EXTENDIBLE HASHING WITH UNIVERSAL CLASS OF HASHING FUNCTIONS

Sridevi G. M [1] *, *Ashoka D. V.* [2], *Ramakrishna M. V.* [3]

[1] Department of ISE, SJB Institute of Technology and Research Scholar, CSE Research Center, JSS Academy of Technical Education, Bengaluru, Karnataka
[2] Department of ISE, JSS Academy of Technical Education, Bengaluru, Karnataka
[3] Dept. of ISE Res. Center, SJB Institute of Technology, Bengaluru, Karnataka
India

* Corresponding Author, e-mail: sridevi.gereen87@gmail.com

Abstract: Extendible hashing is one of the earliest Dynamic Hashing schemes proposed to handle operations on files that are dynamic in nature. Much research has been published on the scheme during the last four decades. All of them still require the table size to start with a power of 2 and requires the hash function to yield a sequence of bits for each key. In this paper, we show how these two constraints can be overcome with the use of Universal class of hashing functions which simplifies the design complexity. Experimental results showed reduced table split and merge operations thereby reducing the index maintenance cost.

Key words: extendible hashing; H1 universal class of hashing functions; dynamic hashing.

1. INTRODUCTION

There has been a swift growth in the emergence of non-volatile memory (NVM) and Persistent memory (PM) systems. Dynamic hashing techniques have been used widely in databases and non-volatile memory systems for handling huge amounts of data. Dynamic hashing techniques were proposed to organize files whose size varies considerably. Extendible Hashing (EH) is one of the first schemes proposed by Fagin as a file organization technique for handling dynamic files [4]. Other dynamic hashing techniques include Dynamic Hashing by Larson [3] and Linear Hashing by Litwin [6]. In dynamic hashing, growing and shrinking files can be accommodated gracefully by increasing or decreasing the size of the table accordingly, with changes in file size. This dynamic characteristic of extendible hashing technique has driven the possibility of extending the method to improve the performance of NVM and PM systems and to reduce the number of accesses for search. A number of improvements on EH have been proposed since then. But, the basic requirements of EH remain as follows:

The hashing function has to provide hash address as a sequence of bits.

The table size always has to remain a power of 2.

Ramakrishna suggested a method for eliminating these constrictions using universal class of hashing functions [7]. The address requirement satisfied by bit sequence hash addresses for EH is as follows: Consider a key x that hashes to address k when table size is m (where $m = 2d$). When the table of size is doubled to $2m$, x should hash to k or $m+k$. Our idea is that, this requirement can be satisfied by choosing functions from the Universal class of hashing functions even if m is not a power of 2. Consider a class of hashing functions H from A to B, where A is the set of keys and B is a set of addresses. According to Carter and Wegman, H is said to be universal, if no pair of distinct keys from the set A collide under more than $(1/|B|)$ of the functions in $|H|$ [5]. They defined a class of Universal hashing functions H_1, whose members are of the form

$$h(x) = ((cx + d) mod\ p) mod\ m$$

where c and d are randomly chosen integers, $c > 0$ and $d \geq 0$, and x is the input key, p is a prime number, 0 to $m-1$ is the address range. Suppose a key x hashes to address k when the table size is m on choosing a hashing function of the form (1). After doubling the table size to $2m$, the function $((cx+d)\ mod\ p)\ mod\ 2m$ will hash the key x to either k or $m+k$, thus satisfying the addressing requirement of EH. Using functions from H_1 class of hashing functions, simplifies the method by reducing the implementation complexity. The proposed idea also enables the scheme easier to understand. This paper gives detailed algorithm for the proposed improvement on EH technique and the implementation results in comparison with the original method. A version of the bit-less Extendible Hashing based on the proposed idea for a small set of keys using modulo method is presented in [15]. We studied the performance of modulo function on varied table sizes.

In section 2, we provide related literature survey with a summary of progress in research since the original paper in 1979. In Section 3, we describe our hashing functions using the universal class which satisfies the requirements of the EH scheme. Section 4 gives details of implementation. Last section provides conclusions.

2. RELATED WORK

Much work on EH is focused on improving the structure to improve the space utilization and to extend it for flash systems. In this section, we provide a brief explanation on the improvements/extensions to EH. Chung et al. proposed Indexed Extendible Hashing scheme which is a variation of the original Extendible Hashing method [11]. It uses a smaller directory structure and provides better space utilization for non-uniform data. Du et al. proposed a document retrieval system for an office environment which expands multi-key extendible hashing and signature files [26].

They use a variation of Extendible hashing method with three levels. They reported faster response time for queries when compared to earlier design by Tsichritzis et al. [9]. Du et al. proposed the use of multi-level Extendible Hashing for handling operations on larger databases [10]. They propose a multi-level directory structure stored on disk along with a linear directory stored in primary memory. Split operations might be performed on both directory pages and leaf pages. They reported better space utilization using their method when compared to EH and EH tree structures.

Ellis et al. presented the possibility of expanding the Extendible hashing technique for distributed environments [25]. They use directory managers and bucket managers to handle the operations on distributed hash file. The proposed approach uses bucket structures with a link field that is used to link to the new bucket created during split to allow concurrent access to a shared distributed hash file. Fatourou et al. proposed a hash table structure to provide wait-free access modeled on EH [24]. For multiple cored systems, concurrent access to data by multiple threads is needed without affecting other operations. Grampone et al. proposed a data structure, Scalable Virtual Distributed Hashing (SVDH) for distributing computational tasks in the cloud [21]. The proposed structure is modeled after Scalable Distributed Data Structures and EH. SVDH uses EH and LH addressing schemes to split the task among multiple nodes on the cloud. Split and merge operations are performed taking the node capacity into consideration. They concluded that the proposed structure can be a practical new approach for autonomous calculations in the cloud.

Kelley et al. presented the implementation of multi-key Extendible hashing as a data access method [19]. They present a solution for applications where multiple attributes of a record can be a part of a key. Nabil et al. provided an analysis of the performance of Extendible hashing method with elastic buckets and concluded that better space utilization can be achieved by applying partial expansions on the table instead of full table expansions [8]. Nam et al. proposed Cache Line-Conscious Extendible Hashing (CCEH) method for effective utilization of cache-lines in Non-Volatile Memory systems [17]. They proposed the use of segments comprising of a bunch of buckets in place of individual buckets kept track of by the directory, in order to decrease the overhead involved in directory management. Overflow and underflow handling actions on buckets inside a segment are similar to that of original EH. They reported decreased overhead in table management and faster access to data. Ou et al. presented a HyEx Hash which is a hybrid EH-based table structure for a flash-based system [18]. The directory structure uses a hash tree for implementation. The proposed method uses a variable-length list for buckets. They report reduced index cost with regard to data transfer rate and read/write bandwidth on using a partial hash table for each directory.

Per et al. proposed a method for fast backup and restoring data using sorted hashes [1]. Hash values are generated for data to be backed up. On accumulation of multiple hash values, it is compared with hash values of already backed up data to avoid backing up duplicates. The method uses dynamic hashing techniques to implement the structure. Tang et al. presented an improved version of Extendible Hashing method called Limited Multilevel Extendible Hashing for a cluster file system to support large file systems [23]. They use delayed-splitting scheme to decrease the number of split operations to some degree. They use an index file for hash table (one index structure for each hash table entry), and a directory file for buckets. Their method uses reduced number of bits for indexing the data to bucket files with improved space utilization. Wang et al. proposed an index based on Extendible Hashing scheme for flash-based database systems including a split-merge factor to make the structure self-adaptive [22]. With a bucket structure including a log area along with a data area, their design aims at reducing in-place updates in flash storage. Performance of split and merge functions depends on the number of log records. They claim that their design reduces the number of erase operations thereby reducing the maintenance cost of the index.

Baotong et al. address the issue of scaling hash tables to Persistent Memory hardware [16]. The bucket structure for the proposed Dash-EH consists of 14 slots with 32-byte metadata including key data structures required for handling hash table operations to support concurrency control and load balancing. One-byte hashes called fingerprints are used to check the possibility of existence of a key to reduce the number of accesses. Dash-enabled hash tables scaled well on multi-core servers. High load factor could be achieved by using bucket load balancing. Jude et al. proposed a fast access index for handling large number of small files in Hadoop Distributed File System (HDFS) [27]. Large numbers of small files are merged together into a larger file to decrease the access time. The Hadoop Perfect File (HPF) is an index system that uses two hash functions: A dynamic hash function is used to index the metadata of the small files and a perfect hash function is used to preserve the position of individual file's metadata in the index file. HPF provides faster access when compared to original HDFS.

Zouhmeh et al. proposed a write-optimized extendible hashing technique for NVM-DRAM hybrid memory [28]. Their design stores key-value data in NVM while maintaining the directory in DRAM to speed up access. An intermediary layer called segment consisting multiple buckets and a stash for handling overflow items is introduced to balance the directory size and access performance. The method decreases write latency with improved throughput. Yasuhiro et al. proposed an improved Extendible Hashing method to provide concurrent insertion and retrieval operations in a multiprocessor system [20]. They used bucket multi-versioning method to provide concurrency for bucket access. Modification of global depth and directory data is done non-synchronously to decrease clashes on the directory locks. Dedicated and shared locks are used for various operations to allow concurrent split operations. Split operation required fewer updates which enhanced the operation simultaneity.

All these methods use bit sequences to hash the data which restricts the table size to be a power of 2. The need for bit sequences adds additional implementation complexity. An idea to eliminate this restriction was put forward by Ramakrishna

[7]. The idea is to use Universal class of hash functions to achieve the addressing property required to implement EH. A simple modulo based hash function can also be used to satisfy the addressing property required. Next section elaborates on the proposed variations to original method.

3. EXTENDIBLE HASHING WITH UNIVERSAL CLASSES

Fagin's Extendible Hashing technique handles multiple insertions and deletions in dynamic files by gracefully increasing or decreasing the table size accordingly. While the original method provides an efficient solution to handle dynamic files, it uses bit sequences to hash the data which complicates the method. Over time, the original method has been varied to improve the performance in terms of index maintenance, average search length, wait-time during concurrent access, space utilization etc. Most variations involve using storage buckets of variable capacities, delaying the index maintenance operations and so on. All these variations still consider bit sequences for the hash address which complicates the implementation and constricts the table size to a power of 2. In this section, we present a general statement of the addressing property required for the original method and present a simpler alternative that achieves the required property without affecting the performance. The proposed variation eliminates the bit sequences altogether and allows the table size to be chosen without any restriction on it being a size of power 2.

3.1. Addressing Property of the original method

Traditional Extendible Hashing technique uses a table/ directory of links to buckets of fixed storage capacity. To make the structure dynamic, bucket overflows are handled by splitting the bucket into 2 and doubling the table if the global depth (gd) of the table is same as the local depth (ld) of the overflow bucket. Underflow is handled by merging the bucket with its buddy bucket which may result in halving the table. To achieve this, Fagin uses hash addresses coming in as a bit of sequences. Depending on the gd of the table, b bits are extracted from the bit sequence where $b=gd$. The records corresponding to the key are inserted into the bucket pointed by the table entry matching the b bits of the address. Local depth (ld) of the bucket indicates the number of bits of the hash address that matches with the table index. This is illustrated in the following example. We have chosen to extract b Least Significant Bits to map the keys to the buckets. Consider a table with gd 2 and bucket capacity 2. Global depth 2 indicates a table of size $m= 4$ with indexes *{00, 01, 10, 11}*. Suppose we hash a sample set of keys $X = \{947,783,951\}$ into the table. Assuming that the hash function $h(x)$, where $x \in X$, generates bit sequences of the key, as hash address, we get

$$h(947) = 1110110011$$
$$h(783) = 1100001111$$
$$h(951) = 1110110111$$

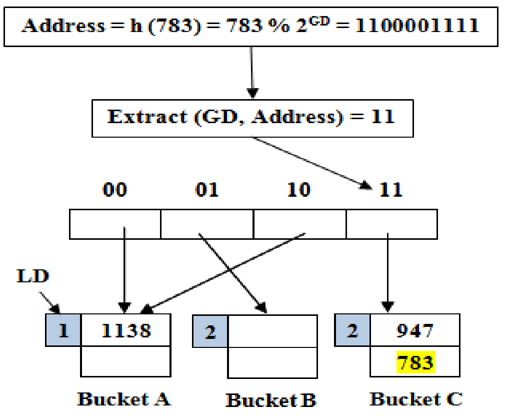
Address = h (783) = 783 % 2^GD = 1100001111
Extract (GD, Address) = 11
00
01
10
11
LD
1
1138
2
2
947
783
Bucket A
Bucket B
Bucket C

Fig. 1. Directory in original Extendible Hashing method with global depth gd=2. The hash function generates hash address in bit format and 2(GD) least-significant bits are extracted to map the keys to buckets.

Since gd is 2, last 2 bits of the bit sequence are taken into consideration to map the keys to the buckets. Thus, all the keys hash to the bucket ending with binary index 11 (Fig. 1). On insertion of the key 951, the bucket overflows and splits. As the local depth of the overflow bucket is also 2, it results in the expansion of the table to size 2m= 8, which increments gd to 3. On expansion, the keys in the over flown bucket are hashed to either the bucket at table index 011 or to the bucket at table index 111 taking 3 bits into consideration (Fig. 2).

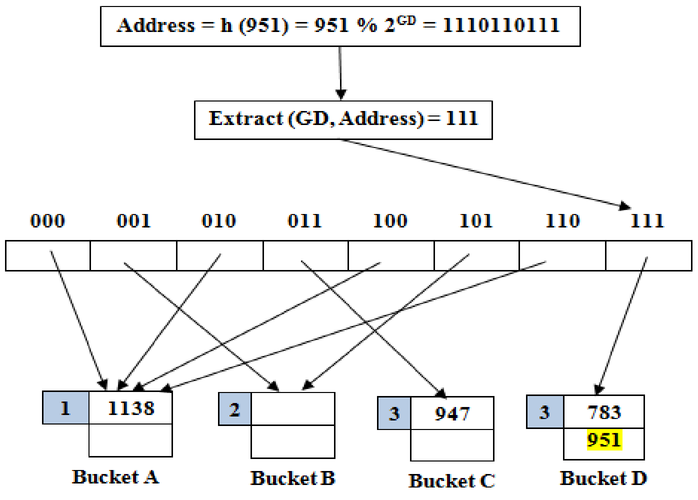

Fig. 2. Directory in original Extendible Hashing method on expansion of the directory to global depth gd=3. 3 least-significant bits are extracted to map the keys to buckets.

It can be observed that the key that hashes to a bucket at index ending with binary 11(3 in decimal) for table size 4, should hash to a bucket at index ending with either binary 011(3 in decimal) or with binary 111(7 in decimal) on expansion of the table to size 8.

In general, the main addressing requirement of traditional EH can be stated as follows: A key that hashes to bucket at index i for table size m, should hash to a bucket at either index i or index i+m on expansion of the table to size 2m. The original method easily achieves this by using bit sequences. The consequence of making use of bit addresses is that the table size will always be a power of 2 i.e. 2gd. This restriction can be eliminated easily by using Universal class of hash functions.

3.2. H$_1$ Universal Class of Hash Functions

Consider a class of hashing functions $H = \{h_1, h_2, h_3 \ldots\ldots\}$ where each h_i is a hash function $A \rightarrow B$ where $A = \{x_1, x_2, x_3 \ldots.\}$ is a set of possible keys and $B = \{0, 1, 2, \ldots.\}$ is a set of memory indices. H is said to be universal, if no pair of distinct keys $x, y \in A$, collide under more than $\left(\frac{|H|}{|B|}\right)$[14]. Carter and Wegman [5] proposed a Universal class of hashing functions *H₁* which is defined as follows:

$$h(x) = \big((cx + d) mod\ p\big) mod\ m \qquad (1)$$

Where p is a prime number, $c > 0, d \geq 0$ and $c, d < p$ where *c* and *d* are chosen at random. The addressing property required for EH can be attained by using a hash function from *H₁* class. Using $h_d(x) = ((cx + d) mod\ p) mod\ m$ at depth *d* for table size m and $h_{d+1}(x) = ((cx + d) mod\ p) mod\ 2m$ at depth *d+1* on expansion of the table to size *2m*, the required addressing property can be achieved as shown in the example below.

Consider a table of size *m=3* and bucket capacity *b = 2*. The values of   and   are chosen at random as *c = 345* and *d = 247* and p is a prime number *p = 79*. The keys 2458 and 1247, hash to the bucket at index *i=1* (Fig. 3). The bucket at index i = 1 overflows on insertion of the key 2759, the table expands to   as the number of links to the overflow bucket is 1. The keys are rehashed taking the table size as   to either the bucket at index 1(i) or to the bucket at index 4 ($i + m$) (Fig. 4).

By using Universal hash functions of the form *((cx + d) mod p) mod m, ((cx + d) mod p) mod 2m, ((cx + d) mod p) mod 4m....* etc, EH can be implemented without the need for address coming in as bit sequences. Hence, table expansion and contraction operations can be implemented gracefully without restrictions on table size and bit sequence hash functions. It must be noted that the table size cannot go lesser than the initial table size chosen m. Also, the table can be of any size m with no restrictions for it to be a power of 2. The proposed idea can also be implemented using a simple modulo method of the form h(x) = x mod m,x mod 2m,x mod 4m .... and is presented in [15].

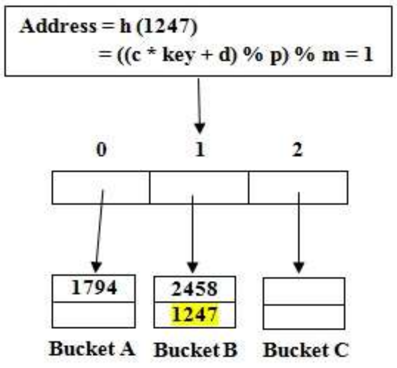

Fig. 3. Directory in proposed Extendible Hashing method with m=3. The hash function generates addresses in decimal format to map keys to buckets.

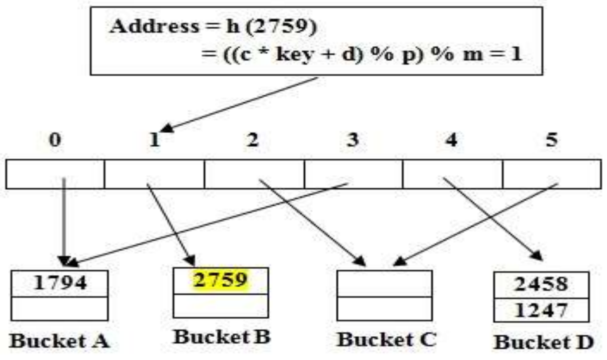

Fig. 4. Directory in proposed Extendible Hashing method on expansion of the directory to m=6.

As the underlying concept of overflow handling by splitting and table expansion; underflow handling by merging and table contraction remain the same as that of the original method, we present algorithms only on the variations to the original method. Elimination of bit sequences eliminates the need for gd and ld. Hence, the structure for the table is modified slightly with the exclusion of the gd. The bucket structure is varied, with inclusion of a field: lcount, indicating the number of table entries/ links pointing to it. Table 1 shows the list of terms used in the algorithms:

```
struct bucket
{
long unsigned int data[B];
int lcount; // number of table links to the bucket
int count; // number of keys inserted to the bucket
}
```

Table 1. List of terms used in the algorithms.

Term	***Description***
DIR	*Table or Directory Array with links to buckets*
DIR[I]	*Bucket at Table index 'I'*
DIR[I]→lcount	*Number of links to the bucket at index 'I'*
M	*Table size*
B	*bucket capacity*

3.3. Algorithms

If a bucket at index 'I' overflows, a new bucket is created. The table is doubled if the number of links to the bucket is 1. Otherwise, half the links to the over flown bucket are linked to the new bucket. The algorithm for the operation is given below.

```
Function Split_Bucket(DIR[I]) {Assuming that the bucket at index 'I' overflows};
array  DIR : an array 0..... m-1 of pointers to buckets
var    m, i, j, diff : integer;
begin
  if (DIR[I] → lcount== 1)
          Double_Table();
  else
          diff := m / DIR[I] → lcount;
          k := I ± diff;
          repeat
                  DIR[k] = New_Bkt;
                  k := k ± (diff*2);
          until (k >= 0 AND k < m)
  end else;
end.
```

On expansion of the table, the entries from index *m* to *2m-1* are updated to entries from *0* to *m-1* respectively. The algorithm for expansion of the table is given below.

```
Function Double_Table()
var    i, j : integer;
begin
  i:= m; j := 0;
  repeat
          DIR[i] = DIR[j];
          i := i + 1; j := j + 1;
  until i < 2m-1;
  DIR[I+m] := New_Bucket;
  m := 2m;
end
```

In case of underflow of a bucket, it can be merged with its buddy bucket. The algorithm to find a buddy bucket for the bucket at index 'I' is shown below.

```
Function Find_Buddy(I)
var    i, j : integer;
begin
  i := I ± (m/(2*DIR[I] → lcount));
  if ((i < m AND i >= 0) AND cur→ lcount== DIR[i]→ lcount)
          buddy: = DIR[i];
  else; // Buddy not found
end
```

Other forms of Universal hash functions proposed include Krovetz's approach to construct variationally Universal hash functions [12]. Brass pointed out that the Universe as defined by Carter and Wegman is finite and presented a family of Universal hash functions for an infinite Universe [13]. Ramakrishna proved that Universal class of hash functions provide practical performance on real files as postulated by theoretical analysis [7]. Using a function from this class makes it easier to implement EH and is simpler to understand. Eliminating bit sequences reduces the complexity in the implementation of the technique. The result of experiments along with the implementation details of the proposed idea is presented in the next section.

4. RESULTS AND DISCUSSION

This section presents the experimental setup, implementation details and results of experiment tabulating the performance comparison between the traditional extendible hashing (EH) method and the extendible hashing method using Universal class of hashing functions or modulo method.

4.1. Experimental Setup

The experiments were conducted on an Intel(R) Core (TM) i3-3110M CPU @ 2.40GHz system with 4GB RAM and 64-bit OS, x64-based processor on Windows 8 Pro operating system.

The performance of the proposed improvement on EH against that of original method was studied by performing hash operations on five different key sets each consisting of 7000 keys. Most hashing functions work on numerical values to hash the data and hence we chose unsigned long integers, generated at random, as input to our hashing function. We chose 50 hashing functions $(h_1, h_2, \dots . h_{50})$ from H_1 class by varying c and d values using a pseudo-random number generator. The value of the prime number for the hash function was kept constant at $p = 2140000007$. Each key set was hashed separately using each of the fifty hash functions. The average performance of traditional EH against EH with Universal Hash functions and EH with modulo method for 7000 keys in terms of number of split operations due to overflow and number of merge operations due to underflow, space utilization and average successful search length was evaluated. Space utilization indicates the percentage of buckets filled and is evaluated using (2). Average successful search length indicates the average number of accesses required to search for all the keys in the buckets.

$$\text{Space Utilization} = \frac{No.\ of\ keys\ inserted}{No.of\ buckets * Bucket\ capacity} * 100 \qquad (2)$$

The performance comparison for initial table size *m = 2*, with bucket capacity *b = 50* is shown in Table 2 and for bucket capacity *b = 100* is shown in Table 3. The result shows that Universal hash functions and modulo methods provide practical performance similar to that of traditional EH when the initial table size is chosen to be *m = 2*. For buckets with larger capacity, hashing the keys to the bucket instead of storing the data linearly shows significant decrease in the number of accesses to the bucket required to access the search element. Storing the data linearly also increases the number of shift operations due to deletion of data which is avoided by hashing the data to buckets. Shift operations require a large number of erase operations on NVM systems which results in performance degradation. Collisions are handled using open addressing technique as explained by Tenenbaum [2], where the next empty slot available in the bucket is used to store the overflow record. Elimination of the restriction on initial size *m* of the table to be a power of 2 allows the user to choose any initial value for *m*.

Table 2. Performance Comparison between traditional EH and EH with Universal Hash Functions and modulo method for initial directory size m = 2 and B = 50.

Method	***Split Count***	***Merge Count***	***Space Utilization***	***Average Search Length***	***Shift Operation count***
Traditional EH	*217*	*84*	*63.927*	*17.416*	*70150*
EH with Universal Hash Functions (Linear Storage)	*211.02*	*71.28*	*65.735*	*23.059*	*50815*
EH with modulo method (Linear Storage)	*209*	*68*	*66.35*	*23.51*	*50546*
EH with Universal Hash Functions (Hashing to buckets)	*216*	*81*	*64.18*	*3.29*	*0*
EH with modulo method (Hashing to buckets)	*217*	*85*	*63.9*	*3.34*	*0*

Table 3. Performance Comparison between traditional EH and EH with Universal Hash Functions and modulo method for initial directory size m = 2 and B = 100.

Method	***Split Count***	***Merge Count***	***Space Utilization***	***Average Search Length***	***Shift Operation count***
Traditional EH	*112*	*50*	*61.4*	*32.95*	*13610*
EH with Universal Hash Functions (Linear Storage)	*107.06*	*39.86*	*64.21*	*44.5*	*93030*
EH with modulo method (Linear Storage)	*107*	*40*	*64.22*	*44.47*	*92876*
EH with Universal Hash Functions (Hashing to buckets)	*112.79*	*49.54*	*61*	*4.73*	*0*
EH with modulo method (Hashing to buckets)	*112*	*50*	*61.4*	*4.94*	*0*

We studied the performance of the proposed EH with Universal hash functions and modulo method for varied sizes of the table for the values m = {3, 5, 7, 11, 13} and the results of comparison for bucket capacity B = 50 are presented in Table 4 and for bucket capacity B = 100 are presented in Table 5. While larger initial table size decreases the number of table maintenance operations, the table itself might grow exponentially.

Table 4. Performance of EH with Universal Hash Functions (UH) and EH with modulo function (Mod) for different initial Table sizes for bucket capacity B = 50.

Initial table size	***Split Count***		***Merge Count***		***Space Utilization***		***Average Search Length***	
	UH	***Mod***	***UH***	***Mod***	***UH***	***Mod***	***UH***	***Mod***
2	*216.18*	*217*	*81.71*	*85*	*64.18*	*63.92*	*3.29*	*3.34*
3	*193.08*	*190*	*37.57*	*35*	*71.42*	*72.54*	*3.15*	*3.08*
5	*182.28*	*185*	*27.02*	*32*	*74.87*	*73.68*	*3.47*	*5.43*
7	*215.21*	*212*	*80.59*	*79*	*63.08*	*63.92*	*3.03*	*3.28*
11	*180.69*	*170*	*24.51*	*17*	*73.30*	*77.34*	*3.30*	*3.34*
13	*201.72*	*193*	*60.19*	*60*	*65.45*	*67.96*	*3.06*	*3.15*

5. CONCLUSION

In four decades, since the original Extendible Hashing paper, scores of papers have appeared on various aspects. But, the original requirement of the hash table size to be a power of 2 and bit sequence hash addresses still exist. In this paper, we used

Universal hashing function to eliminate these two restrictions of traditional EH. This variation of EH simplifies the table structure reducing the implementation complexity and making it easier to understand. We compared the proposed variation of EH with the traditional EH. The results obtained indicate that the proposed approach performs asymptotically similar to that of the original method with an added benefit of reduced split and merge count for directory size 2. Performance of modulo function and functions from H_1 class are found to be highly similar. When prime numbers are chosen as the initial table size, the number of split and merge operations reduces greatly thereby reducing the index maintenance cost. The average search performance showed significant improvement by hashing the keys to buckets instead of storing the data linearly. The proposed method can be extended to Non-Volatile memory systems to improve the search performance and decrease the index maintenance costs.

ACKNOWLEDGEMENT

This research was supported by Visvesvaraya Technological University, Jnana Sangama, Belagavi.

REFERENCES

[1] Per, Yuri S., Maxim V. Lyadvinsky, and Serguei M. Beloussov. System and method for fast backup and restoring using sorted hashes. *U.S. Patent* No. 9,436,558. 6 Sep. 2016.

[2] Tenenbaum, Aaron M. *Data structures using C*, ISBN: 978-81-317-0229-1, Pearson Education, India, 1990.

[3] Larson, P.Å. Dynamic hashing. *BIT Numerical Mathematics*, ISSN: 1572-9125, vol. 18, no. 2, 1978, pp.184-201, https://doi.org/10.1007/BF01931695.

[4] Fagin R, Nievergelt J, Pippenger N. and Strong, H.R. Extendible hashing—a fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)*, ISSN: 0362-5915, vol.4, no.3, 1979, pp.315-344, https://doi.org/10.1145/320083.320092.

[5] Carter J.L. and Wegman M.N. Universal classes of hash functions. *Journal of computer and system sciences*, ISSN: 0022-0000, vol.18, no.2, 1979, pp.143-154, https://doi.org/10.1016/0022-0000(79)90044-8.

[6] Litwin, Witold. Linear Hashing: a new tool for file and table addressing. *In VLDB*, vol. 80, 1980, pp. 1-3.

[7] Ramakrishna M.V. Hashing practice: analysis of hashing and universal hashing. *ACM SIGMOD Record*, ISSN: 0163-5808, vol.17, no.3, 1988, pp.191-199, https://doi.org/10.1145/971701.50223.

[8] Hachem Nabil I. An approximate analysis of the performance of extendible hashing with elastic buckets. *Information processing letters* 48, ISSN: 0020-0190, no. 1, 1993, pp. 13-20, https://doi.org/10.1016/0020-0190(93)90262-8.

[9] Tsichritzis, D. and Christodoulakis, S. Message files. *ACM Transactions on Information Systems (TOIS)*, ISSN: 1046-8188, vol.1, no.1, 1983, pp.88-98, https://doi.org/10.1145/357423.357429.

[10] Du, D.H.C. and Tong, S.R. Multilevel extendible hashing: A file structure for very large databases. *IEEE transactions on knowledge and data engineering*, ISSN: 1041-4347, vol.3, no.3, 1991, pp.357-370, https://doi.org/10.1109/69.91065.

[11] Chung S.M. Indexed extendible hashing. *Information Processing Letters*, ISSN: 0020-0190, vol.44, no.1, 1992, pp.1-6, https://doi.org/10.1016/0020-0190(92)90246-R.

[12] Krovetz Ted, and Phillip Rogaway. Variationally universal hashing. *Information Processing Letters* 100, ISSN: 0020-0190, no. 1, 2006, pp. 36-39, https://doi.org/10.1016/j.ipl.2005.11.026.

[13] Brass Peter. Universal hash functions for an infinite universe and hash trees. *Information processing letters* 109, ISSN: 0020-0190, no. 10, 2009, pp. 461-462, https://doi.org/10.1016/j.ipl.2008.12.012.

[14] Sridevi, G. M, and M. V. Ramakrishna. A Survey of Hashing Techniques for High Performance Computing. *International Journal on Recent and Innovation Trends in Computing and Communication*, vol.4, no.6, ISSN: 2321-8169, 2016, pp. 619-623, https://doi.org/10.17762/ijritcc.v4i6.2379.

[15] Sridevi G.M, Ashoka, D.V: BLEH: Bit-less extendible hashing for DBMS and hard disk drives. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)* 9, ISSN: 2278-3075, 2019, pp. 2191–2197, https://doi.org/10.35940/ijitee.B7539.129219.

[16] Lu, Baotong, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, ISSN: 1147-1161, 2020, https://doi.org/10.14778/3389133.3389134.

[17] Nam Moohyeon, Hokeun Cha, Young-ri Choi, Sam H. Noh, and Beomseok Nam. Write-optimized dynamic hashing for persistent memory. *In 17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, 2019, pp. 31-44.

[18] Ou, Yang, Xiaoquan Wu, Nong Xiao, Fang Liu, and Wei Chen. HIFFS: A Hybrid Index for Flash File System. *In 2015 IEEE International Conference on Networking, Architecture and Storage (NAS)*, ISBN:978-1-4673-7891-8, 2015, pp. 363-364. IEEE, https://doi.org/10.1109/NAS.2015.7255241.

[19] Kelley, Keith L., and Marek Rusinkiewicz. Implementation of multi-key extendible hashing as an access method for a relational DBMS. *In 1986 IEEE Second International Conference on Data Engineering*, ISBN: 978-0-8186-0655-7, 1986, pp. 124-131. IEEE, https://doi.org/10.1109/ICDE.1986.7266214.

[20] Hirano, Yasuhiro, Fumiaki Miura, and Tetsuji Satoh. Extendible hashing for concurrent insertions and retrievals. *In Proceedings of 4th Euromicro Workshop on Parallel and Distributed Processing*, ISBN:0-8186-7376-1, 1996, pp. 235-242. IEEE, https://doi.org/10.1109/EMPDP.1996.500592.

[21] Grampone Silvia, Witold Litwin, and Thomas Schwarz. An Autonomous Data Structure for Brute Force Calculations in the Cloud. *In 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, ISBN:978-1-4673-9560-1, 2015, pp. 347-354. IEEE, https://doi.org/10.1109/CloudCom.2015.17.

[22] Wang Li, and Hanhu Wang. A new self-adaptive extendible hash index for flash-based DBMS. *In The 2010 IEEE International Conference on Information and Automation*, ISBN:978-1-4244-5704-5, 2010, pp. 2519-2524. IEEE, https://doi.org/10.1109/ICINFA.2010.5512045.

[23] Tang Rongfeng, Dan Meng, and Sining Wu. Optimized implementation of extendible hashing to support large file system directory. *In 2003 Proceedings IEEE International Conference on Cluster Computing*, ISBN:0-7695-2066-9, 2003, pp. 452-455. IEEE, https://doi.org/10.1109/CLUSTR.2003.1253347.

[24] Fatourou Panagiota, Nikolaos D. Kallimanis, and Thomas Ropars. An Efficient Wait-free Resizable Hash Table. *In Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ISBN: 978-1-4503-5799-9, 2018, pp. 111-120, https://doi.org/10.1145/3210377.3210408.

[25] Ellis, Carla Schlatter. Extendible hashing for concurrent operations and distributed data. *In Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems*, 1983, pp. 106-116, https://doi.org/ 10.1145/588058.588072.

[26] Du H. C, Subbarao Ghanta, Kurt J. Maly, and Suzanne M. Sharrock. An efficient file structure for document retrieval in the automated office environment. *In 1987 IEEE Third International Conference on Data Engineering*, ISBN:978-0-8186-0762-2, 1987, pp. 165-172. IEEE, https://doi.org/10.1109/ICDE.1987.7272370.

[27] Tchaye-Kondi J, Zhai Y, Lin KJ, Tao W, Yang K. Hadoop Perfect File: A fast access container for small files with direct in disc metadata access. *arXiv preprint arXiv:1903.05838*. 2019 Mar 14.

[28] Zou, Xiaomin, Fang Wang, Dan Feng, Janxi Chen, Chaojie Liu, Fan Li, and Nan Su. *HMEH: write-optimal extendible hashing for hybrid DRAM-NVM memory*.

Information about the authors:

Sridevi G M – Assistant Professor, Department of Information Science and Engineering, SJBIT, Bengaluru, India. Area of Scientific research: Enhancing Hashing Techniques for High Performance Computing, Data Structures and File Structures.

Dr. D.V Ashoka – Dean Research and Professor, Department of Information Science and Engineering, JSSATE, Bangalore. His fields of interest are Requirement Engineering, Operating System, Computer Organization, Software Architecture, Computer Networks and Cloud Computing.

Dr. M. V Ramakrishna – Emeritus Professor and Research Supervisor, Department of Information Science and Engineering, SJBIT, Bengaluru, India. Area of Scientific research: Perfect Hashing.

Manuscript received on 05 January 2021