BLEH: Bit-Less Extendible Hashing for DBMS and Hard Disk Drives

G M Sridevi, D V Ashoka

Abstract*: *Indexing techniques such as extendible hashing and B-trees are widely used to store, retrieve and search for data on files in most file systems. These techniques have been comprehensively explored to enhance the data structure for increasing the faster access to file contents. Extendible hashing is a dynamic hashing technique which handles dynamic files that keep changing in size. Traditional extendible hashing uses bit addresses to hash the data to buckets and restricts the directory size to be a power of 2 which has corresponding complications in implementation. Restriction on directory size also results in uneven distribution of data which increases the possibility of overflows. This in turn increases the cost of index maintenance. In this paper, an efficient and simpler to implement variation of Extendible hashing method named Bit-Less Extendible Hashing (BLEH) for dynamic files is proposed. The proposed method eliminates the need for binary representation of the hash address which reduces the complexity of implementation. Furthermore, it eliminates the need for the directory size to be a power of 2 providing flexibility in the choice of initial directory size. The experimental results show that the proposed method provides better performance in terms of split count upon insertion when compared to the traditional extendible hashing method with good space utilization.

Keywords* : *Dynamic Hashing, Extendible Hashing, Hashing, Universal Hash Functions.

I. INTRODUCTION

With enormous growth in the field of data science, efficient way of storing and fast retrieval of data has become a necessity in data management. One of the issues is how the data is to be stored on secondary storage such that it can be accessed efficiently. Given a set of data records- each record containing a unique key and other data fields of different data types like numbers, string, dates, we want to retrieve the data in shortest possible time. Search is based on the unique key. Many techniques like indexing, B-Trees, B+ Trees and hashing have evolved over time for providing faster access to data. Among those, hashing is the most efficient file-organization technique that provides constant access to files stored on secondary storage. Hashing takes the key as input and converts it to a fixed length integer which is used as a storage address.

While hashing provides efficient access to static files of fixed size, Dynamic Hashing techniques were introduced to handle dynamic files that keep changing in size. Dynamic hashing schemes include: Extendible Hashing of Fagin et.al[1], Linear Hashing of Litwin[2] and Dynamic Hashing of Larson[3]. Extendible hashing is used in Oracle ZFS [4], IBM GPFS [5], Redhat GFS[6], and GFS2 file systems [7]. All dynamic hashing schemes use bit addressing. Extendible hashing restricts the directory size to be a power of 2 which has corresponding implementation complications. Extendible hashing method was proposed by Fagin et al which is a dynamic structure which expands and contracts gracefully as the file grows and shrinks[1]. They merged the concept of data structures for central memory and access methods for secondary storage devices to provide a file organization technique for dynamic files. For static data, records were stored in a data structure called a hash table which was of fixed size. A hash function generates the storage address by taking the unique key of the record as an input. They made the hash table (usually of fixed size) extendible by separating the hash address space from the directory address space. The directory is an array of pointers pointing to bucket structures of fixed size in which the records are stored. The directory address space was accessed by converting the address generated by the hash function into bits and using 'i' leftmost (most significant bits) bits of the storage address as an index into this directory. When a bucket overflows due to many keys hashing to the same bucket, the bucket was split to accommodate the overflow records. This led to the doubling of directory. Deletion of records led to merging of buckets and reducing the directory size by half. Hence the structure grew or shrunk with insertions and deletions to handle dynamic files. This method uses bit addressing which increases the implementation complexity. It also restricts the directory size to be a power of 2. In this paper, we propose some improvements to the existing method. We propose a simple to understand and implement Bit-Less Extendible Hashing (BLEH) scheme which eliminates the need of bit addresses for hashing. Also we eliminate the restriction on the directory size which facilitates the choice of initial directory size to be chosen as a prime number. Choosing a prime number as the initial directory size distributes the keys more uniformly than a directory of size of power 2 which results in lesser overflows. Rest of the paper is organized as follows. Section II briefs on related work done. Section III briefs on the working of the traditional extendible hashing and introduces the proposed method. Implementation and results of evaluation are shown in Section IV. We conclude the paper in section V.

Revised Manuscript Received on December 30, 2019.

* Correspondence Author
G M Sridevi *, Department of ISE, SJB Institute of Technology, Bengaluru, India.
Dr. D V Ashoka, Department of ISE, JSS Academy of Technical Education, Bengaluru, India

II. RELATED WORK

Several researchers have been investigating the indexing techniques such as EH, B-tress, B+ tress and so on to increase the speedy access to contents of the files. Ellis et al studied the possibility to expand the index structures for distributed data [8]. Victoria Hilford et. al proposed a variation of extendible hashing method for distributed environments [9]. The buckets were spread across different servers and clients could access the data in parallel from different systems. As the directory size in EH can grow exponentially, they suggested collapsing the directory structure in EH with cache tables so that it can be brought to main memory. The cache tables were maintained in client and server systems. Access to data is provisioned by searching the cache tables first to find the server in which the data is likely to be present. Li Wang et.al proposed a new extendible hash index for flash-based DBMS, with Split-or-Merge (SM) factor to make it self-adaptive [10]. Flash memory is characterized by asymmetric read or writes speed and erase-before-write. To avoid in-place update, they proposed converting update and delete operations to log records. Their design uses a bucket with data and log areas. When a bucket is full, split or merge operation is done based on the number of log records. Higher number of log records result in merge, whereas, higher number of data records results in split operation. They compared their design with traditional extendible hash index for flash memory in terms of index- maintenance cost and search cost. Extra search cost is incurred due to addition of log files but erase operations are reduced resulting in reduced index- maintenance cost. Yang Ou et.al proposed a hybrid extendible hash-based directory structure for flash file system called NIS [11]. The directory structure is implemented by hash tree. Each subdirectory in the file system is maintained as a separate hash table. Their design uses a varied-length conflictive list instead of fixed length buckets. Split and merge operations are reduced based on the length of conflictive list. They studied the performance of their design against traditional system in terms of index cost, throughput and read/write bandwidth Panagiota Fatourou el.al proposed a wait-free hash table structure based on extendible hashing [12]. For multi-core systems, it is desirable to allow multiple threads to access and update data concurrently without affecting other data. As extendible hashing allows resizing, split and merge operations without effecting other buckets, they proposed a design based on EH for wait-free update and search operations. Cacheline-Conscious Extendible Hashing (CCEH) method was proposed by Moohyeon Nam et al addressing effective usage of cachelines in byte-addressable persistent memory (PM) systems [13]. As the directory size can increase exponentially in traditional EH they proposed an intermediate layer between the directory and the bucket structures called as segment. A segment is a group of buckets pointed to by the directory. Split and merge operations on buckets within a segment is similar to that of EH. The directory in the proposed system uses bit addressing similar to traditional methods. Kim et al proposed a new hybrid hash index called h-hash for flash based solid state drives to reduce and delay the split operations by using overflow buckets as per update and delete ratio [14]. Kim et al proposed T-hash algorithm based on extendible hashing that exploits triple-state bucket directory for flash storage systems [15]. To avoid in-place updates, they log the insert and update operations. According to workload, they assign overflow buckets to delay split operations. Workload patterns are classified as cold (no identical keys), warm or hot(may have identical keys) and T-hash uses triple state flags. More overflow buckets are assigned for hot buckets and fewer overflow buckets are assigned for cold buckets. Their design showed reduced number of split operations.

III. PROPOSED METHODOLOGY

Extendible hashing is a dynamic hashing technique proposed by Fagin et.al for handling dynamic files [1]. Extendible hashing uses a bucket structure of fixed capacity to store the data records. A directory structure is used to keep track of the buckets. A hash function takes the unique key from the data record and generates a hash address in binary format. For convenience, let us assume that the hash function generates the binary code of the key as the hash address. In this section, we describe the insertion process in traditional extendible hashing for a sample key set.

Key set= {24, 41, 46, 32, 92}

Initial Global depth = 1 (Last 1 bit is used to decide on the destination bucket)

Initial Directory size = $2^{\text{Global\_Depth}}$ = 2

Bucket capacity = 2

Hash function = *h(key)* (generates the binary code of the key as the hash address)

Initial Local depth of buckets = 1

Table I shows the hash address generation for the keys 24, 41, 46 and 32 and the corresponding directory structure after inserting 24, 41 and 46 is shown in Fig. 1. The bucket that overflows is split by creating Bucket 2. As local depth of the overflow bucket is same as the global depth, the directory is doubled. The keys in the overflow bucket are rehashed/ redistributed by incrementing the global depth to 2 as shown in Table II. Fig. 2 shows the directory structure after redistribution of data in overflow bucket. Key 92 hashes to Bucket 0 as shown in Table III. Bucket 0 is full resulting in doubling of directory as local depth of the bucket is same as the Global depth. The keys in Bucket 0 are rehashed as shown in Table IV and the final directory structure is shown in Fig. 3.

Table- I: Inserting 21, 41, 46, 32

Key	*h(key)*	Destination bucket
24	11000	Bucket 0
41	101001	Bucket 1
46	101110	Bucket 0
32	100000	Bucket 0(overflows)

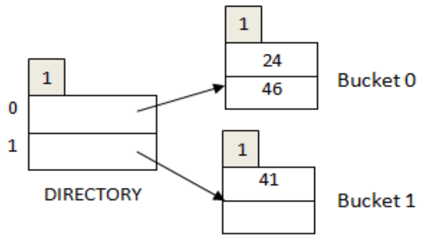

Fig. 1. Directory after inserting 21, 41 and 46

Table- II: Redistributing keys after directory doubling

Key	***h(key)***	**Destination bucket**
24	11000	Bucket 0
46	101110	Bucket 2
32	100000	Bucket 0

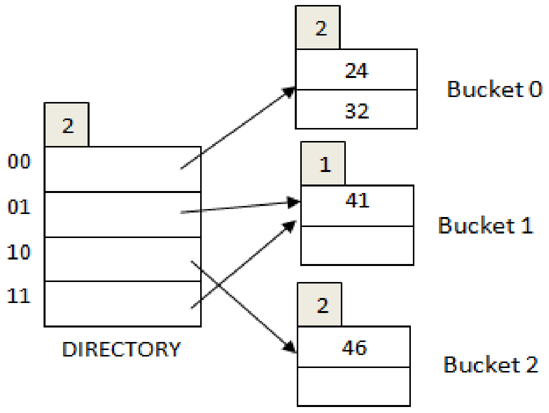

Fig. 2. Directory after inserting 21, 41 and 46

Table- III:Hashing key 92

Key	***h(key)***	**Destination bucket**
92	**1011100**	**Bucket 0(overflows)**

Table IV: Redistribution on insertion of 92

Key	***h(key)***	**Destination bucket**
24	11**000**	Bucket 0
32	100**000**	Bucket 0
92	1011**100**	Bucket 3

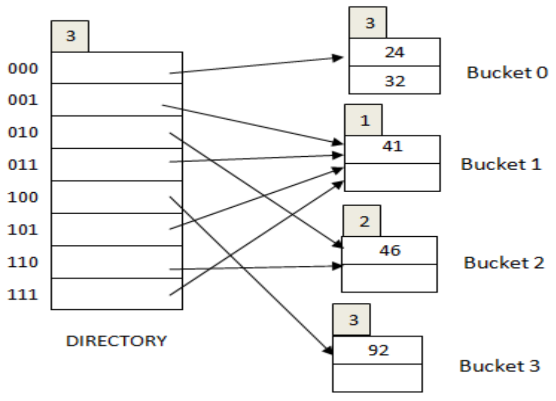

Fig. 3. Directory after inserting 92

In traditional Extendible hashing, extraction of bits for hashing data increases the complexity of implementation and restricts the directory size to be a power of 2. Also, the data is stored in the buckets linearly which increases the number of accesses to buckets thereby increasing the search cost. We propose the elimination of bit addressing without restriction on directory size and also the use of another hash function to hash data into buckets to reduces the number of probes to the bucket during search operation.

A. Proposed Bit-Less Extendible hashing

We propose an algorithm for simpler and enhanced Bit-Free extendible hashing that reduces the complexity of design and implementation with improvements in access time, search time. The proposed design for simpler EH technique avoids bit addressing and does not need the directory size to be a power of 2. The proposed method is flexible with the directory size. We can choose a directory of our choice which is convenient when handling huge amounts of data. The directory points to bucket structures of fixed size in which the records are stored. A hash function hashes the keys to buckets. We chose remainder method to hash the data which is of the form shown in (1).

$$h(x) = key \,\%\, m \qquad (1)$$

where

key= record key taken as input to the hash function,

m is the directory size/ address space.

The hash function generates address in numerical format with no need for binary addressing. The hash function changes dynamically with change in the directory size. When a bucket overflows, it is split and the keys are rehashed between the new bucket and the overflow bucket. Directory size is doubled if the overflow bucket has only one directory entry pointing to it i.e., if the number of pointers pointing to the bucket is 1. If multiple directory entries are pointing to the overflow bucket, then the pointers are distributed between the new bucket and the overflow bucket without doubling the directory. The proposed method is simpler to understand and to implement as it eliminates the need for bit addressing. Flexibility with directory size facilitates in choosing the initial directory size as a prime number which has higher probability of distributing the keys more evenly with less number of collisions. We study the performance of the proposed Extendible hashing in comparison with traditional method.

B. Working of Bit-Less Extendible Hashing

Suppose we choose a sample data set of records with unique keys of integer type. Let us assume a sample key set
Sample key set: { 24, 46, 32, 41, 47, 81, 92, 37, 103 }
For convenience, we assume a small initial directory size as 3 with each entry pointing to a bucket (Buckets A, B and C) in which the data is stored. We have considered the bucket size as 2 which indicates that each bucket can store a maximum of 2 records Hash Function: Key % dir_size (Remainder method). We choose a simple hash function to demonstrate the working of the proposed method.
Table V shows the hash address generated by the hash function and the bucket to which the keys are hashed. Fig. 4 shows the structure after the insertion of 24, 46, 32 and 41.

When 47 is inserted, Bucket C overflows.

Table- V : Hash address generation

Key	Key%dir_size	Bucket
24	24%3 = 0	A
46	46%3 = 1	B
32	32%3 = 2	C
41	41%3 = 2	C
47	47%3 = 2	C(overflows)

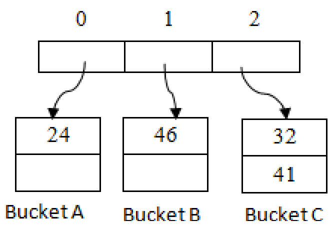

Fig. 4. Insertion into initial directory structure

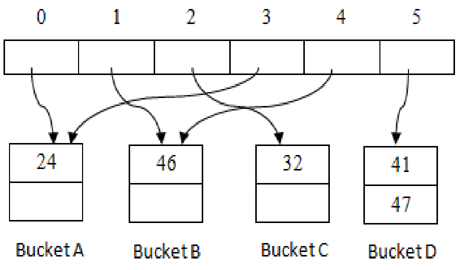

Fig. 5. Directory structure after rehashing/ redistribution

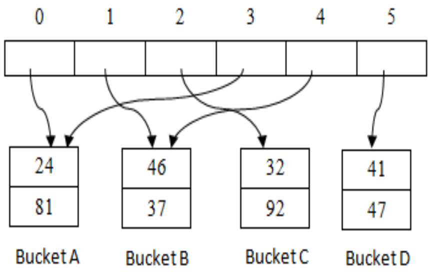

Fig. 6. Directory structure after inserting 81, 92 and 37

Table VI : Redistributing keys in Bucket C

Key	Key%dir_size	Bucket
32	32%6 = 2	C
41	41%6 = 5	D
47	47%6 = 5	D

Table VII : Inserting 81, 92, 37 and 103

Key	Key%dir_size	Bucket
81	81 % 6 = 3	A
92	92 % 6 = 2	C
37	37 % 6 = 1	B
103	103 % 6 = 1	B(Overflows)

Table VIII: Redistributing keys in Bucket B

Key	Key%dir_size	Bucket
46	46 % 6 = 4	E
37	37 % 6 = 1	B
103	103 % 6 = 1	B

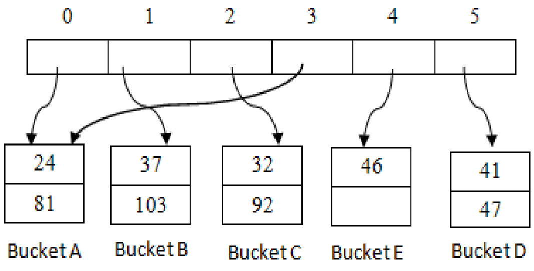

Fig. 7. Directory structure after redistribution

Overflow is handled by splitting the bucket by creating a new bucket. Since bucket C has only 1 directory entry pointing to it, the directory size is doubled and the keys are rehashed with the new hash function as shown in Table VI. The resulting directory structure is shown in Fig. 5. For buckets of larger sizes, a second hash function is used to hash the data into the destination bucket. This results in faster access to data with less number of probes to the bucket during search operation. On inserting 81, 92 and 37 we get the directory structure as shown in Fig. 6. Inserting 103 causes bucket B to overflow (Table VII) resulting in a split operation. Directory doubling is not required as bucket B has 2 directory entries pointing to it. After redistribution (Table VIII), we get the final directory as shown in Fig. 7.

C. Algorithms for the proposed method

Table IX shows the tokens and processes used in the algorithms for insertion and split operations.

Table IX : Tokens and processes used

Token	Description
rec	record to be inserted/searched
rec_key	unique key corresponding to the record
DIR	Directory array
dir_hash(rec_key)	Hash function that generates the index for the destination bucket (dir_index)
bucket_hash(rec_key)	Hash function that generates the bucket index at which the record is to be stored
dir_size	Current directory size
data	Bucket array to store the records/ keys
link	Pointer field in DIR pointing to bucket
no_of_ptrs	No.of pointers pointing to the bucket

Algorithm 1 Insert: The algorithm for insertion is shown in algorithm 1. dir_hash function generates the directory index which points to the destination bucket into which the record is to be inserted. If the destination bucket is full, it is split into 2 buckets.

insert (rec)

```
(1)    dir_index = dir_hash(rec_key)
(2)  cur_bucket = DIR[dir_index]→link
(3)  if (cur_bucket is not full)
(4)  bucket_index = bucket_hash (rec_key)
(5)  if (cur_bucket →data[bucket_index] is unoccupied)
(6)  cur_bucket →data[bucket_index] =  rec_key
(7)  else
(8)  progress to next free space and insert record
(9)  else
(10)  split(cur_bucket)
```

Algorithm 2 Split: Split function creates a new bucket. If the number of pointers pointing to the destination bucket is 1 then the directory is doubled. All the records in the destination bucket are rehashed between the destination bucket and the new bucket.

split (cur_bucket)

```
(1)  create new_bucket
(2)  if(cur_bucket→no_of_ptrs = = 1)//directory doubling
(3)  double the DIR
(4)  for(i = dir_size/2; i<dir_size; i++) //update pointers
(5)  if(i = = dir_index + dir_size/2)   //to new bucket
(6)   (dir + i)→link = new_bucket;
(7)  else
(8)  (dir + i)→link = (dir + i - dir_size/2)→link;
(9)  else  //no directory doubling
(10) diff  = dir_size / cur_bucket→no_of_ptrs;
(11)  i = dir_index - diff, j = dir_index + diff;
(12) while( i ≥ 0 ) //distribute pointers to new bucket
(13)  (dir+i)→link = new_bucket;
(14)   i = i - (diff*2);
(15) while( j < dir_size ) //distribute pointers
(16) (dir+j)→link = new_ bucket;
(17)  j = j + (diff*2);
(18) for each rec in cur_bucket    //redistribute records
(19) insert(rec)
(20)   delete the old record
```

IV. IMPLEMENTATION AND RESULTS

In this section, we present the implementation details and experimental results of performance of the proposed enhanced extendible hashing method (BLEH) against the traditional extendible hashing (EH) method.

A. Experimental Environment

We use a Intel(R) Core(TM) i3-3110M CPU @ 2.40GHz system with 4GB RAM and 64-bit Operating System, x64-based processor. We implemented the traditional and proposed Extendible hashing methods in C++ using Turbo C++ compiler on Windows 8 Pro operating system.

We tested the performance of the proposed Bit-Less extendible hashing method on different key sets by varying the initial directory sizes against traditional extendible hashing method. We choose initial directory sizes to be a prime number as they provide better uniform distribution when compared to other directory sizes. Table X shows the abbreviations used to report the results.

Initial directory size for traditional Extendible hashing = 2

Initial directory sizes considered for proposed Extendible hashing = {3, 17, 19, 23, 29, 31}

Bucket capacity=10

Number of insertion operations performed= Ranging from 100 to 900

Parameters considered for performance evaluation= Number of split operations during insertion, Space Utilization.

B. Split Operations on insertion

We have tabulated the number of bucket splits during insertion for 1 key set in Table XI and results obtained in graph is shown in Fig.8. According to our experimental results, the proposed Bit-Less Extendible Hashing method results in lesser number of bucket splits during insertion operation when compared to the traditional method. Choosing larger initial directory size results in lesser number of bucket splits compared to smaller directory sizes.

Table X : Abbreviations used

Keys	Number of insert or search operations performed
EH	Traditional Extendible Hashing
BLEH3	Bit-Less Extendible Hashing with initial directory size 3
BLEH17	Bit-Less Extendible Hashing with initial directory size 17
BLEH19	Bit-Less Extendible Hashing with initial directory size 19
BLEH23	Bit-Less Extendible Hashing with initial directory size 23
BLEH29	Bit-Less Extendible Hashing with initial directory size 29
BLEH31	Bit-Less Extendible Hashing with initial directory size 31

Table XI : Split count upon insertion

Keys	EH	BLEH3	BLEH17	BLEH19	BLEH23	BLEH29	BLEH31
100	12	12	1	0	0	0	0
200	27	24	10	7	8	2	2
300	42	39	28	22	22	11	13
400	55	54	42	40	33	30	27
500	71	68	50	54	44	44	44
600	81	85	72	66	63	54	57
700	100	98	81	81	78	70	68
800	114	114	95	93	90	89	86
900	130	124	113	110	105	101	97

Lesser split operations mean that the index maintenance cost is lower as there is less number of rehashing/redistribution operations required which increases the index maintenance cost. Larger directory sizes can be used for larger key sets to minimize number of split operations.

C. Space Utilization after insertion

We have tabulated the percentage of space utilized during insertion in Table XII and results obtained in graph is shown in Fig.9. Space utilization indicates the amount of space occupied over the total space available. It is evaluated as shown in (2).

$$\text{Space Utilization} = \frac{\text{Number of keys inserted}}{\text{Bucket capacity} * \text{Number of Buckets}} * 100 \quad (2)$$

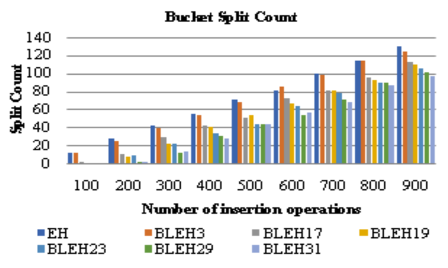

Fig. 8. Split count upon insertion

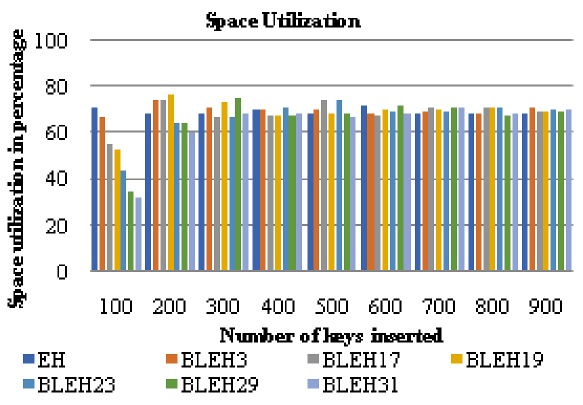

Fig. 9. Space Utilization

Table XII : Space Utilization

Keys	EH	BLEH3	BLEH17	BLEH19	BLEH23	BLEH29	BLEH31
100	71.42857	66.66667	55.55556	52.63158	43.47826	34.48276	32.25807
200	68.96552	74.07407	74.07407	76.92308	64.51613	64.51613	60.60606
300	68.18182	71.42857	66.66667	73.17073	66.66667	75	68.18182
400	70.17544	70.17544	67.79661	67.79661	71.42857	67.79661	68.96552
500	68.49315	70.42254	74.62687	68.49315	74.62687	68.49315	66.66667
600	72.28916	68.18182	67.41573	70.58824	69.76744	72.28916	68.18182
700	68.62745	69.30693	71.42857	70	69.30693	70.70707	70.70707
800	68.96552	68.37607	71.42857	71.42857	70.79646	67.79661	68.37607
900	68.18182	70.86614	69.23077	69.76744	70.3125	69.23077	70.3125

According to our experimental results, the proposed Bit-Less Extendible Hashing method provides similar or better space utilization when compared to traditional extendible hashing method.

V. CONCLUSION

In this paper, we have presented an efficient and simpler to implement variation of Extendible hashing method for dynamic files. The proposed method eliminates the need for binary representation of the hash address which reduces the complexity of implementation. Choosing a prime number as the initial directory size distributes the keys more uniformly than a directory size of power 2. We studied the performance of the proposed method against the traditional method. The results show that the proposed method results in lower split count on insertion and good performance in terms of space utilization, when compared to the traditional extendible hashing method. Lower split operations indicate that the proposed improvements can be extended for SSDs which would be our future work.

VI. ACKNOWLEDGMENT

G M Sridevi thanks Dr. M V Ramakrishna, Research Supervisor at SJBIT, currently residing in Australia for his contribution in the research work and for his guidance in implementation of the proposed work. G M Sridevi thanks Dr. Ajay Prakash B V for reviewing the paper and for his suggestions and assistance in writing this paper. This research was supported by Visvesvaraya Technological University, Jnana Sangama, Belagavi.

REFERENCES

1. Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H Raymond Strong. Extendible Hashing a fast access method for dynamic files. ACM Transactions on Database Systems (TODS), 4(3):315–344, 1979.
2. Witold Litwin. Linear hashing: a new tool for file and table addressing. In VLDB, volume 80, pages 1–3, 1980.
3. P.A. Larson. Dynamic hashing. BIT Numerical Mathematics, 18(2):184–201, 1978.
4. ORACLE. Architectural Overview of the Oracle ZFS Storage Appliance,2018.https://www.oracle.com/technetwork/server-storage/sun-unified-storage/documentation/o14-00-architecture-overview-zfsa-2099942.pdf.
5. F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST '02), Monterey CA, January 2002.
6. Soltis, S. R., Ruwart, T. M., And Okeefe, M. T. The global file system. In Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies (1996), vol. 2, pp. 319—-342.
7. Whitehouse, S. The gfs2 filesystem. In Proceedings of the Linux Symposium (2007), Citeseer, pp. 253–259
8. Ellis, Carla Schlatter. "Extendible hashing for concurrent operations and distributed data." Proceedings of the 2nd ACM SIGACT-SIGMOD symposium on Principles of database systems. ACM, 1983.
9. Hilford, Victoria, Farokh B. Bastani, and BojanCukic. "EH/sup*/-extendible hashing in a distributed environment." Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97). IEEE, 1997.
10. Wang, Li, and Hanhu Wang. "A new self-adaptive extendible hash index for flash-based DBMS." The 2010 IEEE International Conference on Information and Automation. IEEE, 2010.
11. Ou, Yang, et al. "NIS: A New Index Scheme for Flash File System." 2015 Third International Conference on Advanced Cloud and Big Data. IEEE, 2015.

12. Fatourou, Panagiota, Nikolaos D. Kallimanis, and Thomas Ropars. "An Efficient Wait-free Resizable Hash Table." Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures. ACM, 2018.
13. Nam, Moohyeon, et al. "Write-optimized dynamic hashing for persistent memory." 17th {USENIX} Conference on File and Storage Technologies ({FAST} 19). 2019.
14. Kim, Bo-Kyeong, Sang-Won Lee, and Dong-Ho Lee. "h-Hash: a hash index structure for flash-based solid state drives." Journal of Circuits, Systems and Computers 24.09 (2015): 1550128.
15. Kim, Bo-Kyeong, Joo-Young Kang, and Dong-Ho Lee. "A new hash algorithm exploiting triple-state bucket directory for flash storage devices." IEEE Transactions on Consumer Electronics 62.4 (2016): 398-404.

AUTHORS PROFILE

G M Sridevi is working as Assistant Professor in the department of Information Science and Engineering, SJBIT, Bengaluru. She has 7 years of teaching experience. She has M. Tech degree in Computer Science and Engineering and BE degree in Information Science and Engineering from Visvesvaraya Technological University (VTU), India. She is currently pursuing her PhD in Computer Science and Engineering under VTU, India. Her research topic is – Enhancing Hashing Techniques for High Performance Computing She is a member of ISTE and IAENG. Her fields of interest are Data Structures and File Structures.

Dr. D.V Ashoka, is a Professor working in Information Science and Engineering Department, JSSATE, Bangalore. He received his M.Tech in Computer Science and Engineering from VTU, Ph.D degree in Computer Science from Dr. MGR, University, Chennai. He has 23 years of academic teaching and research experience. He has published around 50 papers in reputed international journals like Springer, Elsevier, IEEE, inderscience. He is a member of FELLOW IEI, IEEE, MISTE, MCSI and MIAENG. His fields of interest are Requirement Engineering, Operating System, Computer Organization, Software Architecture, Computer Networks and Cloud Computing.