A STUDY OF DYNAMIC HASHING AND DYNAMIC HASHING WITH DEFERRED SPLITTING

By

HU CHANG

Bachelor of Science

Fu-Jen Catholic University

Taipei, Taiwan

1981

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
December, 1985

Thesis
1985
L456 5s
cop.2

A STUDY OF DYNAMIC HASHING AND DYNAMIC HASHING WITH DEFERRED SPLITTING

Thesis Approved:

Michael J. Folk
Thesis Adviser

Donald D. Fisher

[illegible]

Norman N. Durham
Dean of the Graduate College

PREFACE

This thesis is a discussion and evaluation of both dynamic hashing and dynamic hashing with deferred splitting. The study includes a program design and implementation under the UNIX system. Comparisons and analyses are made using empirical results.

I would like to express sincere gratitude to my major advisor, Dr. Michael J. Folk for his guidance, motivation, and invaluable help. I am also thankful to Dr. Donald D. Fisher and Dr. Donald W. Grace , not only for serving on my graduate committee, but also for their encouragement during my stay at Oklahoma State University.

My wife, Beny, my father, Chien-Yeh Chang, my mother, Chien-Yun Wu, my sisters Rosa and Lily deserve my deepest appreciation for their continual support, moral encouragement, and understanding.

Kevin, my son, came to this world during my work on the thesis and gave me endless courage to finish it.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

For the past two decades, schemes for structuring large files have evolved from two areas that were initially considered as requiring distinct approaches: data structures for main memory, and access methods to slow, high-capacity secondary storage devices.

The first schemes used for structuring data were more appropriate to static than to dynamic data. "Static" means the extent and structure of the data remain unchanged during processing; only values may be changed. "Dynamic" means data elements may be inserted and deleted.

The array and the sequential file are the best known examples of static structures. Insertions and deletions lead to at least one of two undesirable results: the use of special routines (such as a flag to indicate that a record still in the structure should be considered as having been deleted), and frequent expensive restructuring of the entire file (especially when the number of holes left by deletions has grown so large as to degrade performance).

The evolution from static to dynamic data structures proceeded rapidly in those applications where data could be kept in main memory. List structures, invented to accommodate highly dynamic data, became popular during the 1950s [1]. The problem of possible degeneracy of list structures (e.g. when a dynamic tree degenerates into a linear list because of a biased sequence of insertions and deletions) was recognized. The height-balanced tree [2] was a pioneering step toward the development of data structures that adapt gracefully to repeated insertions and deletions.

The development of comparable dynamic file structures for secondary storage devices was slower. With the advent of disks, sequential files appropriate to tapes were quickly modified to indexed-sequential files [3] which permit access to any record, ideally in two steps. First a directory is searched, which points to the proper cylinder or track. Second, this track is searched sequentially. For static files this scheme is as fast as the hardware restrictions on disk access permit. For highly dynamic files indexed-sequential access can lead to poor performance because long linear chains of overflow buckets may be traversed. Balanced trees turned to be a good solution for storing highly dynamic files on disks. The B-tree [4] is the most effective file organization that permits gradual adaptation of structures to fit the data.

Data structures for main memory fall into three categories: linearly or sequentially accessible (in time O(n), where n is the number of items), accessible by tree structures (in time O(log(n))), and directly accessible by hashing [5,6,7,8,9,10] (in time O(1)). Hashing schemes have been adapted to dynamic files on secondary storage devices by the inefficient technique of chaining overflow buckets when needed. If an adaptable hashing scheme can be designed to remain in balance as buckets are inserted and deleted, the suitability of hashing for secondary storage devices would be greatly enhanced.

Dynamic hashing [11] is a file organization technique based on normal hashing. With dynamic hashing the file size can be increased and decreased dynamically without reorganizing the whole file and with no overflow records. The allocated secondary storage is divided into buckets of size b. If a record is to be inserted into a bucket which is full, the bucket is split into two buckets of the same size b, and all the records in the full bucket together with the "overflow" record are distributed between the two buckets.

Dynamic hashing uses an index to the record file. A bucket is associated with the given record's key, and the bucket's location is identified by searching through the index. The size of the index grows and shrinks dynamically according to the number of records. Retrieval is fast if the bucket's location has been found. Since there are no overflow records, only one access to secondary storage is required if the index is small enough to be kept in main memory.

To improve space utilization further, dynamic hashing can be modified by allowing overflow records [12]. Splitting of a bucket is deferred until a certain number of overflow records have been inserted. Retrieval of a record may require more than one disk access. This modified dynamic hashing method provides a smaller index size and a higher space utilization.

The idea of this thesis is to implement dynamic hashing and dynamic hashing with deferred splitting on a UNIX system and compare performance by examining empirical results. Analysis will focus on number of disk accesses, space utilization, index size, and index path length.

Chapter II presents a description of these two file organization methods. Chapter III shows the basic logic design for different routines. Chapter IV illustrates empirical results and discussions. A summary and conclusions are included in Chapter V.

CHAPTER II

GENERAL DESCRIPTION

Dynamic Hashing

Dynamic hashing keeps an index in main memory. The index is organized as a forest of binary trees [13,14] which are closely related to binary tries. Figure 1 shows the binary trie that is formed when the eight keys are treated as 7 bits binary numbers. The keys are shown in octal notation.

Key	7 bit binary number
066	0110110
130	1011000
102	1000010
061	0110001
121	1010001
023	0010011
160	1110000
012	0001010

Figure 2 shows the related binary tree. Notice that the number of trie nodes in Figure 1 is equal to the number of internal nodes in Figure 2. Therefore, the number of nodes in a binary trie is equal to the number of internal nodes in a related binary tree. Knuth stated that if n distinct binary numbers are put into a binary trie as described, then the number of nodes of the tree is equal to the number of partitioning stages required if these numbers

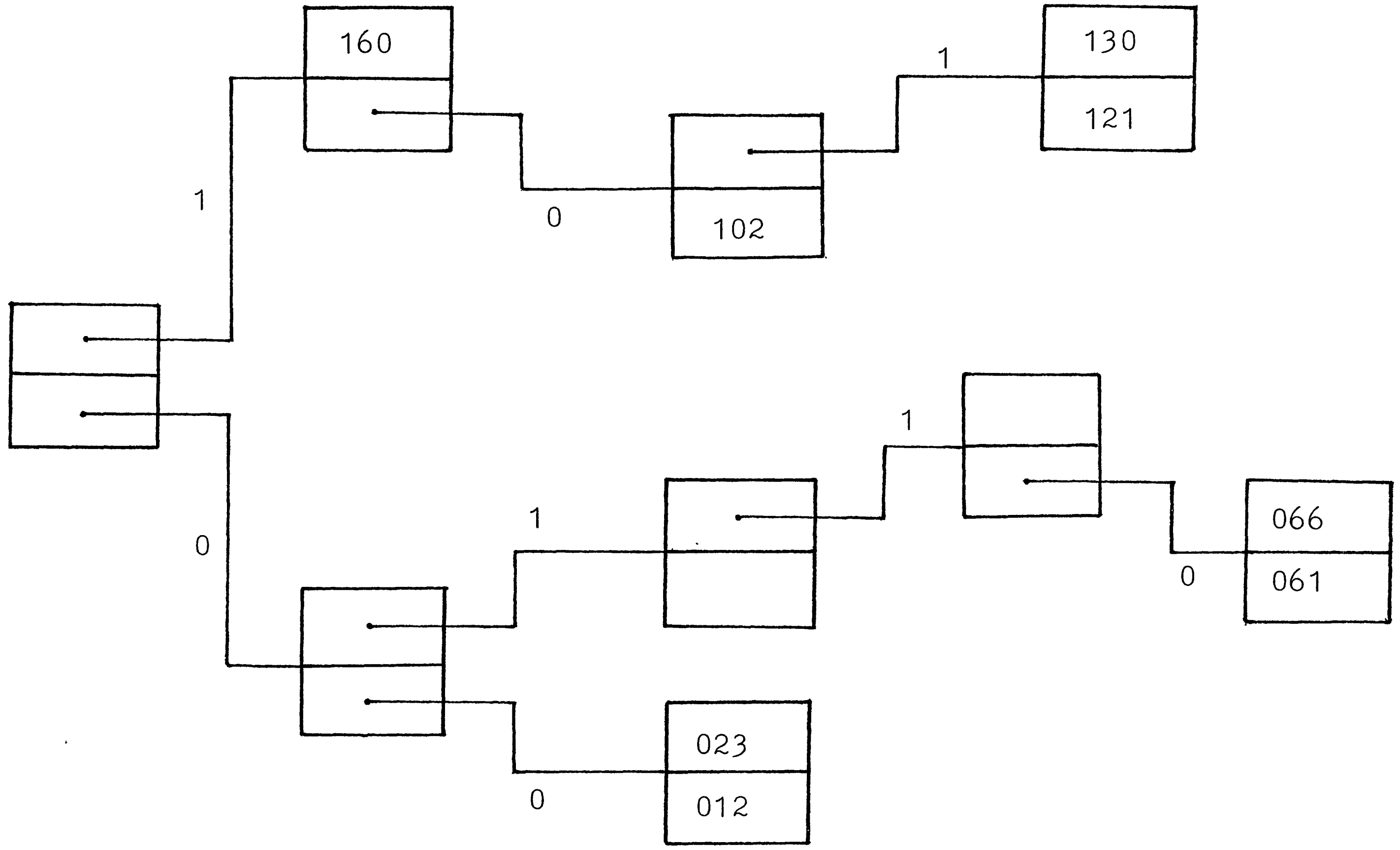


Figure 1. Binary Trie Containing 8 Keys

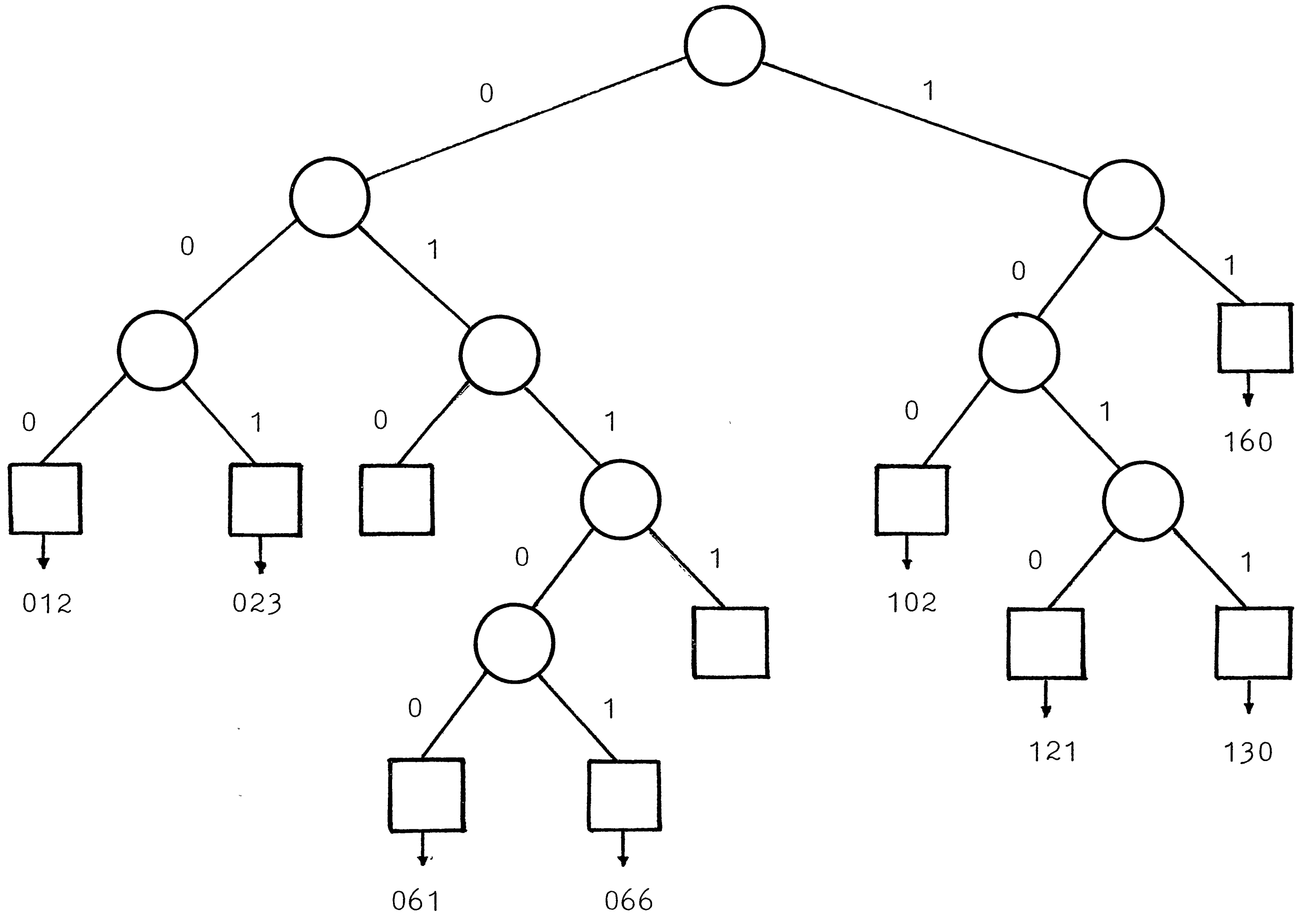


Figure 2. Binary Tree Related to the Binary Trie in Figure 1

are sorted by radix-exchange. Thus, if we assume that our keys are infinite-precision random uniformly distributed real numbers between 0 and 1, the number of trie nodes will be $(n/(\ln 2))+n*g(n)+O(1)$. Here $g(n)$ is a complicated function which may be neglected since its value is always less than $10**(-6)$. Also the number of nodes needed to store random keys in a binary trie, with the tree branching terminated for subfiles of s or fewer keys, is approximately $n/(s*\ln 2)$ [10]. Figure 3 shows the structure of the internal and external nodes of the binary trees.

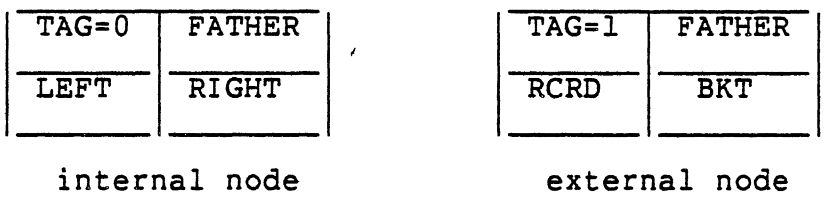


Figure 3. Data Structure of Index Node

TAG is a flag indicating whether a node is internal or external. FATHER is a pointer to the father of the current node; if FATHER is null, then the node is a root node. LEFT and RIGHT are pointers to the left and right sons of the current node. BKT is a pointer to the bucket on secondary storage. RCRD is the number of records in the bucket.

Besides the index, a data file on secondary storage is also employed by dynamic hashing and is organized as a variable number of buckets of fixed size. Here the word "variable" means that the file size is not fixed, it will change dynamically according to the number of records stored in the file.

Let m denote the number of binary trees in the forest. Then at the beginning, m root nodes (FATHER=NULL) are initialized. These root nodes are currently external nodes (TAG=1) and each one contains a pointer (BKT) which points to a bucket on secondary storage, and there are no records stored in the file (RCRD=0). After initialization, m nodes are allocated in main memory and m buckets allocated on secondary storage. Figure 4 shows the initial situation with m = 2.

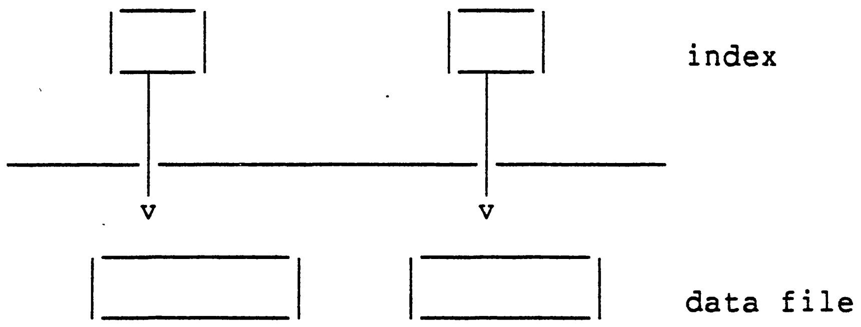


Figure 4. Initial File Structures

Denote the set of keys by $K(i)$, $1 <= i <= n$ where n, the number of keys, may change with time. A normal hashing function is needed to map the set of keys $K(i)$ into the set $\{1,2,3,...,m\}$ where $m <= n$. It defines an entry point in the index.

Before operations can be performed by dynamic hashing, pseudo random function should be introduced. It is designed to generate 0 or 1 with probability of 0.5 when called. The binary sequence generated by this function should be uniquely determined by the seed.

Search

The structure of the index and use of the random function make searching a straightforward procedure:

1. Hash the key to locate an entry in the forest of binary trees.
2. scan down the tree by using the random function with the seed being the key until an external node is reached.
3. Follow the pointer to locate the bucket, and bring the bucket into main memory and search for the key. If the key is in the bucket, the search ends successfully, otherwise the search fails.

The binary sequence generated by the random function constitutes an unique path in the binary tree which guarantees only one disk access to search for a key, either successfully or unsuccessfully.

Insertion

Insertion involves searching for the bucket corresponding to the key. If the bucket is found and is not full, the key is inserted. If the bucket is full, a split is performed, so that the keys in the full bucket together with the "overflow" key are distributed between the two buckets and the index is adjusted accordingly.

One result of bucket splitting is that the internal search path for all corresponding keys is increased by at least one level. Normally the result of bucket splitting will increase the path length by one level. For example, consider the tree in Figure 5 with a bucket size of 4. If

Key	Search Path
A	01
B	01
C	01
D	01

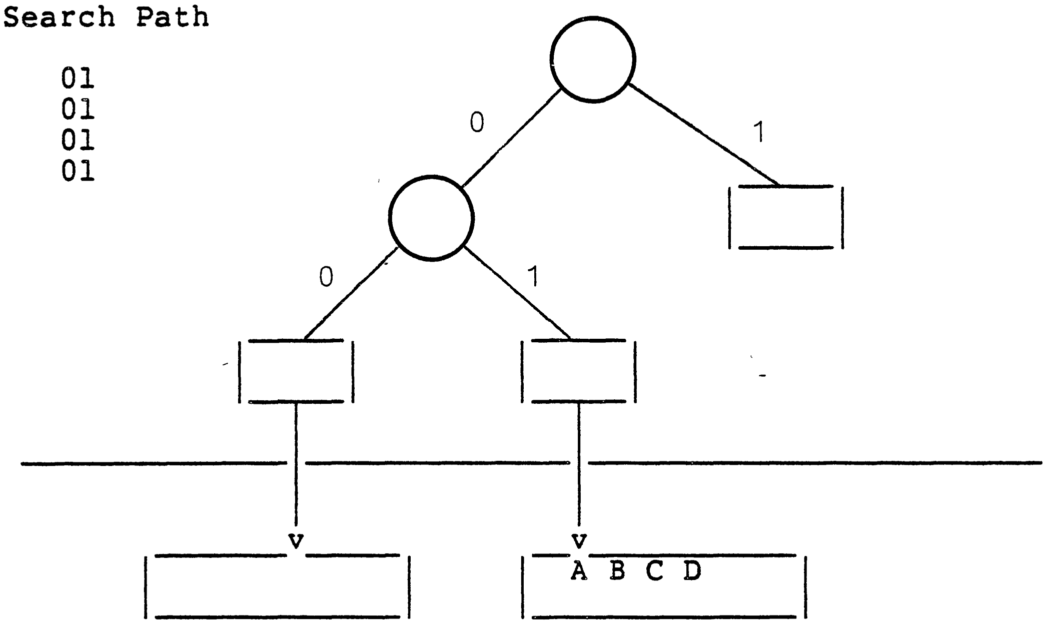


Figure 5. File Structure of Dynamic Hashing

the key E is added with search path 01, it becomes necessary to split bucket 2. The next bit in the search path must be computed by random function for A, B, C, D, and E. Suppose, e.g., the updated paths are the following:

Key	Search Path
A	010
B	010
C	011
D	010
E	011

the revised tree is shown in Figure 6.

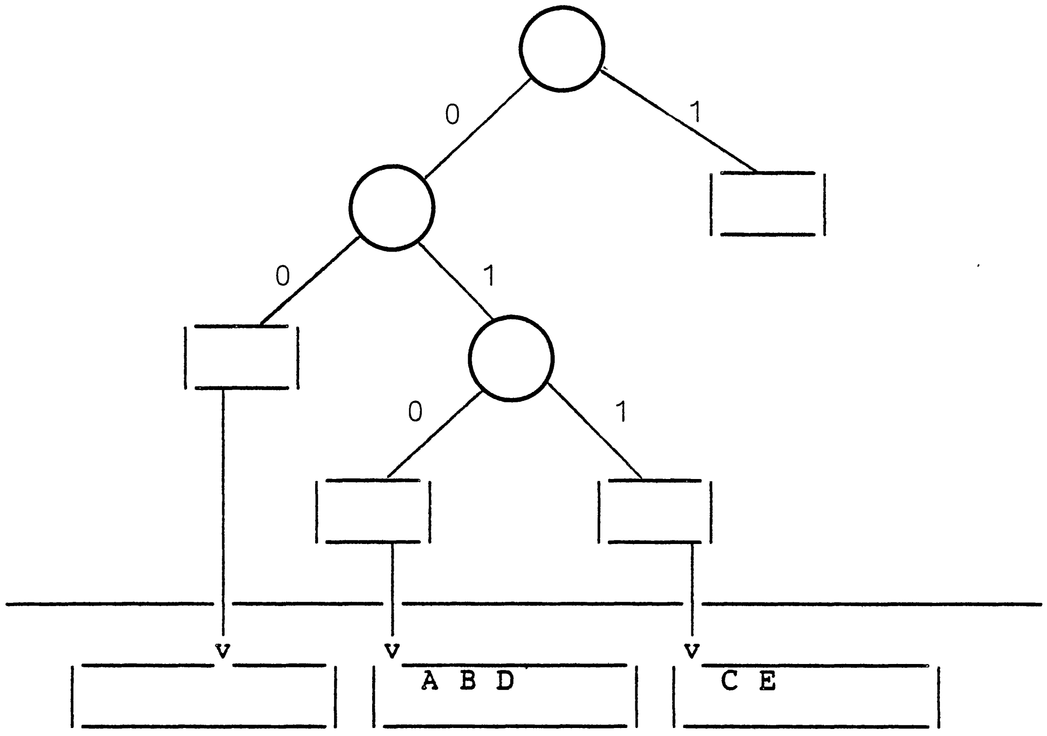


Figure 6. File Structure From Figure 5 After one split

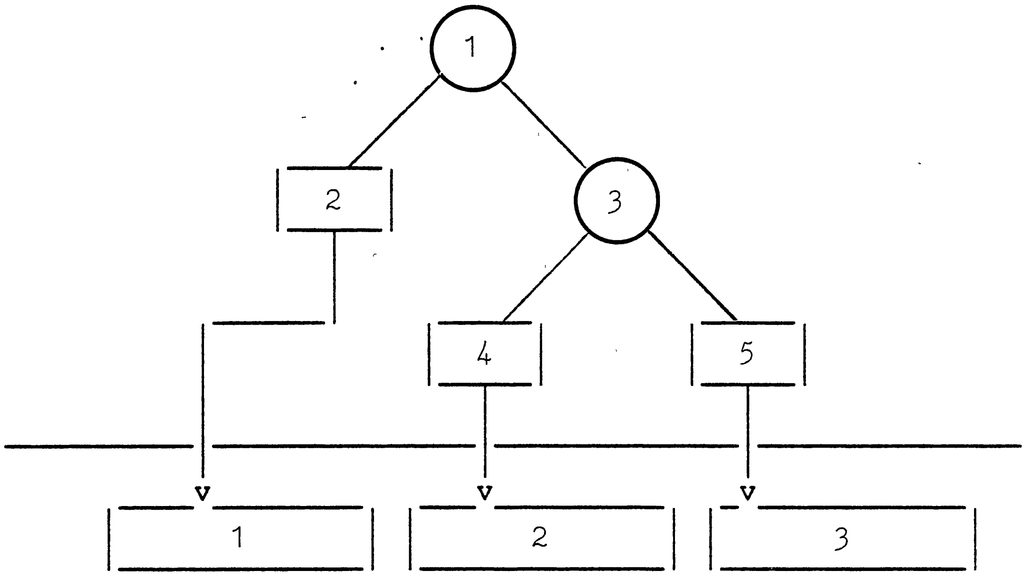
Bucket 2 is split into buckets 2 and 3. Keys A, B, and

D are in bucket 2 and C and E in bucket 3. Node 5 becomes a father and has two sons: node 6 and node 7.

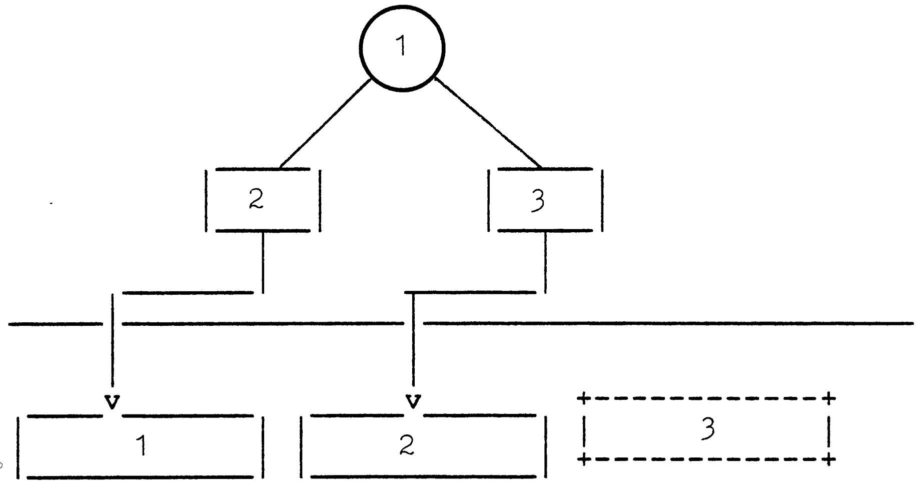
Deletion

To delete a key in the file, first perform the search routine. If the key is not in the bucket, then the key is not in the file, otherwise delete the key from the bucket. It is obvious that after heavy deletion operations the file will become sparsely occupied, which decreases the space utilization of the file. To avoid this setback, a merge routine is associated with the deletion operation. After each deletion operation is performed, a check routine is then performed to see if the total number of records in the current bucket and its brother bucket (the bucket pointed to by the brother node of the node that points to the current bucket) becomes less than or equal to the capacity of one bucket. If it does, a merge routine is called to merge the two brother buckets by moving all keys in either bucket into the other bucket and freeing the bucket that has become empty. At the same time the index is updated. Figure 7 shows the file structure after one merge.

Bucket 2 and bucket 3 are merged, all keys are moved into bucket 2. Bucket 3 is freed. The pointer which points to bucket 3 is copied to node 3, and node 4 and 5 are freed. Node 3 becomes an external node. This merge operation improves the space utilization of the file even if heavy insertions and deletions are involved.



(A) Before Merge



(B) After Merge

Figure 7. File Structures of Dynamic Hashing Before and After One Merge

Performance

It takes only one disk access to search for a key if the index is kept in main memory. This guarantees performance efficiency. The number of disk access needed when inserting a key can be determined and is in table I.

TABLE I

NUMBER OF DISK ACCESSES FOR DYNAMIC HASHING

		Insertion	Deletion
Splitting or Merging does not occur	The bucket is or becomes empty	1 write	1 read
	The bucket is not empty	1 read 1 write	1 read 1 write
Splitting or Merging occurs		1 read 2 write	2 read 1 write

As mentioned, it may take more than one split to insert a key. Let s denote the number of splits and b the bucket size. The probability that s splits are needed to insert a key is

$$P(s) = (1-0.5^{b})*0.5^{b(s-1)} \quad , s = 1,2,3,... \; [15].$$

The derivation of P(s) together with an example will be given later when dynamic hashing with deferred splitting is introduced.

With a bucket size of 10, the probability of 2 splits is then 9.766*10**-4, the number decreases to 8.882*10**-16 when bucket size increases to 50 (** means exponential).

Each index tree in the forest is an extended binary tree in which every node has either two sons or none and the number of internal nodes is always one less than the number of external nodes. Therefore, a forest of m extended binary trees with a total of k internal nodes has k+m external nodes. With the fact that index trees are closely related to binary tries, the number of internal and external nodes in the index is approximated with n being the number of keys in the file:

Number of internal nodes = n/(b*ln2)-m

Number of external nodes = n/(b*ln2)

Assuming each external node is associated with a bucket, the space utilization becomes: (n/b)/(n/(b*ln2)) which has the value of 0.693, i.e., 69.3%.

A Variant - Dynamic Hashing With Deferred Splitting

The space utilization of dynamic hashing can be improved by deferring splitting of a bucket until a certain number of overflow keys have been inserted. The price paid for the improvement of space utilization is that more than one disk access may be needed to search for a key.

When splitting the bucket, b*y+1 keys are to be distributed between two allocated buckets each with a bucket size of b. The random function is called to generate the next binary value for each key. The keys with a binary value of 0 will be put into one bucket, and the keys with a binary value of 1 will be put into the other bucket. However, if the number (b') of keys which generate 0(1) exceeds bucket size b, then the current splitting fails because a bucket can only store at most b keys. At this point two options may be chosen to complete the splitting operation. The first option is that we actually store the remaining (b*y+1)-b' of the keys in one bucket and split the b' keys again until we can successfully separate the keys and store them in two buckets. More than two buckets may be allocated by choosing this option which tends to decrease space utilization. The second option is that we keep splitting the bucket until the (b*y+1) keys can be separated and stored into two buckets. The second option is used in this thesis because it is easier to make a mathematical study and it does not decrease space utilization dramatically like the first option. An example follows.

Let us assume that the random function is called twice to separate the keys, and the index is updated accordingly.

Figure 8 shows the index structure after the split.

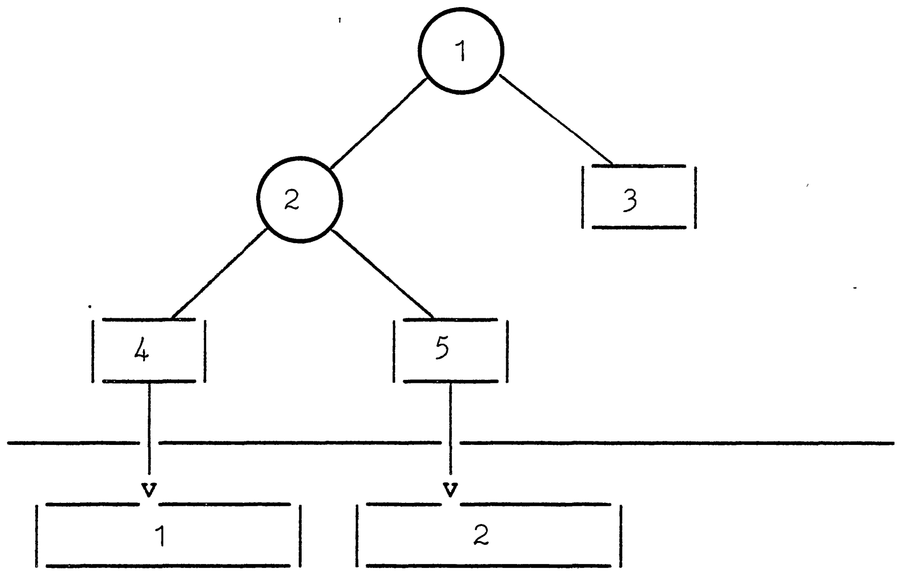


Figure 8. File Structure Containing Inactive Node 3

The first time the random function is called, nodes 2 and 3 are allocated; they both are external nodes. Since the keys cannot be separated, the random function is called again and nodes 4 and 5 are allocated; node 2 becomes an internal node. This time the keys are separated and distributed between buckets 1 and 2. Notice that node 3 is an external node, but it does not have a pointer that points to a bucket on secondary storage; it has a null pointer. Side effect will take place if, after a while, a key needs to be inserted and by traversing the path, node 3 is reached. Since node 3 is an external node, it is assumed that the key should be inserted in the bucket pointed to by node 3. With the fact that there is not a bucket associated with node 3, the key is lost!

To avoid this situation, it should not be assumed that all external nodes point to a bucket. In the process of inserting a key, by scanning down the binary tree, when an external node is reached, one more check is needed to decide if the external node is active (it does have a pointer pointing to a bucket). If a node is found to be inactive (contains a null pointer), a restore process should be performed by traversing back to its father node and going down to its brother node. Then start scanning down the tree again until another external node is reached. At this point the node may or may not be active, so a check should be made whenever an external node is reached until an active node is found. Figure 9 shows a situation in which there are two (nodes 3 and 7) inactive external nodes along a path.

1st Variant

Let us assume that splitting of a bucket is deferred until y*b keys have been inserted into the bucket, where y is a number greater than 1. There are two cases: $y <= 2$ or $y > 2$. First consider the case where $y <= 2$ (Figure 10a). When a "home" bucket has become full (bucket 1) and a key is to be inserted into this bucket, a new "overflow" bucket

(bucket 2) is allocated and chained to bucket 1. The over-flow key is then inserted into the overflow bucket. Any

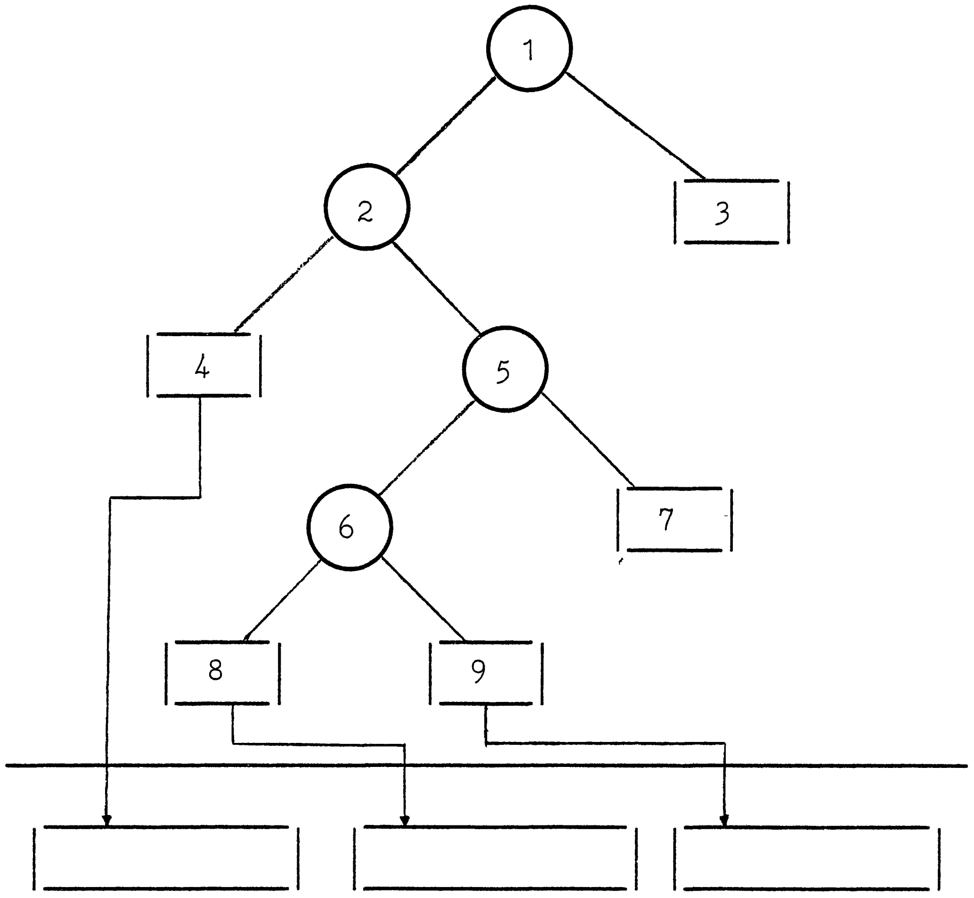
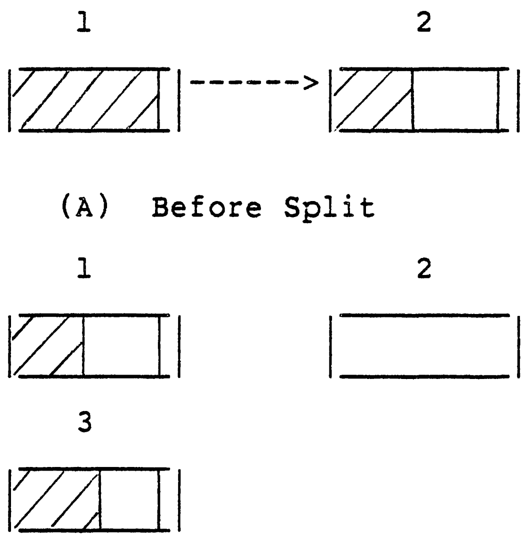


Figure 9. File Structure Containing Two Inactive Nodes

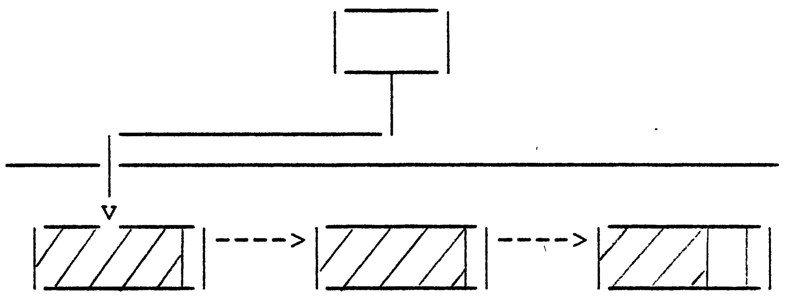
other overflow key which is to be inserted into bucket 1 is now inserted into bucket 2 until there are (y-1)*b keys in bucket 2. The next time a key is inserted into bucket 1, splitting occurs. A new bucket (bucket 3) is allocated and all the keys in buckets 1 and 2, together with the current overflow key, are distributed between buckets 1 and 3. Bucket 2 is freed (Figure 10b). The index is updated in



(B) After Split

Figure 10. Bucket Structure Before and After Split

exactly the same way as in the dynamic hashing scheme. Now consider the case where y > 2 (Figure 11a). When the home bucket is full, the first overflow bucket is allocated and chained to the home bucket. When the first overflow bucket is full, another overflow bucket is allocated and chained to the first overflow bucket, and so on, until y*b keys have been inserted. When splitting occurs (Figure 11b), the index is updated, node 1 becomes an internal node, two allocated external nodes (nodes 2 and 3) become sons of node 1, and


(A) Before Split

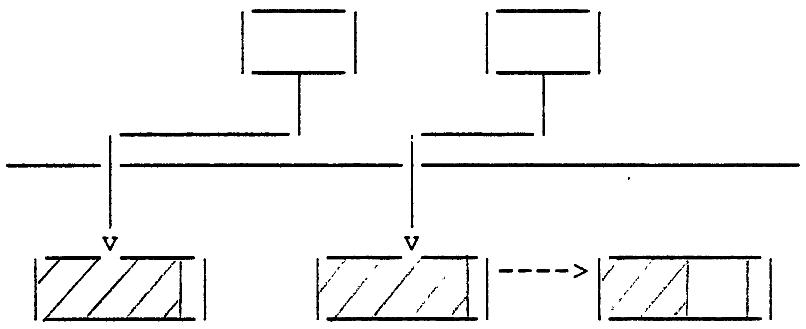

(B) After Split

Figure 11. File Structures of Dynamic Hashing With Deferred Splitting Before And After One Split

each of them points to a chain of one or more buckets. The y*b+1 keys are then separated by the random function and distributed among new chains of buckets.

Deferred splitting of buckets means deferred growing of of index. The approximate numbers of internal and external nodes are:

Number of internal nodes = n/(y*b*ln2)-m

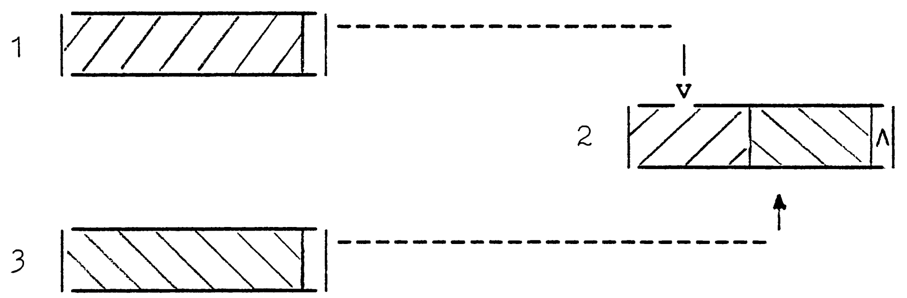
Number of external nodes = n/(y*b*ln2)

The index size is approximately decreased by a factor of (1-1/y). When the index is kept in main memory, the number of disk accesses to search for a key is CEIL(y) in the worst case.

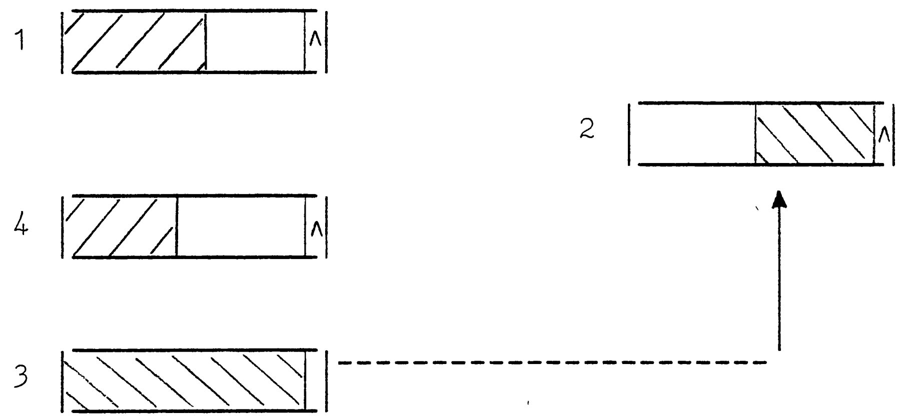
2nd Variant

Space utilization can be improved considerably by using shared overflow buckets. Consider the structure in Figure 10 and assume that the bucket size is 10 and y is 1.5. When splitting occurs, only 5 keys are in bucket 2. The rest of the space is wasted. This wasted space can be utilized when another home bucket (bucket 3) has overflowed. Instead of allocating another overflow bucket to bucket 3, the second half of bucket 2 is used as an overflow area for bucket 3 (Figure 12a).

Sooner or later either bucket 1 or bucket 3 is split. If bucket 1 is split, a new bucket (bucket 4) is allocated and 16 keys are distributed between buckets 1 and 4. Half of the space in bucket 2 is freed (Figure 12b). Two things may occur now. The first is that bucket 2 is split before another home bucket needs the overflow space in bucket 2. A

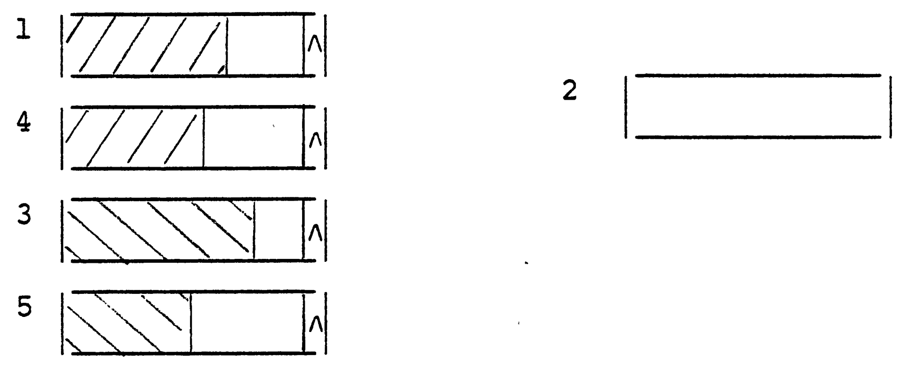


(A) Before Split

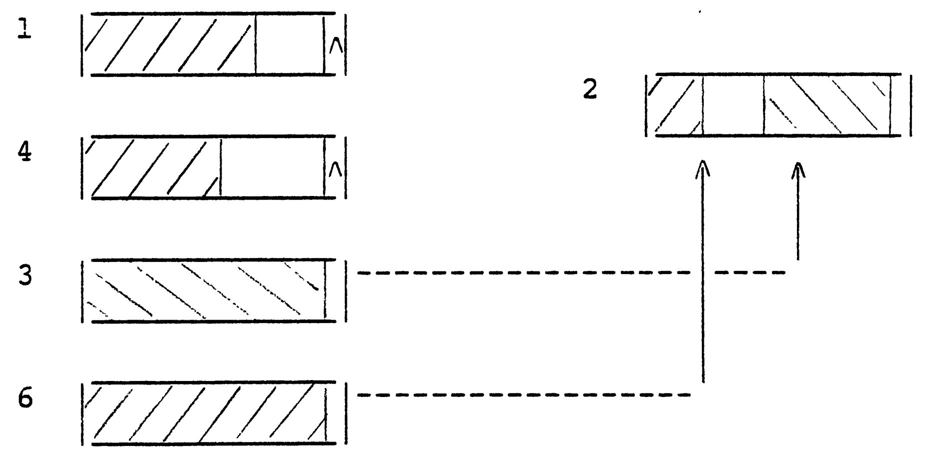


(B) After One Split

Figure 12. File Structures of Dynamic Hashing With Deferred Splitting Before And After One Split



(A) Before Split



(B) After One Split

Figure 13. File Structures of Dynamic Hashing With Deferred Splitting Before And After One Split

new bucket (bucket 5) is allocated, and bucket 2 is freed (Figure 13a). The second thing that may occur is before bucket 2 is split, another home bucket (bucket 6) needs overflow space. Bucket 2 is chained to bucket 6 and the first half of the space is now available to store overflow keys (Figure 13b).

1/(y-1) home buckets may share the same overflow bucket. In this case space utilization should be considerably improved at the expense of complexity in bucket management. The index size and worst case number of disk accesses are the same as the first variant.

Performance

It may take more than one disk access to search for a key. The number of disk accesses needed when inserting or deleting a key (assuming $y < 2$) can be determined and is shown in table II. The probability of s splits to insert a key obeys the geometric probability law with parameter p where $0 <= p <= 1$:

$$P(s) = p(1-p)^{(s-1)} \qquad s = 1,2,3,\ldots$$
$$= 0 \qquad \text{otherwise.}$$

In order to obtain the value of p, which is the probability of success, we need to supply the binomial probability law with parameter n and p', where $n = 1,2,3,\ldots$, and $0 <= p' <= 1$.

$$P(x) = \binom{n}{x} p'^{x}(1-p')^{n-x} \qquad \text{for } x = 0,1,2,\ldots$$

= 0 otherwise.

The expected number of splits is then

$$E(s) = 1/[\sum_{x=c+1-b}^{b} \binom{c+1}{x} 0.5\ 0.5 \quad]$$

An example is as follows: assume b=10, y=1.5, and

TABLE II

NUMBER OF DISK ACCESSES FOR DYNAMIC HASHING WITH DEFERRED SPLITTING

		Insertion	Deletion
Splitting or Merging does not occur	The bucket is or becomes empty	1 write	1 read
	The bucket is not empty	1 read 1 write	1 read 1 write
		1 read 2 write	2 read 1 write
Splitting or Merging occurs		2 read 2 write	2 read 1 write
			3 read 1 write

c=b*y=15. When a split occurs, the keys may be distributed in a number of ways. Table III lists all of the possible different combinations of keys. Notice that only under the condition that 6 <= x <= 10 do we have successful distributions. All other conditions are failure ones. Therefore,

TABLE III

COMBINATIONS OF KEYS DISTRIBUTION

x	bucket 1	bucket 2	condition
0	0	16	failure
1	1	15	failure
2	2	14	failure
3	3	13	failure
4	4	12	failure
5	5	11	failure
6	6	10	success
7	7	9	success
8	8	8	success
9	9	7	success
10	10	6	success
11	11	5	failure
12	12	4	failure
13	13	3	failure
14	14	2	failure
15	15	1	failure
16	16	0	failure

* y=1.5
** b=10

$$P(s) = p(1-p)^{(s-1)} \text{ where}$$

$$p = \sum_{x=6}^{10} \binom{16}{x} p'^{x} (1-p')^{16-x} \text{ where } p' = 0.5.$$

P(s=1) = 0.78988(0.21012)**0 = 0.78988

P(s=2) = 0.78988(0.21012)**1 = 0.16597

P(s=3) = 0.78988(0.21012)**2 = 0.03487

P(s=4) = 0.78988(0.21012)**3 = 0.00732

E(s) = 1/p = 1/0.78988 = 1.266

1.266 splits are expected to distribute successfully the keys each time a split operation is performed!

CHAPTER III

IMPLEMENTATION

An implementation of the two schemes has been done under UNIX and written in C. The implementation of dynamic hashing with deferred splitting is by using shared overflow buckets with y = 1.5. For both schemes, the number of binary trees in the forest is 10 and bucket sizes range from 10 to 50 with an interval of 10. 30,000 random numbers are chosen as keys. However, some keys appeared more than once which tend to decrease the randomness of the keys. The procedures are presented in algorithmic form.

Data Structures

The data structure of a single node in the index is in Figure 14.

A "union" is a type of variable which may hold (in the same place but at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements. Union provides a way to manipulate different kinds of data in a single area of storage, therefore internal nodes and external nodes can be used interchangeably [16].

Two buffers are maintained in main memory; each one can hold a bucket. They serve as an intermediate area

```
struct    {
                   short  TAG
                   int    FATHER
                   union  {
                                   int  LEFT
                                   int  RCRD
                   }
                   union  {
                                   int  RIGHT
                                   int  BKT
                   }
}
```

Figure 14. Data Structure of Index Node

between main memory and secondary storage.

Logic Design

<u>Search</u>

The basic design of the search routine for dynamic hashing is:

1. Hash the key to locate a root node in the index.

2. Initialize random function using the key as the seed.
3. Scan down the tree until an external node is found.
4. Read the bucket associated with the external node into the buffer.
5. Search for the key in the buffer.

The way to scan down the tree is decided by the binary value generated by the random function. If a 0 is generated, go to the left son, otherwise go to the right son. The binary sequence generated by the random function from the root to an active external node determines the unique path of a key in the index.

The design of the search routine for dynamic hashing with deferred splitting is slightly different.

1. Hash the key to locate a root node in the index.
2. Initialize the random function by supplying the key as the seed.
3. Scan down the tree until an external node is found.
4. If the node is inactive, restore the path and go to step 2.
5. Read the home bucket associated with the external node into the buffer.
6. Search for the key.
7. If the key is not found and there is an overflow bucket chained to this home bucket, read the overflow bucket into the buffer and search for the key.

If in a binary sequence, a value leads the search to an inactive external node, then the value is switched to the opposite value and the search goes on. For example, if 1011001 is the path to an active external node, but the second 0 leads the search to an inactive external node, the path becomes 1011101. At step 6, if the key is not found in the home bucket, then read the overflow bucket chained to the home bucket buffer and search for the key again. This is the reason why it may take more than one disk access to search for a key.

Insertion

The insertion routine for dynamic hashing is:

1. Perform the first 4 steps of the search routine.
2. If the bucket is full, perform splitting.
3. Insert the key.

Dynamic hashing with deferred splitting is more complex because more situations have to be handled in order to insert a key successfully. Following is the basic design:

1. Perform the first 5 steps of the search routine.
2. If the number of keys in the bucket is less than b, then insert the key into the home bucket.
3. If the number of keys in the bucket is equal to b, then
   (a) if there is an available overflow bucket, chain the overflow bucket to the home bucket and insert the key into the overflow bucket.
   (b) if there is no available overflow bucket, allo-

cate a new overflow bucket and chain it to the home bucket, then insert the key into the overflow bucket.

4. If the number of keys in the bucket is greater than b but less than y*b, then insert the key into the overflow bucket which is chained to the home bucket.

5. If the number of keys in the bucket is equal to y*b, perform splitting and insert the key.

The complexity is in bucket management. For an overflow bucket, in order to distinguish which part of the space belongs to which home bucket, extra storage needs to be allocated. The way of implementing it is to allocate address fields in the buckets. The address field in a home bucket contains the address of the overflow bucket while the address field in an overflow bucket contains addresses of all the home buckets that are chained to the overflow bucket. The order of the addresses determines which part of the space in the overflow bucket belongs to which home bucket (Figure 15).

Buckets 1 and 2 are home buckets and bucket 3 is the overflow bucket. Bucket 1 uses the first half of the space in bucket 3 because its address is stored in the first address field of bucket 3. Bucket 2 uses the second half of the space in bucket 3 because its address is stored in the second address field of bucket 3.

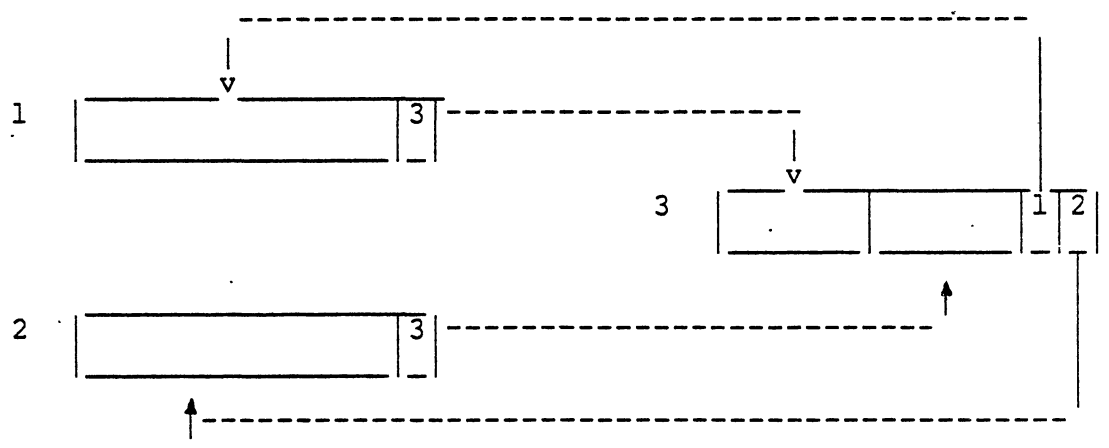


Figure 15. Structure and Relation Between Home Bucket and Overflow Bucket

Deletion

The design of deletion routine for dynamic hashing is

1. Perform search routine.
2. If the key is not found, the key is not in the file, otherwise delete the key.
3. Try to merge two brother buckets if the key is in the file and deleted.

If the total number of keys in the home bucket and it's brother bucket is less than or equal to b, a merge is carried out by moving all the keys in two brother buckets into the left bucket, freeing the right bucket and updating the index by freeing two external nodes associated with two brother buckets.

The same routine for dynamic hashing with deferred splitting is a little more complicated. Following is the design.

1. Perform the search routine.
2. If the key is not found, the key is not in the file, otherwise delete the key.
3. If an overflow bucket is associated with the home bucket and, after the deletion, the number of keys in the home bucket and the overflow bucket is equal to b, the overflow bucket is freed if there is no other home bucket chained to it.
4. Try to merge two brother buckets if the key is in the file and deleted.

Random Function

Let the real number x, $0 <= x < 1$, correspond to the binary sequence $< X(n) >$ where the binary representation of x is (0.X(0)X(1)...) . Under this correspondence, almost all x correspond to binary sequences which are random [17]. With this property in mind, a random function that meets the requirement mentioned in the previous chapter is readily constructed.

When the random function is called for the first time, a seed is supplied, and the random function converts the seed into a real number such that $0 <= x < 1$. This forms the initialization. Subsequent calls (seed = 0) to the random function will cause X(j), j = 1,2,3,... to be extracted and returned as the binary value generated by the random function. For a floating point number, the random function generates up to 24 random binary numbers before it exhausts the mantissa (precision) of the floating point number, while for a double precision floating point number, it can generate up to 56 random binary numbers. If the internal path length of the index tree exceeds 56, the precision can be extended up to infinity. Therefore we can assume that the random function can generate as many random binary numbers as needed. The random function is used in much the same way as a random number generator.

Hash Function

The hash function employed is nothing but an ordinary hash routine implemented by using the division technique. The only thing it does is to locate a root node in the forest. Following is the algorithm.

1. Add up the ASCII value of all the characters in the key.
2. Divide the result of step 1 by m (the number of root nodes in the index) and return the remainder.

CHAPTER IV

RESULTS AND DISCUSSIONS

The empirical results of both dynamic hashing and dynamic hashing with deferred splitting are presented in this chapter. Figures and table indicating empirical results are listed in the Appendix.

Space Utilization

No matter what the bucket size is, average space utilization of dynamic hashing approaches 69% while for dynamic hashing with deferred splitting, the result approaches 81%, a considerable improvement. Figures 16 through 20 show the empirical results of space utilization for both schemes.

It is observed that cyclical variations occur in space utilization performance. The reason is that as buckets become full, space utilization increases. After a while buckets become completely full and are split almost simultaneously and space utilization decreases.

Disk Access

For dynamic hashing, the number of disk access needed to search for a key is always 1. For dynamic hashing with deferred splitting, the number is slightly more than 1.

Figures 21 through 25 show the empirical results.

Again, oscillatory performance of search operations occurs for dynamic hashing with deferred splitting. The frequency of occurrence of overflow keys increases as space utilization increases, resulting in an increase in the cost, in terms of disk accesses, of searches as accessing of the overflow keys becomes increasingly common.

Index Size

Deferred bucket splitting slows down the growth of index trees, i.e. decreases the index size. Choosing a larger bucket size can also decrease index size (Figures 26 through 30).

Path Length

It is obvious that the less the number of splits occur, the shorter the index path length will be. A bucket with a larger size tends to be split less frequently than a bucket with a smaller size. Therefore a larger bucket size causes shorter index path lengths.

As mentioned above the deferred splitting of bucket slows down the growing of index trees. Therefore the index path length is decreased (Figures 31 through 35).

Since the index is organized as a forest of binary trees, it may become unbalanced after all the keys have been loaded. A heavily unbalanced tree will create some very long path lengths and therefore affect performance. Figure

36 to 40 shows the empirical results of tree balancing for both schemes.

Empirical results show that with bucket size 10, the difference of maximum and minimum index path length of dynamic hashing does not exceed 4; the difference is 17 for dynamic hashing with deferred splitting which indicates that the index trees employed by dynamic hashing is more balanced than the index trees employed by dynamic hashing with deferred splitting. The reason for the heavily unbalanced index trees for dynamic hashing with deferred splitting is analyzed in chapter IV. For larger bucket sizes, the index trees are well balanced for both schemes. When b=20, the difference between maximum and minimum index path lengths for dynamic hashing does not exceed 3; the difference does not exceed 4 for dynamic hashing with deferred splitting. When b=30, the difference does not exceed 3 for both schemes. When b=40 and 50, the difference does not exceed 2 for both schemes. Well balanced index trees are observed when larger bucket sizes are employed for both dynamic hashing and dynamic hashing with deferred splitting.

Fixed Main Memory

With fixed main memory size, if all the memory space is used to allocate index nodes, the number of keys that can be stored in the file is increased by using dynamic hashing with deferred splitting. As mentioned in chapter II, the index size is decreased by a factor of y. We can then expect that with a fixed main memory size, the number of keys that can be stored should increase by a factor of y. Table III shows the results under the assumption that main memory size is 14K.

Notice that when b = 10, results indicate the increase is not as large as expected. This is because the number of inactive nodes in the index tends to increase with a small bucket size, and since inactive external nodes do not point to a bucket that contains keys, the number of keys increased is less than what we have expected. For all the other bucket sizes, the percentage of increase of number of records by using dynamic hashing with deferred splitting is very close to y(1.5), which is what we expected.

Bucket Size vs Number of Records

It is obvious from the empirical results that by choosing a larger bucket size, the overall performance for both dynamic hashing and dynamic hashing with deferred splitting is better. However, under certain circumstances this may not be the case. For example, if 30,000 records are loaded on a file only for retrieval purpose, in order to save space, a bucket size of 30 should be chosen because the space utilization with a bucket size of 30 is better than the space utilization with all the other bucket sizes. Larger bucket size may not always result in better performance.

CHAPTER V

SUMMARY AND CONCLUSIONS

Dynamic hashing and dynamic hashing with deferred splitting are two file organization methods that do not require complete file reorganization. They can be very useful for applications that store records in a volatile file maintained on direct access auxiliary storage because a volatile file does not reduce the performance at all. However, if the index is too large to be kept in main memory, part of it must be stored on secondary storage which definitely will affect performance. Therefore any method that can reduce the index size so it can be maintained in main memory is highly desirable.

Conclusions

Dynamic hashing employs an index in main memory to guarantee one disk access for a search operation. Space utilization is about 69%. If index size is not a major factor, this scheme ensures performance to a satisfying degree. The purpose of dynamic hashing with deferred splitting is aimed at improving space utilization. By doing so, index size is decreased too to a certain degree, thus provides a higher probability of keeping the index in main memory. One disadvantage is slight performance degradation in trying to search for a key, another disadvantage is the complexity in bucket management and various routines.

From the empirical results presented above, it is observed that trade-offs exist between dynamic hashing and dynamic hashing with deferred splitting. If time is a major factor, dynamic hashing shows better performance since it guarantees only one disk access to search for a key. However, if main memory size as well as secondary storage is at a premium, dynamic hashing with deferred splitting shows better performance because it employs a smaller main memory size and increases space utilization.

Suggested Future Work

The results in the thesis are obtained by loading 30,000 randomly chosen keys and searching all the keys. It would be an interesting topic if the file becomes dynamic, i.e., if heavy insertions and deletions are involved. This topic is left to future study.

If the keys are in natural order, then after the keys are loaded on the file, the results may be different from the ones in this thesis. This topic is also left to future study.

SELECTED BIBLIOGRAPHY

[1] Newell, A., and Simon, H. A. "The logic theory machine: A complex information processing system." IRE Trans. Inform. Theory, 2,3(Sept. 1956), 61-79.

[2] Adelson-Velskii, G.M., and Landis, Y.M. "An algorithm for the organization of information." Dokl. Akad. Nauk SSSR, 146(1962), 263-266 (Russian). English transl. in Soviet Math. Dokl. 3(1962), 1259-1262.

[3] OS/VS2 ISAM Logic, IBM SY26-3833.

[4] Bayer, R., and McCreight, E. "Organization and maintenance of large ordered indexes." Acta Informatica, 1(1972), 173-189.

[5] E. G. Coffman and J. Eve, "File structures using hashng functions." Communications ACM, Vol 13, No 7, (1970), 427-432.

[6] P. G. Sorenson, J. P. Tremblay and R. F. Deustcher, "Key-to-address transformation techniques." INFOR (Canada) Vol. 16, no 1, (1978), 397-409.

[7] G. D. Knott, "Hashing functions." Computer Journal, Vol 18, (August 1975), 265-278.

[8] J. L. Carter and M. N. Wegman, "Universal classes of hash functions." RC 6687, IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598, 1977.

[9] O. Hanson, Design of Computer Data Files, Computer Science Press, 1982.

[10] D. E. Knuth, The art of computer programming, Vol 3: Sorting and searching, Addison-Wesley, Reading, Mass., 1969.

[11] Larson, P. "Dynamic hashing." BIT, 18(1978), 184-201.

[12] M. Scholl, "New file organizations based on dynamic hashing." ACM TODS, Vol 6, No 1, (1981), 194-211.

[13] J. Nievergelt, "Binary search trees and file organiza-

tion." _Computing Surveys_, Vol. 6, No. 3, (September 1974), 195-207.

[14] E. H. Sussenguth, "Use of tree structures for processing files." _Communications ACM_, Vol. 6, No. 5, (May 1963), 272-279.

[15] A. M. Mood, F. A. Graybill, and D. C. Boes, _Introduction to the Theory of Statistics_. McGraw-Hill series in probability and statistics, 1974.

[16] B. W. Kernighan, D. M. Ritchie, _The C Programming Language_, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[17] D. E. Knuth, _The art of computer programming_, Vol. 2: Semi-numerical algorithms, Addison-Wesley, Reading, Mass., 1969.

APPENDIX

TABLE IV

PERCENTAGE OF INCREASE OF NUMBER OF RECORDS STORED BY DYNAMIC HASHING WITH DEFERRED SPLITTING VS DYNAMIC HASHING WITH MAIN MEMORY SIZE 14K

bucket size	dynamic hashing	dynamic hashing with deferred splitting	percent increased
10	3521	4242	20.48%
20	6876	9916	44.21%
30	10216	14935	46.19%
40	13604	19770	45.32%
50	16742	24878	48.60%

* $y = 1.5$

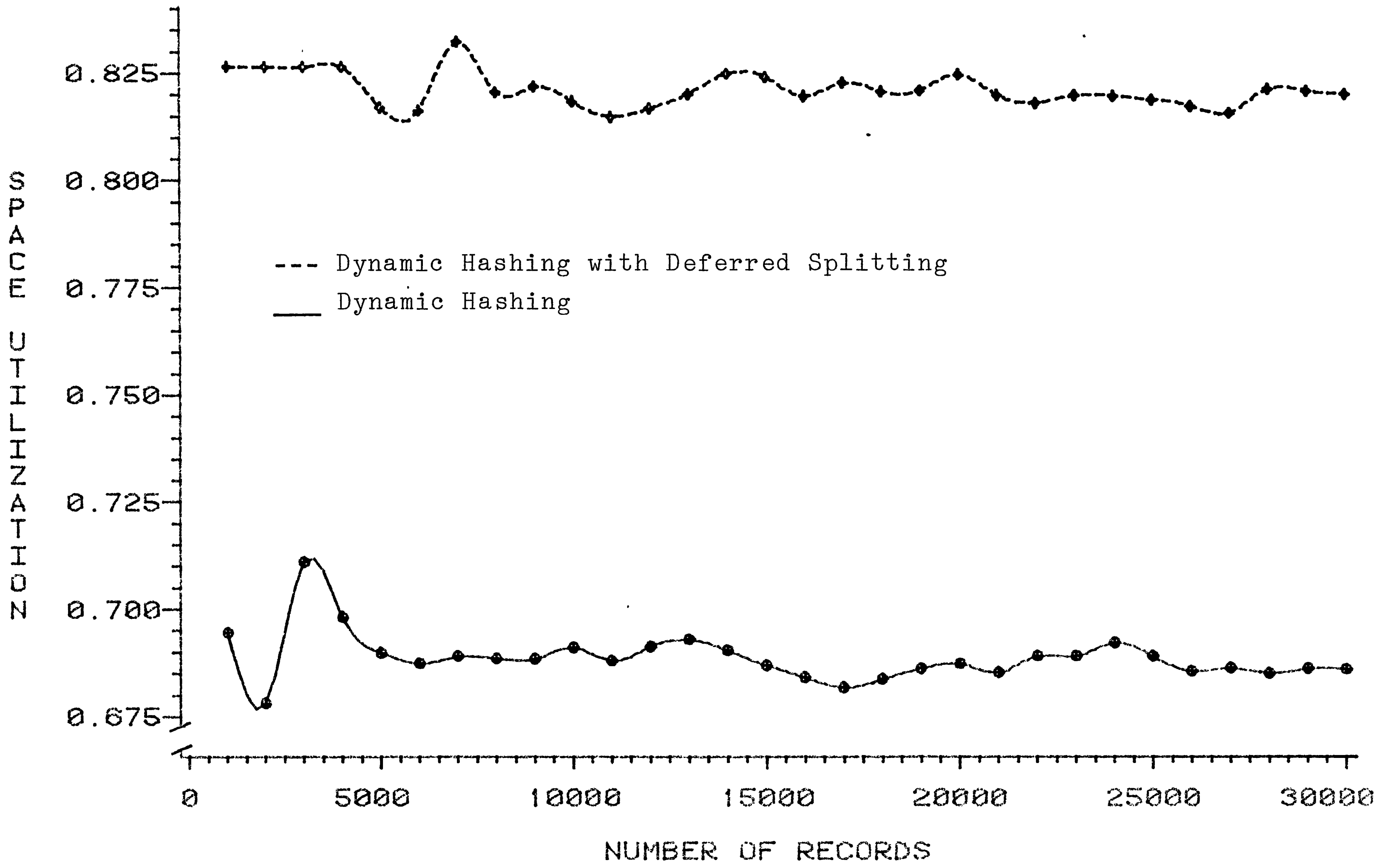


FIGURE 16. SPACE UTILIZATION VS NUMBER OF RECORDS (BUCKET SIZE 10)

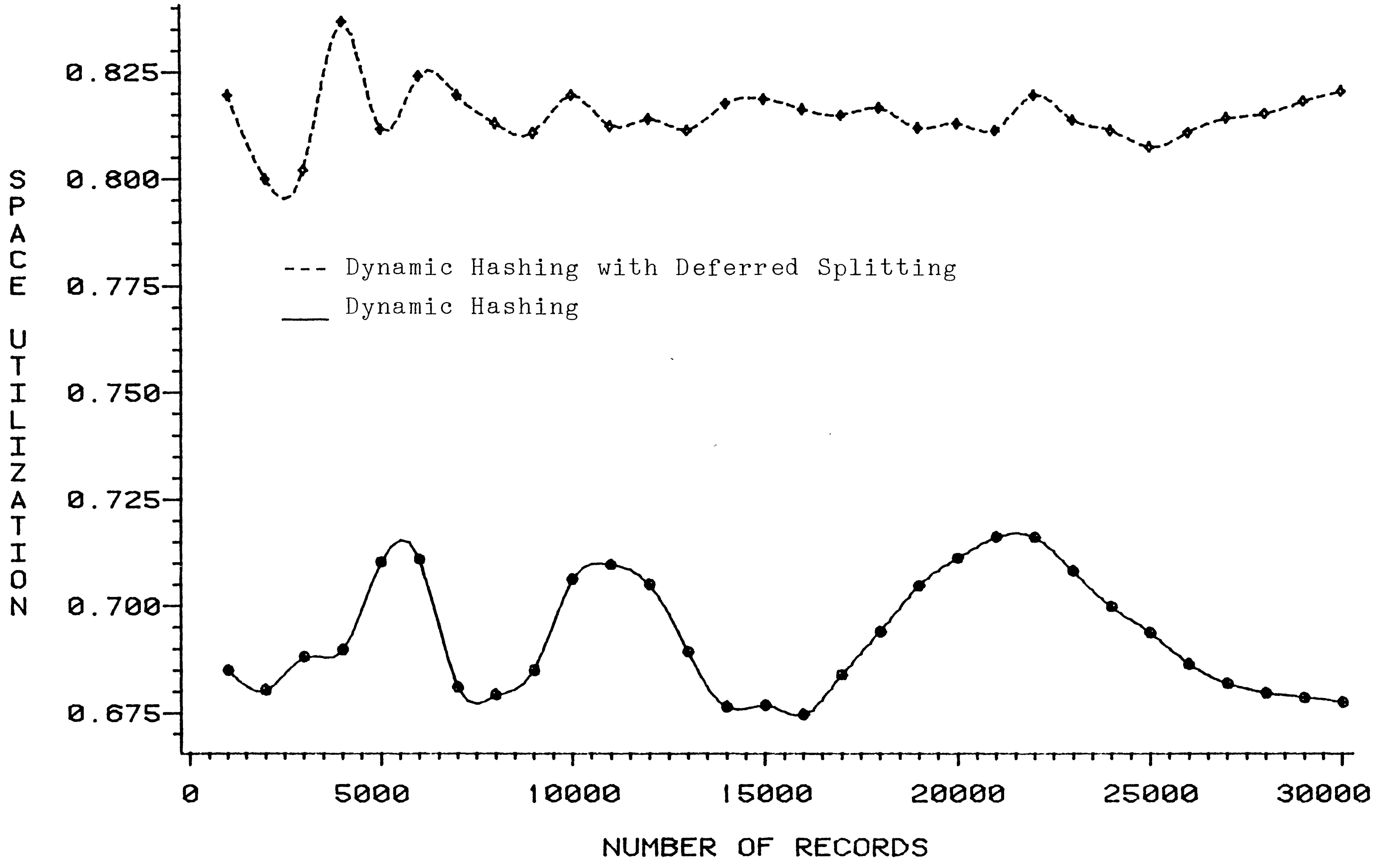


FIGURE 17. SPACE UTILIZATION VS NUMBER OF RECORDS (BUCKET SIZE 20)

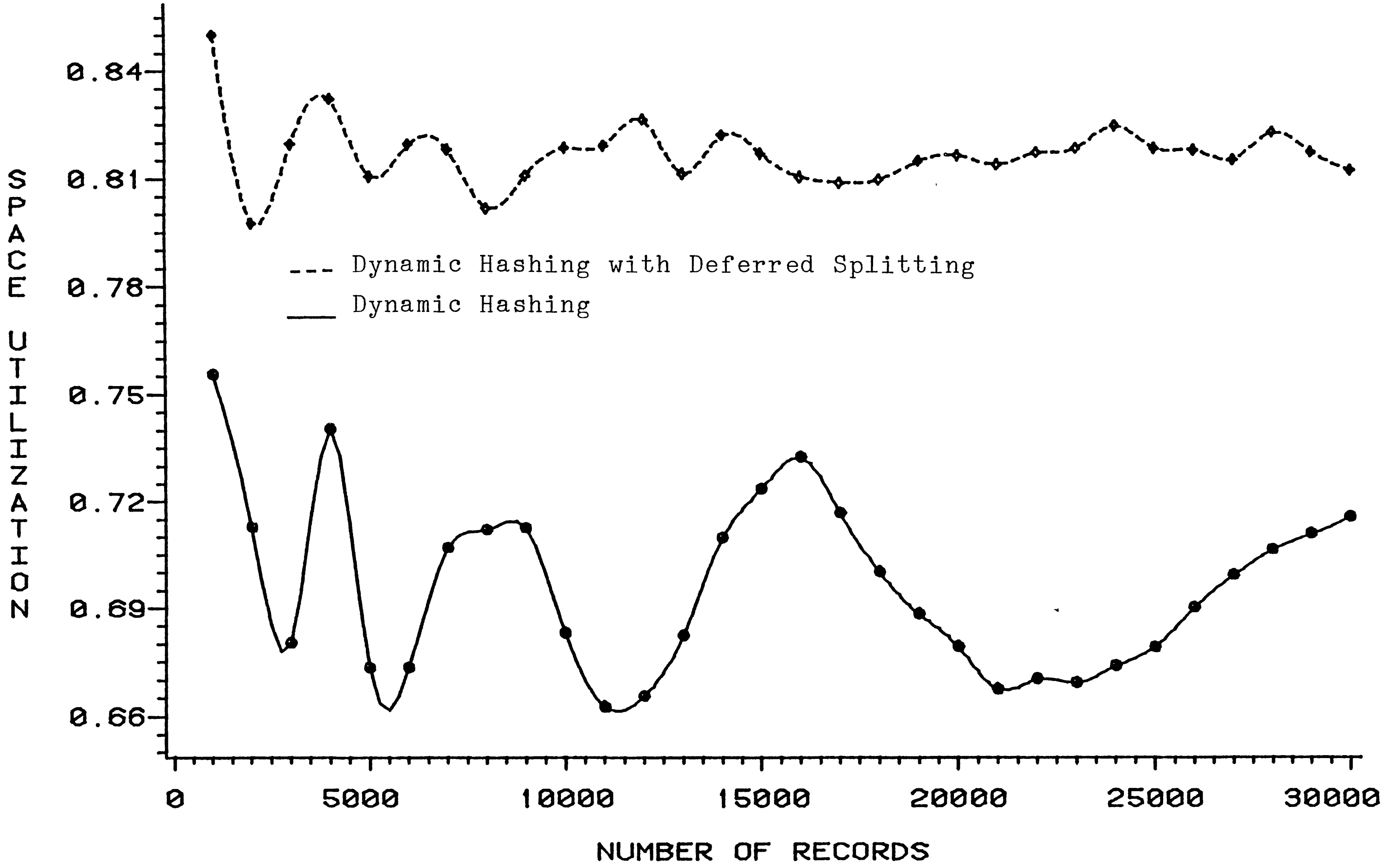


FIGURE 18. SPACE UTILIZATION VS NUMBER OF RECORDS (BUCKET SIZE 30)

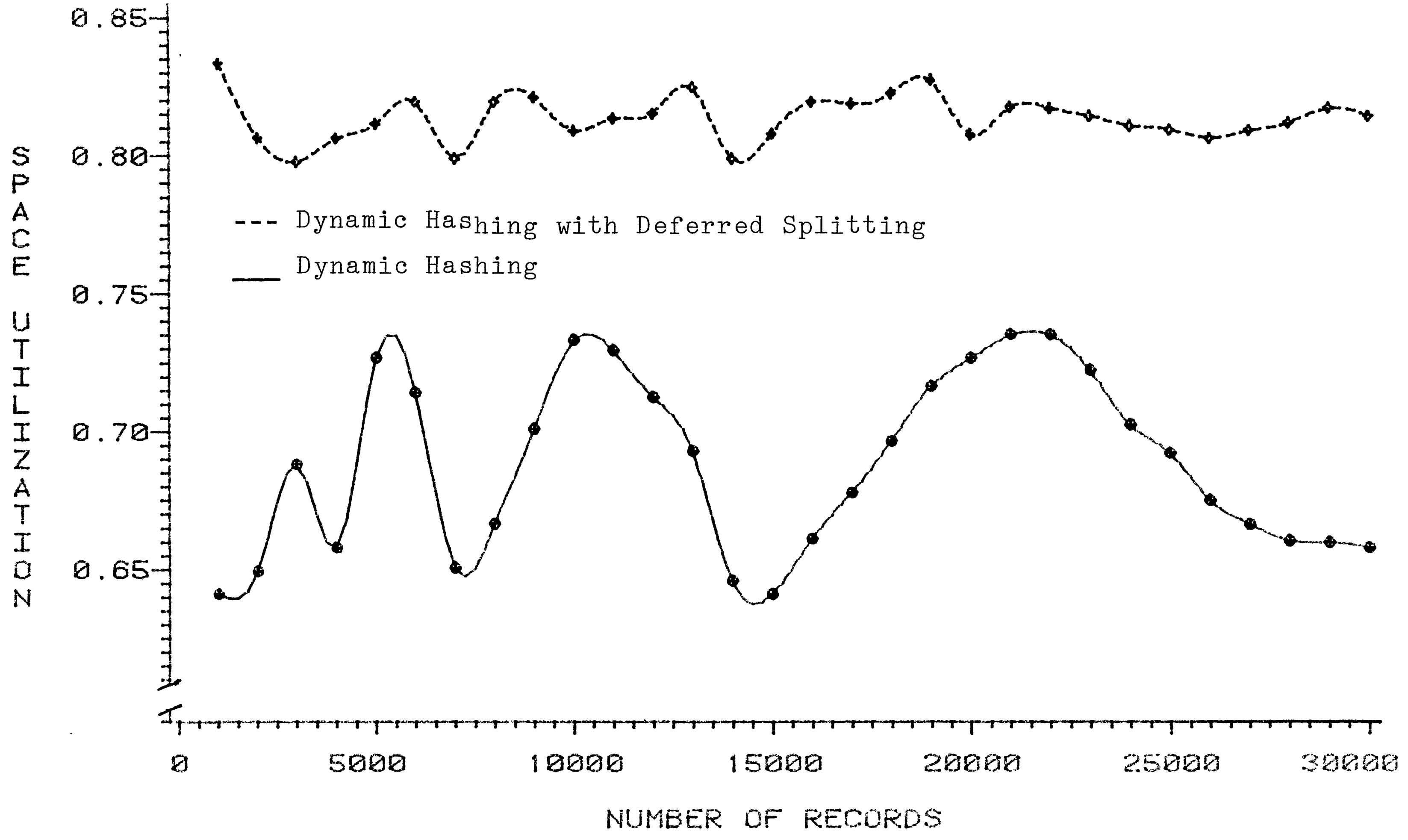


FIGURE 19. SPACE UTILIZATION VS NUMBER OF RECORDS (BUCKET SIZE 40)

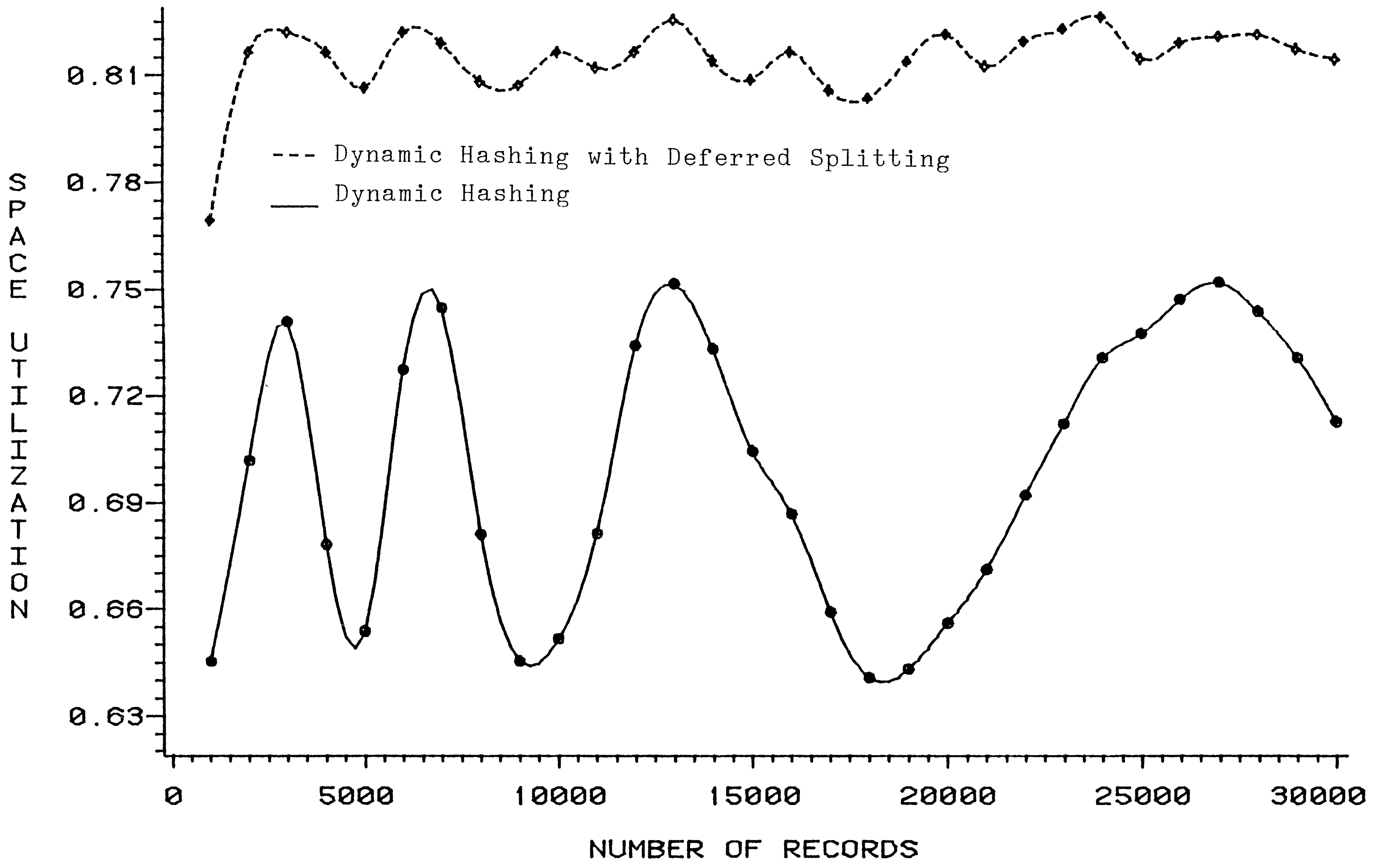


FIGURE 20. SPACE UTILIZATION VS NUMBER OF RECORDS (BUCKET SIZE 50)

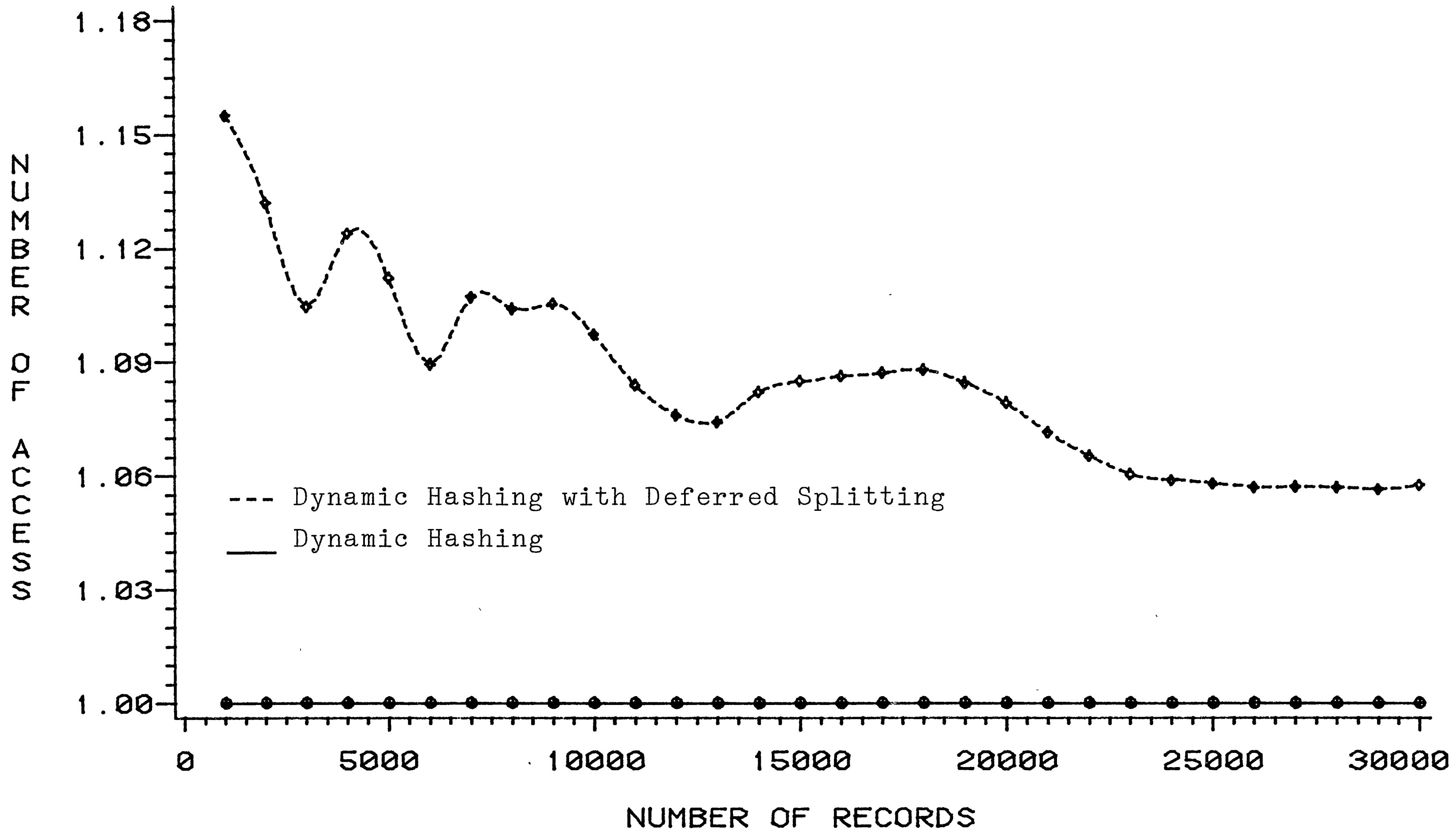


FIGURE 21. AVERAGE NUMBER OF DISK ACCESSES (BUCKET SIZE 10)

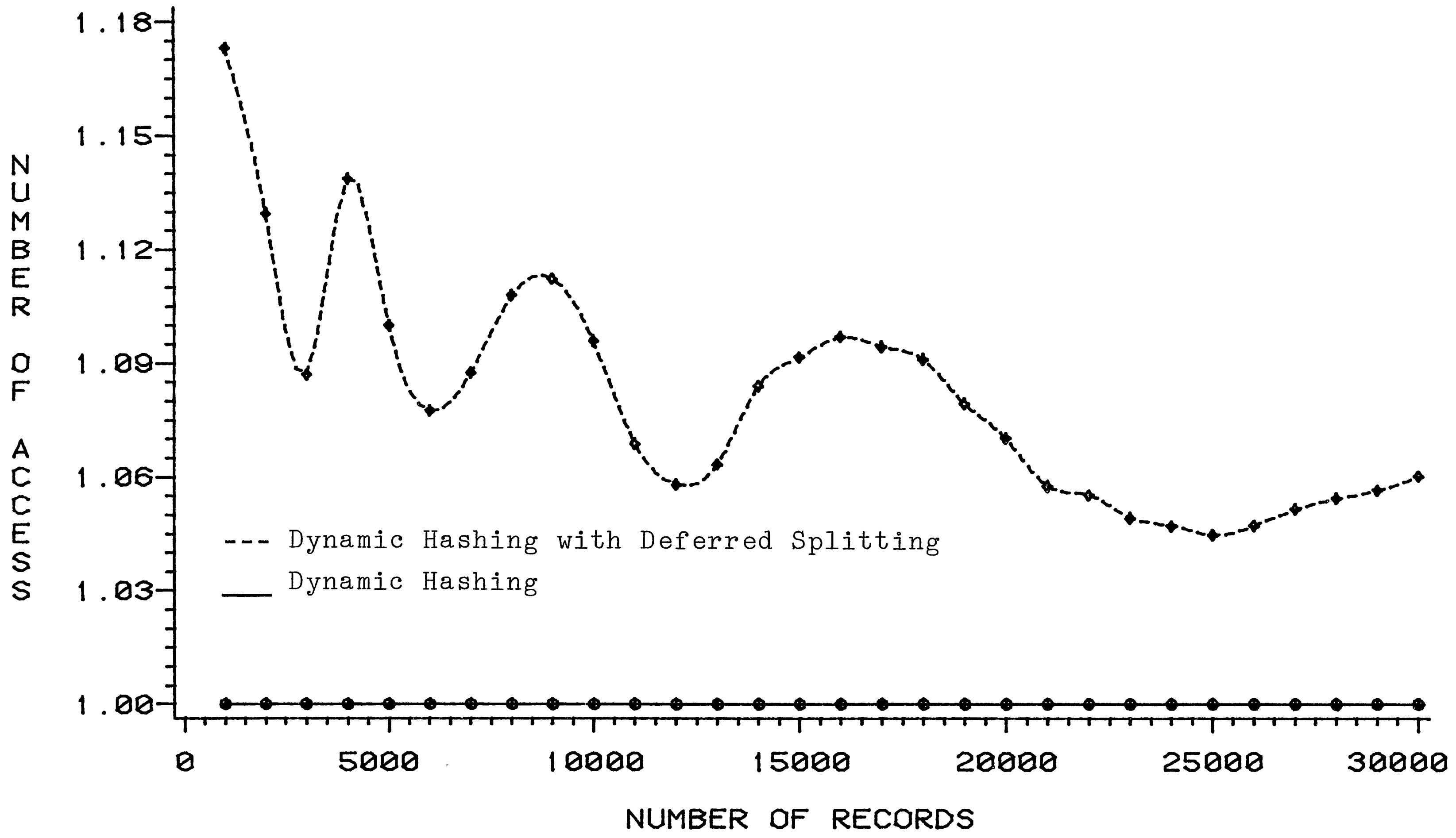


FIGURE 22. AVERAGE NUMBER OF DISK ACCESSES (BUCKET SIZE 20)

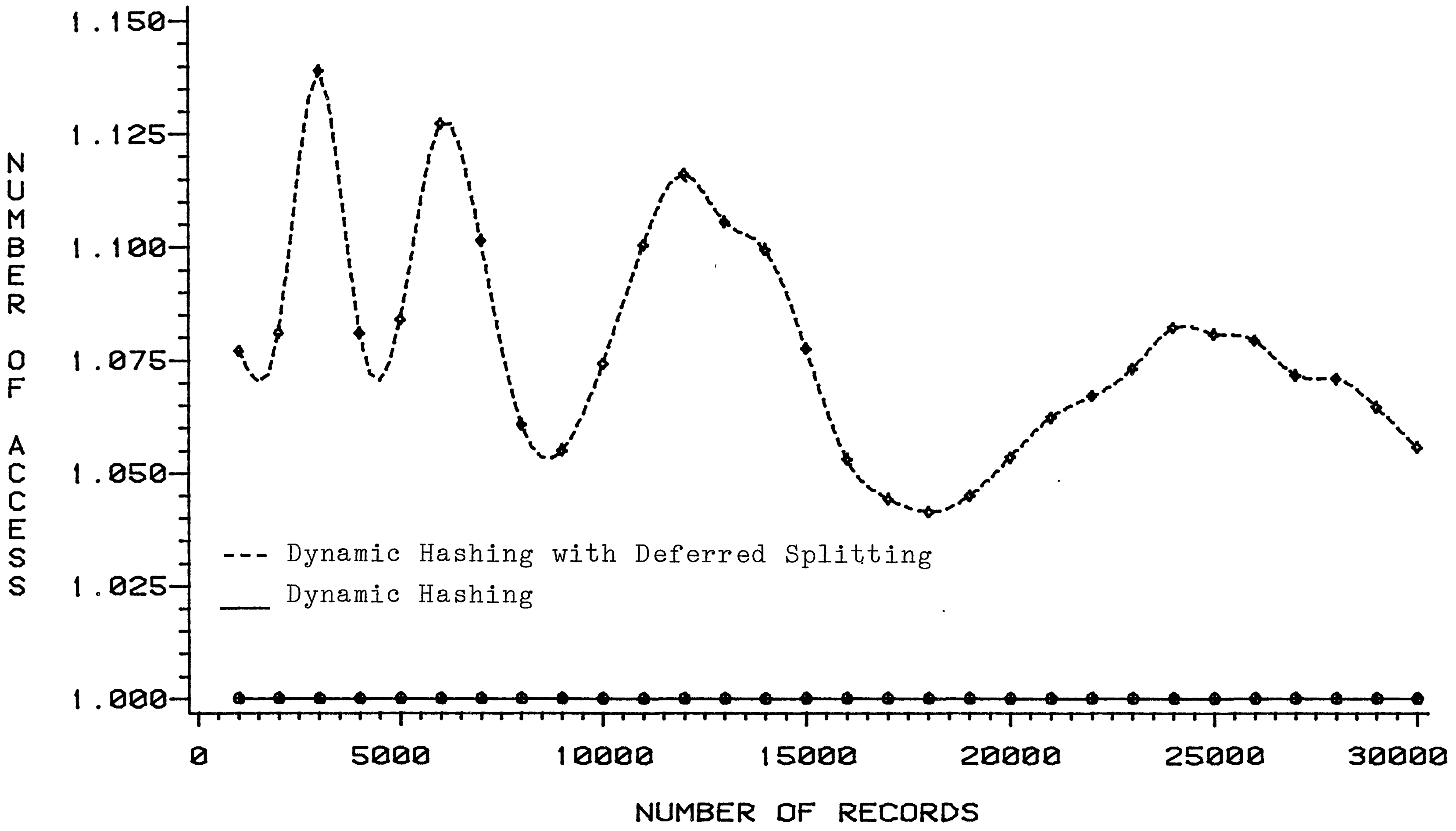


FIGURE 23. AVERAGE NUMBER OF DISK ACCESSES (BUCKET SIZE 30)

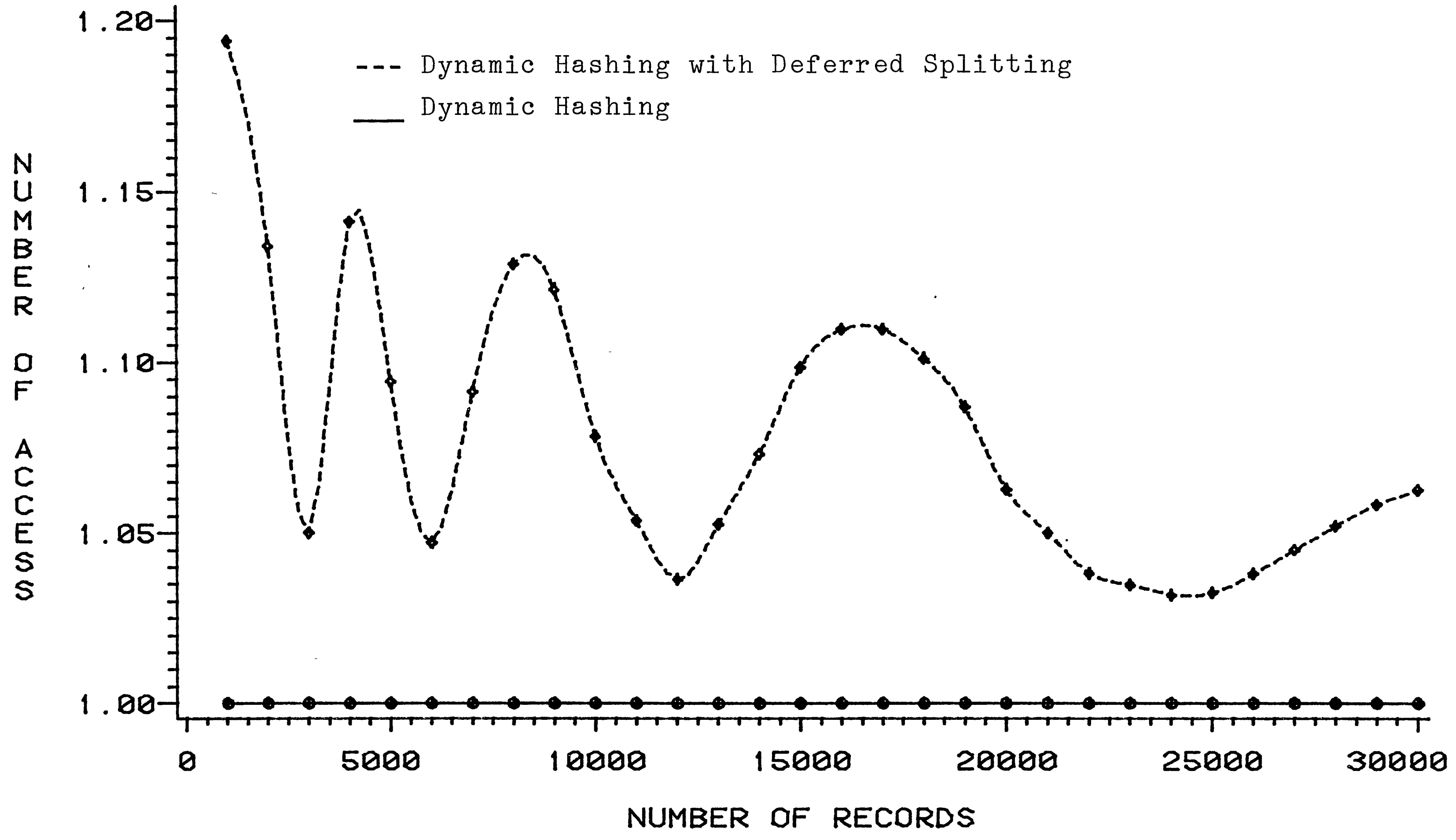


FIGURE 24. AVERAGE NUMBER OF DISK ACCESSES (BUCKET SIZE 40)

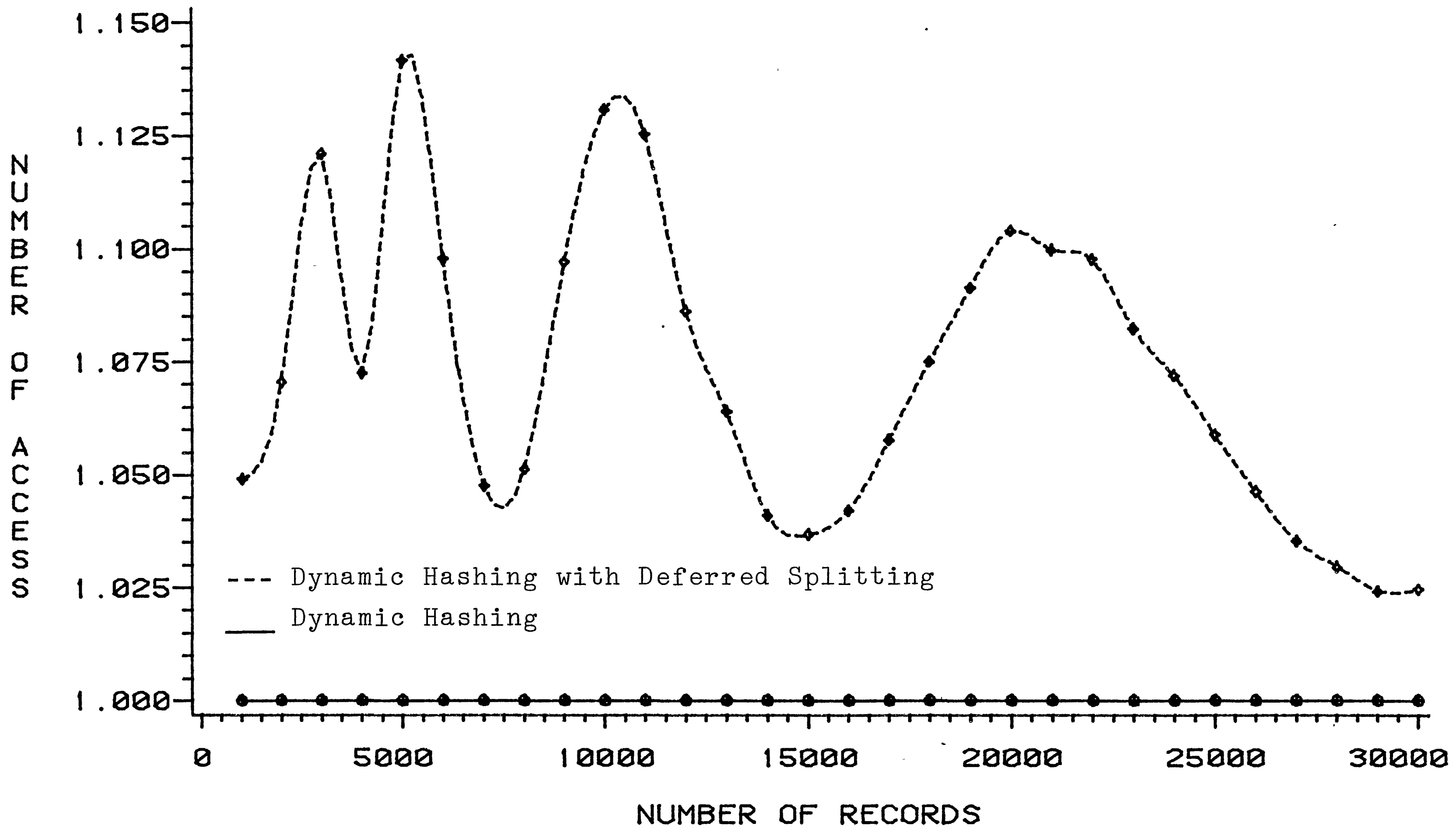


FIGURE 25. AVERAGE NUMBER OF DISK ACCESSES (BUCKET SIZE 50)

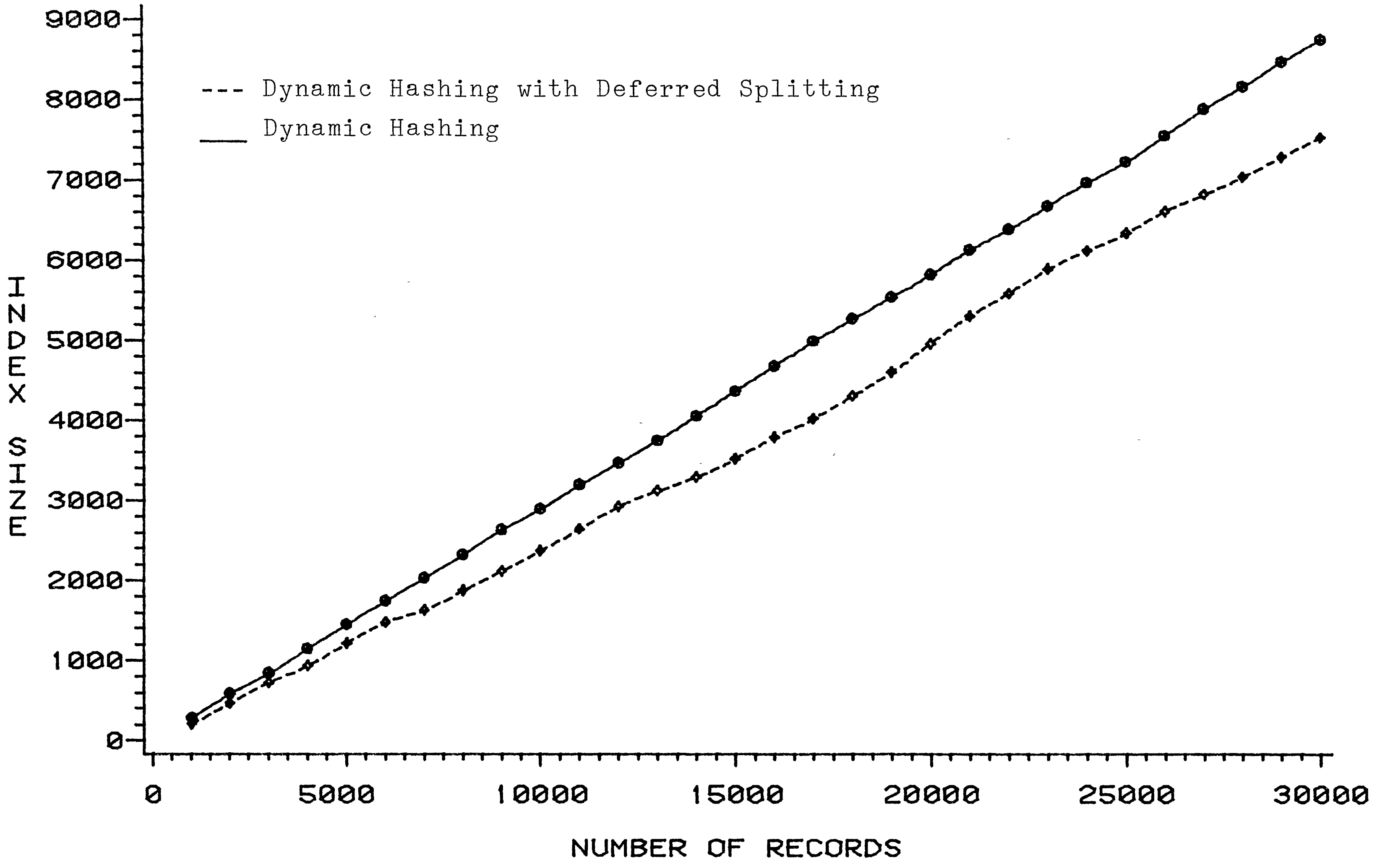


FIGURE 26. INDEX SIZE VS NUMBER OF RECORDS (BUCKET SIZE 10)

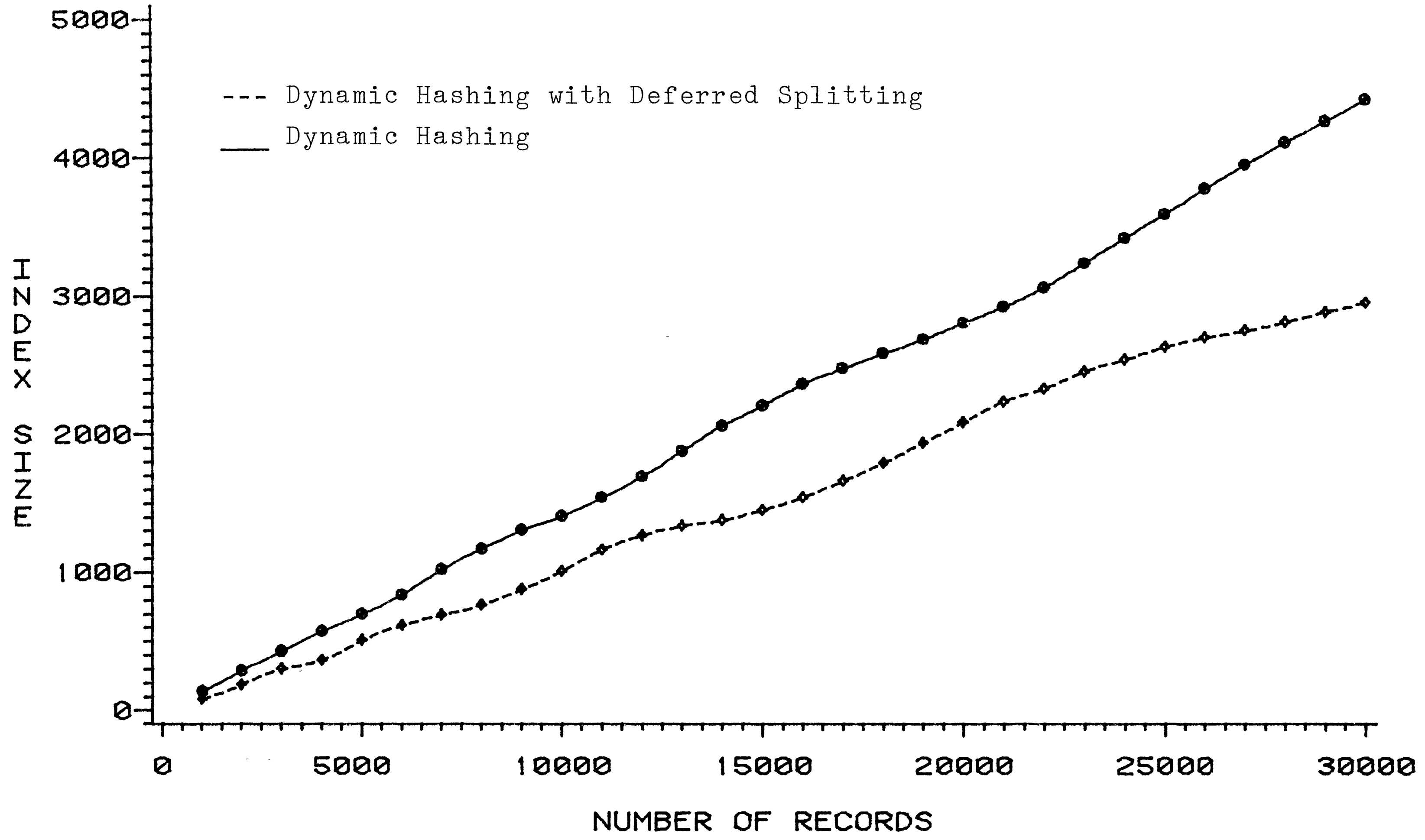


FIGURE 27. INDEX SIZE VS NUMBER OF RECORDS (BUCKET SIZE 20)

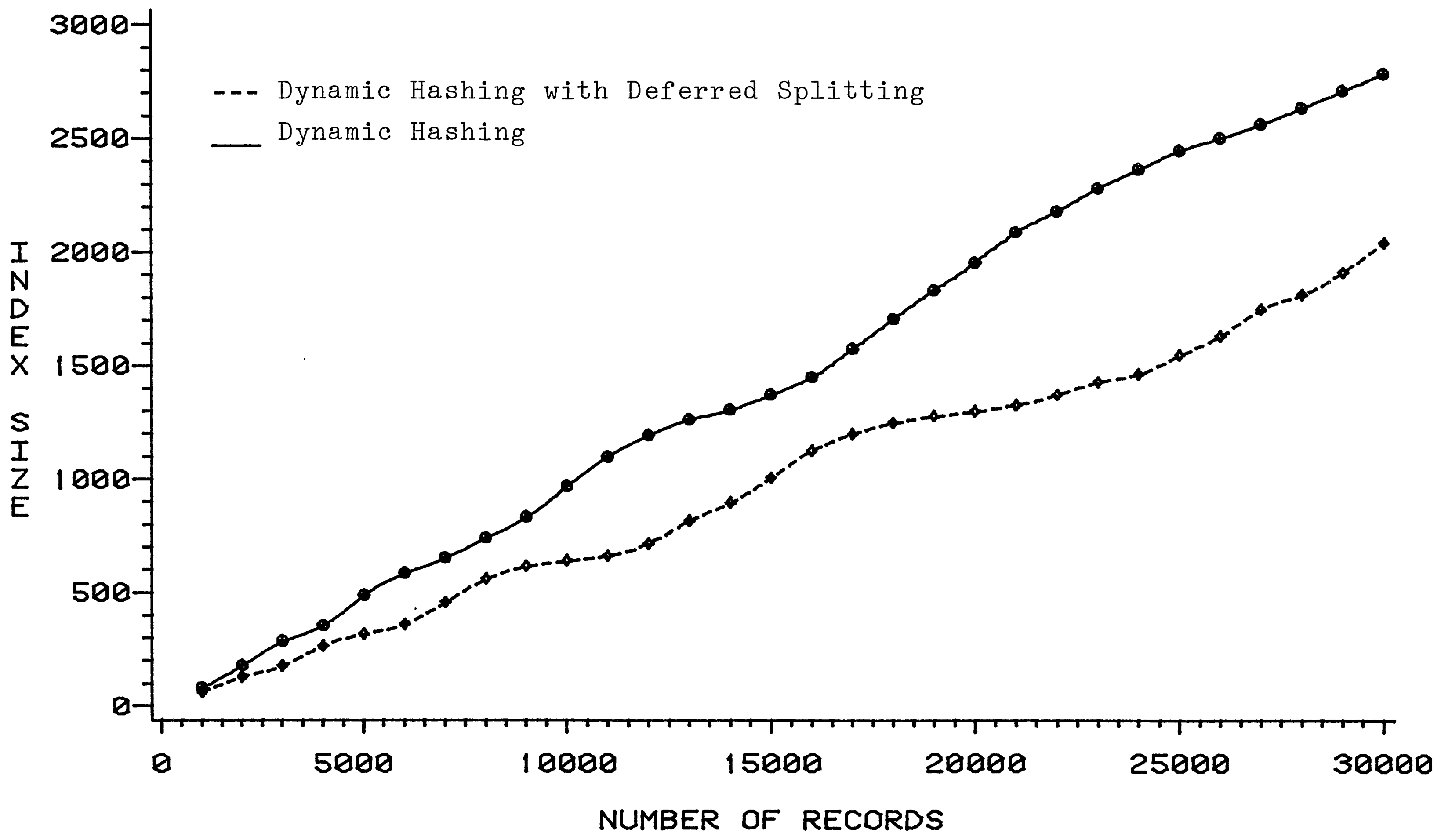


FIGURE 28. INDEX SIZE VS NUMBER OF RECORDS (BUCKET SIZE 30)

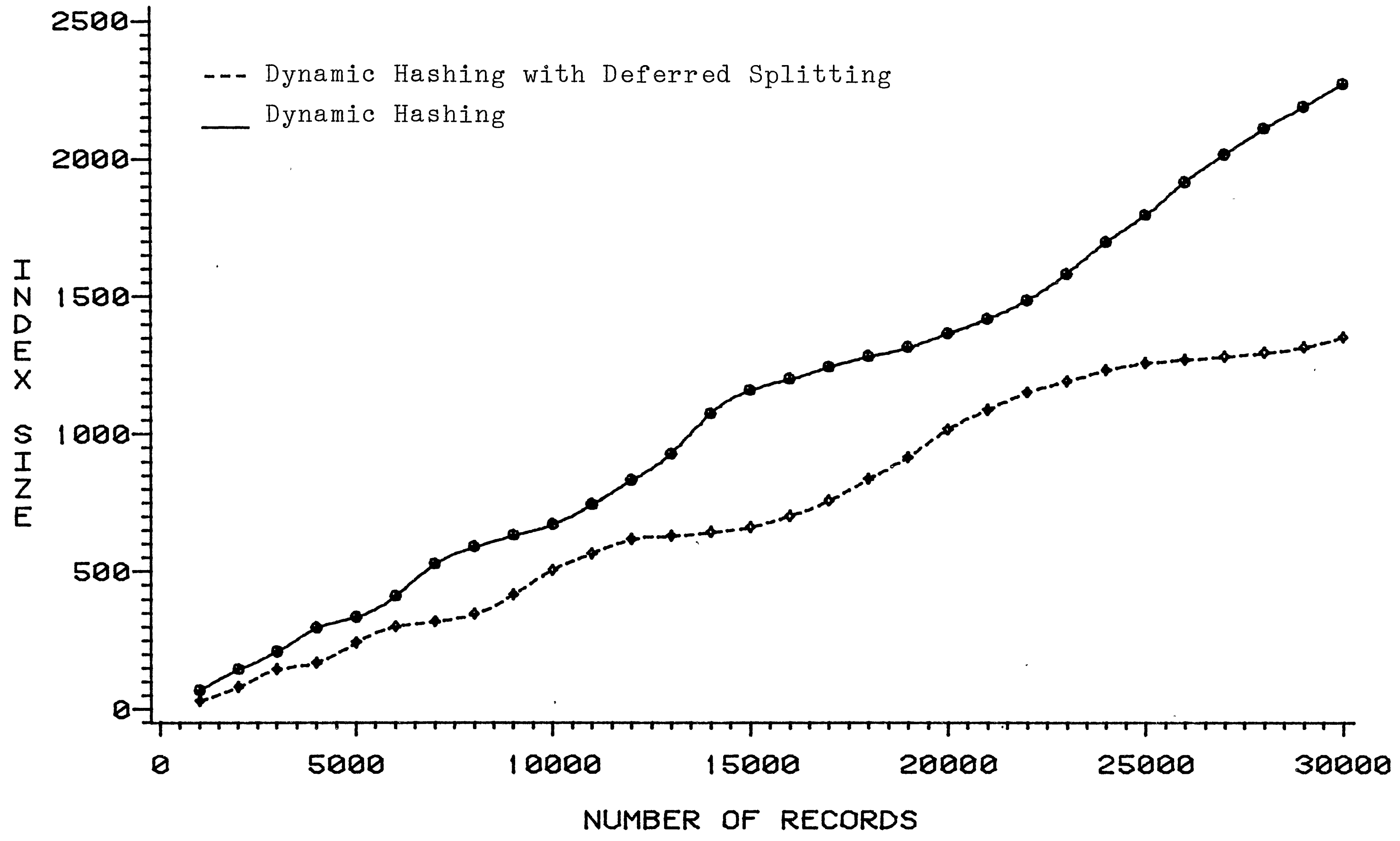


FIGURE 29. INDEX SIZE VS NUMBER OF RECORDS (BUCKET SIZE 40)

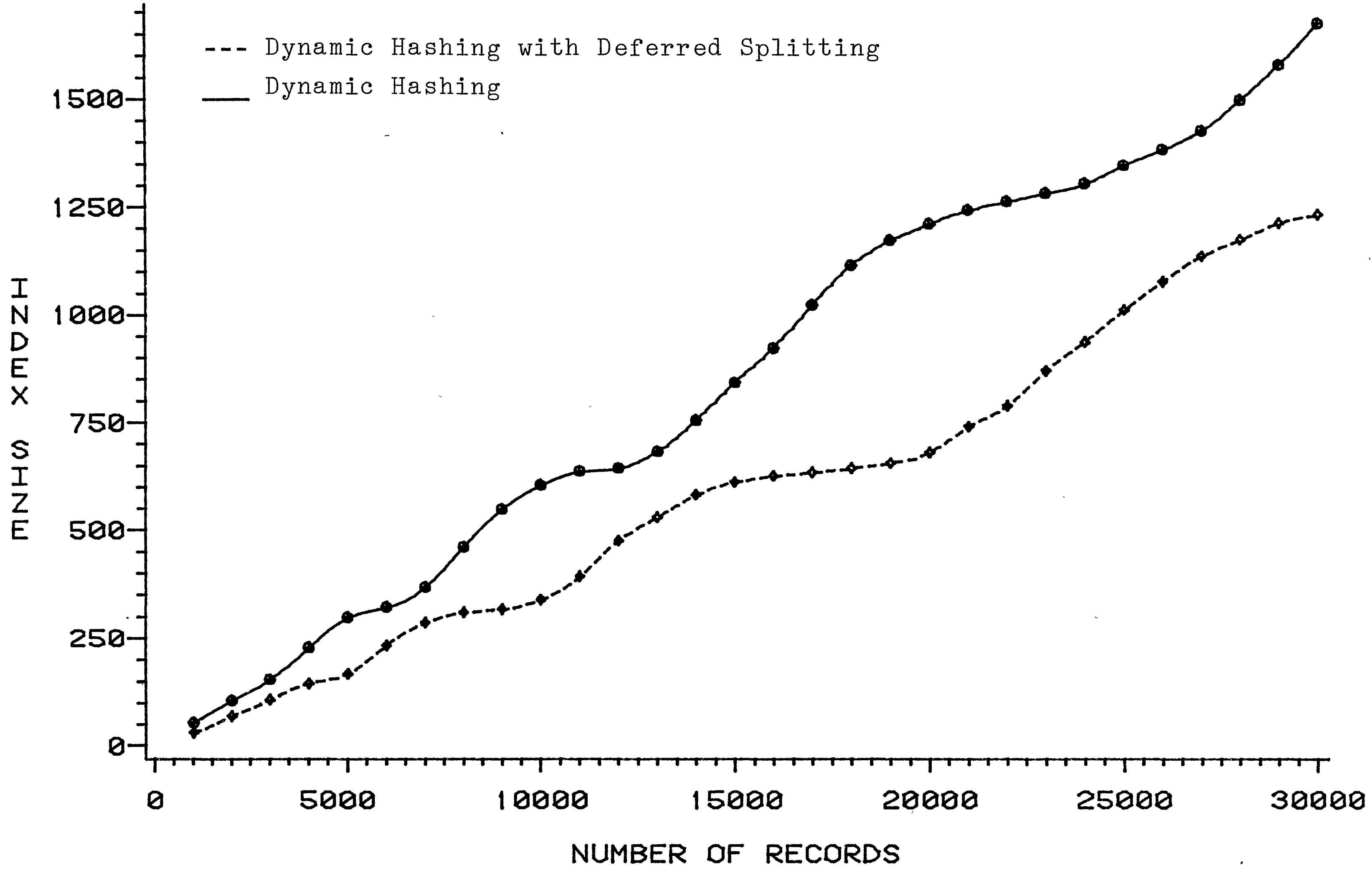


FIGURE 30. INDEX SIZE VS NUMBER OF RECORDS (BUCKET SIZE 50)

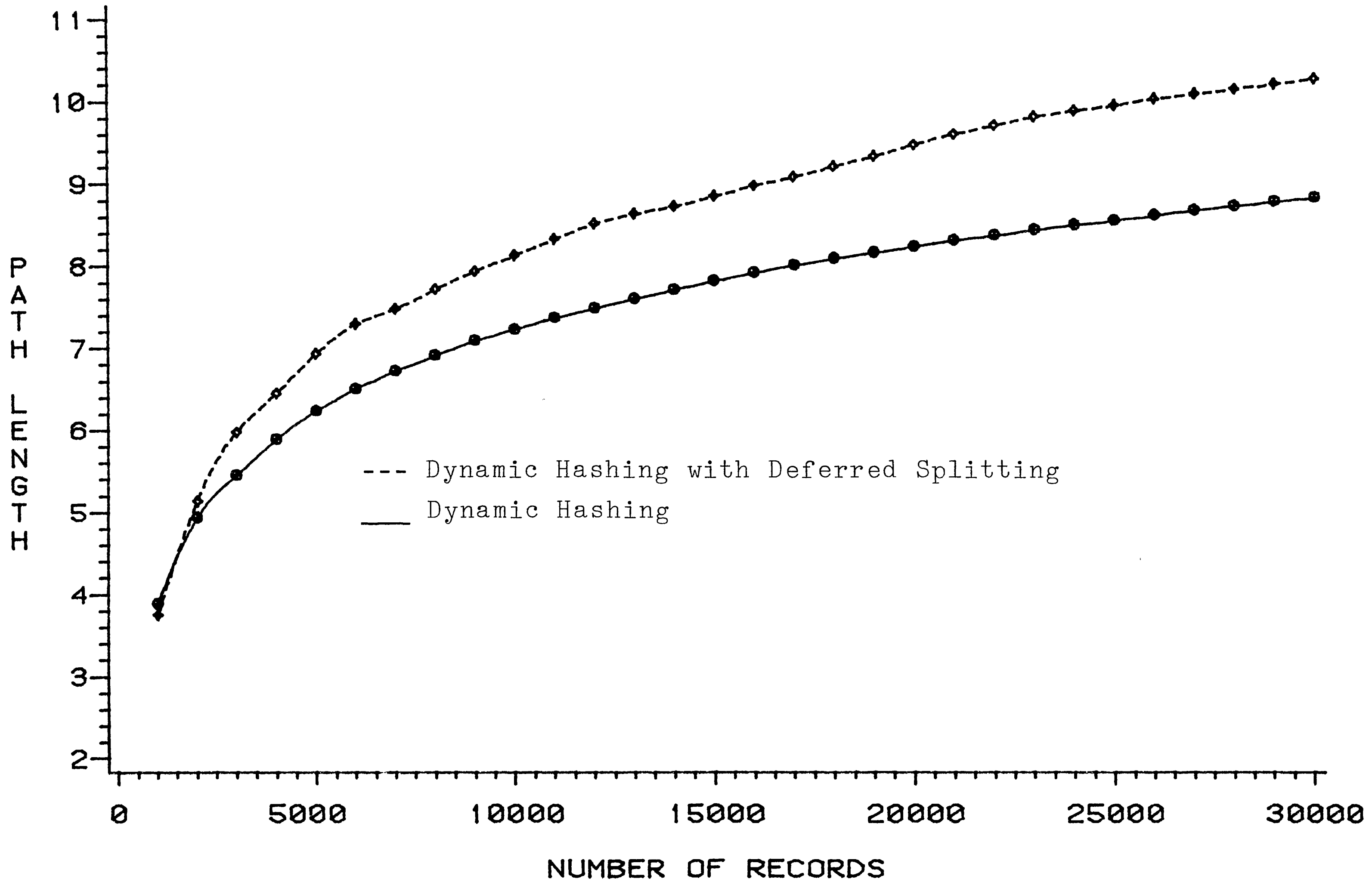


FIGURE 31. AVERAGE INDEX PATH LENGTHS (BUCKET SIZE 10)

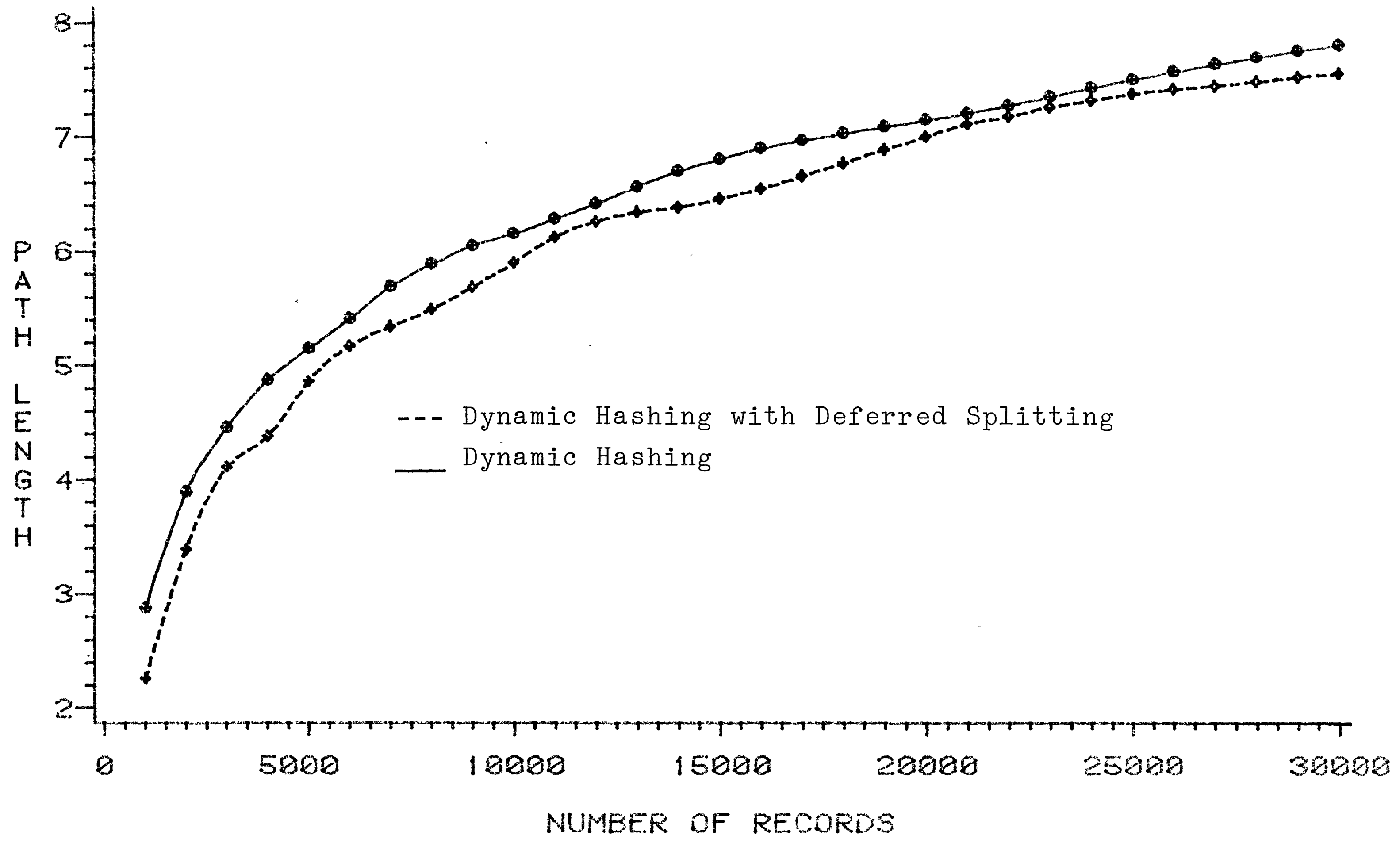


FIGURE 32. AVERAGE INDEX PATH LENGTHS (BUCKET SIZE 20)

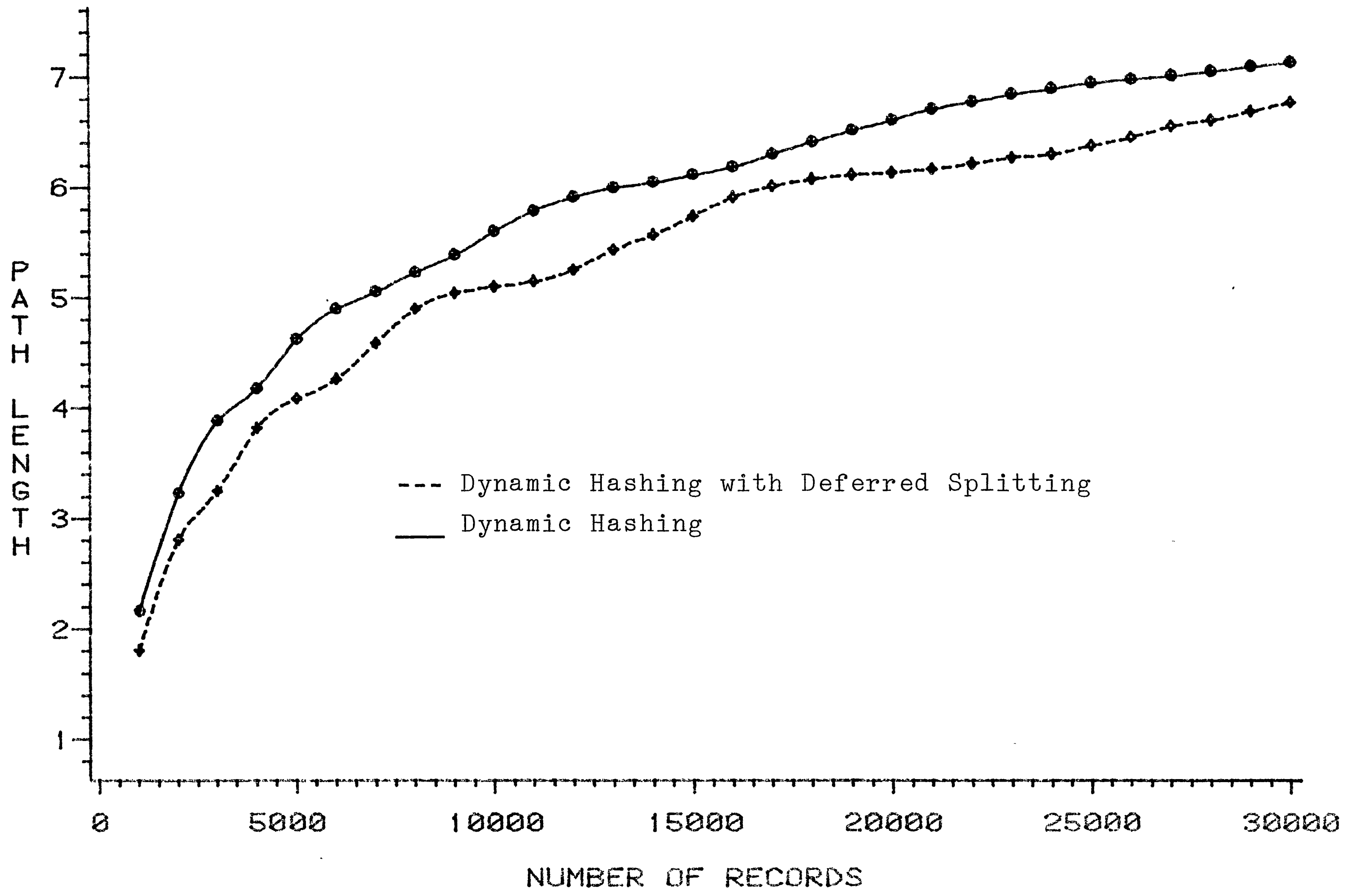


FIGURE 33. AVERAGE INDEX PATH LENGTHS (BUCKET SIZE 30)

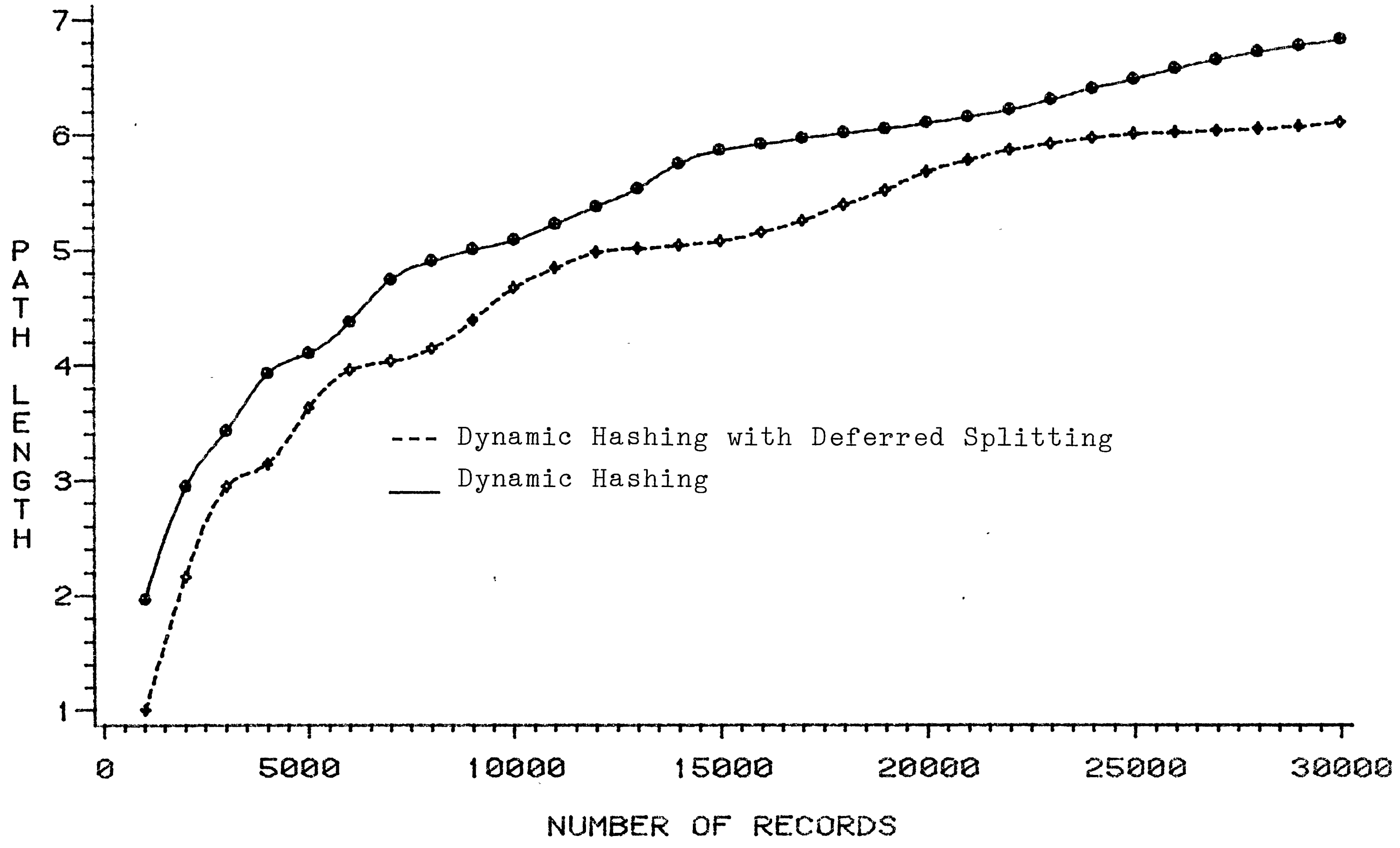


FIGURE 34. AVERAGE INDEX PATH LENGTHS (BUCKET SIZE 40)

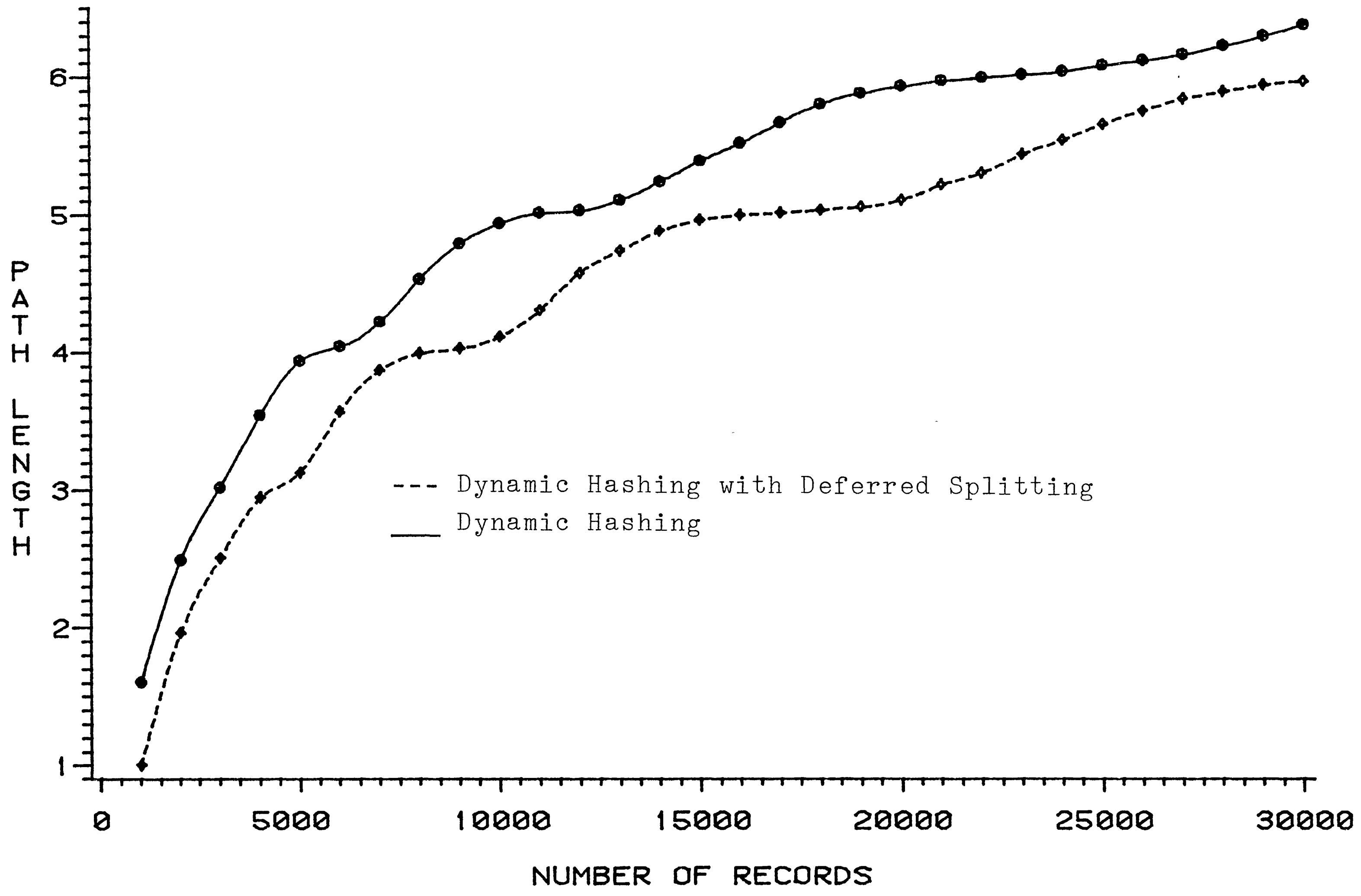


FIGURE 35. AVERAGE INDEX PATH LENGTHS (BUCKET SIZE 50)

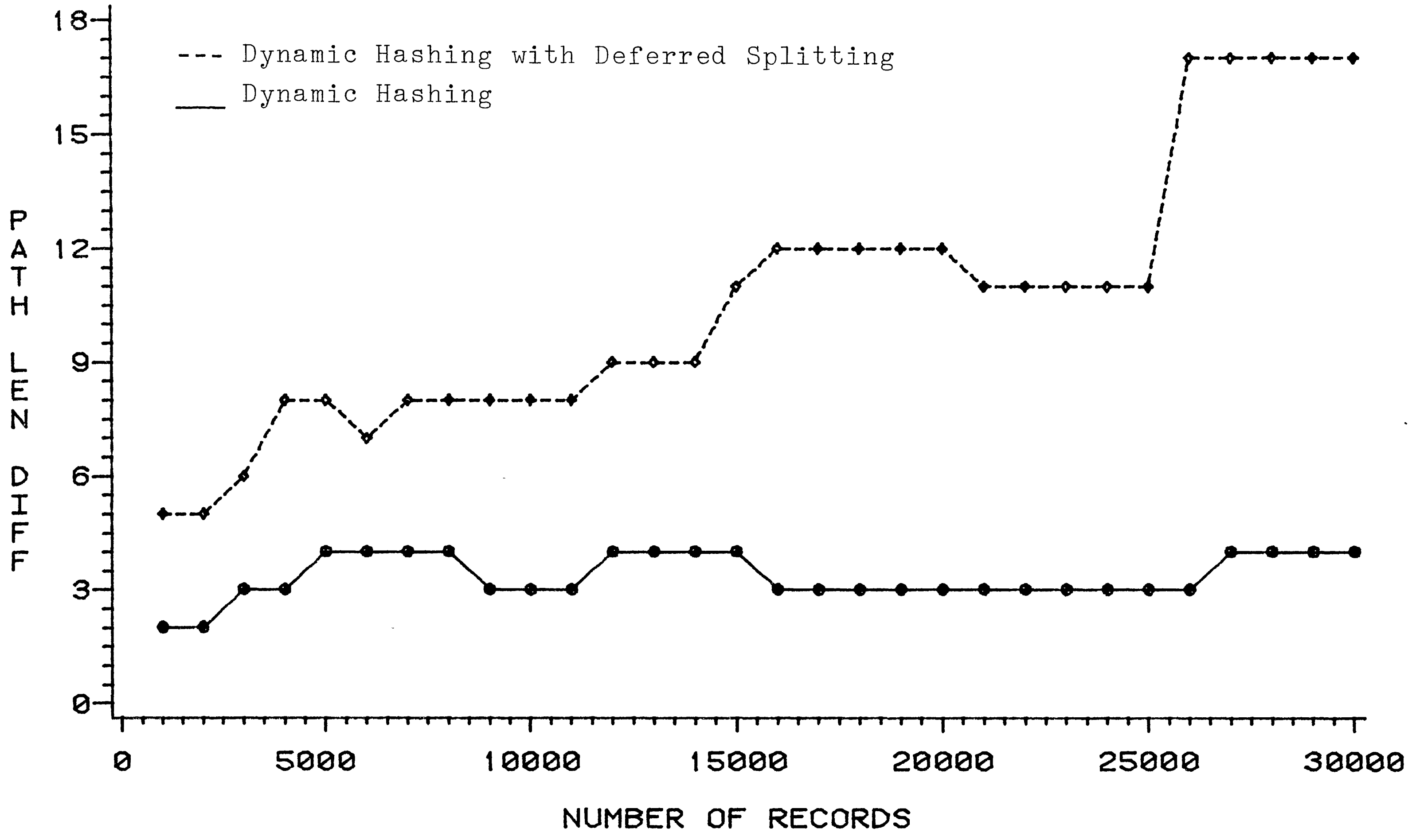


FIGURE 36. DIFFERENCE BETWEEN MAX AND MIN PATH LENGTHS (BUCKET SIZE 10)

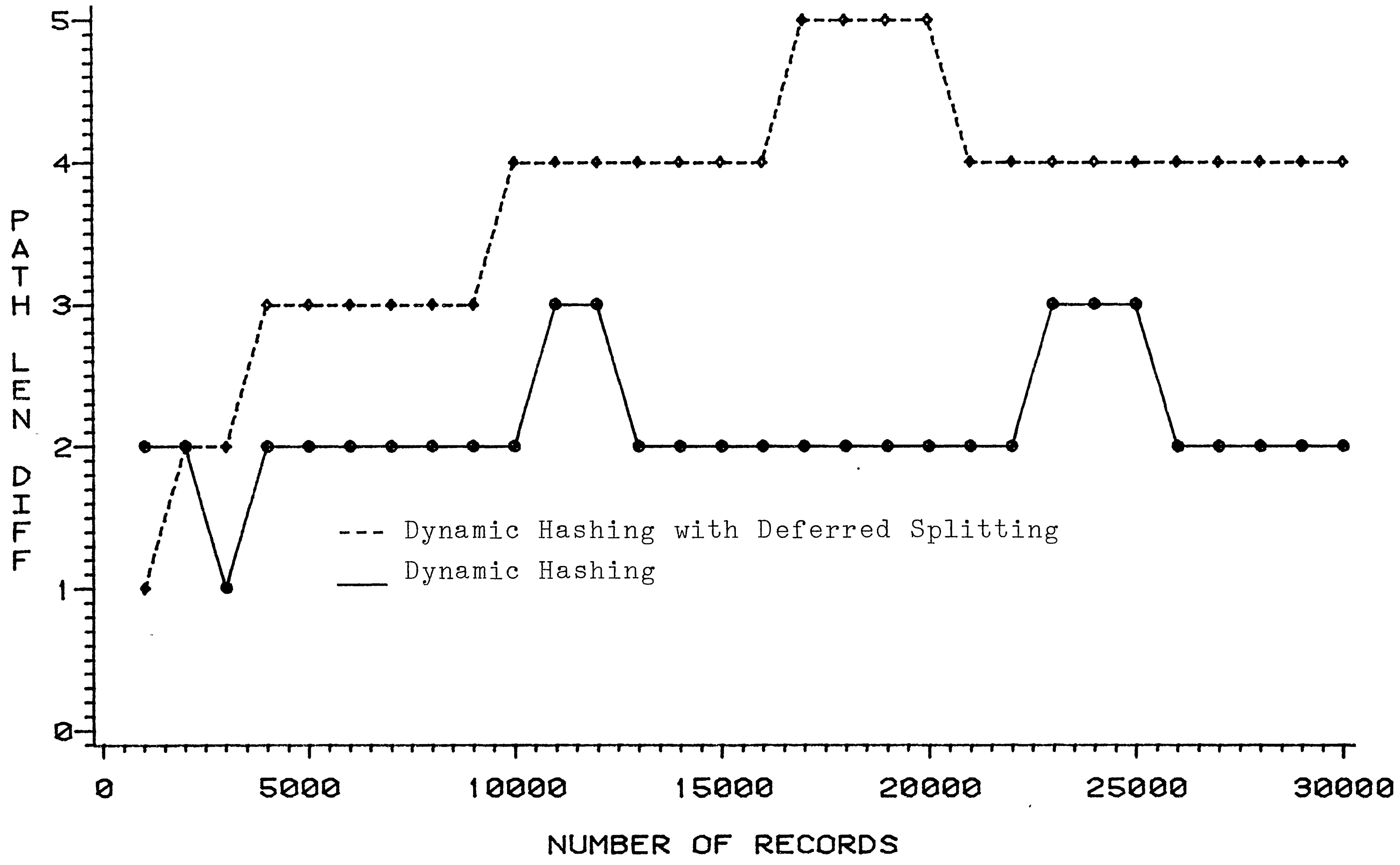


FIGURE 37. DIFFERENCE BETWEEN MAX AND MIN PATH LENGTHS (BUCKET SIZE 20)

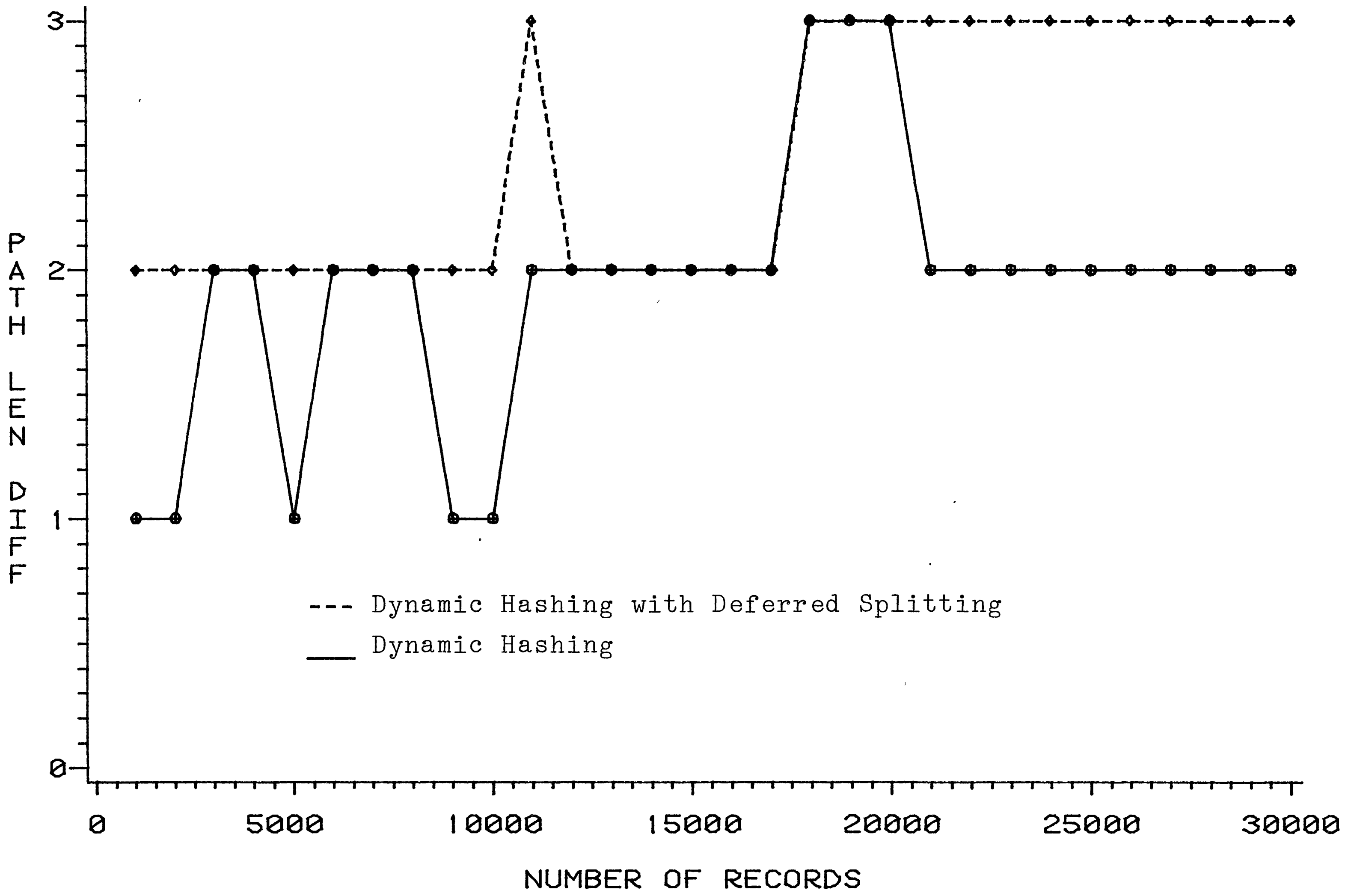


FIGURE 38. DIFFERENCE BETWEEN MAX AND MIN PATH LENGTHS (BUCKET SIZE 30)

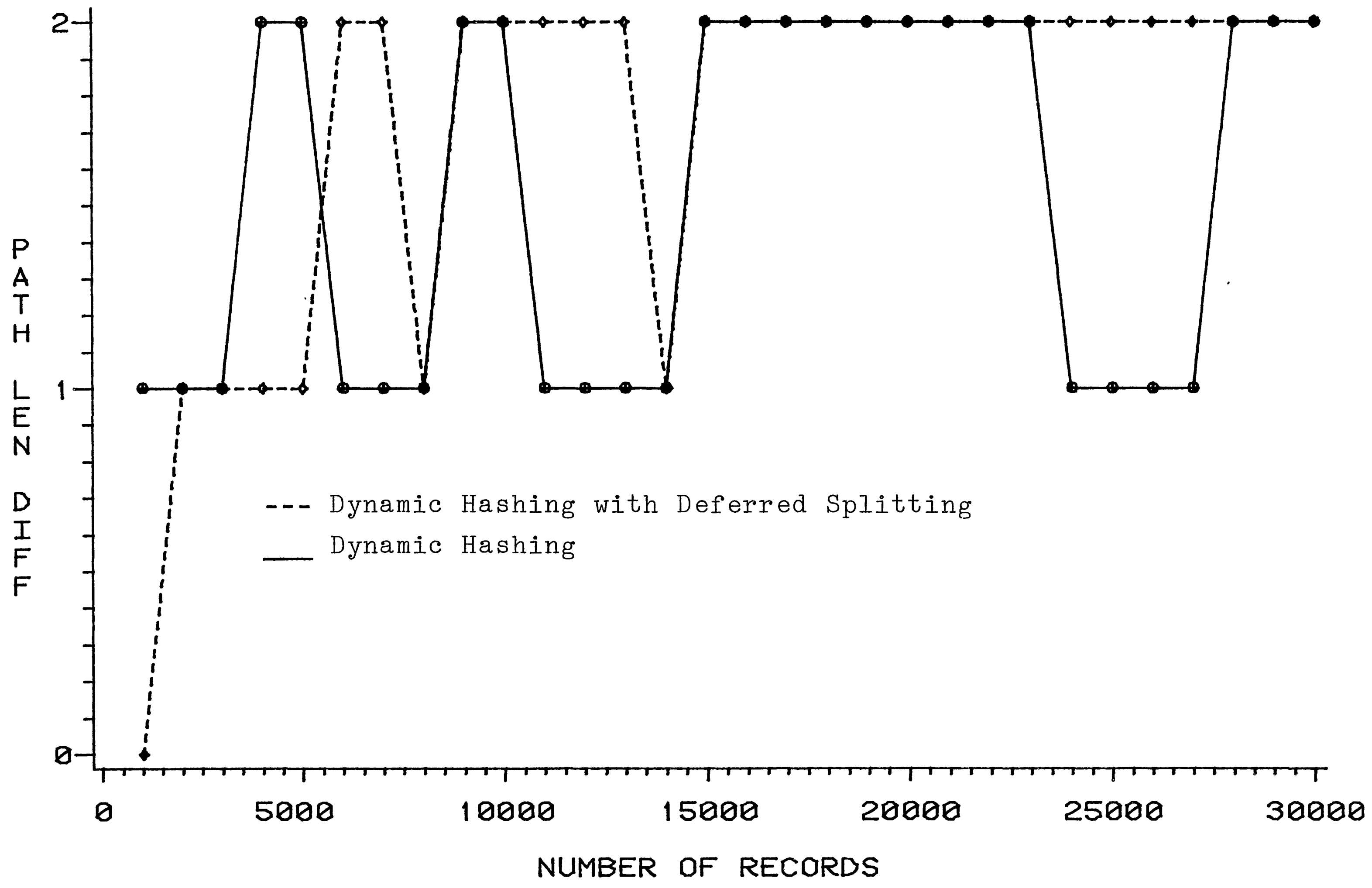


FIGURE 39. DIFFERENCE BETWEEN MAX AND MIN PATH LENGTHS (BUCKET SIZE 40)

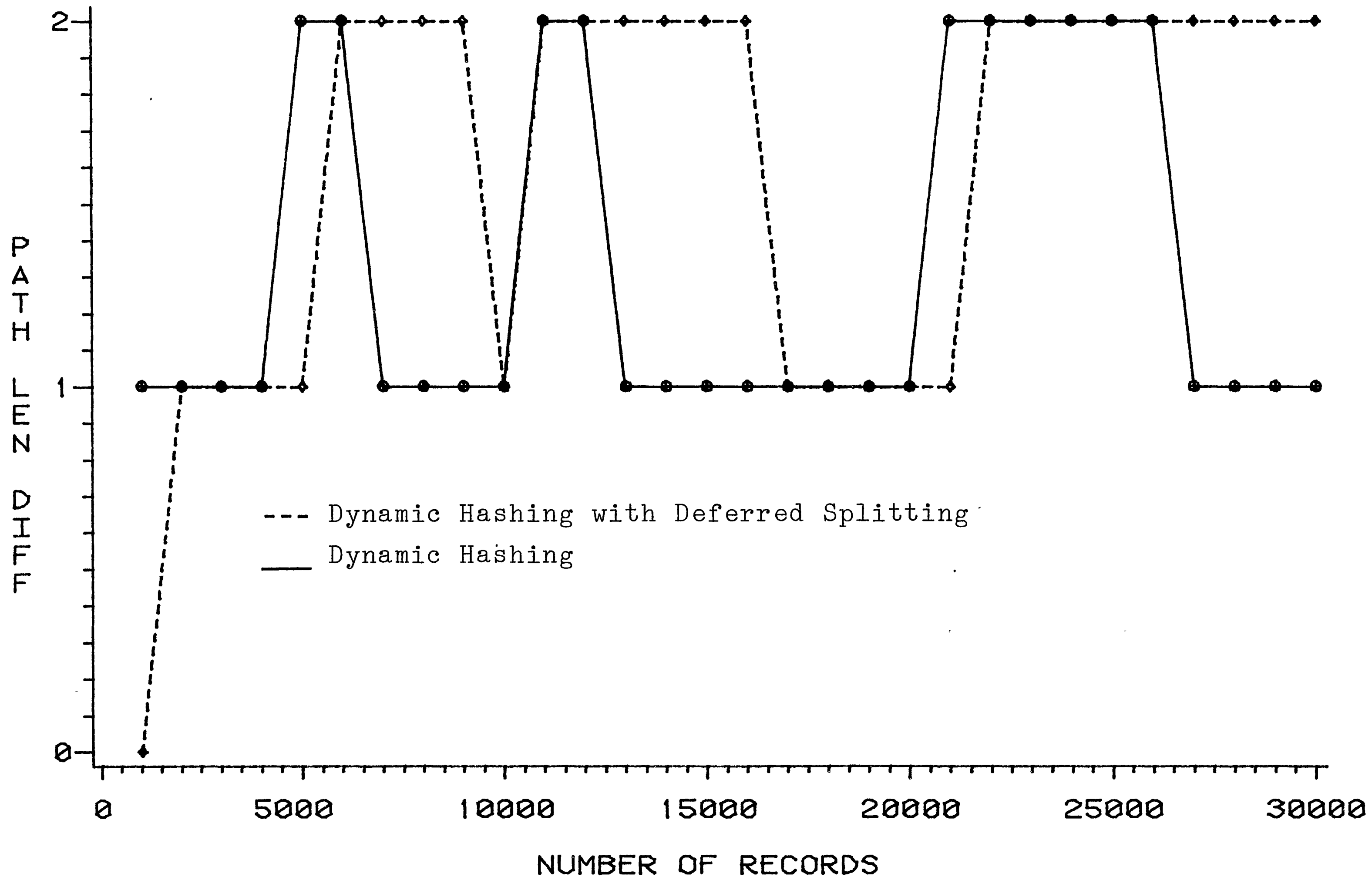


FIGURE 40. DIFFERENCE BETWEEN MAX AND MIN PATH LENGTHS (BUCKET SIZE 50)

VITA

Hu Chang

Candidate for the Degree of

Master of Science

Thesis: A STUDY OF DYNAMIC HASHING AND DYNAMIC HASHING WITH DEFERRED SPLITTING

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in Taiwan, Republic of China, September 7, 1958, the son of Chien-Yun and Chien-Yeh Chang. Married to Be-Ny Wu on July 17, 1983.

Education: Graduated from Chien-Kuo Hign School, Taiwan, R.O.C., in July, 1976; received Bachelor of Science degree in Applied Mathematics from Fu-Jen Catholic University, Taiwan, R.O.c., in May 1981; Completed requirements for Master of Science degree at Oklahoma State University in December, 1985.