ANALYSIS AND COMPARISON OF EXTENDIBLE HASHING AND B+ TREES ACCESS METHODS

By

HARSHAD D. PATEL

Bachelor of Science

Bombay University

Bombay, India

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, 1987

ANALYSIS AND COMPARISON OF EXTENDIBLE HASHING AND B+ TREES ACCESS METHODS

Thesis approved:

M J Folk
Thesis Adviser

Don Lyon

R. E. Hedrick

Norman N. Durham
Dean of the Graduate College

PREFACE

This thesis is a discussion and evaluation of both extendible hashing and B+ tree. The study includes a design and implementation under the UNIX system. Comparisons and analysis are made using empirical results.

I would like to express my sincere appreciation and gratitude to my major advisor Dr. Michael J. Folk, for his guidance, motivation, encouragement, and invaluable assistance. I am also thankful to Dr. G. E. Hedrick and Dr. K. M. George for serving on my graduate committee.

I extend a very special and sincere thanks to Dr. G. E. Hedrick for his unfailing confidence and support.

I am grateful to Mr. and Mrs. U. M. Patel and Mr. and Mr. and Mrs. H. M. Patel for their moral encouragement, direction, and financial support.

Finally, my wife, Maya, my parents, Mr. and Mrs. D. K. Patel and my family members deserve my deepest appreciations for their love, understanding, and sacrifices through out my studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Hashing is a well known technique for organizing direct access files. It provides fast direct access to data records stored either in main memory or an external devices such as disks.

Conventional hashing used as a file access technique has the advantages of being simple and fast. To access a record in a hash file, we first apply the hash function to the key which results in an address to the leaf page, where the record should be. The access time is constant if there is no overflow. Hashing is better in this aspect than sequential access and tree access.

However, if a file grows by very large factors, or if the record distribution over the available storage is not uniform, the number of overflow records may be large and therefore retrieval of records may be significantly slowed down. On the other hand, if the file shrinks, storage space is underutilized. Such situations require the file's rehashing, which is costly, especially in a multiuser environment[1].

Therefore, some novel hashing techniques have been

invented for files that grow and shrink dynamically. They include dynamic hashing[14], extendible hashing[1], virtual hashing[15], and bounded index exponential hashing[4]. With these techniques there are no overflow records. They also allow for the extensibility of the hash table and still guarantee efficiency of retrieval and update operations. Assume that the allocated secondary storage space is divided into buckets having a capacity of b records. When a record is to be inserted into a full bucket, the latter is split into two buckets among which the records are distributed. The "hash" function, which locates a given record provided with a unique key, is dynamically modified and the allocated storage space is dynamically adjusted to the number of records actually stored in the file.[12]

Dynamic hashing and extendible hashing employ an index to the data file. By using the hash function, a bucket associated with the given unique record's key can be found. Once the bucket's address has been found, retrieval is fast: only one access to secondary storage is required, since there are no overflow records. If the file grows steadily, this index, initially available in core memory, will eventually be partly stored in secondary storage. This will slow down searching and updating.

An extendible hashing index is implemented by means

of a buddy system partition. The index has 2**d entries (where, d is the depth of the directory), each of which points to the bucket in which the records are stored. Some of the entries may point to the same bucket. The d most significant bits of H(K), where H is a hash function and K is key, provide an address in the index. When a bucket overflows because of insertions, the corresponding block in the address space is halved and a new bucket is added. When a bucket gets underfilled because of deletions, the corresponding block is merged with its buddy. When the data volume grows, the partition's depth d eventually increases. When this happens, then the index doubles in size[1].

Bounded index exponential hashing, a new form of extendible hashing, combines elements of extendible hashing with elements of spiral storage. Unlike extendible hashing, in bounded index exponential hashing the index size does not increase. Rather, it is the data node that doubles in size so as to accommodate the overflow, instead of the node splitting into two nodes. Thus, multipage data nodes arise as the file grows in size. Each time a page within the data node overflows, the data node doubles again. The doubling, just as the splitting did, divides entries between pages on the basis of the value of the next digit of the key[4].

In contrast to the extendible hashing index, the dynamic hashing index, implemented by means of a tree structure, grows and shrinks more smoothly, but the index node size is larger than that of extendible hashing index entry. Each leaf of the tree contains a pointer to a bucket. When a bucket overflows, the corresponding index leaf becomes an internal node to which two new leaves are appended, the left leaf pointing toward the original bucket, the right leaf pointing toward a new bucket. When two brother buckets get underfilled, they are merged into one bucket and the corresponding index leaves are deleted, their father now pointing toward the resulting bucket[14].

The virtual hashing schemes proposed are similar to extendible hashing but do not employ any index. Retrieval of record then may require only one access to secondary storage. The price to be paid for this is a very low storage utilization, compared to the storage utilization provided by dynamic hashing and extendible hashing, which is in both cases approximately equal to ln 2 = 0.69.

In order to prevent virtual hashing storage utilization degradation, it is suggested that splitting of a bucket be deferred. However, the lower bound on storage utilization is still low, and deletion of a record is a rather complicated operation when the file shrinks[12].

By using extendible hashing, there are no more than

two page faults necessary to locate a key and its associated information even for files that are very large. Therefore, extendible hashing can be used in a large database systems[1].

A B+ tree is a variant of the B-tree data structure. B+ trees were designed to provide a way which is suited to both a random and sequential processing environment. A B+ tree consists of a set of records arranged in key order in a sequence set, coupled with a B-tree index set that provides rapid access to the block containing any particular key/record combination. In a B+ tree all the key and record information is contained in the sequence set[18].

The sequence set can be processed in a truly linear, sequential way, providing efficient access to records in order by key.

The only difference between a simple prefix B+ tree and a plain B+ tree is that the latter structure does not involve the use of prefixes as separators, while the simple prefix B+ tree builds an index set of shortest separators formed from key prefixes.

The simple prefix B+ tree builds separators in the index set that are smaller than the keys in sequence set. More separators can fit into a block. To obtain this compression and consequent increase in branching factor,

we must use an index set block structure that supports variable length fields[18].

The goal of this thesis is to implement extendible hashing and a B+ tree on a UNIX system and compare performance by examining empirical results. Analysis will include storage utilization, random access cost, sequential access cost, and insertion cost.

Chapter II and chapter III present descriptions of extendible hashing and B+ tree respectively. Chapter IV shows the implementation and logic design for different routines. Chapter V illustrates empirical results and discussion. A summary and conclusions are included in chapter VI.

CHAPTER II

EXTENDIBLE HASHING

An extendible hash file is a dynamic data structure that is an alternative to B-trees for use as a database index. In extendible hashing the user is guaranteed that no more than two page faults are necessary to locate a key and its associated information even for files that are very large.

Literature Review

Michel Scholl[12] claims that the expected average storage utilization for extendible hashing is 69.31 %. The formula to calculate the expected storage utilization for sufficiently large number of inputs is given as follows:

$$\text{storage utilization} = \frac{b+1}{b} \sum_{i=b/2+1}^{b} \frac{1}{i}$$

where b is block size.

For sufficiently large number of inputs, we have the following expected storage utilization of block sizes 2, 4, 8, 16, 32, and 64. For large block size, the limit

approaches in 2 = 69.31 %.

Block size	Expected storage utilization
2	75.0 %
4	72.9 %
8	71.4 %
16	70.4 %
32	69.9 %
64	69.8 %

Tamminen[7] claims that asymptotically extendible hashing storage utilization is poorer than that of linear hashing of Litwin[15]. The poor performance of extendible hashing is due to an excessive dependence of directory size on the existence of any random cluster points. The dependence is lessened if 'abnormal' clusters only cause the overflow of a page instead of doubling the directory.

Tamminen[6] studies the behavior of extendible hashing without an assumption of randomness, i.e. he represents some rough estimates of storage requirements and processing costs in case of non random pseudokeys.

The linear hashing of Litwin and Larson is extendible in the same sense as extendible hashing. It does not require a directory but must therefore routinely handle overflow. The method provides a good tradeoff between

expected storage utilization and access time and is efficient.

Performance aspects of extendible hashing have been thoroughly analyzed both by analytical models and by simulation[1,8]. These studies are based on the assumption of a perfect randomization method. The hash function h associates a random pseudokey K' with each key K. Then, whatever the distribution of keys, we can expect the pseudokeys to be distributed nearly uniformly: about half the pseudokeys have first bit 0; about a quarter start with 01, etc.

Fagin[1] and other analyze extendible hashing by analytical models and by simulation, and compare the performance of extendible hashing with B-trees for access time, insert time, and storage utilization.

Mendelson(9) derives performance measures for extendible hashing, and considers their implication on the physical database design. A complete characterization of the probability distribution of the directory size and depth is derived, and its implications on the design of the directory are studied. The expected input/output costs of various operations are derived, and the effects of varying physical design parameters on the expected average operating cost and on the expected volume are studied.

Ellis(3) studied extendible hashing and presents

extendible hashing for concurrent operations and distributed data.

Lomet(4) claims that the bounded index exponential hashing has the important advantages over the most of the other extendible hashing variants of both (i) providing random access to any record of a file in close to one disk access and (ii) having performance which does not vary with the file size. It is straightforward to implement and demands only a fixed and specifiable amount of main storage to achieve this performance. Its underlying physical disk storage is readily managed and record overflow is handled so as to insure that unsuccessful searches never take more than two accesses.

Radix Search Trees

Radix search trees are also known as digital search trees, or tries, which examine a key one digit or letter at a time, have long been known to provide potentially faster access than tree search schemes that are based on comparisons of entire keys. It is clear that radix search trees are naturally extendible. By tracing the path, data can be fetched. However, there are two major disadvantages of radix search trees: (i) they waste space by having to have redundant information, and (ii) they are

not balanced. The reason for this is that a radix search tree usually contains space for many keys not in the tree. Most of the time, the wasted memory space occurs at the nodes near the bottom of the tree. Therefore, in practice, radix search trees tend to be used only for small files. Extendible hashing exploits the speed of radix search trees without paying the penalty in memory space[1].

Classical hash tables are not extendible. Their sizes are intimately tied to the hash function used, and often must be determined before one knows how many records are to be placed in them. A high estimate of the number of records results in wasted space; a low estimate results in costly rehashing, that is, choice of a new table size, a new hash function, and relocation of all records.

Extendible hashing accomplishes two goals[1]:

1. It makes the hash tables extendible, so that they can adapt to dynamic files, and
2. It fills radix search trees uniformly, so that they remain balanced and can improve storage utilization.

In figure 1 a simple radix search tree over the alphabet (0,1) is presented.

Records are stored in the leaf nodes of the tree according to the leading bits of their keys. When a leaf

overflows, it is simply replaced by an internal node to which two leaves are attached.

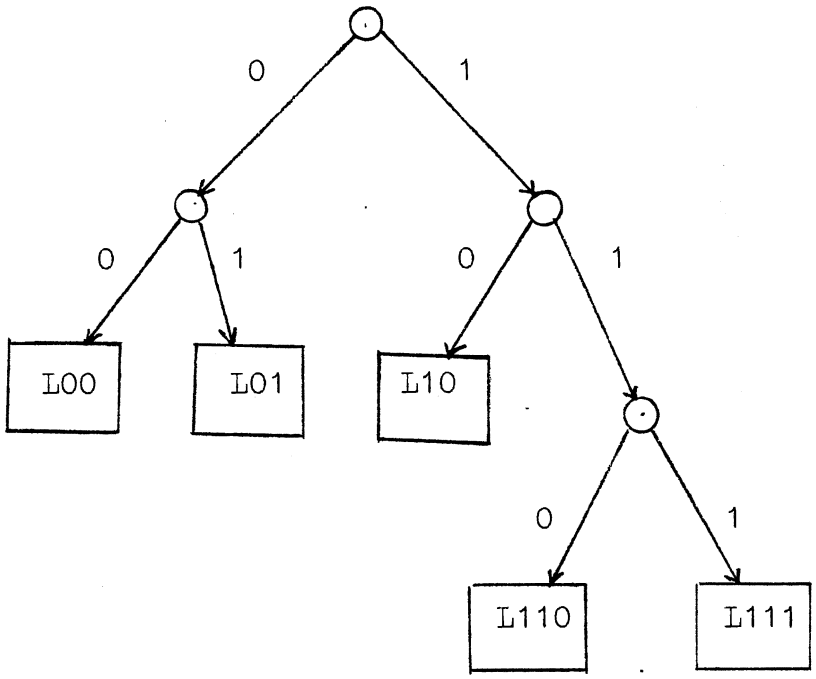

Figure 1: Radix search tree

Speed of access to a radix search tree can be increased if instead of comparing one digit of a key at a time, the top levels of the tree are flattened into an array of several pointers. The example in figure 2 is a modified version of figure 1.

If we can afford to waste some more storage for redundant information to improve access time, then the

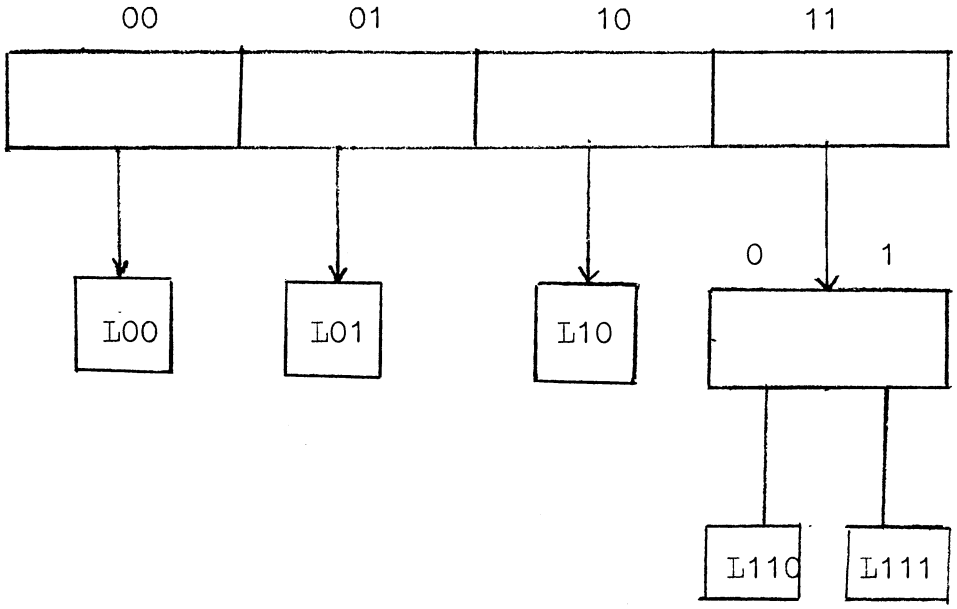

Figure 2: Radix search tree with the two top levels are compressed to one.

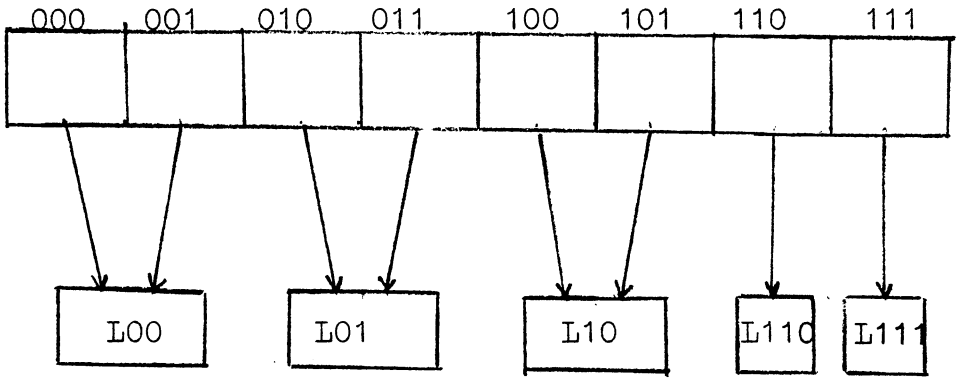

Figure 3: Degenerate radix search tree.

directory may be extended to a higher depth. This is shown in figure 3.

To search for a record in the file, we first select one of the pointers in the array according to the first three bits of the key. The pointers will lead us to a leaf page containing a record. Thus the access cost is constant.

Extending Hash Tables

In classical hashing, each entry of the directory (hash table) points to a leaf page of fixed size. This traditional method has a disadvantage of not allowing files to grow. When a leaf page overflows, we must use another leaf page to store the overflow records. Allowing overflow slows the search time. One way to eliminate overflow leaf pages is to rehash the records into more leaf pages. However, it will take O(n) time (where n is the number of records) to accomplish this task.

Extendible hashing uses the well-known "buddy system" for storage management. In figure 4, the hash function maps the key space s on to a large address space A. A partition P splits A into m blocks; each block has one leaf allocated for its use and the directory shows the

correspondence between blocks and leaves[1].

Assuming that P is defined by m+1 boundaries a(0),a(1),...., a(m) (m = 2**d), leaf Li contains all keys

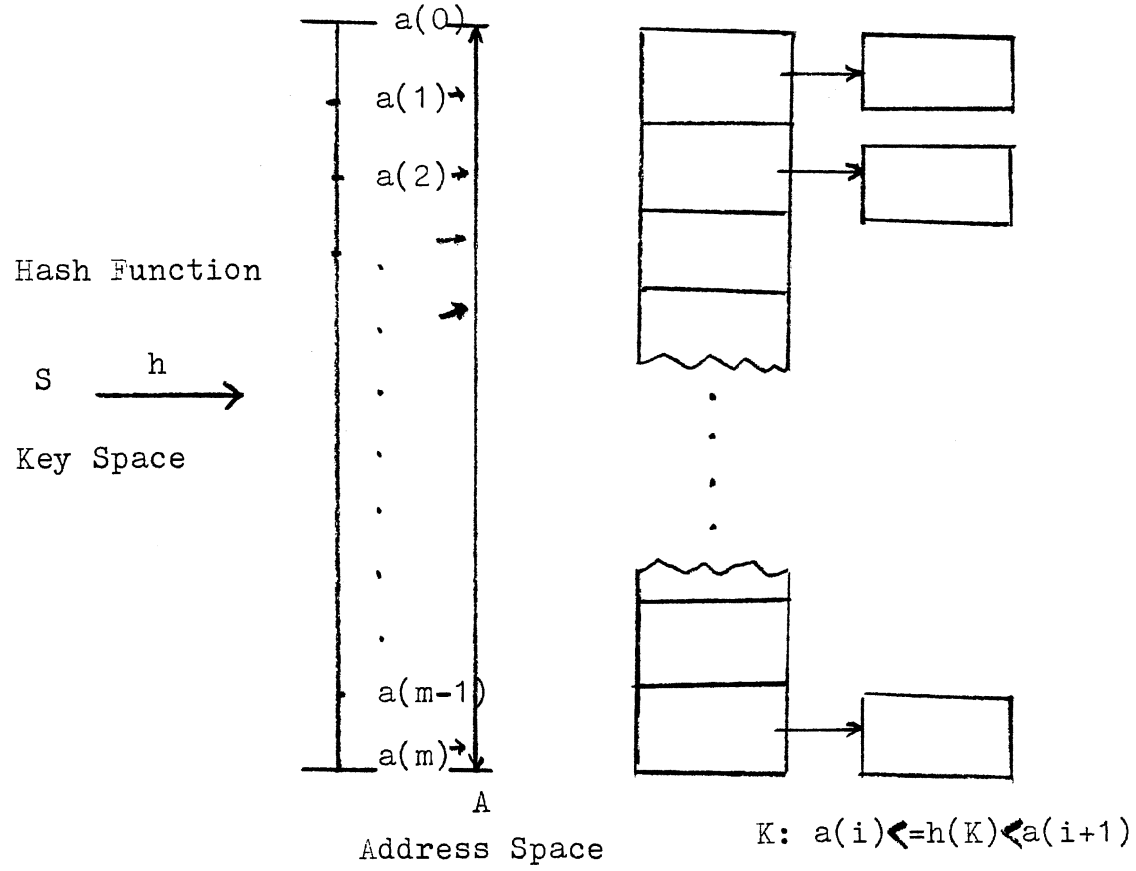

Figure 4: Hashing into a large address space.

K with a(i-1) <= h(K) < a(i). This scheme is flexible because if a leaf overflows, we can change the partition,

perhaps by as little as shifting one boundary a(i), and relocating only those keys that are affected by this shift. Therefore buddy system partitions have the advantage that when a leaf overflows, the corresponding block in the address space is halved, a new leaf is added, and only the keys in the halved block are affected. Halving any block of a buddy system partition leads to another such partition. When a block becomes underfilled because of deletions, and when its buddy has enough room, the two blocks can be merged easily into one block.

Let the depth d of a buddy system partition be the least integer such that each member of the buddy system partition is the union of some of 2**d equal sized intervals obtained by continued halvings. Thus, d is minimal such that for each block [a(l),a(l+1)] of the partition, (a(l)-a(l-1)) >= 2**(n -d). A directory with 2**d entries, some of which may point to the same page, allows one to take the d most significant bits of hash address h(K) as the index in the address space A = {0,.....,2**d - 1} of the directory. When the depth of a partition increases, the directory doubles in a size.

Figure 4, gives an example on how to hash into a large address space by using the buddy system.

A Specific Extendible Hashing Scheme

One extendible hashing scheme is described in this section. Its most important characteristic is its speed. Even for very large files, there are never more than two page faults necessary to locate a key together with its associated information.

Let h be a random hash function. If K is a key, then K' = h(K) is the pseudokey associated with K under the hash function h. Usually a pseudokey can be a fixed length such as 32 bits. The hash function h can be randomly selected from a universal class of hash function, as defined by Carter and Wegman. Then, whatever the distribution of keys, we can expect the pseudokeys to be distributed nearly uniformly: about half the pseudokeys have first bit 0; about a quarter start with 01, etc[1].

The data structure consists of two parts : a set of buckets and the directory.

The buckets(leaves) reside on secondary storage and contain keys and associated information.

Directory

The number of bits of the pseudokey actually used to index into the directory is called the depth d of the

directory and changes as the file grows or shrinks. The depth of the directory is an integer "header" associated with the directory. The array of pointers is of size 2**d. The directory contains an array of pointers to leaf pages. Figure 5 shows an example of a hash file with directory header = 2. Three pages of memory are allocated in this case.

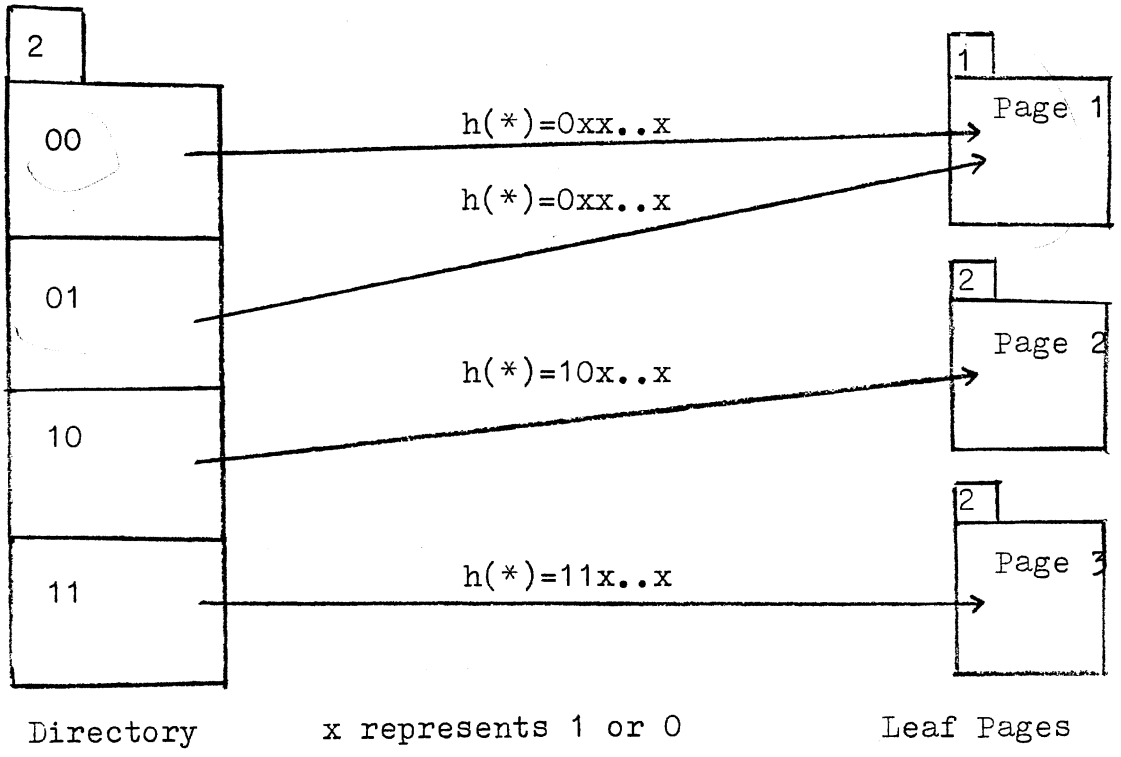

Figure 5: An example of extendible hashing with header = 2

In general, the pointers are laid out as follows. First, there is a pointer to a leaf that stores all keys K

for which the pseudokeys K' = h(K) starts with d consecutive zero bits. This is followed by a pointer for all keys whose pseudokeys have their first d bits equal to 01, and then a pointer to all keys whose pseudokeys begin with 10, etc., lexicographically. Thus altogether there are 2**d pointers and the final pointer is for all keys whose pseudokey begins with d consecutive ones.

To store a record with key equal to K0, h(K0) is calculated first, and its first d bits extracted. Thus d bits are used as an index to the pointer array(directory). The pointer in the corresponding element of the directory will point to a page where K0 should be.

Leaf page (Bucket)

Each leaf page has a local depth d' for the leaf page. The local depth d' may be less than or equal to the global(directory) depth d. The local depth d' indicates that the pseudokeys of the records it contains agree only in that number of bits. If d' < d, that means multiple directory entries will point to the same bucket. 2**(d - d') entries will point to that bucket. For example, the local depth of page 1 in figure 5, is 1. This means there are two pointers (which must be buddies) pointing to the page. When a page splits into two, the local depths of the

two split pages are increased by one.

As in figure 5, when the leaf page 1 overfills then it "splits" into two leaf pages, each with local depth two. All keys whose pseudokeys begin with 00 appear on the first of these pages, and all keys whose pseudokeys begins 01 appear on the other. The result is shown in figure 6. The header of the directory is the maximum of the local depths of all the leaf pages.

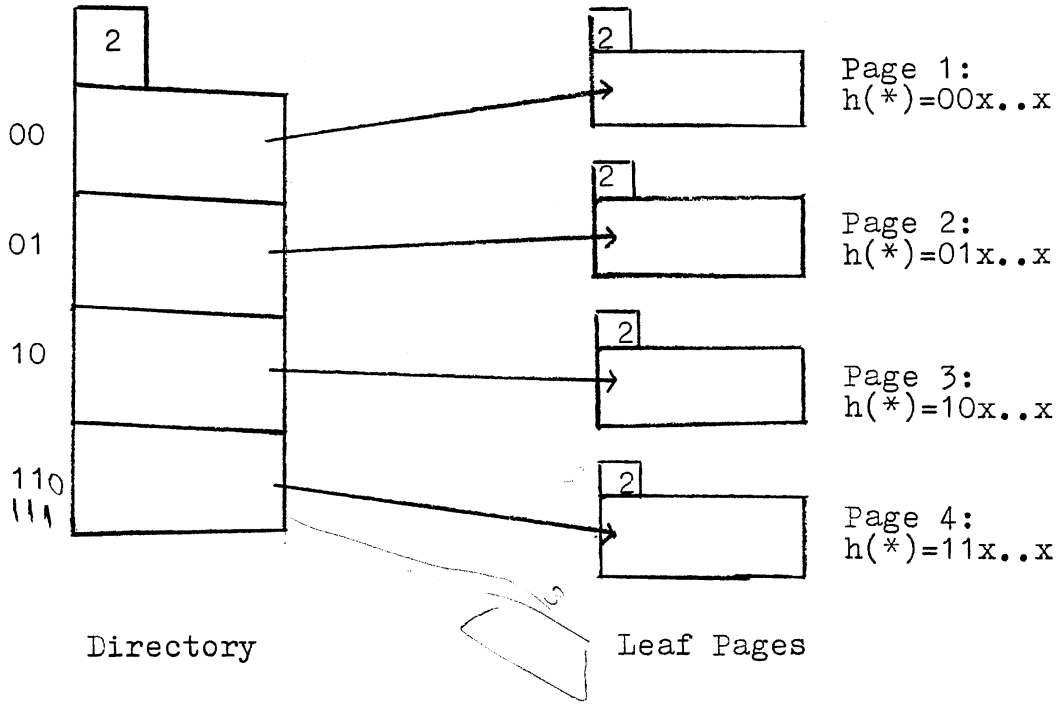

Figure 6: Modification of figure 5, after splitting of page 1.

What happens if a leaf page overfills and the local depth of the leaf page equals the depth (header) of the directory? The directory has to double its size so that

the header(d) can be increased by one. For example if page the example in figure 5 overflows, the directory (of

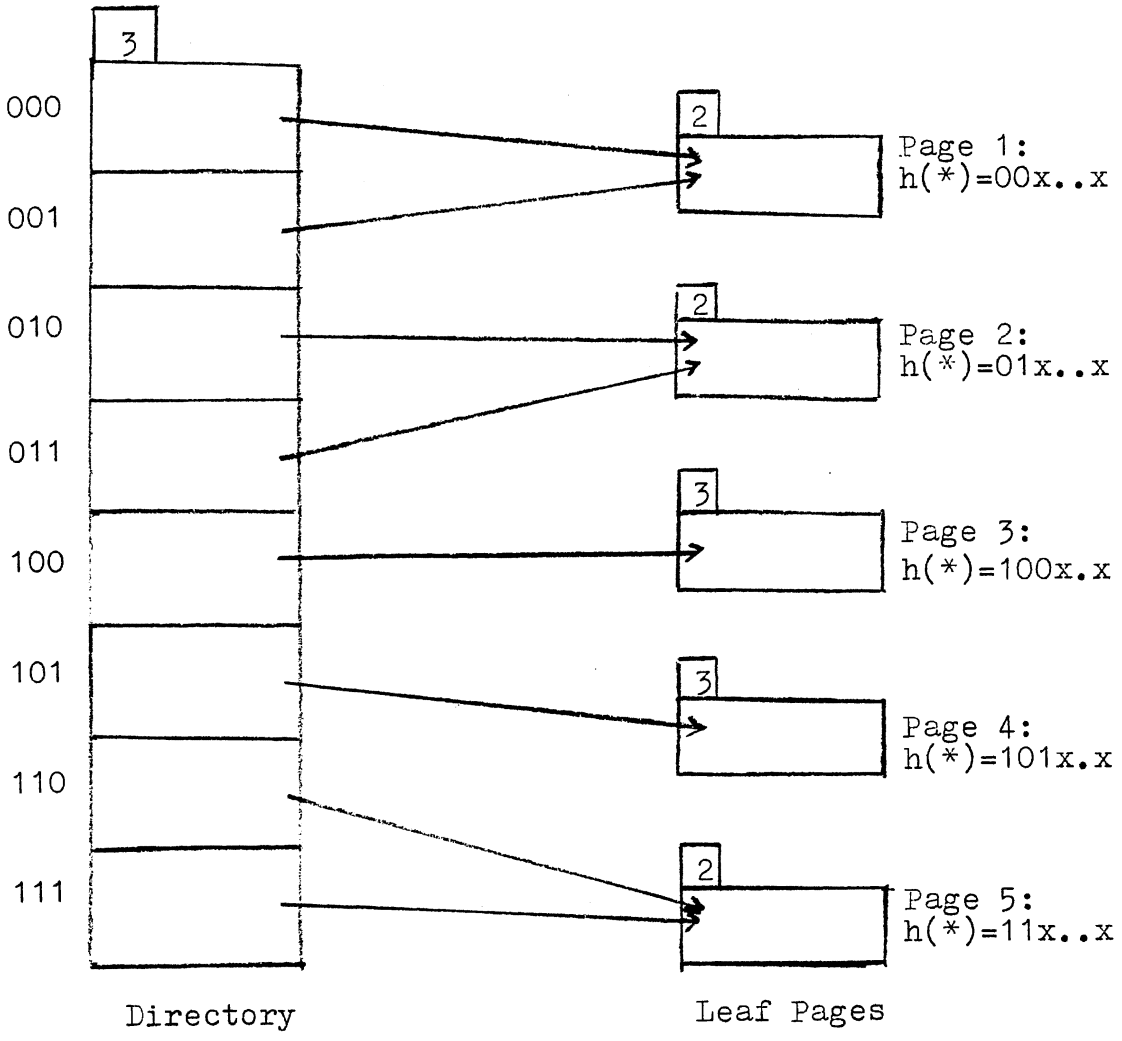

Figure 7: Modification of figure 6, after doubling directory.

pointers) must increase its size. Each pointer splits into two pointers pointing to the same page with the exception

of the overflow page. The overflow page splits into two pages according to the (d+1)st bit of the pseudokey. The global header is increased by one.

Figure 7, shows the result of splitting of page 3.

The process of doubling the directory size is not expensive because no leaf pages need to be touched (except for the leaf page that caused the split and its new sibling). This is essentially a one pass algorithm that proceeds by working from the bottom of the old directory up to the top of the old directory.

By using extendible hashing, there is at most one page fault in locating the appropriate directory page. Because the structure of the directory is an array, the location of each pointer can be determined by an easy address computation. Further, there is at most one page fault in obtaining the appropriate leaf page. So no more than two page faults are necessary to locate a key and its associated information. In cases where the directory is small, it can be kept in main memory.

The speed to implement a relational database management system by using extendible hashing is reasonable. If there are too many records that the directory has to be stored in secondary storage, then since the directory is stored continuously, it can be streamed into main memory in large blocks. If there are a

few million keys, when the directory doubles, and if the secondary storage device has a data transfer rate of around a million bytes per second (roughly comparable to that of IBM 3330 disk), then it is straightforward to estimate that the time involved in doubling the directory would be less than a second if there were 400 records per leaf page. Even in the extreme case of billion keys, the time involved in doubling the directory would be less than a minute[1].

The internal structure of the leaves is independent of the relationship between the pages. If deletions form such a large proportion of the operations of an application then space will be saved by coalescing pages. This can be accomplished by keeping in the directory the number of entries on each page as well as the pointer to the page. Then at each deletion, the total number of entries in the page deleted from together with an appropriate sibling page can be checked without any extra accesses. However, this additional complexity will probably not be justified for those applications where we can expect new growth to rapidly replace any deletions.[1]

The scheme shows that extendible hashing provides a dynamic file structure that has a fast (constant) access time and efficient implementation.

Sequentiality

Hashing usually cannot support sequential processing of a file according to the natural order on the keys. Sequential processing requires sorting, an O(nlogn) operation which makes fast random access useless.

Sequentiality means two things. In a weak sense it means that the entire set of keys (and corresponding data) can be processed efficiently one at a time, where each page of keys is referenced only once. Sequentiality in the usual stronger sense means that the order of sequential processing coincides with the natural order defined on the space of keys. It is possible to store the set of keys within each leaf in a natural order, so that sequential processing in natural order can be obtained for the cost of linking all leaves, as opposed to sorting the entire file.

CHAPTER III

B+ TREES

A new approach to external searching by means of multiway branching was proposed in 1970 by Bayer and McCreight[22]. They called this new kind of data structure a B-tree.

B+ trees are probably the most widely used variant of the original B-tree. VSAM, IBM's general purpose B-tree based organization and access method, is a well known example of using a B+ tree approach.

Motivation of B+ trees

The conventional B-tree is good for indexing dynamic random access files, but it has an apparent weakness in the case that required sequential processing. To extract all the keys in order a simple preorder traversal can be used, but a significant amount of primary memory may be required to stack all the nodes along a path from the root to the leaf to avoid reading these nodes twice. Additionally, processing a "find next" operation may

require tracing a path through several nodes before reaching the desired key[21].

B+ trees were designed to remove these weakness and provide a way that is suited to both a random and sequential processing environment.

Characteristics of B+ trees

The characteristics of B+ trees are summarized by the following:

1. All keys of B+ trees reside in leaves (bottom level).
2. Only the keys in the bottom level are associated with data records.
3. Each leaf node of B+ trees has a link field which points to the next leaf node to the right, except the rightmost leaf.
4. The index set has the structure of a B-tree.

A B+ trees consists of two independent parts: (i) an index set and (ii) a sequence set. The structure of a B+ trees is illustrated in figure 8.

The index set consists of separators that provide information about the boundaries between the blocks in the sequence set of a B+ tree. The index set can be used to locate the block in the sequence set that contains the

record corresponding to a certain key.

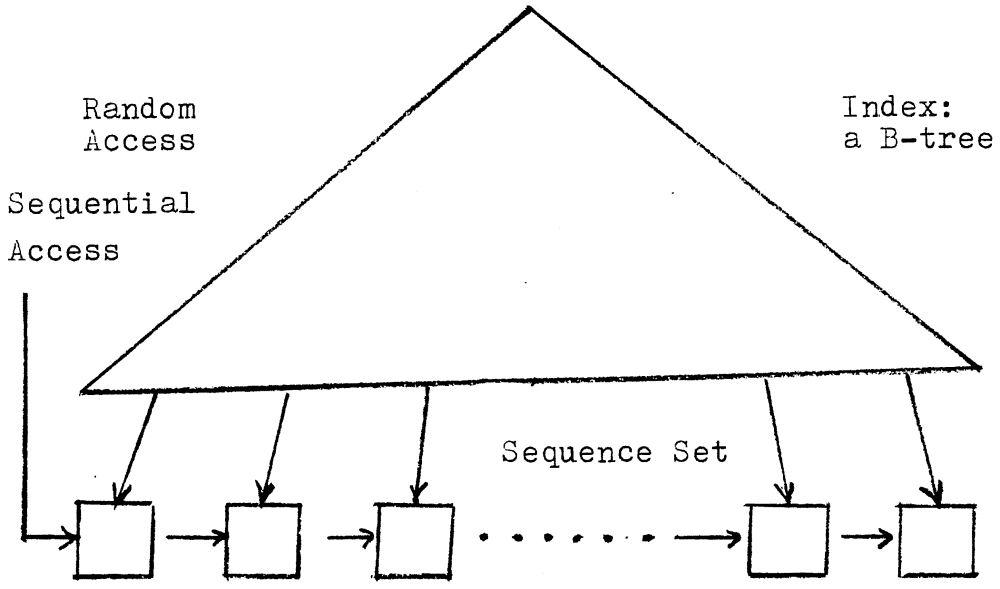

Figure 8: A structure of B+ trees.

The sequence set is the base level of an indexed sequential file structure. It contains all the records in the file in a natural order.

A search in a B+ trees starts at the root but it is confirmed only when a matching key is found at the leaf level. Sequential processing begins at the leftmost leaf and is aided by following the horizontal links across the leaves. The key values in a certain range can be identified by locating the lower limit of the range in the bottom level and processing sequentially until the key

value exceeds the higher limit of the range.

Bayer and Unterauer[21] propose a refined structure, the simple prefix B+ trees, which stores shortest separators or prefixes of the keys rather than copies of the actual keys, in the index part of a B+ trees. The major advantage of a simple prefix B+ trees is that it decreases access time as well as saves space. According to Bayer and Unterauer's[21] experimental results, for trees having between 400 and 800 pages, simple prefix B+ trees require 20-25 percent fewer disk accesses than a B+ trees.

CHAPTER IV

ANALYSIS AND COMPARISONS

Extendible hashing and B+ tree has been implemented under UNIX in the C programming language.

In order to analyze and compare the performance of extendible hashing with that of a B+ tree implementation, the following four performance factors were measured:

1. Random access cost (in terms of page faults);
2. Sequential access cost (in terms of number of pages);
3. Insertion cost (in terms of page faults);
4. Space utilization.

These measures are examined as functions of the following three database and system parameters:

a. Database size (in terms of number of records);
b. Page size (in terms of number of records);
c. Buffer size (number of pages resident in primary storage at a time).

a. Database size: The database sizes range from 1000 to 30000 records with an interval of 1000. Thirty thousand random alphabets keys were chosen as record identifier

or keys.

b. Page size: The page size is the maximum number of records that can reside in a single page; i.e., page capacity. The page sizes range from 10 to 70 records with an interval of 10.

c. Buffer size: This parameter is used to count the number of page faults. Buffer sizes range from 10 to 70 pages with an interval of 10.

Extendible Hashing

The structure used to implement extendible hashing is shown in figure 5 (chapter II).

There are two main structures used:

1. Directory
2. Pages

1. Directory: Each entry in the directory has the address of the particular page. Some entries in the directory might have the same value. If d is the depth of the directory then the total number of entries in the directory is 2**d.
2. Page: A binary tree is used as the internal structure of the page to store the keys in order and to get sequential access in natural order. The page has fixed

capacity (or page size) in terms of number of records. Each page is linked to the next page, except the last page.

Initially, starting with a directory depth(d) of 1, there are two entries in the directory (2**1 = 2) and two pages. Thirty thousand keys were generated using the system function random(). All keys were formed using capital letters from A to Z. Each character in a key was converted to a 5-bit binary number ranging from 00000(A) to 11001(Z). These binary strings were concatenated to form the pseudokey. For example the key AY would be converted to 0000011000.

To calculate the address in the directory for the key, the leftmost d bits were used from the pseudokey. d was the current directory depth.

For example, if d is 7 then the leftmost 7 bits are used, and the address is 0000011. So entry 3 in the directory would have the page address for key AY. Since this result directly gives the address of the page in the directory, there is no need to store any keys in the directory or index.

LOGIC DESIGN

Search: The basic design of the search routine for

extendible hashing is :

1. Read the key, K.
2. Determine the entry in the directory using the above method.
3. Follow the pointer to a page P.
4. Search the binary tree in page P for key K.
5. If the key K is found then return successful else return unsuccessful.

Insertion: The insertion routine for extendible hashing is:

1. Apply all five steps of search, using key K.
2. If the search is successful then return.
3. If by inserting key K on page P, we would exceed our page capacity(page size), then go to step 7.
4. Otherwise, insert the key K in a binary tree in page P.
5. Increment the counter of number of records in page P.
6. If key K has been successfully inserted then return else print Error and exit.
7. At this point, we know there is not sufficient space on page P. Obtain new page P1.
8. Obtain a temporary area Q to store all records appeared on page P, along with the new record.

9. Set the local depth of each P and P1 to d'+1, where d' is the old local depth of P.
10. After storing all records from page P erase all records from page P.
11. If the new local depth of P is bigger than the current directory depth d then do the following:
    a. Increase the depth d of the directory by one.
    b. Double the size of directory, and update the pointers in obvious manner.
    c. Set the count for number of records on page P and P1 to zero.
12. Insert all records one at a time from temporary area Q.

B+ TREES

The structure used to implement B+ tree is shown in figure 8 (chapter III).

There are two main parts of a B+ tree structure:

1. Index set
2. Sequence set

1. Index set: The index set consists of separators that provide information about the boundaries between the blocks in the sequence set of a B+ tree.

2. Sequence set: The sequence set is the base level of a B+

tree. It contains all of the records in the file. The Sequence set is made up of different pages that are linked to the next page except the rightmost page.

Logic design

Search: The basic design of the search routine for B+ tree is:

1. Read the key K.
2. Start from the root of index set and follow the pointer, in the index set until the leaf page P is found according to following rule:

Relation of search key and separator			Decision
Key	<	Separator	Go left
Key	=	Separator	Go right
Key	>	Separator	Go right

3. As the internal structure of page is binary tree, search the binary tree of page P for the key K.
4. If the key, K is found then return "successful" else return "unsuccessful".

Insertion: The basic design of the insertion routine for B+ tree is:

1. Apply all four steps of above search routine.
2. If the search was successful then return.
3. If by inserting key K on page P, we would exceed

our capacity(page size), then go to step 7.

4. Otherwise insert the key in a binary tree in page P.
5. Increment the counter of number of records on page P.
6. If key K has been successfully inserted, then return, else print error and exit.
7. At this point, we know there is not sufficient space on page P, obtain a new page P1.
8. Obtain a temporary area Q to store all records that appeared on page P, along with the new record.
9. Promote the root key R of binary tree in page P to the parent index page PI. By inserting the key R to the index page PI of page P, if count of number of keys on page PI will exceed the capacity of page PI then go to step 11.
10. Insert the key R to the index page PI at the appropriate position and go to step 13.
11. Copy the index page PI to the temporary area QI with the promoted index key R at the appropriate position. Obtain a new index page I. If PI is the root page then go to step 12. Promote the middle key R from the temporary area QI to the parent index page NI of page PI. Copy the first half of

QI to PI and erase the rest of the information from PI. Copy the second half without the middle key to the page I and update the pointers in the obvious manner. If the page NI will overflow then NI becomes PI and repeat the step 11, else go to step 13.

12. Obtain a new index page RI. Copy the middle key of QI to the first position in RI. RI is the new root page. Copy the first half of QI to the page PI and the other half of QI, without the middle key to the page I.
13. Update the pointers in obvious manner and insert all records one at a time from temporary area Q.

Page faults

There is a buffer in primary memory that can hold b pages. Whenever we require a page not in the buffer, there will be a page fault. A least recently used (LRU) page management algorithm is used. For extendible hashing directory page faults and for B+ tree index page faults are also considered.

The number of entries (n) in the directory of extendible hashing can reside on a single page given as follows:

n = page_size/size of pointer.

Only the size of the pointer is considered because directory contains only pointers.

Analysis

In this section some analytical results concerning the number of leaf pages and number of page faults for accessing a record will be derived.

Let us postulate a paged memory, with p equal to the maximum number of records that can reside in a single page and pb equal to the page size in bytes. There is a buffer in a primary memory that can hold b pages, and, whenever a required page is not in the buffer, there will be a page fault. The total number of records will be n. The parameters n, p, and b are common to both extendible hashing and B+ tree.

Let UT(n) be the average occupancy in entries divided by p. UT(n) will of course be different for extendible hashing and the B+ tree. It is assumed that each page has exactly UT(n)*p entries.

Extendible Hashing

Let lp be the number of leaf pages:

$$lp = \lceil n/(UT(n)*p) \rceil$$

Number of directory entries (nd) = 2**d

where d is directory depth.

If dp is the number of directory pages,

$$dp = \lceil (nd * \text{size of pointer})/pb \rceil .$$

Now we can compute the probabilities dpf (page fault referencing directory page) and lpf (page fault referencing leaf page):

$$dpf = \max(0, 1-b/dp)$$

and

$$lpf = \max(0, 1-(\max(1,b-dp))/l).$$

Finally, we have our approximation for expected random access cost in terms of page faults for extendible hashing :

random access cost = dpf + lpf.

B+ tree

Let db is the depth of the B+ tree index set:

$$db = 1 + \log_{\lceil m/2 \rceil}(n+1)/2.$$

where, m is the order of B-tree in index set.

So, $m = p + 1$;

If lp is the number of leaf pages,

$$lp = \lceil n/(UT(n)*p) \rceil .$$

The number of keys in the index (nl) = lp - 1.

Let lUT(nl) be the average occupancy of index page in terms of number of keys divided by p.

If nlp is the number of index pages,

$$nlp = \lceil nl/(lUT(nl)*p) \rceil .$$

Now we can compute the probabilities lpf (page faults referencing index page) and lpf (page fault referencing leaf page):

$$lpf = min(db, max(0, db-b/nlp))$$

and

lpf = max(0, 1-(max(1,b-nlp))/l).

Finally, the approximation for expected random access cost in terms of page faults for B+ tree:

Random access cost = lpf + lpf.

CHAPTER V

RESULTS AND DISCUSSION

The experimental results of both extendible hashing and B+ trees are presented in this chapter. Figures and tables indicating empirical results are listed in the Appendix.

Storage Utilization

The average storage utilization of both extendible hashing and B+ trees approaches 68% regardless of the page size. Figure 10 and 15 as well as table II and VII show the empirical results of storage utilization for both extendible hashing and the B+ trees.

As the page size increases the variations in storage utilization both for extendible hashing and for the B+ trees increase. Extendible hashing has higher variations in storage utilization than that of the B+ trees.

As the database size increases, the variations in storage utilization for both extendible hashing and B+ trees decreases.

Cyclical variations are observed in storage

utilization performance. The reason is that as pages become full, storage utilization increases. After some time pages become completely full and are split almost simultaneously and storage utilization decreases. After a while the page becomes full and storage utilization increases.

As little as 57% and as much as 76%, storage utilization for extendible hashing is achieved, and for B+ trees the low is 63% and high is 70%.

Overall, the storage utilization for B+ trees is more consistent than that of extendible hashing.

Random Access Cost

Random access cost was measured in terms of page faults. After inserting a certain number of records, 1000 records were accessed and the number of page to access those records was measured. Figure 11, 16, and 19 as well as table III, VIII, and XI show the empirical results.

It is observed that the random access cost for extendible hashing is always less than that of the B+ trees access methods. The reason behind this is that in extendible hashing the key directly gives the directory entry and that entry contains the address of the page in which the record should be. Since in a B+ trees access

methods the index set has to be traversed until the leaf page is found, there are more page faults required to search the index.

It is obvious that the higher the page size, the lower the number of pages in the database. There are more chances of getting the page from the buffer, which is in resident memory, so there are fewer page faults.

In addition, it is observed that the higher the page size, the lower the random access cost. As the page size increases, the relative decrease in random access cost also decreases. This is found in both extendible hashing and B+ trees access methods.

A step function is observed in random access cost with an increase in database size for both extendible hashing and B+ trees. For extendible hashing, whenever the directory size doubles in a size, a step is observed; and for B+ trees whenever the root page splits (i.e. the height of index set increases), a step is observed.

It is obvious that the higher the buffer size, the more pages can reside in a resident memory. Hence there are more chances of getting a page from the buffer, resulting in fewer number of page faults. With an increase in buffer size the corresponding decreasing number of page faults is greater for extendible hashing than that of a B+ trees. The reason behind this is that more directory pages

of extendible hashing can reside in resident memory, so fewer page faults are needed for searching the directory.

Insertion Cost

Insertion cost was measured in terms of page faults. After inserting a certain number of records, 1000 additional records were inserted and page faults were measured during those insertions. The empirical results are shown in figure 12, 17, and 20 as well as table IV, IX, and XII.

The following observations were made for insertion cost:

1. The insertion cost for extendible hashing is always less than that of B+ trees access methods.
2. For both methods, as the page size increases, the insertion cost decreases; and as the page size increases, the corresponding decrease in insertion cost also decreases.
3. A step function is observed for insertion cost with increases in database size for both extendible hashing and B+ trees access methods.
4. The number of decreasing page faults with increase in buffer size is higher for extendible hashing than that of B+ trees access methods.

The reasons for the preceding observations are the same as those explained for random access cost.

Sequential access cost

Sequential access cost is measured in terms of the number of pages in the database. Figure 9 and 14 as well as table I and VI show the empirical results of sequential access cost for both extendible hashing and the B+ trees access methods.

It is obvious that as the page size increases, the total number of pages in a database decreases. It is observed that for the B+ trees method the decrease in the number of pages with an increase in page size is more consistent than that of extendible hashing. It is also observed that with an increase in the database size the increase in the number of pages for B+ trees is more nearly linear than that of extendible hashing.

The reason behind the above observations is the more consistent storage utilization of B+ trees than that of extendible hashing.

Directory size(depth)

The knowledge of the distribution of the directory size or depth is important for the design of an extendible hashing file system. Since this size changes by factors of two, its fluctuations may be quite significant. The directory size is largely dependent on the existance of clusters. An implementation of extendible hashing that accomodate some overflow would lessen the frequency of doubling the directory.

The empirical results are shown in figure 13 and 17 as well as table V and X. It is observed that the corresponding decrease in the directory size with an increase in page size also decreases. It is also observed that the increase in directory size with the increase in database size is a step function.

CHAPTER VI

SUMMARY AND CONCLUSION

Given that an index resides on discs or drums, searching it must be done by accessing secondary storage. The time required to access secondary storage is the main component of the total time required to retrieve information from databases[20]. Minimizing the number of accesses to secondary storage is highly desirable.

Extendible hashing and B+ trees access methods are two index sequential access methods that do not require complete file reorganization. They can be very useful for applications that require random access and sequential access in natural order.

Conclusions

The average storage utilization of both extendible hashing and B+ trees is about 68%. A B+ tree has more consistent storage utilization than that of extendible hashing. The performance of extendible hashing can be degraded by the existence of a cluster.

The random access cost of extendible hashing is

always less than that of B+ trees. This can be still further improved if there is not an excessive dependance of the directory size on the existence of a cluster. There are never more than two page faults necessary to locate a key and its associated information for extendible hashing.

The sequential access cost of B+ trees methods is more consistent than that of extendible hashing. This is due to the fact that extendible hashing results in more variations in storage utilization than B+ tree methods.

The insertion cost of extendible hashing is always less than that of B+ trees methods. This is due to a maximum of one page fault to search an index for a key in extendible hashing.

If the directory size is small and can be kept in primary memory, then there is a maximum of one page fault to access a record in extendible hashing. If an order preserving hash function is used that can break up clusters, then there will be quite an improvement in performance for extendible hashing.

Suggested Future Work

The result in the thesis are obtained just for search and insertion. It would be an interesting topic if deletions were implemented. This topic is left to future

study.

In this study, a simple implementation of a B+ tree is compared to extendible hashing with sequential access. The results are based on comparisons of these two methods. Refinements of the implementation of either or both methods could produce different results, and could be subject of further study.

SELECTED BIBLIOGRAPHY

[1] Fagin, R., Nievergelt, N., Pippenger, N., and Strong H. "Extendible Hashing - A fast access method for dynamic files." ACM Transactions on Database Systems, 4, 3(Sept 1979), 315-344.

[2] Bechtold, U., and Kuspert, K. "On the use of extendible hashing." Information Processing Letters, 19(1984), 21-26.

[3] Ellis, C. "Extendible Hashing for concurrent operations and distributed data." Proceddings of the Second ACM SIGACT- SIGMOD Symposium on principle of database systems 21 - 23 March 1983, Atlanta, Georgia. 106-116.

[4] Lomet, D. "Bounded index exponential hashing." ACM Transactions on Database Systems, 8, 1(March 1983), 136-165.

[5] Tamminen, M. "The Extendible cell method for closest point problems." BIT, 22(1982), 27-44.

[6] Tamminen, M. "Order preserving extendible hashing and bucket tries." BIT, 21(1981), 419-435.

[7] Tamminen, M. "Extendible hashing with overflow." Information Processing Letters, 15, 2(1982), 227-232.

[8] Yao, A. "A note on extendible hashing." Information Processing Letters, 11, 2(1980), 84-86.

[9] Mendelson, H. "Analysis of extendible hashing." IEEE transactions on software engineering, SE 8, 6(Nov 1982), 611-619.

[10] Stephen, H. "Multidimensional extendible hashing for partial match queries." International Journal Of Computer And Information Sciences, 14, 2(1985), 73-83.

[11] Chang, C. "The study of an ordered minimal perfect hashing scheme." Communication of the ACM, 27, 4(1984), 384-387.

51 [12] Scholl, M. "Newfile organization based on dynamic hashing." ACM Transactions On Database Systems, 6, 1(March 1981), 194-211.

[13] Knuth, D. The Art Of Computer Programming, vol 3 : Sorting and Searching. Reading, MA : Addison - Wesley, 1973.

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

[15] Litwin, W. "Virtual hashing : A dynamically changing hashing." Proc. Very Large Databases Conf., Berlin, 1978, 517-523.

[16] Coffman, E. and Eve, J. "File structures using hashing functions." Communication of the ACM, 13, 7(July 1970), 427-436.

[17] Nakamura, T. and Mizoguchi, T. "An analysis of storage utilization factor in block split data structuring scheme." International Conference On Very Large Databases 4 th , 1978, West Berlin, Germany, Sept 13-15, 1978.

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

[19] Bell, D. and Deen, S. "Hash Trees Versus B-Trees." The Computer Journal, 27, 3(1984), 218-224.

[20] Larson, P. "Linear Hashing with Overflow-Handling by Linear Probing." ACM Trans. on Database Systems, 10, 1(March 1985), 75-89.

[21] Grimson, J. and Stacey, G. "A Performance study of Some Directory Structures for Large Files." Information Storage and Retrieval, 10(1974), 357-364.

[22] Bayer, R. and Unterauer, K. "Prefix B-trees." ACM Trans. on Database Systems, 2, 1(March 1977), 11-16.

[23] Bayer, R. and McCreight, E. "Organization and Maintainance of Large Order Indexes." Acta Informatica, 1(1972), 173-189.

APPENDIX

EMPIRICAL RESULTS

TABLE I

COMPARISONS OF SEQUENTIAL ACCESS COST
WITH NUMBER OF PAGES

Number of Records	Sequential Access Cost (in terms of Number of Pages)	
	Extendible Hashing	B+ tree
1000	51	51
2000	100	98
3000	163	147
4000	198	195
5000	253	244
6000	307	296
7000	337	351
8000	366	395
9000	422	441
10000	489	484
11000	550	533
12000	602	595
13000	636	648
14000	678	705
15000	702	748
16000	728	792
17000	786	836
18000	831	886
19000	888	928
20000	960	987
21000	1033	1037
22000	1108	1083
23000	1190	1120
24000	1262	1172
25000	1319	1221
26000	1360	1278
27000	1393	1333
28000	1432	1387
29000	1465	1433
30000	1512	1478

TABLE II

COMPARISONS OF NUMBER OF RECORDS WITH PERCENTAGE STORAGE UTILIZATION

Number of Records	Pecentage Storage Utilization	
	Extendible Hashing	B+ tree
1000	65.35	65.35
2000	66.66	67.86
3000	61.34	67.22
4000	67.34	68.52
5000	65.87	68.00
6000	65.14	67.03
7000	69.23	66.57
8000	72.85	67.91
9000	71.09	68.55
10000	68.16	68.78
11000	66.66	68.64
12000	66.44	66.63
13000	68.13	66.60
14000	68.82	66.23
15000	71.22	67.03
16000	73.26	67.67
17000	72.09	68.17
18000	72.20	68.12
19000	71.32	68.54
20000	69.44	67.58
21000	67.76	67.57
22000	66.18	67.55
23000	64.42	68.19
24000	63.39	67.77
25000	63.17	67.90
26000	63.72	67.60
27000	64.60	67.45
28000	65.17	67.27
29000	65.98	67.53
30000	66.08	67.74

TABLE III

COMPARISONS OF NUMBER OF RECORDS
WITH RANDOM ACCESS COST

Number of Records	Random access cost (in terms of page faults)	
	Extendible Hashing	B+ tree
1000	832	911
2000	901	1101
3000	944	1397
4000	1107	1407
5000	1133	1554
6000	1141	1663
7000	1142	1658
8000	1147	1670
9000	1137	1674
10000	1142	1747
11000	1377	1753
12000	1379	1848
13000	1409	1885
14000	1644	1887
15000	1634	1869
16000	1663	1898
17000	1664	1887
18000	1643	1903
19000	1657	1907
20000	1646	1899
21000	1661	1932
22000	1660	1925
23000	1651	1948
24000	1662	2048
25000	1630	2043
26000	1647	2051
27000	1639	2051
28000	1633	2172
29000	1652	2159
30000	1613	2093

TABLE IV

COMPARISONS OF NUMBER OF RECORDS
WITH INSERTION COST

Number of Records	Insertion Cost (in terms of page faults)	
	Extendible Hashing	B+ tree
1000	985	1104
2000	1092	1357
3000	1163	1500
4000	1340	1547
5000	1337	1691
6000	1261	1736
7000	1259	1729
8000	1357	1735
9000	1398	1755
10000	1365	1821
11000	1549	1858
12000	1541	1983
13000	1665	1996
14000	1700	1952
15000	1731	1975
16000	1805	1977
17000	1766	1999
18000	1815	1987
19000	1862	2014
20000	1856	2031
21000	1844	2021
22000	1865	2021
23000	1845	2153
24000	1807	2139
25000	1760	2169
26000	1744	2166
27000	1740	2287
28000	1728	2260
29000	1731	2174
30000	1745	2189

TABLE V

COMPARISONS OF NUMBER OF RECORDS
WITH DIRECTORY SIZE

Number of Records	Directory Size (for Extendible Hashing)
1000	128
2000	256
3000	256
4000	1024
5000	1024
6000	1024
7000	1024
8000	1024
9000	1024
10000	1024
11000	2048
12000	2048
13000	2048
14000	4096
15000	4096
16000	4096
17000	4096
18000	4096
19000	4096
20000	4096
21000	4096
22000	4096
23000	4096
24000	4096
25000	4096
26000	4096
27000	4096
28000	4096
29000	4096
30000	4096

TABLE VI

COMPARISONS OF SEQUENTIAL ACCESS COST WITH NUMBER OF PAGES

	Sequential Access Cost (in terms of Number of Pages)	
Page Size	Extendible Hashing	B+ tree
10	2251	2227
20	1147	1105
30	702	748
40	585	552
50	409	447
60	342	366
70	332	310

TABLE VII

COMPARISONS OF PAGE SIZE WITH PERCENTAGE STORAGE UTILIZATION

	Percentage Storage Utilization	
Page Size	Extendible Hashing	B+ tree
10	66.63	67.57
20	65.38	67.87
30	71.22	66.84
40	64.10	67.93
50	73.34	67.11
60	73.09	68.30
70	64.54	68.68

TABLE VIII

COMPARISONS OF PAGE SIZE WITH
RANDOM ACCESS COST

	Random Access Cost (in terms of Page Faults)	
Page Size	Extendible Hashing	B+ tree
10	1953	3179
20	1735	2151
30	1634	1869
40	1303	1660
50	1140	1634
60	1021	1418
70	987	1375

TABLE IX

COMPARISONS OF PAGE SIZE WITH
INSERTION COST

	Insertion Cost (in terms of Page Faults)	
Page Size	Extendible Hashing	B+ tree
10	2115	3477
20	1914	2332
30	1731	1975
40	1412	1735
50	1221	1672
60	1178	1478
70	1011	1455

TABLE X

COMPARISONS OF PAGE SIZE WITH
DIRECTORY SIZE

Page Size	Directory size (for Extendible Hashing)
10	16384
20	4096
30	4096
40	2048
50	2048
60	2048
70	1024

TABLE XI

COMPARISONS OF BUFFER SIZE WITH
RANDOM ACCESS COST

	Random Access Cost (In terms of Page Faults)	
Buffer Size	Extendible Hashing	B+ tree
10	1782	2173
20	1634	1869
30	1500	1791
40	1391	1705
50	1306	1661
60	1229	1604
70	1173	1551

TABLE XII

COMPARISONS OF BUFFER SIZE WITH INSERTION COST

	Insertion Cost (in terms of Page Faults)	
Page Size	Extendible Hashing	B+ tree
10	1857	2244
20	1731	1975
30	1632	1878
40	1549	1802
50	1488	1750
60	1459	1693
70	1424	1632

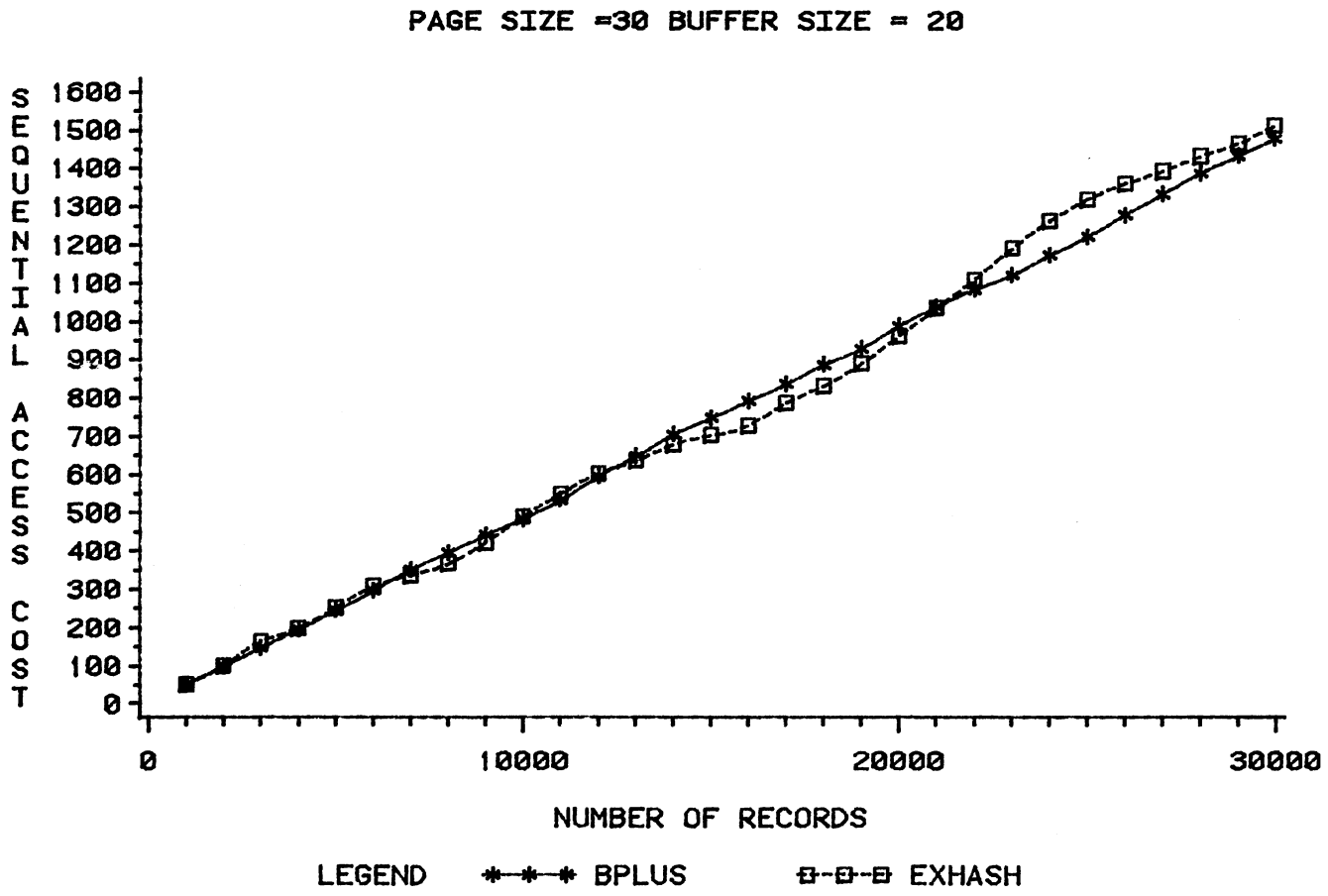

FIGURE 9: SEQUENTIAL ACCESS COST (IN TERMS OF NUMBER OF PAGES) VS NUMBER OF RECORDS

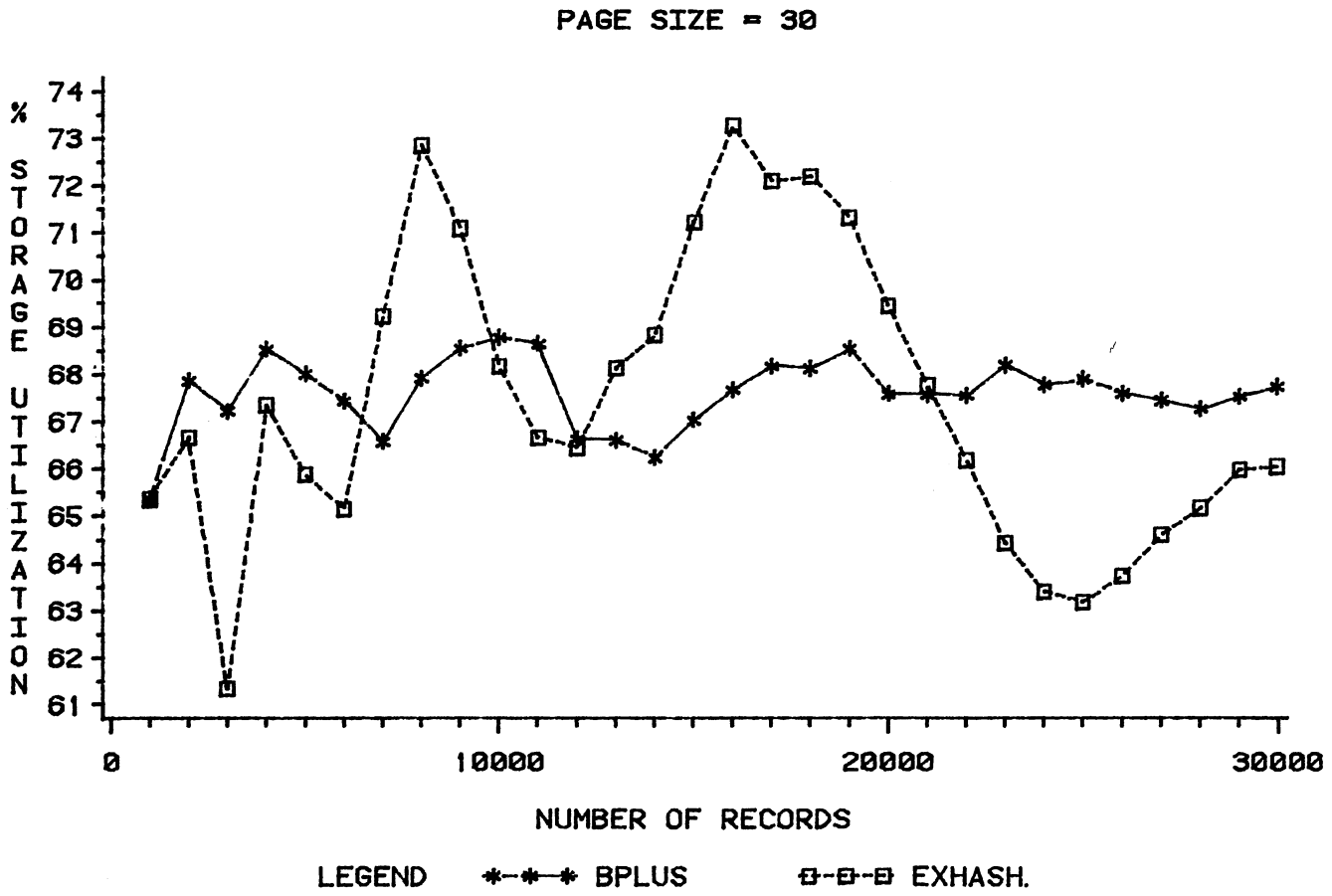

FIGURE 10: PERCENTAGE STORAGE UTILIZATION VS NUMBER OF RECORDS

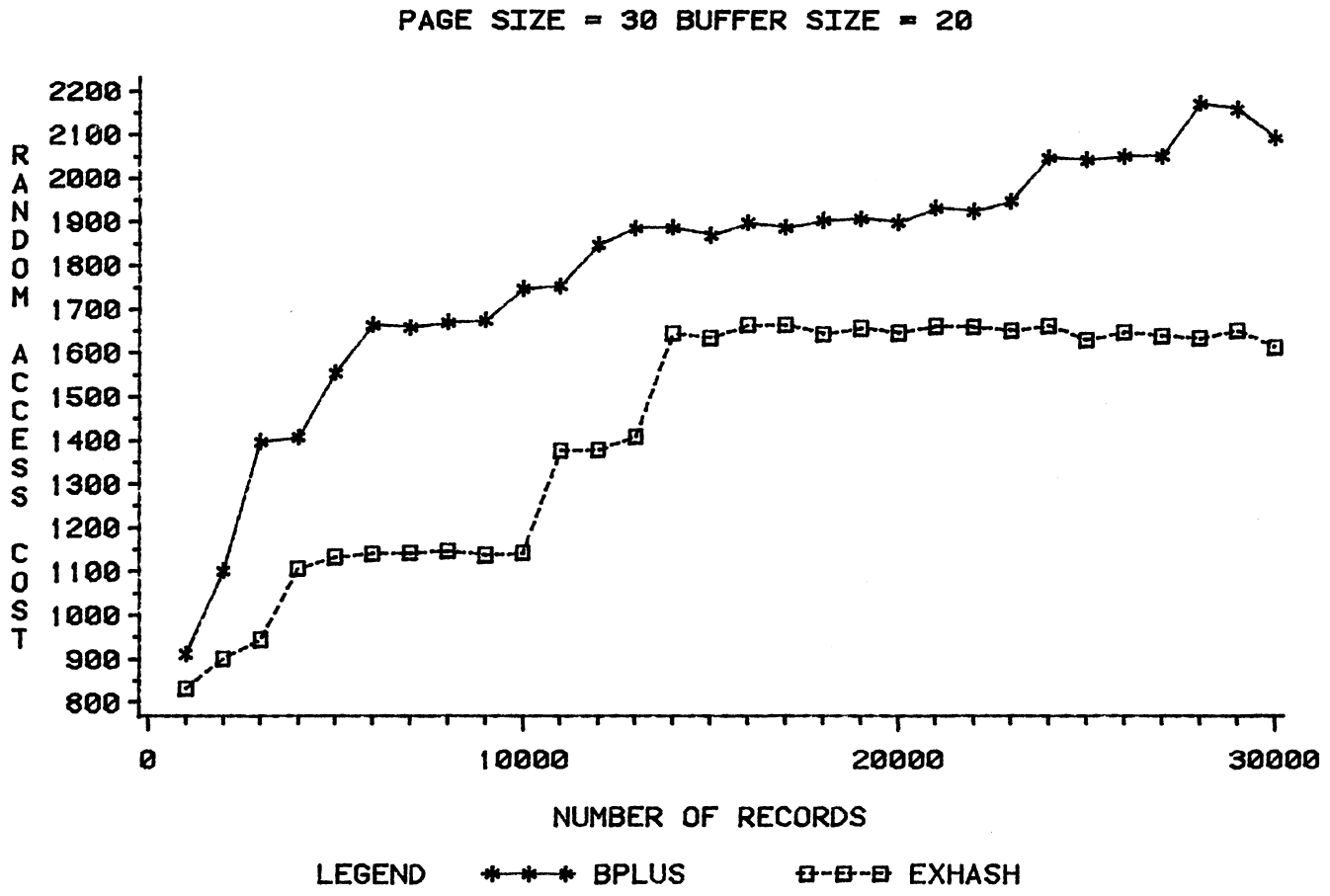

FIGURE 11: RANDOM ACCESS COST (IN TERMS OF PAGE FAULTS) VS NUMBER OF RECORDS

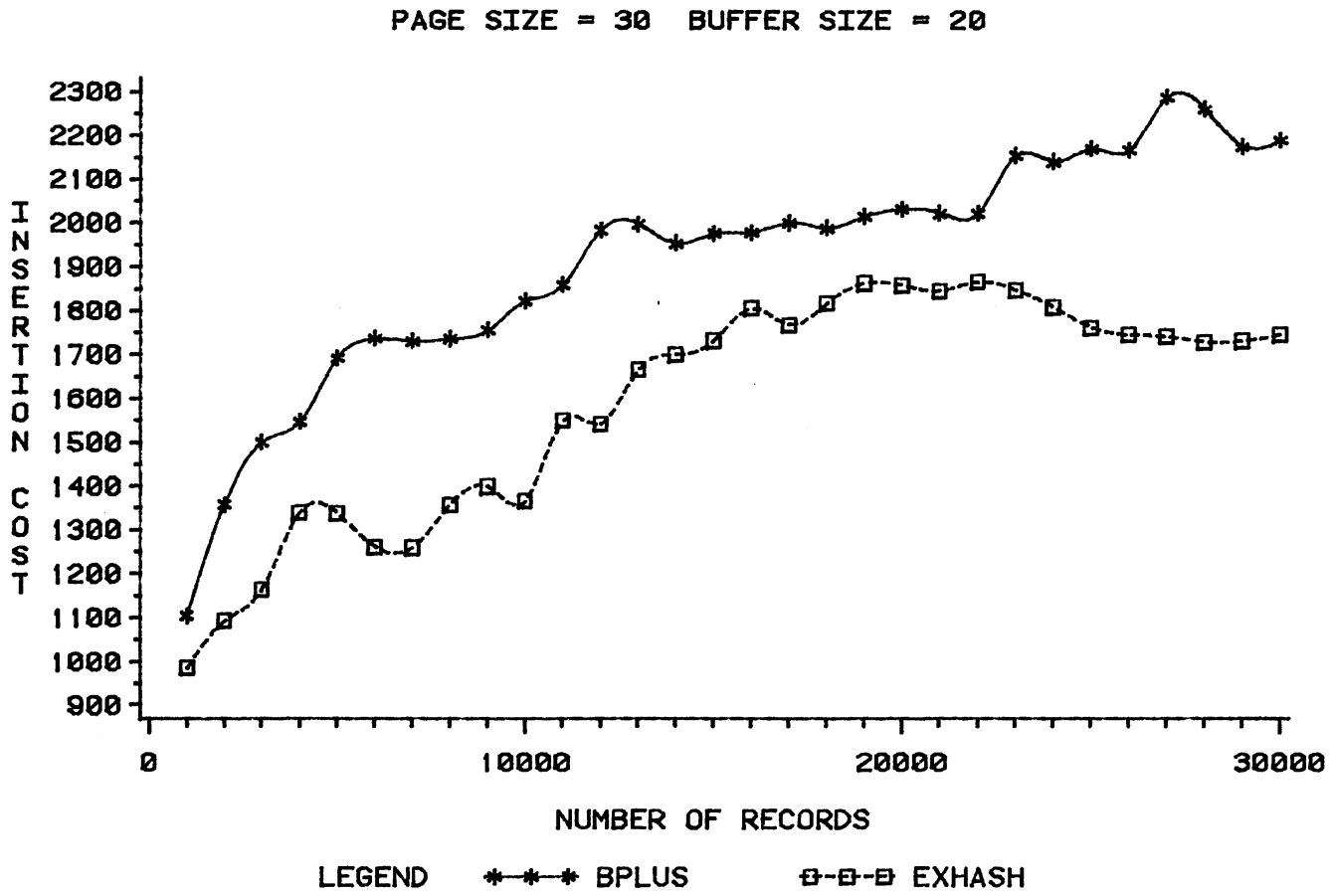

FIGURE 12: INSERTION COST (IN TERMS OF PAGE FAULTS)
VS NUMBER OF RECORDS

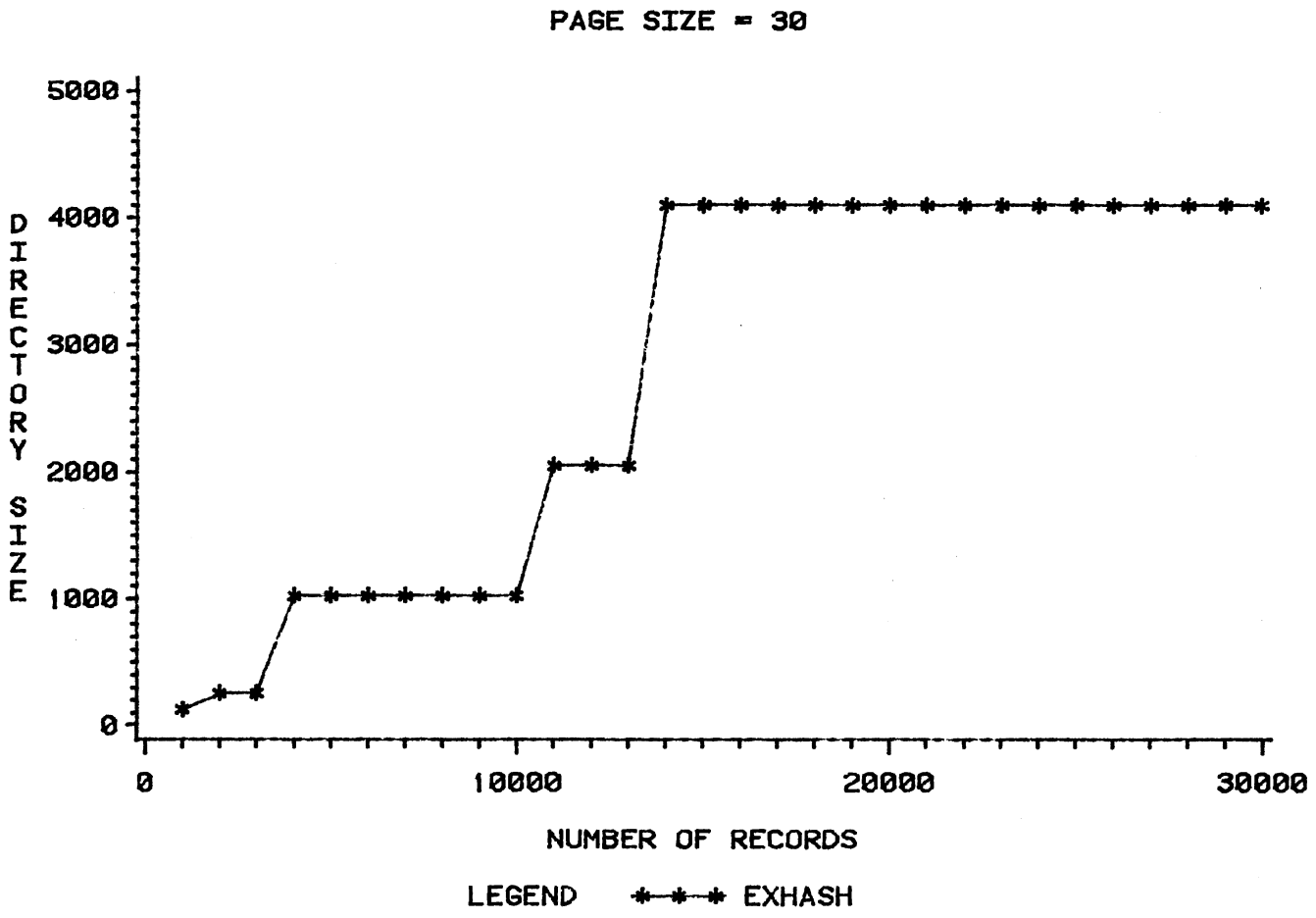

FIGURE 13: DIRECTORY SIZE VS NUMBER OF RECORDS

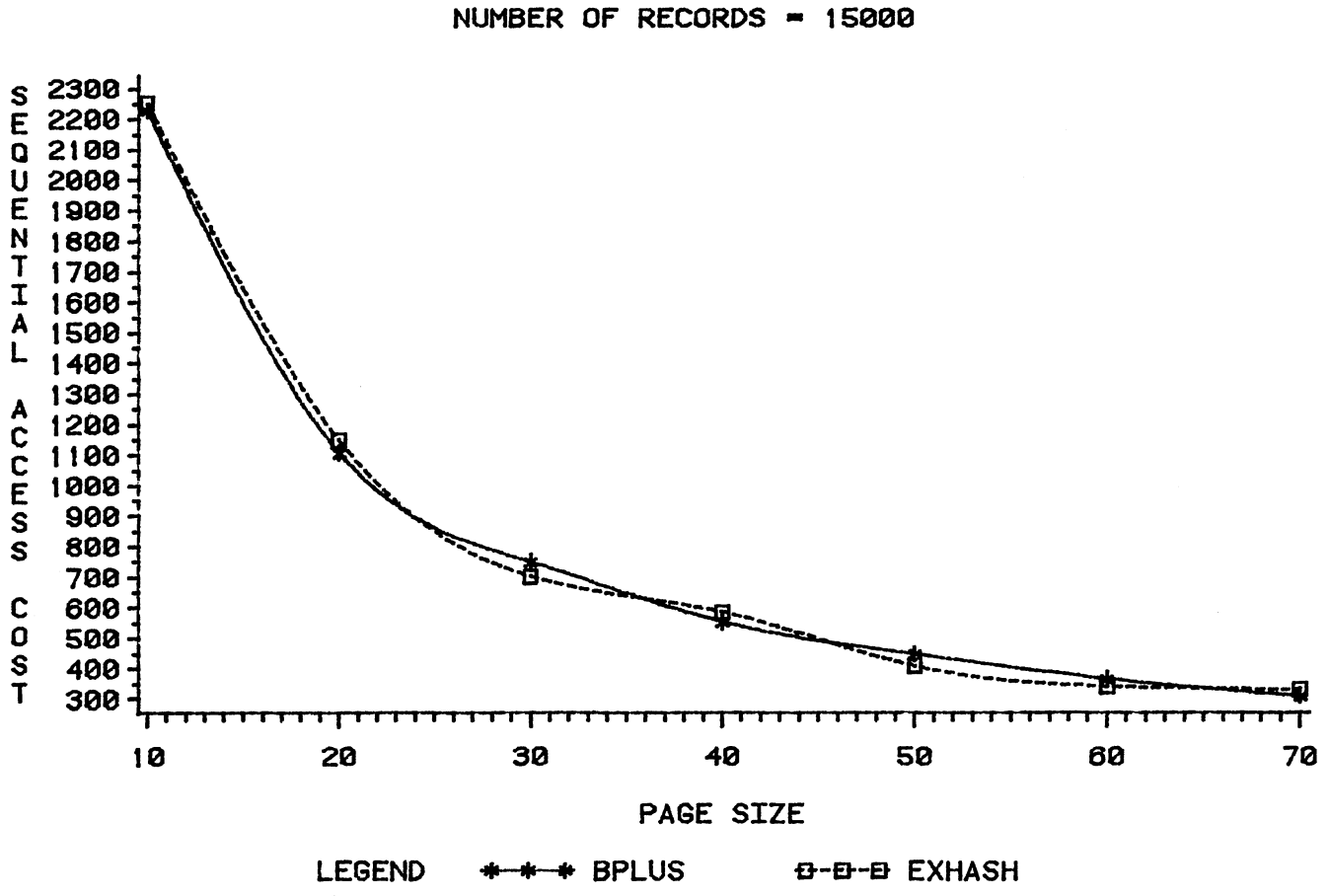

FIGURE 14: SEQUENTIAL ACCESS COST (IN TERMS OF NUMBER OF PAGES) VS PAGE SIZE

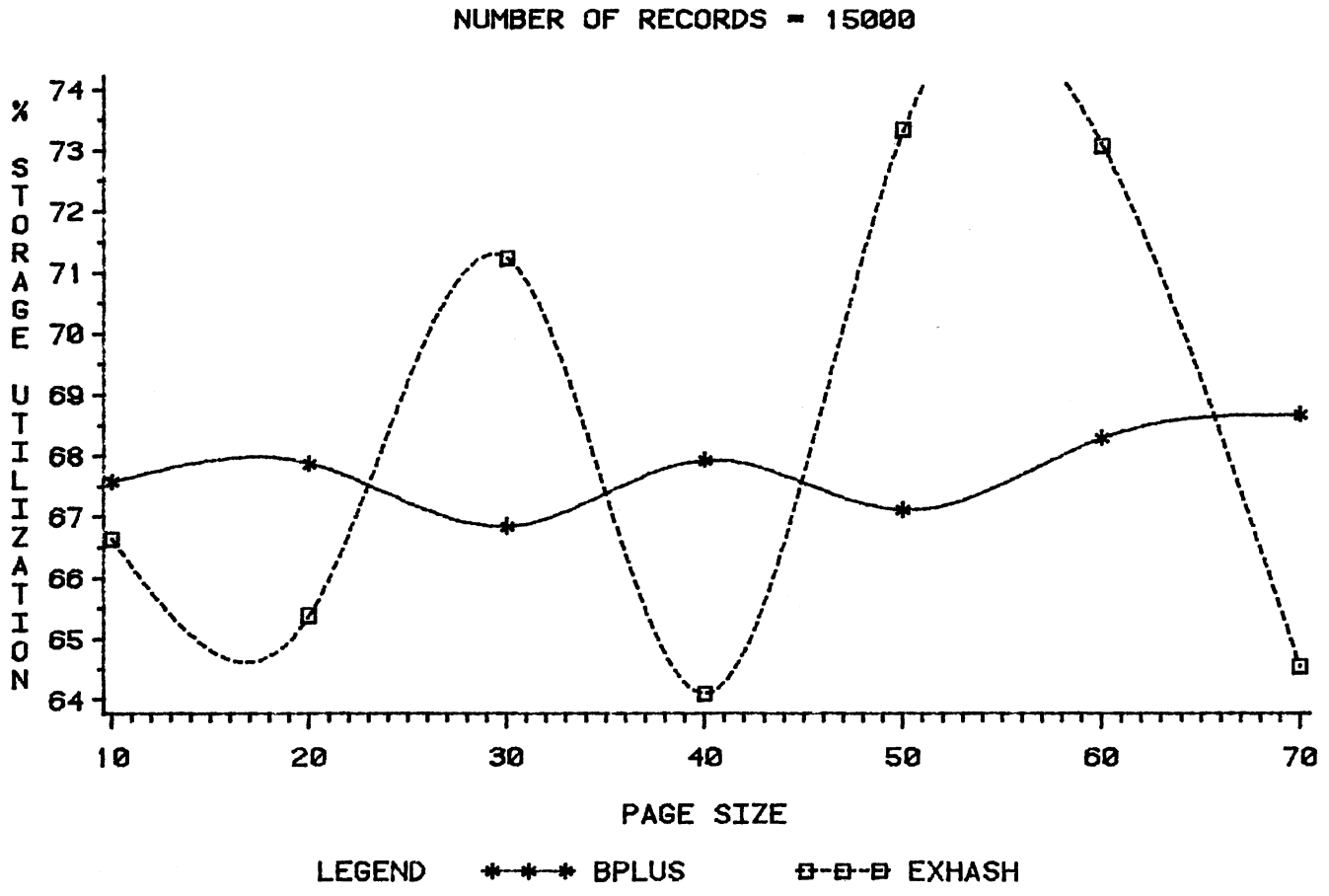

FIGURE 15: PERCENTAGE STORAGE UTILIZATION VS PAGE SIZE

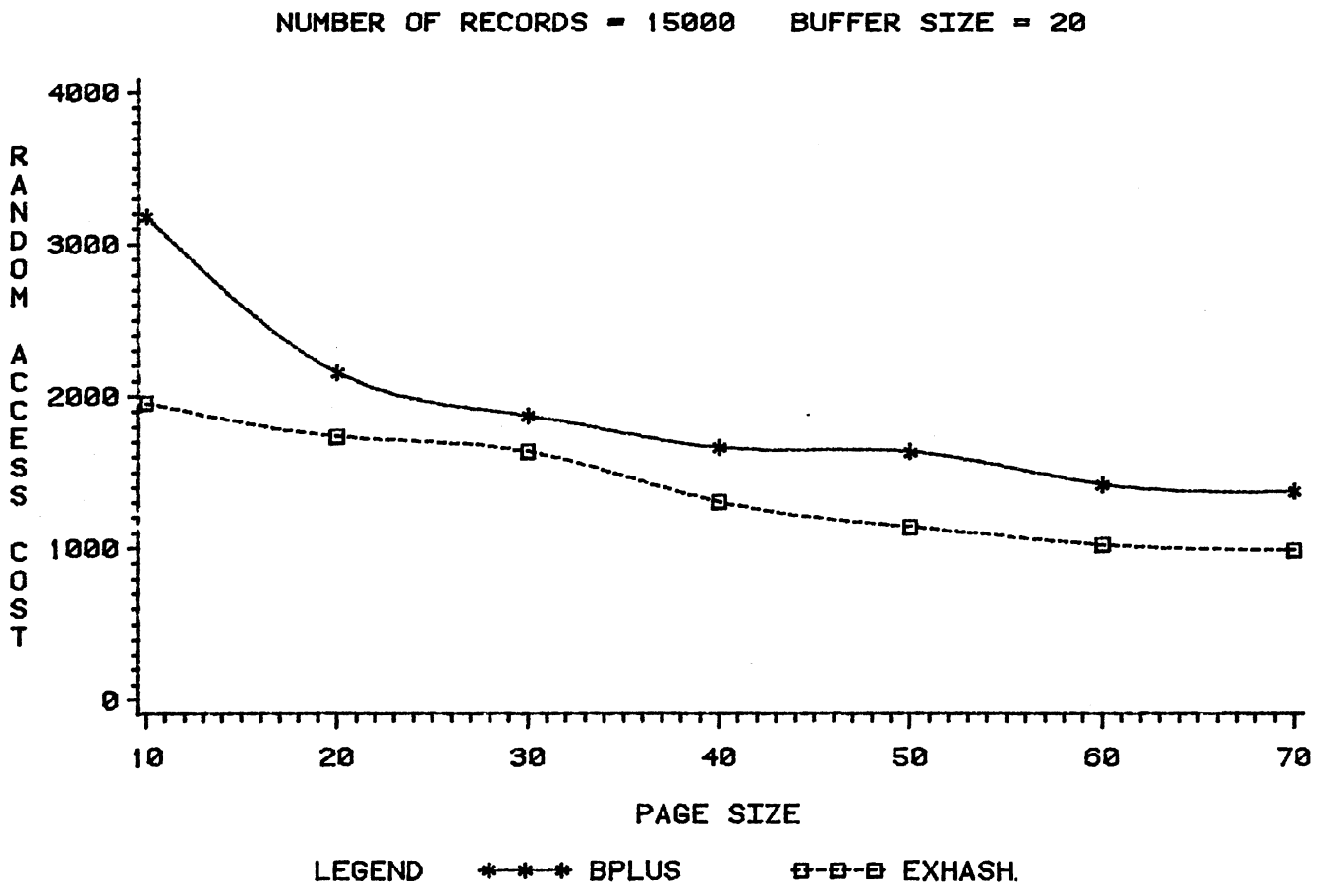

FIGURE 16: RANDOM ACCESS COST (IN TERMS OF PAGE FAULTS) VS PAGE SIZE

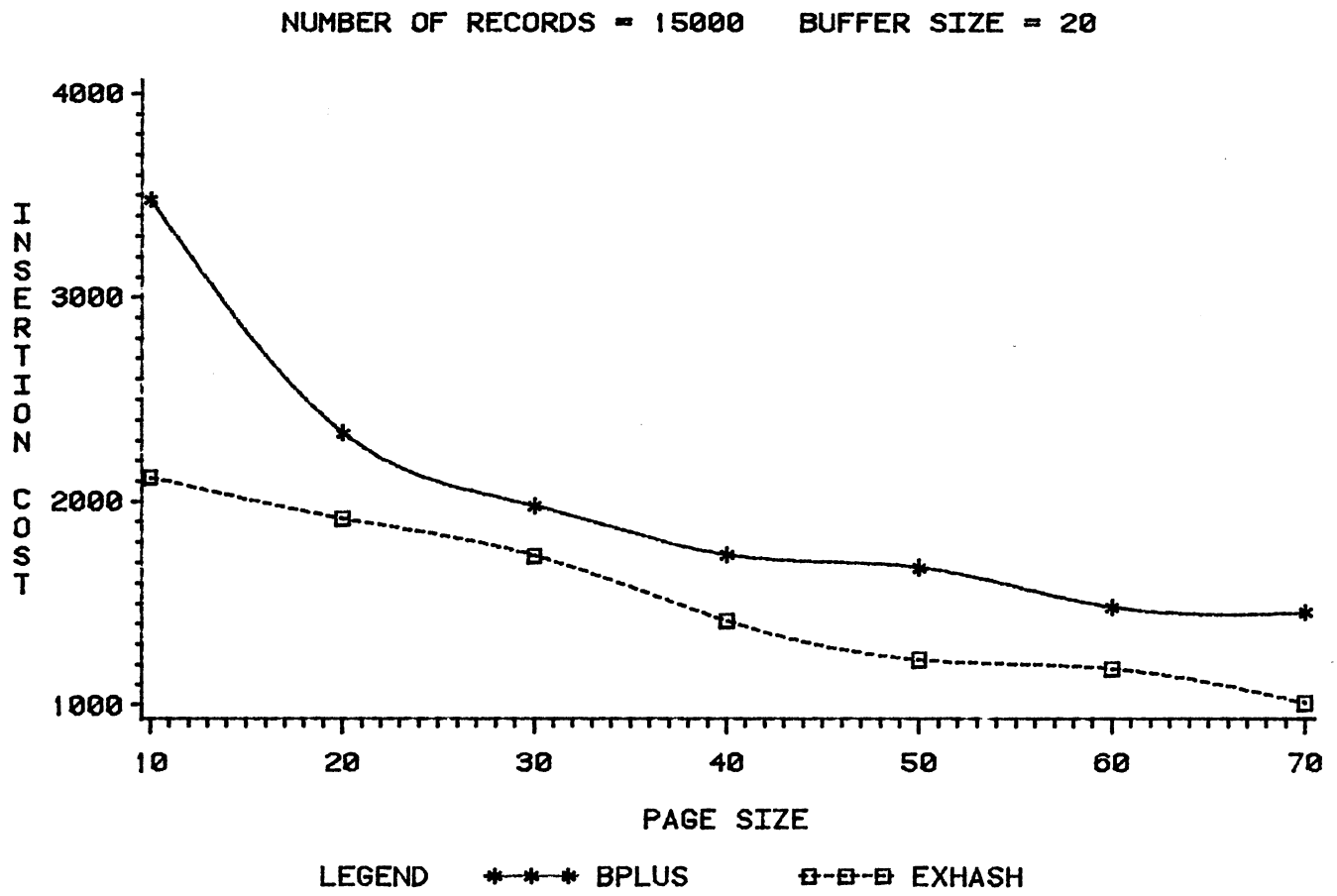

FIGURE 17: INSERTION COST (IN TERMS OF PAGE FAULTS) VS PAGE SIZE

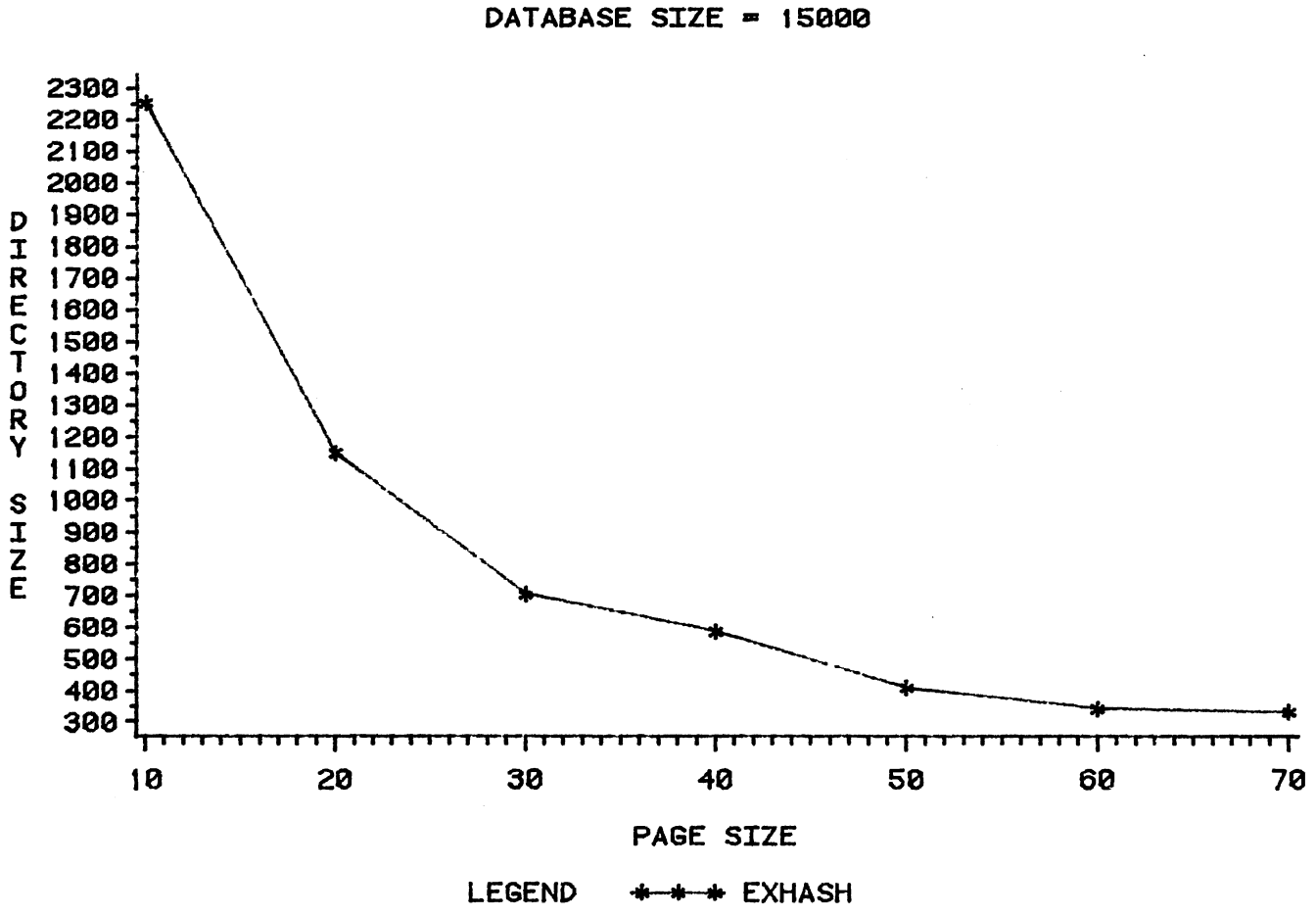

FIGURE 18: DIRECTORY SIZE VS PAGE SIZE

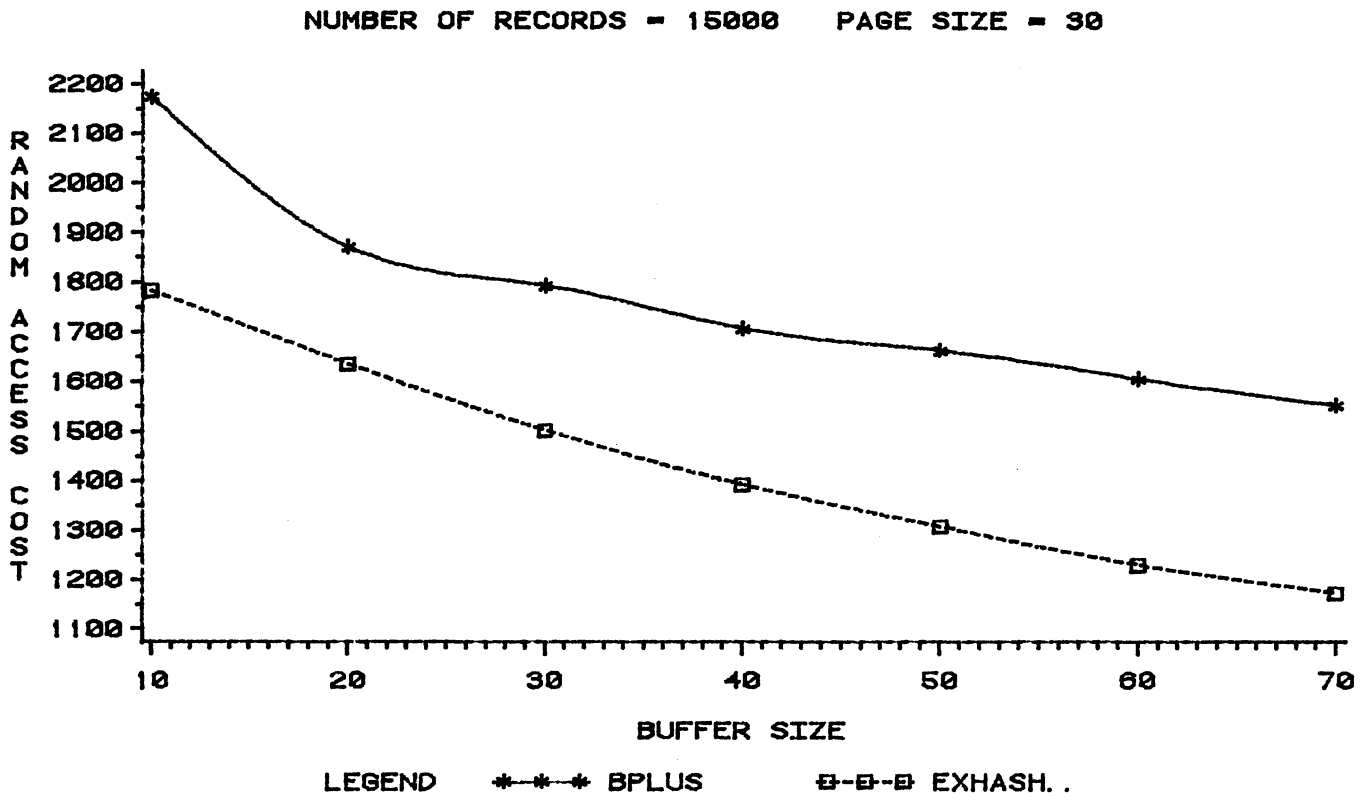

FIGURE 19: RANDOM ACCESS COST (IN TERMS OF PAGE FAULTS) VS BUFFER SIZE

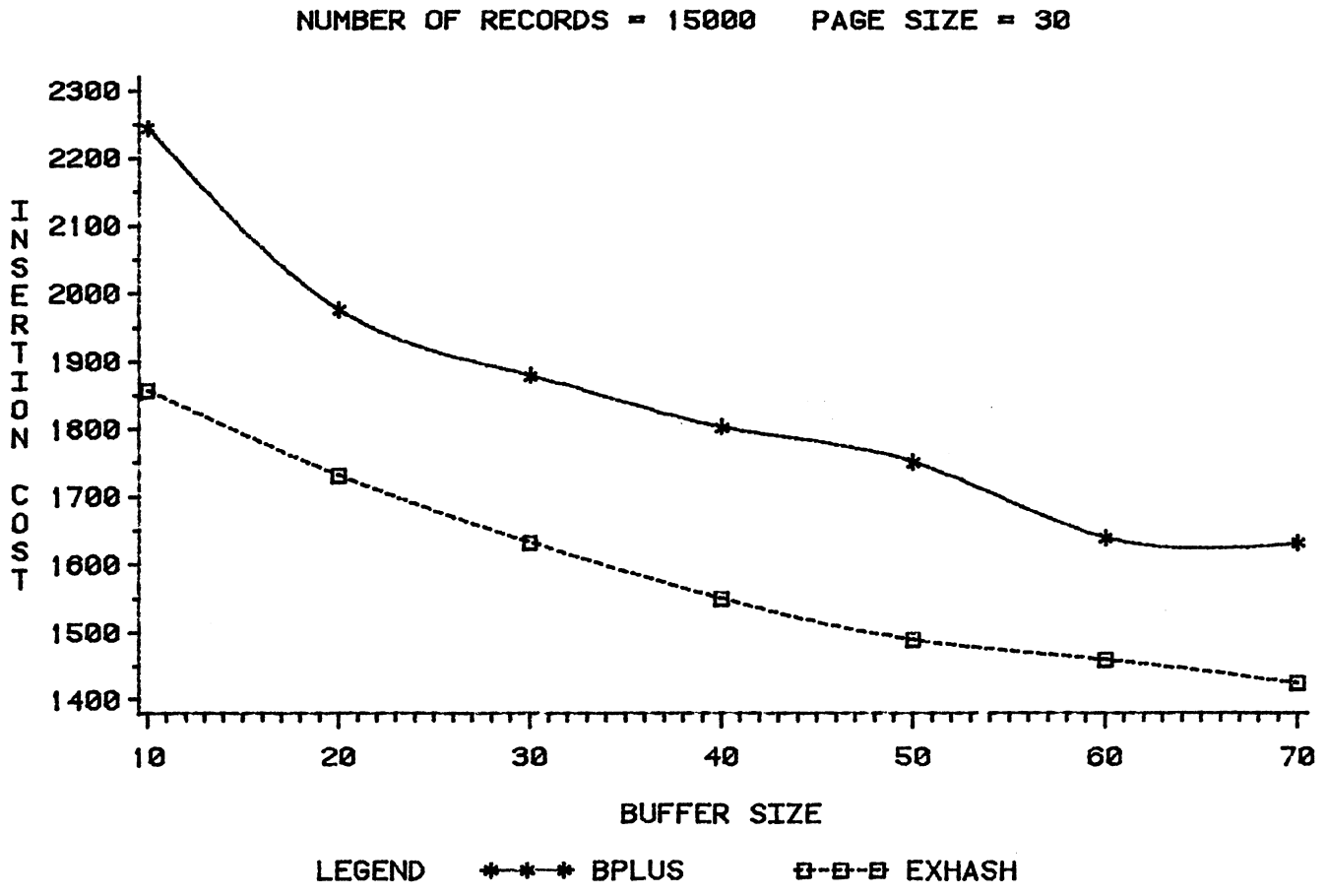

FIGURE 20: INSERTION COST (IN TERMS OF PAGE FAULTS)
VS BUFFER SIZE

VITA

Harshad D. Patel

Candidate for the Degree of

Master of Science

Thesis: ANALYSIS AND COMPARISON OF EXTENDIBLE HASHING AND B+ TREES ACCESS METHODS

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in India, April 4, 1961, the son of Devabhai and Shantaben Patel. Married to Maya Patel on December 15, 1983.

Education: Graduated from K.V.S. High school, Kharel, India, in May 1976; received Bachelor of Science degree in Chemistry from Sardar Patel University, V.V.Nagar, India, in July 1981; received Bachelor of Science Technology degree in chemical engineering from Bombay University, Bombay, in July 1984; Completed requirements for Master of Science degree at Oklahoma State University in December, 1987.