needed, since it is impossible to avoid collision altogether.

There are several interesting and enlightening research articles on hash functions. These articles are categorized into: storage techniques [3, 4], hash functions performance [5], clustering [6], Hash table techniques [7], quadratic searching [8], reducing retrieval times [8], and dynamic hashing [9,10,11].

Our study is based on a number of hash functions ranging from simple to sophisticated ones. One of the interesting linear hash function outlined in the McKenzie paper [20] was:

```
usigned long cbuhash ( string key ) ;
        usigned long h = 0 ;
        for i from 0 to size (key) –1
                h = h<<2 + key [I] ;
    return h;
```

An odd number must divide the resulting hash value and the remainder used as hash address. This function works well for natural language text. Another interesting hash function proposed by Peter K. Pearson in his paper published in [21] was:

```
        unsigned byte pkphash ( string key ) ;
                unsigned byte ptab [256] = { 235, 26, 38, . . . } ;
```

```
unsigned byte h = 0;
for i from 0 to size (key) –1
        h = ptab [ h xor key [i]    ] ;
return h;
```

The above algorithm works well for hash ranges within 0-255. It becomes more costly for ranges greater than 0-255. For example, for a 64-bit range, Robert Uzgalis [22] claims, “it costs roughly 7 times more to compute than a range of 0-255” and he proposed an improved version as shown below:

```
unsigned long pkphash8 ( string key ) ;
unsigned byte ptab0 [256] = { 36, 28, . . . } ;
unsigned byte ptab1 [256] = { 96, 111, . . . } ;
        . . .
unsigned byte ptab7 [256] = {231,    1,  . . .  };
unsigned byte h1 = 0;
unsigned byte h2 = 0;
        . . .
unsigned byte h7 = 0;
for i from 0 to size {key} –1
{       h0 = ptab0 [h0  XOR  key [i] ];
        h1 = ptab1 [h1  XOR  key [i] ];
        . . .
        h7 = ptab1 [h7  XOR  key [i] ];
        }
return  (h0<<56  +  h1<<48  +  . . .  +  h7) ;
```

From the above discussions we conclude that a hash function is a function that maps a bit string or arbitrary length to a fixed length bit string. It has many advantages in storing, retrieving, and cryptography. It has many advantages related to information transmission security including [28]:

- Hashing is a one-way function and thus it is not possible to compute the original message from its has,
- Any change in the message will change the message abstract- thus change will immediately be detected,
- Hash function receives messages of any length and produces hash of fixed leghth which is smaller than the message itself, and
- Hashing algorithms are faster than any symmetric and asymmetric encryption algorithms.

A survey of more detailed hashing functions will be found in [22, 23].

ANALYSIS OF HASHING METHODS

The general scheme of a hashing procedure is to generate an address, or an index from given keys. When the generated address is empty, the key and the associated items of information are stored there. But, when a collision is encountered; that is, some other item is already occupied that address, an additional address or sequence of addresses would be generated until an empty location is found for that item.

Practically, most items are stored at their first generated addresses, but a few items may need to be stored at their second addresses; a fewer at their third, and so forth. Items are retrieved by the same process looking at the first generated hash address. When a match does not occur a second address is calculated and so on until a match is found.

Experimental studies done by authors in [11, 12] show that hashing mechanisms provide a very good performance over conventional techniques such as searching methods and balanced trees. For example, the complexity of searching a list of n records, using sequential search method, can on average be O(n/2) and in the worst cases may reach up to O(n). Similarly the binary search method has a complexity of O(Log n) + extra cost. Extra cost may be attributed to sorting mechanisms. For these reasons hash functions are promising mechanisms for storing, deleting and retrieving items of information.

The results of the analysis studied by other authors [14, 15, 16], show that the performance of a hash method depends on the approach used for handling overflows. The above conclusion is only true when the identifiers are selected at random

from the identifier space. In practice, there is a tendency to make a biased use of identifiers. Many identifiers have a common suffix or prefix or simple permutations of other identifiers. Hence, in practice, we expect different hash functions to result in different hash table performance [5].

Loading Factors

Loading factor is defined as the ratio of records in the file to the records that can be stored in the address or storage space. In other words,

Loading factor = Values actually stored in the table/size of the table

The load factor can be any number between 0 and 1 to indicate how full the hash table is. A small factor means more free space for future addition of records and decreases the chance of collisions and overflow conflict.

The performance trade-off between three techniques - linear probing, rehashing, and chaining- has been studied by numerous researcher including [12, 13, 16]. Figure (2) shows the number of collisions of each technique (the graph is drawn using formulas from [13, 27]).

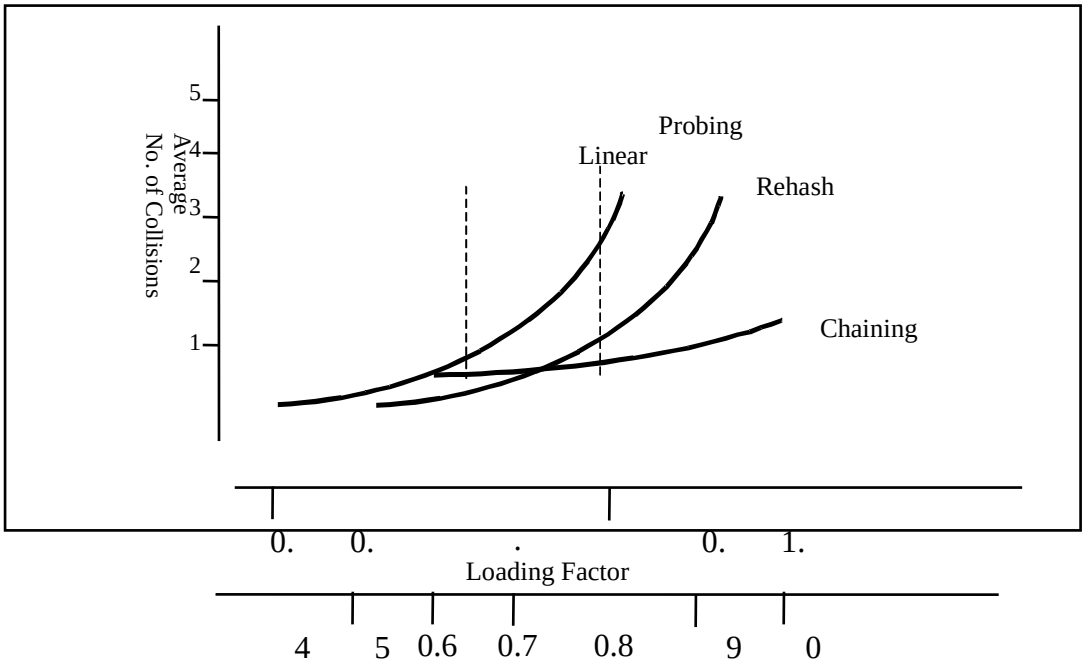

Figure (2): Comparisons of Hashing Techniques: successful cases

Figure 2 shows some important points:

1. All hashing techniques behave equally well for low loading factors.
2. The chaining method performs better for the entire range of the loading factor.

HASHING TECHNIQUES

Hashing techniques can be in general divided into two main categories:

1) Open addressing technique: in which overflow record is stored in the next address location if it is empty, otherwise, proceeds to search for other nearest empty location until an address is found, and the record will be stored there. The algorithm below is used for this purpose.

i. Transforming non-integer hash field values into integer as follows:

A = 1;

For I = Lower to Upper do A = A × code (ki);

Hash-address = A mod tablesize;

ii. Collision resolution:

j = Hash-address;

if location j is occupied then

{ j = (j+1) mod tablesize;

```
while ( j ≠ hash-address) and location j is occupied
        j = (j+1) mod tablesize;
if ( j = hash-address) then all positions are occupied
else  new-hash-address = j;
}
```

2) Chaining technique: in which a pointer is stored in the home address to indicate where the overflow record is stored.

In the proposed algorithm, we have used two different hash functions for each category. These hash functions are:

a. **Division method:** it operates by dividing a data item's key value by the total size of the hash table and using the remainder of the division as the hash function return value. This method has the advantage of being very simple to compute and very easy to understand. It has the following structure.

h(k) = f(x) mod TableSize

The best choice of table size is numbers that do not divide ri +a, where i and a are small numbers and r is the radix of the chosen language character set. In general, A good rule of thumb in selecting a hash table size for use with a division method hash function is to pick a prime number that is not close to any power of two (2, 4, 8, 16, 32...).

```
int hash_function(data_item item)
{
return item.key % hash_table_size;
}
```

b. Multiplicative method

The multiplication method provides an easy method of computing hash functions without the imposed restrictions encountered in the division method. It can be used with hash tables with a size that is a power of two. The data item's key is multiplied by a constant, k and then bit-shifted to compute the hash function return value.

A good choice for the constant, k is $N * (sqrt(5) - 1) / 2$ where N is the size of the hash table. The product key * k is then bitwise shifted right to determine the final hash value. The number of right shifts should be equal to the log2 N subtracted from the number of bits in a data item key. For instance, for a 1024 position table (or 210) and a 16-bit data item key, you should shift the product key * k right six (or 16 - 10) places. For simplicity and efficiency we used the following structure to compute h(k).

$h(k) = Trunc\{\Phi \times f(k) - Trunc(\Phi \times f(k)) \times TableSize\}$

Where Φ is the inverse of the golden ratio,

$\Phi = 2/(1 + \sqrt{5})) \cong 0.618339887$.

EMPIRICAL EXPERIMENTS

To measure the performance of hashing algorithms we carried out a series of experiments. Various distributions of data items were provided and the performance of such hashing algorithms was recorded.

To represent the Arabic character set, we may use the Nafitha software developed by 01 System, Manama, Bahrain (Nafitha 88), or any of the latest versions of MS Windows. We based our implementation on Nafitha-Mussa'ed Alarabi/2 (code page 786) coding scheme.

In this experiment, four different algorithms were investigated using two different sets of data. The experiments were conducted on alphabetic keys of length n, where n = 3, 4, ..., 255, with different hashing table sizes ranging from 20 up to 2000 entries.

The program has been tested using random data and real data files. It has also been run on newspaper corpus selected from English and Arabic newspapers covering similar topics.

For each experiment, we use the following methodology:

1. Generate n character alphabetic keys $k_{n-1}\ k_{n-2} \ldots k_0$.
2. Compute f(x) as follows:

$f(x) = \sum\{ (code (k_j) - i) \times M^j \}$, for $j = 0 \ldots$ No of Character-1, ***i*** is used as a scalar to reduce the chance of overflow. We chose M to be 2.

3. Compute h(x) for both division and multiplication methods.

4. Count number of collisions using the two collision resolution techniques

(That is: Open addressing and chaining).

Our proposed implemented system uses a number of methods (functions) as follows:

- Random number generator: it generates random numbers.
- Character generator: it generates character from the underlying language character set.
- Key list generator: it transforms the random numbers into a list of keys.
- Division list generator: it applies division method to generate list of keys
- Multiplication list generator: it generates a list of keys by applying multiplication method.
- Open address generator: it generates a list of keys using linear probing method.
- Chain table generator: it generates a table of chains of keys.
- Performance generator: it counts the number of collisions and comparison operations done by each hashing method.
- Loading factor generator. The load factor of a hash table is just a technical way of saying how full the hash table is. If the hash table is empty, the load factor is 0. If the hash table is full, the load factor is 1.

A number of experimental cases have been conducted as outlined below.

CASE 1: Random Data Files

Random characters are generated from $\sum_A$ for the Arabic language and $\sum_E$ for the English language. The recursive relation used is of the form:

xn = f(xn-1, xn-2, …, x0)

One such function is

xn = a xn-1 + b mod m

where xn's are integers between 0 and m-1, constants a and b are non negative. For more details see [26, 21].

In random data files, we implement the following steps:

- Generate records of length n-character alphabetic keys kn-1 k n-2. . . k0 where each character ki is uniformly generated using the formula:

k_i = Char *(Int [Uniform Distribution (Seed)$_\times$ Size] + code (1)).*

$k_i \varepsilon \Sigma$.

where the variable Size represents the number of characters in a language. For example English language has 26 characters. Code(1) represents the Ascii code for the first alphabetic letter; that is: 65 for 'A' and 97 for 'a'.

- Observe the number of collisions using different data records and different hash table sizes, a number of experiments was performed as follows.

First experiment

This experiment uses the additive function III in appendix A and the random function I in appendix A. The number of collisions for each method is recorded and results plotted as shown in figure (3).

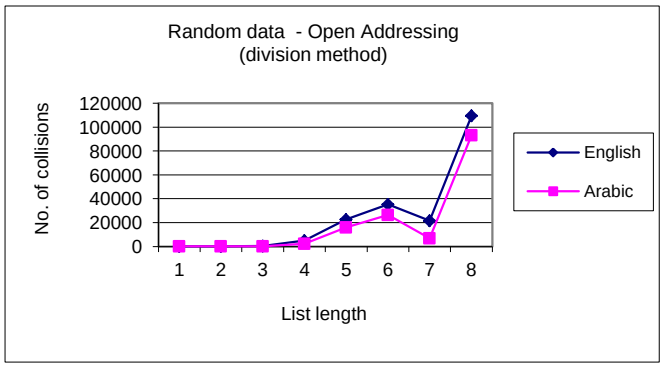

Figure: (3-a)

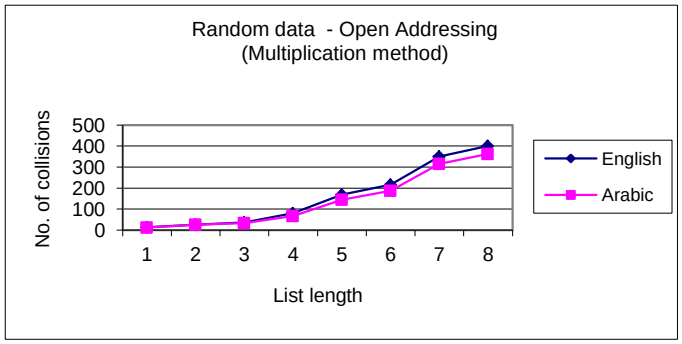

Figure: (3 - b)

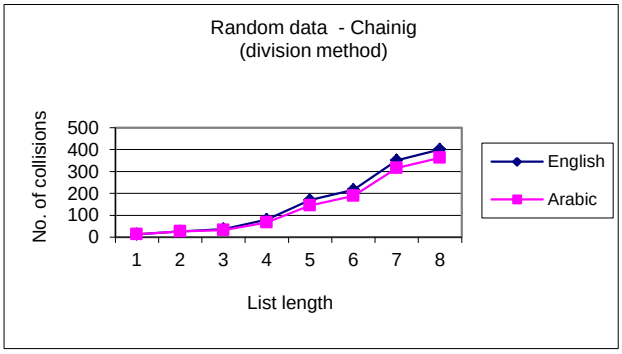

Figure: (3-c)

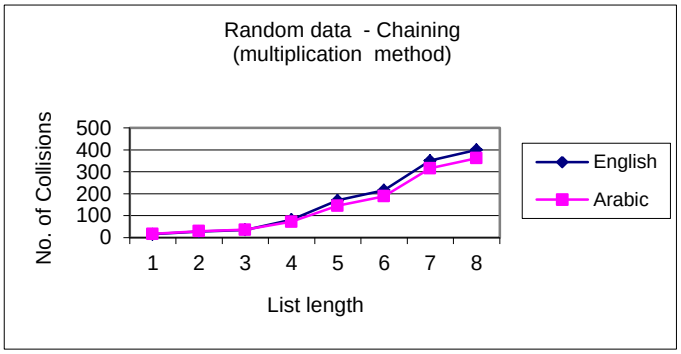

Figure (3 - d)

From figure (3) above, one can conclude that for all cases Arabic text shows better performance than English text.

Second experiment

The second experiment used additive function III in appendix A and the random function II in appendix A. Number of collisions for each method is recorded and results plotted as shown in figure (4).

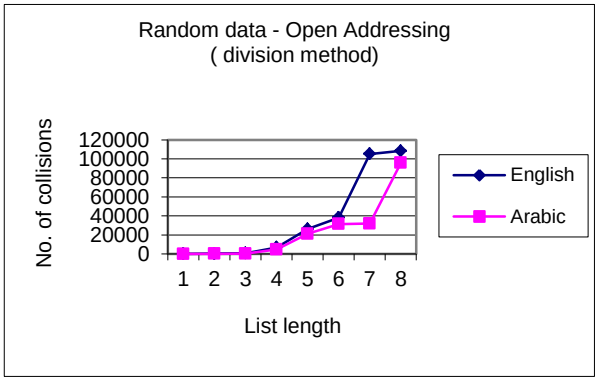

Figure: (4 - a)

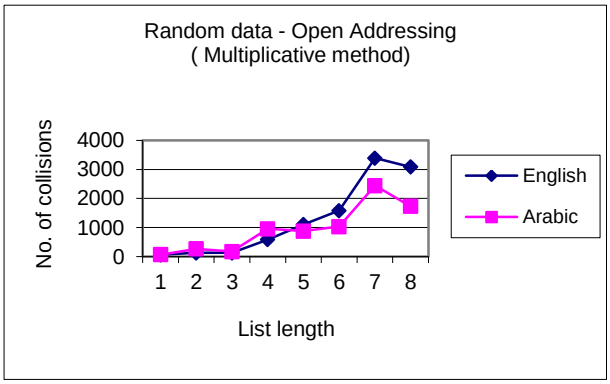

Figure: (4 - b)

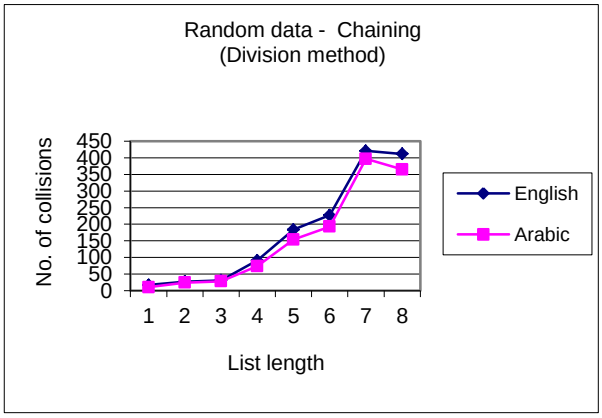

Figure: (4 - c)

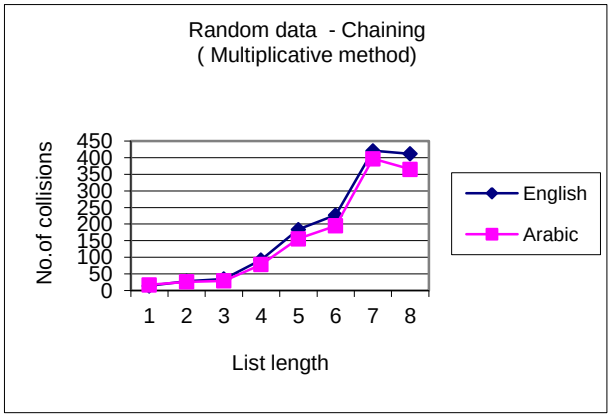

Figure: (4 - d)

Figure (4) displays number of collisions against list length. Although we used different random number generator, the performance for Arabic language text is shown to be better.

Third experiment

In this experiment we compare random strings generated from both the English and Arabic character sets using random function I in appendix A and hash function IV in appendix A. The experiment is repeated three times with different list lengths for table

sizes 20, 40, 80, …, and so on. Averages were computed for each table size. The number of collisions is summarized in figure (5).

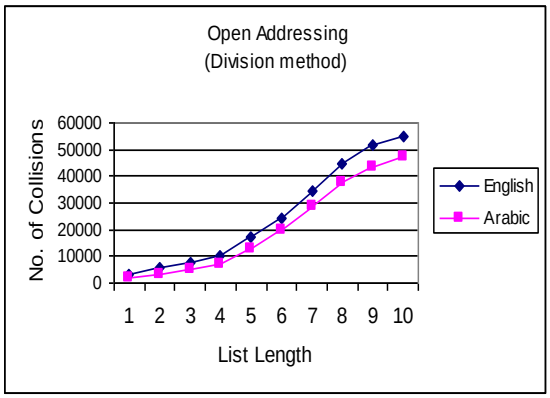

Figure: (5-a)

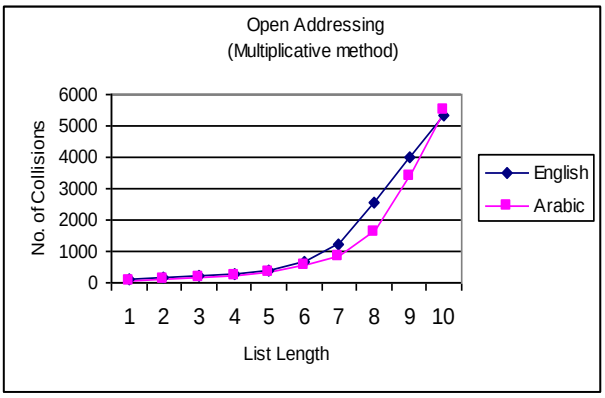

Figure: (5 - b)

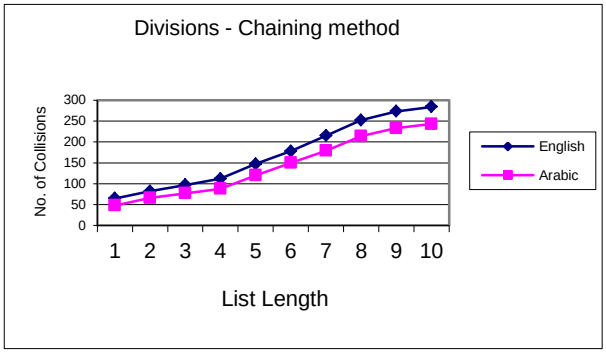

Figure :(5-c)

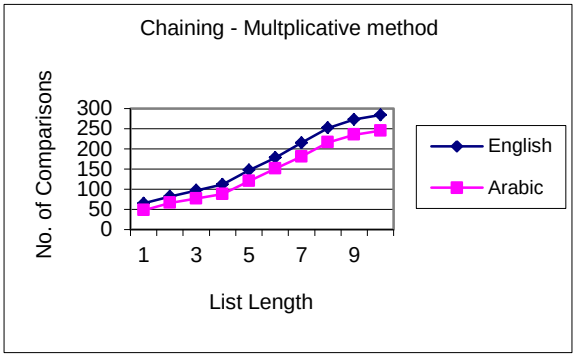

Figure: (5 - d)

Figure 5 shows that all hashing techniques used have better performance on Arabic random text than on English random text.

Fourth Experiment:

This experiment shows the performance of the two hashing methods for each collision resolution technique. It demonstrates to readers the importance of choosing the right hashing technique.

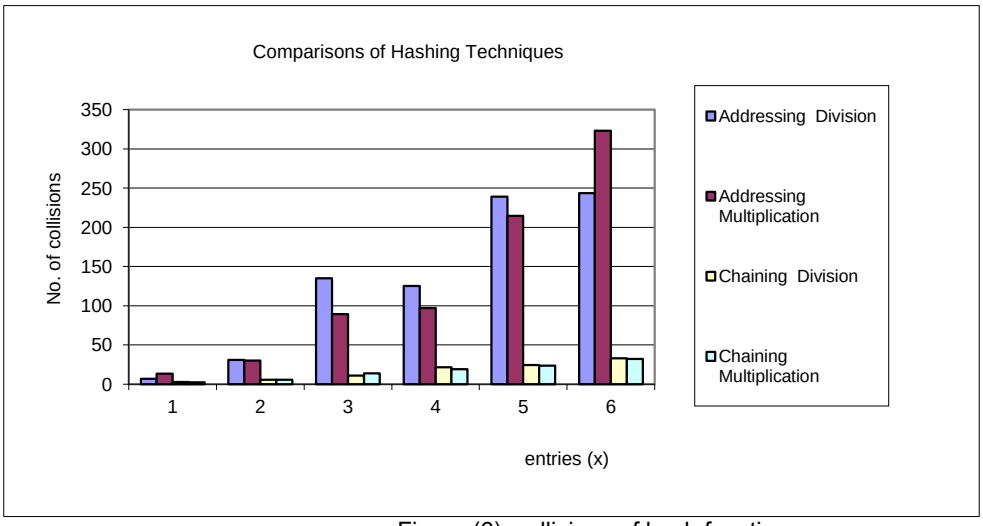

Figure (6): collisions of hash functions

From figure (6) one can see that the chaining technique shows significant performance over the open addressing technique.

Case 2: Real Data Files

In real data file, records are read from text files. A number of experiments has been conducted and analyzed as follows:

First Experiment

Proper Arabic and English names using hash function of the form (see hash function IV appendix A):

$$f(x) = \sum\{ (code (k_j) - i) \times M^{j} \},$$

have been generated.

The experiment is repeated three times for table-size lengths 20, 40, 80, …, and so on. Averages are sampled for each table-size length. The number of collisions for each method and table size has been recorded.

Results from the experiment are recorded for each collision resolution. A summary of the number of collisions is plotted in figure (7).

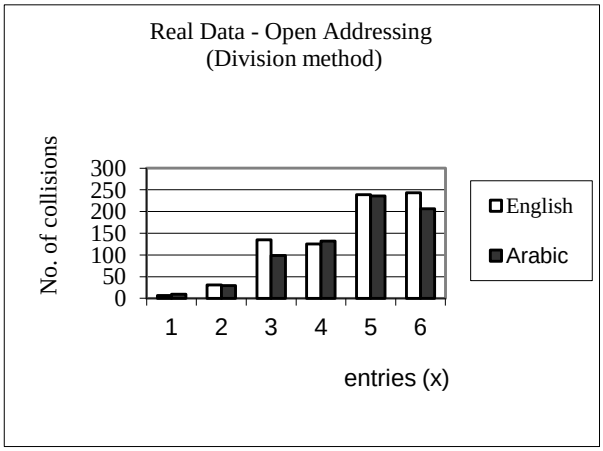

Figure: (7 -a)

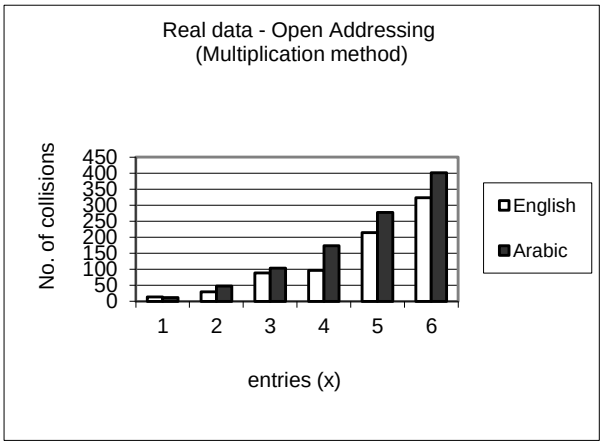

Figure: (7 -b)

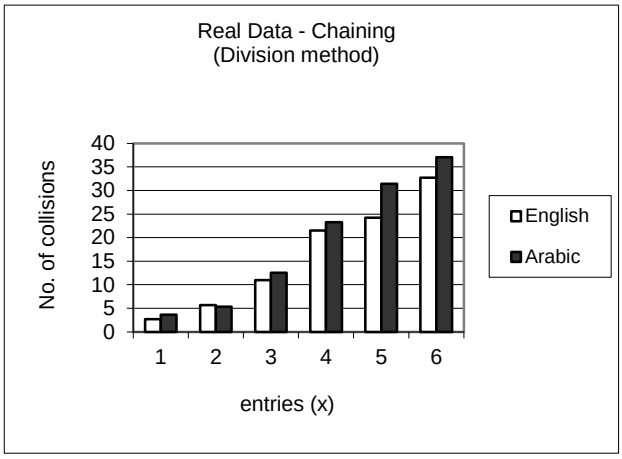

Figure: (7 - c)

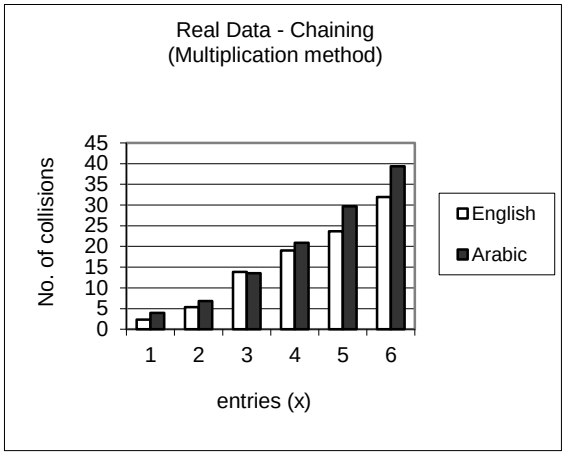

Figure: (7 – d)

From figure (7), one notices that the performance of the hashing algorithm is better on English names than on Arabic names. This fact is due to the dependency in Arabic names. For example, there are many Arabic compound names that start with similar prefixes such as:

عبد الرحيم، عبد الخالق، عبد الجواد، عبد الصمد، ...، الخ

These names have at least five common characters. Arabic language text is unlike other languages. Look, for example, at the following words: library (مكتبة), author (كاتب), writing (كتابة), letter (مكتوب), book (كتاب) . . ., etc. One can realize that there are a least three common letters among all above words. This phenomenon is not available in other languages. This makes the structure of the Arabic language more suitable for computations (derivation processes).

Second Experiment

This experiment shows real English strings and real Arabic strings chosen from newspaper reports dealing with the same subject. The hash function III in appendix A is used. A Summary of the results is plotted in figure (8).

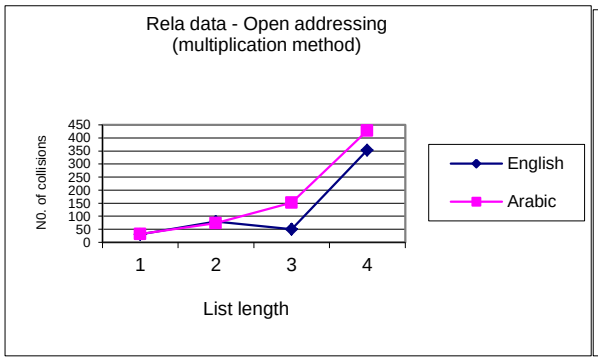

Figure: (8 - a)

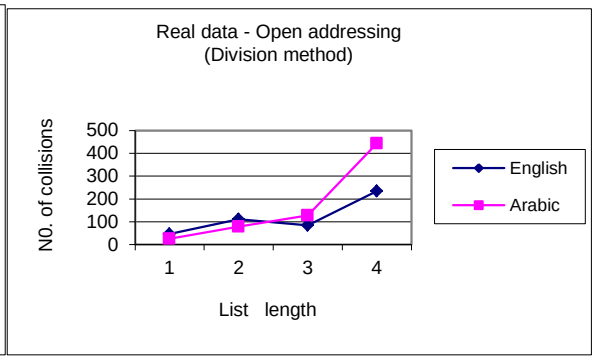

Figure: (8 - b)

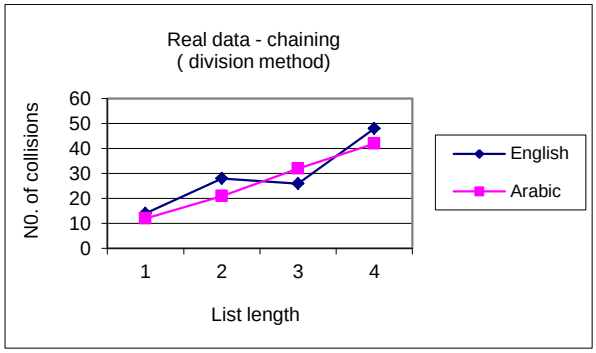

Figure: (8 - c)

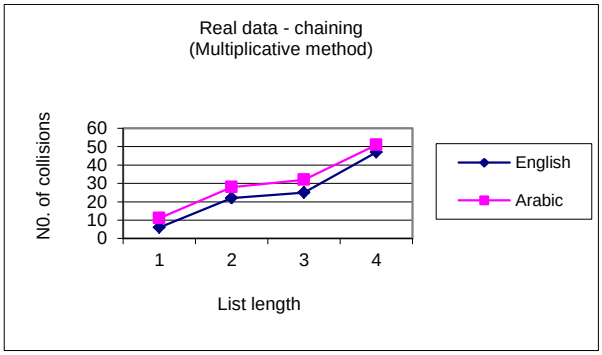

Figure: (8 - d)

From figure (8) one sees that there is a fluctuation in the number of collisions. Figure (8-c) shows that the behavior of the algorithm on Arabic text is steadier than its behavior on English text.

Third Experiment

This experiment shows real English and Arabic strings chosen from two published books. Each chapter is chosen from a book. The hash function in appendix B is used. The experiment considers investigating the behavior of loading factors from 0.1, 0.2, 0.3, …, 1.0 against the number of collisions (Arabic and English) for each method. A Summary of results for Open Addressing Technique – Division Method is plotted in figure (9-a).

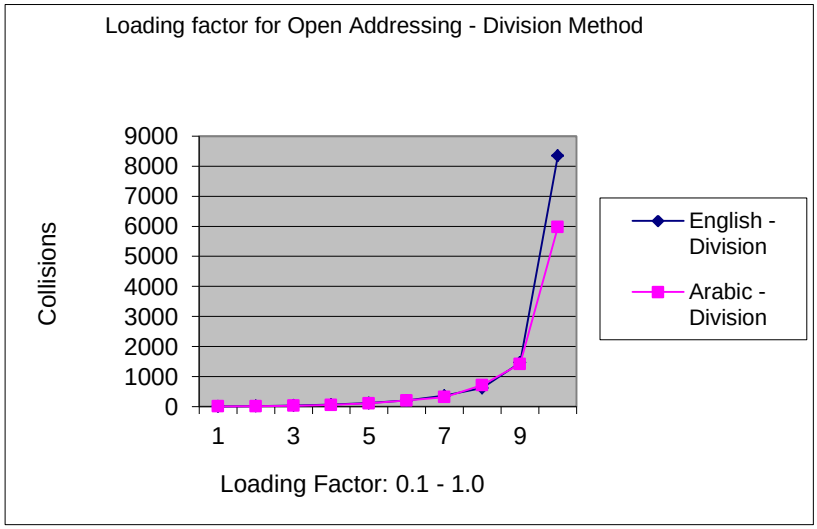

Figure 9-a: Loading factors for Open Addressing Technique

Division Method

A Summary of results for Open Addressing Technique – Multiplicative Method is plotted in figure (9-b).

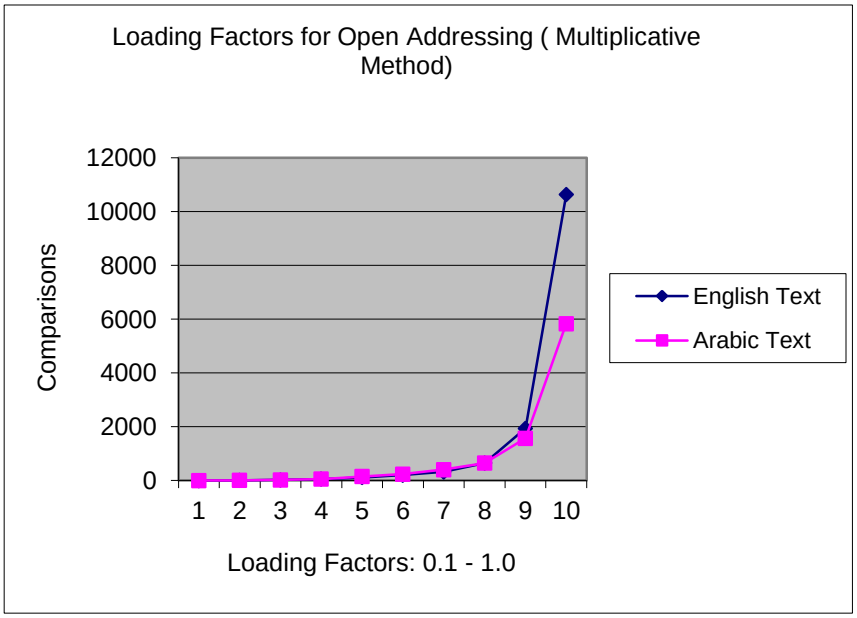

Figure 9-b: Loading factors for Open Addressing Technique

Multiplicative Method

A Summary of results for Chaining Technique – Division Method is plotted in figure (9-c).

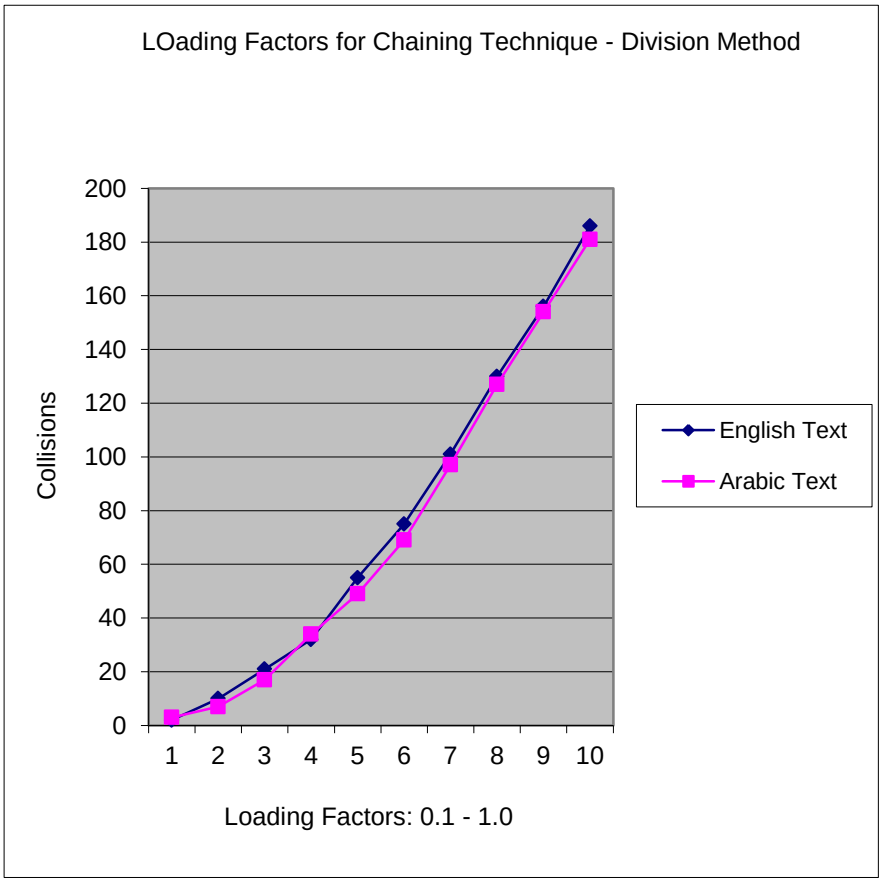

Figure 9-c: Loading factors for Chaining Technique

Division Method

A Summary of results for Chaining Technique – Multiplicative Method is plotted in figure (9-d).

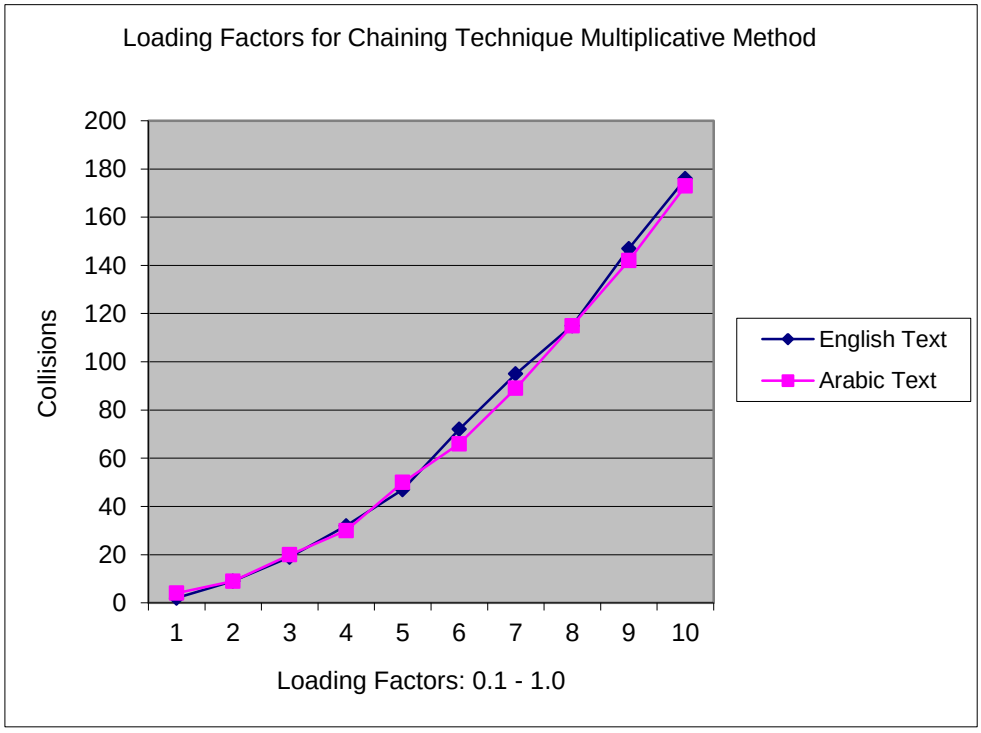

Figure 9-d: Loading factors for Chaining Technique

Multiplicative Method

Several conclusions can be drawn from the above figures. First, it is clear that text generated uniformly from the Arabic language shows fewer numbers of collisions than text from the English language. The performance of the algorithm on uniformly generated Arabic text is due to the fact that the English set is 26 alphabets while the Arabic set is 28 (sometimes 29). In practice, identifiers (keys) have prefixes and suffixes or permutations of other identifier parts. This fact is experienced with Arabic identifiers. One can look for new mechanisms that help to improve the efficiency of hashing methods i.e. eliminating unnecessary prefixes and suffixes.

RECORDS ASSIGNMENT BEHAVIOR

From previous experiments we noticed that it is nearly impossible to accomplish a uniform distribution of records among the available addresses in a table. It is important to be able to predict how records are likely to be distributed. Understanding the behavior of records assignment in advance will help software designer to use the appropriate hashing functions and techniques. When r records are hashed, the probability that an address will be chosen x times and not chosen r-x times can be expressed as:

$$P(x) = C\, a^{r-x} b^{x}, \qquad \text{Where } C = r!/(r-x)!x!$$

C is the formula for the number of ways of selecting x items out of a set of r items. The formula for P(x) becomes

$$P(x) = C(1- 1/n)^{r-x}\,(1/n)^{x},$$ n is the number of the addresses available

This formula, for large values of n and r, can be approximated by poisson function given by the formula $[(r/n)^{x}e^{-(r/n)}/x!]$ [17,18,19, 25]

We can now calculate the expected number of addresses with x records assigned to them. This can be accomplished as follows:

Expected addresses that have x records = TableSize $\times$ P(x)

The expected probabilities for r =100, 150, …, 500 and x= 1,2, 3, …, 6 are tabulated in table 1 Appendix C. For example, when x=1 and r = 100, p(1) = 0.163746. When x > 6, the P(x) is very small and can be neglected so we did not include them in the table. Expected number of addresses is tabulated in Table 2, Appendix C.

Table 3 and Table 4 show the observed number of collisions (Observed number of addresses with x records assigned to them). We used Chi-square (χ^2) test statistic to determine whether the observed proportions differ significantly from the theoretical expected proportions. The calculated Chi-square at the 0.01 level is less than the critical $\chi 2$. We conclude

that the assumptions of no significant difference between the expected and calculated values are accepted. Table 3 and 4 demonstrate that Arabic text shows better performance than English texts.

CONCLUSIONS

Hashing mechanisms and their associated operations are essential in many applications and their associated operations. For instance, efficient searching is required to determine whether a vocabulary exists in a dictionary, inserting new vocabularies and their attributes, and/or deleting specific attributes. Experimental evaluation of hashing techniques shows that they have a higher performance over conventional techniques such as balanced trees.

In this research, empirical study has been conducted using four hashing techniques with different length of data items and different file sizes. Real data was chosen from both languages Arabic and English. Data has also been generated randomly from character sets of each language.

According to the observations of collisions using different data items and different hash tablesizes, a number of cases were analyzed and the following conclusions were drawn from the experiments:

1. Chaining consistently generates fewer numbers of collisions than does open addressing.
2. Multiplication method of chaining generates less number of collisions compared with its counterpart division method of chaining technique itself. This argument is experienced with texts from both languages.
3. Chaining technique performance is always better than open addressing technique.
4. In the case of randomness, experiments show that the performance of the algorithm on Arabic text is better than the performance on English text but on real data the performance on English text is better than on Arabic text.

To improve efficiency we suggest truncating most frequent common parts. For example, in the Arabic language the prefix "AL ـــــــال" is frequently encountered with many identifiers. Also, eliminating common parts can enhance performance of hashing algorithms on Arabic texts.

The higher performance of Arabic texts over English texts, in the case of randomness, is due to the fact that the English language set is 26 letters while the Arabic set is 28.

Several areas for new search may be opened up by the results presented above for adapting new hash functions. Looking for more efficient hash functions may reduce collisions immensely and thus enhancing efficiency.

REFERENCES

1. M. B. Feldman, "Software Construction and Data Structures with Ada 95", Addison-Wesley, 1997.
2. B. S Majewski, N. C. Wormald, G. Havas and Z. J. Czech, " A Family of Perfect Hashing Methods", The Computer Journal, Vol. 39, No. 6, 1996, pp547-554.
3. R. Morris, "Scatter Storage Technique", CACM, 11:1, 1968, PP. 38-44.
4. R. Brent, "Reducing the retrieval time of Scatter Storage techniques", CACM, 16:2, 1973, pp. 105-109.
5. V. Lum, P. Yuen, and M. Dodd, "Key to Address transform techniques: A performance Study on the Large existing format files", CACM, 14:4, 1971, pp. 228- 239.
6. J. Bell, "The Quadratic quotient method: A Hash Code Eliminating Secondary Clustering", CACM, 13:2, 1970, PP. 107-109.
7. W. Maurer and T. Lewis," Hash Table Method", ACM Computing Survey, 7:1, 1975, pp. 5-20.
8. A. Day, "Full table quadratic searching for scatter storage", CACM, 13:8, 1970, pp. 481- 482.
9. P. Larson, "Dynamic Hashing", BIT, 18, 1978, pp. 184-201.
10. R. Enbody and H. Du Schemesm, "Dynamic Hashing", ACM Computing Survey,20:2, 1988, pp. 85-113.
11. H. Mendelson, "Analysis of Extendible hashing", IEEE, Trans. On Software Engineering, se-8, 6, 1982, pp. 611-619.
12. E. Horowitz and S. Sahni, "Fundamentals of Data Structures in C++", Computer Science Press, New York, 1995.
13. R. Cruse, "Data Structures and Program Design", Prentice-Hall, Third Edition, pp.360-377, 1994.
14. M. A. Weiss, "Data Structure and Algorithm Analysis", The Benjamin / Cumming Publishing Company, INC., Second edition, 1994, pp 147 - 172.
15. C. Radke, "The Use of quadratic Residue Research", CACM, 13:2, 1970, pp. 103-105.
16. D. Knuth, "The Art Of Computer Programming: Sorting and Searching, vol. 4, Addison-Wisely, Reading, MA., 1997.
17. J. Bentley, "Programming Pearls: A spelling Checker", Communications of the ACM, Vol. 28, No. 5, May 1985, pp. 456-462.

18. D. J. Dodds, “Pracnique: Reducing Dictionary Size by using a Hashing Techniques”, Communications of the ACM, Vol. 25, No 6, June 1982, pp. 368-370.
19. M. J. Folk, B. Zoellick, and G. Riccardi, “ File Structure: An Object-Oriented Approach with C++, Addison Wesley, 1998.
20. Bruce Mckenzie, R. Harries, and T. Bell, “Selecting a Hashing Algorithm”, Software Practice and Experience, Vol. 20, No. 2, pp.209-224, 1990.
21. Peter K. Pearson, “Fast Hashing of Variable-Length Test Strings”, CACM, 33(6): 677-680, June 1990.
22. R. Uzgalis, “Library Hash Functions”, http://serve.net/buz/hash.adt/java.002.html
23. G. D. Knott, “Hash Functions”, Computer Journal, Vol.18, No. 3, pp.265-278, 1974.
24. Fox, E. A., Chen Q-F., Daoud, A. M., and Heath, L. S. (1991). Order-preserving minimal perfect hash functions and information retrieval. ACM Transactions on Information Systems, 9(3), 281-308.
25. Fox, E. A., Heath, L. S., Chen Q-F., Daoud, A. M.(1992). Practical minimal perfect hash functions for large databases. Communications of the ACM, Jan. 1992, 35(1): 105-121.
26. Jain, R. The Art of Computer Systems Performance Analysis, John Wiley 1991.
27. Turner, E. T. Data Structures: From Recipes to C, Wm. C. Brown Publishers, 1994: 512-515.
28. Banday, M. T. (2016). Improving Information Security Practices Through Computational Intelligence. In "Applications of Digital Signature Certificates for Online Information Security", Awad, W. S., E. M., El-Alfy, and Bastaki, Y., IGI Global, 2016.

APPENDIX A

I. RANDOM FUNCTION

```
function Rand ( Var  Seed : longInt ) : Real;
begin
   Rand := Seed / 65535;
   Seed   := ( 25173 * Seed + 13849) mod 65536;
end;
```

II. RANDOM FUNCTION

```
function Rand(var X: real): real;
const
   a = 16807.0;        (* multilier *)
   m = 2147483647.0;   (* modulus   *)
   q = 127773.0;       (* m div a   *)
   r = 2836.0;         (* m mod a   *)
var
  XdivQ, XmodQ, XNew   : real;
  begin
     XdivQ   := Trunc(x/q);
     XmodQ   := x - q * XdivQ;
     XNew    := a * XmodQ - r * XdivQ;
     if Xnew > 0.0 then x := XNew else x:= Xnew + m-1;
     Rand := x/m;
end;
```

III. HASH FUNCTION

```
function hashfunc ( key : keytype): longint;
     var
```

```
    hash, i, len     : integer;
  begin
    len := length(key);
    hash := len;
    for i:= 1 to len do
      hash := hash + ord(key[i]);
    hashfunc := hash
  end; { hashfunc}
```

IV. HASH FUNCTION

```
function HashFunc ( Var K : KeyType ) : longInt;
 Const
      M = 2;
 var
   J, C, L  :  longInt;
   begin
    L     := Ord ( K[1]);
    C     := 1;
    For   j := 1 to NoOfChars-1  do
     begin
       C := C * M;
       L := L + (Ord ( K[J])-64) * C;
    End;
     HashFunc := L;
 End;
```

Appendix B

Table 1: Loading factors

	Loading Factors								
factor	Length	English Text				Arabic Text			
		OA-Division	*OA-Mult*	*Chaining-Division*	*Chaining-Mult*	*OA-Division*	*OA-Mult*	*Chaining-Division*	*Chaining-Mult*
0.1	50	3	2	2	2	3	4	3	4
0.2	100	14	9	10	9	8	10	7	9
0.3	150	35	25	21	19	24	28	17	20
0.4	200	61	57	32	32	55	54	34	30
0.5	250	116	117	55	47	108	153	49	50
0.6	300	204	204	75	72	199	239	69	66
0.7	350	361	326	101	95	312	408	97	89
0.8	400	617	652	130	115	711	650	127	115
0.9	450	1465	1942	156	147	1416	1570	154	142
1	500	8355	10634	186	176	5976	5834	181	173

Data is drawn from two chapters. One from an English book and the other Arabic book. The experiments have been repeated for different length of record size with fixed Table space. Each record has a length of 20 characters. The table represents number of collisions.

The hash function used is an additive one but with different form

```
Function hash( var K: KeyType): longint;
Begin
        Sum := 100 * Ord(K[1]) + Ord(K[2]);
        L := 1002427; { large prime number}
        N := 3;
        While ( N <= NoOfChars) do
        Begin
                Sum := (Sum + (100 * Ord(K[N]) + Ord(K[N+1]) ) mod L;
                Inc(N);
        End;
        Hash := Sum;
End;
```

Appendix C

Table 1: Expected Probabilities

r	1	2	3	4	5	6
100	0.163746	0.016375	0.001092	5.45821E-05	2.18328E-06	7.27761E-08
150	0.222245	0.033337	0.003334	0.000250026	1.50016E-05	7.50078E-07
200	0.367879	0.053626	0.00715	0.000715008	5.72006E-05	3.81338E-06
250	0.303265	0.075816	0.012636	0.001579507	0.000157951	1.31626E-05
300	0.329287	0.098786	0.019757	0.002963583	0.00035563	3.5563E-05
350	0.34761	0.121663	0.028388	0.004967922	0.000695509	8.11427E-05
400	0.359463	0.143785	0.038343	0.007668548	0.001226968	0.000163596
450	0.365913	0.164661	0.049398	0.011114598	0.002000628	0.000300094
500	0.367879	0.18394	0.061313	0.01532831	0.003065662	0.000510944

Table 2: Expected Number of Addresses with x records assigned to them

r	1	2	3	4	5	6
100	81.87308	8.187308	0.545821	0.027291025	0.001091641	3.6388E-05
150	111.1227	16.66841	1.666841	0.125013075	0.007500784	0.000375039
200	183.9397	26.8128	3.57504	0.357504025	0.028600322	0.001906688
250	151.6327	37.90817	6.318028	0.789753463	0.078975346	0.006581279
300	164.6435	49.39305	9.878609	1.481791417	0.17781497	0.017781497
350	173.8049	60.8317	14.19406	2.483961072	0.34775455	0.040571364
400	179.7316	71.89263	19.17137	3.834273827	0.613483812	0.081797842
450	182.9563	82.33036	24.69911	5.557299037	1.000313827	0.150047074
500	183.9397	91.96986	30.65662	7.664155024	1.532831005	0.255471834

Table 3: Expected number of Collisions observed using data from experiment 3 (Real Data)

Arabic Text

r	1	2	3	4	5	6
100	84	5	2	0	0	0
150	101	19	1	2	0	0
200	126	25	5	1	1	0
250	152	31	5	4	1	0
300	156	44	10	4	2	0
350	165	56	13	6	2	0
400	174	62	12	10	4	1
450	183	74	14	11	3	3
500	184	86	22	10	4	3
Chi-test	0.00922	0.59	0.06218	1.49681E-11	1.80904E-17	7.18378E-17

From Table 3, the average and variance of chi-square goodness of fits are 0.110233 and 0.055829 respectively.

Table 4: Expected number of Collisions observed using data from experiment 3 (Real Data)

English Text

r	1	2	3	4	5	6
100	79	9	1	0	0	0
150	107	18	1	1	0	0
200	124	31	3	0	1	0
250	136	47	5	0	1	0
300	141	61	9	1	0	1
350	151	68	17	0	1	0
400	166	74	23	1	0	1
450	167	82	30	4	0	1
500	175	90	36	6	0	1
Chi-Test	0.000161	0.569425	0.81157	0.11775652	8.6921E-08	2.36175E-12

From Table 4, the average and variance of chi-square goodness of fits are 0.249819 and 0.12446 respectively.