PERFORMANCE COMPARISON OF EXTENDIBLE HASHING AND LINEAR HASHING TECHNIQUES

Ashok Rathi, Huizhu Lu, G.E. Hedrick

Department of Computing and Information Sciences
Oklahoma State University
Stillwater, Oklahoma 74078
Phone: (405) 744-5668

ABSTRACT

Based on seven assumptions, the following comparison factors are used to compare the performance of linear hashing with extendible hashing: 1. storage utilization; 2. average unsuccessful search cost; 3. average successful search cost; 4. split cost; 5. insertion cost; 6. number of overflow buckets. The simulation is conducted with the bucket sizes of 10, 20, and 50 for both hashing techniques. In order to observe their average behavior, the simulation uses 50,000 keys which have been generated randomly.

According to our simulation results, extendible hashing has an advantage of 5% over linear hashing in terms of storage utilization. Successful search, unsuccessful search, and insertions are less costly in linear hashing. However, linear hashing requires a large overflow space to handle the overflow records. Simulation shows that approximately 10% of the sapce should be marked as overflow space in linear hashing.

Directory size is a serious bottleneck in extendible hashing. Based on the simulation results, the authors recommend linear hashing when main memory is at a premium.

I. INTRODUCTION

A number of file structures and access methods, e.g. B+ tree [Knu73]. inverted file [Knu73], heap [Mar77], grid file [Nie84] [Chu89], BANG file [Fre87] [Lia89], AVL data structure with persistent technique [Ver87], and hashing are widely used in current database design. Among those techniques, hashing is a well-known technique for organizing direct access files. The method is simple: Retrieval, insertion, and deletion of records is very fast. In traditional hashing, the size of the file must be estimated in advance, and storage space must be allocated for the entire file. To overcome these drawbacks, several dynamic hashing schemes were developed in late seventies and early eighties.

The dynamic hashing scheme [Lar78] and the dynamic hashing scheme with deferred splitting [Sch81] both keep an index in main memory. In these schemes, the random access cost is high. A spiral storage scheme [Mul85] seeks to provide a uniform performance regardless of the file size. This scheme involves a very complex address computation to determine the appropriate buckets. Also, the expansion process is both slow and complex.

To overcome the shortcomings of the spiral storage scheme, W. Litwin [Lit80] and Fagin et al. [Fag79] presented hashing schemes called linear hashing and extendible hashing respectively. Later, Ellis applied concurrent operations to extendible hashing in a distributed database environment [Ell82]. The address computation and expansion processes in both linear hashing and extendible hashing is easy and efficient [Lar82] [Lar85] [Bra86].

Both Litwin [Lit80] and Fagin et al. [Fag79] claimed their respective hashing techniques to be efficient. However, no comparison results of the two techniques were reported. Hence, the objective of this paper is to compare both linear hashing and extendible hashing.

Section II of this paper briefly reviews linear hashing and extendible hashing. Section III discusses the simulation setup for comparison and section IV presents the simulation results and conclusions (Mathematical derivations have been shown regarding search costs, insertion cost etc. They are omitted here due to space limitations).

II. LINEAR HASHING AND EXTENDIBLE HASHING

The linear hashing scheme, referred to as LINHASH hereafter, is a directory-less scheme which allows a smooth growth of the hash table [Ram82]. The following example is due to Larson [Lar88]. Consider a hash table

consisting of N buckets with addresses 0,1,..,N-1. LINHASH splits the buckets in predetermined order; i.e., the first bucket has address 0, then bucket 1, and so on, up to and including bucket N-1. In figure 1(a), the table size N is 3 and the next bucket to be split is bucket 0. Pointer p always indicates the bucket to be split next. Figure 1(b) shows the status after bucket 0 has been split. Notice that pointer p has moved to bucket 1. Next, bucket 1 is split into bucket 1 and bucket 4. The current expansion is considered complete when the last bucket of the table is split. After the split, pointer p is reset to bucket 0. In our example, the expansion will be complete when bucket 3 is split, as shown in figure 1(d).

The extendible hashing technique, referred to as EXHASH hereafter, was developed by Fagin et al. [Fag79]. This scheme uses the leading (or trailing) bits, denote by d, of the key to index into the directory. Global depth, d, and local depth, d', imply the depth of the directory and the depth of a bucket, respectively.

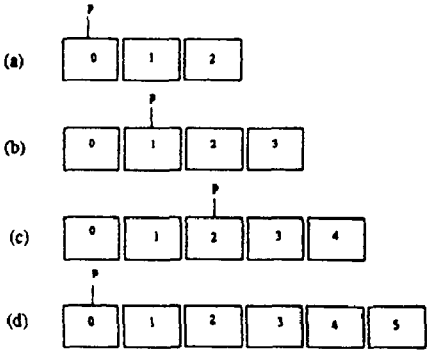


Figure 1: Expansion Process in Linear Hashing

There are 2**d directory entries. Often more than one directory entry points to the same bucket. Figure 2 expands this discussion. Upon expansion of the table, the local depth of the 2 buckets involved is increased by 1. If d' of any bucket is greater than d, then the directory size is doubled (shown in figure 3), and the global depth is increased by 1.

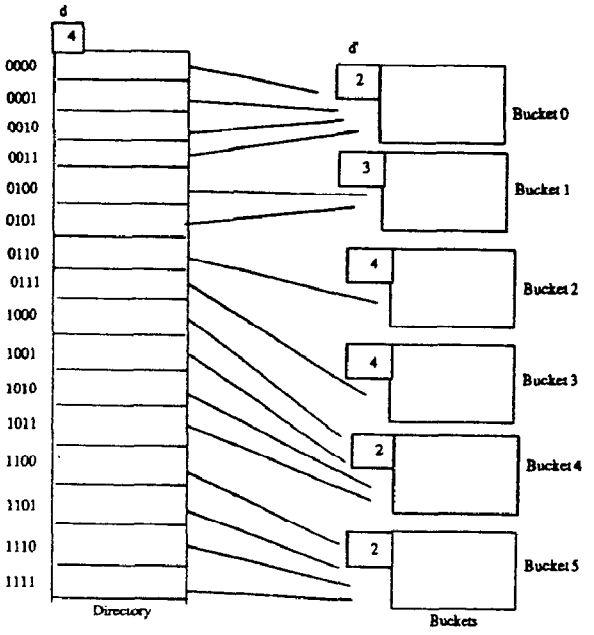
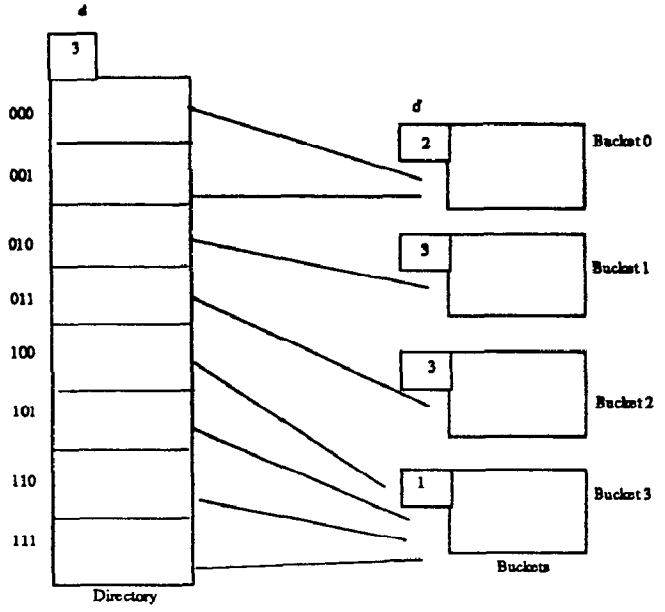
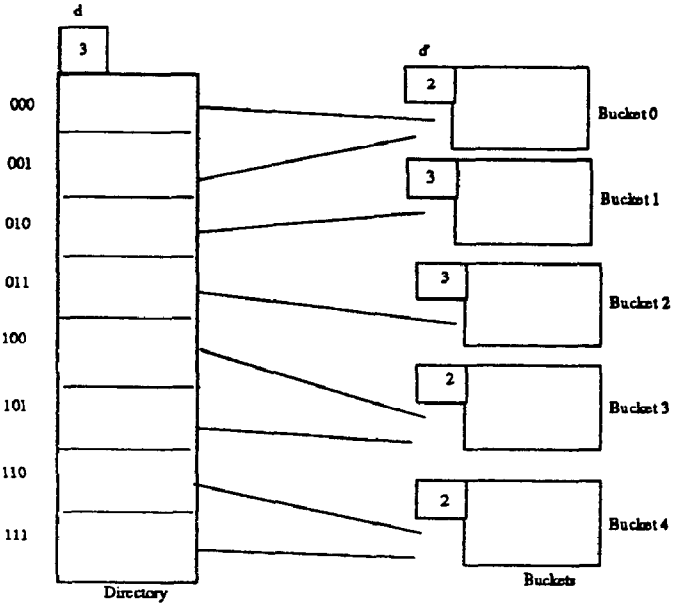


Figure 3: Hash Table Doubled After Splitting Bucket 2 of Figure 2(b)



(a) : Extendible Hash Table Before Split



(b) : After Splitting Bucket 3 of Figure 2(a)

Figure 2: An Example of Extendible Hash Table

III. SIMULATION PREPARATION

The performance comparison factors for simulation are based on the following assumptions.

1. Assumptions

(1) The keys are distributed uniformly, so each key has equal access probability.
(2) Records are of fixed length.
(3) The bucket capacity is fixed in terms of the number of records that it can hold.
(4) Expansion takes place as soon as a bucket overflows.
(5) Enough main memory is available to handle the expansion.
(6) EXHASH: (a) The most significant bits are extracted from the key to find the directory entry. (b) The overflow bucket is split at most once. In other words, a second split is not attempted even though the first split may fail to release the overflow bucket. (c) Main memory can hold a maximum of 1024 directory entries. The rest of the directory must reside on the secondary storage.
(7) LINHASH: A simple division method with modulo arithmetic is used to find the relevent bucket.

According to assumption (1), we use a random function that broadly satisfies the properties of a minimal random function. Given a minimal random function "f(z) = az mod m", the value of "a" should pass the three tests as defined in [Par88] such that f(z) should (i) be a full period generating function; (ii) be random for all the sequences generated; and (iii) be implemented efficiently with 32-bit arithmetic. Further, the hash functions used in the simulation also satisfy the basic properties listed by Carter and Knuth in [Car79] [Knu73].

2. Comparison Factors

Following notations are used to define the comparison factors:

N : The number of records in the current hash table

B : The number of buckets in the current hash table

b : Bucket capacity

bs : The number of buckets accessed for successful search + 1 if the directory entry is not in main memory. (only in EXHASH).

bu : Number of buckets accessed for unsuccessful search + 1 if the directory entry is not found in main memory (only in EXHASH).

s : Number of successful searches

u : Number of unsuccessful searches

* : Arithmetic multiplication symbol

/ : Arithmetic division symbol

Following factors have been considered to analyze the relative performance of LINHASH and EXHASH:

(1) Storage utilization : N /(B*b)

(2) Average unsuccessful search cost : bu / u

(3) Average successful search cost: bs / s

(4) Split cost (expansion cost): In LINHASH, a split bucket is usually different from the bucket where insertion took place. Hence additional accesses are needed to read the split bucket chain.

LINHASH: 1 access to read the primary bucket
+ k accesses to read k overflow buckets
+ 1 access to write old bucket
+ extra accesses to write the overflow buckets attached to old and new buckets

EXHASH: 1 access to write old bucket
+ 1 access to write new bucket
+ extra accesses to write the overflow buckets attached to old and new buckets
+ accesses needed to update the directory pointers if the directory resides on the secondary storage

(5) Insertion cost: Unsuccessful search cost + Split cost

(6) Number of overflow buckets

The above factors have been simulated with the bucket sizes of 10, 20, and 50 for both EXHASH and LINHASH. In order to observe their average behavior, the simulation uses 50,000 keys which have been generated randomly.

IV. RESULTS & CONCLUSION

1. Simulation Results

For all bucket sizes, EXHASH produces consistently better storage utilization than LINHASH. LINHASH gives cyclic storage utilization since the buckets are split linearly regardless of their load. In both EXHASH and LINHASH, as the bucket size rises, the storage utilization becomes more fluctuating (see figures 4,5,6). EXHASH has an advantage of approximately 5% over LINHASH in storage utilization. Such a performance is wholly attributable to the way the buckets are split under the two schemes. The corollary is that LINHASH requires more buckets to hold the same number of records than EXHASH does.

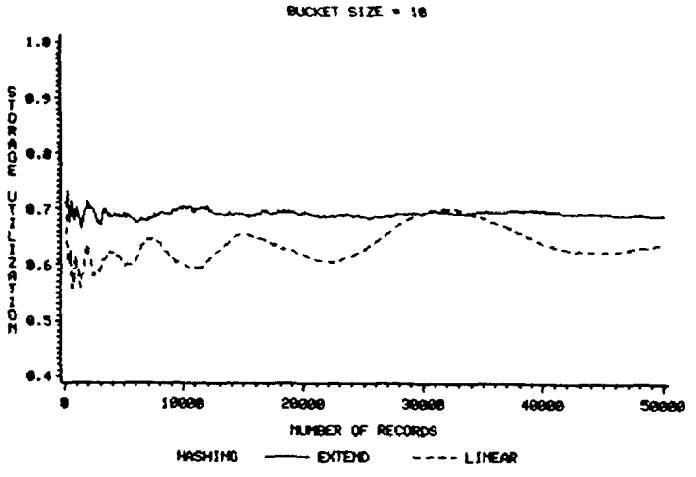


FIGURE 4 . STORAGE UTILIZATION VS. NUMBER OF RECORDS

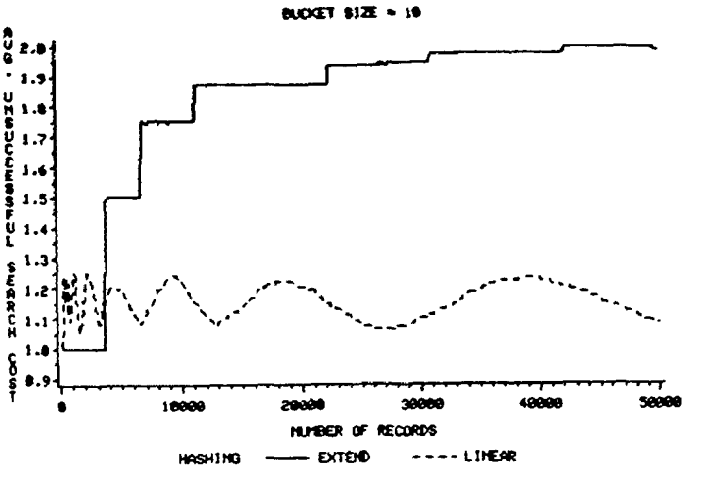


FIGURE 7 . UNSUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

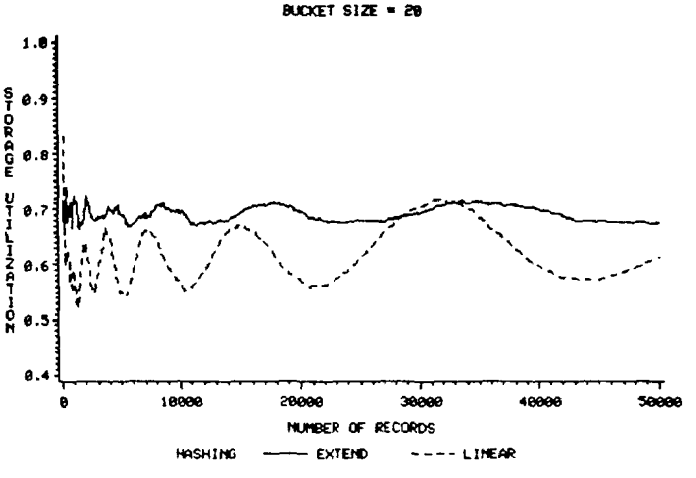


FIGURE 5 . STORAGE UTILIZATION VS. NUMBER OF RECORDS

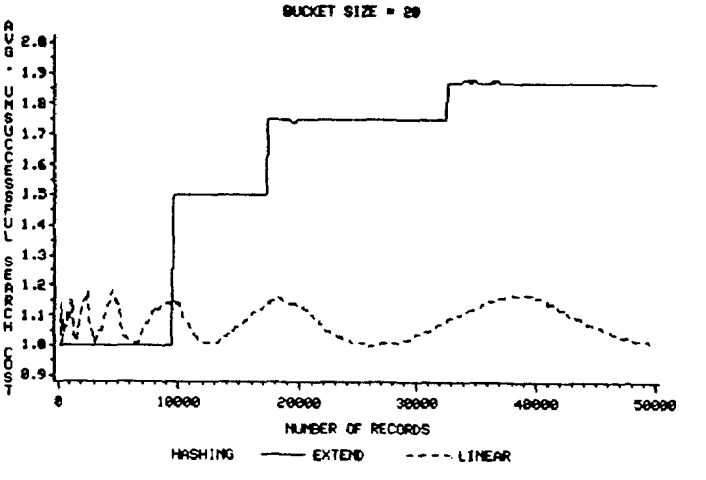


FIGURE 8 . UNSUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

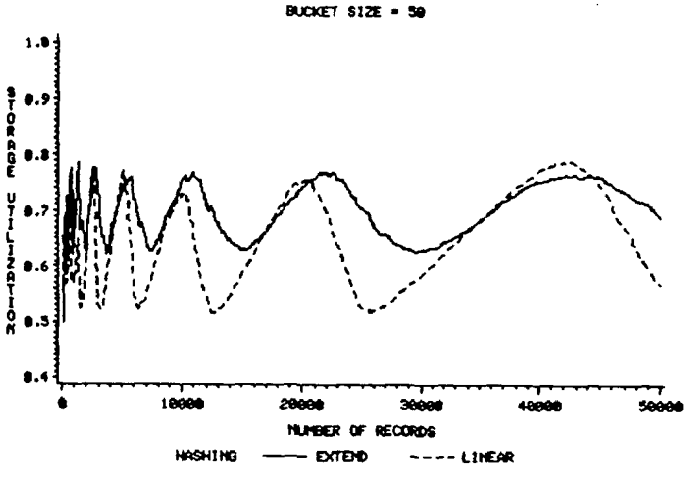


FIGURE 6 . STORAGE UTILIZATION VS. NUMBER OF RECORDS

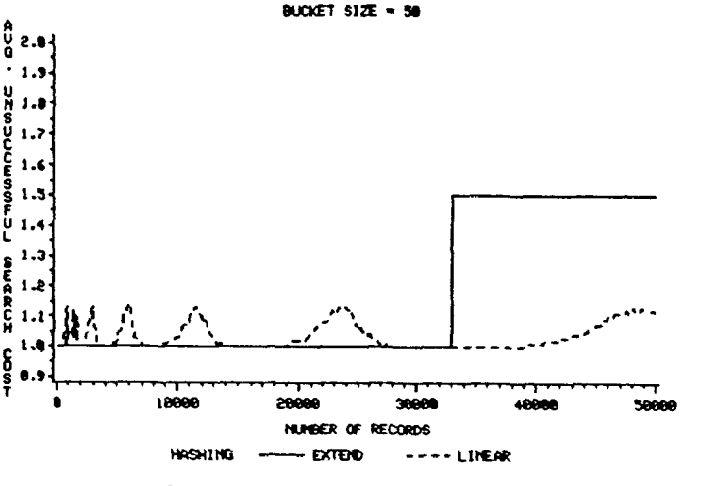


FIGURE 9 . UNSUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

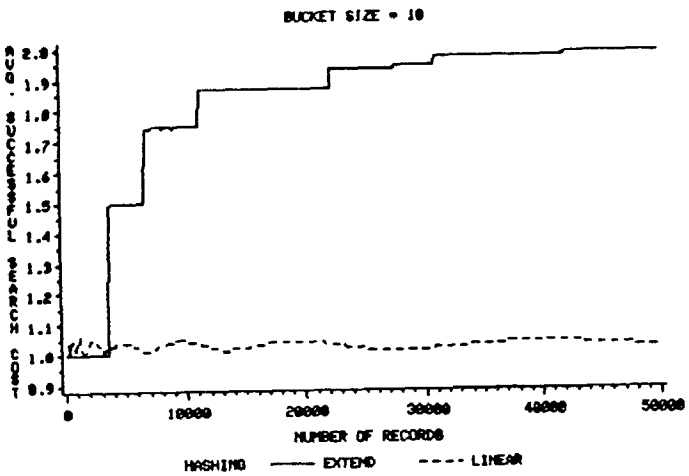


FIGURE 10 . SUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

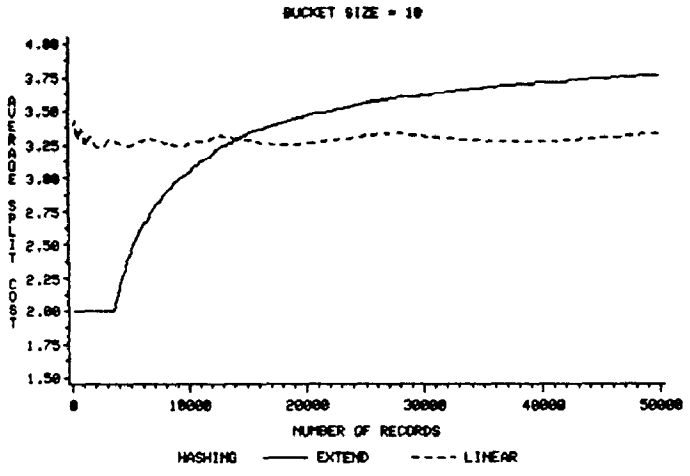


FIGURE 13 . SPLIT COST VS. NUMBER OF RECORDS

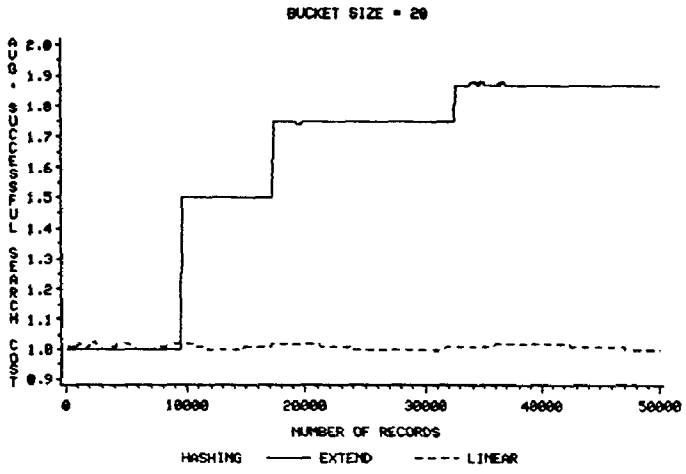


FIGURE 11. SUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

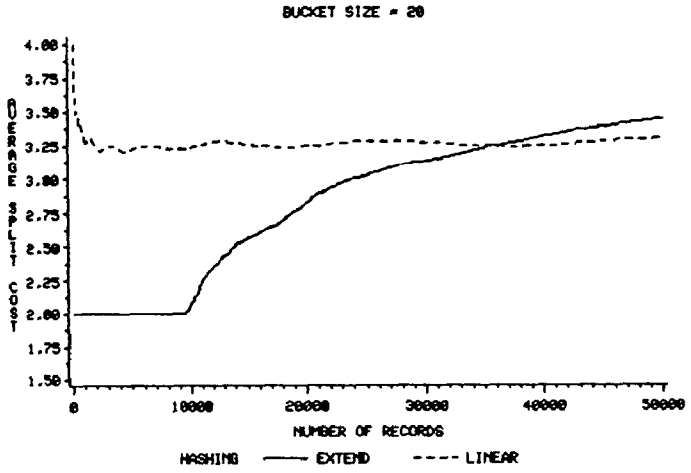


FIGURE 14. SPLIT COST VS. NUMBER OF RECORDS

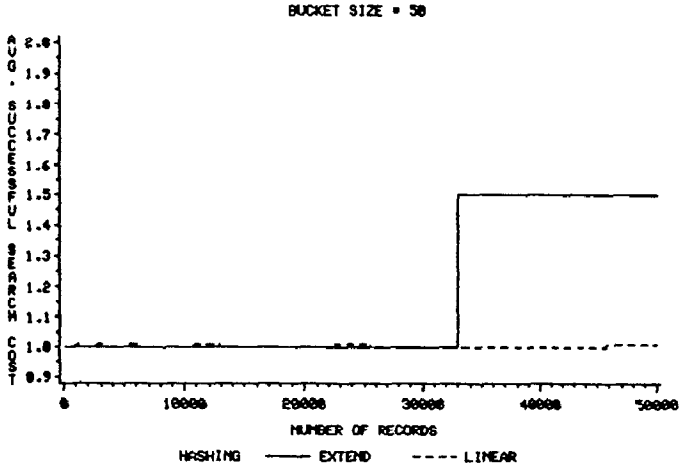


FIGURE 12 . SUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

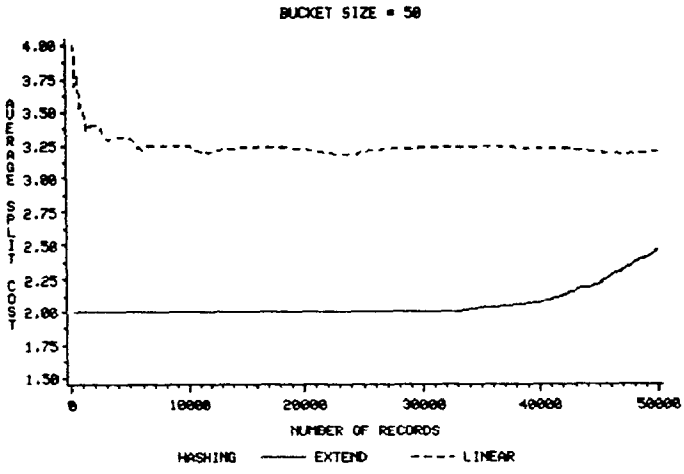


FIGURE 15 . SPLIT COST VS. NUMBER OF RECORDS

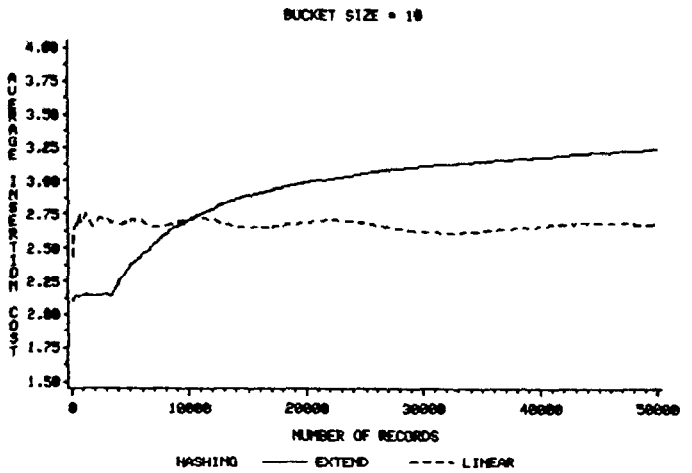


FIGURE 16. INSERTION COST VS. NUMBER OF RECORDS

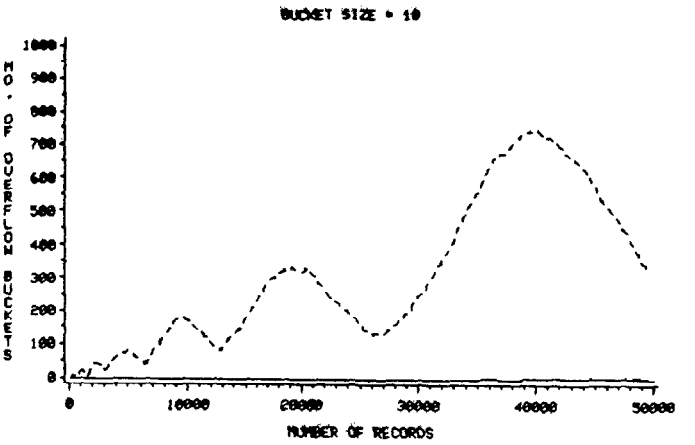


FIGURE 19. OVERFLOW BUCKETS VS. NUMBER OF RECORDS

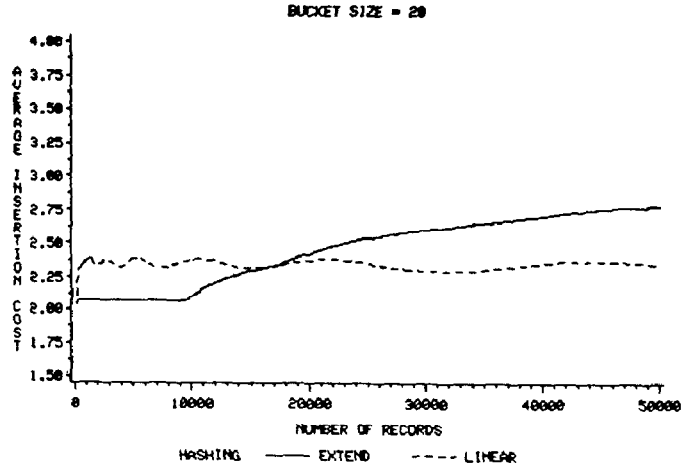


FIGURE 17. INSERTION COST VS. NUMBER OF RECORDS

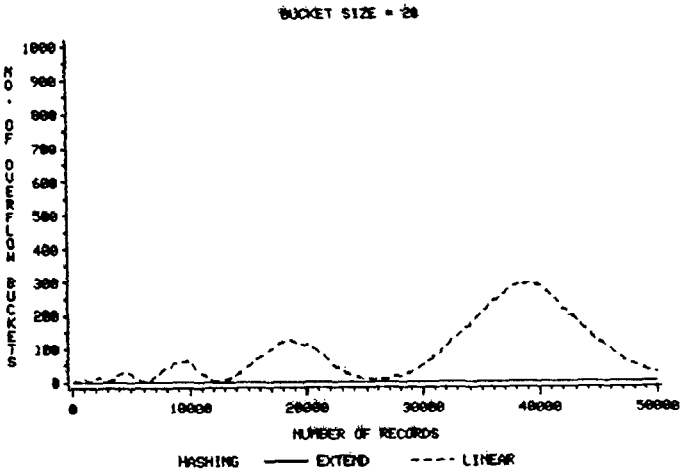


FIGURE 20. OVERFLOW BUCKETS VS. NUMBER OF RECORDS

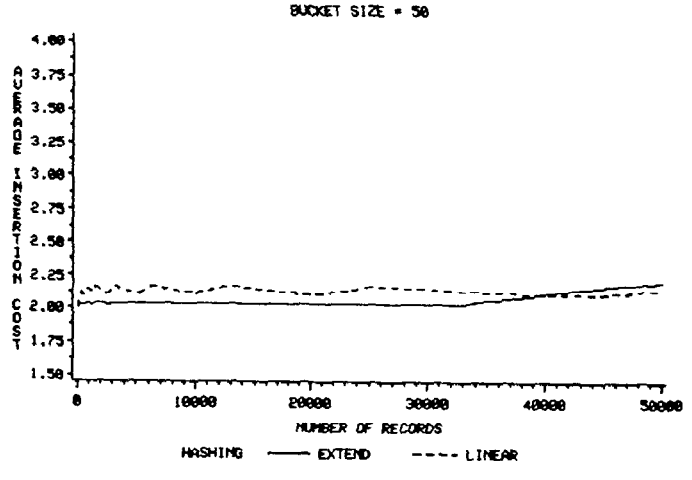


FIGURE 18. INSERTION COST VS. NUMBER OF RECORDS

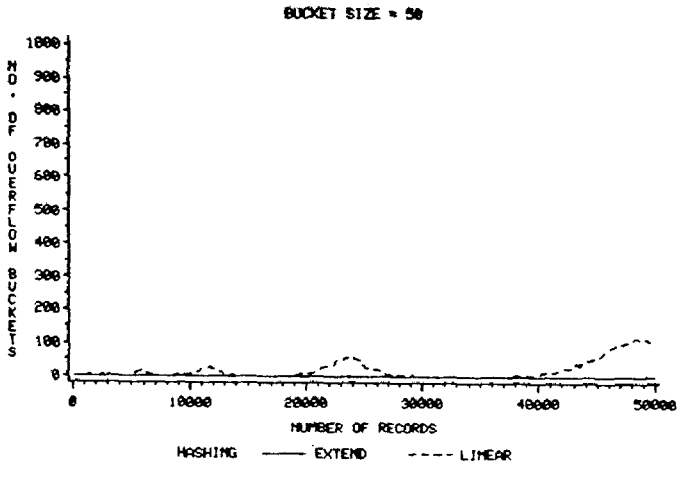


FIGURE 21. OVERFLOW BUCKETS VS. NUMBER OF RECORDS

LINHASH performs better for all the bucket sizes with an unsuccessful search becomes less costly as the bucket size rises. On an average, an unsuccessful search cost stays close to 1 for all the bucket sizes in LINHASH (see figures 7,8,9). Similar observations hold true for the cost of a successful search (see figures 10,11,12). The successful search and the unsuccessful search are equally costly in EXHASH. This is due to the fact that the overflow buckets are almost non-existent in EXHASH. Overflow buckets are mandatory in LINHASH. In EXHASH, the search cost can be kept to 1 regardless of the bucket size when the entire directory can be kept in the main memory.

The splitting of a bucket is costlier in LINHASH. This is due to the fact that an extra read access is needed to read the bucket to be split (see figures 13,14,15). The insertion cost is slightly higher in EXHASH for the bucket sizes 10 and 20. However, for the bucket size 50, this cost is slightly less in EXHASH (see figures 16,17,18).

As expected, LINHASH performed poorly with respect to the number of overflow buckets. The number of overflow buckets decreases as the bucket size increases. The simulation shows that a maximum of 10% of the total space should be marked as an overflow area in LINHASH. Overflow buckets are almost non-existent in EXHASH (see figures 19,20,21).

2. Conclusion

Based on simulation results, the linear hashing technique is recommended when main storage is at a premium since it requires no directory. This scheme is particularly useful in a small computer environment. However, this scheme is not devoid of its pitfalls. Since there is no control over the length of an overflow chain, the search cost may become high. However, the simulation has shown that the maximum search cost is 2 for all the bucket sizes in linear hashing. Extendible hashing could be useful if sufficient main memory is available to hold the directory. Doubling and halving the directory size is expensive. In both the schemes, the bucket size does not affect the performance significantly. However, a bucket size of 20 seems to be a good choice since it gives fairly reasonable storage utilization and search times.

REFERENCES

[Bra86] Bradley, J. "Use of Mean Distance between Overflow Records of Compute Average Search Lengths in Hash Files with Open Addressing." Computer J., 29, 2(1986), pp. 167-170.

[Car79] Carter, J.L. and Wegman M. "Universal Class of Hash Functions," J. of Comp. & Sys. Sci., 18, 1(1979), pp. 143-154.

[Chu89] Chun, S.H., Hedrick, G.E., Lu, H., Fisher, D.D., "A Partitioning Method for Grid File Directories," will appear in Proc. of the IEEE Computer Society's 13th Annual International Computer Software and Applications Conference, Sept. 18-22, 1989 Orlando applications Conference, Sept. 18-22, 1989 Orlando

[Ell82] Ellis, C.S. "Extendible Hashing for Concurren Operations and Distributed Data." ACM SIGMOD, 1982

[Fag79] Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H.R. "Extendible Hashing - A Fast Access Method for Dynamic Files." ACM Transactions on Database Systems, 14, 3(Sept.1979), pp. 315-344.

[Fre87] Freeston, M. "The BANG file." Proc. of ACM SIGMOD Conf., 16, 3(Dec. 1987), 260-269.

[Knu73] Knuth, D. The Art of Computer Programming, vol. III: Sorting and Searching. Reading, MA: Addison-Wesley, 1973.

[Lar78] Larson, P. "Dynamic Hashing." BIT, 18(1987), pp. 184-201.

[Lar82] Larson, P. "Performance Analysis of Linear Hashing with Partial Expansions." ACM Transactions on Database Systems, 7, 4(1982), pp. 566-587.

[Lar85] Larson, P. "Performance Analysis of a Single-file Version of Linear Hashing." Computer J.,28, 3(1985), pp. 319-326.

[Lar88] Larson, P. "Dynamic Hash Tables." Communications of the ACM, 31, 4(April 1988), pp. 446-457.

[Lia89] Lian, T., Fisher, D., Lu, H., "Implementation and Evaluation of Grid and Bang (Balanced and Nested Grid) File Structures," IEEE Proc. of Workshop on Applied Computing 1989, pp. 80-85.

[Lit80] Kitwin, W. "Linear Hashing: A New Tool for File and Table Addressing." Proc. of the 6th Conference on Very Large Databases, 1980, pp. 212-223.

[Mar77] Martin, J. Computer Data-Base Organization (2nd edition). Prentice-Hall, 1977.

[Mul85] Mullin, J.R. "Spiral Storage: An Efficient Dynamic Hashing with Constant Performance." Computer J., 28, 3(1985), pp. 330-334.

[Nie84] Nievergelt, J., Hinterberger, H., & Sevcik, K.C., The Grid File: An Adaptable, Symmetric Multikey File Structure", ACM Transactions on Database Systems, Vol. 9, No. 1, 1984, pp 38-71.

[Par88] Park, S.K. and Miller, K.W. "Random Number Generators: Good Ones Are Hard to Find."

Communications of the ACM, 31, 10 (October 1988), pp. 1192-1201.

[Ram82] Rammohanrao, K. and Lloyd, J.K. "Dynamic Hashing Schemes." Computer J., 25, 4(1982), pp. 478-485.

[Sch81] Scholl, M. "New File Organizations Based on Dynamic Hashing." ACM Transactions on Database Systems, 6, 1(Mar. 1981), pp. 194-211.

[Ver87] Verma, V., & Lu, H., "A New Approach to Version Management for Databases," Proceedings of National Computer Conference (AFIP Conference) vol. 56, 1987, pp. 645-651.