PERFORMANCE COMPARISON OF LINEAR HASHING AND EXTENDIBLE HASHING

By

ASHOK K. RATHI

Bachelor of Commerce
University of Rajasthan
Jaipur, India
1982

Master of Business Administration
Oklahoma State University
Stillwater, Oklahoma
1987

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
Master of Science
May, 1989

PERFORMANCE COMPARISON OF LINEAR HASHING AND EXTENDIBLE HASHING

Thesis approved:

Huizhu Lu
Thesis Advisor

Scott Tilley

Mansur Samadzadeh H.

Norman N. Durham
Dean of the Graduate College

ACKNOWLEDGEMENTS

This thesis simulates extendible hashing and linear hashing schemes to evaluate their relative performance in terms of storage utilization, search cost, insertion cost, overflow requirements etc. The study includes a design and implementation of these schemes under XELOS (a version of UNIX) operating system on Perkin-Elmer 3230. The empirical results obtained from simulation are discussed.

I thank Drs. H. Lu, M. Samadzadeh, and S. Turner for serving on my graduate committee. Additionally, I would like to express my sincere appreciation and gratitude to my principal advisor Dr. H. Lu for her guidance, motivation and assistance. I am also thankful to Dr. M. Samadzadeh for his useful and constructive suggestions.

Finally, I thank my family members for their love, understanding and sacrifices throughout my studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Motivation

Hashing is a well-known scheme for organizing direct access files. In hashing, retrieval, insertion and deletion of records are very fast except when there is a long overflow chain of records [Enb88]. There are 2 different storage allocation schemes: (1) Static Storage Allocation: In this scheme, the size of the file must be estimated in advance and physical storage space must be allocated for the whole file. In other words, the amount of allocated storage space is fixed and cannot be altered without reorganizing the whole file. This scheme performs well only if a file or a table is relatively static in its size. (2) Dynamic Storage Allocation: This scheme allocates the storage space dynamically, i.e., allocate only as needed. Hence there is no need to estimate the storage space in advance.

In most situations, the storage requirements are difficult to estimate in advance. Also, if there exist a few records currently and rapid growth is expected in future, huge amount of extra space will have to be allocated in the static scheme. Dynamic storage allocation scheme saves the space and also overcomes the difficulty of estimating the

file size in advance.

Literature Review

Hashing schemes which allow dynamic storage allocation without total reorganization of the hash table are called dynamic hashing schemes. Over the past few years, a number of dynamic hashing schemes have been proposed. With limited reorganization, the files can be expanded and contracted according to the number of records. The dynamic hashing schemes include dynamic hashing [Lar78], dynamic hashing with deferred splitting [Sch81][Cha85], spiral storage scheme [Mul81], extendible hashing [Fag79] and linear hashing [Lit80][Lar80,82,83,85,88][Mul81][Ram84].

Chang [Cha85] compared "dynamic hashing" scheme with "dynamic hashing with deferred splitting." Dynamic hashing with deferred splitting is found to have improved space utilization but shows poor average retrieval time per record. This is attributable to the existence of overflow buckets to defer the splitting. Both the schemes are discussed in detail in Chapter II.

Fagin et al. [Fag79] analyzed and compared extendible hashing scheme with B-tree. Extendible hashing scheme is algorithmically simple and guarantees no more than 2 secondary storage accesses to retrieve the data associated with a given key. Patel [Pat87] replicated the above study using B+ tree in place of B-tree and concluded that the average storage utilization for both the schemes is about 69%. A B+

tree has more consistent storage utilization than extendible hashing. Extendible hashing performs much better in terms of random access cost and insertion cost. This scheme is discussed in detail in Chapter III.

Larson [Lar88] compared linear hashing scheme, developed by Litwin [Lit80], with spiral storage scheme. Interestingly, both the schemes are directoryless, meaning that they do not use directory data structure which most other dynamic hashing schemes do. In spiral storage scheme, expansion process and address calculations were found to be slower and more complex than in linear hashing.

Objective

As discussed above, Fagin [Fag79] found extendible hashing fairly efficient. Litwin [Lit80] and Larson [Lar85] showed that linear hashing is algorithmically simple and computationally fast. Both the schemes are claimed to be very efficient by the respective authors. So far, however, no attempt has been made to compare extendible hashing and linear hashing schemes. Hence the objective of this thesis is to compare the performance of these two schemes by way of simulation. A number of performance factors have been evaluated. Definition of the performance factors, simplifying assumptions, and simulation details are contained in Chapter V. The simulation will lead us to conclude which hashing scheme is more efficient than the other in terms of these performance factors.

Thesis Organization

The organization of this thesis is as follows. Chapter I introduces the concept of dynamic hashing, reviews the past work and lays down the thesis objective.

Chapter II briefly reviews various dynamic hashing schemes that have been proposed over the past few years. The schemes, other than linear hashing and extendible hashing, include dynamic hashing, dynamic hashing with deferred splitting, and spiral storage.

Chapter III is devoted to extendible hashing and linear hashing schemes. This chapter includes examples illustrating directory and bucket structures, address computations, insertion of records, and expansion process.

Chapter IV discusses random number generation and its behavior, hash functions used and the expected behavior of the extendible and linear hashing schemes.

Chapter V lays down the simulation implementation details. The details include data structures for directory and bucket; algorithms to find and insert a record; simple assumptions for simulation; and the performance comparison factors.

Chapter VI embodies empirical results and observations. The observations pertain to space utilization, search costs, overflow area etc. Tables and figures are contained in the Appendix. At the end, conclusions are drawn and future research directions are suggested.

CHAPTER II

AN OVERVIEW OF DYNAMIC HASHING TECHNIQUES

A number of dynamic hashing techniques [Enb88] have been proposed over the past few years. In this chapter, we discuss some popular hashing techniques: dynamic hashing, dynamic hashing with deferred splitting, and spiral storage. Extendible hashing and linear hashing techniques are discussed in more detail in the next two chapters.

Dynamic Hashing

In dynamic hashing [Lar78], the index resides in main memory and can be organized as a forest of binary trees. Consider an example where pseudokeys are generated by the hash function on the actual keys. Each key is converted into a 7-bit binary number. The pseudokeys and their corresponding 7-bit binary numbers are given below

Key	7-bit binary number
052	0 101 010
134	1 011 100
176	1 111 110
005	0 000 101
123	1 010 011

Figure 1 shows a binary tree corresponding to the set of keys given above. The 9-bit binary representation for

the key 123 is 001 010 101. However, the key should be converted into 7 bits only. Assuming that only right-most bits are considered, the 7-bit binary representation of the key 123 is 1 010 101. Notice that we have truncated the 2 leftmost bits. To access the key 123, follow the tree starting at the root. Process the bits from left to right. Follow left subtree when 0 is encountered and follow right subtree when 1 is encountered until an external node is reached. In 1 010 101, the leftmost bit is 1. So we follow the right subtree out of the root. Then follow left for bit 0, right for bit 1 and left for bit 0. Now we have reached an external node which contains the desired key. If the desired key is not there, the search is considered unsuccessful.

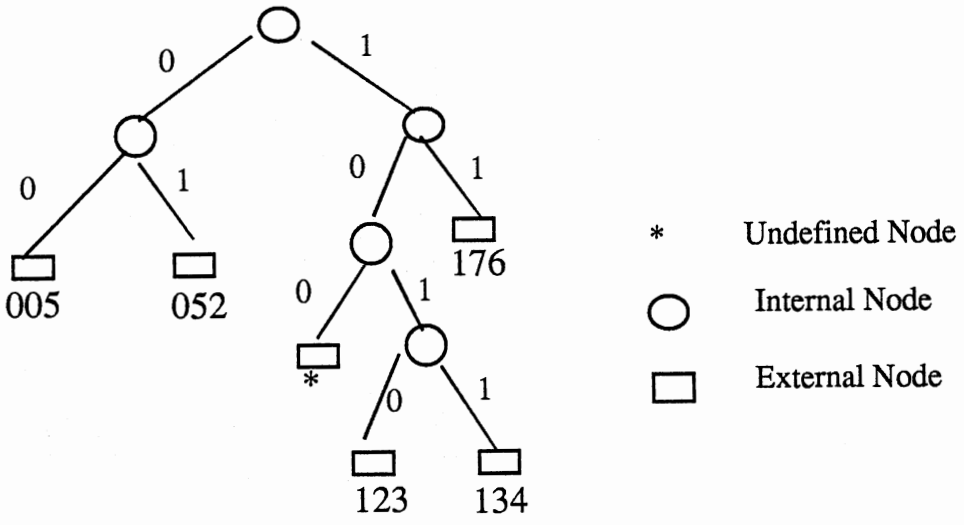
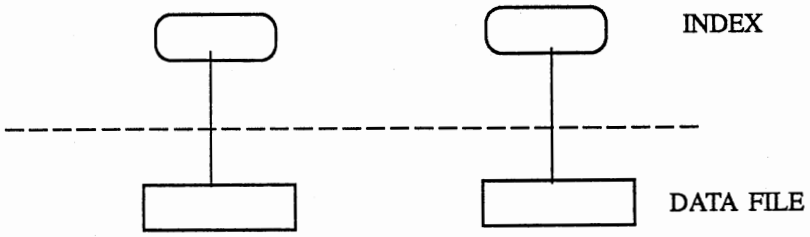


Figure 1. Binary Search Tree in Dynamic Hashing

Sometimes an external node can be undefined. In Figure

1, for example, the external node for the keys corresponding to the 7-bit binary numbers prefixed with 100 is undefined. Search is considered unsuccessful if an undefined node is encountered. Figure 2 shows a file structure as used in dynamic hashing. Data file which consists of the buckets resides on the secondary storage. Number of buckets changes according to the number of records in the file. Buckets are accessed by the index.



Source: Larson, P. "Dynamic Hashing." BIT, 18(1978), p. 185.

Figure 2. File Structure in Dynamic Hashing

Insertion involves finding the relevant bucket X which contains the key. The key is inserted, if bucket X is found and is not full. In case bucket X is full, a split is performed to distribute the keys between bucket X and a new bucket. The splitting of the bucket increases the internal path by one. Consider the tree in Figure 3(a) for the bucket size of 2. On inserting the key 050, the bucket corresponding to the search path 0 overflows. This necessitates a split operation on that bucket. The updated tree is

shown in Figure 3(b). Sometimes more than 1 split operation is necessary to break the overflow.

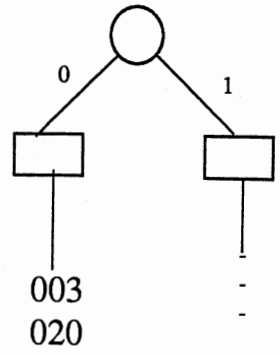
Dynamic hashing has an advantage over the static hashing scheme in that only partial reorganization of the hash file is required. However, storage utilization remains low in this scheme. In order to improve the storage utilization, splitting of a bucket should be deferred. Such a scheme is discussed next.

Dynamic Hashing With Deferred Splitting

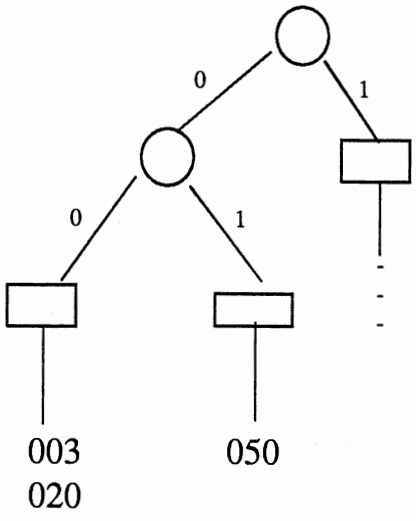
In conventional dynamic hashing discussed so far, splitting occurs as soon as the primary bucket becomes overfull. In dynamic hashing with deferred splitting [Sch81], overflow buckets are allowed to contain additional keys with a few restrictions. This is illustrated below.

Let b be the maximum bucket size and y be the "factor of b" defining maximum number of overflow keys in the overflow bucket(s). Bucket is not split until y*b keys have been inserted into it. This involves 2 cases: one when y<=2 and the other when y>2.

Let y=1.5 and b=10, then y*b=15. This implies that a maximum of 15 keys can be inserted with 10 keys into the primary bucket and 5 keys into the overflow bucket. Assume that both the primary bucket and the overflow bucket are full now. The next key will necessitate a split operation since total number of keys has exceeded 15. As a part of split operation, a new bucket is allocated and all keys in



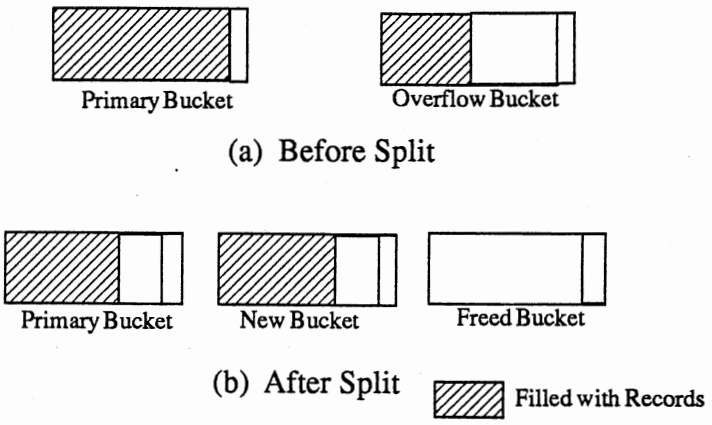
(a) Before Split



(b) After Split

Figure 3. Example of "Split" in Dynamic Hashing

the primary bucket and the overflow bucket are distributed between the primary bucket and a new bucket. The overflow bucket is released. This is shown in Figure 4.



Source: Scholl, M. "New File Organizations Based on Dynamic Hashing." ACM Transactions on Database Systems, 6, 1(Mar. 1981), p. 199.

Figure 4. Bucket Structure

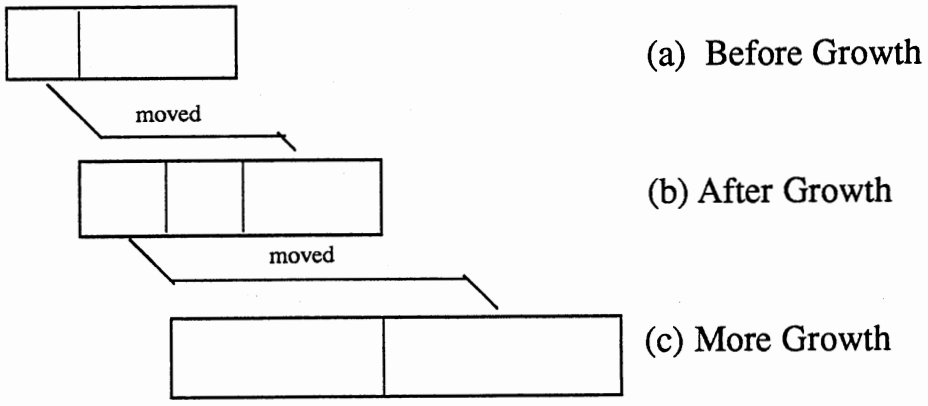
For y>2, when the primary bucket becomes full, the first overflow bucket is allocated. When the first overflow bucket becomes full, another overflow bucket is allocated. Overflow buckets are allocated until y*b keys have been inserted. Thereafter, the bucket chain has to be split in the same manner as for y<2.

The objective of deferred splitting is to defer the growth of the index and to improve the storage utilization. The small index results in faster search for the bucket. Space utilization can be further improved by using the shared overflow buckets [Sch81].

We have noticed that the dynamic hashing involves a lot of pointers. The search time under this scheme is relatively high. This is due to an index associated with the keys. The hashing scheme we discuss next does not use indices at all.

Spiral Storage

In a few hashing schemes, particularly linear hashing which shall be discussed later, the retrieval cost of a record increases and decreases in cycles [Lar88]. Such cyclic variations can be largely eliminated by using a hashing scheme called spiral storage scheme in which records are unevenly distributed to seek uniform performance [Mul85]. To be more specific, the address space at the beginning of the table has higher load than at the end.



Source: Mullin, J.K. "Spiral Storage: Efficient Dynamic Hashing with Constant Performance." Computer Journal, 28, 3(1985), p. 330.

Figure 5. File Growth with Spiral Storage

Figure 5 shows three stages of expansions. Figure 5(a) shows an address space before growth. Due to growth in records, the address space needs to be expanded. In such a situation, the records contained in the leading space in Figure 5(a) are moved to the trailing space in Figure 5(b). Notice that the trailing space in Figure 5(b) is larger than the leading space in Figure 5(a). The difference implies additional address space. Figure 5(c) shows more growth in records.

The following notations and formulas [Lar88] will be used in the example below:

```
H(K)     : A hash function for key K, 0<=H(K)<1.
d        : Growth factor.
[A,B)    : Hash interval for the entire address space.
           FRACT(S) is a fractional value of S.
           A=FRACT(S), B=FRACT(S+1) if FRACT(S) =/= 0
           A=0.0000, B=1.0000 if FRACT(S)=0
x        : A real number for the key K, 0<=x<1.
y=d**x   : Growth function.
y1=Floor(2**S) : Start bucket address for expansion
y2=Ceil(2**(S-1)) : End bucket address for expansion
```

Hash interval [A,B) can be computed by finding S in (d**S)(d-1)=required_address_space where d and the required address space are known. Let us illustrate the above by an example for d=2.

Let us start off with 1 bucket whose address space and hash intervals are given below

bucket 1 [0.0000,1.0000)

Now suppose we have to increase the address space to 2. So we need 2 buckets such that all the keys mapped into

bucket 1 can be relocated to the 2 buckets. Hence

```
(d**S)(d-1) = 2
2**S = 2
S = 1.0000
```

For S=1.0000, [A,B)=[0.0000,1.0000). To relocate the keys, [A,B) must be spread over the 2 buckets. The address of the starting bucket, y1, and the last bucket, y2, would be

```
y1 = Floor(2**S)          y2 = Ceil(2**(S+1)) - 1
   = Floor(2**1.0000)        = Ceil(2**2.0000) - 1
   = 2                       = 3
```

Hash interval for bucket 2 would be [0.0000,FRACT(r)) where r is obtained as

```
2**r = last address
2**r = 3
r = 1.5849
So, FRACT(r) = .5849
```

Thus hash interval for buckets 2 and 3 would be [0.0000,.5849) and [.5849,1.0000) respectively. The address space and the hash intervals are shown below.

```
bucket 2          [0.0000,0.5849)
bucket 3          [0.5849,1.0000)
```

Now suppose the address space needs to be increased to 3. So we need 3 buckets such that all the keys mapped into bucket 2 are relocated to 2 new buckets. Hence

```
2**S = 3
S = 1.5849
```

For S=1.5849, [A,B)=[0.5849,0.5849). To relocate the keys of bucket 2, [A,B) must be spread over the 2 buckets. The address of the starting bucket, y1, and the last bucket, y2, would be

```
y1 = Floor(2**1.5849)    y2 = Ceil(2**2.5849) - 1
   = 3                       = 5
```

Now the address space ranges from bucket 3 to bucket 5 giving us 3 buckets. The hash interval for bucket 3 would be the same as before. The hash interval for buckets 4 and 5 (the last 2 buckets in the current address space) can also be obtained in a similar fashion.

```
2**r = 5
r = 2.3219
FRACT(r) = .3219
```

Now we have the following address space and hash intervals:

```
bucket 3          [0.5849,1.0000)
bucket 4          [0.0000,0.3219)
bucket 5          [0.3219,0.5849)
```

Similarly for 4 buckets, the address space and the hash intervals would be:

```
bucket 4          [0.0000,0.3219)
bucket 5          [0.3219,0.5849)
bucket 6          [0.5849,0.8074)
bucket 7          [0.8074,1.0000)
```

If there is a key K of which, for example, H(K) = .6219, then this key will fall in address space 6. Note that every time a hash table is expanded, the first bucket from the current address space is freed and 2 new buckets are allocated at the end of the table. In practice, however, the old bucket is reused and only one new bucket is allocated.

Mullin [Mul85] analyzed and simulated the spiral storage allocation method in order to study its behavior. The major advantage of this method is that performance does

not vary cyclically during file growth or shrinkage. A high growth factor increases search time but results in less work during file expansion. However, Larson [Lar88] found this process to be very expensive. Further, he claimed that the expansion procedure of this scheme is very complex and slow.

Summary

All the hashing schemes discussed so far suffer from some disadvantages. Original "dynamic hashing" scheme gives poor storage utilization due to non-existence of overflow buckets. "Dynamic hashing with deferred splitting" needs a long index to access the required bucket, which increases the search time considerably. Spiral storage suffers from very complex and slow expansion procedure. In view of these demerits, we need a hashing scheme which:

(1) has a short search path so that the retrieval time is acceptable;

(2) has simplicity in terms of address computation and expansion procedure; and

(3) can use overflow buckets, if needed, to further improve storage utilization.

In the next chapter, we discuss two such dynamic hashing schemes called extendible hashing and linear hashing which broadly satisfy the above requirements.

CHAPTER III

EXTENDIBLE HASHING AND LINEAR HASHING

Extendible Hashing

Developed by Fagin [Fag79], extendible hashing is dependent on the number of bits extracted from the pseudo-keys. A pseudokey consists of 0's and 1's. This key is used in indexing into the bucket which contains the actual key. Given a random hash function H and an actual key K, a pseudokey K' can be computed with K'=H(K). Pseudokey must be of fixed length.

The data structure consists of a set of buckets and a directory. Usually a partial or the whole directory (depending on its size) is kept in the primary storage. Buckets must reside on the secondary storage. The buckets contain keys and the associated information.

Directory

The global depth of the directory, call it d, changes as file grows and shrinks. The directory size is computed to be 2**d. The directory contains an array of pointers to the buckets. Figure 6 shows an example of an extendible hash file for d=3.

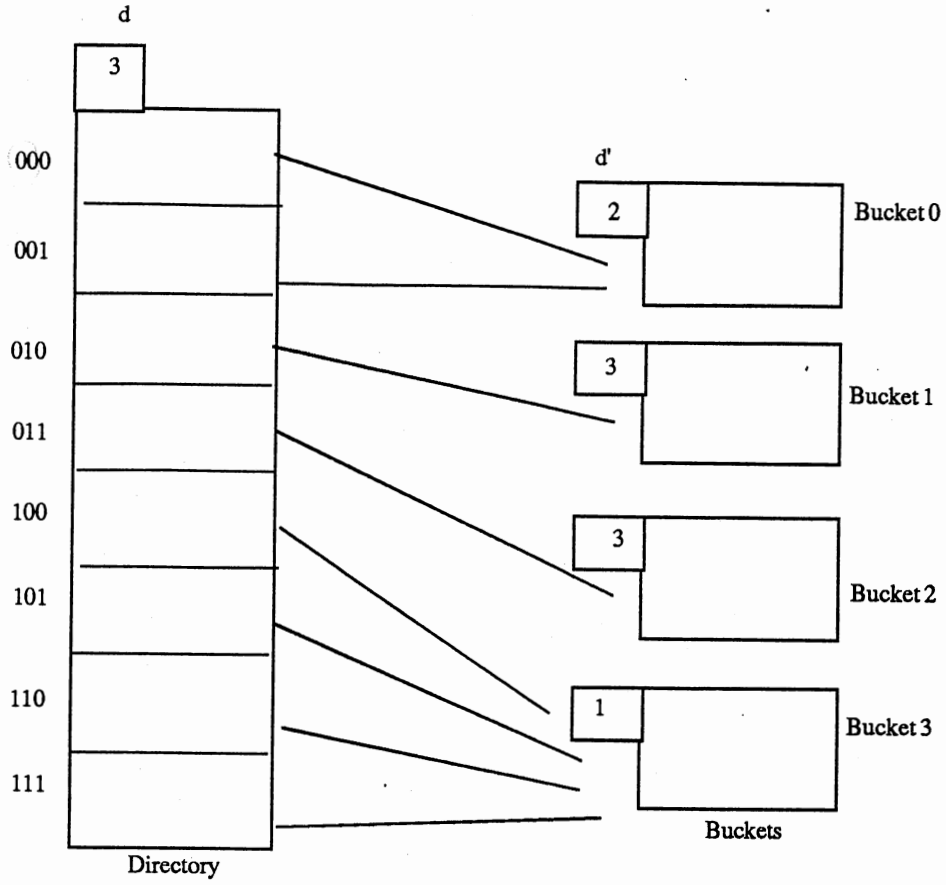


Figure 6. Directory Entries = 2**3
and 4 Buckets

Buckets

Each bucket has a local depth d' which must always be less than or equal to the global depth d. All the keys contained in a particular bucket agree in the number of bits equal to d'. If d'<d, then there exist at least 2 pointers indexing into the same bucket. To be more precise, 2**(d-d') entries point to the same bucket. In Figure 6, bucket 3 should agree only on the first (most significant) bit. Hence there are 2**(3-1)=4 entries indexing into bucket 3.

When a bucket splits into two due to an overflow, the

local depth of the two buckets involved is incremented by 1. Suppose bucket 3 of Figure 6 overflows. Hence, bucket 3 shall be split into two buckets - bucket 3 and bucket 4 as shown in Figure 7. All pseudokeys starting with 10 will hash to bucket 3 and those starting with 11 to bucket 4.

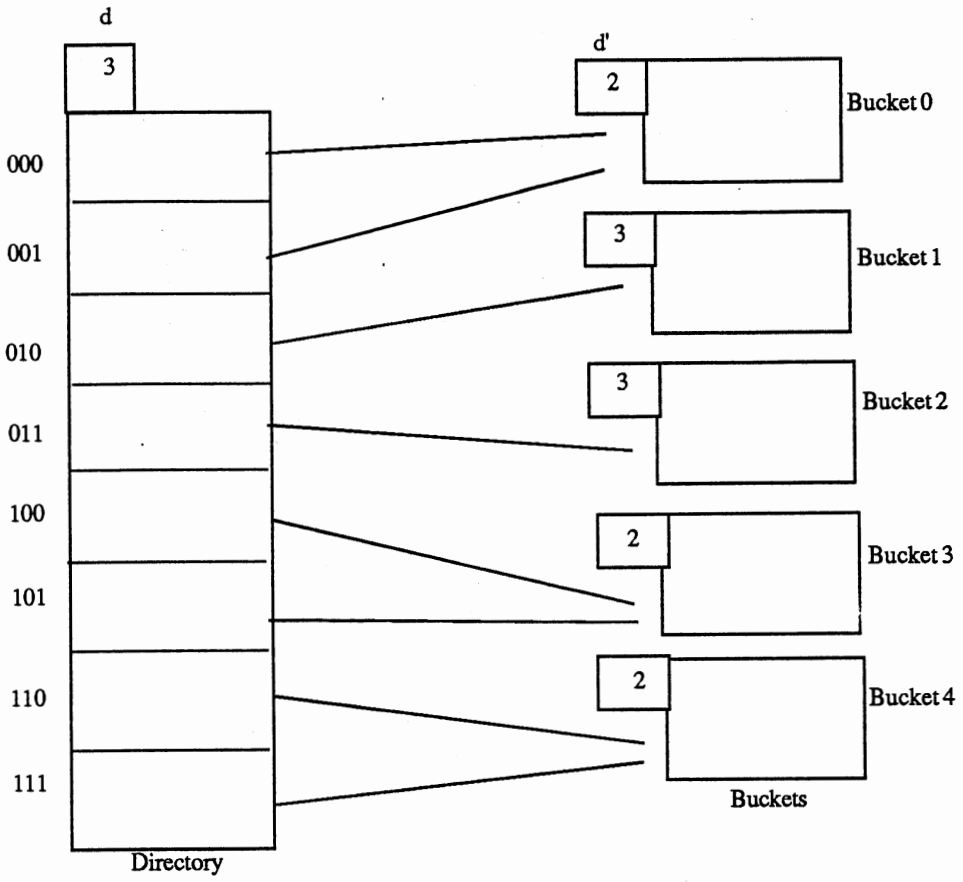


Figure 7. After Splitting Bucket 3 of Figure 6 into Buckets 3 and 4

When a bucket overflows and the local depth of the bucket equals the depth of the directory, the directory size has to be doubled. Suppose bucket 2 of Figure 7 overflows.

The local depth d' currently equals the global depth. To accommodate a new bucket resulting from the split operation, the directory size will have to be doubled by increasing the global depth d to 4. This is shown in Figure 8. The process of doubling the directory size is not expensive since no buckets other than the ones which caused the split are touched [Fag79]. It is claimed that no more than 2 accesses are required in extendible hashing - one access in locating the appropriate directory bucket (only if a partial directory is kept in the primary storage) and the other access in obtaining the appropriate bucket. This claim holds only when no overflow buckets are used. If overflow buckets are used, more than 2 accesses may be required.

Linear Hashing

Linear hashing scheme, developed by W. Litwin [Lit80], is a directoryless scheme. Non-existence of directory implies a need for less main memory. In this scheme, the address space undergoes a smooth growth with the addition of one bucket on each split at the end of the table. When a new bucket is added at the end of the address space, only a limited local reorganization is performed [Lar88]. Address computations and expansion process are illustrated in the following sections.

Example

Let N be the minimum number of buckets before any

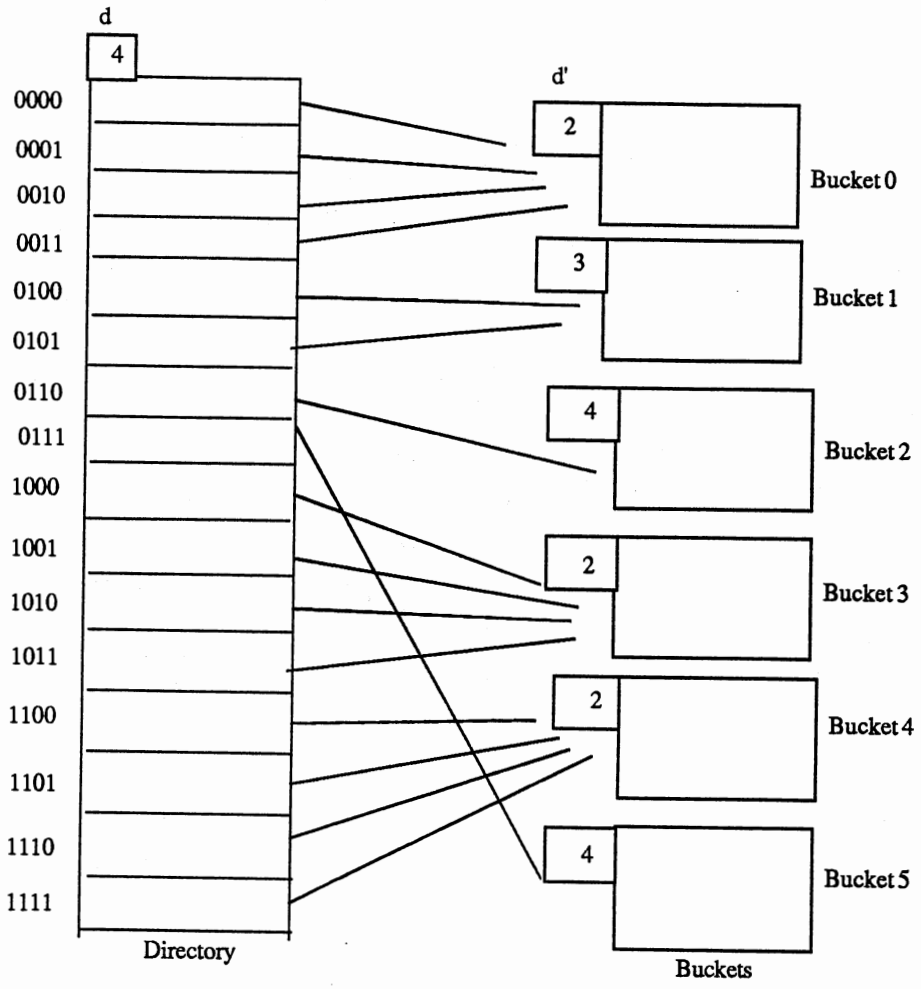


Figure 8. Hash Table Doubled After Splitting Bucket 2 of Figure 7

expansion, L be the number of times the address space has doubled and p be the bucket to be split next. Address space is expanded in a linear order i.e. from bucket 0 to bucket N*(2**L)-1. After splitting the last bucket during the current expansion, pointer p is reset to bucket 0.

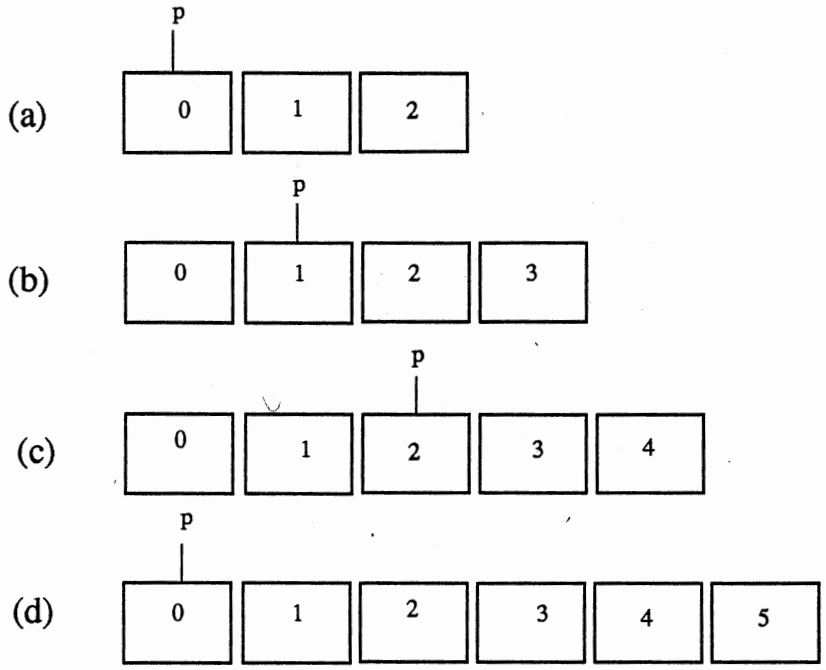


Figure 9. Expansion Process in Linear Hashing

Figure 9 illustrates the splitting process for N=3 i.e. for 3 buckets. Suppose one of the buckets overflows. So bucket 0 is split into bucket 0 and bucket 3. The updated status is depicted in Figure 9(b). Notice that p has moved to bucket 1 now.

Now suppose after a few more splits, we arrive at the situation as depicted in Figure 9(c) and bucket 2 is split

into bucket 2 and bucket 5. In such a situation pointer p is reset to bucket 0 i.e. first expansion cycle has been completed. An expansion cycle is defined to be a cycle which starts when p=0 and ends when [N*(2**L)-1]st bucket is split. Each expansion cycle results in doubling the table size relative to the size when p=0.

Address Computation

Linear splitting of buckets results in simple address calculations. The address space consists of 2 parts - split and unsplit. Both the parts are accessed using 2 different hash functions. The address is computed with the assumption that the record belongs to the unsplit part. When the computed address, on comparison with pointer p, is found to be in the split part, second hash function is used to determine whether the record is contained in the old bucket or the newly allocated bucket. In general, the current address of any record with key K can be computed as follows [Lar88]:

```
address := H(L,K)
if address < p then
   address := H(L+1,K)
```

Split Operation

This section discusses the timing and methodology for splitting a bucket. There are 2 schemes to split a bucket. The first scheme is called a controlled split. In a controlled scheme, a bucket is split only when an overall load

factor is violated. The overall load factor, call it @, is defined as the number of records inserted divided by the number of buckets allocated.

The overall load factor @ has lower and upper bounds called @(L) and @(U) respectively. If @>@(U), the bucket pointed to by p is split. If @<@(L), the bucket prior to the bucket pointed to by p should be merged with the last bucket and p should move back to the previous bucket. Figure 10 illustrates both split and merge operations.

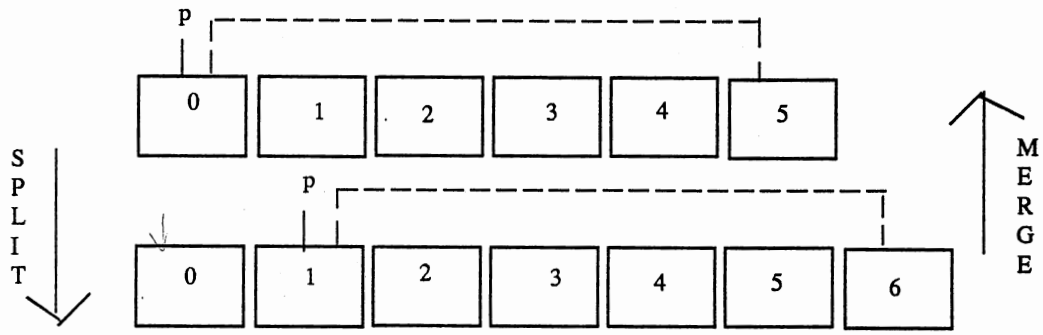
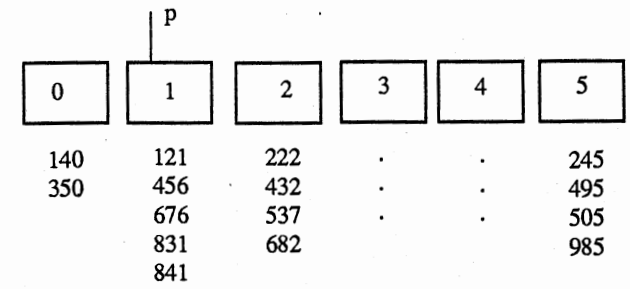
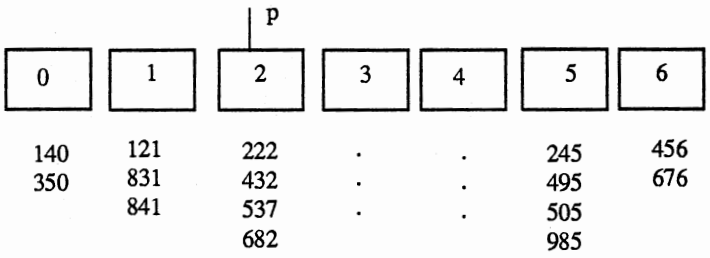


Figure 10. Split and Merge Operation

The second scheme is called uncontrolled split. In such a scheme, a bucket is split regardless of the overall load factor. The bucket size, call it b, must be predefined. A bucket is split when the current bucket size exceeds b. The name "uncontrolled split" is derived from the fact that there is no control over the space utilization. We have considered the second scheme in our simulation.



(a) Before Split



(b) After Split

Figure 11. Example of "Split" in Linear Hashing

Figure 11 illustrates the splitting process of bucket 1 for N=5. Bucket 0 has already been split in Figure 11. Now assume that some bucket overflows. This implies a split operation on the bucket 1 pointed to by p. The entries in bucket 1 are 121, 456, 676, 831, 841 as shown in Figure 11(a). All these entries are separated by the hash function H(1,K)=(K mod 10). The entries with H(1,K)=1 are retained in bucket 1 and the entries with H(1,K)=6 are hashed to bucket 6 which is allocated at the end of the hash table. This situation is shown in Figure 11(b). Notice that pointer p has moved to the bucket 2. That means bucket 2 shall be split next time.

Summary

In this chapter, we studied extendible hashing and linear hashing schemes. Both the schemes allow smooth growth in address space by allocating one bucket at a time. However, there are some differences between the two schemes, which are mentioned below:

(1) Extendible hashing uses directory but linear hashing does not.

(2) Overflow space requirement is mandatory in linear hashing while this can be avoided in extendible hashing by propagating split operation until the overflow space is released.

(3) In extendible hashing, split operation is performed on the bucket which overflows. In linear hashing, split

operation is performed on the bucket pointed to by pointer p regardless of where the overflow has occurred.

Despite the above mentioned differences, the address computation and the expansion process are simple to understand and easy to implement in both the schemes. The relevant data structures, algorithms and simulation implementation details of both the schemes are discussed in Chapter V.

CHAPTER IV

ANALYSIS OF EXTENDIBLE HASHING AND LINEAR HASHING

This chapter discusses the random function for generating the keys, the hash functions for linear hashing and extendible hashing, and briefly revisits the expected performance analysis for both schemes.

Random Function

We assume that the keys are uniformly distributed [Car79]. In both hashing schemes, uniform distribution results in improved storage utilization and better search times. If keys are not uniformly distributed, an appropriate conversion function can be used to get a nearly uniform set of keys. Several such methods have been discussed by Folk [Fol87]. This assumption implies that each key has equal probability of being accessed. Hence we need a function that can uniformly distribute the keys.

The XELOS (a version of UNIX) operating system [XEL85] on Perkin-Elmer 3230 provides built-in random functions "lrand48", "drand48", "lcong48" etc. An initial seed is provided by a function called "srand48". All three random functions work by a linear congruential formula. Upon

experimentation, it was discovered that the results were almost identical across different random functions. Hence, "lrand48" was arbitrarily selected for simulation. However, the choice of "lrand48" does not imply any bias against "drand48" and "lcong48". In other words, the simulation results will remain the same with any of the three random functions.

Hash Function

There are several hashing methods which can be potentially better than random. These methods include key-folding method; division method; mid-square method; and radix transformation method. Folk [Fol87] found radix transformation method to be more reliable than the others. Radix transformation method involves transforming the actual key into some decimal or hex number, and then taking its modulo. In our simulation, we simply consider the modulo arithmetic to find the bucket address since the keys are already transformed into hex numbers. In linear hashing, only two hash functions HASH1 and HASH2 are used at any given time. HASH1 deals with the unsplit buckets, whereas HASH2 deals with the split buckets. When a bucket is split, HASH2 hash function is used in separating the keys between the old bucket and the new bucket.

In extendible hashing, the prefix d (global depth of the directory) bits of the keys are extracted from the keys and used in indexing into the directory location which

points to the relevant bucket. When a bucket is split, the local depth of that bucket is used. The keys are separated by looking at (local_depth+1)st bit. If that bit is 0, retain that key in the old bucket; otherwise move it to the new bucket.

Analysis

The expected behavior of extendible hashing and linear hashing can be mathematically analyzed [[Fag79] [Ram82] [Lar82,83,85]. The simulation results discussed in Chapter VI are mostly consistent with the mathematical analysis. In this section, we simply outline the work done by others. For elaborate derivations and details, refer to the literature cited above. The costs are calculated in terms of secondary storage accesses.

Extendible Hashing

It is assumed that the entire directory is kept in main memory. The following notations have been used:

b Bucket Capacity
n Total number of records
ln Logarithmic value with base 2
e Inverse of ln

Some important costs are (Sources: [Men82] and [Fla83]):

Insertion Cost = 1 + [1/(b ln 2)]

Search Cost = 1

Directory Size = [e n**(1+1/b)] / (b ln 2)

Storage Utilization = ln 2

Number of Buckets = n / (b ln 2)

Linear Hashing

The following notations have been used:

y	Number of records/bucket
b	Primary bucket capacity
c	Secondary bucket capacity
z	Load factor = y/b
P(i,z)	probability that i records hash to a bucket, given the load factor z. This implies binomial probability. For infinite number of records and buckets, binomial probabilities converge to Poisson probabilities [Lar83]. Hence P(i,z) = [e**(-zb) * (zb)**i] / i!
k	Number of buckets on a bucket chain, k>=1
j	Number of overflow records in the last overflow bucket only if overflow bucket exists
s(z),S(z,x)	Cost for successful search
u(z),U(z,x)	cost for unsuccessful search
a(z),A(z,x)	Cost for insertion
t(z),T(z,x)	Number of slots allocated per bucket
E(z,x)	Cost for expansion
V(z,x)	Overflow space per record
$$	Sign for infinity

Some important costs are (Sources: [Mul81] and [Lar85]):

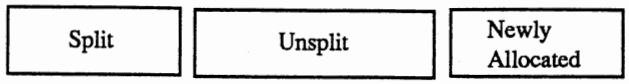
$$s(z) = 1+(1/zb) \sum_{k=1}^{\$\$} k \sum_{j=1}^{c} [(k-1)c/2+j]P(b+(k-1)c+j,z)$$

$$u(z) = 1 + \sum_{k=1}^{\$\$} k \sum_{j=1}^{c} P(b+(k-1)c+j,z)$$

$$t(z) = b + c \sum_{k=1}^{\$\$} k \sum_{j=1}^{c} P(b+(k-1)c+j,z)$$

$$a(z) = 1 + u(z) + \sum_{k=0}^{\infty} P(b+(k-1)c+j,z)$$

A linear hash table consists of 2 parts: (1) the buckets that have not yet been split during the current expansion, and (2) the buckets that have been split during current expansion (see Figure 12). Split part and newly allocated part should be considered identical.



Fraction	x/2	1-x	x/2
Expected no. of records	z/2	z	z/2

Figure 12 : Representation of split, unsplit and newly allocated parts.

Let x indicate the proportion of the file which has been split. Hence

$S(z,x) = xs(z/2) + (1-x)s(z)$

$U(z,x) = xu(z/2) + (1-x)u(z)$

$T(z,x) = [2xt(z/2)+(1-x)z]/(zb)$

$V(z,x) = [2x(t(z/2)-b)+(1-x)(t(z)-b)]/(zb)$

$A(z,x) = xa(z/2) + (1-x)a(z)$

$E(z,x) = [u(z)+2u(z/2)](1-x)/b$

CHAPTER V

IMPLEMENTATION DETAILS

This chapter describes how the extendible hashing scheme and the linear hashing scheme are simulated. The description includes data structures, algorithms and the performance factors for both schemes.

Extendible Hashing

Data Structures

There are 2 main data structures to be used to implement the extendible hashing scheme.

1. Directory 2. Bucket

The directory contains pointers to the bucket which holds the records. Some consecutive entries in the directory may have the same value.

Bucket has a fixed capacity in terms of number of records. We assume that the keys are stored in a sequential fashion without any order. Each bucket is linked to the next bucket except the last bucket.

Algorithms

Notations. The following notations have been used in the algorithms:

K : Key
P : Bucket
T : New bucket
A : Temporary storage area
d : Global depth of the directory
d': Local depth of P

Algorithm 1: Find. 1. Get the key K.

2. Extract the first d bits of the key.

3. Determine the entry in the directory based on the bits extracted.

4. Follow the bucket pointer to a bucket P.

5. Search the keys in bucket P in a sequential fashion. If key K is found, return "successful" - else (i) set P to the next bucket pointer, and (ii) if P is nil, return "unsuccessful" - else go to step(5).

Algorithm 2: Insert. 1. Apply "Find" to search the key K.

2. If key K exists (successful search), then

- Print message: key K already exists.
- Return.

3. If bucket P is full, go to step 5.

4. Insert key K and increment the counter for number of records in bucket P by one and then return.

5. The bucket P will overflow if the key K is inserted. Obtain new bucket T.

6. Obtain a temporary area A and store all the records of bucket P along with the new record associated with key K in A.

7. Set the local depth of bucket P and T to d'+1.
8. If the new local depth of bucket P exceeds the directory depth d, then do the following:
   - Double the size of the directory.
   - Increment the depth d of the directory by 1.
   - Update the pointers in the directory.
   - Set the count for the number of records on bucket P and T to 0.
9. Insert all records one at a time from the temporary area A into bucket P or bucket T depending upon the key. Note that no "Find" operation is needed for these records and only bucket P and bucket T are going to be affected. To insert, repeat step 4 as many times as the number of records in A.

Linear Hashing

Data Structures

Unlike extendible hashing, the linear hashing can be implemented using only the bucket structure. It should be noted that no directory is used in linear hashing since it is a directoryless scheme and the relevant buckets can be accessed directly by the hash functions.

Bucket has a fixed capacity in terms of number of records. The keys are stored in a sequential fashion without any order. There is a pointer p which points to the

bucket to be split next when the overflow occurs.

Algorithms

Notations. The following notations have been used in the algorithms:

K : Key
L : Number of times the table size has doubled
N : Minimum number of initial buckets in the linear hash table
P : Bucket of a fixed size
T : New bucket
A : Temporary storage area
p : Pointer to the next bucket to be split

Algorithm 3: Find. 1. Get the key K.

2. Hash key K according to P=H(L,K). Key K may reside in bucket P.

3. If P<p, then bucket P has already been split. Hence hash key K using P=H(L+1,K).

4. Search the keys in bucket P in a sequential fashion. If key K is found, return "successful" - else (i) set P to the next bucket pointer, and (ii) if P is nil, return "unsuccessful" - else go to step(4).

Algorithm 4: Insert. 1. Apply "Find" to search key K.

2. If key K exists (successful search), then

- Print message: key K already exists.
- Return.

3. Insert key K in bucket P, increment the counter for number of records in bucket P by 1.

4. Return if bucket P does not overflow. Otherwise, obtain

a temporary area A and store all the records contained in the bucket pointed to by p.

5. Obtain a new bucket T.

6. Insert all the records one by one from the temporary area A into the bucket pointed to by p and bucket T by using a hash function H(L+1,K) where K refers to the keys in A. If necessary, use overflow buckets chained with the bucket pointed to by p or with bucket T.

7. Increment the pointer p by 1.

8. If p = N*(2**L), reset p to 0 and L to L+1. Return.

Assumptions

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

(1) The keys are uniformly distributed, meaning that each key has equal probability of being accessed.

(2) Records are of fixed length.

(3) Bucket capacity is fixed in terms of 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) Extendible Hashing: (a) Most significant bits are extracted from the key to find the directory entry. (b) Overflow bucket is split only once. In other words,

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) Linear Hashing: A simple division method with modulo arithmetic is used to find the relevant bucket.

According to assumption (1), we use a random function "lrand48" (discussed in Chapter IV) 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 at least 32-bit arithmetic. Further, the hash functions used in simulation also satisfy the basic properties listed in [Car79] [Knu73].

Comparison Factors

The following factors are used to compare the linear hashing and the extendible hashing schemes:

(1) Average Space Utilization: Divide the current number of records in all the buckets by the maximum number of records that these buckets can hold. Here the buckets include primary as well as overflow buckets.

(2) Number of Buckets: Number of both primary and overflow buckets used in inserting the records. This factor is tied to average space utilization. The higher the space utilization, the fewer the buckets.

(3) Average Unsuccessful Search Cost (in terms of bucket accesses): Reading a primary or an overflow bucket amounts to 1 bucket access. Hence, add the number of buckets, both primary and overflow, that are accessed to search a non-existent record. Divide that sum by the total number of unsuccessful search operations. The cost for a single unsuccessful search is equal to the number of buckets on a particular chain i.e. 1 for the primary bucket plus 1 for each additional overflow bucket attached to the primary bucket.

(4) Average Successful Search Cost (in terms of bucket accesses): Reading a primary or an overflow bucket amounts to 1 bucket access. Hence, add the number of buckets, both primary and overflow, that are accessed to search an existing record. Divide that sum by the total number of successful search operations. Only 1 access is required to retrieve the record contained in the primary bucket. If the record belongs to the overflow bucket, 2 or more accesses are required.

(5) Cost of Expansion: Expansion cost and split cost are synonymously used. Since the expansion process for both

extendible and linear hashing is different, the expansion cost calculations also differ.

Extendible Hashing: 1 or more accesses to write the old bucket
+ 1 or more accesses to write the new bucket
+ 1 access to update the directory pointer (if directory is on secondary storage)
+ Accesses to update the directory pointers in case of doubling the directory size (if directory is on secondary storage)

Linear Hashing: 1 or more accesses to read the bucket to be split
+ 1 or more accesses to write the old bucket
+ 1 or more accesses to write the new bucket

(6) Cost of Insertion: Cost of insertion is based on the number of accesses needed to insert a new record. Cost of connecting a record to the bucket and cost of allocating a new bucket in case of split are being ignored in our analysis. Such costs are system dependent and stay constant for all the records and the buckets. Further they are identical for both the hashing schemes.

Cost of insertion consists of unsuccessful search cost and cost of expansion in case of split. The unsuccessful search cost is the same as (3) above. In case of split, all the records contained in the bucket have to be redistributed between the old bucket and the newly allocated bucket. In linear hashing, 1 extra access is required to update the next pointer if the last bucket on the chain is full.

(7) Size of the Overflow Area: Count the overflow buckets chained with the primary buckets.

(8) CPU time for insertion: System dependent functions are used to derive the CPU time for inserting the keys.

The above performance factors are plotted on the line chart against the number of records currently in the table. Also average, maximum and minimum values have been generated for space utilization, search cost, split cost, and insertion cost. Tables and figures are contained in the Appendix.

CHAPTER VI

SIMULATION RESULTS AND CONCLUSIONS

In this chapter, LINHASH and EXHASH mean linear hashing and extendible hashing, respectively. All the figures and tables referred to are contained in the Appendix.

Simulation Results

For all the bucket sizes, EXHASH has produced consistently better storage utilization than LINHASH. LINHASH, as mentioned before, 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 13,14,15,16 and Table I). 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 needs more buckets to hold the same number of records than EXHASH does (see Figures 17,18,19,20).

LINHASH, for all the bucket sizes, has done better as to unsuccessful search cost than EXHASH. In general, unsuccessful search becomes less costly as the bucket size rises. It is interesting to note that on an average unsuccessful

search cost stays close to 1 for all the bucket sizes in LINHASH (see Figures 21,22,23,24 and Table II). Similar observations hold true for successful search cost (see Figures 25,26,27,28 and Table III). It is observed that 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. Note that overflow buckets are mandatory in LINHASH. In EXHASH, the search cost can be kept close to 1 regardless of the bucket size if the entire directory can be kept in the main memory.

Splitting of a bucket or table expansion 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 29,30,31,32 and Table IV). Insertion cost is slightly higher in EXHASH for the bucket sizes 10, 20 and 30. However, for the bucket size 50, this cost is slightly less in EXHASH (see Figures 33,34,35,36 and Table V).

As expected, LINHASH performed poorly as 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 percent of the total space should be marked as an overflow area in LINHASH. Overflow buckets are almost non-existent in EXHASH (see Figures 37,38,39,40). As mentioned before, overflow buckets are mandatory in EXHASH.

LINHASH is definitely superior to EXHASH as far as cpu time for insertion is concerned. However, it is warned that

the cpu time referred here is not the actual time. The better way to evaluate Figures 41,42,43,44 is to compute the difference between the two curves. The difference can be interpreted as the time spent on directory doubling and additional address computations in EXHASH. Obviously, the time spent on directory doubling is enormous and it has increased very sharply with increasing directory size. With the bucket size 10 and 50,000 records, the directory size grows to 2**16, which implies that the directory is doubled 16 times during file expansion.

Based on the simulation results, linear hashing scheme is recommended if the main memory is at a premium since it does not need any directory. Address computations and expansion process are also simple and efficient. 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 access time may sometimes be high. Extendible hashing scheme could be useful if sufficient main memory is available to hold the directory. If the file size grows and shrinks frequently, doubling and halving the directory size may become very expensive. In both the schemes, the bucket size has not affected the performance significantly. However, a bucket size of 30 seems to be a good choice since it gives fairly reasonable storage utilization and access times.

Conclusion

Dynamic hashing is a well-known scheme for organizing direct access files. Dynamic hashing schemes include dynamic hashing, dynamic hashing with deferred splitting, spiral storage scheme, extendible hashing, and linear hashing. Both extendible hashing and linear hashing schemes are considered algorithmically simple and efficient. Hence, we have simulated both the schemes to evaluate their relative performance.

Simulation has been performed on Perkin-Elmer 3230 running under XELOS operating system. We have assumed that the keys are uniformly distributed; records and buckets are fixed in size; and main memory can hold upto a maximum of 1024 directory entries in extendible hashing. The performance comparison factors include average space utilization, average successful and unsuccessful search cost in terms of bucket accesses, cost of expansion, cost of insertion, size of the overflow area, and CPU time for insertion.

Simulation results have suggested that extendible hashing scheme uses the space more efficiently while linear hashing produces better search times. It has been observed that with large number of keys (e.g. 50000) and a small bucket size (e.g. 10), the directory size in extendible hashing may grow very large. Insertions and expansion are less costly in linear hashing. However, linear hashing suffers from the problem of large overflow area.

Approximately 10 percent of the total space is needed for overflow purposes. Because of simplicity in address computation and absence of directory, linear hashing gives better CPU time for insertion. Based on these observations, linear hashing scheme is recommended if main memory is at a premium. This scheme can be particularly useful in a small computer environment.

Suggested Future Work

The following suggestions are made for the future work:

(1) Use a non-uniform distribution of keys [Enb88].

(2) Use frequency counts of the records in hashing.

(3) Consider splitting more than one bucket in LINHASH to reduce the storage fluctuations. Some work has already been done on partial expansion of table in LINHASH [Ram84].

(4) Consider the existence of cache memory. Also, use different sizes of buffer to see how the performance is affected.

BIBLIOGRAPHY

[Car79] Carter, J.L. and Wegman, M. "Universal Class of Hash Functions." Journal of Computer & System Sciences, 18, 1(1979), pp. 143-154.

[Cha85] Chang, H. A Study of Dynamic Hashing and Dynamic Hashing with Deferred Splitting. M.S. Thesis, Oklahoma State University, Stillwater, OK 74074, 1985.

[Enb88] Enbody, R.J. and Du, H.C. "Dynamic Hashing Schemes." ACM Computing Surveys, 20, 2(June 1988), pp. 85-113.

[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(Sep. 1979), pp. 315-344.

[Fla83] Flajolet, P. "On the Performance Evaluation of Extendible Hashing and Trie Searching." Acta Informtica, 20, (1983), pp. 345-369.

[Fol87] Folk, M.J. and Zoellick, B. File Structures: A Conceptual Toolkit. Reading, MA : Addison-Wesley 1987.

[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(1978), pp. 184-201.

[Lar80] Larson, P. "Linear Hashing with Partial Expansions." Proc. of the 6th International Conference on Very Large Databases, 1980, pp. 224-232.

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

[Lar83] Larson, P. "Analysis of Uniform Hashing."Journal of ACM, 30, 4(1983), pp. 805-819.

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

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

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

[Men82] Mendelson, H. "Analysis of Extendible Hashing." IEEE Transactions on Software Engineering, SE-8, 6(Nov. 1982), pp. 661-619.

[Mul81] Mullin, J.K. "Tightly Controlled Linear Hashing Without Separate Overflow Storage." BIT, 21, 4(1981), pp. 389-400.

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

[Pat87] Patel, H.D. Analysis and Comparison of Extendible Hashing and B+ Trees Access Methods. M.S. Thesis, Oklahoma State University, Stillwater, OK 74074, 1987.

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

[Ram84] Rammohanrao, K. and Sachs-Davis "Recursive Linear Hashing." ACM Transactions on Database Systems, 9, 3(1984), pp. 369-391.

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

[XEL85] XELOS: Programmer Reference, (60-183F04). Ocean-port, New Jersey: Perkin-Elmer Corporation, 1985.

APPENDIX

TABLE I

Storage Utilization

	Bucket Size	Avg.	Min.	Max.
EXHASH	10	.693	.667	.732
	20	.692	.664	.741
	30	.694	.625	.769
	50	.703	.500	.788
LINHASH	10	.639	.556	.714
	20	.621	.522	.833
	30	.637	.515	.786
	50	.653	.500	.791

TABLE II

Unsuccessful Search Cost

	Bucket Size	Avg.	Min.	Max.
EXHASH	10	1.83	1.00	1.99
	20	1.61	1.00	1.88
	30	1.41	1.00	1.75
	50	1.17	1.00	1.50
LINHASH	10	1.15	1.00	1.25
	20	1.08	1.00	1.18
	30	1.04	1.00	1.20
	50	1.04	1.00	1.14

TABLE III

Successful Search Cost

	Bucket Size	Avg.	Min.	Max.
EXHASH	10	1.83	1.00	1.98
	20	1.61	1.00	1.88
	30	1.41	1.00	1.75
	50	1.17	1.00	1.50
LINHASH	10	1.03	1.00	1.06
	20	1.01	1.00	1.03
	30	1.00	1.00	1.02
	50	1.00	1.00	1.01

TABLE IV

Split Cost

	Bucket Size	Avg.	Min.	Max.
EXHASH	10	3.34	2.00	3.77
	20	2.83	2.00	3.44
	30	2.41	2.00	2.99
	50	2.05	2.00	2.46
LINHASH	10	3.29	3.23	3.43
	20	3.26	3.20	4.00
	30	3.25	3.19	4.00
	50	3.23	3.00	4.00

TABLE V

Insertion Cost

	Bucket Size	Avg.	Min.	Max.
EXHASH	10	2.93	2.10	3.23
	20	2.46	2.03	2.79
	30	2.22	2.01	2.50
	50	2.06	2.00	2.21
LINHASH	10	2.67	2.44	2.75
	20	2.34	2.10	2.40
	30	2.22	2.05	2.27
	50	2.13	2.00	2.16

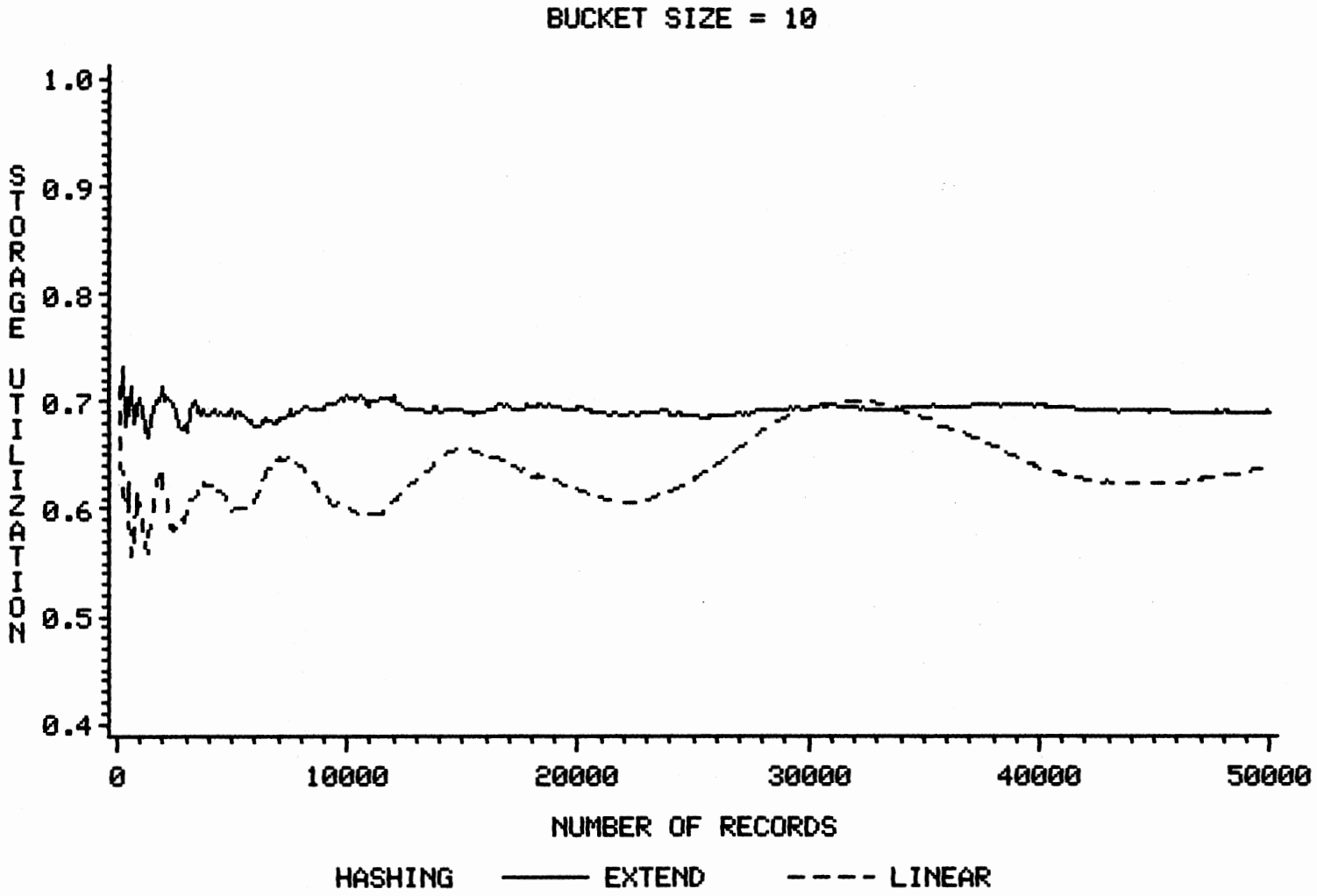


FIGURE 13. STORAGE UTILIZATION VS. NUMBER OF RECORDS

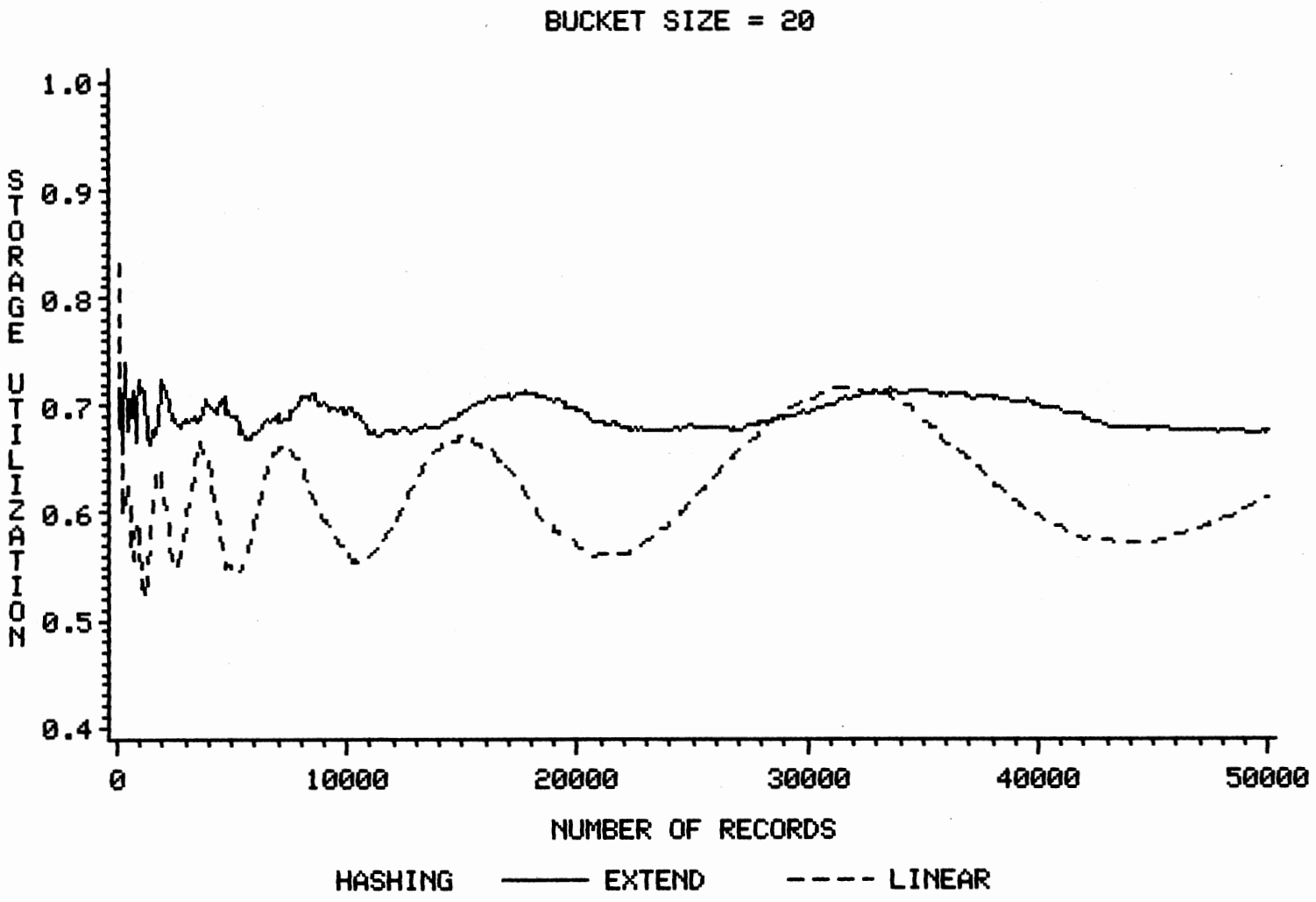


FIGURE 14. STORAGE UTILIZATION VS. NUMBER OF RECORDS

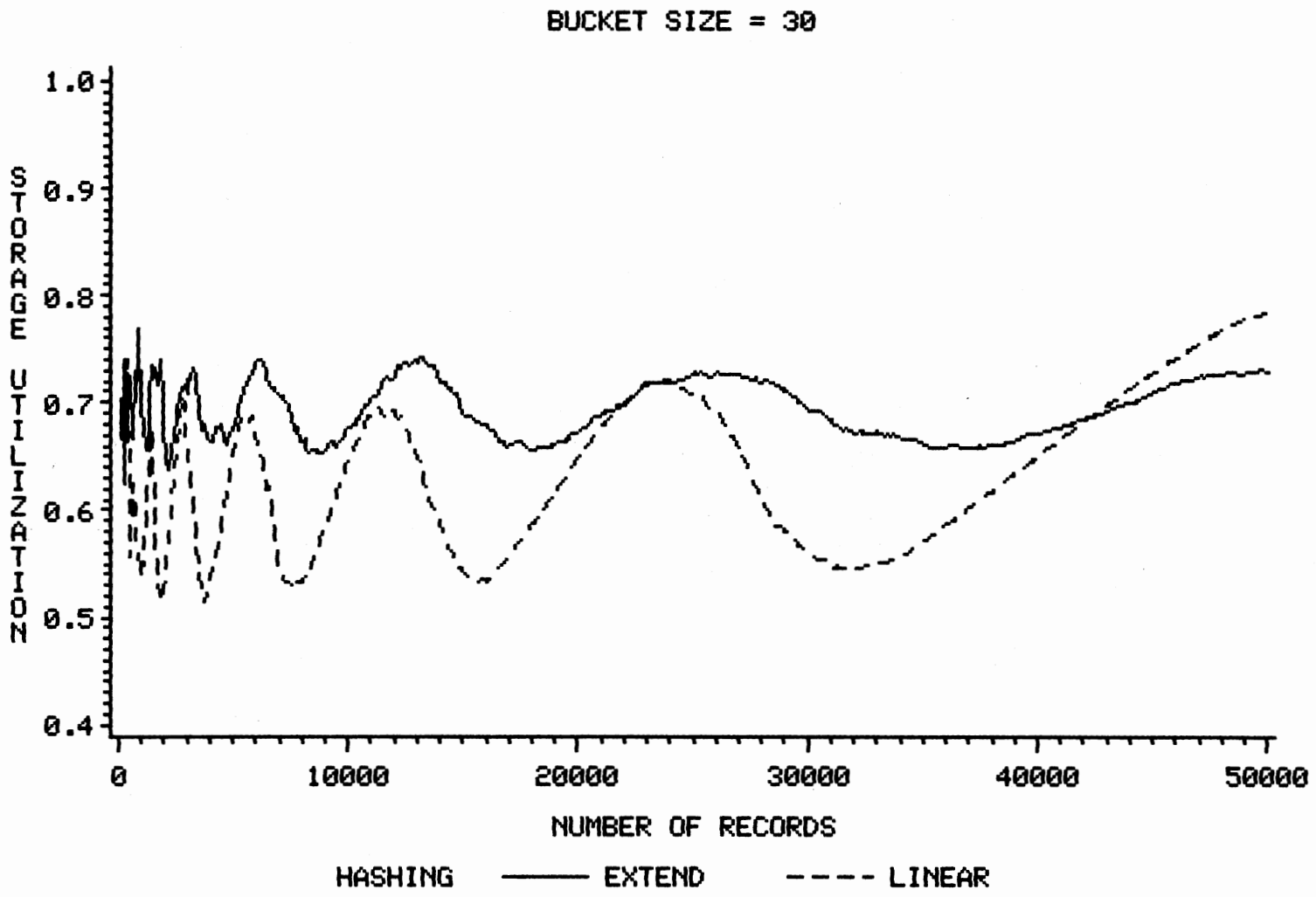


FIGURE 15. STORAGE UTILIZATION VS. NUMBER OF RECORDS

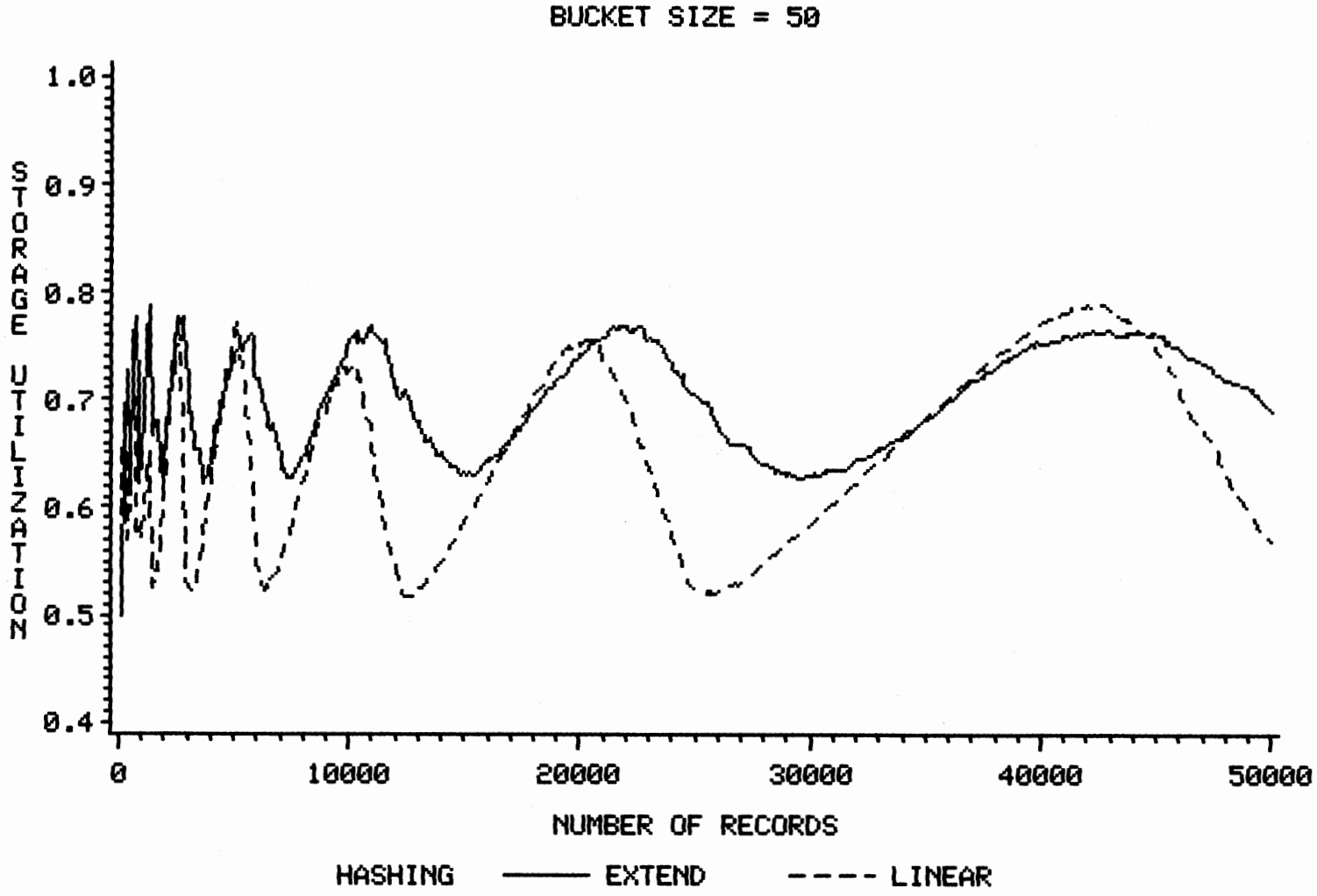


FIGURE 16. STORAGE UTILIZATION VS. NUMBER OF RECORDS

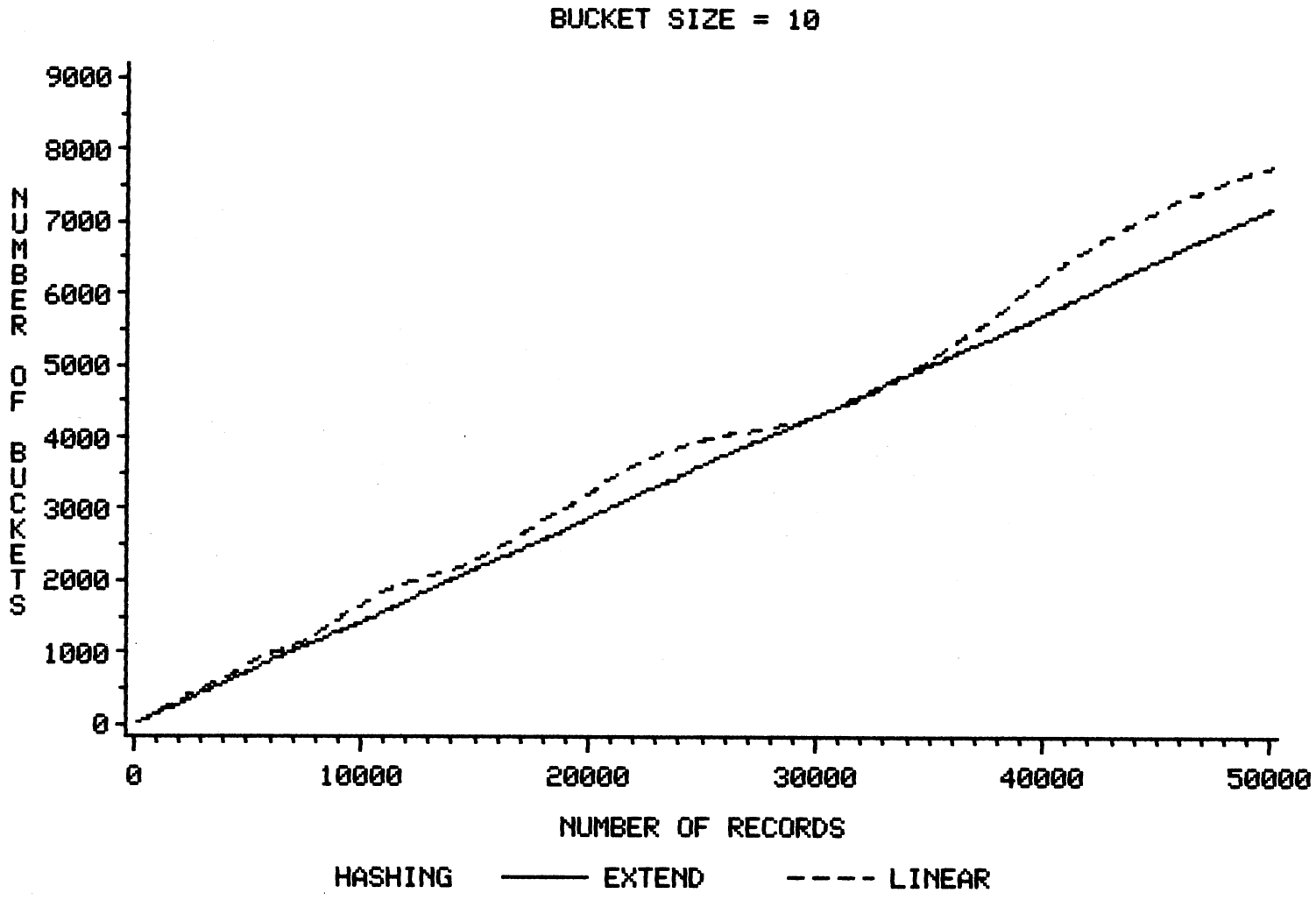


FIGURE 17. NUMBER OF BUCKETS VS. NUMBER OF RECORDS

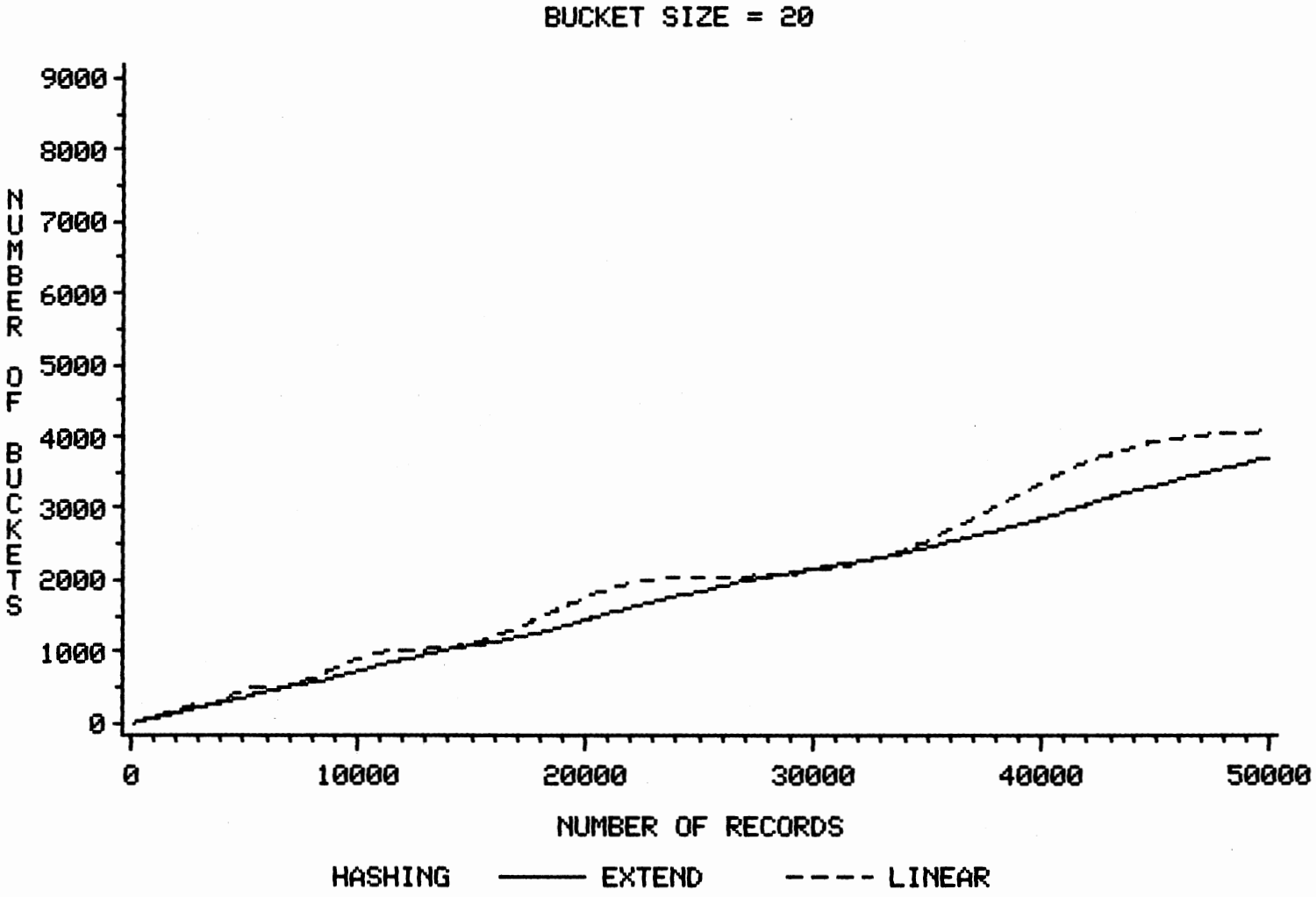


FIGURE 18. NUMBER OF BUCKETS VS. NUMBER OF RECORDS

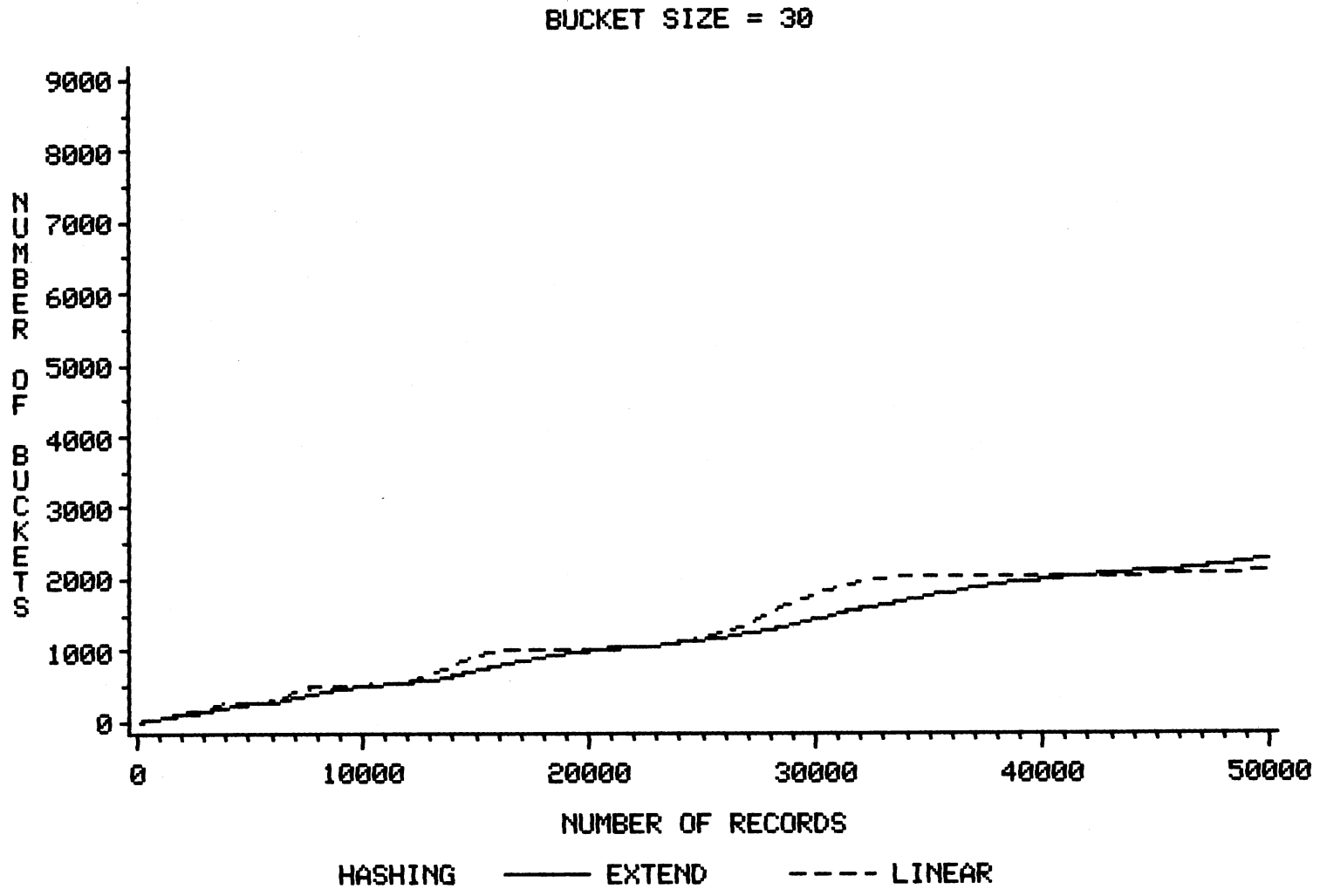


FIGURE 19. NUMBER OF BUCKETS VS. NUMBER OF RECORDS

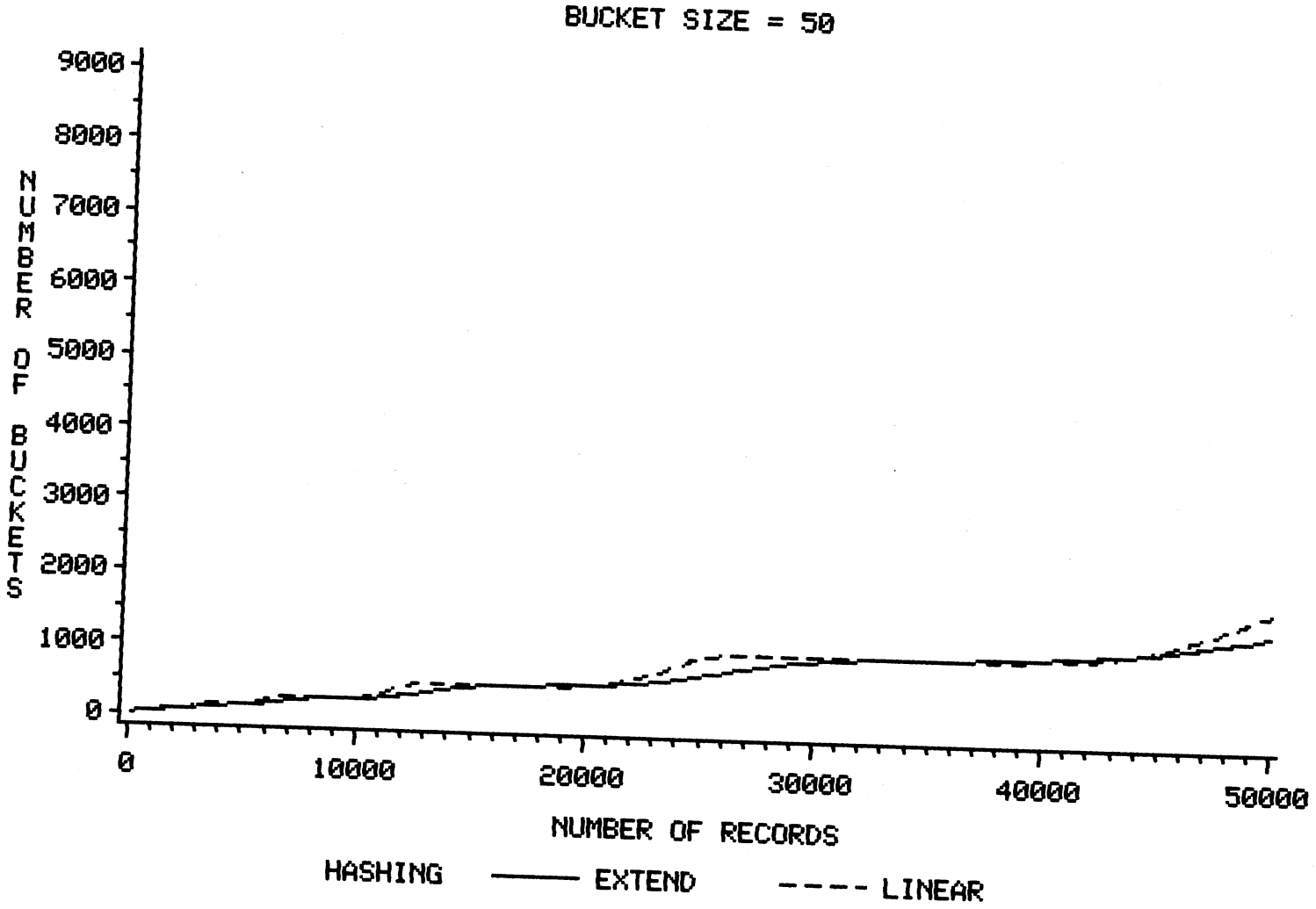


FIGURE 20. NUMBER OF BUCKETS VS. NUMBER OF RECORDS

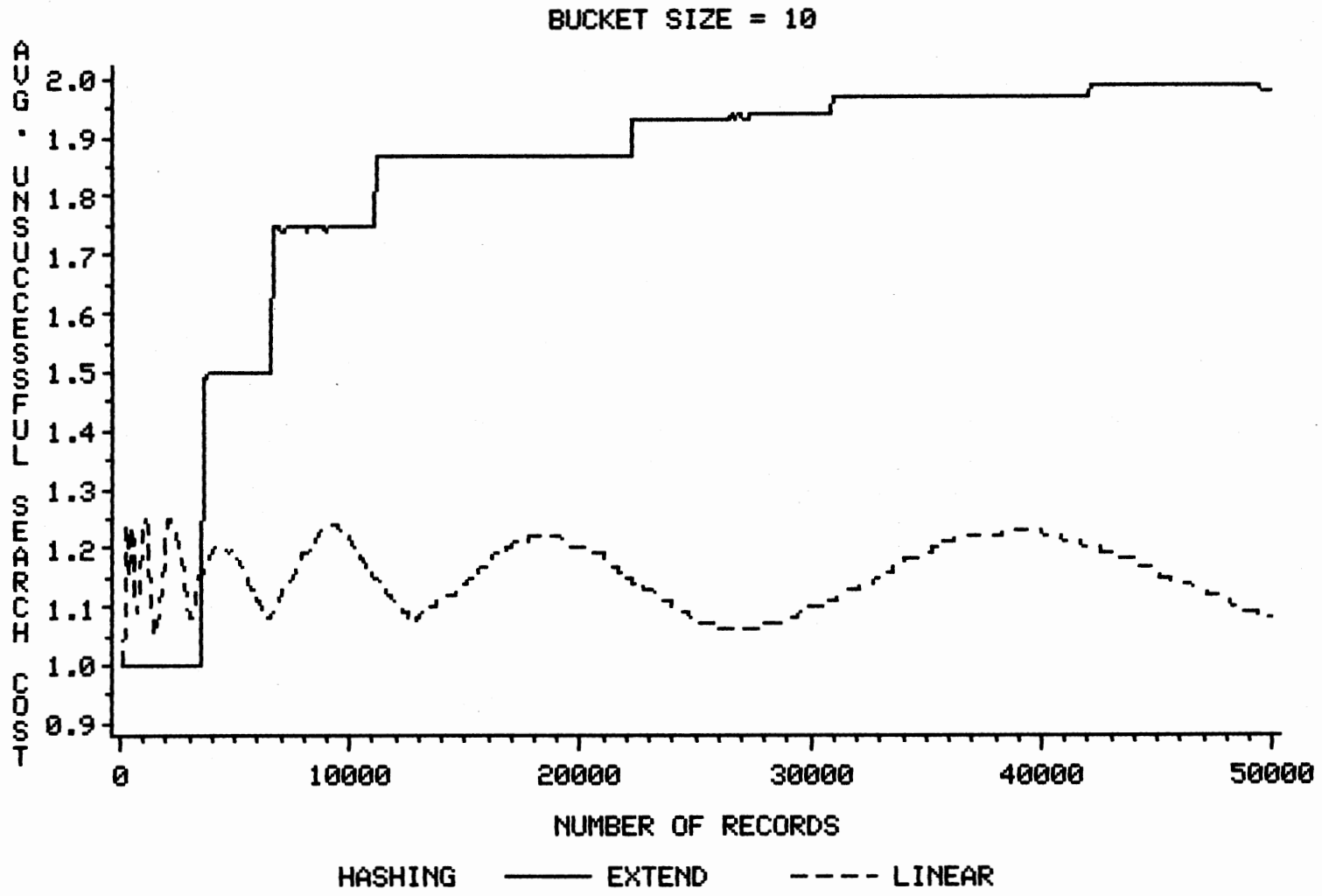


FIGURE 21. UNSUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

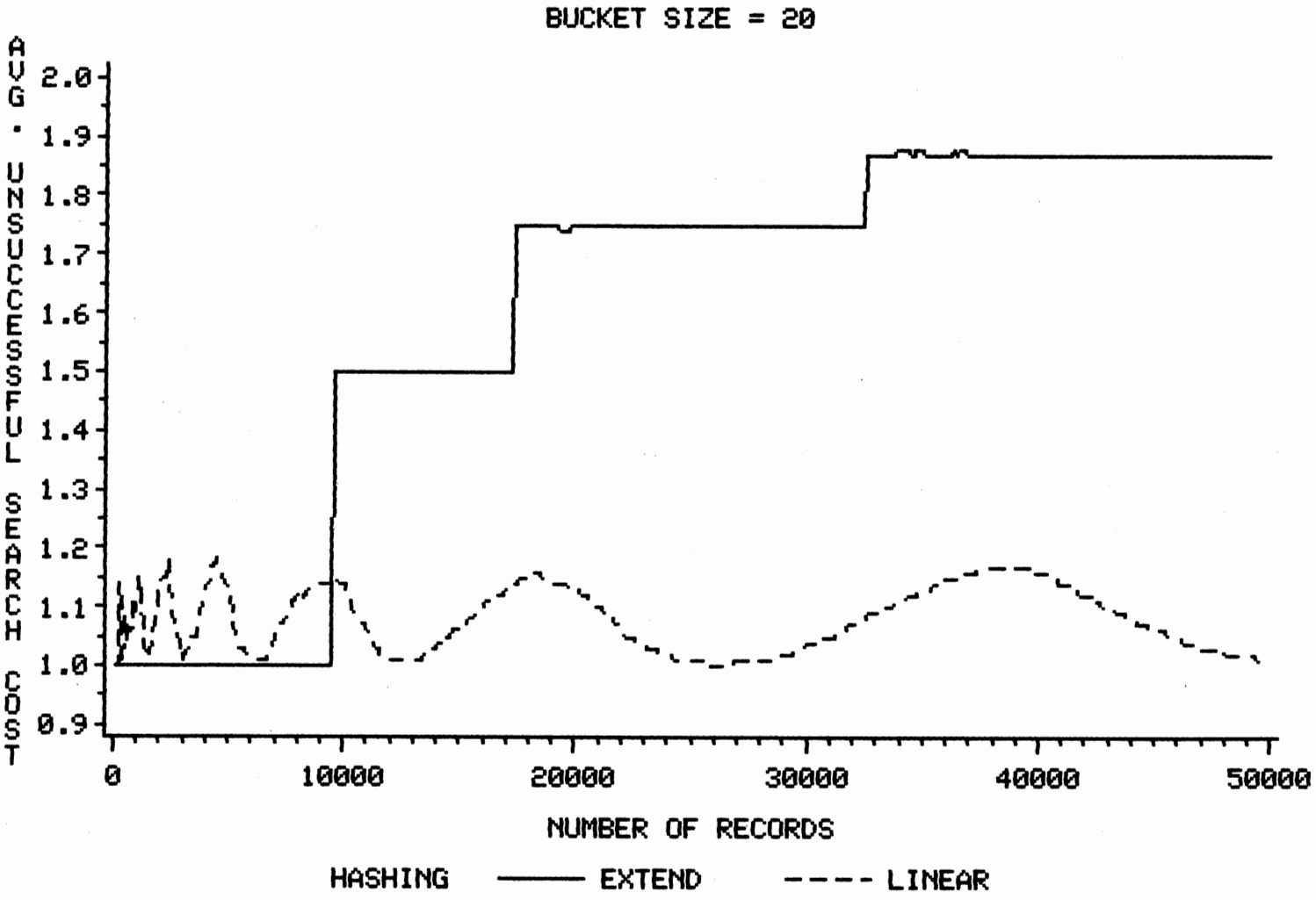


FIGURE 22. UNSUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

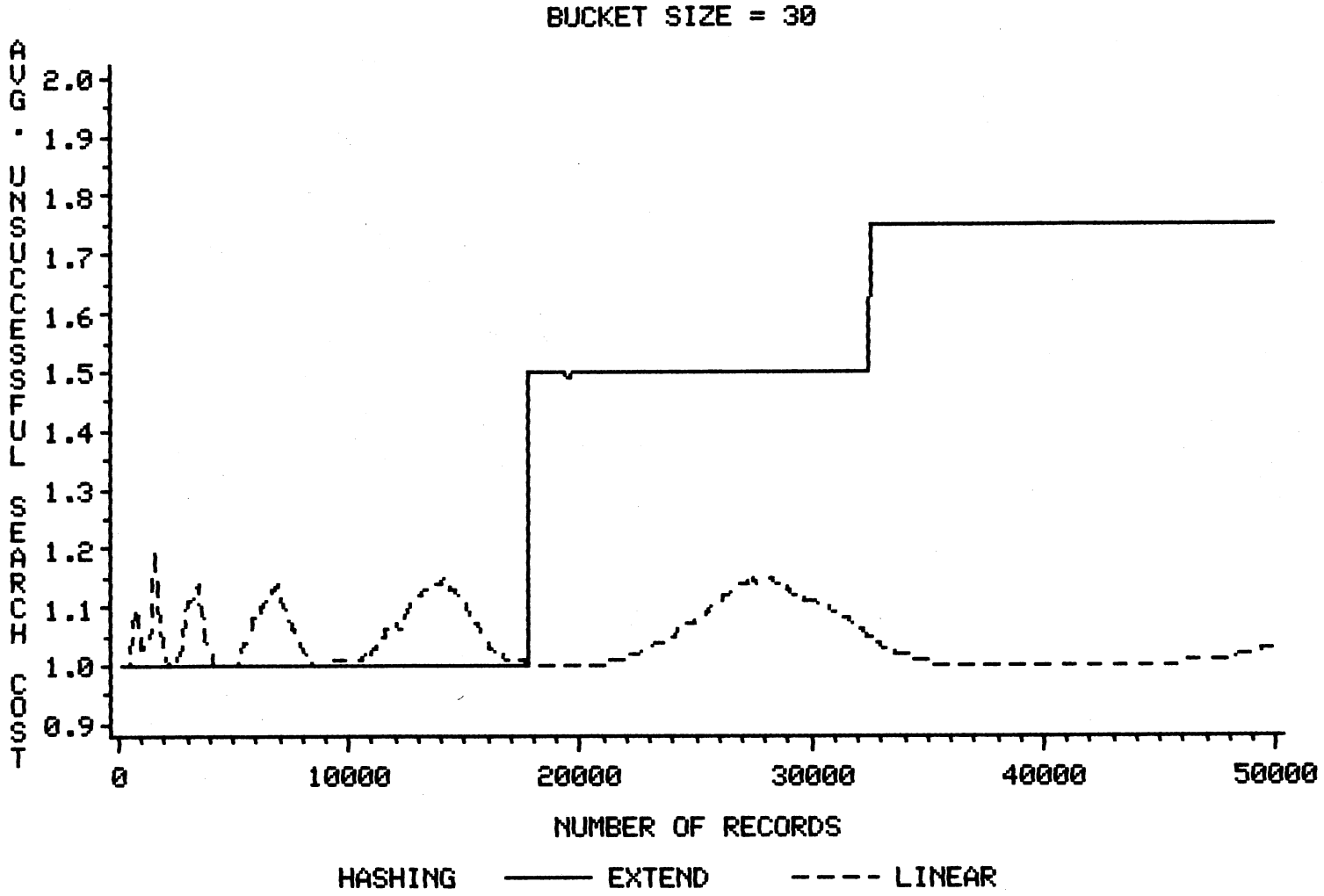


FIGURE 23. UNSUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

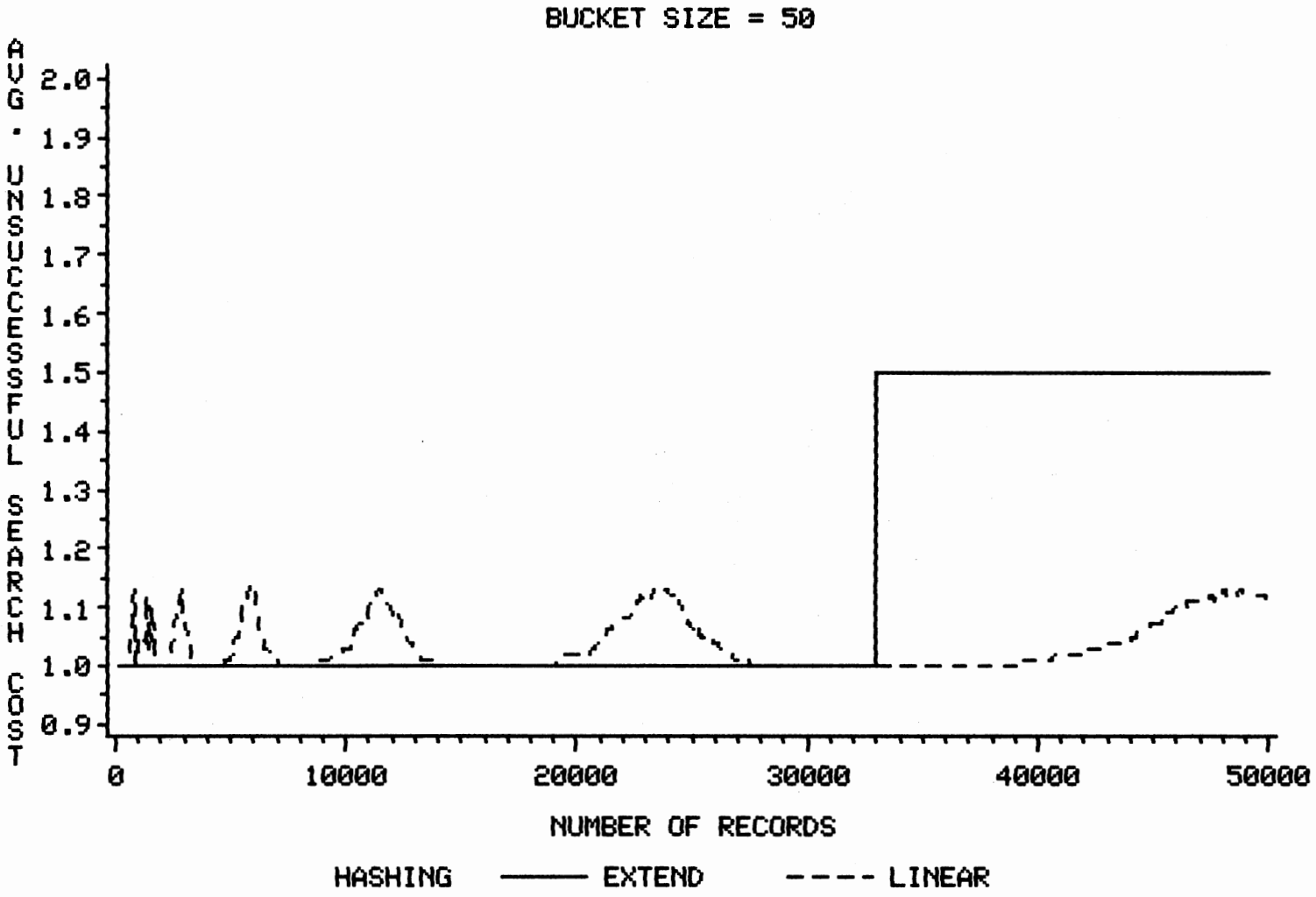


FIGURE 24. UNSUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

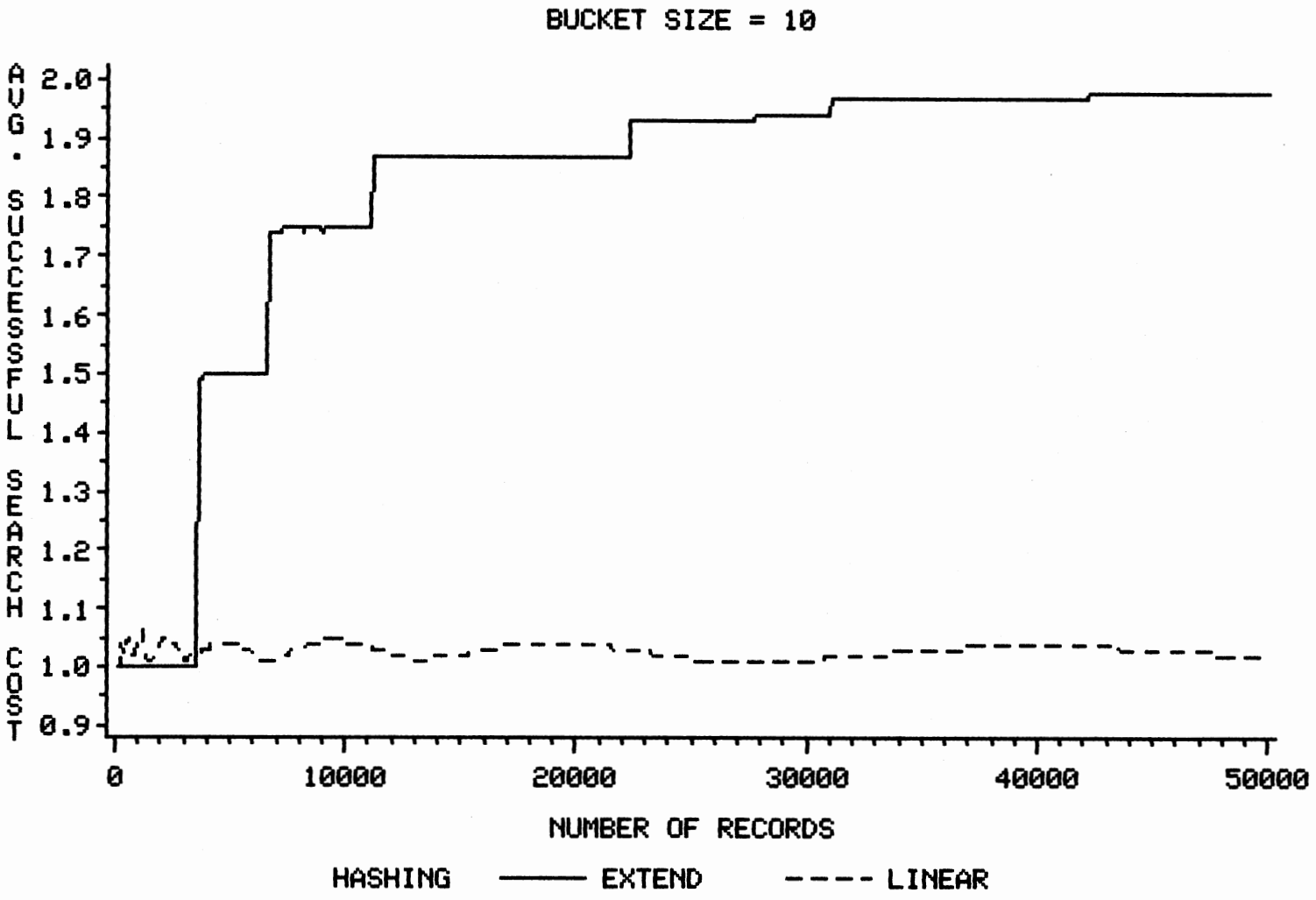


FIGURE 25. SUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

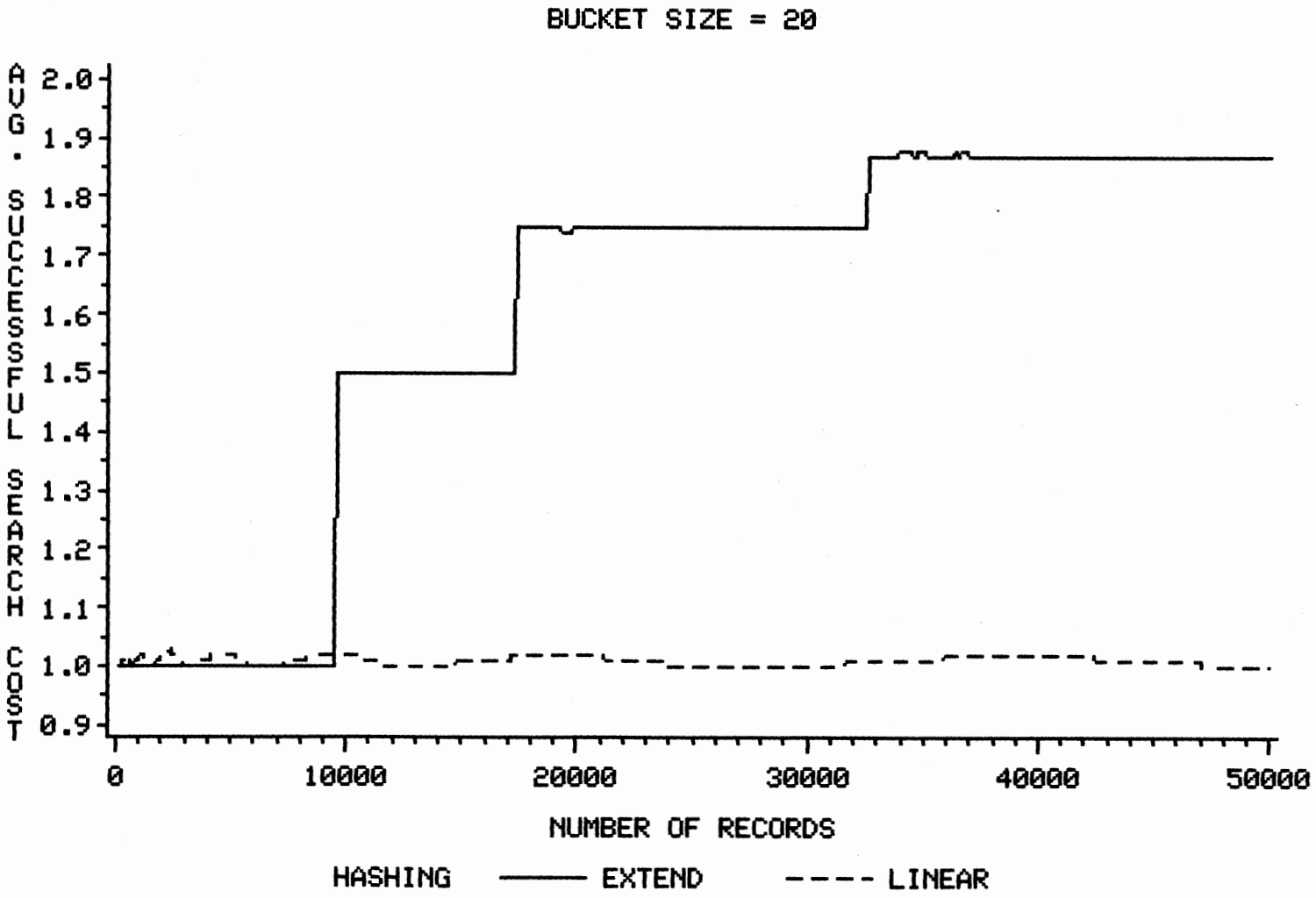


FIGURE 26. SUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

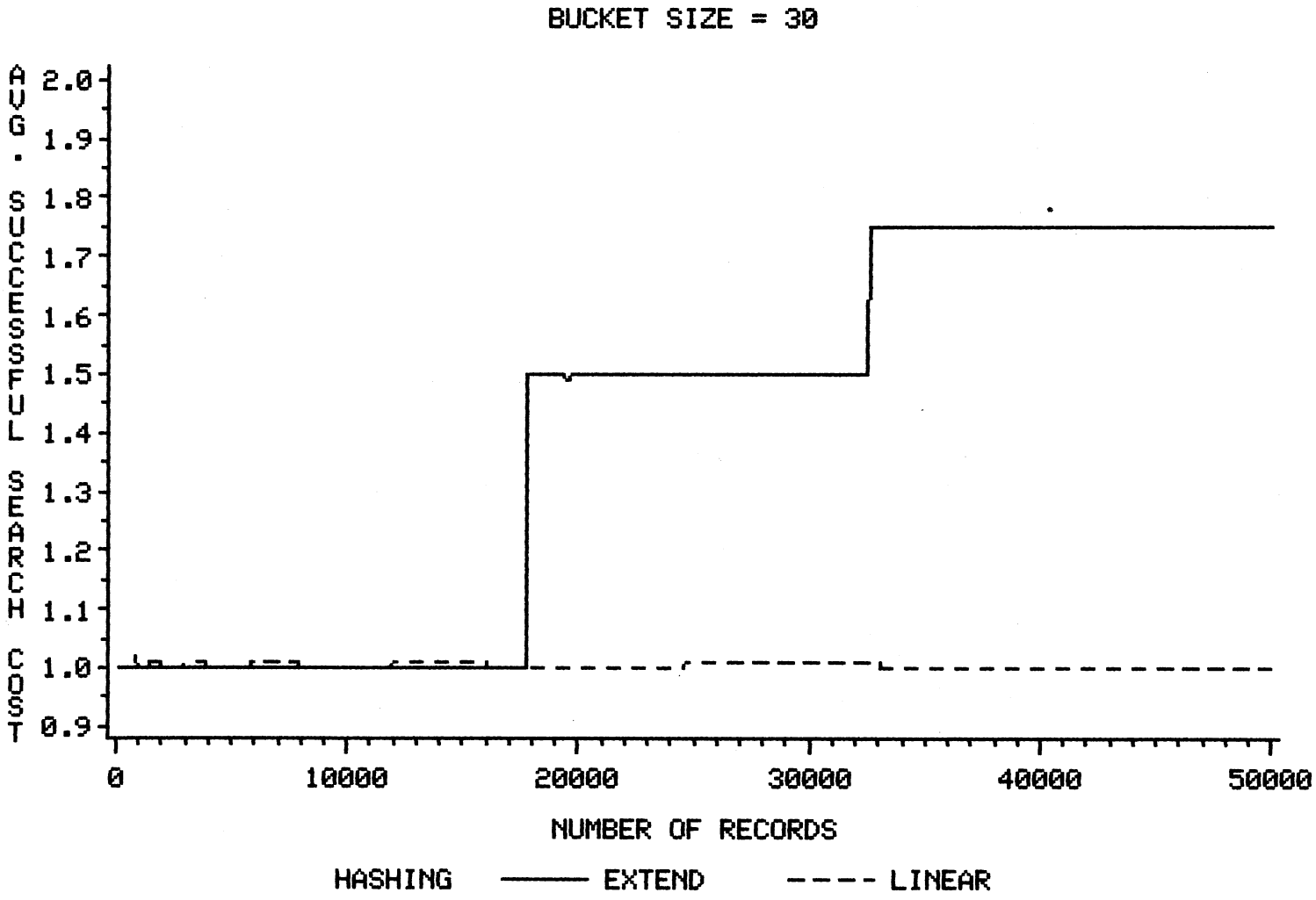


FIGURE 27. SUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

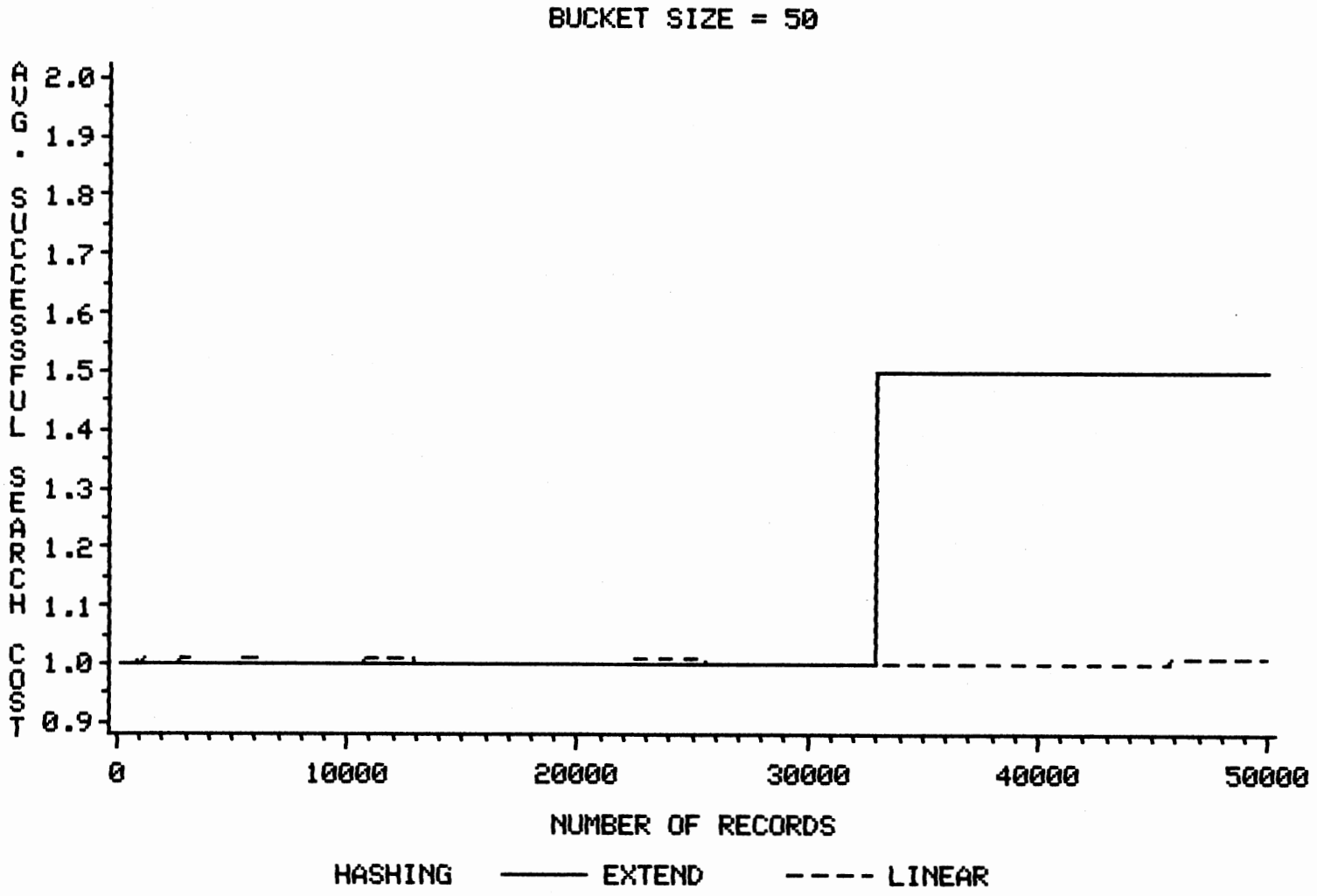


FIGURE 28. SUCCESSFUL SEARCH COST VS. NUMBER OF RECORDS

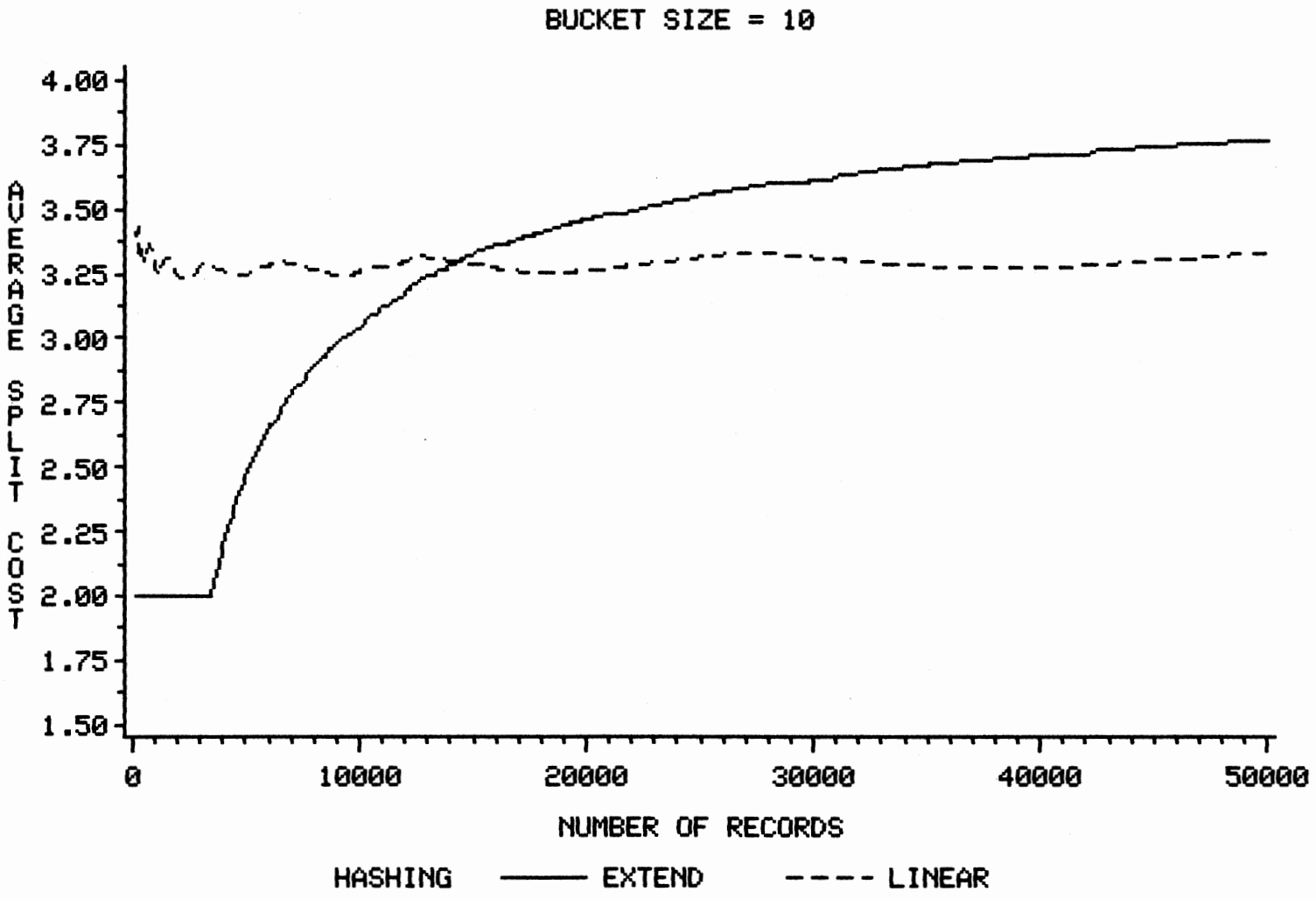


FIGURE 29. SPLIT COST VS. NUMBER OF RECORDS

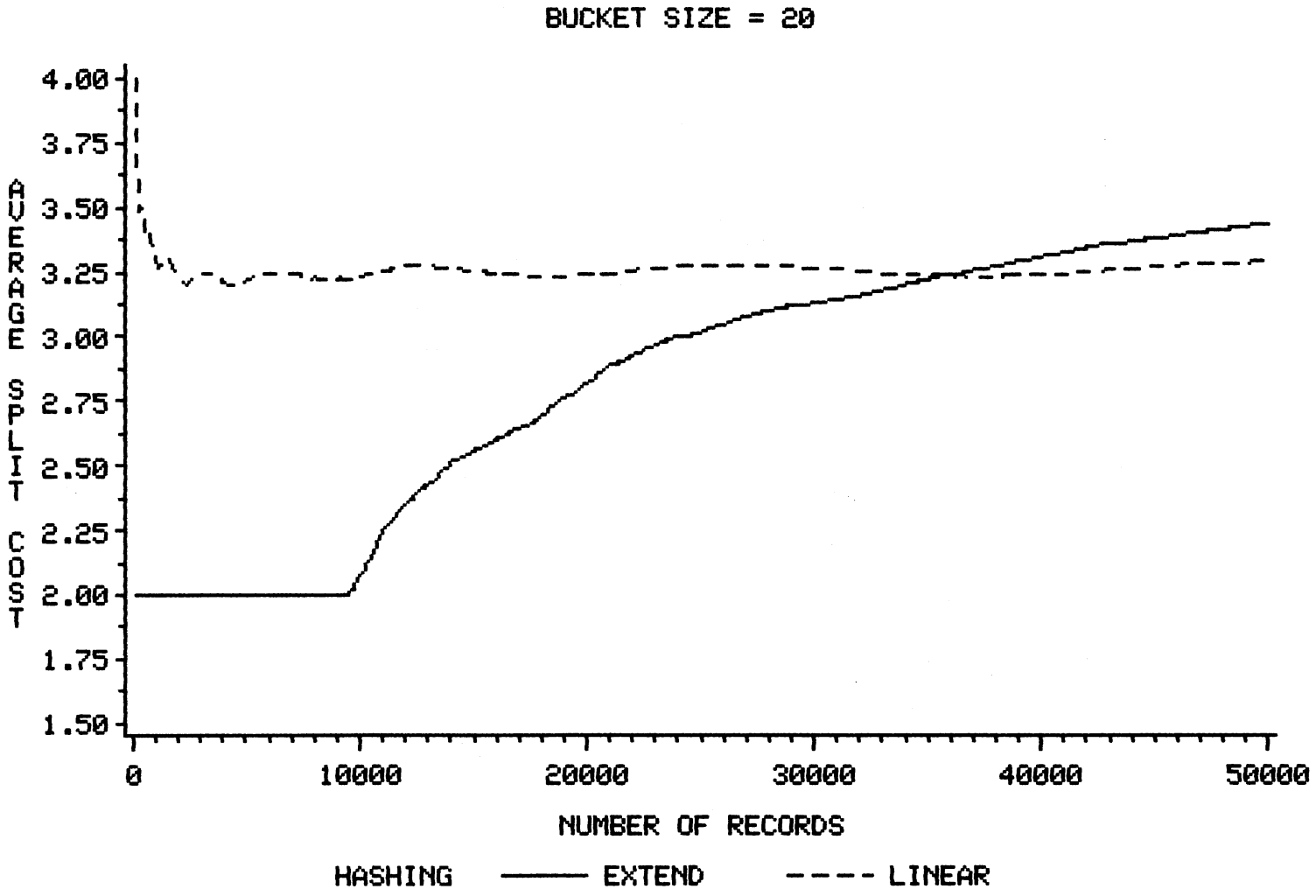


FIGURE 30. SPLIT COST VS. NUMBER OF RECORDS

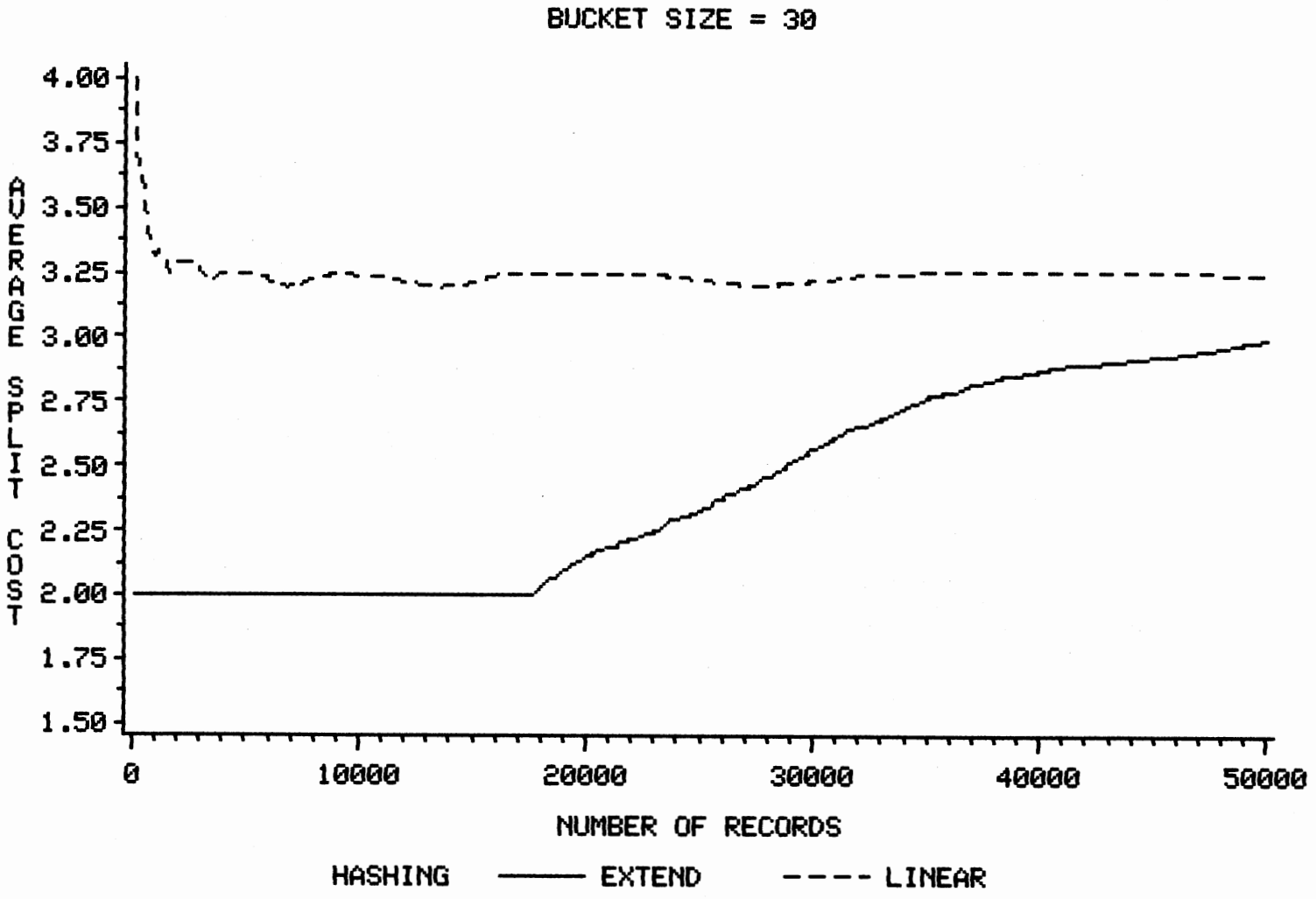


FIGURE 31. SPLIT COST VS. NUMBER OF RECORDS

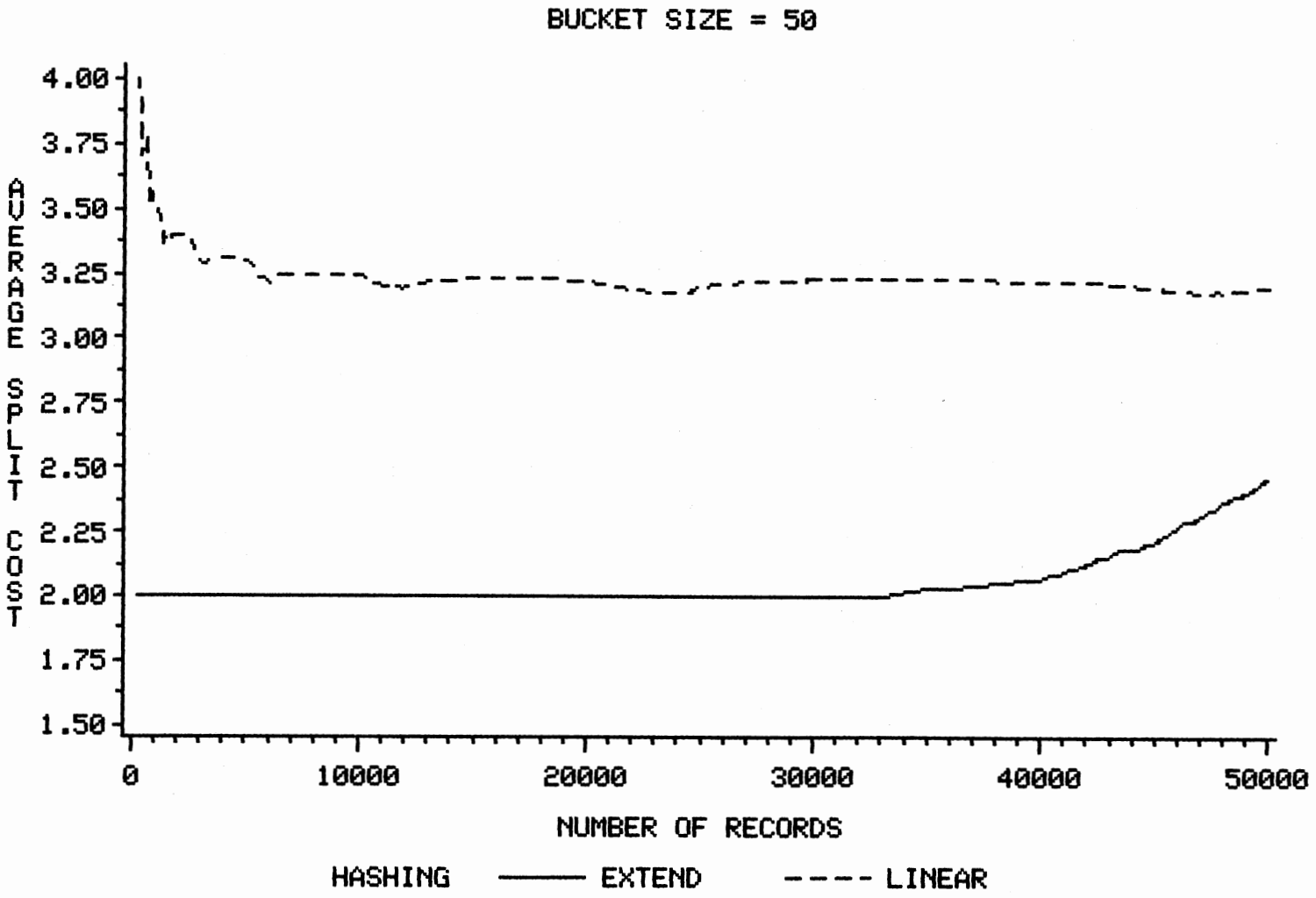


FIGURE 32. SPLIT COST VS. NUMBER OF RECORDS

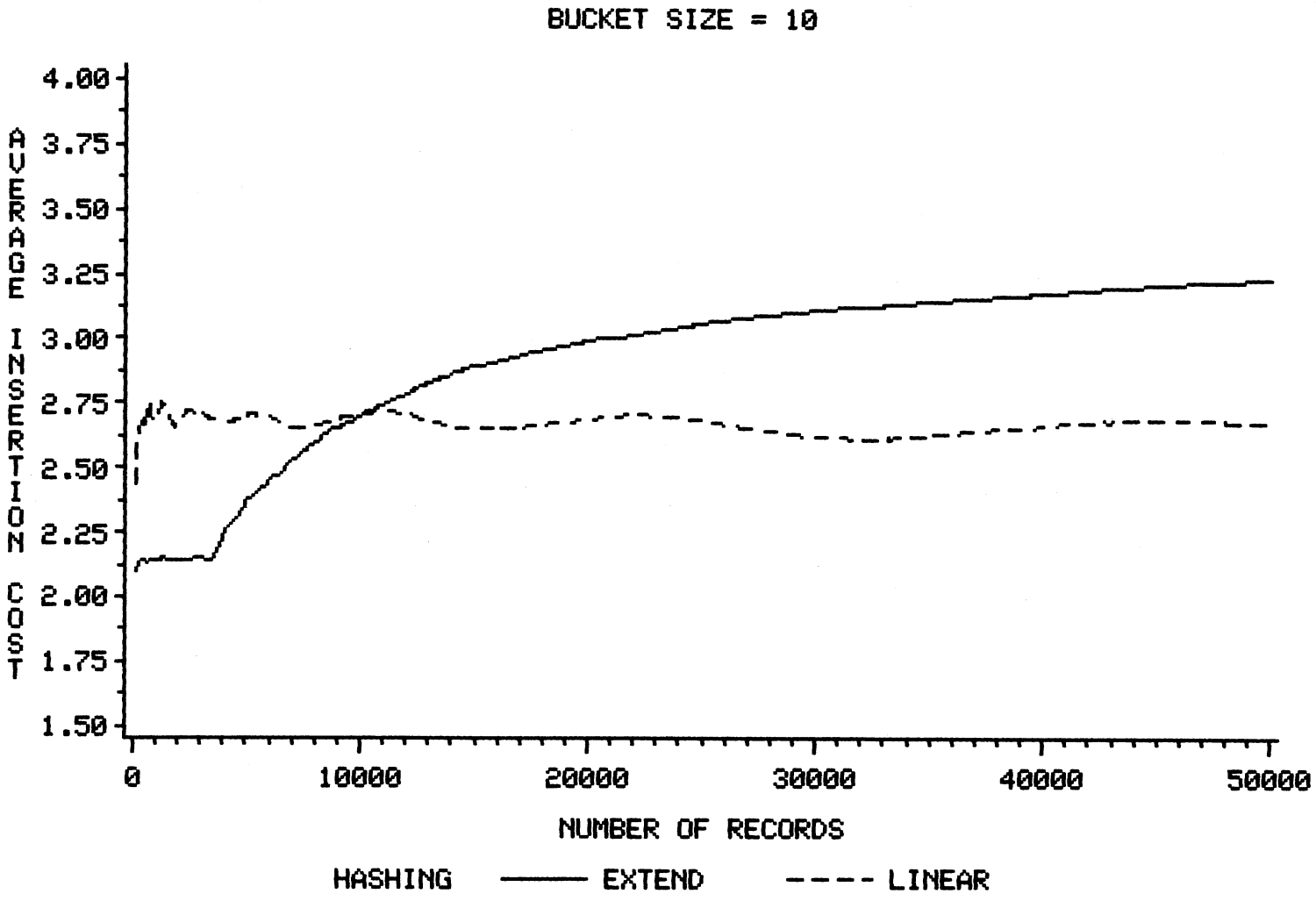


FIGURE 33. INSERTION COST VS. NUMBER OF RECORDS

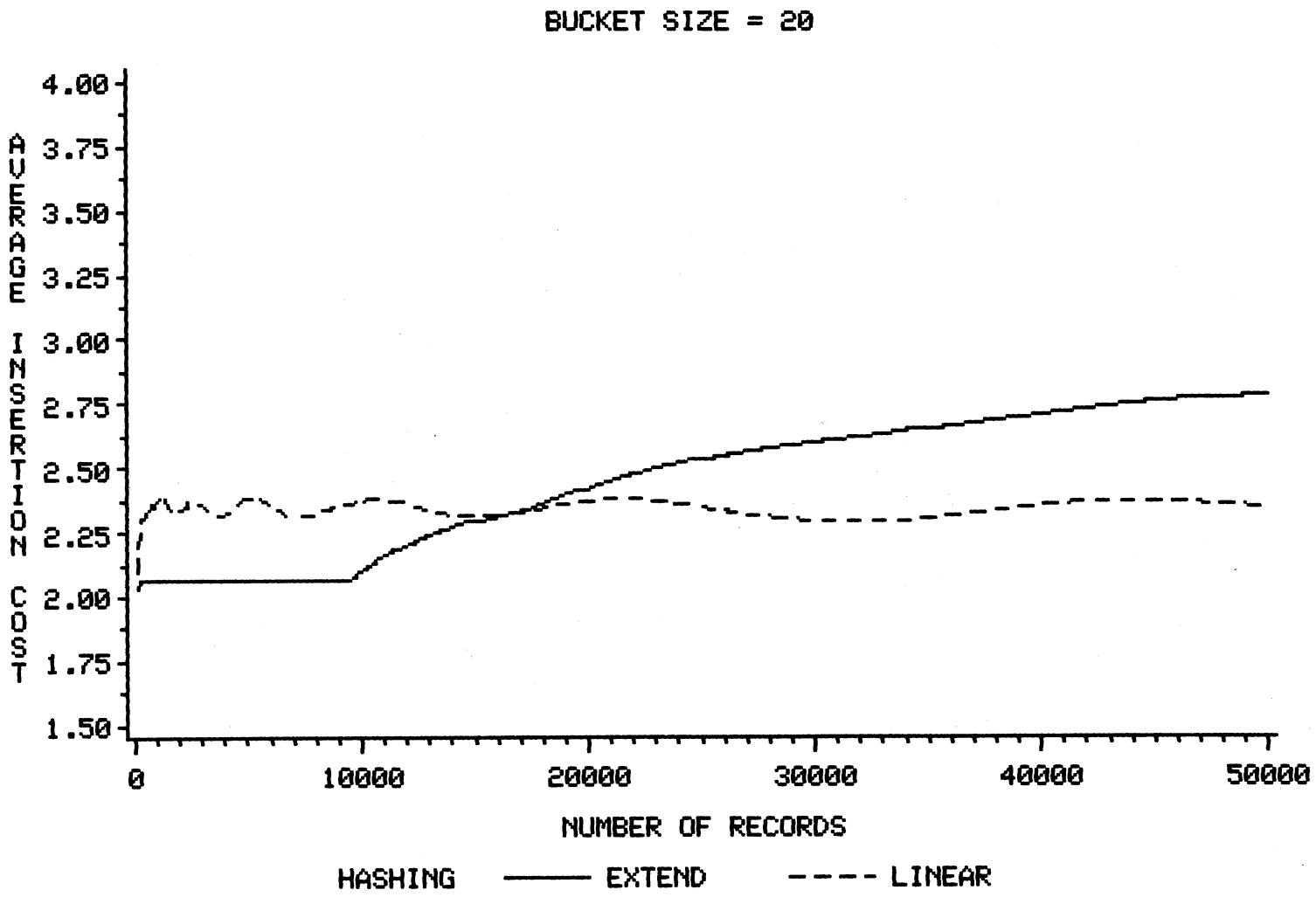


FIGURE 34. INSERTION COST VS. NUMBER OF RECORDS

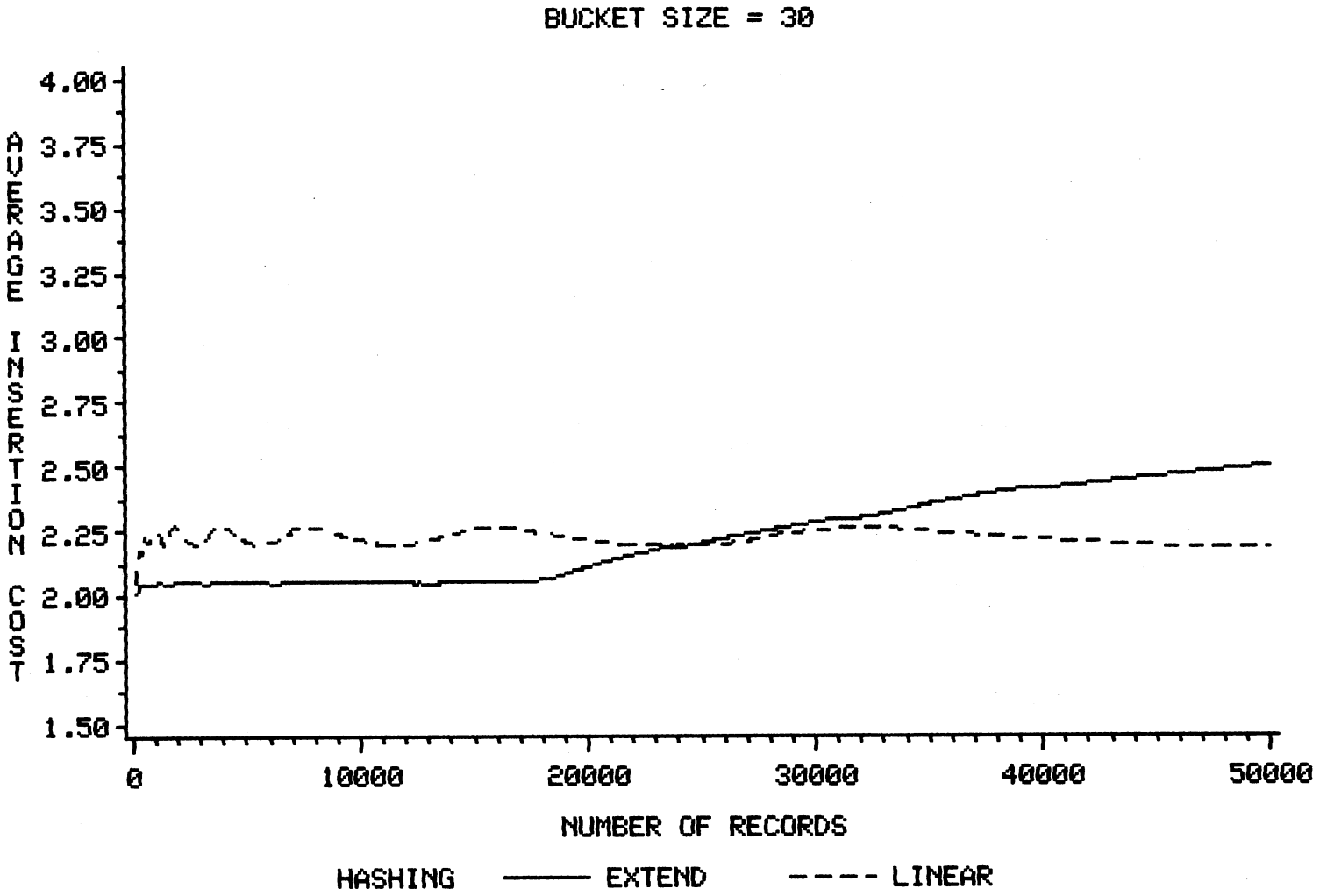


FIGURE 35. INSERTION COST VS. NUMBER OF RECORDS

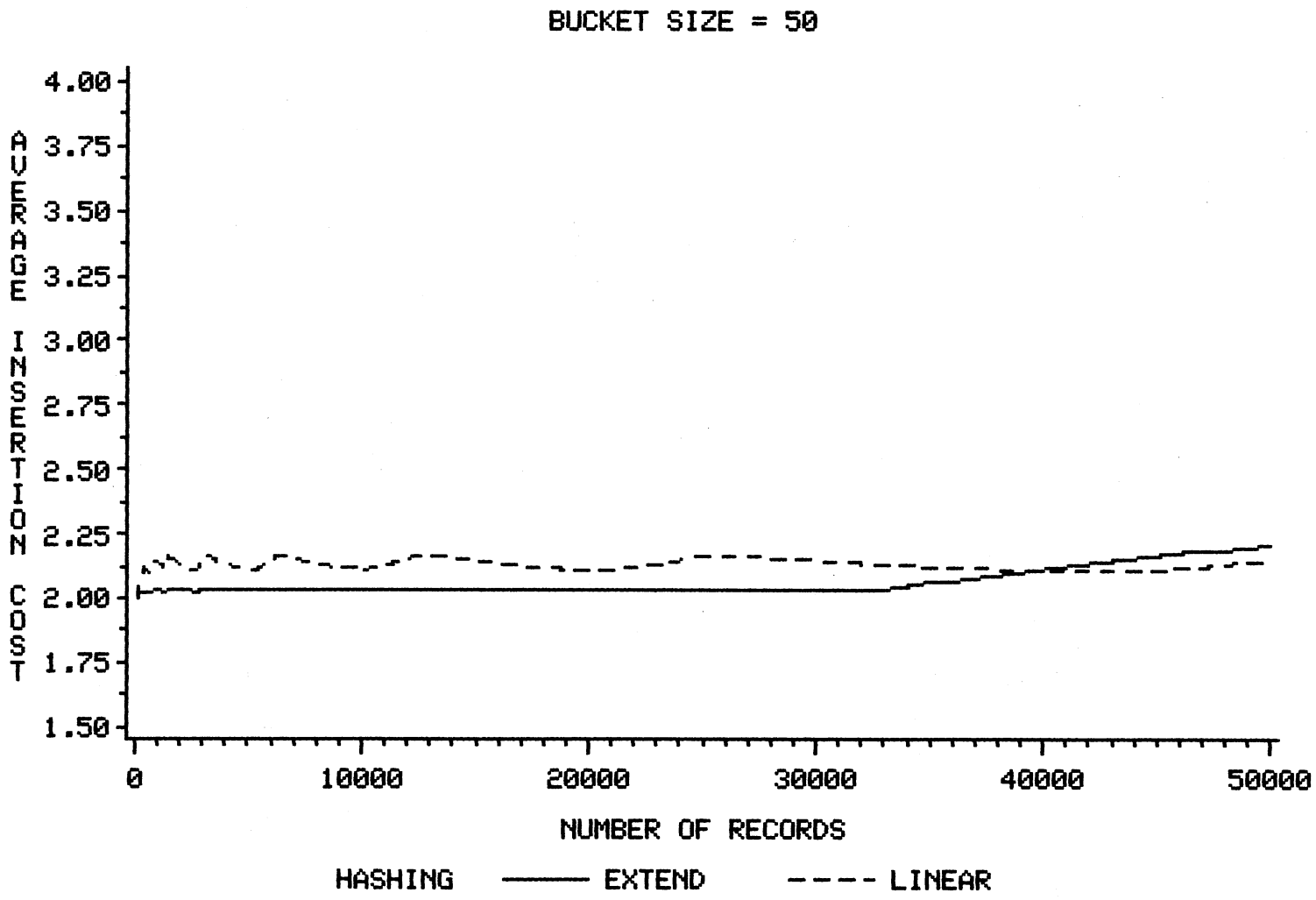


FIGURE 36. INSERTION COST VS. NUMBER OF RECORDS

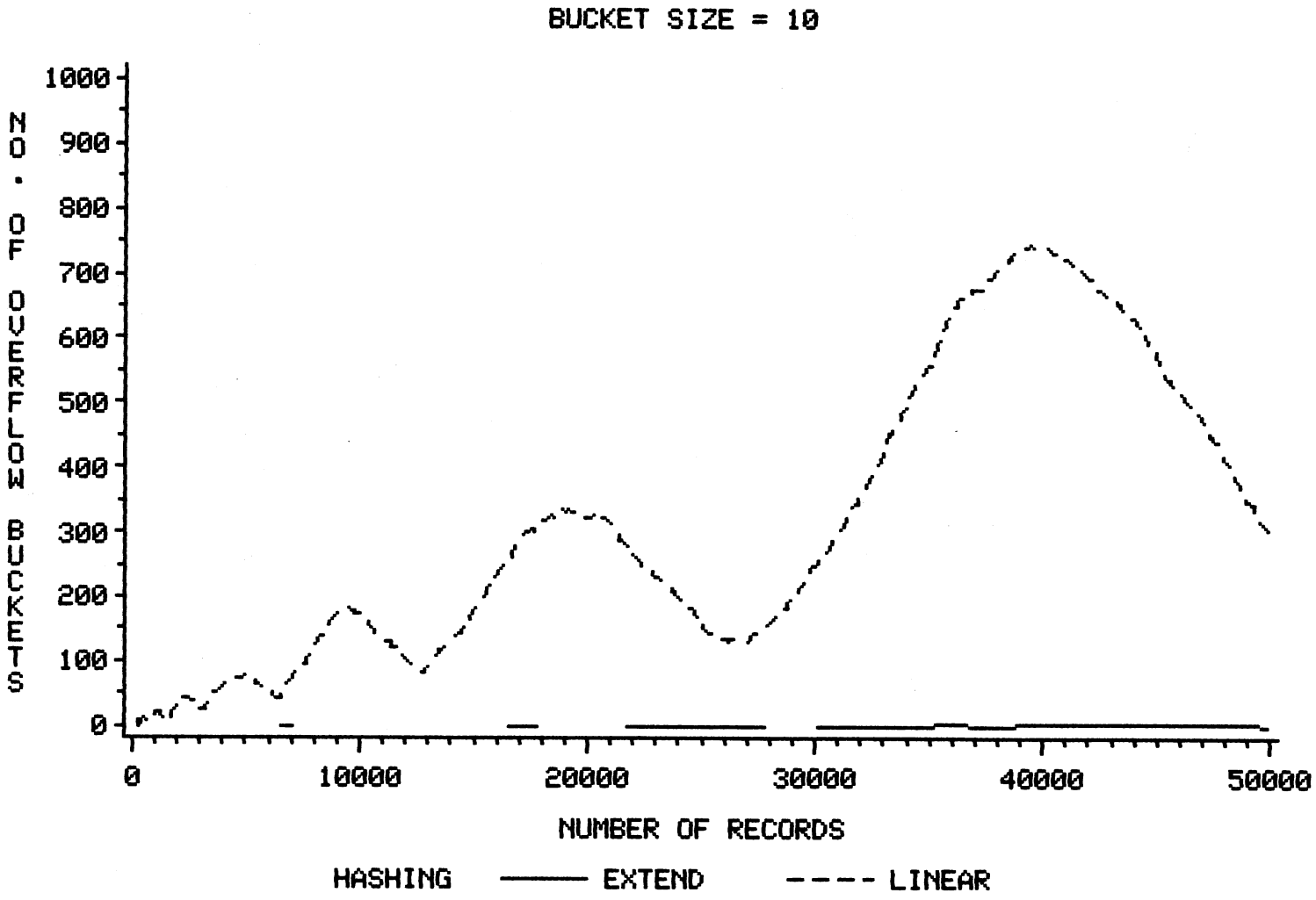


FIGURE 37. OVERFLOW BUCKETS VS. NUMBER OF RECORDS

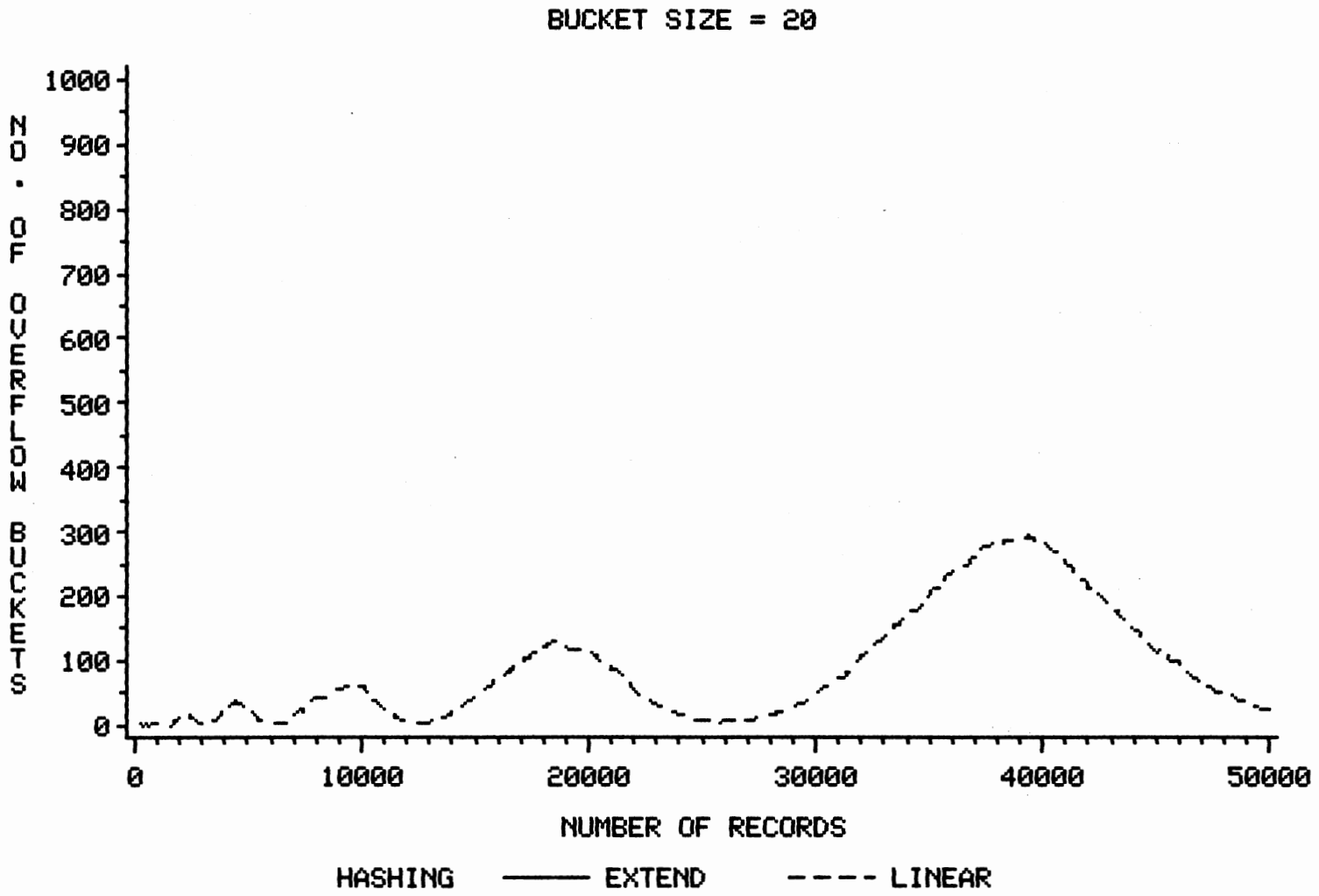


FIGURE 38. OVERFLOW BUCKETS VS. NUMBER OF RECORDS

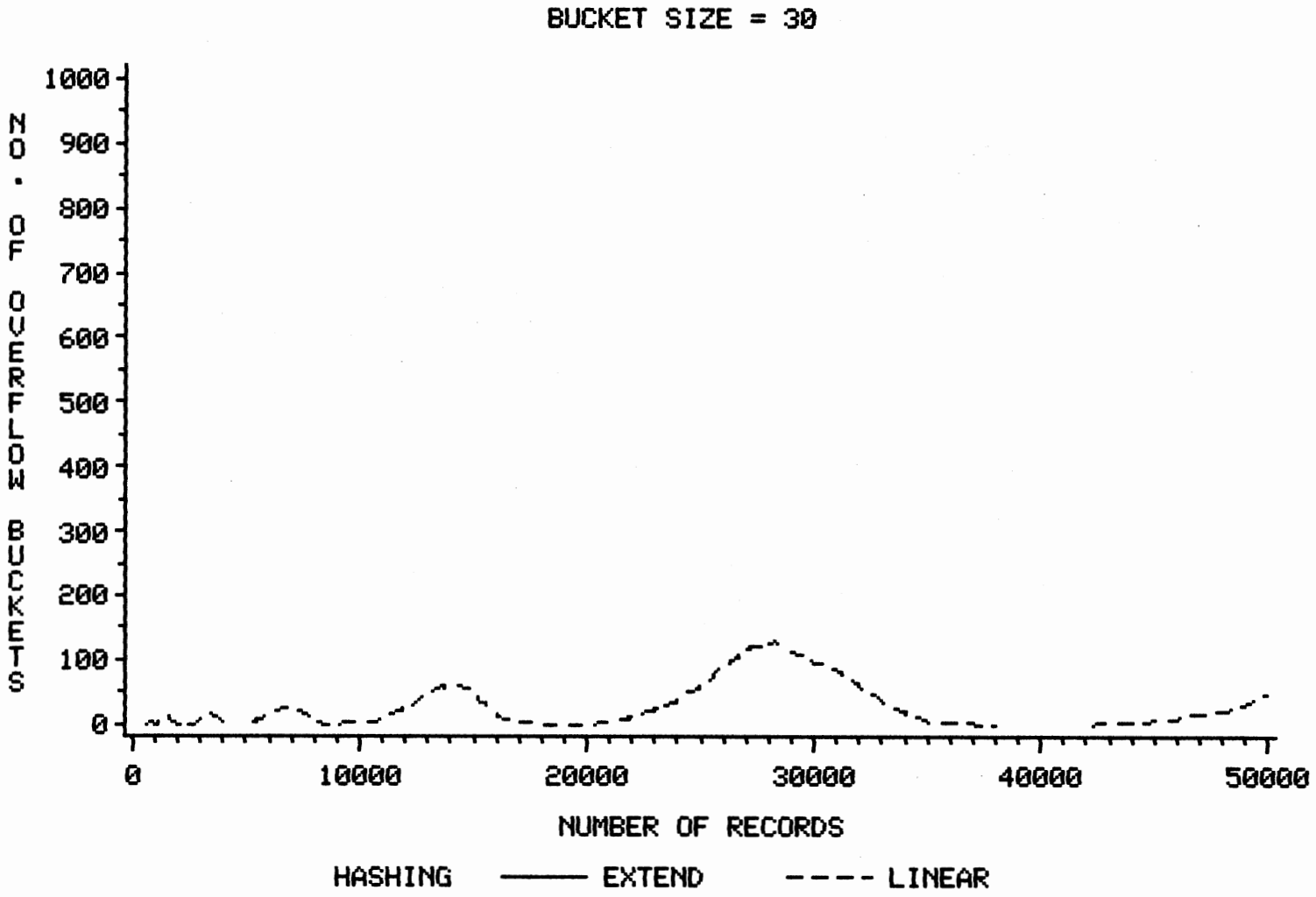


FIGURE 39. OVERFLOW BUCKETS VS. NUMBER OF RECORDS

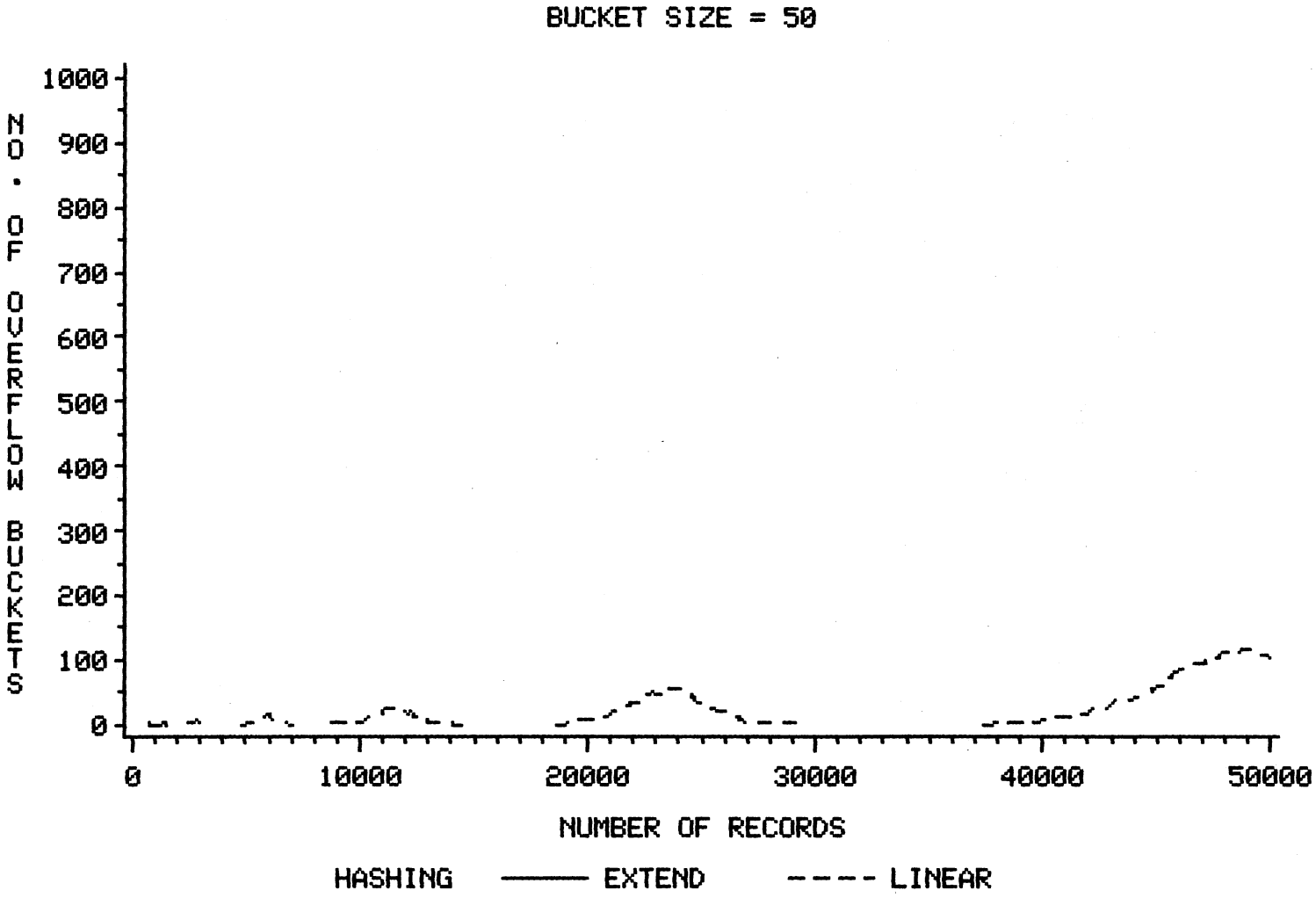


FIGURE 40. OVERFLOW BUCKETS VS. NUMBER OF RECORDS

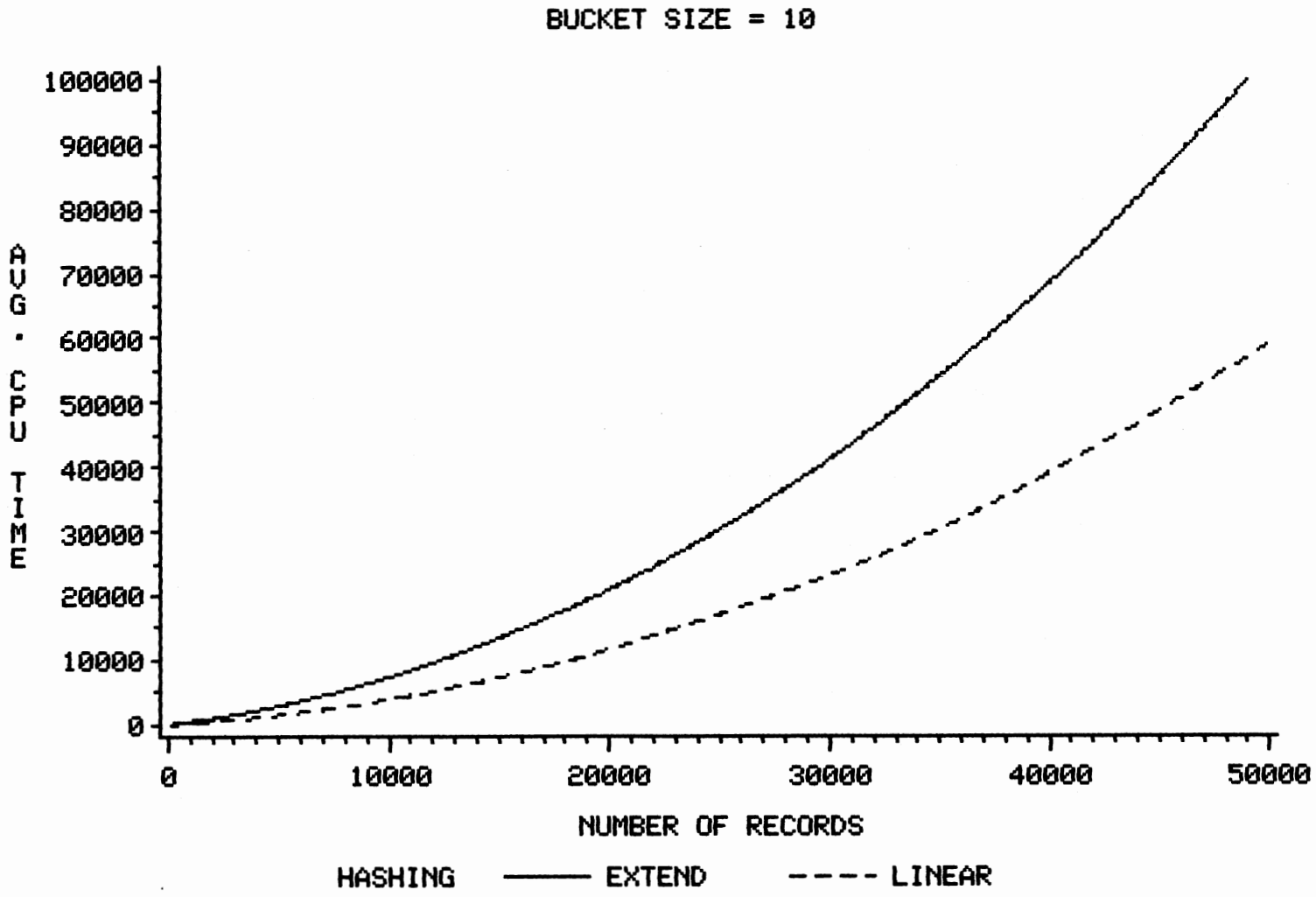


FIGURE 41. CPU TIME VS. NUMBER OF RECORDS

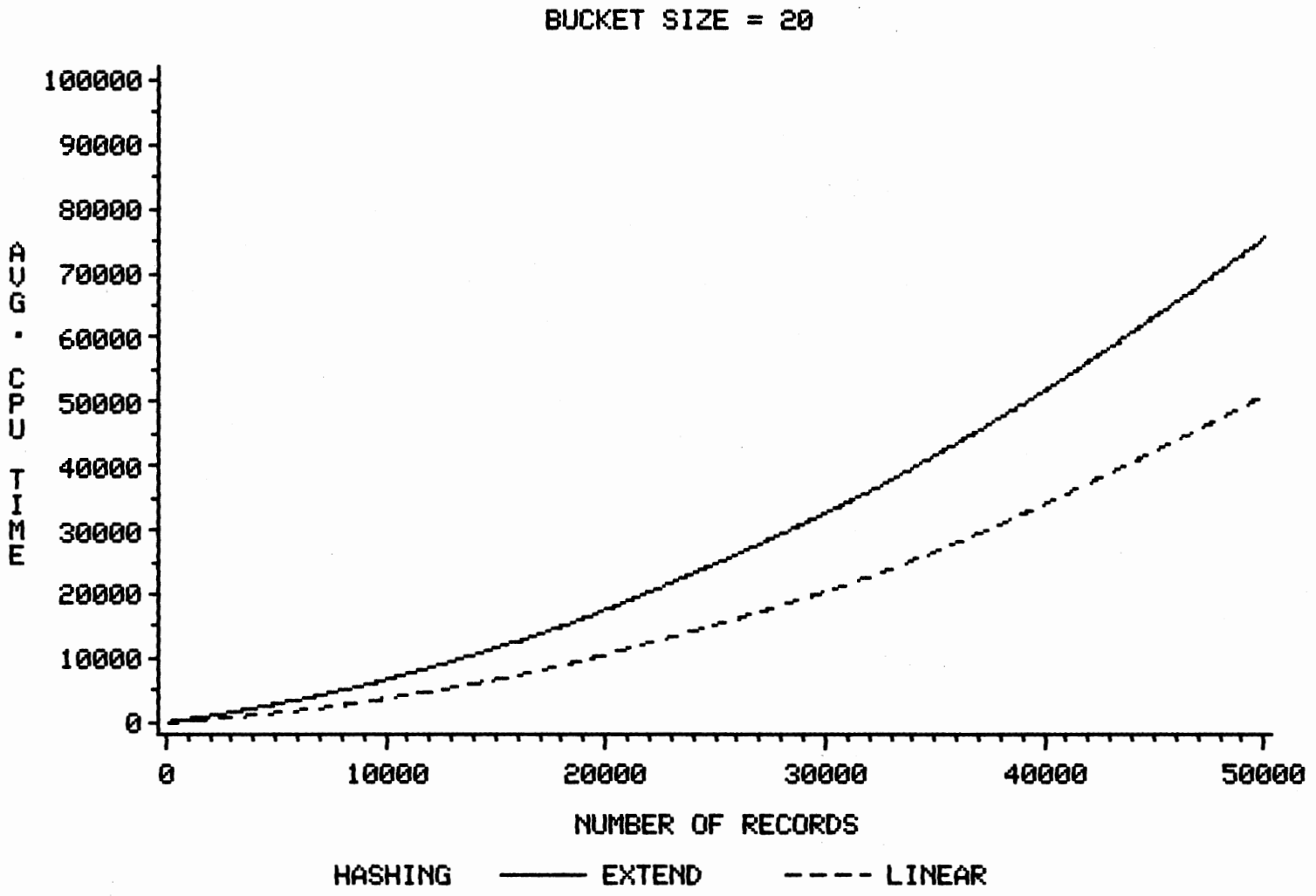


FIGURE 42. CPU TIME VS. NUMBER OF RECORDS

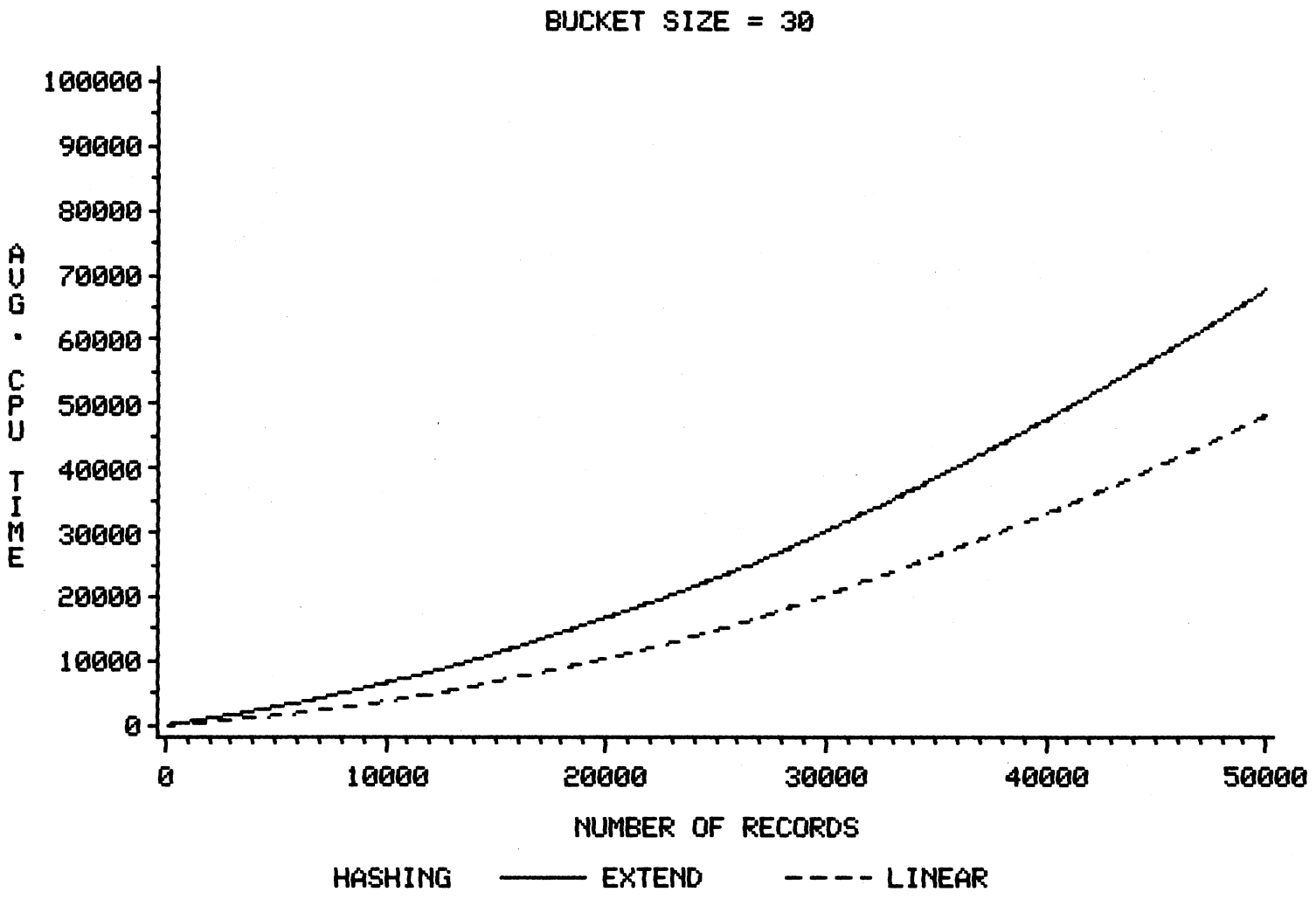


FIGURE 43. CPU TIME VS. NUMBER OF RECORDS

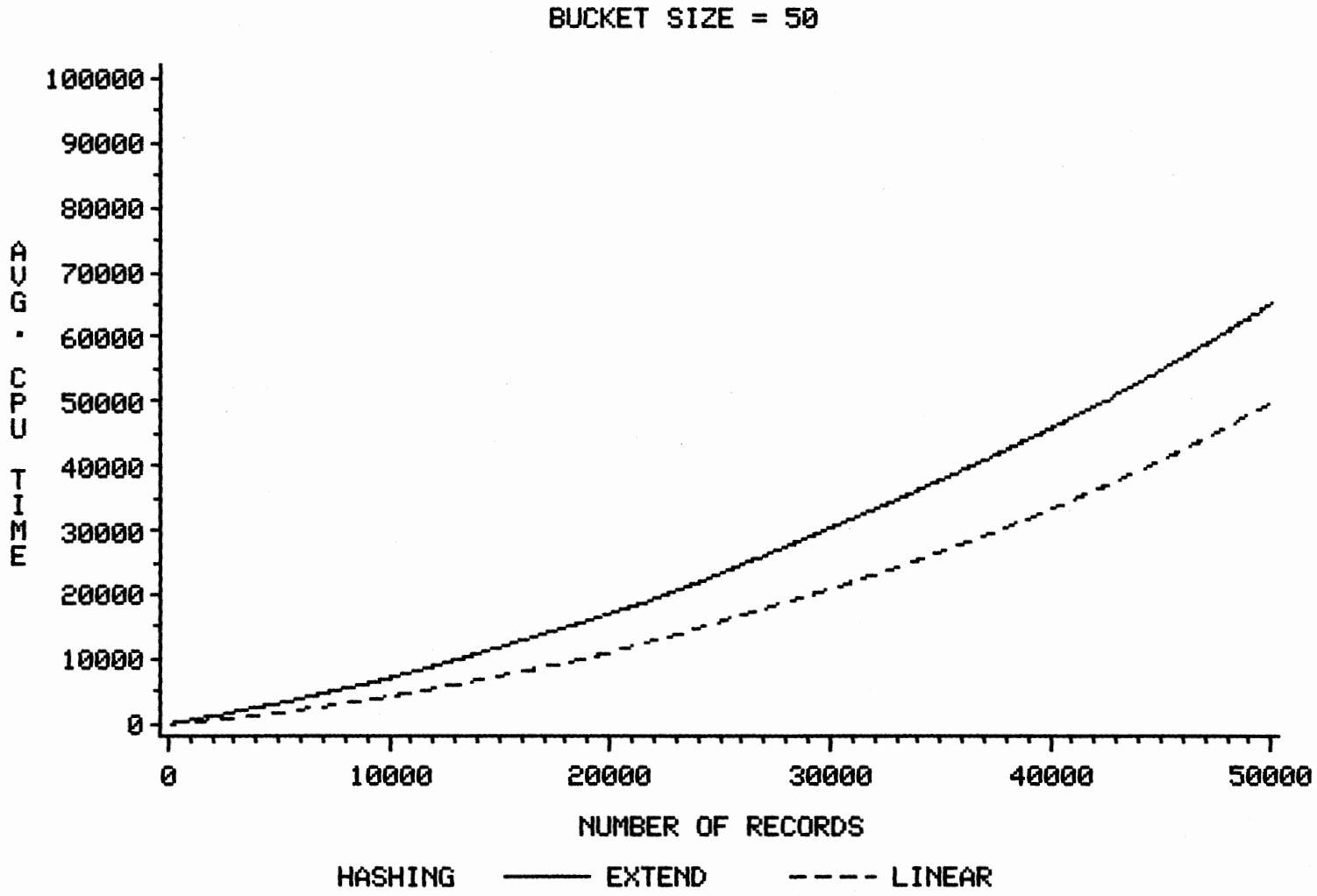


FIGURE 44. CPU TIME VS. NUMBER OF RECORDS

VITA[2]

Ashok K. Rathi

Candidate for the Degree of

Master of Science

Thesis: PERFORMANCE COMPARISON OF LINEAR HASHING AND EXTENDIBLE HASHING

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Ajmer, India, December 4, 1962, the son of Mr. and Mrs. C.K Rathi. Single.

Education: Graduated from O.J.H.S. School, Ajmer, India, in July, 1980; received Bachelor of Commerce degree in Accounting and Business Statistics from the University of Rajasthan, India, in July, 1982; received Chartered Accountancy (Intermediate) certificate from the Indian Institute of Chartered Accountancy in July, 1984; received Master of Business Administration from Oklahoma State University, Stillwater, Oklahoma, USA in May, 1987; completed the requirements for Master of Science degree at Oklahoma State University in May, 1989.