Yugoslav Journal of Operations Research
7 (1997), Number 1, 65-77

GUARANTEED SINGLE DISK ACCESS FOR VERY LARGE DATABASE FILES

Dejan SIMIĆ
Mihajlo Pupin Institute
P.O. Box 15, 11000 Belgrade, Yugoslavia

Dušan STARČEVIĆ
Faculty of Organizational Sciences
Jove Ilića 154, 11000 Belgrade, Yugoslavia

Emil JOVANOV
Mihajlo Pupin Institute
P.O. Box 15, 11000 Belgrade, Yugoslavia

Abstract: Interactive applications such as expert systems, CAD/CAM and multimedia impose an increasing demand on a data management system that efficiently supports basic operations on very large files and provides data retrieval with a guaranteed single disk access. The synergism of a conventional B^+ tree and a hash function represents a possible solution to the problem. We have developed a class of algorithms that allow a single disk access. The purpose of the paper is to compare and contrast several fast and simple hash functions suggested in the literature that can be used in such a class of algorithms.

Keywords: Algorithms, file structures, B^+-trees, perfect hashing, physical design.

1. INTRODUCTION

Disk and main memory data access time are significantly different. The difference is about three orders of magnitude. At the same time the most often used multimedia applications, expert systems and CAD/CAM applications are interactive by their very nature. Because of that, we should provide the algorithms and data structures with the smallest possible number of disk data accesses. The number of new applications requiring single disk access data retrieval is growing rapidly. It is known that for large files ranging from several hundred megabytes to several hundred gigabytes, different internal organization makes a huge difference in performance [17], as well.

One of the most commonly used data access structures in commercial DBMS applications is the B^+-tree index [1, 5, 7, 18]. This structure allows fast access to a record with a particular key value and efficient retrieval of a set of records within a given range of key values. This is possible because of the order preserving access data structure embedded in B^+-tree algorithms. Moreover, contemporary multi-user database systems often use it as a concurrent search structure [8]. In the B^+-tree structure the data are only in the leaves. The main disadvantage of this structure is the unpredictable number of data accesses to very large database files, which is unsatisfactory for the aforementioned class of interactive applications. In addition, the associated index data structure may be too large to be resident in the available main memory even in the case of moderate size data files.

Hashing is one of the most frequently used data access techniques suitable for guaranteed time retrieval [4, 11, 12]. Although hashing resolves the problem of a single key value retrieval, it is inappropriate for set retrieval operations. In order to guarantee O(1) access time, perfect hash functions should be applied [10]. There are many algorithms supporting large static file organization [3]. Nevertheless, applications that use very large files with a time dependent number of records require dynamic hashing schemes [2, 14].

Recently, a new class of algorithms based on the synergism of a conventional B^+ algorithm and a hash function has been developed to efficiently support basic operations on very large dynamic database files and to provide data retrieval with guaranteed O(1) disk accesses [13, 16, 19]. One of the most important characteristics of very large dynamic database files is the load factor that is defined as the number of records in the file divided by the number of places for records. Since the load factor influences the data unit storing cost it is desirable to achieve as high a load factor as possible. However, a high load factor increases the record insertion cost for dynamic database files. In practice, the folding function or its variation is often used for dynamic files [10, 6, 19]. On the other hand, Pearson proposed a fast and simple hash function specifically tailored to variable-length text strings [15]. Our goal is to investigate the hash function impact on the load factor as a part of the modified B^+ algorithm [19].

Section 2 gives an overview of the modified B^+ algorithm that has been proposed for data retrieval with a guaranteed single disk access for very large dynamic database files. In Section 3 we present the modified folding function and two variants of Pearson's function that can be used in the modified B^+ algorithm. Section 4 contains some important implementation notes. Section 5 presents the load factor experimental results of simulated data with uniform key distribution. Section 6 discusses the characteristics of the suggested hash functions applied to real data with a non-uniform key distribution. In section 7 we conclude the paper.

2. THE MODIFIED B^+ ALGORITHM

We assume that the input file is a large collection of fixed-size records and that every record has a unique key. The problem is to provide an algorithm that guarantees data retrieval with the O(1) disk access where a limited amount of the main memory is available. In addition to O(1) disk access, we are seeking an acceptable performance for

insert and delete operations. In other words, the algorithm should work efficiently not only with static files but also with dynamic files. It is well known that the B^+-tree is the most accepted method which can guarantee data retrieval in a single disk access when the whole index structure resides, in the main memory. For the sake of simplicity, a B^+-tree of order 2 is used as a primary index as illustrated in Fig. 1.

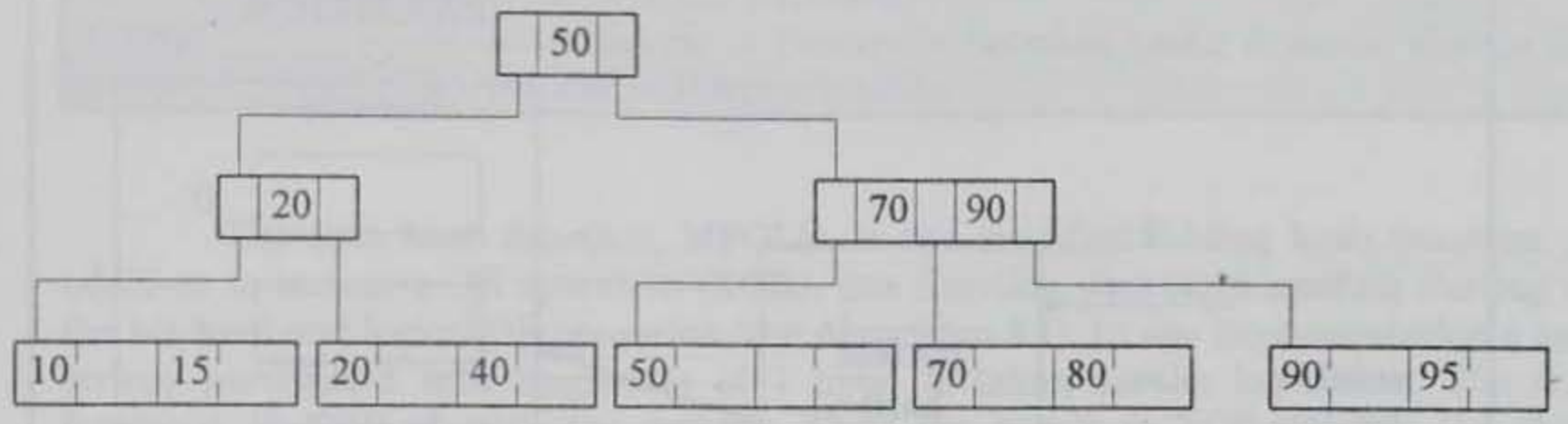

Figure 1. Primary B^+-tree index structure

An algorithm based on a modified B^+ tree organization, which fulfils the aforementioned requirements based on the limited main memory available, is presented in [19]. Unlike the conventional B^+ -tree, the leaves in the proposed algorithm may vary in size. Each pointer *p* addresses the contiguous set of *m* data buckets in the leaf, that is, in the partition. The parent of the leaf node is a collection of triplets (*key, m, p*) where *m* is the number of data buckets and *p* is a pointer to the partition (Fig. 2).

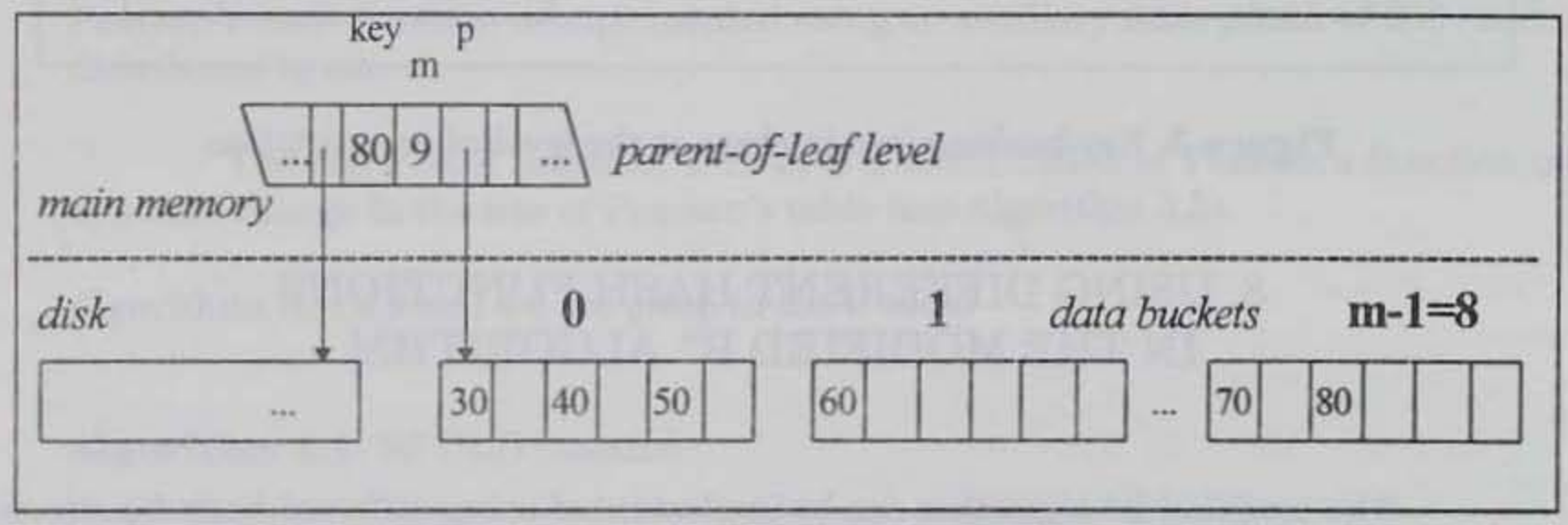

Figure 2. A fragment of a parent-of-leaf and leaf level of the modified B^+-tree

Here, we will explain the search algorithm (how to determine the data bucket within the partition holding the requested record). We make use of the ordinary hash function. The hash function generates an integer in a range [0, *m*-1], which determines the data bucket containing the record with a given key value *k*. The input parameters of the hash function are the key, the number of data buckets in the partition and the key length (Fig. 3). The output value is the data bucket number in the range [0, *m*-1].

Therefore, to insert a record with a primary key, we determine a partition and then the potential data bucket using the hash function at the parent-of-leaf level. If the

data bucket in the main memory becomes full, rehashing of the corresponding partition is necessary. Otherwise, the record will be instantly inserted. After that, we have to write on the disk the data bucket or the partition in the case of the rehashing.

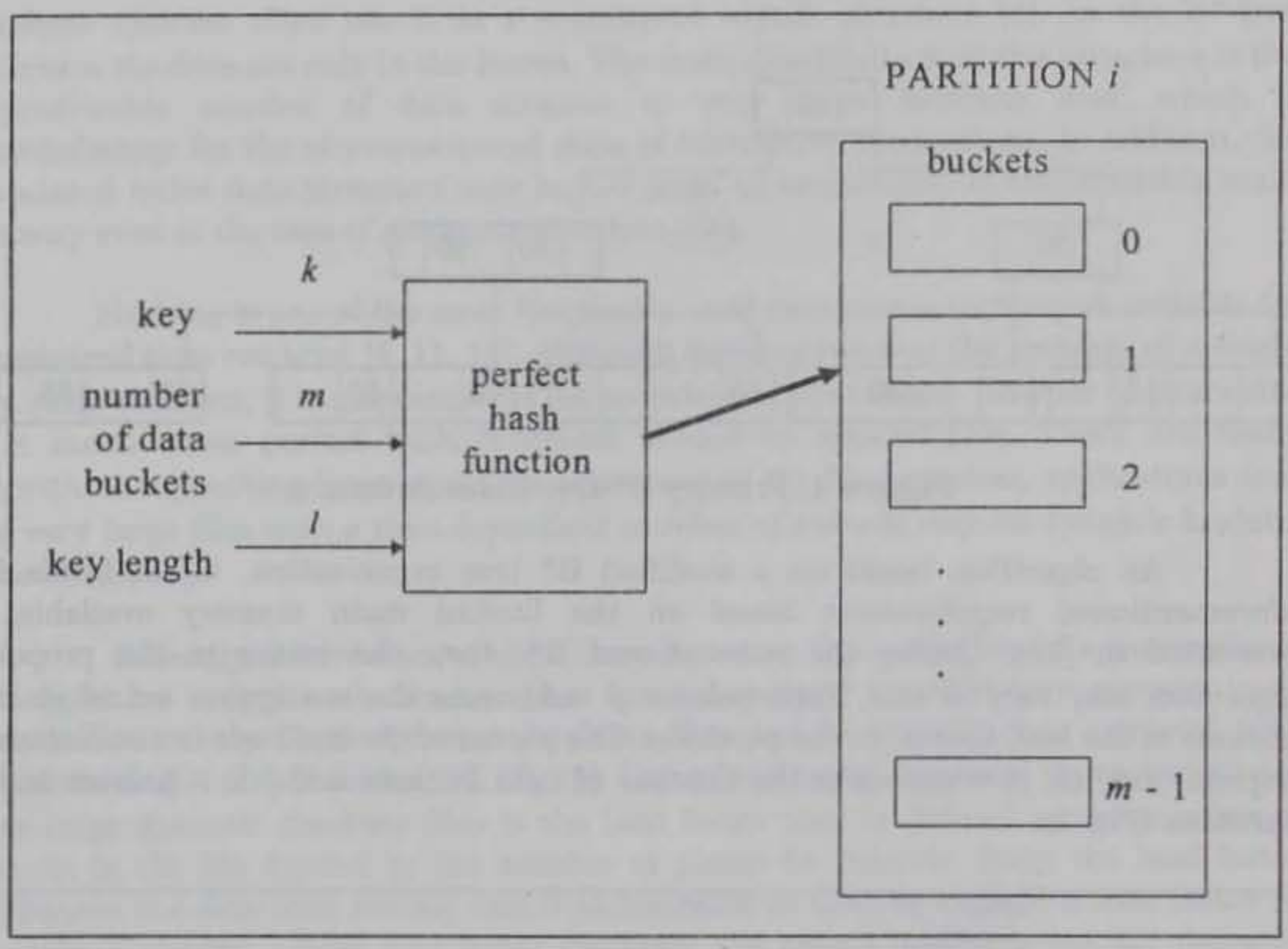

Figure 3. Key hashing into buckets at the level of one partition

3. USING DIFFERENT HASH FUNCTIONS IN THE MODIFIED B+ ALGORITHM

The modified B+ algorithm can be implemented using different hash functions. As the modified B+ algorithm is designed for dynamic database files, the hash function should be simple, fast, intended for variable length records that achieve as high a load factor as possible, while maintaining low record insertion cost. Many functions were suggested in the literature, but the folding function has been implemented in the hardware in some database processors [6, 19]. On the other hand, Pearson proposed a fast and simple hash function specifically tailored to variable-length text strings [15].

In order to investigate the hash function's impact on the load factor as a part of the modified B+ algorithm, we have compared the performances of three hash functions. Table 1 summarizes the hash functions investigated in this paper and the abbreviations we use to identify them.

Table 1. Competing functions

abbreviation	full name
MFOLD	modified folding
PEARS_MOD	Pearson's function with the addition of the MOD number of buckets in the partition
DYNB	modification of Pearson's function using dynamic change in the size of Pearson's table

The first hash function, MFOLD, is the modified folding hash function. In addition to exclusive OR operation (XOR), this function uses right and left shifting on the bit level and logical OR operation (see Algorithm 3.1). In our implementation a byte string, partitioned into fragments of 1 byte, is taken as the key value. The first fragment is XOR-ed with the second. Then the result is XOR-ed with the third fragment, etc. Therefore, any key is folded into a positive 1-byte integer.

Bearing in mind that the original Pearson's function is designed for a fixed number of buckets in a partition, the other two functions are modifications of Pearson's function that support the variable number of buckets needed in the modified B^+ algorithm. Therefore, the second hash function, PEARS_MOD, is the original Pearson's hash function with the addition of the MOD number of buckets in the partition (see Algorithm 3.2). Pearson's hash function is specifically tailored to variable-length strings. This function takes as input a word W consisting of n characters, C_1, C_2, ..., C_n, each character being represented by one byte, and returns an index in the range 0-255. Pearson's hash function is implemented using an auxiliary table *ptable* of 256 randomly distributed bytes.

The third hash function, DYNB, is a modification of Pearson's function using dynamic change in the size of Pearson's table (see Algorithm 3.3).

Algorithms 3.1, 3.2 and 3.3 are given in the C code:

Algorithm 3.1: MFOLD function

```
h = key[0];                          /* Take the first key fragment          */
for (i = 1; i  < key_length; i++) {
    h =  (h << 1) | (h >> 7);        /* Rotate left one position, rotate right
                                        seven positions and OR operation     */
    h =   h ^ key[i];                /* XOR with the next key fragment       */
}
return h%m;
```

Algorithm 3.2: PEARS_MOD function

```
h = 0;                                  /* initialization                                    */
 i = 0;
do {
     h = ptable[(int) (h ^ key[i])];   /* implementation of Pearson's function              */
     i++;
} while (key[i] != '\0');
return (int) h%m;        /*the addition of the MOD number of buckets in the partition */
```

Algorithm 3.3: DYNB function

```
h = 0;                                          /* initialization                    */
i = 0; k = 0;
for (i = 0; i < 256; i ++) {
     if (m > ptable[i] {                        /* dynamic change in the size        */
        t[k] = ptable[i];                       /* of Pearson's table                */
        k++;
     }
}
do {
     h = t[(int) (h ^ key[i]) % m];             /* implementation of Pearson's function */
     i++;
}
while (key[i] != '\0');
return (int) h;
```

4. IMPLEMENTATION NOTES

We investigated the performance of the three described hash functions under the constraint of single user application. Another simplified assumption is that the file is a collection of arbitrary fixed-size records. All the suggested modifications of the B^+ algorithm were implemented in the C programming language. The modified B^+ algorithm is designed in such a way as to occupy up to 7 KB of operating memory and support up to 1000 partitions, but in our tests we used only up to 256 partitions. Also, we limited the number of data buckets per partition up to 256 because we used hash functions that generate 256 different values which is still enough to work with very large datafiles.

We can speak about O(1) data disk access only when each partition is placed in contiguous disk space. It is much better to work with only one file descriptor, which is the case when all partitions are part of one file with contiguous disk space. For the sake of simplicity, in our implementation of the modified B^+ algorithm, we used files for partitions. We used two groups of test data to analyze the performance of all three des-

cribed hash functions in the basic, incremental mode. The first group of tests worked with very large database files with simulated keys. The key length was 10B, because in [20] it has been shown that the average key length in current databases is 9.5 bytes. The second group of tests used an English-Serbian dictionary as the input file and the key length was 16B.

We compared and contrasted the performance of the algorithms with respect to the obtained load factor and the maximum number of buckets in a partition. The simulation results are presented in Section 5 and Section 6.

5. SIMULATION RESULTS

Performance evaluation of the modified B^+ algorithm was performed using a SCO UNIX System V 3.2.v.4.2. operating system on a PC 486 DX2/66 computer with 8 MB of main memory. CPU times were neglected. The standard random number generator *random* ( ) was used to generate the keys. The initial value for *random* ( ) was obtained from the system time. Simulation parameters are given in Table 2.

Table 2. Simulation parameter settings

Number of records (N)	100, 200, 500, 1 000, 2 000, 5 000, 10 000, 100 000, 1 000 000
Record size	10 B
Key length	10 B
Key distribution	uniform
Bucket size	1 KB, 2 KB, 4 KB, 16 KB

Efficient use of storage space is represented by load factor α. It is the ratio of the number of hashed records and the available record places (slots) in the allocated file. The load factor represents one of the most important hash characteristics. Fig. 4 shows the load factor α as a function of the number of records N for various bucket sizes in the incremental mode. The hash function is the MFOLD. The load factor oscillates for a small number of records N. The oscillations are due to fragmentation. For a large number of records ($N > 1000$), the load factor becomes stable and close to ln 2. For $N > 10{,}000$ the load factor becomes larger than ln 2 due to rehashing.

Similar simulation results were observed when using PEARS_MOD as the hash function. As can be seen from Fig. 5, both MFOLD and PEARS_MOD hash functions offered better load factors than DYNB.

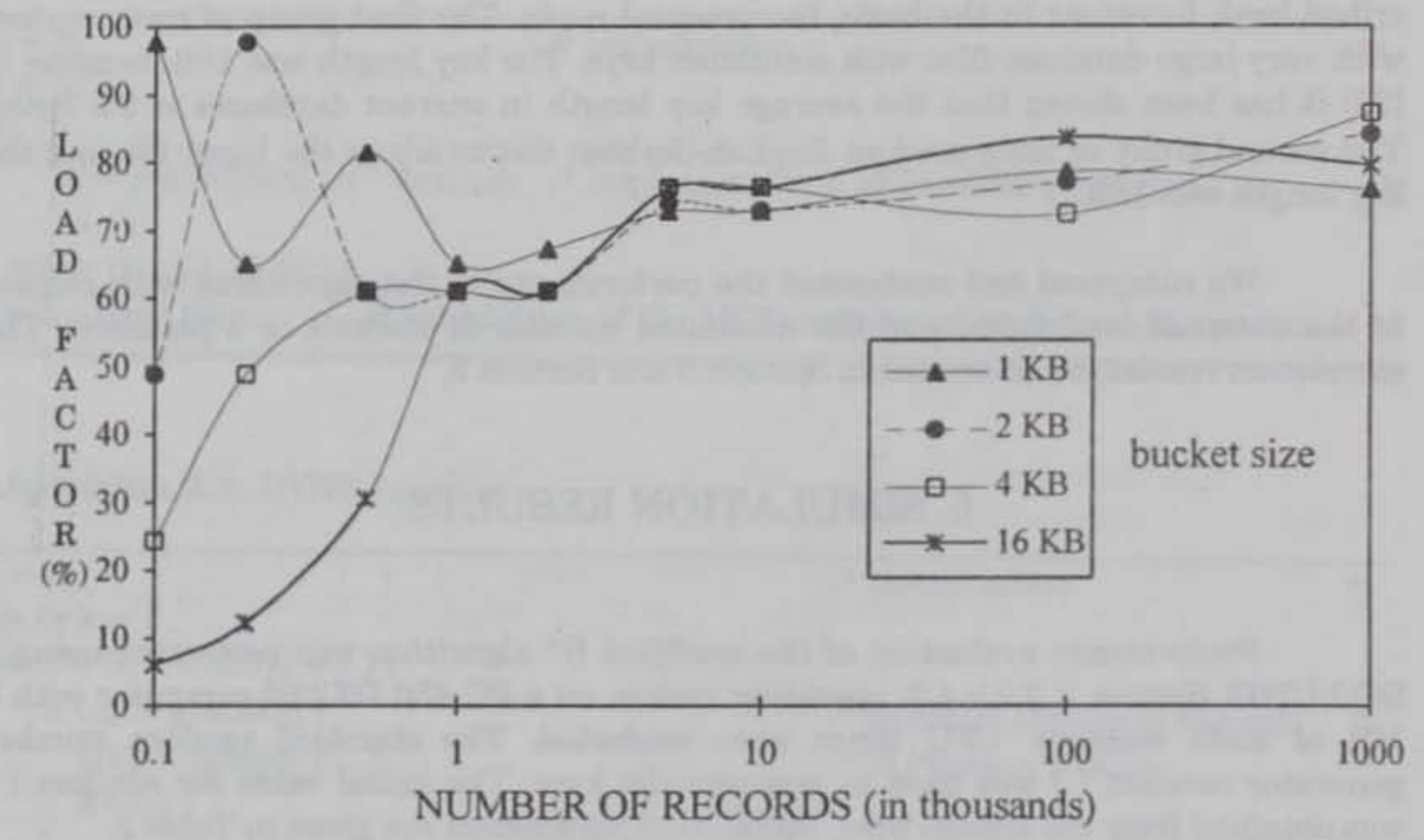

Figure 4. Load factor as a function of the number of records for various bucket sizes; the hash function is MFOLD

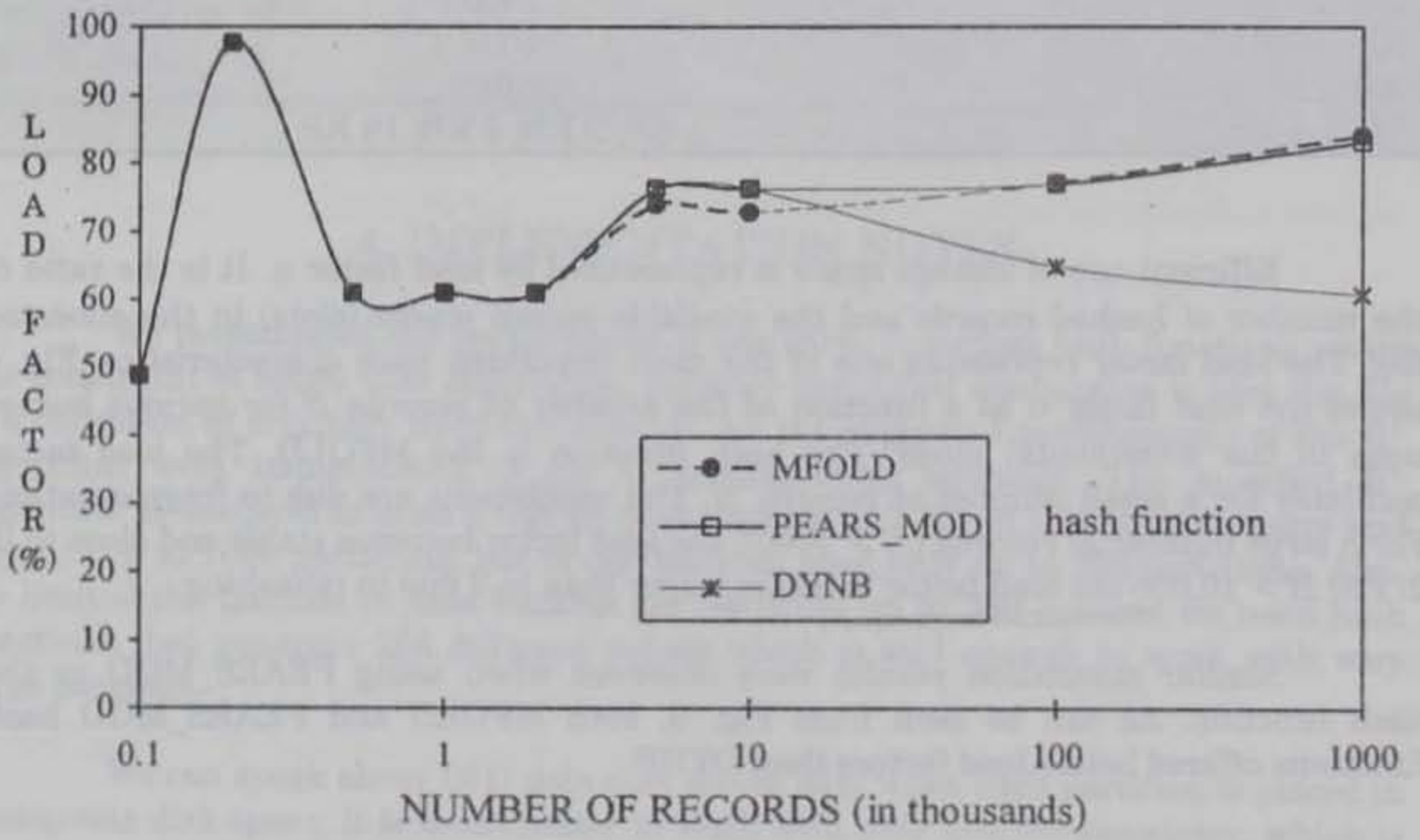

Figure 5. Load factor as a function of the number of records for various hash functions; bucket size $C = 2$KB

Fig. 6 represents the performance of the MFOLD algorithm as a function of the number of records. When the number of records is smaller than A, the modified B^+ algorithm works almost identically to the standard B^+ algorithm. When the number of records approaches B, the maximum assiged number of partitions is reached. We have to use the hash function again to perform rehashing. At that point the characteristics of the standard B^+ tree are still dominant. When the number of records becomes larger than C, the load factor becomes larger than 75%, and rehashing takes over.

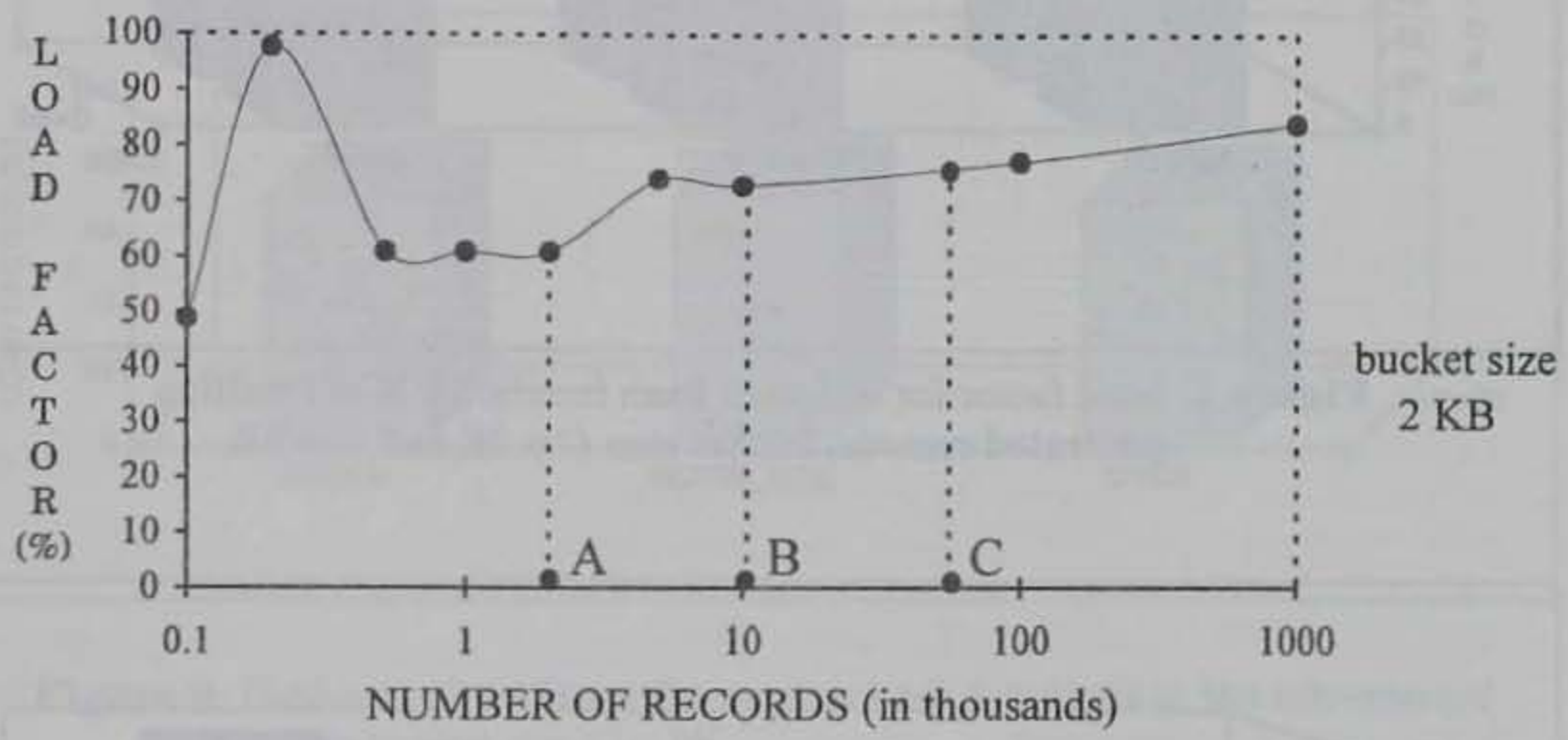

Figure 6. Load factor as a function of the number of records for bucket size C=2 KB; the hash function is MFOLD

Bearing in mind that the modified B^+ algorithm is intended for very large databases, it is interesting to compare the effect of using different hash functions for large files (Fig. 7). It should be noted that MFOLD and PEARS_MOD functions have almost the same impact on α, while the DYNB function offers a lower load factor. However, as the bucket size increases the load factor increases for the DYNB hash function. For example, for C = 8K the load factor is nearly the same for all three hash functions. Similarly, as the bucket size increases, the maximum number of buckets in one partition for the three given hash functions becomes nearly the same (Fig. 8).

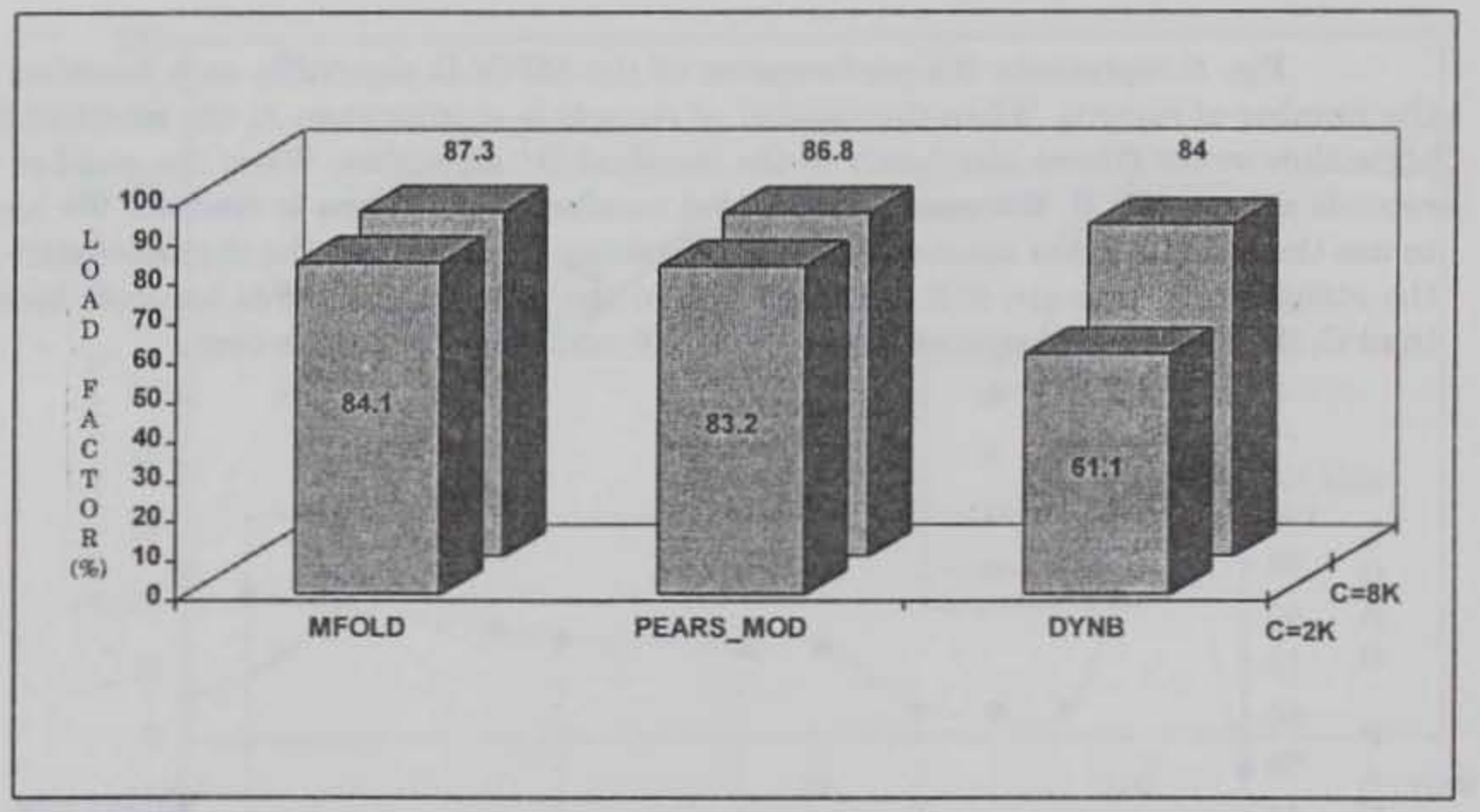

Figure 7. Load factor for different hash functions; N = 1 million generated records, bucket size $C = 2K$ and $C = 8K$.

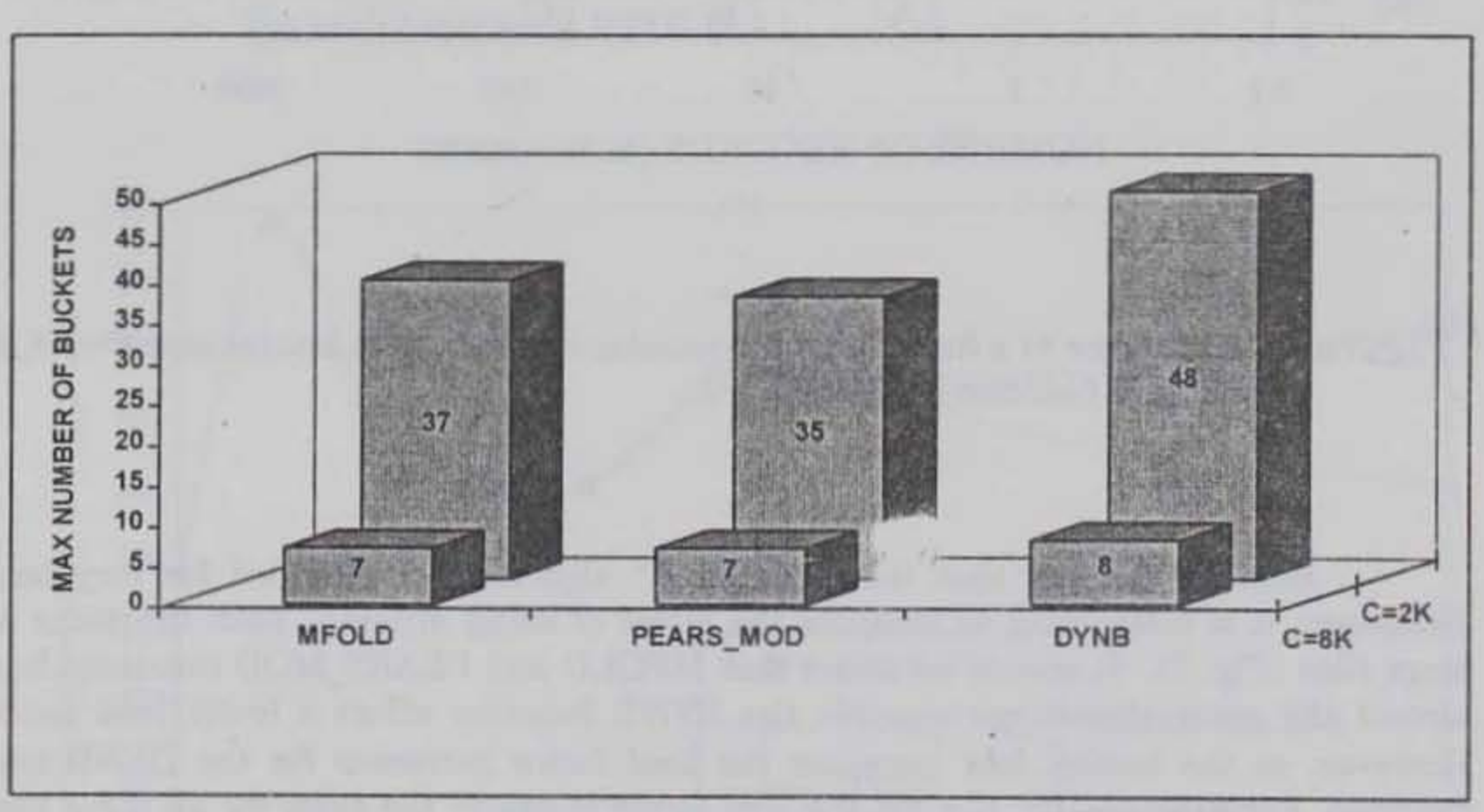

Figure 8. Maximum number of buckets in one partition for the three given hash functions for a generated file of 1 million records; bucket size $C = 8K$ and $C = 2K$

6. REAL DATABASE PERFORMANCE

We tested the performance of the modified B^+ algorithm implemented by the three given hash functions in the incremental mode. The input file was an English-Serbian dictionary with 35,638 records. The record size was 16 bytes.

The dictionary load factor for all three hash functions in the incremental mode is shown in Fig. 9. As the figure shows, the 68.9% result for the PEARS_MOD function is better than the 67.9% result for the MFOLD function and the 66.6% result for the DYNB function.

Fig. 10 shows the maximum required number of buckets in one partition for simulated hash functions. It can be seen that our modification of Pearson's function, PEARS_MOD, requires the smallest number of buckets, while the MFOLD hash function requires a slightly larger number of buckets.

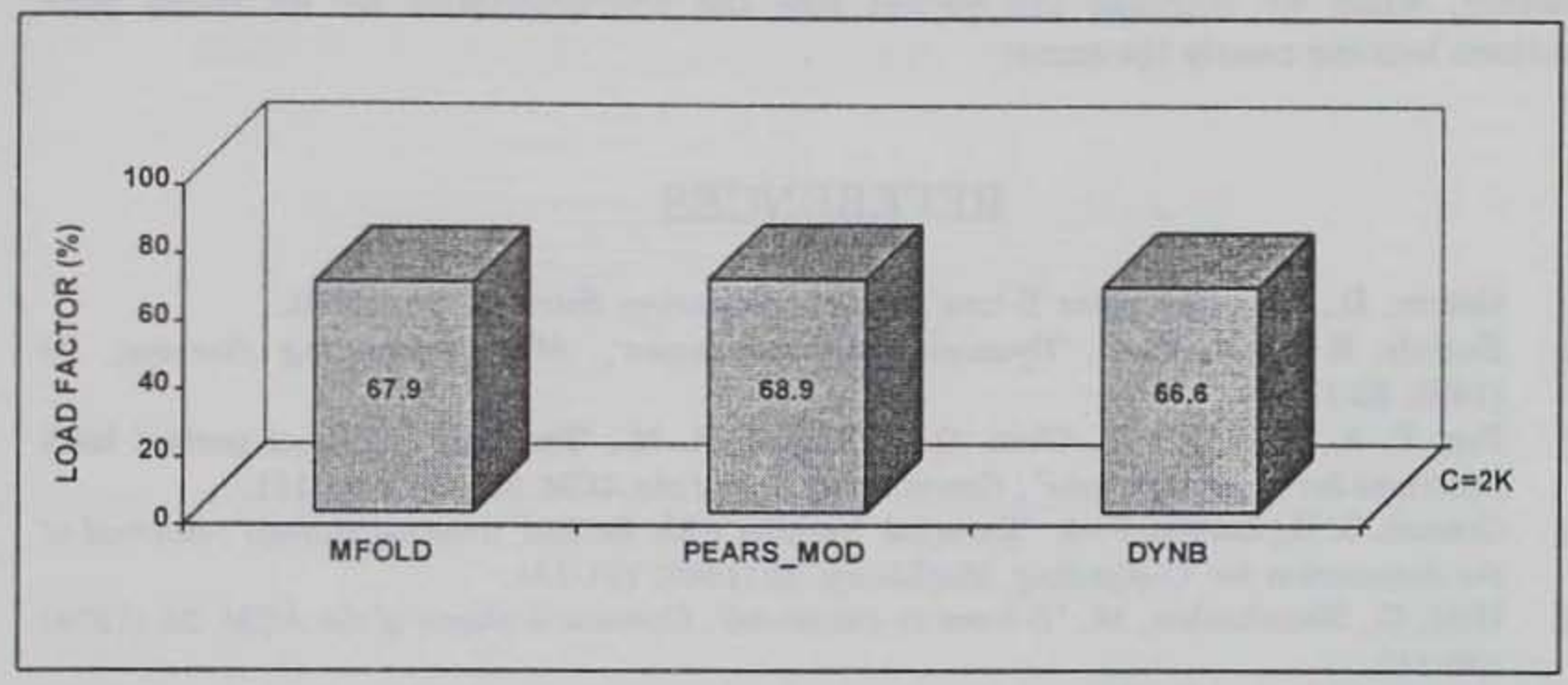

Figure 9. Dictionary load factor for various hash functions in the incremental mode; bucket size $C = 2$K

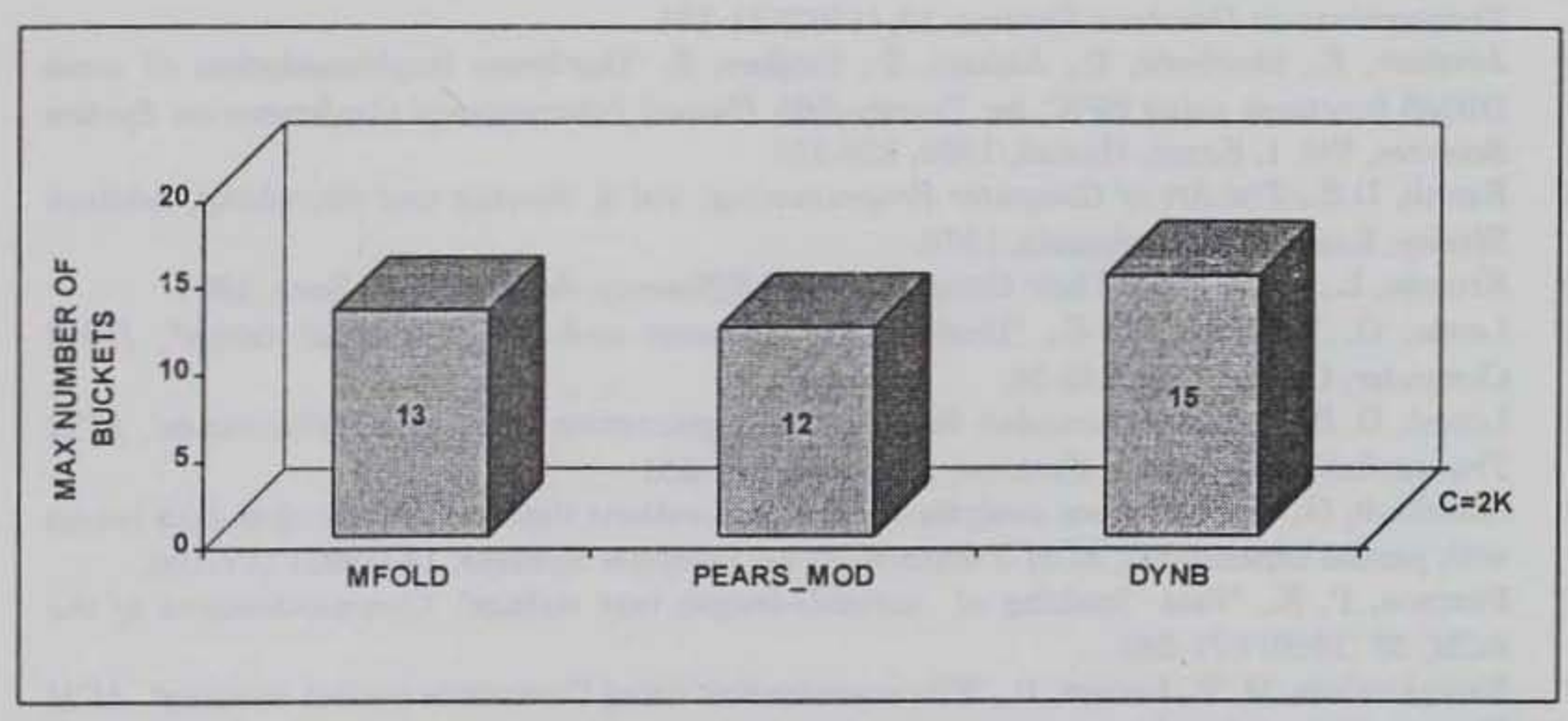

Figure 10. Maximum number of buckets in one partition for various hash functions; bucket size $C = 2$K

When we increase the bucket size, the characteristics for all three hash functions become nearly the same.

7. CONCLUSION

A class of algorithms based on the synergism of a conventional B^+ algorithm and a hash function could be the solution for efficient support of standard operations on very large dynamic files, and provide data retrieval with a guaranteed single disk access. We used the modified B^+ algorithm as a representative of the mentioned class of algorithms. Also, in this paper we investigated three fast and simple hash functions that can be used in the modified B^+ algorithm. Our simulation results showed that for small bucket sizes, MFOLD and PEARS_MOD offer superior performance over DYNB. However, when we increase the bucket size the characteristics for all three hash functions become nearly the same.

REFERENCES

[1] Comer, D., "The ubiquitous B-tree", *ACM Computing Surveys*, 11 (1979).

[2] Enbody, R. J., Du, H. C., "Dynamic hashing schemes", *ACM Computing Surveys*, 20 (1988) 85-113.

[3] Fox, E. A., Heath, L. S., Chen, Q. F., Daoud, A. M., "Practical minimal perfect hash functions for large databases" , *Communications of the ACM*, 35 (1992) 95-121.

[4] Gonnet, G. H., Larson, P.-A. "External hashing with limited internal storage", *Journal of the Association for Computing Machinery*, 35 (1988) 161-184.

[5] Held, G., Stonebraker, M., "B-trees re-examined", *Communications of the ACM*, 21 (1978) 139-143.

[6] Inoue, U., Satoh. T., Hayami, H., Takeda, H., Nakamura, T., Fukuoka, H., "RINDA A relational database processor with hardware specialized for searching and sorting", *IEEE Micro*, December 1991, 61-70.

[7] Jannink, J., "Implementing deletion in B^+-trees", *SIGMOD RECORD*, 24, (1995) 33-38.

[8] Johnson, T., Shasha, D., "The performance of concurrent B-tree algorithms", *ACM Transactions on Database Systems*, 18, (1993) 51-101.

[9] Jovanov, E., Starčević, D., Aleksić, T., Stojkov, Z. "Hardware implementation of some DBMS functions using SPR", in: *Twenty-fifth Hawaii International Conference on System Sciences*, Vol. 1, Kanai, Hawaii, 1992, 328-337.

[10] Knuth, D.E., *The Art of Computer Programming, Vol 3, Sorting and Searching*, Addison Wesley, Reading Massachusets, 1973.

[11] Kronsjo, L., *Algorithms: Their Complexity and Efficiency*, John Wiley & Sons, 1987.

[12] Lewis, G. T., Cook, R. C., "Hashing for dynamic and static internal tables", *IEEE Computer*, October 1988, 45-56.

[13] Lomet, D. B., "A simple bounded disorder file organization with good performance", *ACM Transactions on Database Systems*, 13, (1988) 525-551.

[14] Matsliach, G., "Performance analysis of file organizations that use multibucket data leaves with partial expansions", *ACM Transactions on Database Systems*, 18 (1993) 157-180.

[15] Pearson, P. K., "Fast hashing of variable-length text strings", *Communications of the ACM*, 33 (1990) 677-680.

[16] Ramakrishna, M. V., Larson, P., "File organization using Composite perfect hashing", *ACM Transactions on Database Systems*, 14 (1989) 231-263.

[17] Salzberg, B., *File Structures: An Aanalytical Approach*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[18] Sartori, C., Scalas, M. R., "Partial indexing for nonuniform data distributions in relational DBMSs", *IEEE Transactions on Knowledge and Data Engineering*, 6 (1994) 420-429.

[19] Starčević, D., Jovanov, E., Large File Operations Support Using Order Preserving Perfect Hashing Functions, *Yugoslav Journal of Operations Research*, 3 (1993) 171-188.

[20] Yu, P.S., Chen, M.S., Heiss, H.U., Lee, S., "On workload characterization of relational database environments", *IEEE Trans. on Software Engineering* 18 (1992) 347-355.