Extendible Hashing—A Fast Access Method for Dynamic Files

RONALD FAGIN
IBM Research Laboratory
JURG NIEVERGELT
Institut Informatik
NICHOLAS PIPPENGER
IBM T. J. Watson Research Center
and
H. RAYMOND STRONG
IBM Research Laboratory

Extendible hashing is a new access technique, in which the user is guaranteed no more than two page faults to locate the data associated with a given unique identifier, or key. Unlike conventional hashing, extendible hashing has a dynamic structure that grows and shrinks gracefully as the database grows and shrinks. This approach simultaneously solves the problem of making hash tables that are extendible and of making radix search trees that are balanced. We study, by analysis and simulation, the performance of extendible hashing. The results indicate that extendible hashing provides an attractive alternative to other access methods, such as balanced trees.

Key Words and Phrases: hashing, extendible hashing, searching, index, file organization, radix search, access method, *B*-tree, trie, directory, external hashing
CR Categories: 3.72, 3.73, 3.74, 4.33, 4.34, 5.25

1. EVOLUTION OF FILE ORGANIZATION SCHEMES

Over the past two decades, schemes for structuring large files of data have evolved by merging concepts and techniques from two areas that were initially perceived as requiring distinct approaches: data structures appropriate for central memory, and access methods appropriate to slow, high-capacity secondary-storage devices. This distinction is becoming more and more blurred. We will briefly trace some relevant developments in both areas, and show their convergence towards general schemes for structuring data whose volume is allowed to grow and shrink by large

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
Some of this research was carried out while J. Nievergelt was a visitor at the IBM Research Laboratory in San Jose and while N. Pippenger was a visitor in the Department of Computer Science at the University of Toronto.
Authors' addresses: R. Fagin and H. R. Strong, IBM Research Laboratory, San Jose, CA 95193; J. Nievergelt, Institut Informatik, ETH, Zurich, Switzerland; N. Pippenger, IBM T. J. Watson Research Center, Yorktown Heights, NY 10598.
© 1979 ACM 0362-5915/79/0900-0315 $00.75

factors, such that the data are always accessed by the same algorithm (requiring no "exception routines"), and worst case performance is almost never significantly worse than average performance.

The first schemes used for structuring data were more appropriate to static than to dynamic data. *Static* means that the *extent* and *structure* of the data remain unchanged during processing; only *values* may be updated. *Dynamic* means that data elements may be *inserted* and *deleted*, and relationships between data elements (such as links) may be changed. The distinction between static and dynamic data is of course not clear-cut (e.g. changing a link means updating a pointer value), but in practice it is usually unambiguous and serves a useful purpose.

The array (as a data structure for central memory) and the sequential file (the only feasible structure on media restricted to sequential access, such as tape), are the best known examples of static structures. Insertions and deletions (except, possibly, at the end of a file) lead to at least one of two undesirable consequences: the introduction of *ad hoc mechanisms* (such as a flag to indicate that a record still present in the structure should be considered as having been deleted, or pointers to an overflow bucket which holds records that cannot be squeezed into their rightful place), and frequent expensive *restructuring* of the entire data collection (typically when the number of holes left by deletions, and overflow areas created by insertions, has grown so large as to degrade performance severely).

The evolution from static to dynamic data structures proceeded rapidly in those applications where data could be kept in central memory. List structures, invented to accommodate highly dynamic data, became an identifiable technique during the 1950s (see, for example, Newell and Simon [15]). The problem of possible degeneracy of list structures (for example, when a dynamic tree degenerates into a linear list because of a biased sequence of insertions and deletions) was recognized and attacked early. The height-balanced trees of Adelson-Velskii and Landis [1] were a pioneering step toward the development of data structures that adapt gracefully and gradually to repeated insertions and deletions. The concept of data structures that adapt their structure in response to external demands is now widely known.

The development of comparable dynamic file structures for seconday-storage devices was slower. With the advent of disks, the sequential files appropriate to tapes were quickly modified to indexed-sequential files (see, for example, [6]), which, ideally, permit access to any record 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 accessing permit; for highly dynamic files indexed-sequential access can lead to poor performance; instead of a 2-step access to data, long linear chains of "overflow buckets" may be traversed.

Balanced trees turned out to be a good solution for storing highly dynamic files on disks, just as they were for dynamic lists in central memory. The *B*-trees of Bayer and McCreight [3] (a decade after the discovery of balanced trees for list structures!), were the first file organization scheme that addressed the issue of gradual adaptation of structure to fit the data.

Since balanced trees are a successful technique for storing dynamic files, one

might well be tempted not to look further. We have attempted, however, to make a systematic search for other adaptable file organization schemes suitable for dynamic files, and we saw two approaches that appeared promising.

First, the analogy we have outlined, between file structures for secondary-storage devices, and data structures for central memory, leads one to investigate another general class of file organization techniques. Data structures for central memory fall into three broad categories: linearly or sequentially accessible (in time $O(n)$, where n is the number of items in the collection), accessible by tree structures (in time $O(\log(n))$), and directly accessible by key-to-address, or "hash," transformations (in time $O(1)$). Hashing schemes have so far been adapted to dynamic files on secondary-storage devices only by the inelegant and inefficient technique of attaching overflow buckets whenever needed, thus slowly but surely changing the $O(1)$ access time characteristic of hashing towards the $O(n)$ time characteristic of sequential allocation. If one can design adaptable hashing schemes that remain balanced as pages are added and deleted, the suitability of hashing for secondary-storage devices would be greatly enhanced.

Second, radix search trees (also known as digital search trees, or tries (Fredkin [5])), 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, for the simple reason that one comparison leads to a larger fan-out (equal to the number of characters in the alphabet underlying the key.space). In practice, however, radix search trees tend to be used only for small files, since they often waste memory. The scheme of allocating a field for each character of the alphabet at each node is better suited to representing the entire key space rather than the contents of a particular file. Thus a radix search tree usually contains space for many keys not in the table. Usually, the wasted memory space occurs at the nodes near the bottom of the tree. Attempts to exploit the speed of radix search trees without paying the penalty in memory space usually combine radix search for some prefix of the key with other search techniques for the suffixes (see, for example, Walker [18]). If, instead of switching from radix search to, say, binary search at some arbitrary depth in the tree, one could find a balancing scheme that would keep the tree uniformly filled, then radix search trees might provide an attractive alternative to balanced trees based on key comparison, such as B-trees.

We pursued both goals: (1) making hash tables extendible, so that they can adapt to dynamic files and (2) filling radix search trees uniformly, so that they remain balanced. These two apparently distinct goals merged into a single file organization scheme which has both aspects. It will be described in detail after the necessary concepts and terminology have been developed.

After preparing this paper, the authors learned that three similar but distinct schemes had been independently proposed, under the names expandable hashing [8], dynamic hashing [10], and virtual hashing [11]. The reader interested in the scheme described in this paper should also consult [8, 10, 11]. Expandable and dynamic hashing are very similar to our scheme. Both our scheme and the schemes presented in [8] and [10] use a directory (or index) pointing to leaves (or buckets), and all three schemes distribute records among buckets in the same way. The main difference is in the structure of the directory: Knott [8] and Larson [10] use linked access to a tree, while we use direct access to a contiguously

allocated table. Our scheme will be no worse than the schemes of Knott and Larson as regards access time, and with virtual memory ours will be much better: we risk but one page fault to find the bucket containing a record, while Knott and Larson will risk several, depending on the depth of the tree. The comparison as regards space depends on system-dependent parameters such as pointer lengths, and also on the particular file being stored. For practical values of the parameters, our directory will be small with probability very nearly 1 (in a sense that will be made precise in Section 5). This difference will usually be unimportant, however, since the space used for the directory will be small compared with that used for the records themselves. Comparison of virtual hashing with our scheme is more difficult. The definition of virtual hashing ("any hashing which may dynamically change its hashing function") is quite broad and could be taken to include all of the above schemes. The specific virtual hashing schemes Litwin [11] describes, however, are different enough so that comparison with our scheme appears to require further specification of implementation details and values of system-dependent parameters.

In addition to the details mentioned, other differences between our paper and the papers of Knott [8], Larson [10], and Litwin [11] are that we have a more comprehensive analysis (Section 5), and we describe (in Sections 2 and 3) an overall approach to designing file organizations.

2. CHARACTERISTICS OF DYNAMIC FILE ORGANIZATION SCHEMES

This section defines the concepts and terminology used throughout the paper, and illustrates them by means of well-known data structures and file organizations (see also [17, Ch. 6].

A *file* is a collection of *records*, each one identified by a *key*; usually a *natural order* is defined on the space of keys, which induces a natural order on the file. When accesses to the file occur according to the natural order, we speak of *sequential access* or *processing* of the file; otherwise we speak of *random access*.

A file organization scheme is a logical storage structure into which a file can be mapped, along with the algorithms needed to manage this structure. A scheme manages a collection of *pages* or *blocks*, usually of fixed size. To specify a scheme one has to describe the *relationship between the pages* as well as the *internal structure of a page,* and algorithms for *file maintenance* (inserting and deleting records) and access.

STRUCTURE BETWEEN PAGES. Pages are accessed by starting at a *root page* and following an *access path* which leads from page to page. A file organization scheme suitable for dynamic files imposes a *constraint on the balance* of the structure, which states that the length of access paths is bounded by some expression in the total number p of pages (for example, path length = $O(\log p)$ or perhaps $O(1)$).

INTERNAL STRUCTURE OF A PAGE. In general, a page contains *records* and *pointers* to other pages, i.e. a page is a storage area as well as a directory. If a page contains only pointers, it is called a *directory page*. If a page contains only keys, or keys and their associated records, then it is called a *leaf page*, or *leaf*. Usually the *occupancy*, or *load factor* λ of a page is bounded, i.e. constants α and β are specified such that $0 \leq \alpha \leq \lambda \leq \beta \leq 1$. The purpose of the lower bound α is to prevent the creation of pages that are underfilled; the purpose of the upper

bound β, when it differs from 1, is to reduce the number of undesirable events (such as collisions in hash tables) due to crowding.

File maintenance algorithms guarantee that the constraints on the balance of the entire structure, and on the load factor of each page, are always satisfied. Early file organization schemes did not include file maintenance as an integral part of the file structure. As a consequence, file maintenance algorithms were crude and not specified in much more detail than: "when there are so many holes and overflow buckets that access performance is severely degraded, *restructure* the whole file." Dynamic file organization schemes, on the other hand, enforce rigorously stated balance and occupancy constraints. As soon as an insertion or deletion causes an occupancy parameter to fall outside its allowed range, a "small" *rebalancing* operation is performed. Usually, an *underfilled page borrows* records from a neighbor, if there is one who can spare records; or an underfilled page is *merged* with a neighbor who can absorb it; and an *overfilled page* is *split* into two partially filled pages.

The difficulty of designing a dynamic file organization scheme lies in meeting all, or most, of the criteria above in a uniform way, by means of a set of simple concepts and algorithms. As an illustration, let us describe some well-known data structures and file organizations in terms of the concepts introduced above.

Consider binary search trees. Each node can be considered as a page with a simple internal structure: It contains precisely one key (or record) and two pointers (which may be nil). The load factor of each node is always 100 percent. If the tree is allowed to grow and shrink unchecked in response to random insertions and deletions, the $O(\log p)$ access performance expected of binary trees cannot be guaranteed; the worst case behavior will be $O(p)$. The height-balanced trees of Adelson-Velskii and Landis [1] enforce the following balance constraint: at any node, the heights of the two subtrees of this node may differ by at most 1. The weight-balanced trees of Nievergelt and Reingold [16] enforce the following constraint: at any node, the ratio of the weights (e.g. the total number of nodes) of the two subtrees of this node must lie within certain bounds. Both balance constraints guarantee access, insertion, and deletion in time $O(\log p)$. Both classes of trees have balancing algorithms based on local transformations called rotations, which restore the balance of a tree that was disturbed by a single insertion or deletion, in time $O(\log p)$.

Consider "paginated binary search trees," where each page may contain at most m records (nodes of the binary tree). The structure between pages is that of a multiway tree, with a page containing i keys having $i + 1$ pointers to other pages. Page faults will be minimized when each page contains a connected subtree. In order to achieve this, Muntz and Uzgalis [14] proposed the following constraint on paginated binary trees: If a newly inserted key has no place in the page of its father, then it is entered into a newly allocated page (rather than into any page that has an empty space). Unfortunately, this constraint leads to the creation of many nearly empty pages, with a corresponding waste of memory and access time.

Bayer and McCreight [3] proposed a more efficient occupancy constraint: when a newly inserted record does not fit into the page where it should go according to the natural order of its key, then split that page into two half-filled pages. Their B-trees satisfy the occupancy constraint $\frac{1}{2} \leq \lambda \leq 1$ for all pages (with the possible

exception of one, the root page). The internal structure of a page is not completely specified, except that it contains records as well as pointers to other pages. Access to records within a page was originally intended to be sequential. If a page has the internal structure of a binary tree, then a B-tree can be interpreted as being a paginated binary tree.

These examples should suffice to illustrate the concepts introduced at the beginning of this section. The reader who experiments with various combinations of balance and occupancy constraints and with various structures between the pages and internal to a page, will be able to discover an unlimited number of reasonable dynamic file organizations. Most of these will be variations on well-known themes. The following section combines these concepts to form a novel class of dynamic file organization schemes.

3. EXTENDIBLE HASHING EQUALS BALANCED RADIX SEARCH TREES

A clear understanding of the characteristics and components of dynamic file organization schemes, as presented in Section 2, allows one to design such schemes on demand. The results tend to cluster around a few basic types, however, one of which is the well-known balanced tree concept. A new basic type, to be described in this section, can be understood from two different viewpoints, and, accordingly, obtained by modifying two distinct known methods: hashing and radix search trees. These two addressing schemes are usually considered to be unrelated. Their interplay in our novel file organization scheme achieves two striking goals:

(1) Hashing, conventionally using a table of fixed size, can be made to be extendible.
(2) Radix search trees, conventionally seen to grow randomly, can be made to be balanced.

This section gives an intuitive, high-level description of two design processes that lead to the same goal, a file organization scheme we call *extendible hashing*. The detailed description is left to Section 4. By presenting not only the final result, but also the method that leads to its discovery, we hope that the reader will gain a deeper understanding of the essential issues.

3.1 Balancing Radix Search Trees

Thesis. Radix search trees are naturally extendible. By addressing their nodes via a hash function that provides a uniform distribution of keys they become balanced.

Let us illustrate the above brief statement of a design principle by means of an example.

(a) Consider the radix search tree in Figure 1, over the alphabet {0, 1, 2}. Assume that leaf L_{01} contains all keys that start with the digits 01, for example, the keys 012 and 01110. When a leaf overflows, as might have happened in our example to a previously present leaf L_{10}, it is simply replaced by an internal node to which three new leaves are attached (L_{100}, L_{101}, L_{102} in our example).

(b) Access to a radix search tree can be speeded up if, instead of comparing one digit of a key at a time (resulting in a fan-out equal to the size r of the

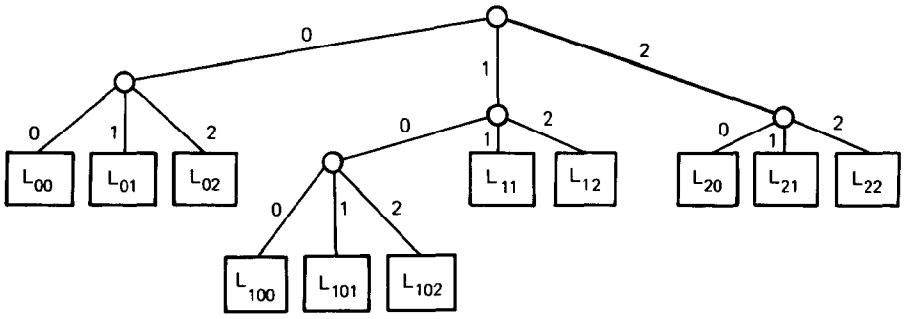

Fig. 1. A radix search tree

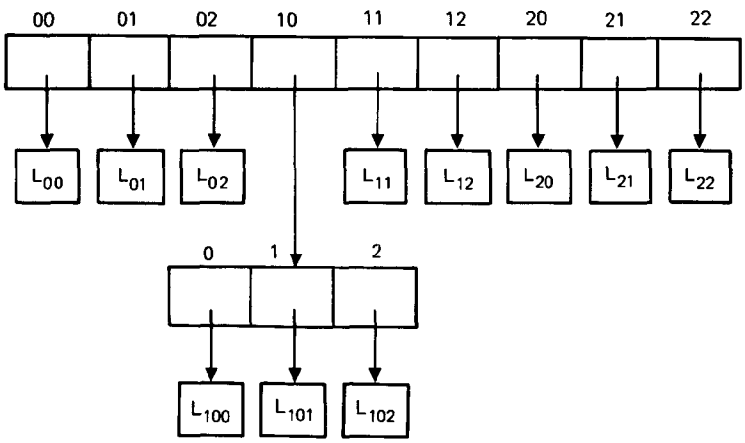

Fig. 2. Radix search tree with two levels compressed into one

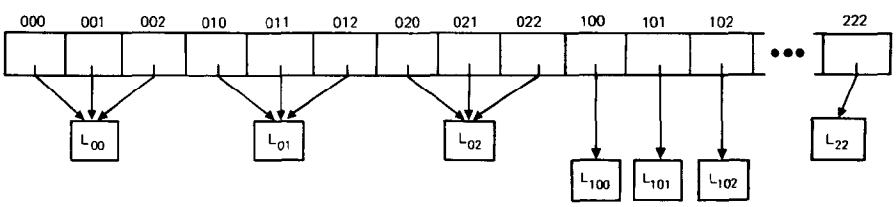

Fig. 3. Degenerate radix search tree

underlying alphabet at each node), we flatten out the top d levels into an array of r^d pointers; by using the d leftmost (= most significant) digits of the key as an index into this directory, we achieve a fanout of r^d at the root. For $d = 2$ our example is displayed in Figure 2.

(c) If we can afford to waste some space for redundant information, then we may extend the directory to a greater depth, i.e. to cover more levels than the shortest root-to-leaf path justifies, thus trading memory for speed. This happens in our example if we choose $d = 3$ (see Figure 3). Notice that each leaf at depth 2 (or level 2) in the tree is being pointed at from three different entries in the directory; only the leaves at depth 3 make full use of the expanded length of the directory.

(d) So far we have considered radix search trees whose top d levels have been flattened into a directory of depth d; leaves at level $l \leq d$ are then accessed with a single probe, i.e. by following a single pointer from the directory (= root of the tree). Leaves at level $l > d$, on the other hand, would require more than one probe, as Figure 2 shows. By choosing the depth of the directory sufficiently large ($d \geq$ the length of the longest root-to-leaf path) we can guarantee access in a single probe to any leaf. The radix search tree has degenerated into a direct (= one-step) access mechanism. Using the key as an address yields the ultimate in speed at a usually extravagant cost in memory—unless the space from which the keys are drawn is uncharacteristically small, or (and this is the key observation) the keys in the file are uniformly spread over the key space.

(e) Hash functions have been used for two decades to convert a nonuniform, usually unknown, distribution into another one which one hopes is close to uniform. Only recently Carter and Wegman [4] have given a mathematical foundation to this hope. Armed with this insight, we now envision the following file organization scheme which is both extendible and balanced (see Figure 4).

This is a summary of the ideal picture. The details are described in Section 4. An analysis which justifies the high expectations mentioned above occurs in Sections 5 and 6. Let us now describe how the same goal can be reached by another design process, which starts with conventional hash tables and tries to extend them.

3.2 Extending Hash Tables

Thesis. Hash tables are naturally balanced. By separating the hash address space from the directory address space, hash tables can be made extendible.

Hashing (or key-to-address transformation, or scatter storage techniques) is

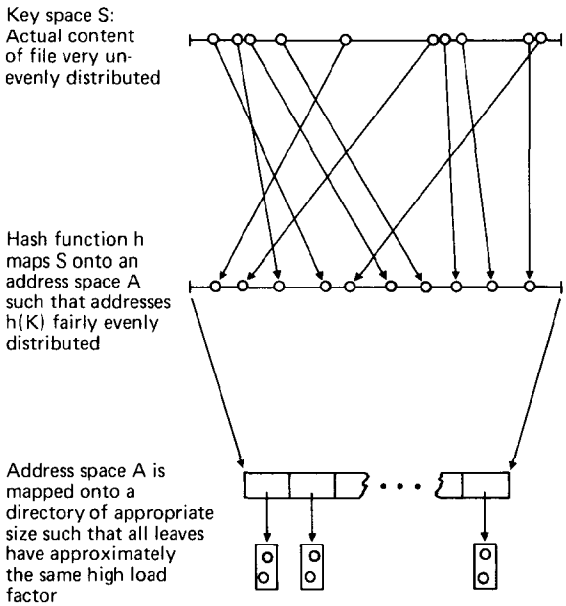

Fig. 4. Radix search tree being accessed through a hash function

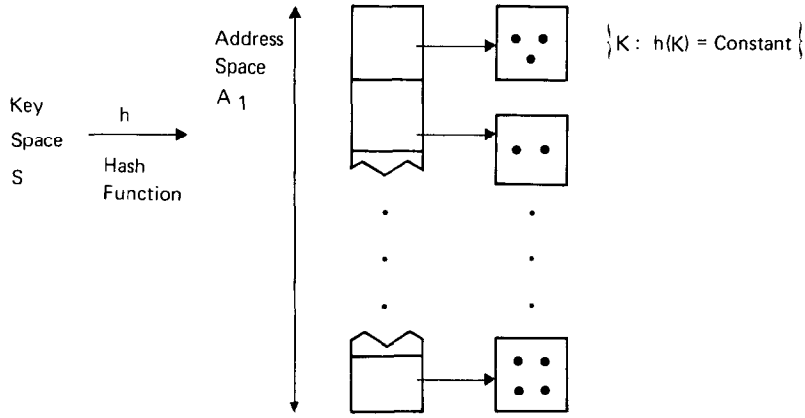

Fig. 5. Hashing into a directory

recognized in practice as providing the fastest random access to a file. This empirical evidence is supported by theoretical analysis, which indicates that access time to a hash table is independent of the number of records; instead, access time depends on the load factor of the table, and in practice load factors as high as 90 percent allow hashing to be competitive with other access schemes.

In contrast to the fast $O(1)$ access time, hashing is burdened with two disadvantages that prevent its use in many applications. First, hashing usually cannot support sequential processing of a file according to the natural order on the keys. Sequential processing requires sorting, an $O(n \log n)$ operation which makes the fast random access useless. Second, traditional hash tables are not extendible—their size is intimately tied to the hash function used, and often must be determined before one knows how many records are to be placed in it. 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.

Because of the two preceding disadvantages, hashing has usually been confined to tables which fit into main memory, and whose size can be estimated reliably. Where such a table is a directory of a file stored on disk, the necessary file maintenance algorithms to make the file organization scheme truly dynamic, in the sense described in Section 2, have not previously been worked out.

In this section we describe a broad class of file organization schemes based on hashing which are extendible in the sense of Section 2. They also go part way toward solving the first traditional shortcoming of hash tables: They support sequential processing to a limited extent. More specifically, it is possible to process the keys in hash order, without referencing the same page more than once. Let the following examples illustrate the design approach.

(a) Consider a hash table organized as a directory with address space A_1, with each entry of the directory pointing to a bucket (= leaf page) of fixed size (see Figure 5). This traditional picture has the disadvantage that it does not suggest a way of making the file extendible: When a bucket overflows, because too many keys K arrive with $h(K)$ equal to a given address α, there appears to be no alternative to rehashing.

(b) The following figure serves as a starting point for generalization, since it contains an additional component that can be manipulated (see Figure 6). The hash function maps the key space S onto a large address space A_1. A partition π splits A_1 into m blocks; each block has one leaf allocated for its use and the directory somehow implements the correspondence between blocks and leaves. Assuming that π is defined by $m + 1$ boundaries $\alpha_0, \alpha_1, \ldots, \alpha_m$, leaf L_i contains all keys K with $\alpha_{i-1} \leq h(K) < \alpha_i$. The added flexibility of this scheme is shown by the following possibilities: if a leaf overflows, we may be able to change the partition, perhaps by as little as shifting one boundary α_i, and relocating only those keys that are affected by this shift. Notice that h need not be changed.

(c) We are thus led to make the hash table extendible by varying a partition π on a large address space A, while keeping the hash function unchanged. The question arises immediately as to what kind of partitions can be efficiently managed. Since we aim at a very large, theoretically unbounded, capacity of the entire file, while keeping the bucket capacity constant, the partitions we deal with must have a variable number of blocks. Among many conceivable families of partitions, the well-known "buddy system" for storage management (see, for example, Knuth [9, Vol. 1, p. 442]) suggests itself immediately because of its simplicity. Let $A = \{0, \ldots, 2^n - 1\}$ for some large fixed n; then $\alpha_0 = 0 < \alpha_1 < \alpha_2 < \ldots < \alpha_m = 2^n - 1$ are the boundaries of a partition of the buddy-system type iff all intervals $[\alpha_{i-1}, \alpha_i)$ can be obtained by repeated halving of intervals in A. The example in Figure 7 shows a buddy-system partition with $n = 3$.

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 gets underfilled because of deletions, and its buddy has enough room, these two blocks can easily be merged into a buddy-system partition with one block less.

(d) There remains the question of how a buddy-system partition is efficiently implemented in a directory. Again there is an obvious efficient solution. 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 the 2^d equal-sized intervals obtained by continued halvings. Thus d is minimal such that for each block

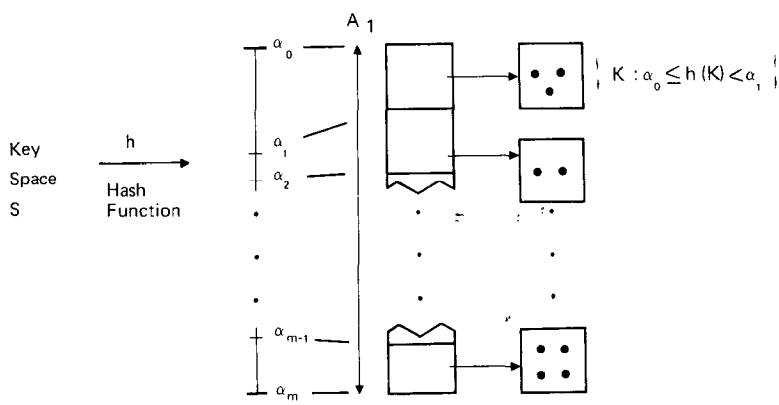

Fig. 6. Hashing into a large address space

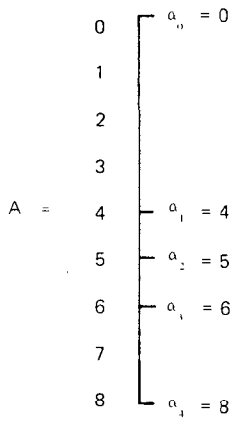

Fig. 7. A partition of the buddy-system type

$[\alpha_{i-1}, \alpha_i)$ of the partition, $(\alpha_i - \alpha_{i-1}) \geq 2^{n-d}$. A directory with 2^d entries, some of which may point to the same bucket, allows one to take the d most significant bits of the hash address $h(K)$ as the index in the address space $A_1 = \{0, \ldots, 2^d - 1\}$ of the directory. When the depth of a partition increases, then the directory doubles in size.

(e) The attentive reader will have noticed that we have now arrived at precisely the same scheme developed in Section 3, starting from a radix search tree; it is presented here for the special case where the radix $r = 2$, which is natural if one thinks of hash addresses as bit strings. The details of the file organization scheme thus found at the intersection of two distinct approaches are presented in Section 4.

3.3 Balance Versus Sequentiality

Two points mentioned earlier in this section remain to be discussed: balancing and sequential processing. Balancing, in the case of a two-level tree as we have in extendible hashing, merely means that the occupancy of leaf pages is bounded above and below. Sequentiality can mean 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 (e.g. lexicographic order) defined on the space of keys. Either of these desirable goals (balance and sequentiality) can be achieved in extendible hash tables; both can be achieved simultaneously to some extent, but not fully.

Balancing is achieved in two distinct ways. First, partitioning the address space A into blocks of variable length achieves a balancing effect regardless of the distribution of hash addresses $h(K)$ over A: in regions where hash addresses of keys in the table cluster, the partition is finer than in sparsely populated regions. Second, a main purpose of hash functions in general is to distribute a set of keys that is nonuniformly distributed over the key space S uniformly over an address space A.

Sequentiality in the weak sense (by hash address or "pseudokey") is trivially

achieved with extendible hashing. For many applications, sequentiality in pseudokey order is just as good as sequentiality in key order. For example, consider a batch-update application, where there is a master file and a (presumably smaller) update file. The updates are "batched together" into an update file, which is then used to update the master. An efficient procedure for actually applying the update to the master file is to always store the master file in sorted order, then to sort the updates by key, and then to apply the updates to the master. In this way, no page of the master file is retrieved more than once. In this application, one can just as well sort in pseudokey order as in key order. As another example, it is often important to do a "first, next, next, next, . . ." and touch every record (or key) exactly once. Again, pseudokey order will do just as well as key order.

Sequentiality in natural order, that is, by key, tends to conflict with balancing: a compromise is the result. Hash functions that tend to distribute hash addresses uniformly over A, even for biased sets of keys with many clusters, ignore (destroy) the natural order on the keys (whence the name "hash"). If such a conventional hash function is used in order to improve the balance, then extendible hashing provides sequentiality only in the weak sense. Note, however, that it is possible to store the set of keys *within each leaf* in natural order, so that sequential processing in natural order can be obtained for the cost of merging all leaves, as opposed to sorting the entire file.

Order-preserving hash functions, which satisfy the condition "$K < L$ iff $h(K) < h(L)$" permit sequential processing in natural order. They are rarely used in practice because they do not sufficiently break up clusters of adjacent names, and thus fail to provide a uniform occupancy of the address space. Since extendible hashing induces a partition of the address space into *variable-length* blocks, the occupancy of leaf pages (buckets) can be made to be significantly more uniform than the occupancy in the address space is. Hence order-preserving hash functions should be seriously considered as a means of allowing true sequential access in extendible hash tables.

4. A SPECIFIC EXTENDIBLE HASHING SCHEME

In this section we describe in more detail one extendible hashing scheme. Probably its most important performance characteristic is its speed. Even for files that are very large by current standards, there are never more than two page faults necessary to locate a key and its associated information.

We assume that we are given a fixed hash function h. If K is a key, then we call $K' = h(K)$ the *pseudokey* associated with K. We choose pseudokeys to be of fixed length, such as 32 bits. A good choice for the hash function h is one randomly selected from a universal class of hash functions, as defined by Carter and Wegman [4]. 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. Note that although the pseudokeys are of fixed length, the keys need not be.

The file is structured into two levels: directory and leaves. The leaves contain pairs $(K, I(K))$, where K is a key, and $I(K)$ is associated information: either the record associated with K, or a pointer to the record.

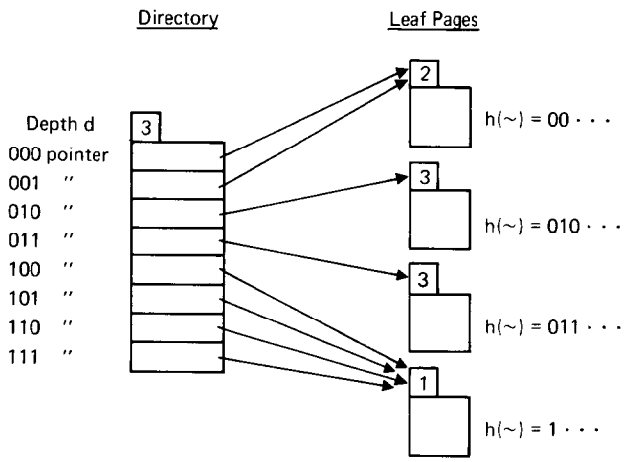

Fig. 8

The directory has a header, in which is stored a quantity called the *depth d* of the directory. After the header, the directory contains pointers to leaf pages. The pointers are laid out as follows. First, there is a pointer to a leaf that stores all keys K for which the pseudokey $K' = h(K)$ starts with d consecutive zeros. This is followed by a pointer for all keys whose pseudokeys begin with the d bits $0 \ldots 01$, and then a pointer for all keys whose pseudokeys begin $0 \ldots 010$, and so on lexicographically. Thus altogether there are 2^d pointers (not necessarily distinct), and the final pointer is for all keys whose pseudokey begins with d consecutive ones. If $d = 3$, then the directory looks like the left side of Figure 8.

Assume that we want to locate key K_0 and its associated information. Calculate $h(K_0)$, and find its first d bits. Do a simple address computation to find the location in the directory of the pointer that corresponds to this d-bit prefix. If we follow this pointer, then we find a leaf page that contains $(K_0, I(K_0))$, provided K_0 is a key in the file at the moment.

Each leaf page has a header that contains a *local depth d'* for the leaf page. For example, if we follow the 000 pointer in the directory of Figure 8, we reach a leaf page with local depth 2. Local depth 2 means that not only does this leaf page contain all keys whose pseudokey begins with 000, but even more, it contains all keys whose pseudokey begins with the 2 bits 00. Thus the 001 pointer also points to this leaf page. The depth of the directory is the maximum of the local depths of all of the leaf pages.

In our example of Figure 8, there are so few keys whose pseudokey begins with a 1 that there is a single leaf page (with local depth 1) associated with all such keys. What happens when this leaf page finally overfills (or reaches a predetermined unacceptably full level, such as 90 percent full)? Then as in Figure 9, it "splits" into two leaf pages, each with local depth 2. All keys whose pseudokey begins 10 appear on the first of these leaf pages, and all keys whose pseudokey begins 11 appear on the other.

What happens if a leaf page overfills, and the local depth of the leaf page already equals the depth of the directory? Then the directory doubles in size, its depth increases by 1, and the leaf page splits. For example, if we start with the

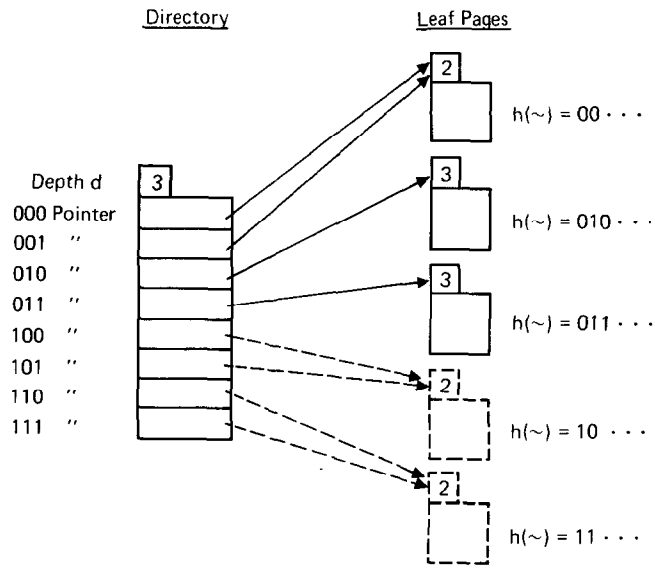

Fig. 9

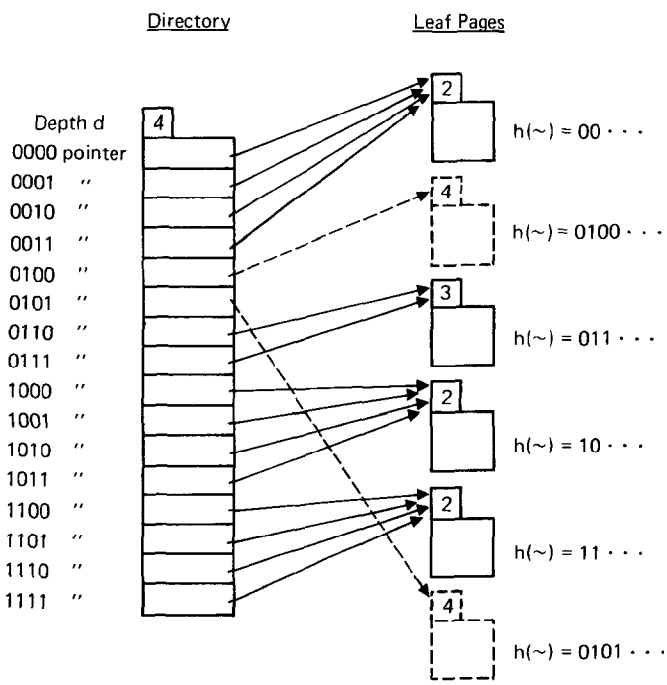

Fig. 10

situation as in Figure 9, and if the leaf page pointed to by the 010 pointer overfills, then we get Figure 10. This process of doubling the directory is not expensive because no leaf pages need to be touched (except, of course, for the leaf page that caused the split and its new sibling). There is an easy, essentially one-pass algorithm for doubling the directory, that proceeds by working from the bottom

of the old directory up to the top of the old directory. The simple details are left to the reader. If there are so many keys that the directory is in secondary storage, then since the directory is stored contiguously, 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 the IBM 3330 disk), then it is straightforward to estimate that the time involved in doubling the directory (which is mainly data transfer time) would be less than a second if there were 400 keys per leaf page. Even in the extreme case of a billion keys, the time involved in doubling the directory would be less than a minute.

We note that if we had used suffixes of pseudokeys instead of prefixes, then the algorithm for doubling the directory would be especially easy: it would essentially consist of making a second copy of the nonheader portion of the directory, immediately after the first copy. However, we chose to use prefixes for the sake of intuitive simplicity (thus, by using prefixes the keys can easily be accessed in pseudokey order, rather than in inverted pseudokey order).

The internal structure of the leaves is independent of the relationship between the pages. In the interest of speed, we choose to organize the leaves as (traditional) hash tables. It is natural to use the "ignored" bits of the pseudokey K' to hash within the page. Any standard collision-resolution technique, such as open addressing or chaining, is acceptable, as long as it stores colliding keys within the same page.

If deletions form such a large proportion of the operations of an application that space will be saved by coalescing pages, then 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 the 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.

There is at most one page fault in locating the appropriate directory page, because the structure of the directory is so simple that 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 many natural situations the directory will be so small that it can be kept resident in main memory. For example, if the page size is 4K bytes, if keys are 7 bytes long and pointers to pages are 3 bytes long, then after a million inserts, the directory can be expected to be 3 pages in size.

A number of advantages accrue from the simple, intuitive structure of extendible hashing. The most obvious is the simplicity of coding (thus leading to lower likelihood of "bugs"). Our extendible hashing algorithm is easily modified to accommodate individual needs: for example, it might be desirable in some contexts to "initialize" by starting with a directory depth d greater than zero and individually initializing 2^d leaf pages.

We close this section by giving in more detail the algorithms for extendible hashing.

ACCESS (given key K_0)

1. Calculate $K_0' = h(K_0)$.
2. Read d, the depth of the directory.
3. Take the initial d bits of K_0', interpret them as an integer base 2, and call this number r.
4. Let v be the length in bytes of the region (one for each pointer in the directory) that tells the numbber of entries on that leaf page. If this information is not being stored in the directory then let $v = 0$.
5. Find the pointer that is located $r(l + v)$ bytes from the start of the nonheader portion of the directory, when l is the length of each pointer in bytes.
6. Follow this pointer to a leaf page P.
7. Use the *trailing* s bits of the pseudokey to hash onto leaft page P (where s is a fixed, system-determined parameter).
8. If necessary, follow the collision-resolution scheme within page P.

INSERT (given $(K_0, I(K_0))$)

1. Apply the first seven steps of ACCESS, using key K_0.
2. If by inserting key K_0 on leaf page P, we would exceed our threshold, then go to Step 7.
3. If there is sufficient free space at the location calculated at the end of Step 1, then insert $(K_0, I(K_0))$ there.
4. Otherwise, follow the collision-resolution scheme to insert $(K_0, I(K_0))$ on leaf page P, if this is possible.
5. (Optional) For each directory pointer that points to page P, increment by one the entry that tells the number of entries on the leaf page.
6. If $(K_0, I(K_0))$ has been successfully inserted, then stop.
7. At this point, we know there is not sufficient free space on page P. Obtain a new page P^* to use as a leaf page.
8. Obtain a temporary area Q to store all $(K, I(K))$ pairs that appeared on page P, along with the new $(K_0, I(K_0))$.
9. Set the local depth of each of P and P^* to $d' + 1$, where d' is the old local depth of P.
10. Erase all nonheader information from page P.
11. If the new local depth of P is bigger than the current directory depth, then do the following.
    a. Increase the depth of the directory by one.
    b. Double the size of the directory, and update the pointers in the obvious manner.
    c. (Optional) Set to zero the entry giving the number of entries on the leaf pages P and P^*.
12. INSERT all $(K, I(K))$ pairs one at a time from the temporary area Q.

Note that the INSERT routine can (repeatedly) call itself recursively (in Step 12).

DELETE (given K_0)

1. ACCESS, using K_0.
2. If K_0 does not appear, then stop (and send the appropriate return code).
3. Delete by writing the deleted sign over the entry or by unchaining, depending on the collision-resolution strategy.
4. (Optional) If the sum of the number of entries on this page and its sibling page are below the threshold, then coalesce these two pages as follows:
    a. Copy all $(K, I(K))$ entries from these two pages into a temporary region Q.
    b. Throw away (i.e. return to free space) one of the two pages. Make all pointers that point to it point to the remaining page.
    c. Decrement the depth on the remaining page P by one.
    d. Erase all $(K, I(K))$ entries on page P.
    e. Set to zero the "number of entries on page" values associated with all pointers to P.
    f. INSERT all $(K, I(K))$ pairs one at a time from the temporary area Q.
5. (Optional) If every pointer in the directory equals its sibling pointer, then do the following:

a. Decrease the depth of the directory by one.
b. Halve the size of the directory, and update the pointers in the obvious manner.

5. ANALYSIS

In this section we shall derive some analytical results concerning the number of leaf pages and directory entries used by extendible hashing. As with all hashing schemes, performance in the worst case is intolerable; it is possible, for example, for a file with just one more record than will fit into a leaf page to cause the directory to expand until there is a separate entry for every possible pseudokey! The probability of this happening is astronomically small, of course, and in speaking of "probability" here we need not entrust our fate to the source of our file; for *any* file, we may take the dice into our own hands and choose our hashing function at random (see Markowsky, Carter, and Wegman [12]). In describing our insertion algorithm we assumed that the index contained no duplicate keys and that an index page was never filled with duplicate pseudokeys. In fact, if the hashing function maps even two distinct keys onto the same pseudokey, this collision might be considered an indication that either the space of pseudokeys is not large enough or that we have been unlucky in our choice of hash function and should choose again from the class of hash functions available. The results of Markowsky, Carter, and Wegman [12] allow us to make conservation estimates of the probability of collisions in the pseudokey space independent of the distribution of keys in the key space. For example, if pseudokeys are 128 bits long, the probability of even a single collision in filling an index with one billion inserts is less than one quadrillionth (10^{-15}).

Our interest in what follows will be in average performance; to study this we shall assume natural probability distributions, setting aside the question of whether the randomness is provided by the source of the file, the choice of the hashing function, or (as will usually be the case) some combination of the two.

For the analysis of average performance, it is traditional to assume that the file has some particular number of records and that these records have uniformly and independently distributed pseudokeys. This will be called the Bernoulli model in what follows. The best way to handle this model seems to be to start with another model, in which the pseudokeys are again uniformly and independently distributed, but in which the number of records is itself a random variable. This will be called the Poisson model in what follows. Our strategy will be to analyze the Poisson model, then show that the Bernoulli model can be reduced to the Poisson model. Although our main interest is in the Bernoulli model (since this allows comparisons with simulations and other published analyses), the Poisson model is of some interest in its own right: if, for example, records arrive for insertion with exponentially distributed interarrival times and are deleted after arbitrarily distributed lifetimes, then the equilibrium distribution follows the Poisson model (this is the situation $M/G/\infty$ in ther terminology of queueing theory; see Khinchine [7, Section 25]).

THE BERNOULLI MODEL. For this model the number of records has a deterministic value, say n. If we consider a pseudokey interval of length p, the number J of records whose pseudokeys fall in this interval is a Bernoulli distributed random variable:

$$P(J = j) = \binom{n}{j} p^j (1-p)^{n-j}.$$

More generally, if we consider r disjoint intervals with lengths $p_1, \ldots, p_r$, the numbers $J_1, \ldots, J_r$ of records whose pseudokeys fall in these intervals have the joint distribution

$$P(J_1 = j_1, \ldots, J_r = j_r) = \binom{n}{j_1 \ldots j_r} p_1^{j_1} \ldots p_r^{j_r} (1-p)^{n-j},$$

where $p = p_1 + \cdots + p_r$ and $j = j_1 + \cdots + j_r$.

THE POISSON MODEL. For this model the number N of records is a Poisson distributed random variable:

$$P(N = n) = e^{-\nu} \nu^n / n!.$$

The parameter ν is the average number of records. The number J of records whose pseudokeys fall in an interval of length p is also Poisson distributed:

$$\begin{aligned} P(J = j) &= \sum_{j \le n < \infty} (e^{-\nu} \nu^n / n!) \binom{n}{j} p^j (1-p)^{n-j} \\ &= (e^{-\nu} \nu^j p^j / j!) \sum_{j \le n < \infty} \nu^{n-j} (1-p)^{n-j} / (n-j)! \\ &= e^{-\nu p)} \nu p)^j / j!. \end{aligned}$$

More generally, the numbers $J_1, \ldots, J_r$ of records whose pseudokeys fall in disjoint intervals with lengths $p_1, \ldots, p_r$ are independently Poisson distributed:

$$\begin{aligned} P(J_1 = j_1, \ldots, J_r = j_r) &= \sum_{j \le n < \infty} (e^{-\nu} \nu^n / n!) \binom{n}{j_1 \ldots j_r} p_1^{j_1} \ldots p_r^{j_r} (1-p)^{n-j} \\ &= (e^{-\nu} \nu^j p_1^{j_1} \ldots p_r^{j_r} / j_1! \ldots j_r!) \sum_{j \le n < \infty} \nu^{n-j} (1-p)^{n-j} (n-j)! \\ &= (e^{-\nu p_1} (\nu p_1)^{j_1} / j_1!) \ldots (e^{-\nu p_r} (\nu p_r)^{j_r} / j_r!). \end{aligned}$$

The aspects of extendible hashing that we shall study are closely related to a variant of radix-exchange sorting. If we assume that leaf pages split when they contain more than m records, we should assume that the sorting routine calls itself recursively when there are more than m records to be sorted, but terminates nonrecursively when there are m or fewer records. There will then be a one-to-one correspondence between the leaf pages in extendible hashing and the terminal invocations of the sorting routine. Straight radix-exchange sorting (the case $m = 1$) has been analyzed by Knuth [9, Vol. 3, Section 5.2.2] and it is natural to try to extend that analysis to $m \ge 2$. This is done in [9, Vol. 3, Section 6.3, Exercises 19 and 20], but the form in which the results are given there is unsuitable for our purposes, since it involves a sum of m different Fourier series. The method we use in what follows leads straightforwardly to a single Fourier series. Although it gives coarser error terms, it gives a natural interpretation for the main approxi-

mation involved: a Bernoulli distribution is approximated by a Poisson distribution.

5.1 The Poisson Model

We begin with the average number of leaf pages. Consider a binary tree that has the "ghosts" of pages that have split as its internal nodes and has the current leaf pages as its external nodes (see Figure 11).

The number of leaf pages is greater by one than the number of ghosts. A ghost at level k (the root is at level 0) corresponds to a pseudokey interval of length 2^{-k} that contains more than m records. Since there are 2^k potential ghosts at level k, the average number of leaf pages is

$$1 + \sum_{0 \le k < \infty} 2^k P(>m, 2^{-k}), \tag{5.1.1}$$

where $P(>m, 2^{-k})$ denotes the probability that a pseudokey interval of length 2^{-k} contains more than m records.

Substituting the Poisson distribution (with average number of records as ν) into (5.1.1) yields

$$1 + \sum_{0 \le k < \infty} 2^k \sum_{m < j < \infty} e^{-\nu 2^{-k}} (\nu 2^{-k})^j / j!. \tag{5.1.2}$$

We shall show that if m remains fixed and $\nu \to \infty$ this expression behaves like

$$(\nu/m)\phi_m(\log \nu) + O(\nu^{1/2} \log \nu), \tag{5.1.3}$$

where

$$\phi_m(x) = \sum_{-\infty < h < +\infty} c_{m,h} e^{-2\pi i h x} \tag{5.1.4}$$

and

$$c_{m,h} = (\log e)(m - 1 + 2\pi i h \log e)!/(1 - 2\pi i h \log e)(m - 1)!. \tag{5.1.5}$$

The logarithms are to the base 2.

A few words concerning this result are in order. The average number of records is ν; if these records were packed m to a page, they would occupy ν/m pages. In the expression (5.1.3), $\phi_m(\log \nu)$ should therefore be interpreted as the storage expansion ratio: the ratio of the average number of pages for this algorithm to

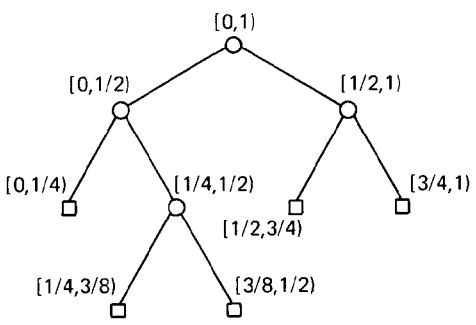

Fig. 11

the average of the minimum possible number. The Fourier series (5.1.4) shows this function to be periodic with period 1, so when ν doubles the expansion ratio returns to its original value. The constant term (corresponding to $h = 0$) is $\log e = 1.442\ldots$; this is the "average" value of $\phi_m(\log \nu)$ if ν is "distributed" according to the logarithmic law (so that $\log \nu$ is uniformly distributed modulo 1). The first harmonic terms (corresponding to $h = \pm 1$) are smaller in magnitude by at least a factor of $(1 + (2\pi \log e)^2)^{1/2}$, since $|(x + iy)!| \leq x!$; the succeeding harmonic terms continue to decrease in magnitude, and eventually they decay exponentially.

To evaluate (5.1.2), we write it as

$$1 + \sum_{m<j<\infty} (\nu^j/j!) \sum_{0\leq k<\infty} 2^{k(1-j)} e^{-\nu 2^{-k}}$$

and substitute the integral representation

$$e^{-w} = (1/2\pi i) \int_{-1/2-i\infty}^{-1/2+i\infty} (z!/w^{z+1})\, dz.$$

This gives

$$1 + \sum_{m<j<\infty} (\nu^j/j!) \sum_{0\leq k<\infty} 2^{k(1-j)} (1/2\pi i) \int_{-1/2-i\infty}^{-1/2+i\infty} [z!/(\nu 2^{-k})^{z+1}]\, dz$$

$$= 1 + \sum_{m<j<\infty} (\nu^j/j!)(1/2\pi i) \int_{-1/2-i\infty}^{-1/2+i\infty} [z!/\nu^{z+1}(1 - 2^{z-j+2})]\, dz.$$

To evaluate this integral, let us consider the poles of the integrand. There are poles at $z = -1, -2, \ldots$ due to $z!$, and poles at $z = j - 2 + 2\pi i h \log e$ (for $h = \ldots, -1, 0, +1, \ldots$) due to the zeros of $1 - 2^{z-j+2}$. If the path of integration is shifted to the right of the latter poles, the value of the integral is augmented by the sum of the residues at these poles. This gives

$$1 + \sum_{m<j<\infty} (\nu^j/j!) \sum_{-\infty<h<+\infty} (\log e)(j - 2 + 2\pi i h \log e)!\, \nu^{-j+1-2\pi i h \log e}$$

$$+ \sum_{m<j<\infty} (\nu^j/j!)(1/2\pi i) \int_{j-3/2-\epsilon-i\infty}^{j-3/2-\epsilon+i\infty} [z!/\nu^{z+1}(1 - 2^{z-j+2})]\, dz,$$

where ϵ is a small parameter ($0 < \epsilon < \frac{1}{2}$) which will be chosen later. To complete the derivation of (5.1.3) we shall show that the double sum is $(\nu/m)\phi_m(\log \nu)$ and that the remaining sum is $O(\nu^{1/2} \log \nu)$.

The double sum can be rewritten as

$$\nu \sum_{-\infty<h<+\infty} \nu^{-2\pi i h \log e} \sum_{m<j<\infty} (\log e)(j - 2 + 2\pi i h \log e)!/j!$$

$$= \nu \sum_{-\infty<h<+\infty} e^{-2\pi i h \log \nu} (\log e) \sum_{0\leq j<\infty} (m + j - 1 + 2\pi i h \log e)!/(m + j + 1)!.$$

The inner sum is the hypergeometric series

$$F(m + 2\pi ih \log e, 1; m + 2; 1)(m - 1 + 2\pi ih \log e)!/(m + 1)!$$
$$= (m - 1 + 2\pi ih \log e)!/(1 - 2\pi ih \log e)m!.$$

This gives

$$\nu \sum_{-\infty<h<+\infty} e^{-2\pi ih \log \nu}(\log e)(m - 1 + 2\pi ih \log e)!/(1 - 2\pi ih \log e)m!$$
$$= (\nu/m) \sum_{-\infty<h<+\infty} c_{m,h} e^{-2\pi ih \log \nu}$$
$$= (\nu/m)\phi_m(\log \nu).$$

For the remaining sum we have

$$(1/2\pi i) \int_{j-3/2-\epsilon-i\infty}^{j-3/2-\epsilon+i\infty} [z!/\nu^{z+1}(1 - 2^{z-j+2})]\, dz = O((j - 1 - \epsilon)!/\nu^{j-1/2-\epsilon}(\tfrac{1}{2} - \epsilon))$$

and

$$\sum_{m<j<\infty} (\nu^j/j!)O((j - 1 - \epsilon)!/\nu^{j-1/2-\epsilon}(\tfrac{1}{2} - \epsilon)) = O(\nu^{1/2+\epsilon}/\epsilon(\tfrac{1}{2} - \epsilon)).$$

By choosing $\epsilon = 1/\ln \nu$ we obtain $O(\nu^{1/2} \log \nu)$.

Let us now consider the depth and the number of directory entries used by extendible hashing. We have seen that there are about $(\nu/m)\log e = \nu/m \ln 2$ leaf pages on the average; if these all appeared at two successive levels, these two levels would be

$$a = \lfloor \log (\nu/m \ln 2) \rfloor$$

and

$$b = \lceil \log (\nu/m \ln 2) \rceil,$$

and the directory would have

$$2\lceil \log (\nu/m \ln 2) \rceil$$

entries. This last expression can be written as $(\nu/m)\psi_m(\log \nu)$, where $\psi_m(x)$ is a periodic function with period 1. This function has a Fourier series with constant term $(\log e)^2 = 2.079 \ldots$, so there would be $(\nu/m)(\log e)^2 = \nu/m(\ln 2)^2$ directory entries on the average.

It does not always happen, of course, that all the leaf pages appear at two successive levels; we shall show, however, that it happens with probability very nearly 1 for practical values of m and ν. First, consider the probability that there is a leaf page at level $a - 1$ or less. This can happen if one of the 2^{a-1} potential ghosts at level $a - 1$ fails to be a ghost; the probability of this is at most

$$2^{a-1}[1 - P(>m, 2^{1-a})] = 2^{a-1} \sum_{0\le j\le m} e^{-\nu 2^{1-a}}(\nu 2^{1-a})^j/j!.$$

Since

$$2^{a-1} \le \nu/2m \ln 2$$

and

$$\nu 2^{1-a} \ge 2m \ln 2,$$

this expression is at most

$$(\nu/2m \ln 2) \sum_{0 \le j \le m} e^{-2m \ln 2}(2m \ln 2)^j/j!$$

$$= (\nu/2m \ln 2)[e^{-2m \ln 2}(2m \ln 2)^m/m!] \sum_{0 \le j \le m} (2m \ln 2)^{j-m} m!/j!.$$

Bounding the sum by a geometric series, we find that this expression is at most

$$(\nu/m(2 \ln 2 - 1))[e^{-2m \ln 2}(2m \ln 2)^m/m!],$$

and using the inequality $m! \ge m^m e^{-m}(2\pi m)^{1/2}$ we find that this expression in turn is at most

$$\nu/m^{3/2}(2\pi)^{1/2}(2 \ln 2 - 1)(2/e \ln 2)^m. \tag{5.1.6}$$

By similar reasoning, if there is a leaf page at level $b + 1$ or more, one of the 2^b potential ghosts at level b must actually be a ghost; the probability of this is at most

$$\begin{aligned} 2^b P(>m, 2^{-b}) &= 2^b \sum_{m<j<\infty} e^{-\nu 2^{-b}}(\nu 2^{-b})^j/j! \\ &\le (2\nu/m \ln 2) \sum_{m<j<\infty} e^{-m \ln 2}(m \ln 2)^j/j! \\ &\le (2\nu/m \ln 2)[e^{-m \ln 2}(m \ln 2)^m/m!] \sum_{m<j<\infty} (m \ln 2)^{j-m} m!/j! \\ &\le (2\nu/m(1 - \ln 2))[e^{-m \ln 2}(m \ln 2)^m/m!] \\ &\le 2\nu/m^{3/2}(2\pi)^{1/2}(1 - \ln 2)(2/e \ln 2)^m. \end{aligned} \tag{5.1.7}$$

The bounds (5.1.6) and (5.1.7) are both of the form ν/E_m, where E_m grows exponentially with m. This means that when m is moderately large, these bounds are very small unless ν is very large. If $m = 200$, for example, (5.1.7) is less than $\nu/130{,}000{,}000$, and (5.1.6) is smaller still. For practical values of m and ν, then, the depth and number of directory entries can be predicted with considerable confidence.

If m remains fixed and $\nu \to \infty$, however, we have no satisfactory estimates for the average depth and the average number of directory entries. The depth will exceed k if and only if there is a ghost at level k; the probability of this is

$$Q(>k) = 1 - [1 - P(>m, 2^{-k})]^{2^k}.$$

(We have used the fact that disjoint pseudokey intervals contain independent numbers of records.) Thus the average depth is

$$\sum_{0 \le k < \infty} Q(>k) = \sum_{0 \le k < \infty} \left\{1 - \left[1 - \sum_{m<j<\infty} e^{-\nu 2^{-k}}(\nu 2^{-k})^j/j!\right]^{2^k}\right\}$$

and the average number of directory entries is

$$1 + \sum_{0 \le k < \infty} 2^k Q(>k) = 1 + \sum_{0 \le k < \infty} 2^k \left\{1 - \left[1 - \sum_{m<j<\infty} e^{-\nu 2^{-k}}(\nu 2^{-k})^j/j!\right]^{2^k}\right\}.$$

These sums have us stumped. It is natural to suppose that they are $O(\log \nu)$ and $O(\nu)$, but we have not succeeded in obtaining bounds better than $O(\nu)$ and $O(\nu^2)$; these bounds are obtained by applying $Q(>k) \le 2^k P(>m, 2^{-k})$ and proceeding as for the average number of leaf pages.

5.2 The Bernoulli Model

We again begin with the average number of leaf pages. Substituting the Bernoulli distribution into (5.1.1) yields

$$1 + \sum_{0 \le k < \infty} 2^k \sum_{m < j \le n} \binom{n}{j} 2^{-kj}(1 - 2^{-k})^{n-j}. \tag{5.2.1}$$

We shall show that this expression differs from

$$1 + \sum_{0 \le k < \infty} 2^k \sum_{m < j < \infty} e^{-n2^{-k}} (n2^{-k})^j / j! \tag{5.2.2}$$

by a term of the form $O(n^{2/3})$, and thus that the average number of leaf pages for the Bernoulli model is the same as that found in Section 5.1 for the Poisson model (with $\nu = n$), except for a slight deterioration of the error term. We shall go from (5.2.1) to (5.2.2) in three steps. First, we shall show that terms with k small or j large do not contribute much to (5.2.1), so these terms can be omitted without much effect. Second, we shall show that for the remaining terms the summand of (5.2.1) approximates that of (5.2.2). Finally, we shall show that terms with k small or j large do not contribute much to (5.2.2), so these terms can be restored without much effect.

To make this argument precise, let

$$k_0 = \log en^{2/3}$$

and

$$j_0 = en^{1/3}.$$

Expression (5.2.1) differs from

$$\sum_{k_0 \le k < \infty} 2^k \sum_{m < j \le n} \binom{n}{j} 2^{-kj}(1 - 2^{-k})^{n-j} \tag{5.2.3}$$

by

$$1 + \sum_{0 \le k < k_0} 2^k \sum_{m < j \le n} \binom{n}{j} 2^{-kj}(1 - 2^{-k})^{n-j};$$

this sum is $O(n^{2/3})$, since the inner sum (a Bernoulli probability) is at most 1. Expression (5.2.3) in turn differs from

$$\sum_{k_0 \le k < \infty} 2^k \sum_{m < j \le j_0} \binom{n}{j} 2^{-kj}(1 - 2^{-k})^{n-j} \tag{5.2.4}$$

by

$$\sum_{k_0 \le k < \infty} 2^k \sum_{j_0 < j \le n} \binom{n}{j} 2^{-kj}(1 - 2^{-k})^{n-j} = \sum_{j_0 < j \le n} \binom{n}{j} \sum_{k_0 \le k < \infty} 2^{k(1-j)}(1 - 2^{-k})^{n-j};$$

this sum is $O(n^{2/3}e^{-en^{1/3}})$, since $(1 - 2^{-k}) \leq 1$ and $\binom{n}{j} \leq (en/j)^j$.

Let us adopt the notation $U(f(n))$ for a factor of the form $e^{O(f(n))}$. The expression $U(f(n))$ is equivalent to $1 + O(f(n))$ when $f(n) \to 0$ as $n \to \infty$. Thus we have

$$\binom{n}{j} = n(n-1) \cdots (n-j+1)/j!$$
$$= U(n^{-1/3})n^j/j!$$

and

$$(1 - 2^{-k})^{n-j} = e^{(n-j)\ln(1-2^{-k})}$$
$$= U(n^{-1/3})e^{-n2^{-k}}$$

when $k_0 \leq k$ and $j \leq j_0$. Thus (5.2.4) can be rewritten in the form

$$U(n^{-1/3}) \sum_{k_0 \leq k < \infty} 2^k \sum_{m < j \leq j_0} e^{-n2^{-k}}(n2^{-k})^j/j!. \tag{5.2.5}$$

We have seen in Section 5.1 that the double sum in this expression (even extended over $0 \leq k < \infty$ and $m < j < \infty$) is $O(n)$, so (5.2.5) differs from

$$\sum_{k_0 \leq k < \infty} 2^k \sum_{m < j \leq j_0} e^{-n2^{-k}}(n2^{-k})^j/j! \tag{5.2.6}$$

by a term of the form $O(n^{2/3})$.

Expression (5.2.6) differs from

$$\sum_{k_0 \leq k < \infty} 2^k \sum_{m < j < \infty} e^{-n2^{-k}}(n2^{-k})^j/j! \tag{5.2.7}$$

by

$$\sum_{k_0 \leq k < \infty} 2^k \sum_{j_0 < j < \infty} e^{-n2^{-k}}(n2^{-k})^j/j! = \sum_{j_0 < j < \infty} (n^j/j!) \sum_{k_0 \leq k < \infty} 2^{k(1-j)}e^{-n2^{-k}};$$

this sum is $O(n^{2/3}e^{-en^{1/3}})$, since $e^{-n2^{-k}} \leq 1$ and $j! \geq (j/e)^j$. Expression (5.2.7) in turn differs from (5.2.2) by

$$1 + \sum_{0 \leq k < k_0} 2^k \sum_{m < j < \infty} e^{-n2^{-k}}(n2^{-k})^j/j!.$$

This sum is $O(n^{2/3})$, since the inner sum (a Poisson probability) is at most 1. From this chain of estimates we conclude that (5.2.1) and (5.2.2) differ by a term of the form $O(n^{2/3})$, and thus that average number of leaf pages for the Bernoulli model is essentially the same as that found in Section 5.1 for the Poisson model:

$$(n/m)\phi_m(\log n) + O(n^{2/3}).$$

Yao [19] has analyzed B-trees using the Bernoulli model; since the structure of a B-tree depends on the order in which the records are inserted, he made the assumption that all $n!$ orders of insertion are equally probable. He found that the average number of pages is $(n/m) \log e$ asymptotically, which is the same as the leading term in the expansion of $(n/m)\ \phi_m(\log n)$; the oscillations of $\phi_m(\log n)$ do not occur for B-trees, however.

Let us now reconsider the depth and the number of directory entries. The probability that there is a leaf page at a level less than

$$a = \lfloor \log(n/m \ln 2) \rfloor$$

is at most

$$2^{a-1}[1 - P(>m, 2^{1-a})] = 2^{a-1} \sum_{0 \le j \le m} \binom{n}{j} 2^{(1-a)j}(1 - 2^{1-a})^{n-j}$$

$$\le (n/2m \ln 2) \sum_{0 \le j \le m} \binom{n}{j} [(2m \ln 2)/n]^j [1 - (2m \ln 2)/n]^{n-j}.$$

Using results of Anderson and Samuels [2], it is easy to show that for $n \ge 2m \ln 2$ (that is, for $a \ge 1$) this Bernoulli probability is less than the corresponding Poisson probability, and thus that this expression is at most

$$(n/2m \ln 2) \sum_{0 \le m \le j} e^{-2m \ln 2} (2m \ln 2)^j / j! \le n/m^{3/2}(2\pi)^{1/2}(2 \ln 2 - 1)(2/e \ln 2)^m.$$

By similar reasoning, the probability that there is a leaf page at a level greater than

$$b = \lceil \log(n/m \ln 2) \rceil$$

is at most

$$2^b P(> m, 2^{-b}) = 2^b \sum_{m < j \le n} \binom{n}{j} 2^{-bj}(1 - 2^{-b})^{n-j}$$

$$\le (2n/m \ln 2) \sum_{m < j \le n} \binom{n}{j} [(m \ln 2)/n]^j [1 - (m \ln 2)/n]^{n-j}.$$

Again using the results of Anderson and Samuels [2], it is easy to show that for $n \ge m \ln 2$ this expression is at most

$$(2n/m \ln 2) \sum_{m < j < \infty} e^{-m \ln 2} (m \ln 2)^j / j! \le 2n/m^{3/2}(2\pi)^{1/2}(1 - \ln 2)(2/e \ln 2)^m.$$

Thus the probability bounds we derived in Section 5.1 for the Poisson model hold for the Bernoulli model as well.

6. SIMULATION

In order to analyze the performance of extendible hashing, we wish to estimate three performance factors ((1) expected access time, (2) expected insert time, and (3) total space required) as functions of the following database and system parameters: (a) database size (i.e. number of entries), (b) page size, (c) entry size, (d) directory entry size, (e) buffer size (number of pages resident in primary storage at a time), (f) expected page fault time, and (g) expected entry page search time (as a function of page occupancy). Analyzing at this level, we can compare the performance of extendible hashing with that of a typical B-tree model [3]. In this section we present sample results of a fairly detailed Monte Carlo simulation at this level plus performance estimates obtained from much simpler analytic models suggested by the simulation results and by results of Section 5.

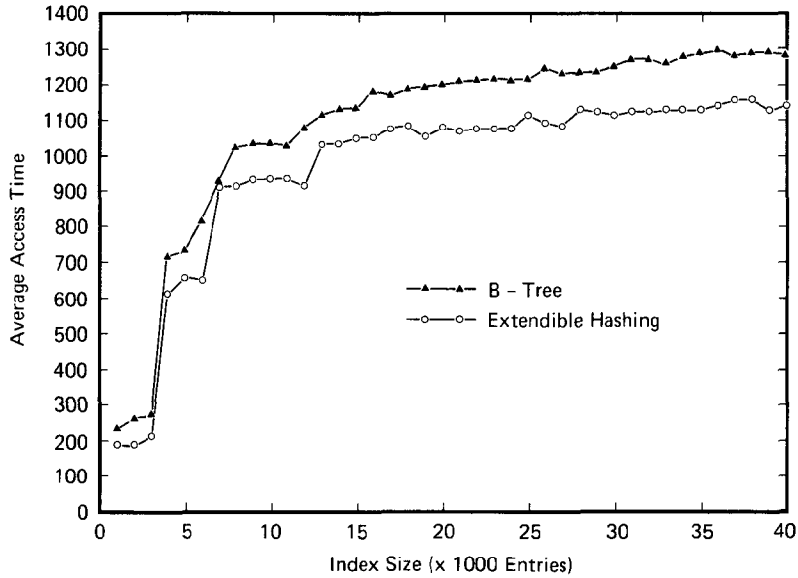

Fig. 12. Access time. Detailed simulation of access times averaged over 1000 accesses for index sizes in multiples of 1000 entries. The time to fetch a page is 1000. The maximum number of entries on a page is 400.

6.1 Detailed Models

We postulate a paged memory, with p equal to the maximum number of entries (key-pointer pairs) that can reside on a page. (For directory pages, the maximum occupancy will be d.) There is a buffer in primary memory that can hold b pages; and, whenever we require a page not in the buffer, there will be a page fault time cost of f. (No time except page fault time will be charged for searching the directory.) The time to search a page containing x entries will be $S(x)$. An approximation for $S(x)$ used in our simulations is $S(x)$ = (probe time) $\lceil \log_2 x \rceil$. This is an upper bound for extendible hashing and a lower bound for our B-tree model.

The total number of entries will be n. The parameters n, p, b, f, and S are common to both detailed models. We simulated the insertion of n entries, following Section 4 for EXHASH (extendible hashing) and a standard B-tree insertion algorithm for B-TREE. Then we averaged the time costs of 1000 ACCESSes to obtain approximate expected access times, and we averaged the time costs of 1000 INSERTs to obtain approximate expected insert times (for $n + 500$). Sample results with $p = 400$, $b = 10$, $f = 1000$, and $S(x) = 21 \lceil \log_2 x \rceil$ are given in Figures 12 through 14. The page replacement algorithm for the buffer was "Least Recently Used" [13].

6.2 Simpler Analytic Models

The detailed simulation suggests that we can approximately characterize the space requirements of both EXHASH and B-TREE by assuming uniformly filled leaf pages. Let $UT(n)$ be the average page occupancy in entries divided by p. $UT(n)$ will of course be different for EXHASH and B-TREE. Our simulation

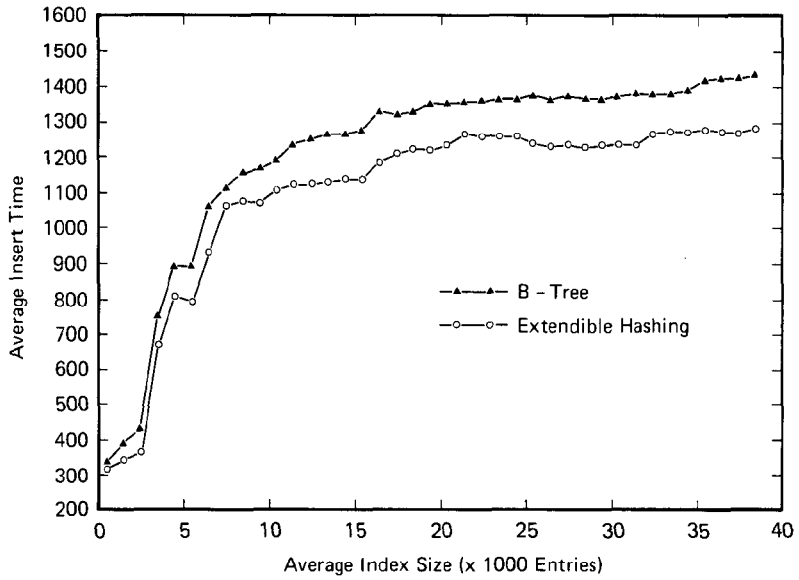

Fig. 13. Insert time. Detailed simulation of insert times averaged over 1000 inserts. The average insert time is plotted against the index size after 500 of the 1000 inserts. The time to fetch a page is 1000. The maximum number of entries on a page is 400.

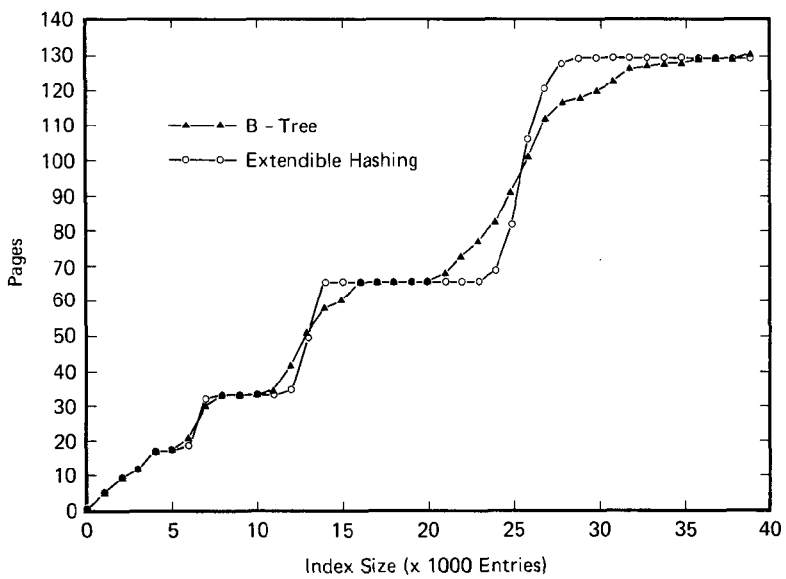

Fig. 14. Space required. Detailed simulation of the number of pages required as a function of the index size. The maximum number of entries on a page is 400.

suggests that for B-TREE $UT(n)$ will quickly reach a steady state, even while the database size n grows. However, EXHASH seems to have a UT function which oscillates with scarcely decreasing amplitude far into the region of very large database size. In any case, we will factor our simplified models into a page

utilization model ($UT(n)$) and a time performance model which assumes each leaf page has exactly $UT(n)p$ entries.

6.2.1 ACCESS TIME. We need to calculate probabilities of page fault, i.e. of a page not residing in the buffer when it is required. We simplify our LRU page management algorithm by assuming that the buffer has one page reserved for each level of the index, and that the remaining buffer slots are occupied by pages drawn at random from the highest (root or directory) level or from succeeding levels if the highest were exhausted before the buffer. This assumption makes the page fault probabilities extremely easy to calculate. First we consider EXHASH.

6.2.1.1 EXHASH. Let l be the number of leaf pages:

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

Then, if d_1 is the number of directory pages,

$$d_1 = \lceil l/d \rceil.$$

Now we can compute the probabilities b_1 (page fault referencing directory page) and b_2 (page fault referencing leaf page):

$$b_1 = \max(0, 1 - (b - 1)/d_1))$$

and

$$b_2 = \max(0, 1 - (\max(1, b - d_1))/l)).$$

Finally we have our approximation for expected access time:

$$\text{ACCESS} = (b_1 + b_2)f + S(UT(n))p.$$

Next we consider B-TREE.

6.2.1.2 B-TREE. Rather than write a general formula for the arbitrarily many levels of a B-tree index, we assume that there are at most four such levels. (The generalization will be obvious.) Since there is only one root page it will always be resident in the buffer. Thus our page fault probabilities will be b_1 (page fault at second level), b_2 (page fault at third level), and b_3 (page fault at fourth or leaf level). We assume that at each level except the root, pages have at most $x = UT(n)p$ entries. Thus in order to have four levels, we must have

$$px^2 \leq n \leq px^3.$$

In this case,

$$\begin{aligned} l &= \lceil n/x \rceil, \\ d_2 &= \lceil l/x \rceil, \\ d_1 &= \lceil d_2/x \rceil, \\ b_1 &= \max(0, 1 - (b - 3/d_1)), \\ b_2 &= \max(0, 1 - (b - d_1 - 2)/d_2), \\ b_3 &= \max(0, 1 - ((b - d_1 - d_2 - 1)/l)), \end{aligned}$$

and

$$\text{ACCESS} = (b_1 + b_2 + b_3)f + S(d_1) + 3S(x).$$

6.2.1.3 Comparison. We expect that $S(x) \ll f$, so that a rough estimate on the extra time required for B-TREE over that required for EXHASH is the difference in page fault times. For an index with more than one page ($n > p$), B-TREE requires at least as many levels as EXHASH. Moreover, we expect the number

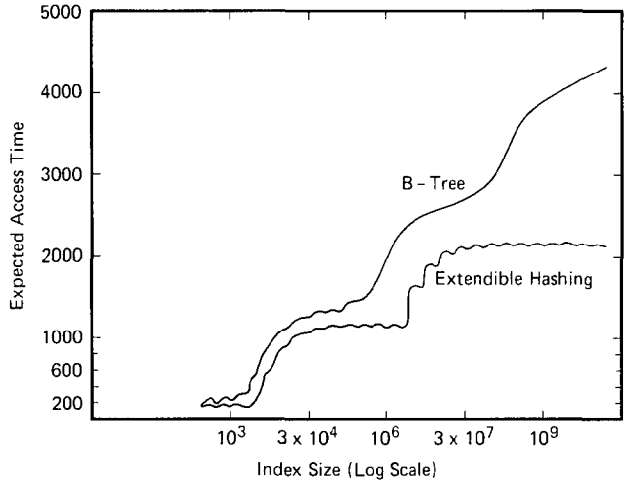

Fig. 15. Access time. Analytic approximation to expected access time for index sizes plotted on a log scale. The time to fetch a page is 1000. The maximum number of entries on a page is 400. The analytic model depends on simulation for the page utilization: the average number of entries per page at a given index size.

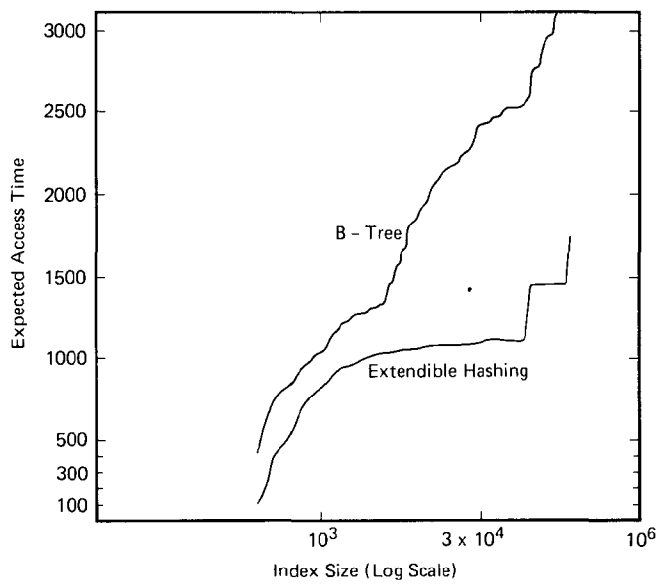

Fig. 16. Access time. Analytic approximation to expected access time for index sizes plotted on a log scale. The time to fetch a page is 1000. The maximum number of entries on a page is 40 as opposed to 400 for Figures 12 through 15. As in Figure 15 the analytic model depends on simulation for page utilization. For index sizes greater than 100,000, this simulation is unstable for extendible hashing. Different simulations give widely varying results, so we have plotted an *upper bound* to our approximation to expected access time based on 30 simulation runs.

of directory pages for EXHASH to be significantly less than the number of second level pages for B-TREE. Thus it should not be surprising that the difference in expected access times between B-TREE and EXHASH grows proportional to the log of the database size. Figure 15 plots our approximations for expected access times out to n = one billion entries, given the parameters of our earlier simulation. Here we have assumed a steady $UT(n) = 0.69$ for B-tree (see Section 5).

Simulation of space utilization for EXHASH indicates that the function $f(n) = UT(2^n)$ is roughly periodic (as predicted in Section 5), with period 1, for $n < 9$. For $n \geq 9$, we expect $0.53 < f(n) < 0.94$ (cf. Section 5). The small waves in the access time graphs for EXHASH correspond to this periodic behavior. In particular, for integers n, $f(n) = 0.64$, and for n equal to an integer plus 0.17, $f(n) = 0.72$. We have also simulated space utilization for the case of 40 keys per page (Figure 16).

REFERENCES

1. 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.
2. ANDERSON, T.W., AND SAMUELS, S.M. Some inequalities among binomial and Poisson probabilities. Proc. 5th Berkeley Symp. Math. Statist. and Probability, 1965, pp. 1–12.
3. BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Informatica 1* (1972), 173–189.
4. CARTER, J.L., AND WEGMAN, M. Universal classes of hash functions. Res. Rep. RC 6687, IBM T.J. Watson Res. Ctr., Yorktown Heights, N.Y., 1977. To appear in *J. Comptr. Syst. Sci.*
5. FREDKIN, E. Trie memory. *Comm. ACM 3*, 9 (Sept. 1960), 490–499.
6. OS/VS2 ISAM Logic, IBM SY26-3833.
7. KHINCHINE, A.Y. *Mathematical Methods in the Theory of Queueing*. Griffin, London, 1969.
8. KNOTT, G.D. Expandable open addressing hash table storage and retrieval. Proc. ACM SIGFIDET Workshop on Data Description, Access, and Control, 1971, pp. 186–206.
9. KNUTH, D.E. *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., 1973.
10. LARSON, P. Dynamic hashing. *BIT 18* (1978), 184–201.
11. LITWIN, W. Virtual hashing: A dynamically changing hashing. Proc. Very Large Data Bases Conf., Berlin, 1978, pp. 517–523.
12. MARKOWSKY, G., CARTER, J.L., AND WEGMAN, M. Analysis of a universal class of hash functions. *Lecture Notes in Computer Science 64*, 1978, pp. 345–354.
13. MATTSON, R., GECSEI, J., SLUTZ, D., AND TRAIGER, I. Evaluation techniques for storage hierarchies. *IBM Syst. J. 9*, 2 (1970), 78–117.
14. MUNTZ, R, AND UZGALIS, R. Proc. Princeton Conf. on Inform. Sci. and Syst., 1970, pp. 345–349.
15. 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.
16. NIEVERGELT, J., AND REINGOLD, E.M. Binary search of trees of bounded balance. *SIAM J. Comptng. 2* (1973), 33–43.
17. REINGOLD, E.M., NIEVERGELT J., AND DEO, N. *Combinatorial Algorithms*. Prentice-Hall, Englewood Cliffs, N.J., 1977, Ch. 6.
18. WALKER, W.A. Hybrid trees as a data structure. Ph.D. Diss., Dept. Comptr. Sci., U. of Toronto, Toronto, Ont., Canada, 1975.
19. YAO, A. C.-C. Random 3-2 trees. *Acta Informatica 9* (1978), 159–170.

Received October 1978; revised January 1979