Dynamic Hashing Schemes

R. J. ENBODY

Department of Computer Science, Michigan State University, East Lansing, Michigan 48823

H. C. DU

Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455

A new type of dynamic file access called *dynamic hashing* has recently emerged. It promises the flexibility of handling dynamic files while preserving the fast access times expected from hashing. Such a fast, dynamic file access scheme is needed to support modern database systems. This paper surveys dynamic hashing schemes and examines their critical design issues.

Categories and Subject Descriptors: E.2 [**Data**]: Data Storage Representation—*hash-table representations*; H.2.2 [**Database Management**]: Physical Design—*access methods*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Dynamic hashing

INTRODUCTION

Users typically interact with modern database systems via a set-oriented, high-level language. This database language can be in the form of a self-contained query language, or it can be embedded in a host language. To retrieve the appropriate data, the database management system must translate queries into the low-level operations provided by the operating system. To have a good response time, it is critical that the access system be efficient and support the high-level database manipulation language well. In addition, the system must work efficiently in an environment in which the volume of data may grow and shrink considerably as a result of insertions and deletions. Most of the studies on such access methods only consider single-key retrieval operations whereby one record is retrieved at a time. More sophisticated access methods that support multikey retrieval and set-oriented associative language facilities are clearly more desirable but are beyond the scope of this paper.

The basic purpose of access methods is to retrieve and update data efficiently. Access methods fall into three general categories: sequential schemes, with $O(n)$ retrieval time, where n is the number of elements in the database; tree-structured schemes, with $O(\log n)$ retrieval time; and hashing schemes, with $O(1)$ retrieval time. If we are trying to minimize access time, hashing schemes are very attractive. The traditional, static, hashing schemes, however, did not perform well in a dynamic environment. They required that data storage space be allocated statically. This meant that when a file exceeded the allocated space, the entire file had to be restructured at great expense. In addition, overflow handling schemes often increased the retrieval time as files approached their space limits. To eliminate these problems, dynamic hashing structures have been proposed. These file structures and their

CONTENTS

associated algorithms adapt themselves to changes in the size of the file, so expensive periodic database reorganization is avoided. In this paper we survey these dynamic hashing schemes, with special emphases on the various design issues.

The goals of this paper are to provide a basic understanding of dynamic hashing, to outline some of the techniques that have been developed, and to show how the various design parameters relate to performance. The techniques can be combined in various ways to develop new variations of dynamic hashing for specific performance goals. We begin with a short discussion of tries, because they provide a means for understanding dynamic hashing, as well as a good foundation for the performance analysis that has been done on the dynamic schemes. Next, we describe two basic dynamic hashing schemes, directory and directoryless, so that the various techniques can be discussed in a concrete context. The techniques address three major issues: space utilization, smooth expansion, and physical implementation. There is overlap in the handling of these issues, but we separate them for ease of discussion. Some of the techniques mentioned in this survey are adaptations of techniques from static hashing; other techniques may also be applicable, but if they have not been studied in the context of dynamic hashing, we have not included them.

1. WHY *DYNAMIC* HASHING?

In order to understand the need for dynamic hashing, we present a quick look at static hashing and point out some of its major weaknesses. Hashing is one of many addressing techniques used to find a record, given its key. A *record*, for example, could be the personal data of an employee with the name of the employee as the *key*. An example of a hash function is a mapping of the key (name) to the address of the employee's record. The function can be as simple as converting the first five characters of the last name to their ASCII equivalents and using their concatenation as the address of the record. In practice, however, these functions are often much more complex.

Consider an example using a simple function of a single character of the key as the address. Figure 1 shows a hashing scheme that maps a key with its ηth character in the range *a–f* into the first slot, *g–m* into the second, *n–t* into the third, and *u–z* into the last. If we use the first character as the key ($\eta = 1$), keys *beth, julie, sue,* and *wally* will map into each of the four slots as shown in Figure 1b. If the keys should be *dot, ed, beth,* and *duffy,* however, all would map into the first slot. Mapping to the same address is called a *collision.* If there is not room in the first slot for four records, these collisions cause an *overflow,* which must be handled. The function could be modified to consider the last letter of a key that would evenly distribute these keys over the four slots, as shown in Figure 1c. This solution works if these are the only four keys used. What happens, however, if we attempt to map a fifth record? If each slot can hold only one record and each is full, there will be a collision and an overflow. One solution is to handle overflows by keeping a list of records at each slot, but searching such a list takes time. In a dynamic environment, the length of that list can increase until

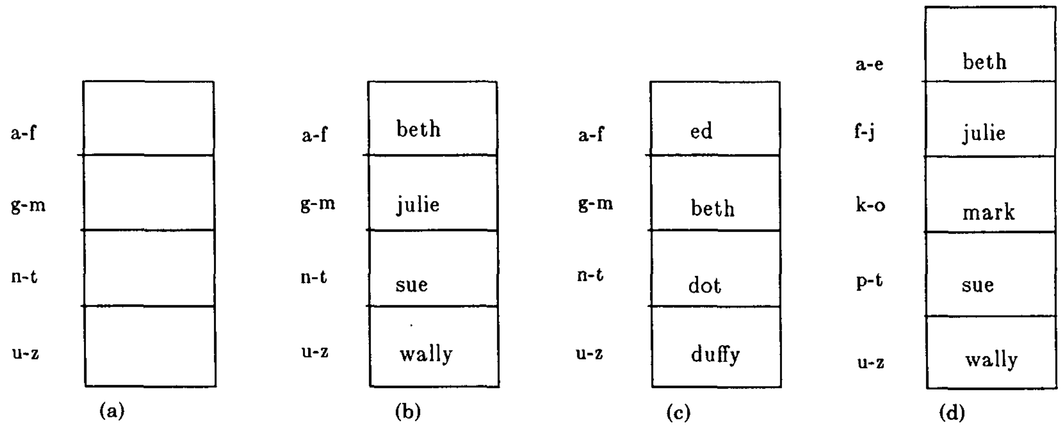


Figure 1. Static hashing example.

access time exceeds an acceptable limit. Another solution in static hashing is to develop a new function and remap all the records. In this case we could create a new function that maps into five ranges: *a–e*, *f–j*, *k–o*, *p–t*, and *u–z*. If the function used the first character and the new key *mark* was added to the original keys (*beth*, *julie*, *sue*, and *wally*), the records would be distributed evenly, as shown in Figure 1d. If the new key was *bob*, *harry*, *sharon*, or *wendy*, however, we would still have a collision and overflow. A different function would be necessary to handle those keys without overflow; a function to handle all possible name keys would be difficult, if not impossible, to determine.

The preceding example shows that if we know the keys in advance, we can often design a static hash function to satisfy our needs. However, in a dynamic environment in which the keys, particularly the number of keys, would be unknown, a file of records might need to be restructured many times in order to accommodate the variety of keys. The goal of dynamic hashing is to design a function and file structure that can adapt in response to large, unpredictable changes in the number and distribution of keys. Tree structures can adapt to such changes, but they tend to have longer retrieval times than hashing schemes. Dynamic hashing attempts to be adaptable while maintaining fast retrieval times. In addition, as a file of records changes in size, it should no longer be necessary to remap all the records; that is, restructuring, which can take a long time, should be kept to a minimum. One way to save restructuring time is to restructure in an incremental fashion on the parts of the file.

2. DEFINITIONS AND TERMINOLOGY

To make our presentation of dynamic hashing more precise, we define some terms. Implicit in these definitions is the assumption that the volume of data is large and therefore stored in random-access secondary storage devices such as disks.

(1) In this paper we only consider single-key retrieval in which a *record* is represented by a *key* $k \in K$, where K is the key domain.
(2) A *file* $F \subseteq K$ is a collection of records.
(3) A *bucket* (or *page*) corresponds to one or several physical sectors of a secondary storage device such as a disk. Let the capacity of a bucket be b records.
(4) Since the time for disk accesses is considerably longer than the time for main memory access or internal machine instructions, the time taken to respond to a query is simply measured in terms of the number of distinct disk accesses issued. It is assumed that each disk access will fetch a page (bucket) of data.

(5) *Space utilization* is the ratio between n and $m \cdot b$, where n is the number of records in the file, m is the number of pages used, and b is the capacity of the page.

3. STATIC HASHING

We begin with a short look at the traditional hashing schemes [Severance and Duhne 1976]. As mentioned earlier, hashing takes a record's key and maps it, using a hash function, into an addressable space of fixed size (Figure 2). Hash functions are chosen so that the records will be uniformly distributed throughout the table for storage efficiency. Carter and Wegman [1979] have shown that classes of such functions to choose from exist. It is not necessary for the original key distribution to be uniform, nor is it necessary for the original order (if any) to be preserved. The latter is a desirable trait, but uniformity and order are difficult to achieve at the same time. Records are accessed by applying the hash function to the key and using the function value as an address into the address space. If more than one record maps to the same bucket, a *collision* occurs; if more than b records map to the same bucket, an *overflow* occurs (remember that b is the capacity of a bucket).

A good hash function will randomize the assignment of keys over the entire address space. Numerous functions have been proposed, and a good comparison can be found in a study by Lum et al. [1971]. The functions include division, folding, midsquare, character and bit analysis, radix transformations, and algebraic coding. As one might expect, if given a fixed set of keys, some specific transformation will work best. However, when we are presented with an arbitrary set of keys, division performs well, and the divisor need not be prime.

In general, we can expect collisions and overflows, and these overflowing records need to be stored elsewhere. Often a separate area is reserved for overflows. A number of basic alternatives have been proposed:

(1) *Linear probing.* An overflow record is stored in the first successive bucket in which an open slot is found. This technique is effective because of its simplicity.

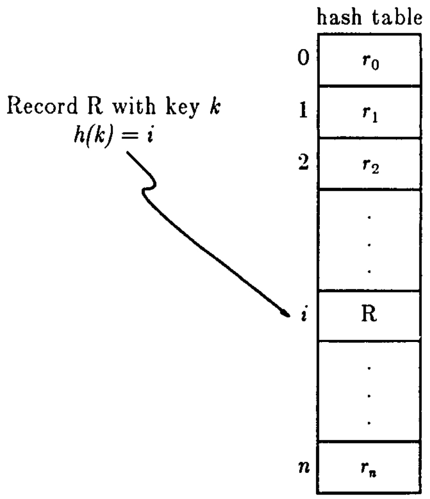


Figure 2. A hashing scheme.

(2) *Coalesced chaining.* When all slots in a primary bucket (and all currently chained overflow buckets) are full, store a new record in a free slot in any unfilled (primary or overflow) bucket and link this new bucket to the current chain of the primary bucket. Overflow from a primary bucket A, when stored in another primary bucket B, may subsequently precipitate the premature overflow of B and result in the coalescing of synonym chains. Counter to one's intuition, however, this method can provide better utilization of storage space than linear probing [Severance and Duhne 1976]. This improvement is due to the fact that, as the storage space fills, the performance of linear probing drops off much faster than coalesced chaining.

(3) *Separate chaining.* The coalesced chaining algorithm is modified to preclude the merging of synonym chains. For example, coalescence can be avoided either by storing overflow records in separate and independent overflow buckets or by relocating an overflow record whenever its storage slot is subsequently required as a primary address. With this technique, one gains slight retrieval advantage over coalesced chaining at the expense of algorithm complexity and storage overhead.

(4) *Rehashing.* Multiple pseudorandom probes generated by a sequence of transformations are used either to locate a free slot for an overflow record or to find that record for subsequent retrieval. Because this method generates random probes, it can have the greatest access time if records are stored on a secondary storage device such as a disk. The nonlocality of the random accesses can result in large rotational delays.

In general, the problem is that as more overflows occur, more records are not where the hash function indicates they should be. Searching for those records can increase the access time beyond the ideal $O(1)$, possibly to the undesirable $O(n)$, if an open addressing method is employed, where n is the total number of buckets used.

The traditional static hashing scheme was restricted by the fact that the designer had to fix the size of the address space at file creation time; that is, the number of buckets and the capacity of each bucket were parameters. If too large an address space was chosen, space would be wasted. If the designer guessed too small, the number of overflows and the access time would increase. The only solution was an expensive reorganization of the whole file. The reorganization process first allocated a larger address space, then rehashed all the records (using a different hash function) to the larger address space, and finally released the old address space. Typically, this reorganization was a time-consuming process.

With static hashing schemes, there is a direct trade-off between space utilization and access time in a dynamic environment. The smaller the space allocation, the more likely are overflows and poorer performance. The dynamic hashing schemes attempt to reduce this space versus access-time trade-off and avoid the expensive reorganization. As we discuss dynamic hashing, we shall concentrate on the issue of space utilization and access time. In general, these schemes have good performance ($O(1)$ disk accesses) with space utilizations that range from 60 to almost 100%. However, we shall still see a trade-off between utilization and access time.

Before beginning our discussion of dynamic hashing, we introduce tries. This data structure is important because it forms a basis for dynamic hashing. For example, one of the original dynamic hashing schemes [Fagin et al. 1979] was presented as a collapsed trie. *Dynamic hashing* is a combination of hashing techniques with trie structure. The tries provide a means of analyzing the performance of dynamic hashing and the intuition for comparing dynamic hashing with tree file structures.

The discussion of tries is followed by a presentation of two basic dynamic hashing schemes, which have been chosen to give us a basis for explaining the variety of schemes that have been proposed. We then address the issues that apply to single-key schemes in this section. We discuss the way the schemes handle space utilization, expansion, and physical implementation. It should be kept in mind that many of these techniques can be combined in various ways to adapt a specific implementation to particular needs.

4. TRIES AS A BASIS FOR DYNAMIC HASHING

A *trie* is a tree in which branching is determined not by the entire key value but by only a portion of it. In addition, branching is based on consideration of that key alone, not on the comparison of a search key with a key stored inside the node. The keys can be from any character or number set, but we limit our discussion to keys that are binary numbers for the purpose of introducing dynamic hashing. Any other key representation can be easily converted to binary.

Let the key under consideration be a binary number with n bits. The bits may be chosen in any order, but for simplicity we will let the branching at level i be determined by the ith most significant bit. As an example consider a key with three bits. Figure 3a shows a trie that addresses three pages A, B, and C. Records beginning with the bit sequence $00 \cdots$ are placed in page A, those with bit sequence $01 \cdots$ are placed

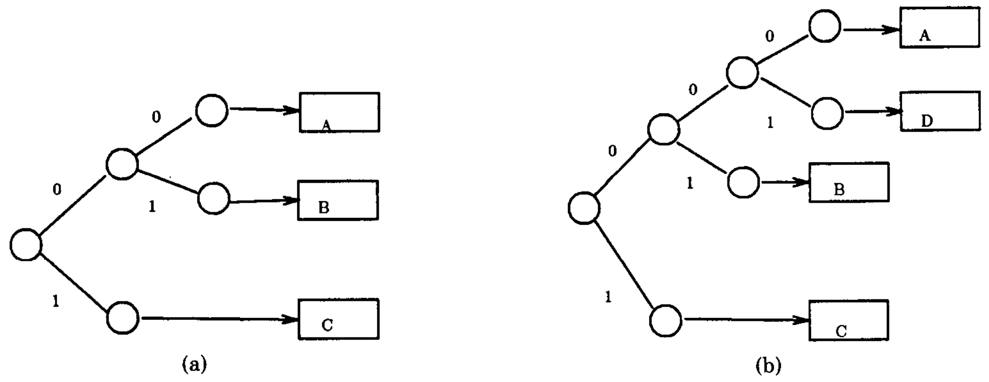


Figure 3. Tries.

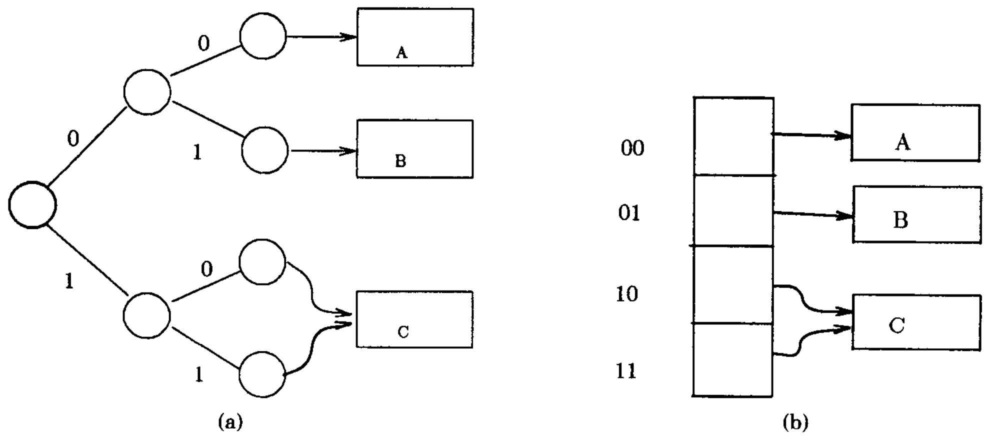
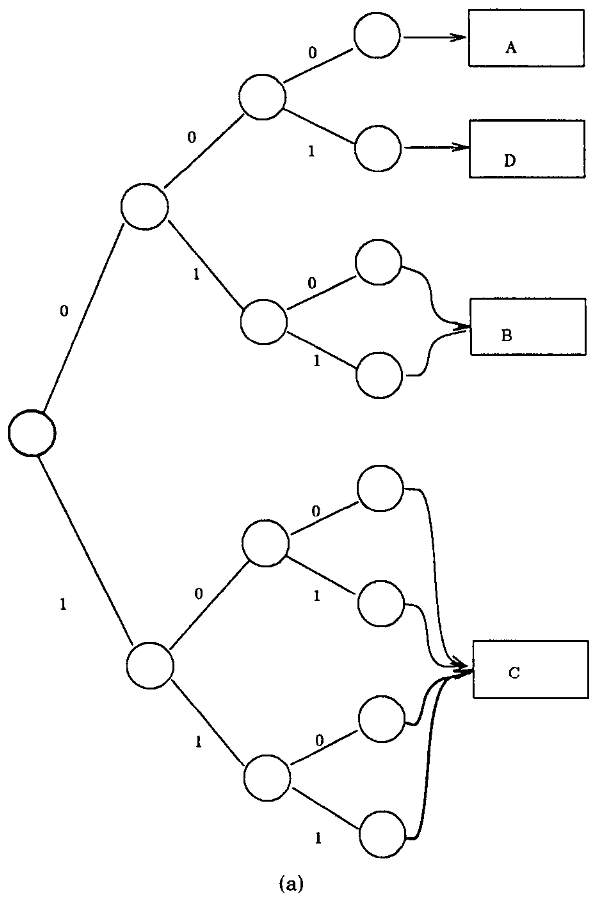


Figure 4. Collapsing a trie to a directory.

in page *B*, and those with bit sequence 1 ··· are placed in page *C*. If page *A* overflows, it can be split into two pages, as shown in Figure 3b. An additional bit is needed to divide the records that were all in *A* between *A* and the new page *D*. Now page *A* contains records beginning with bit sequence 000 ··· , and those with sequence 001 ··· will be mapped to page *D*.

The trie shown in Figure 3 is very flexible and can grow and shrink as the number of records grows and shrinks. The trie, however, must be searched in order to find a desired record. If the trie is skewed owing to a skewed key distribution, that can lead to long access times. The trie could be balanced by choosing a different order of bits, but creating a minimum depth trie is NP-complete [Comer and Sethi 1977]. Dynamic hashing attempts to address both of these problems. To begin with, the search is shortened by collapsing the trie into a directory. This directory can then be addressed by a hash function. If the chosen function is uniform, the implied, collapsed trie will be balanced, which, as we shall see, will tend to keep the directory small.

Collapsing a trie to a directory is illustrated in Figure 4. The depth of the trie in Figure 3a was 2 because 2 bits were needed to decide how to distribute the records among the pages. The second bit was not needed for page *C* in the trie, but it is shown to illustrate how the complete directory is



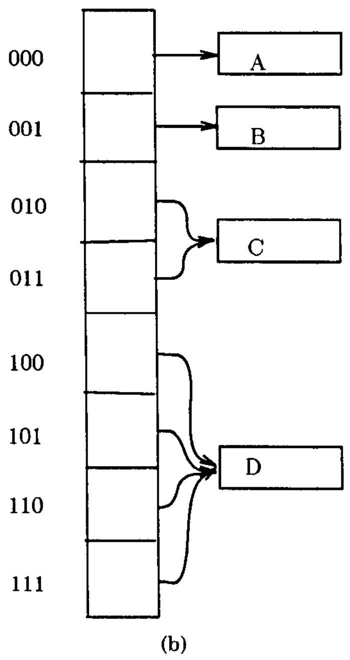


Figure 5. Doubling the directory.

formed. Once a complete trie has been formed as in Figure 4a, the bits on the search path can become indexes to the directory. The trie in Figure 4a can then be collapsed into the directory shown in Figure 4b. The first 2 bits of the keys are now an index into the directory, so there is no tree to be searched. There are, however, redundant directory entries since two directory entries are used for page *C*. This problem of redundant directory entries is exacerbated by a skewed trie. If a uniform key distribution can be found, the trie will be balanced, which will minimize the redundancy. We see later that a hash function can be used to generate such keys, which in turn can be used as indexes to the directory.

Now consider the situation in which page *A* overflows and an additional page *D* is added. This solution is similar to what was done in Figure 3. The trie from Figure 3b can be filled in, as in Figure 5a, and then collapsed into the directory shown in Figure 5b. Note that, since an extra bit was required to differentiate between pages *A* and *D*, 3 bits and hence three levels were needed. These extra bits result in a directory with eight entries (2^3). Of those, many are redundant. Note that page *C* has four directory entries.

There is a problem with having a directory: It adds a potentially undesirable level of indirection that can increase access time. A different approach to collapsing the trie, shown in Figure 6, eliminates the directory.

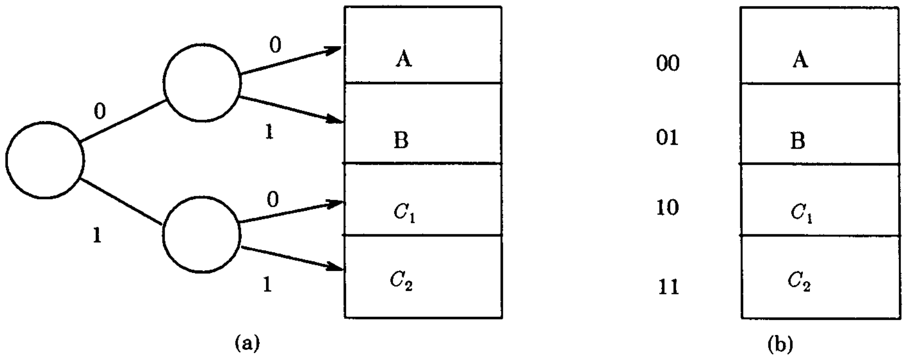


Figure 6. Collapsing without a directory.

The directory can be avoided by maintaining the pages in a contiguous address space. Eliminating the directory, however, also eliminates the indirection allowed by the pointers in the directory. Again, consider page C in the directory scheme in Figure 4. The two pointers to page C were reproduced in the directory from the lowest level of the trie. With the directory removed, the search path in the trie forms the address of the page rather than the index to the directory. This addressing method requires that there be as many pages in the contiguous address space as there were entries in the directory. The result is that addresses 10 and 11, which through the indirection of the directory both pointed to page C, now must point to two different pages. Thus, the contents of page C must be split across two pages, as shown in Figure 6. Now when the trie is collapsed, no directory is necessary. The 2 bits that previously indexed the directory can now be used to address the pages themselves, as shown in Figure 6b.

As with the directory scheme, there must be a way to expand (and contract) the file dynamically. However, when a page overflows in the directoryless scheme, the file expands one page at a time rather than doubling. We defer explanation of this expansion until the addressing technique has been properly defined.

5. BASIC DYNAMIC HASHING SCHEMES

In this section we present a general description of a dynamic hash function and two basic dynamic file systems. The dynamic hashing systems can be categorized as either directory schemes or directoryless schemes that correspond directly to the trie models presented in the previous section. Classifying the systems will give us a basis for describing the important issues for dynamic schemes.

In order to have a hashed file system that dynamically expands and contracts, we need a hash function whose range of values can change dynamically. Most of the dynamic schemes use a function that generates more key bits as the file expands and fewer key bits as the file contracts. One such hash function can be constructed using a series of functions $h_i(k)$, $i = 0, 1, \ldots,$ which map the key space K into the integers $\{0, 1, \ldots, 2^i - 1\}$ such that for any $k \in K$, either

$$h_i(k) = h_{i-1}(k)$$

or

$$h_i(k) = h_{i-1}(k) + 2^{i-1}.$$

There are many ways to obtain the functions $h_i(k)$. We might typically use a function $H(k)$, which maps the key space into random bit patterns of length m, for m sufficiently large. Then $h_i(k)$ may be defined as the integers formed by the last i bits of $H(k)$. Thus, if

$$H(k) = b_{m-1} \cdots b_2 b_1 b_0,$$

where each b_i is either 0 or 1, then

$$h_i(k) = b_{i-1} \cdots b_1 b_0.$$

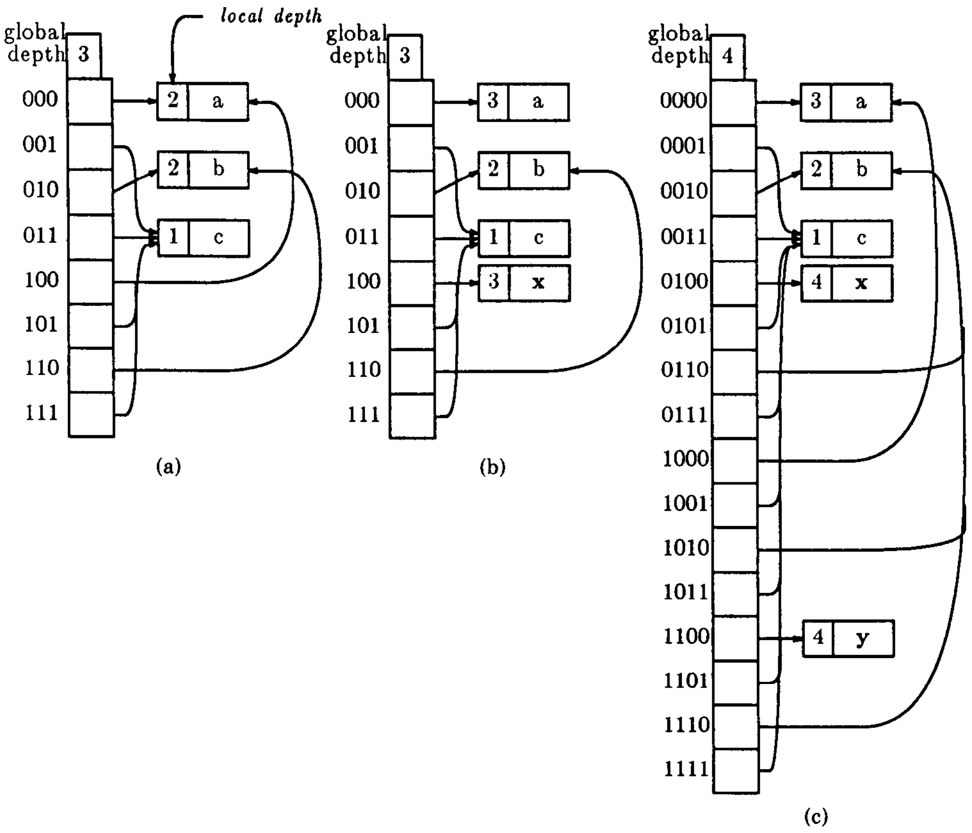


Figure 7. Extendible hashing scheme. (a) Before split. (b) Split *a* into *a* and *x*. (c) Split *x* into *x* and *y*.

Note that h_i defined using the *last* i bits is a different h_i than was implied in Figures 3–6. We used the prefix rather than suffix because the intuition of dynamic hashing based on tries is easier to explain using the *first* bits of $H(k)$. However, the details of the schemes are easier to understand if we define h_i using the last bits. Therefore, we will use the suffix for the remainder of this survey.

In the dynamic hashing schemes, the file goes through a series of expansions. We can begin with expansion 0, a file with two pages addressed by 1 bit (b_0). Doubling the size of the file (to expansion 1) will create a file with four pages addressed by bits b_1 and b_0. At the start of the dth (full) expansion, the file contains $s = 2^d$ directory entries or, for a directoryless scheme, primary data pages (i.e., not counting overflow pages). Consider a directory scheme first. At this dth expansion stage, the record with key k will be found in the primary page whose address is stored in the directory at index $h_d(k)$. With a directoryless scheme, $h_d(k)$ will be the address of the primary page itself. If a primary page overflows, we may have to look further to find the record.

The first scheme we consider has a directory (hash table) with 2^d entries (*extendible hashing* [Fagin et al. 1979]), as shown in Figure 7. As mentioned above, $h_d(k)$ will be a d-bit number that maps to a directory entry that points to the primary data page that contains the record with key k. An important characteristic of this directory is that several directory entries may point to the same primary data page. If exactly two entries point to the same page, all records in that page will have the same value in bits $b_{d-2} \cdots b_1 b_0$; that is, they differ only in bit b_{d-1}; for example, in Figure 7a, all records in page a will have 00 as their last 2 bits. If four entries point to the same page, the records will have the same value in bits $b_{d-3} \cdots b_1 b_0$, and so on; for example, in Figure 7a, all records in page c will have a 1 as the last bit.

To simplify this introduction, we will not allow overflow pages. As a result, if a page overflows, a new primary data page must be added. This page can be added in one of two ways. Assume that page a, which has two pointers pointing to it, overflows as shown in Figure 7a. A new page x will be added so that each of those two pointers will point to their own page. The records that were in page a will now be split between the two pages a and x according to their $h_d(k)$ values. In effect, they will be split using bit d_{d-1}, as shown in Figure 7b. Remember that h_d is defined as the last bits of H rather than the first, as was indicated in earlier figures using tries. After the split, all records in page a have 000 as the last 3 bits of the hash function $H(k)$. Similarly, all records in page x have 100 as their last 3 bits. As a result of the split, there is an implicit buddy system based on the bits of the hash values. The records in this pair of pages have 00 as the last bits of the function $H(k)$ and differ in the third from last bit. We say the overflowing page *split* into two pages. In a similar way those two buddy pages can be combined into one page if the file were to shrink.

Consider the case in which page x overflows. This page has only one directory entry pointing to it, so it cannot split to form two buddy pages. An option at this point would be to use one of the overflow techniques from static hashing. For example, we could use separate chaining and add an overflow page with a pointer to it in page x. These links, however, can increase retrieval time. The solution taken here is to increase the number of directory entries so there will be at most one page for every directory entry. To do that, the whole directory must be split, causing the directory to double in size, as shown in Figure 7c. This doubling can be accomplished by using the function $h_{d+1}(k)$ in place of $h_d(k)$; that is, use $h_{d+1}(k)$, which maps one more bit, doubling the number of directory entries. If this extra bit differentiates between the records in page x, those records can be split between two pages, x and y. This extra bit in the hashing function may not be enough to divide the records into *both* pages. We may still hash all the records into one page, which will still overflow. The only solution is to split again with the hope that the additional bit will be sufficient to differentiate among the pages. The probability of this situation occurring is smaller with uniform distributions.

One problem with splitting a node is knowing whether we need to double the directory when we make a split. This decision can be aided with the addition of another field in each data page and one additional entry for the whole directory. The directory must keep track of the "global depth" d, that is, the number of bits needed for addressing. The "local depth" d_l stored within each data page is the number of bits used in addressing that page. There are 2^{d-d_l} pointers to each data page. When we split page a in Figure 7a, the local depth was 2 and the global depth was 3. Hence, only 2 of the 3 bits being used in the directory were significant for accessing this page. We can split page a and create two pages with local depth 3 without doubling the directory. If, however, the global and local depths are the same, then all the directory bits are used to access that page. This was the case when page x needed to be split, in Figure 7b. If we split such a page, the directory must double to get an extra bit to distinguish among the records within the page. Notice in Figure 7c that when we double the directory only the local depths within the split pages are changed. The other pages can be distinguished using the same number of bits as before, so their local depths remain the same. Another problem emerges here from this information.

It is through doubling the size of the directory that the file can expand dynamically. In a similar way the file can contract. If pairs of pages with single pointers can map to and fit in single pages, then the file can shrink by combining those pages and contracting the directory to half its size. A page can be combined with the "buddy" page that was formed when a page split. After pages are combined, the directory can be contracted when the local depth of all pages is less than the global depth. As before, the other data pages will not be affected; there will merely be a decrease in the number of pointers to their data pages. The hash function $h_d(k)$ will become $h_{d-1}(k)$, so the index will be referenced with one less bit. The directory in Figure 7a, for example, could be contracted since only 2 bits are required to differentiate records in any page.

The other basic scheme is a *directoryless* scheme. We will describe the logical mapping assuming that the file will be stored in contiguous logical address space, as shown in Figure 8. In Figure 8a the file is at the beginning of the second expansion ($d = 2$), so the file contains $s = 2^d = 4$ primary data pages. In this scheme the directory has been eliminated, so $h_d(k)$ will be the address of the desired primary data page.

The expansion of this scheme is similar to the previous one in that overflows can cause pages to split. It differs, however, in that the overflowing page need not be the one that gets split. Instead, pages are split in sequential order with no regard for whether the page that is split is the one that actually overflowed. In this directoryless scheme, the hash function directly addresses the pages, so there must be a *page* for each function value, whereas in the directory scheme there was a *directory entry* for each function value. The pointers in the directory scheme provide a level of "indirection" in the addressing that, for example, allows fewer pages than directory entries. If we perform an expansion in a directoryless scheme using h_{d+1}, we will double the number of pages in the same way in which a directory doubled in the previous scheme. However, doubling the directory in the previous scheme did not double the number of pages. Doubling the number of pages, as opposed to doubling the number of directory entries, does not provide the desired smooth growth of the file. The solution is to not double the file but to add pages one at a time in sequential order. We will give an example to illustrate this solution.

Figure 8 shows an expansion of a directoryless scheme. We begin with four pages shown in Figure 8a, that is, $d = 2$. The 2 bits needed to address those four pages are shown in the figure. In this example the file will expand whenever any page overflows. Let page b be the first page to overflow. The file will expand by adding page A and splitting the records of page a into pages a and A. As before, in order to differentiate between records for page a and those for page A, an extra bit is needed. Therefore, $h_2(k)$ is still used to address pages b, c, and d, but $h_3(k)$ is used for pages a and A. This situation is illustrated in Figure 8b. Since page a rather than page b split, an overflow page (w) is needed for page b. After a number of insertions, let page d overflow next (Figure 8b). It is page b's turn to split, so a page B is added to the file and the records from page b and its overflow page w are split between pages b and B. $h_2(k)$ is still used to address pages c and d, but $h_3(k)$ is used for pages a, b, A, and B. Assume more insertions, and let page d overflow again. Page C will now be added, and page c will split as shown in Figure 8c. Finally, let page B overflow causing page D to be added and page d to split as shown in Figure 8d. Now all pages require 3 bits for their addresses; that is, all are using function $h_3(k)$. This point marks the completion of expansion 2 and the beginning of expansion 3. Expansion continues in the same way; contraction is done in an analogous manner.

There are some issues to consider with this scheme. The first is the *splitting policy*. In our example, we split whenever any page overflowed. Splitting could be done after a fixed number of insertions or overflows, or after the pages have reached some average percentage of their total capacity. Another issue is *overflow handling*. As in static hashing, there are a number of ways to handle overflows. Finally, expansion can be smoothed out further. In our example, we split one page into two pages—the original page and a new one. It is possible to arrange the expansion so a page is split across more than two pages. We examine these issues in detail later.

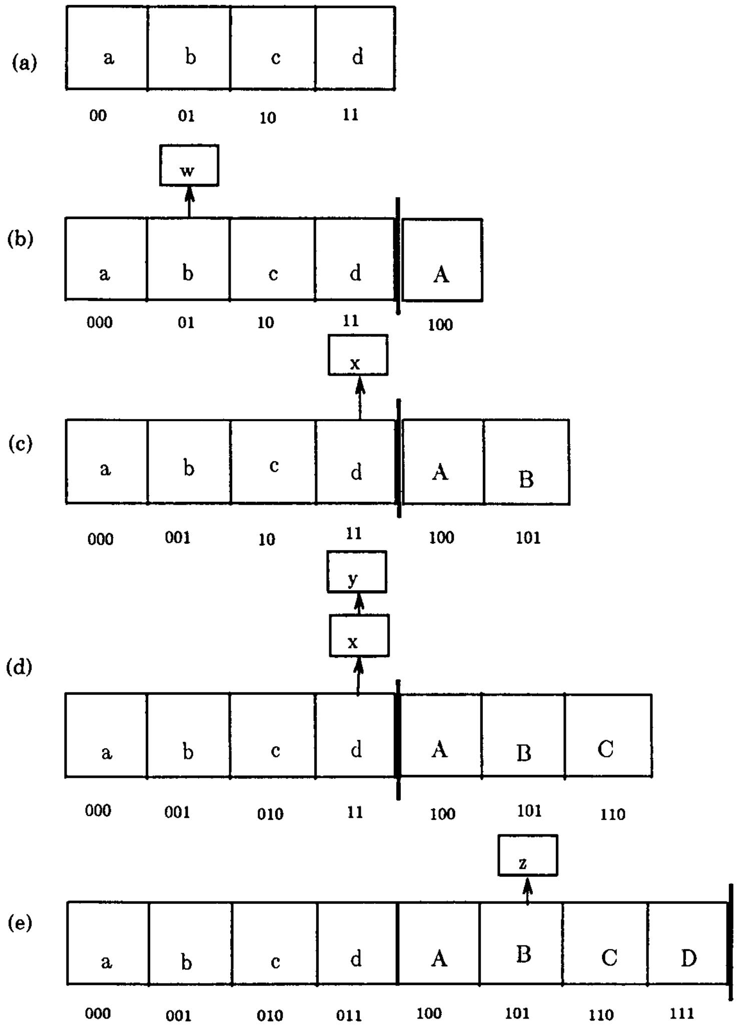


Figure 8. Directoryless example.

Figure 9 shows the notation used for directoryless schemes, also known as *linear hashing*. It shows a file at the beginning of the dth expansion, so it will contain $s = 2^d$ primary data pages and will be addressed by $h_d(k)$. As more records are inserted into the file, new pages will be added to the end of the file. To determine which page will be split into this new page, we introduce an index p. At the start of the dth expansion, the pointer p will point to page 0 of the file. When another primary page is appended to the file, page 0 (pointed to by p) will be split between page 0 and the new page. Since there are $s = 2^d$ pages, the new appended page will be page $s + p$. The page p is split so that the records that previously were in page p will be distributed between pages p and $s + p$ using the hash function $h_{d+1}(k)$. The pointer p will then be incremented by 1. When the file size has doubled (i.e., the split pointer p reaches s), there will be 2^{d+1} pages in the file. We then increment d and return the split pointer to the beginning of the file, and the next expansion begins.

With directory schemes, addressing is trivial. The function $h_d(k)$ indexes the

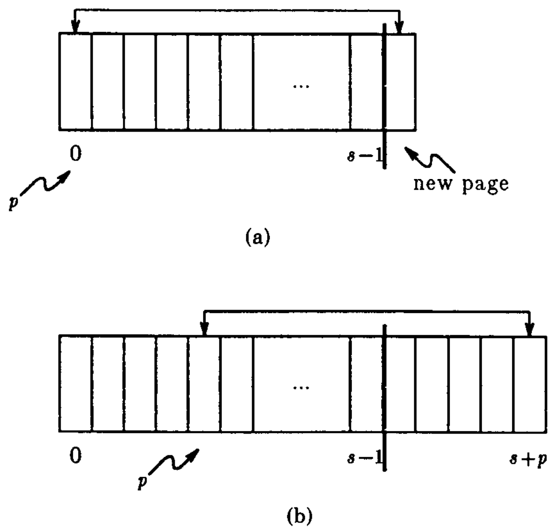


Figure 9. Linear hashing.

directory where the pointer is followed to the desired page. Directoryless schemes, however, use both $h_d(k)$ and $h_{d+1}(k)$ to access the primary data page. The procedure is as follows:

```
begin
  if $h_d(k) \geq p$ then page := $h_d(k)$
    else page := $h_{d+1}(k)$;
  if necessary, chase the overflow chain
end
```

The basic idea is that to the left of the pointer p each page has been split. For each page to the left of p that has been split, there is a corresponding buddy page to the right of page $s - 1$; see Figure 9. In order to decide which of the corresponding pair of pages to access, an extra bit is needed. The function $h_{d+1}(k)$ provides that discriminating bit. For records that hash into the unsplit pages between the pointer p and page $s - 1$, the function $h_d(k)$ is still used. Notice that, since $h_d(k)$ only uses d bits, it cannot reference to the right of $s - 1$, that is, the right edge of the file at expansion d. At the end of the current expansion d, all pages will be addressed by $h_{d+1}(k)$. When we move the pointer p back to point to page 0, we will also increment d, so in the next expansion the function $h_d(k)$ is the same as the previous expansion's $h_{d+1}(k)$.

Now that both basic schemes have been introduced, one of the fundamental differences between them can be pointed out. A directory scheme without overflows can guarantee two accesses for every request, one for the directory and then one for the desired page. A directoryless scheme, on the other hand, can keep many requests to only one access because the directory fetch has been eliminated. With a directoryless scheme, however, overflows are required because the pages are split in some predefined order. The result of having these overflows is that some requests will require two or more accesses. This comparison gets more complicated if one also allows overflows in a directory scheme. The distinction is that directoryless schemes must allow overflows, whereas the directory schemes may allow overflows.

6. SPACE UTILIZATION

In this section we deal with the issue of increasing space utilization in the various dynamic hashing schemes. Given n records, w data pages, and b records per page, we define *utilization* to be $n/(wb)$; that is, utilization is the ratio of records to space reserved for records. A large utilization ratio is desirable because it means that there is little wasted space; that is, each page will be close to full. There is, however, a price to be paid for increasing utilization. Within any scheme, space utilization can be increased by modifying the overflow structure or by adding overflows if they do not

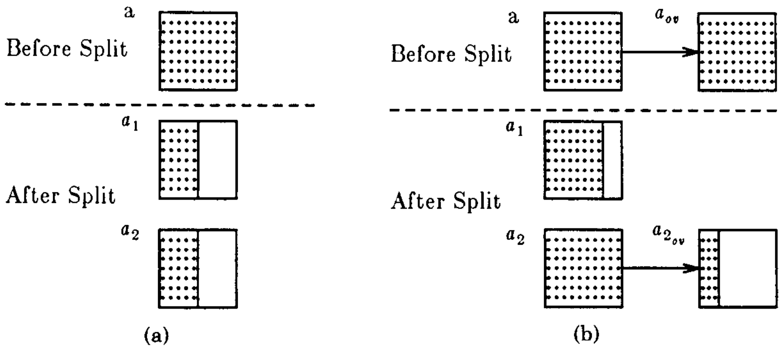


Figure 10. Overflow pages are used to increase space utilization. (a) Split without overflow. (b) Split with overflow.

already exist. Therefore, as utilization goes up for a particular scheme, the access time tends to increase also. First we look at the effect in general of overflow pages on utilization and then at some of the implementations that have appeared in the literature.

If there are no overflow buckets in a dynamic hashing scheme, we can expect the utilization to be approximately 69% [Fagin et al. 1979; Larson 1978; Mendelson 1982]. The intuition behind that figure is as follows. Whenever a full page is split, two roughly half-full pages are created. With b records (one full page) spread over the two new pages (with total capacity $2 \cdot b$), the utilization of the new pages is $b/2b = 50\%$, as shown in Figure 10a. After a number of splittings we expect the utilization over the whole file to be greater than 50% but less than 100%, since pages will tend to be half-full (recently split) or somewhat full. Over time, the utilization of such a scheme will approach $\ln 2 = 0.69$, or 69%, because of the successive splitting in half of the pages.

This space utilization of 69% derives from analysis of tries associated with dynamic hashing and assumes a uniform key distribution [Fagin et al. 1979; Larson 1978; Mendelson 1982]. Let $L(k)$ be the expected number of leaf pages needed in a trie for k records. These leaf pages would correspond to pages in a dynamic hashing scheme. Clearly $L(k) = 1$ for $k \le b$, where b is the maximum number of records in a page. Considering a trie with $k > b$, the number of records in each subtree at the root can be viewed as having a symmetric binomial distribution. From that we can expect that with probability $\binom{k}{j}(\frac{1}{2})^k$, j keys will be in the left subtree and $(k - j)$ keys will be in the right subtree. Therefore, the expected number of leaf pages in each subtree will be $L(j)$ and $L(k - j)$, respectively. It follows that

$$L(k) = \sum_{j=0}^{k} \binom{k}{j}\left(\frac{1}{2}\right)^k [L(j) + L(k - j)].$$

Using this approach, Mendelson [1982] showed that

$$L(k) \approx \frac{k}{b \ln 2}.$$

From the expected number of leaf pages the space utilization can be determined as follows:

$$\text{Utilization} = \frac{k}{b \cdot L(k)} \approx \ln 2 \approx 0.69.$$

The same results were achieved by Fagin et al. [1979] assuming keys with a Bernoulli distribution. Larson [1978] applied similar techniques to linear hashing, but his results are complicated by the fact that the trie for linear hashing may not have all leaves at the same level during an expansion. His results were dependent upon the page size but were very close to the same 69% utilization.

Overflow pages allow the utilization to be increased upon splitting. Let us first

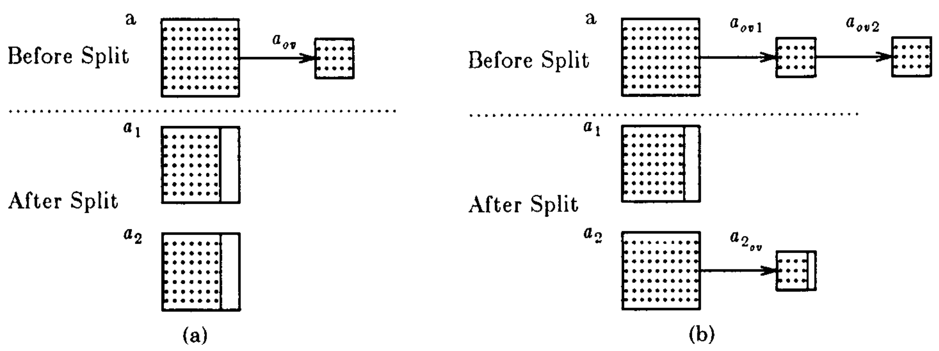


Figure 11. Using overflow pages of smaller size. (a) Split with one small overflow. (b) Split with two small overflows.

assume that the overflow pages are the same size as the primary data pages. If both a primary page and its overflow page are full before splitting, that is, a total of $2b$ records, then after splitting the utilization will be $2b/3b = 66\%$. This situation is illustrated in Figure 10b. There are three pages after the split because we expect one of the pages still to be overflowing since an exact split is not guaranteed. This technique yields an improvement over the 50% per split page without overflows, so we would expect the overall utilization to improve, too. Further improvement can be expected if we have overflow pages that are smaller than the primary page. For example, let the overflow page size be $b/2$. If a primary page and its overflow page are full with a total of $3b/2$ records, then, when split, there is a good chance that no overflow will be needed, as shown in Figure 11a. Those $3b/2$ records, spread over two primary pages with capacity $2b$, yield a utilization of $(3b/2)/2b = 75\%$. If an overflow were needed after splitting, this utilization could be as low as 60%. If there were two overflow pages before splitting, however, then, with those $2b$ records, we would expect an overflow after splitting with a resulting utilization of $2b/(5b/2) = 80\%$, as shown in Figure 11b.

There are many possible overflow page sizes, each with a different effect on utilization. The preceding discussion gives a feeling for that; more details are available in the tables at the end of Larson [1982], which give an indication of the relationship among utilization, primary page size, and secondary page size for a directoryless scheme. Two conclusions are apparent from those tables. One is that for low utilizations (e.g., 80% or less), a wide range of overflow page sizes will work. As the utilization increases, however, the range of good page sizes becomes quite narrow. The other conclusion is that to achieve high utilization, longer overflow chains will be necessary. These chains have the undesirable effect of slowing down access time. However, a side effect for directory schemes is that the use of overflows can keep the size of the directory down. This improvement is especially noticeable if the distribution of records is not uniform.

A point to consider with respect to directory schemes is how large a directory is needed for a given number of keys. Fagin et al. [1979] in their original introduction to extendible hashing estimated that $2 \cdot \lceil \log(k/b \ln 2) \rceil$ directory entries would be needed given n records. They assumed that the number of records that fell within an address range was Poisson distributed and showed that the probability of the local depth differing from the global depth by more than 2 was very small.

A better estimate of the size of the directory was determined by Flajolet [1983]. He assumed a Bernoulli model that intuitively corresponds to the uniform measure over n-tuples of the (infinite) binary sequences that make up the hash keys. He determined

Table 1. Expected Directory Size[a]

	b					
n	5	10	20	50	100	200
10^3	1.50K	0.30K	0.10K	0.00K	0.00K	0.00K
10^4	25.60K	4.80K	1.70K	0.50K	0.20K	0.00K
10^5	424.10K	68.20K	16.80K	4.10K	2.00K	1.00K
10^6	6.90M	1.02M	0.26M	62.50K	16.80K	8.10K
10^7	111.11M	11.64M	2.25M	0.52M	0.26M	0.13M

[a] From Flajolet [1983].

a probability that a trie built on n keys has height less than or equal to h. This probability is equal to the probability for n numbers in [0, 1] such that no more than b fall into any of the intervals determined by directory indexes. This turns out to be a counting problem of permutations with limited repetitions. Although Flajolet's probability formula is too complex to reproduce here in its entirety, we can give his reduced coarse approximation for directory size:

$$\frac{e}{b \log 2} n^{1+1/b} \approx \frac{3.92}{b} n^{1+1/b}.$$

In addition, Flajolet provided a table that displays the expected values of the directory size using extendible hashing for various values of n, the number of records, and b, the page capacity (see Table 1). The entries show the directory size in thousands ($K = 10^3$) and millions ($M = 10^6$).

7. OVERFLOW HANDLING TECHNIQUES

7.1 Splitting Control

When we described the basic schemes in the Introduction, we used a simple splitting control policy. In both the directory and directoryless schemes, a page was split as soon as any page overflowed. In this section we describe other ways of determining when to split a page. We give this description here because splitting control has a direct effect on how much overflow will be tolerated. One can delay splitting by allowing pages to overflow; the longer splitting is delayed, the longer the overflow chains become. In the previous section we saw that overflows can improve space utilization so the splitting policy chosen ultimately affects utilization.

Rather than split a page as soon as it overflows, splitting can be deferred by utilizing an overflow page. This deferral is implicit in the directoryless scheme because pages are split systematically, not necessarily when they themselves overflow. Deferring can also be applied to a directory scheme. In a directory scheme, deferring the splitting of a page is particularly helpful if splitting that page would cause the directory to double. Consider Figure 7. In Figure 7c many directory entries point to the same page, resulting in a waste of directory space. If page x had been allowed to overflow, the split that would have resulted in the doubling of the directory could have been deferred. If a number of pages were allowed to overflow before splitting, the result might be a higher utilization, especially of the directory.

Deferring splitting is closely related to the sharing of overflows. The former will tend to increase the number of accesses required because of the increased use of overflows. This increase in retrieval cost, however, can be partially offset by sharing the overflow pages. Sharing the overflow pages decreases accesses by allowing a given overall space utilization to be reached using fewer overflow pages. With fewer overflow pages, the average number of disk accesses should be smaller. This sharing of overflow space is common in static hashing where overflows are often kept in one overflow area.

Deferred splitting was introduced by Larson [1978] as a means of improving utilization in directory schemes. Scholl [1981] investigated the details of implementing deferred splitting and added sharing of the overflow buckets to improve utilization of the overflow pages. Sharing overflow pages is similar to having overflow pages that are smaller than the primary data pages. A similar result can be achieved without having special overflow pages. Veklerov [1985] logically paired all pages; each page and its pair were referred to as "buddies." When a page overflowed, Veklerov proposed putting the overflow records into the buddy page if the buddy itself was not full. When there were too many overflow records or the buddy needed the space, the original page would split. Deferring the splitting achieved two goals: It increased utilization and by deferring splitting it kept the size of the directory down.

Once it has been decided to defer splitting, a policy of when to split is needed. Unfortunately, in order to get a high utilization more overflow buckets are needed, which can increase the access time. A natural solution is to split when utilization reaches some threshold. For example, a page could be split whenever utilization of the file became greater than 80%. This policy allows coarse control of both the utilization and the length of the overflow chains.

Since overflows are required for directoryless schemes, splitting control policies were a natural part of their development. Litwin [1980] presented two methods of splitting for linear hashing: controlled splitting and uncontrolled splitting. In *uncontrolled splitting* the node pointed to by the "split" pointer p is split when *any* page overflows (Figure 9). Since there is little correlation between the overflowing page and the split page, the utilization of this scheme is low, that is, approximately 60%. *Controlled splitting* is a deferred splitting policy where the node pointed to by the "split" pointer is split only when the space utilization is above some threshold, for example, 80%. Pages are allowed to overflow until the desired space utilization is achieved, and only then is the next page split.

7.2 Recursive Overflow Handling

Once the decision has been made to share an overflow space, a natural question is how should that space be managed. A static overflow space could defeat the flexibility of a dynamic hashing scheme. One possibility is to manage the overflow space as a dynamically hashed file. Since the pages in the overflow space may themselves overflow, this process can be extended in a recursive fashion creating many levels of dynamic files, one overflowing into the next. Each level has the shared overflows from the previous level, so each level should have fewer records than the previous level. Over time the file of overflows at any level can expand to accommodate an expanding number of overflows from the previous level. They can contract in a similar fashion.

An implementation using linear hashing at each level is shown in Figure 12 [Ramamohanarao and Sacks-Davis 1984]. Unlike most overflow schemes, the overflows in *recursive linear hashing* are not explicitly linked from the primary page. When an attempt is made to insert a record into a full primary page, no explicit overflow page is created. Instead, an attempt will be made to insert the record into the next level. Thus, a level has the shared overflows from the previous level, but in a dynamic rather than static way. A file at any level behaves exactly like a normal dynamic file and will expand as more records are inserted (or attempted to be inserted) into that level. Previously, when a page split, it would collect records from its own overflow pages and divide them between itself and the added page (p and $s + p$ in Figure 9). In the course of the split, one or both of the primary pages might still require overflow pages. In recursive linear hashing, a page collects overflows not from its own overflow pages but from the dynamic file in the next level. If either primary page still overflows, those overflow records will be put back into the next level. Although the number of levels can in theory continue indefinitely, it has been claimed that with a uniform distribution more than three levels are rarely required [Ramamohanarao and Sacks-Davis 1984]. However, a nonuniform distribution could create many levels of overflows.

In a normal dynamic scheme, the process of splitting a page includes collecting overflows to put into the newly created pages. The same is true with recursive hashing. When a node at level i is split using some control policy, overflow records are collected from level $i + 1$ to fill the resulting pair of pages. Overflow records are collected from successive lower levels ($i + 2, \ldots$) so that any nonfull page cannot have records that overflow into the next level. Since any

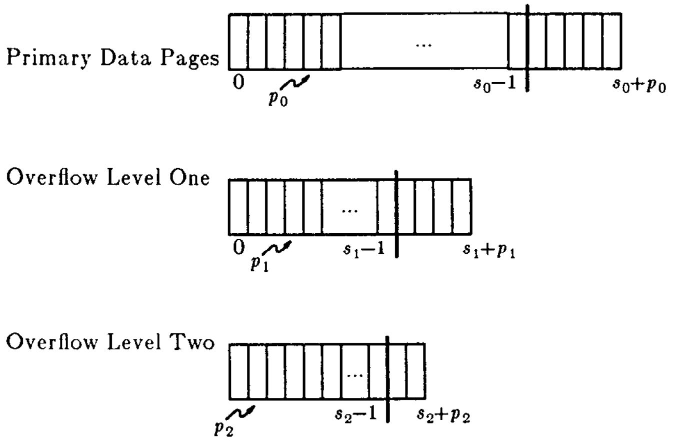


Figure 12. Recursive linear hashing scheme.

lower level is made of overflows from higher levels, we expect the lower level files to be smaller. In a linear hashing implementation by Ramamohanarao and Sacks-Davis [1984], all records stored at level $i + 1$ whose primary page at level i is q will be stored in a single page at level $i + 1$, and the address of this page is determined by q.

Records are addressed in recursive linear hashing using the following procedure. The procedure *search(p, l, k, status)*, which searches page p of level l for a record with key k and returns with status either *found* or *not_found*, is assumed to exist. Assume that the current number of levels in the file is *max_level*:

```
begin
  $l := 0$;   (* initialize level to zero *)
  repeat
    $l := l + 1$;
    if $h_d(k) \geq p$ then page := $h_d(k)$
      else page := $h_{d+1}(k)$;
    search(page, $l$, $k$, status);
  until status = found or $l$ = max_level
end
```

The recursive scheme has been implemented as a linear hashing scheme, but a directory scheme can be implemented in a similar way.

7.3 Multipage Nodes

Lomet [1983] proposed a different way of handling overflows in directory schemes. Rather than have overflow pages as separate entities, he allowed the primary page to expand over a number of contiguous pages. The page then became a "multipage node," as shown in Figure 13. A fixed upper limit was placed on the size of the directory, and once that limit was reached it was the nodes and not the directory that expanded. A record was accessed first by consulting the directory and then by searching the corresponding multipage node.

The multipage node can be managed in a number of ways. The simplest would be to keep the records in the same order in which they were inserted and perform a sequential search. A more complicated approach would be to manage each multipage node as a dynamically hashed file. For example, the number of pages in the node could be kept in the directory, which would allow one to manage the contiguous multipage node as a directoryless hashing scheme. Each multipage node would grow and shrink as a normal directoryless hashed file. One could even allow overflows within the multipage node directoryless file.

A different multipage node proposal was made by Ramakrishna and Larson [1988]. Their composite perfect hashing is a static hashing scheme that can achieve high utilization without overflows. With no overflows, only one disk access is required, thus the name "perfect" hashing. Using this perfect hashing with a directory scheme

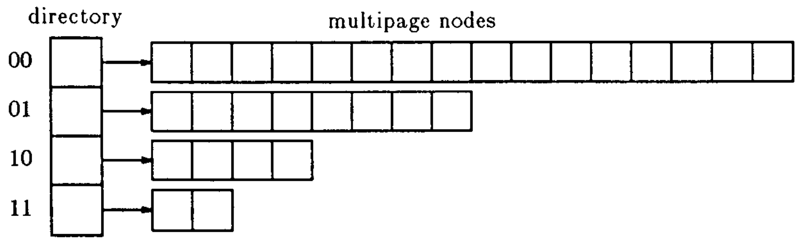


Figure 13. Bounded index exponential hashing.

preserves the two-disk access bound found in extendible hashing. Expansion of composite perfect hashing requires rehashing, but the amortized cost is kept low by keeping the frequency of expansions small.

7.4 Internal Storage of Overflows

Some static hashing techniques have been directly applied within the dynamic hashing framework. For example, the contiguous address space of directoryless dynamic hashing lends itself to open addressing mentioned earlier. Larson [1985] proposed that open addressing be used for storage of overflows in a directoryless scheme. Open addressing searches the file from the overflowing page in a linear fashion to find a nonfull page in which to place the overflowing record. In dynamic hashing the problem is that pages in the unsplit region will tend to need the most overflow pages, and few nonfull pages can be expected until some split pages are reached, for example, to the right of page $s - 1$ in Figure 9. The solution is to order the splitting of pages so that the split and unsplit pages will be interleaved throughout the file. That will tend to put nonfull split pages (with room for overflows) near the full unsplit pages, which will keep the overflows nearby. Access time is kept reasonable by keeping the overflows close, which prevents the overflow search from getting too long.

Chaining can also be used. Mullin [1981] proposed a directoryless scheme that stores all overflows in the empty space of primary data pages that are not full. These overflows are then chained together. Vitter [1982] mentioned that coalesced chaining may be applicable to dynamic hashing. The details of that idea, however, have not been published.

8. EXPANSION TECHNIQUES

Dynamic hashing schemes can have oscillatory performance. If the hash function can distribute the hashed keys uniformly over the pages of the file, the pages fill up uniformly and become completely filled almost simultaneously. Within a small period of further file growth, a majority of file pages overflow, and their entries must be split over pairs of pages. The result is that utilization increases from 50 to almost 100% and then suddenly falls to 50% during the short splitting period. In addition, the cost of doing an insertion is comparatively low at low utilizations but is considerably higher during the splitting period because so many insertions lead to page splitting. Finally, if overflow records are required by the technique, then the frequency of occurrence of overflow increases as utilization approaches 100%, resulting in a sharp increase in the cost of insertions and searches as overflow page accesses increase.

Fortunately, random variations in hashing functions, nonperfect uniformity of the functions, and large files all work to keep those oscillations in check. However, a couple of techniques have been proposed to explicitly smooth out the expansion. First we look at one that deals with a way to smooth out the expansion given a uniform distribution. The second smooths out the process by changing the distribution to an exponential one. We then look at directoryless and directory schemes that have been designed to handle such nonuniform distributions.

8.1 Uniform Key Distribution

In this section we will consider schemes to improve access time. We concentrate on directoryless schemes because directory schemes without overflows are optimal with respect to access time given their structure.

The problem with our basic directoryless scheme (linear hashing) is that the overflow chains get too long near the right end of the unsplit region, that is, near page $s - 1 = 7$ as shown in Figure 14. This problem occurs because these pages are the last to be split, so their overflow chains can become quite long. The recently split pages are underutilized, and the ones that are last to be split get overutilized with long chains. One way to cut down on the length of those chains is not to wait so long before splitting the buckets. Larson [1980] proposed *partial expansion*, which allows the file to go through its doubling in smaller steps. By expanding in smaller steps, each page will get a chance to expand partially long before the whole file is doubled. For example, consider a basic directoryless file that splits after L insertions, and let there be s pages. The last page to be split, that is, page $s - 1$, would be split after sL insertions. Now assume that the file expanded in two stages, with each page partially expanded during each stage. Page $s - 1$ will be partially expanded after only $(sL)/2$ insertions. As a result, we would expect the overflow chains to be shorter and the average access time less.

Larson's linear hashing with partial expansion works by dividing files into groups. We will begin with a look at the concept and consider the details of addressing later. Let us examine one partial expansion for the case of eight pages ($d = 3$, $s = 2^d = 8$) that are paired in four *groups* of size 2. The pages are paired so that page i and page $i + 4$ are in the same group, called group i. When we decide to split a page, we actually partially split the pair of pages in a group. Consider group 0; when we add a new page (page 8), we shall spread the records from both pages in group 0 across the three pages (the original pair in group 0 plus the new page), as shown in Figure 15. If pages 0 and 4 are full (a total of $2b$ records), then the three pages 0, 4, and 8 (with combined capacity $3b$) will be roughly two-thirds full ($2b/3b$). If we were splitting with only one page in each group, that is, our basic directoryless scheme, then the final utilization would only be one-half for the two resulting pages.

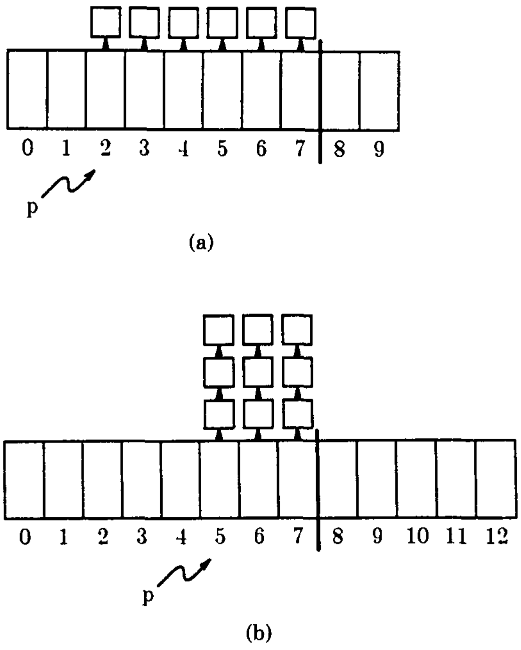


Figure 14. Uneven distribution of overflow chains.

After one partial expansion, we have the situation in Figure 16. The pointer p has made one pass through the four groups and has returned to group 0. We now have visited each page in the original file and have added four pages to the file. Thus we are halfway to a full doubling of the file but have already visited each original page once. Group 0 now consists of pages 0, 4, and 8. The next page added will be page 12, and the records from pages 0, 4, and 8 will be spread over pages 0, 4, 8, and 12. If the three pages being "split" were full, then after the "split" the four pages will be $3b/4b = 75\%$ utilized. When this partial expansion is finished, there will be 16 pages, that is, the file will have doubled and one full expansion will have been completed. Now, instead of having a group with four pages, we return to groups with two

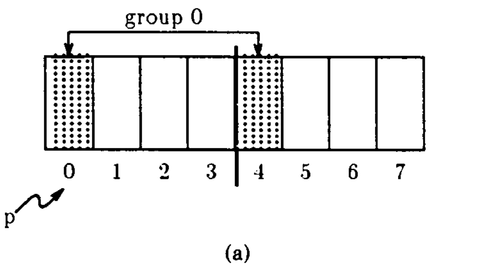
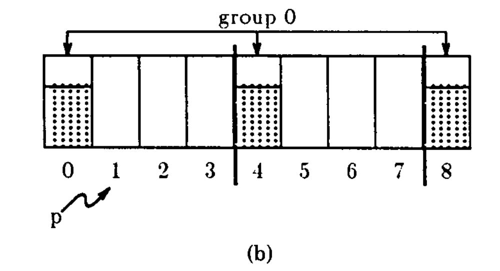



Figure 15. Partial expansion. (a) Before split. (b) After split.

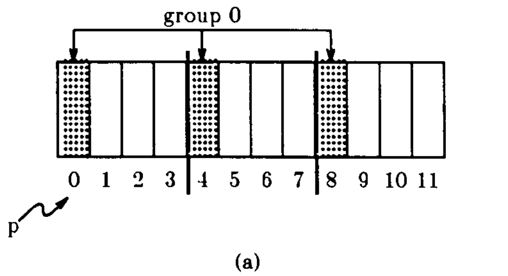
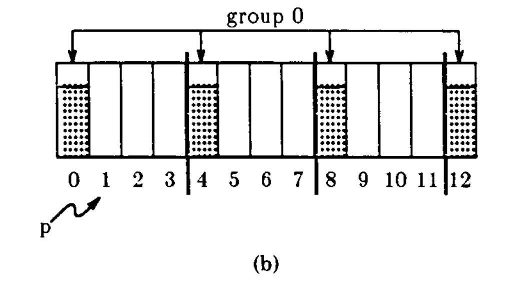



Figure 16. Partial expansion, second pass. (a) Before split. (b) After split.

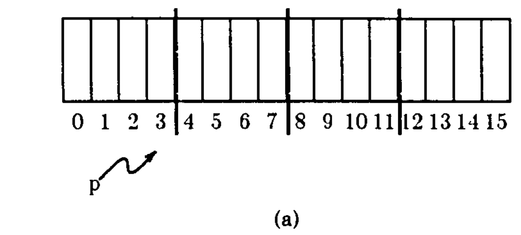
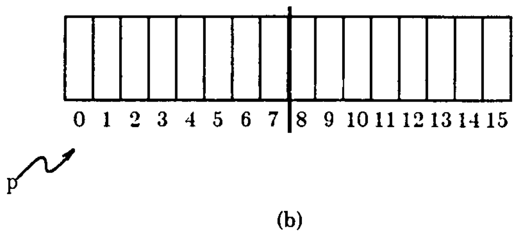



Figure 17. Partial expansion, beginning of next full expansion. (a) $d = 3$. (b) $d = 4$.

pages as shown in Figure 17. The number of groups has now increased to eight groups with two pages each and the whole process will repeat.

The problem is how to address the file as it expands. Larson [1980] and Ramamohanarao and Lloyd [1982] have each proposed a scheme for addressing the records. To confuse the issue they each defined the word "group" differently. We used Larson's definition above. Here is the scheme proposed by Ramamohanarao and Lloyd [1982]: The address is a function of the record's key k, the depth of the (full) expansion d, the partial expansion n, and the split pointer p. The variables k, d, and p are as we have defined them earlier. Only the variable n is new. The address of the desired record's primary data pages is

$$r(k, d, n, p) = group + 2^{d-1} \cdot offset,$$

where *group* is as we defined it above and offset is the *offset* (in terms of number of pages) within the group. *Offset* will be determined by another hash function $i_j(k)$, which maps into the number of pages within a group. The value 2^{d-1} is the distance between pages in a group. For example, to address page 8 in Figure 15, we first go to page 0 ($group = 0$), then skip over two ($offset = 2$) sets (of size $2^{d-1} = 4$) of pages to get to page 8. The value of *group* will be determined by the hash function $h_d(k)$ as in the basic directoryless scheme.

The function $i_n(k)$ is similar to $h_d(k)$ but maps into the size of a group in the current partial expansion (e.g., three pages for group 0 in Figure 15b), and $i_{n-1}(k)$ maps into the size of a group in the previous partial expansion (e.g., two pages for group 0 in Figure 15a). The address is then determined by

```
r(k, d, n, p)
:= if $h_d(k) \geq p$
     then $h_d(k) + 2^{d-1} \cdot i_n(k)$
     else $h_{d+1}(k) + 2^{d-1} \cdot i_{n-1}(k)$.
```

The above scheme, presented by Ramamohanarao and Lloyd [1982], moves records around within the group as partial expansions occur. Larson's original scheme is similar, but it only moved the records that were destined for the newly expanded page. The reader is referred to Larson [1980] for details.

8.2 Nonuniform Key Distribution

8.2.1 *Nonuniform Key Distribution for Directoryless Schemes*

As explained earlier, with our basic directoryless scheme (linear hashing) and a uniform distribution we can expect more overflow pages to the right of the pointer p (and to the left of page $s - 1$) than the left (Figure 14). This situation occurs because those pages to the right are split later than those to the left, so they will tend to get more overflows. In other words, we expect the pages to the right to be overutilized and those to the left to be underutilized. A decreasing exponential distribution of keys can map more records to the left end of the file than the right. This distribution will tend to map more records into pages that are already split (i.e., to the left of the pointer p); that is, more records get mapped to the underutilized pages and fewer to the overutilized pages. The result is a more uniform distribution of overflow pages.

An approach for handling such an exponential distribution of keys in a directoryless scheme is spiral storage [Martin 1979; Mullin 1985]. The spiral approach is a neat, logical description of a way of handling an exponential distribution of keys. The resulting file, which has no directory, can then be mapped onto a directoryless scheme.

Directoryless schemes have a fixed origin and expand by adding pages at one end of the file. Spiral storage differs in that, during expansion, pages are deleted at one end of the file and added at the other end. The result is a contiguous file space that moves through memory in a way similar to a queue. Expansion is achieved by adding more pages than are deleted. Records from the deleted portion are spread across the added pages. The essential part of the spiral approach is an addressing function that can adapt to the remapping of the records that are moved. We will begin with a discussion of the spiral addressing technique and then show how it maps into a directoryless scheme.

Spiral storage works by taking keys that initially have a uniform distribution, giving them an exponential distribution, and then assigning logical page addresses. This technique is illustrated in Figure 18. The keys are uniformly distributed along the x axis between c and $c + 1$ using a function $G(c, k)$. G takes a key k, applies a uniform hash function $hash(k)$, $0 \leq hash(k) \leq 1$, and then distributes the values between c and $c + 1$:

$$G = \lceil c - hash(k) \rceil + hash(k).$$

The y values are determined by b^G, with $b > 1$; the logical page addresses are $\lfloor b^G \rfloor$. As the interval $(c, c + 1)$ is moved to the right along the x axis, the size of the interval (b^c, b^{c+1}) will increase, causing the number of pages to increase. Moving the interval $(c, c + 1)$ to the left, causes the number of pages to decrease. During expansion the parameter c is chosen to eliminate the current first logical page completely. The new value of c is obtained from the inverse of the exponential function

$$new\_c = \log_b(first + 1),$$

where *first* is the current first logical page.

This process is best illustrated with an example as shown in Figure 19. Initially, the file is as shown in Figure 19a. The file is shown on the left, and the mapping on the right. All records within the interval $(c, c + 1)$ map into pages 1 and 2. In

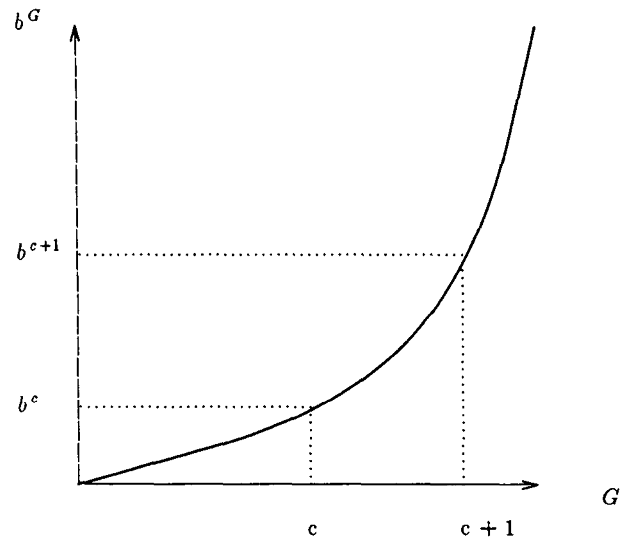


Figure 18. Graph of b^G versus G.

Figure 19b, the file has expanded, deleting page 1 and adding pages 3 and 4. The mapping on the right shows how all records within the interval $(c, c + 1)$ map into pages 2–4. Shifting the interval $(c, c + 1)$ to the right, changes the mapping so logical page addresses fall within pages 2–4. The parameter c is selected so that page 1 can be eliminated. The formula to determine the new c is $c = \log_b(\mathit{first} + 1)$, where *first* is the leftmost page, that is, the page to be eliminated. In Figure 19b, $\mathit{first} = 1$ so $c = \log_2(1 + 1) = 1$. A further expansion is shown in Figure 19c, with page 2 deleted and pages 5 and 6 added; the corresponding mapping is shown on the right with $c = \log_2(2 + 1) = \log_2 3$. Contraction is performed in an analogous fashion, with the interval $(c, c + 1)$ sliding to the left.

One apparent problem in what has been described so far is that the file moves through the address space. Unless the space taken up by the "deleted" pages can be reused, that space can be wasted in most operating systems. The solution is to map these logical pages into actual pages. The result is a file structure that looks like our basic directoryless scheme.

Let us return to the previous example to illustrate how this mapping can be done. When a page is "deleted" it should be reused immediately. Figure 20 shows the actual pages used for the logical pages of Figure 19. The numbers within the pages are the *logical* numbers of the pages; the numbers outside the pages are the *actual* numbers of the pages. In the first expansion, when logical page 1, which resides in actual page 0, is mapped into logical pages 3 and 4, the actual page 0 is reused for the logical page 3. Logical page 4 (physical page 2) is added to the (right) end of the file. In the second expansion, when logical page 2 is deleted and logical pages 5 and 6 are added, actual page 1 will be reused for logical page 5. Logical page 6 will be added to the end of the file. An additional expansion is shown in Figure 20d: Logical page 3 is deleted, and logical pages 7 and 8 are added.

The procedure to find the actual page from the logical page is as follows: If the logical page resides in the first instantiation of the actual page, then the actual page is the difference between the last logical page and the current first logical page, that is, the current number of logical pages. For example, the actual page for logical page 8 is $8 - 4 = 5$. To find a reused page, such as where logical page 7 resides, one must search back through the logical instantiations to find when the actual page was instantiated. For example, we know that logical page 7 is in actual page 0. That actual address can be determined as

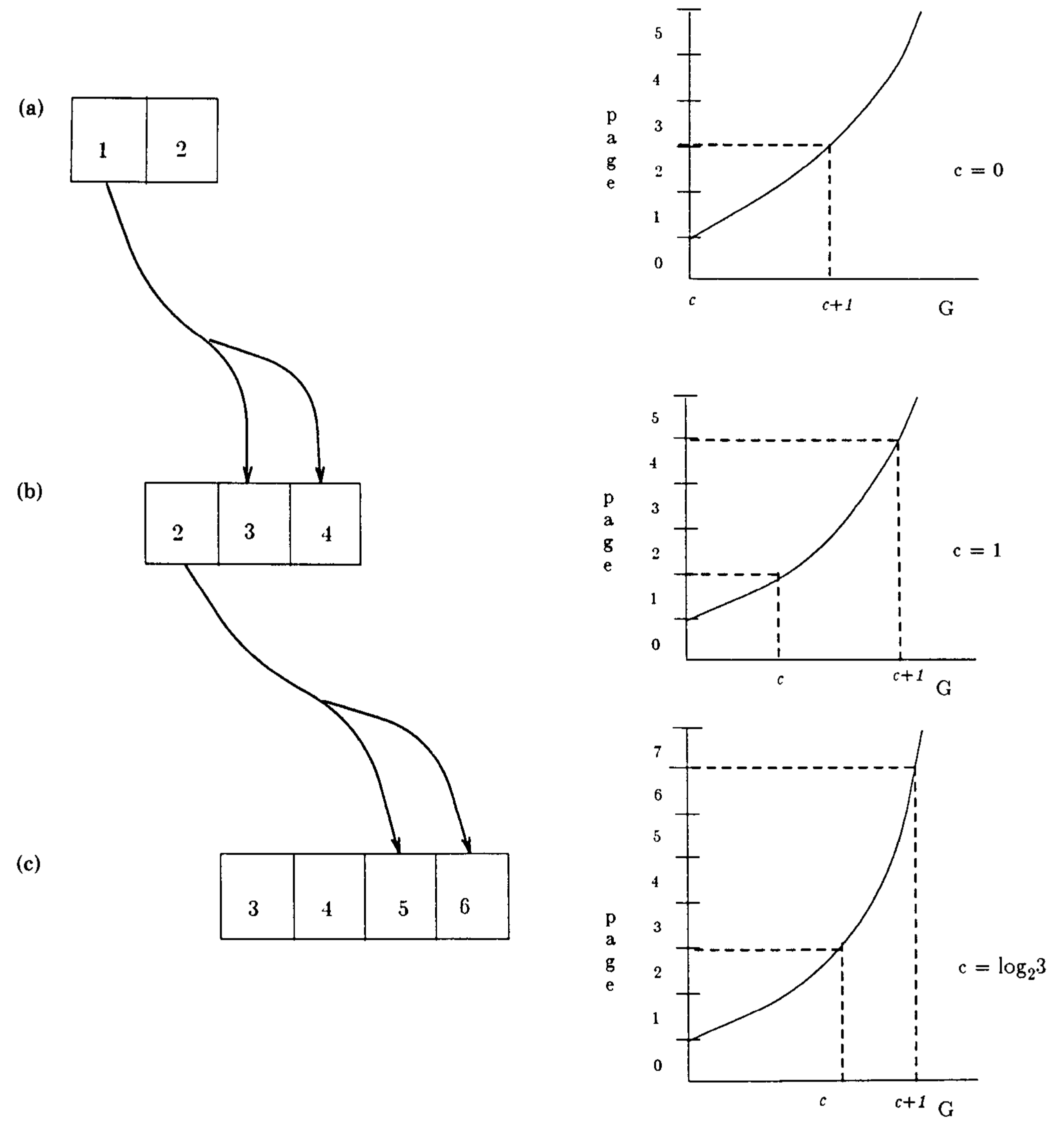


Figure 19. Spiral storage example.

follows: The records in logical page 7 were previously mapped into logical page 3 before the last expansion. Before that expansion, the records in logical page 3 had mapped to logical page 1, which resided in the initial actual page 0. Therefore, logical page 7 resides in actual page 0.

The recursive function to determine the actual page given the logical page follows; remember that *logical_page* = $\lfloor b^G \rfloor$:

```
FUNCTION actual_page (logical page)
  (* determine immediate "ancestor" pages *)
  high := $\lfloor (1 + logical\_page)/b \rfloor$
  low := $\lfloor logical\_page/b \rfloor$
  if low < high
    then
      (* reused page; keep searching *)
      actual_page := actual_page(low)
```

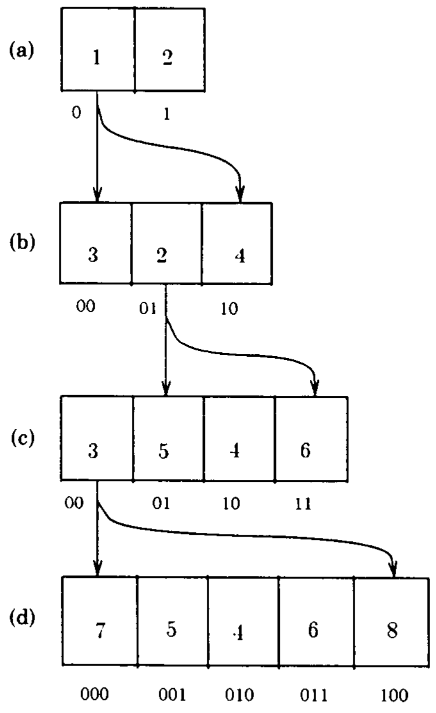


Figure 20. Mapping of logical pages to actual pages.

```
else
    (* low = high; found first instantiation
       of page *)
    actual_page := logical_page-low
end
```

Mullin [1984] presented a scheme to unify linear hashing (our basic directoryless scheme), linear hashing with partial expansions, and spiral storage. By using a piecewise linear curve with a shape that is similar to the curve in Figure 18, he showed how to implement linear hashing and linear hashing with partial expansions using the spiral approach.

8.2.2 Nonuniform Expansion of a Directory Scheme

As mentioned earlier, a nonuniform distribution can smooth out the expansion process in dynamic hashing schemes. With our basic directory scheme, however, a nonuniform hash function would lead to a nonuniformly utilized directory. For example, consider page *c* in Figure 7c, which has eight pointers pointing to it. Fully half of the directory entries point to one page, which amounts to a tremendous waste of directory space. Bounded Index Exponential Hashing (Section 7.3) was proposed as a way of handling nonuniform distributions by expanding some nodes more than others; Figure 13 shows such a distribution. In that figure there is an exponential distribution of pages (and records), yet the directory is fully utilized.

A different way of handling a nonuniform distribution with a directory is a scheme presented by Larson [1978]. His approach is a variation of our basic directory scheme in that it does not store its directory as a simple table. Larson proposed a forest of index tries whose roots are located by hashing to a table. The data pages are leaf nodes. Each index trie can grow or shrink according to the load on the file. Whenever a leaf node's page gets too full, the node can split into two. A page is addressed by first using a subset of the key to hash into the table to locate the root of the appropriate trie. The (binary) trie is then searched using the bit values to determine which branch to take. Larson did not let his directory grow or shrink, only the index tries did that. His scheme, however, can be modified to allow a dynamic directory. This modification is possible because the directory can be viewed as a trie collapsed into a table.

Otto [1988] has proposed a method of implementing an extendible hashing scheme using a balanced *m*-way tree as a directory. This results in a linear growth of the directory even in the case of nonuniform distributions.

9. PHYSICAL IMPLEMENTATION

Up to this point we have not considered the issue of physical implementation. For example, a directoryless scheme assumes a contiguous address space. If this space had to be contiguous in physical memory, a large piece of physical memory would have to be reserved, which would remove some of the advantages of dynamic hashing. The easiest solution is to let the operating

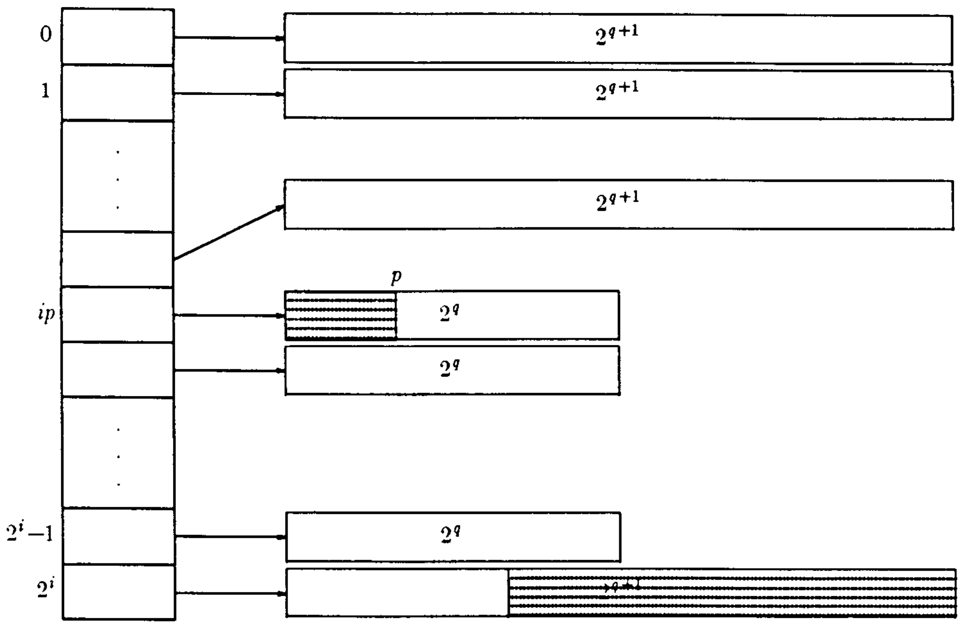


Figure 21. Physical mapping of a directoryless scheme.

system do most of the work. Let the dynamic file system map into a contiguous *logical* space and let the operating system handle the mapping to a dynamic, possibly noncontiguous, *physical* address space.

In this section we examine a proposal by Ramamohanarao and Sacks-Davis [1984] that considers a mapping of a contiguous directoryless space to a physical address space that is allocated in quanta greater than single pages, but smaller than the whole address space of the file. This mapping can be done with a fixed table of pointers to contiguous blocks of memory. Figure 21 shows a scheme without partial expansions. Quanta of memory are allocated in blocks with 2^x pages for various values of x. At the beginning of a full expansion, the physical blocks will all be the same size; as the dynamic hash file expands, larger blocks of memory will be used. The figure shows two different size blocks in use, one with 2^q pages and the other with 2^{q+1} pages. The blocks that are adjacent in the table are adjacent in the logical scheme. The pointer p in Figure 21 is the same split pointer shown in Figure 9. Blocks above the split pointer p in Figure 21 are larger (size 2^{q+1}) than the block that p is currently in (size 2^q) and store pages that have already been split, thus the need for twice as many pages. The last directory entry (2^i) is a temporary storage area in which pages currently being split can be stored. Pages from the block currently containing p are split into this temporary storage area, thus the need for twice as many pages. The shaded region to the left of p represents pages currently not being used. They have been split and remapped into the unshaded region in the left part of the temporary storage page. The shaded part of the temporary storage page contains the pages that the rest of the pages in the block currently containing p will be split into. A key part of this technique is that when a page is split the resulting pair of pages will be stored in physically *adjacent* pages.

A closer look at how addressing is done will make the structure clearer. The depth will be $i + q$, where each block at the current level is of size 2^q, and there are i directory entries. Let x be the number of pages to the left of p, that is, the shaded region in Figure 21. Since p denotes the number of pages that have been split during the current

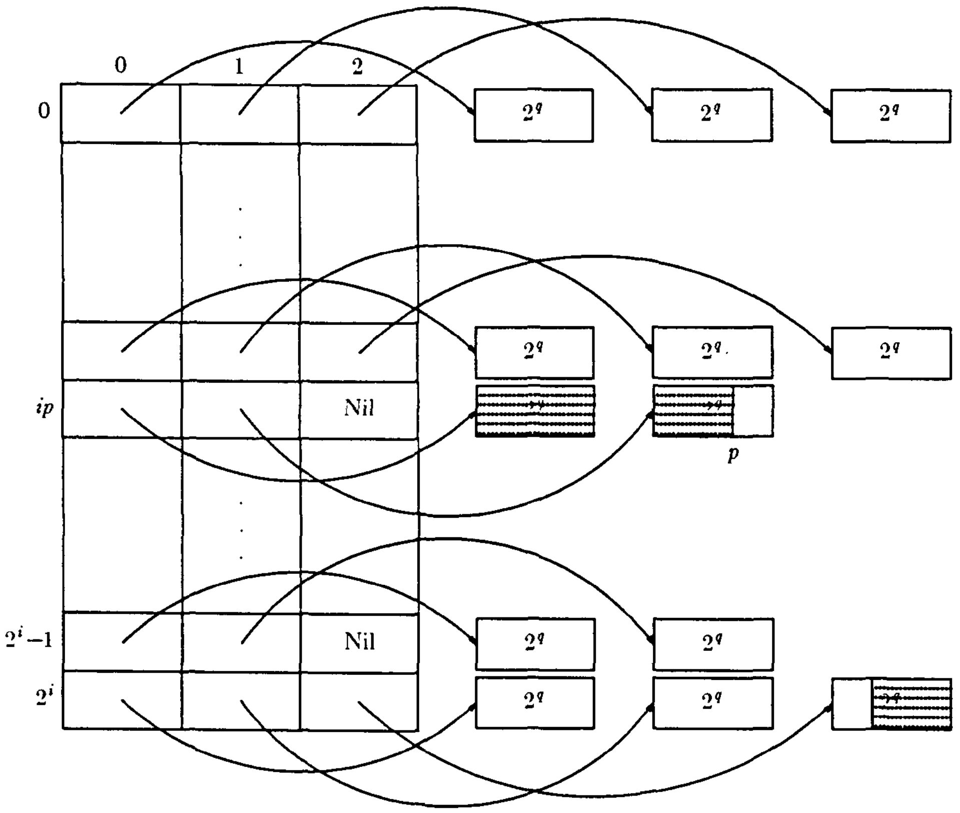


Figure 22. Physical mapping of a directoryless scheme with partial expansion.

expansion,

$$x = p \bmod 2^q,$$
$$ip = p \operatorname{div} 2^q.$$

We form the address by considering the leftmost $i + q$ bits of $H(k)$ to get $h_{i+q}(k)$. Addressing is then done by the following procedure:

```
begin
  if $h_{i+q}(k) \geq p$
  then the leftmost $i$ bits determine the directory entry and the next $q$ bits determine the offset within that block.
  if $h_{i+q}(k) < p$ and the leftmost $i$ bits choose a directory entry less than $ip$
  then the primary page has been split and is in one of the newly formed blocks of size $2^{q+1}$; the offset will be determined by the next $q + 1$ bits.
  if $h_{i+q}(k) < p$ and the leftmost $i$ bits choose directory entry $ip$
  then the primary page has been split and is in the buffer block $2^i - 1$; the offset will be determined by the next $q + 1$ bits.
end
```

If we are using partial expansions, there will be a contiguous block of storage for each page in a group. For example, if each group has two pages, there will be two blocks of storage. When a split occurs and the two pages are split over three pages, this expansion will be done into a third block of storage. Figure 22 illustrates a partial expansion in progress. Once again, p is in the ip region. Some of the records have been split into three pages whereas others to the right of p are still stored in two pages. The directory for partial expansions must be able to have as many pointers as there are possible pages within a group. On successive full expansions the size of the contiguous memory blocks will double in size, as above.

Consider the mapping illustrated in Figure 21. On the left side of the figure is a directory, with each entry pointing to a directoryless file. Compare that with Figure 13, which illustrates Lomet's Bounded Index Exponential Hashing approach [Lomet 1983]. In that scheme each multipage node pointed to by a directory entry could behave like a simple directoryless scheme. The result is that both figures look somewhat like a directory of directoryless files. This observation could prove useful for developing a combined scheme.

10. CONCLUSION

Both directory and directoryless schemes should be able to handle dynamic file systems better than static hashing schemes. They can expand and contract gracefully while maintaining reasonable access times and space utilization. In this paper we have presented a number of variations that let one adapt a scheme to one's needs.

A comparison of performance has not been done between basic schemes. However, some observations can be made. Directoryless single-key schemes have the potential for lower *average* access times since they have eliminated the directory. The partial expansion variations with directoryless schemes allow a file to expand with more grace than the doubling that is necessary with a directory scheme. Directory schemes also have unique capabilities. Since overflows are optional in a directory scheme, an upper bound can be placed on the maximum overflow allowed and hence the maximum access time. This limit could be important in some performance-oriented applications. Directoryless schemes cannot place a similar bound on the length of overflows. Also, the subtle effects of physical implementations on performance of many of the variations presented are not clear.

There are unresolved issues with respect to the distribution of hashed key values. Most schemes assumed a uniform distribution, but little work has been done on nonuniform distributions and their effect on performance. It has been proposed that in some cases a decreasing exponential distribution is desirable. This distribution can work well with a directoryless scheme, but any skewed distribution can cause a directory scheme to generate too large a directory. Also, an *increasing* exponential distribution would amplify the problems that a decreasing exponential distribution attempts to solve in a directoryless scheme.

The handling of overflows is another important part of the dynamic hashing schemes. If one wants space utilization above 69%, then overflows are necessary in either basic scheme. There are many proposals about how to handle overflows, but it is not clear what is the best way to manage them.

Dynamic hashing has developed to the point at which useful implementations can be made. In this paper we only discussed the design issues for single-key dynamic hashing. Many of these schemes can be extended to fit multikey associative retrievals that are more desirable in many database applications. The design issues for multikey retrieval merit a survey of their own.

ACKNOWLEDGMENT

This research was supported in part by NSF grants DCR-8405498 and DCR-8420935.

REFERENCES

CARTER, J. L., AND WEGMAN, M. N. 1979. Universal classes of hash functions. *J. Comput. Syst. Sci. 18*, 2 (Apr.), 143–154.

COMER, D., AND SETHI, R. 1977. The complexity of trie index construction. *J. ACM 24*, 3 (July), 428–440.

FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. R. 1979. Extendible hashing—A fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 3 (Sept.), 315–344.

FLAJOLET, P. 1983. On the performance evaluation of extendable hashing and trie searching. *Acta Inf. 20*, 345–369.

LARSON, P.-A. 1978. Dynamic hashing. *BIT 18*, 184–201.

LARSON, P.-A. 1980. Linear hashing with partial expansions. In *Proceedings of the 6th Conference on Very Large Databases* (Montreal). Very Large Database Foundation, Saratoga, Calif.

LARSON, P.-A. 1982. Performance analysis of linear hashing with partial expansions. *ACM Trans. Database Syst. 7*, 4 (Dec.), 566-587.

LARSON, P.-A. 1985. Linear hashing with overflow-handling by linear probing. *ACM Trans. Database Syst. 10*, 1 (Mar.), 75-89.

LITWIN, W. 1980. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th Conference on Very Large Databases* (Montreal). Very Large Database Foundation, Saratoga, Calif., pp. 212-223.

LOMET, D. B. 1983. Bounded index exponential hashing. *ACM Trans. Database Syst. 8*, 1 (Mar.), 136-165.

LUM, V. Y., YUEN, P. S. T., AND DODD, M. 1971. Key-to-address transform techniques: A fundamental performance study on large existing formatted files. *Commun. ACM 14*, 4 (Apr.), 228-239.

MARTIN, G. N. 1979. Sprial storage: Incrementally augmentable hash addressed storage. Tech. Rep. 27, Univ. of Warwick, Coventry, U.K.

MENDELSON, H. 1982. Analysis of extendible hashing. *IEEE Trans. Softw. Eng. SE-8*, 6 (Nov.), 611-619.

MULLIN, J. K. 1981. Tightly controlled linear hashing without separate overflow storage. *BIT 21*, 390-400.

MULLIN, J. K. 1984. Unified dynamic hashing. In *Proceedings of the 10th Conference on Very Large Databases* (Singapore). Very Large Database Foundation, Saratoga, Calif., pp. 473-480.

MULLIN, J. K. 1985. Spiral storage: Efficient dynamic hashing with constant performance. *Comput. J. 28*, 3, 330-334.

OTTO, E. J. 1988. Linearizing the directory growth in order preserving extendible hashing. In *Proceedings of the 4th Data Engineering Conference.* IEEE, New York, pp. 580-588.

RAMAKRISHNA, M. V., AND LARSON, P.-A. 1988. File organization using composite perfect hashing. *ACM Trans. Database Syst.* To be published.

RAMAMOHANARAO, K., AND LLOYD, J. W. 1982. Dynamic hashing schemes. *Comput. J. 25*, 4, 478-485.

RAMAMOHANARAO, K., AND SACKS-DAVIS, R. 1984. Recursive linear hashing. *ACM Trans. Database Syst. 9*, 3 (Sept.), 369-391.

SCHOLL, M. 1981. New file organizations based on dynamic hashing. *ACM Trans. Database Syst. 6*, 1 (Mar.), 194-211.

SEVERANCE, D., AND DUHNE, R. 1976. A practitioner's guide to addressing algorithms. *Commun. ACM 19*, 6 (June), 314-326.

VEKLEROV, E. 1985. Analysis of dynamic hashing with deferred splitting. *ACM Trans. Database Syst. 10*, 1 (Mar.), 90-96.

VITTER, J. S. 1982. Implementations for coalesced hashing. *Commun. ACM 25*, 12 (Dec.), 911-926.

Received November 1986; final revision accepted October 1987.