A HIGH PERFORMANCE, UNIVERSAL, KEY ASSOCIATIVE ACCESS METHOD

David B. Lomet

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

Abstract

A new file organization is proposed that combines the advantages of digital B-trees and extendible hashing methods into one organization that can be used universally. The method, like these predecessors, relies on digital searching. The key notions are: (i) that multipage nodes are addressed by the root and can have both data and index entries, the mix of entries changing over time; and (ii) that these nodes can be doubled with file growth and, when this occurs, data nodes at the next level of the tree are absorbed into the pages of these nodes, frequently keeping data closer to the root and simultaneously improving utilization. The result is an unbalanced tree that we call a digital lopsided tree or DL-tree. The paper describes DL-trees and their operations, and examines their properties. The most important engineering issues involve the doubling process and the methods used to optimize the tree properties. Ways of dealing with these issues are suggested.

I. Introduction

There are two properties that one would like an ideal access method to possess. They are: (i) applicability to all files; and (ii) performance that is as good as special purpose access methods for the situations that those methods were designed for. No current access method meets the requirements of an ideal access method. Our claim is that the access method proposed here comes surprising close to the ideal and hence should be seriously considered for almost all key accessed files.

Let us enumerate the specific criteria by which an ideal universal access method must be judged. The access method must:

1. support random access using a key in approximately one disk access when the keys are uniformly distributed, as done by the best performing hashing methods [7,9,11].
2. support sequential access from some arbitrary, key determined starting value. The best performing such method is digital B-trees[8], where the initial probe requires two disk accesses, even for rather skewed key distributions. Subsequent sequential access must successfully retrieve several records per disk access, as is common in tree organized indices.
3. cope with virtually all key distributions while maintaining adequate storage utilization. B-trees [1] are most flexible in this regard.
4. adapt smoothly to file size changes over an enormous range without sacrificing its access performance or storage utilization properties. B-trees, DB-trees, and several variants of extendible hashing possess this property.
5. be implementable in a relatively straightforward way and not be some amalgam of methods, with complicated case analysis and format changes required as one shifts from one method to another. That is, it must be one integrated method.

Measured by any of the five criteria, the access method introduced here does very well. Its most important potential limitation is poor utilization that might result from arbitrary and potentially adverse key distributions. A major aspect of the method is its response to this problem. The method uses digital searching, an effective way of achieving high performance.

Digital Searching of Files

Tree search methods share certain characteristics. Data stored high in the tree is used in conjunction with the search argument to locate the place lower in the tree where the search is to be continued. An important characteristic of all tree searches is that at each descent to the next lower level of the tree, a smaller portion of the file remains as the possible location of the data associated with the search argument. For

B-tree files, one typically searches within a node for the smallest index term that is larger than the search argument. Associated with this index term is a pointer to the node at the next lower level of the tree where the search is to continue. This form of searching we call *comparison file searching* since the path down the tree is determined solely by search argument to index term comparison.

With *digital file searching,* the appropriate index term must be located within a page as is done with comparison file searching. However, in addition, an address computation is then used to further narrow the search. In digital B-trees (DB-trees [8]), each entry consists of a key fragment together with a pointer and a size field. The size field is the logarithm, base two, of the number of pages contained in the node referenced by the pointer. Figure 1 illustrates the form of a DB-tree. Searching in DB-trees proceeds as follows. If a prefix of the search argument matches the index term, then the next several bits of the search argument that follow this prefix, the number of bits indicated by the size field, are added to the pointer associated with the index term to determine the precise page of the node at the next lower level of the tree at which the search is to continue. This is like many "index terms" sharing a pointer and using address computation to identify the location of further search.

Extendible hashing methods [4,6,7,9,11] are included in this description of digital file searching as degenerate cases. In these methods, the root of the tree can usually be considered as consisting of a single index term which is the empty string. All search arguments can be treated as having the empty string as a prefix. The size of the next lower level and its starting location must also be known. The result is that search proceeds from the "root" to the next lower level of the (implicit) tree by pure address computation, in exactly the same fashion as the address computation part of the search occurs in the more general case.

Advantage of Digital Searching

The advantage of digital searching on search speed results from the fact that each node of the tree has a much larger fanout than the fanout possessed by a B-tree node using comparison searching alone. Because of the larger fanout, the height of the digital tree is usually smaller than the height of a B-tree for the same size file, because tree height is given by

$$height = \log_{fanout}(file\_size).$$

In fact, it is possible to hold the height of a digital search tree to any desired level by increasing the number of pages in the nodes of the tree and hence increasing the amount of data accessed through each index term. This is accomplished by doubling the size of the node when too much data must reside on a single page of the node. This node doubling then splits the data of each original page between two new pages of the doubled node. Node doubling can accomodate part or all of the file growth. In comparison search methods, such as B-trees, file growth is accomodated by splitting the overflowing node into two nodes and placing a new index term in the higher level node that referenced the original node. Such splitting eventually leads to an increase in the height of the tree when the root becomes full and is split.

This ability to keep the height of the tree at any desired level has the marvelous consequence that the number of disk accesses can be kept to any desired number greater than zero. (It can only be zero if the entire file can fit in main memory.) Since the number of disk accesses is the determining factor for response time and the dominant factor in system throughput, due to the computational cost of servicing a disk access, this is of enormous importance. It is this property that permits data in key sequenced files to be reached with two disk accesses per random retrieval using DB-trees [8] while data in hashed key files can be reached in close to one disk access per random retrieval using BEH hashing [9].

Difficulties of Digital Searching

Digital search trees do have a disadvantage not present with comparison searching. Comparison search trees can be re-balanced easily because arbitrary tree rotations can be performed within a node. This is reflected in the fact that any key in the key space spanned by a node can become the new index term when the node is split. Digital search trees cannot be rotated at all. All balancing of digital search trees results from either (i) a page of the tree containing an unbalanced tree fragment or (ii) the ability of index terms to "cover" dramatically different sized portions of the file, as illustrated in Figure 2. The use of (i) is limited by the page size used. Method (ii) is realized by doubling the number of pages in lower level nodes referenced by the index terms and can be carried out indefinitely. However, each doubling makes it more difficult to maintain adequate storage utilization in the node. Since pages of the node fill up unevenly, even when the distribution of keys is uniform (Poisson), one may be forced to double a node even when some of its pages are under utilized. With skewed key distributions, the result can be disasterous.

In our previous papers, two responses to this storage utilization problem were explored.

1. If the part of the file affected by the utilization decline is a small fraction of the file size, e.g. if it represents only the upper nodes of the tree, then

fairly large declines in storage utilization can be accepted without there being a sizable impact on the overall storage utilization of the file. This is the approach pursued with digital B-trees.

2. If node doubling is employed at the leaves of the search tree, where the great bulk of the file's storage exists, even small declines in utilization must be resisted. If the key distribution is fairly uniform, it becomes possible to maintain adequate utilization by resorting to overflow pages for those pages of the node that become full prior to adequate utilization being achieved for the entire node. This response has been used in the variant of extendible hashing called bounded index exponential hashing (BEH) [9]. Data is then reachable in close to a single disk access. Only data on overflow pages requires two disk accesses.

If a file organization is to be universal, then a dilemma exists. The organization must use digital searching of nodes containing data when the key distribution is approximately uniform so as to insure good performance. On the other hand, should the key distribution be skewed, giving rise to a utilization problem in multipage nodes, then multipage nodes must be used mainly as index nodes, a large part of the data being stored in the next level of the tree in order to avoid a file utilization disaster. In the next section, it is shown how digital lopsided trees accomplish precisely this.

II. Digital Lopsided Trees

Overview

Digital lopsided trees escape from the good performance/poor utilization dilemma by storing both data and index entries in pages of the nodes that are immediately below the root of the tree. We now refer to nodes immediately accessed from the root as primary nodes. Nodes accessed from the primary nodes are called secondary nodes. There are three key notions involved in making this approach work.

1. A page of a primary node initially contains only data and references no secondary nodes. This data is called the primary data and satisfies search requests in a single disk access when this page is accessed from the root. When such a page first overflows, an index is constructed which will contain two index entries. One index entry will refer to the primary page itself which will continue to contain approximately half of the data entries. The second index term will refer to a newly created secondary node that will contain the remainder of the data entries. As the file continues to grow, additional index terms refering to additional secondary nodes can be included in this primary page index. The subtree spanned by a primary page can contain anywhere from less than one page of data entries to many secondary nodes worth of data entries. Thus, a page of the primary node can slowly be converted from a data page to an index page. The data moves from being one disk access from the root to being two disk accesses from the root. This idea first appeared in a report by Diehr[3].
2. The primary nodes can increase in size by doubling. This will replace each page of a primary node by two pages and hence make extra space available in the primary node only one disk access from the root. This space is used for additional primary data entries that are absorbed from the secondary nodes back into primary pages. If doubling and absorption are performed in a timely manner, DL-tree search performance can be kept close to one disk access for a uniform key distribution. For skewed distributions of keys, the data is always, on average, somewhat less than two disk accesses from the root because primary data will always be present in some primary pages.
3. Even after absorbing all primary data, it is possible for a primary page to be dramatically under utilized. We call such a page a virtually empty primary page. If such pages were allowed to proliferate, which might occur if the key distribution were highly skewed, the overall file utilization would be severely compromised. However, such pages can be used to contain the secondary nodes of other primary pages. The secondary data so absorbed is called "alien" data as it is stored in a primary page whose first purpose is to provide space for data entries whose keys form a different part of the key space. While this form of absorption does not move data closer to the root, it does solve the utilization problem. Further, since primary node utilization becomes satisfactory at an earlier point, the primary nodes can double at an earlier time. This will permit more primary data to be absorbed and will result in a performance improvement. Our intent is to store most, if not all, secondary nodes in under-utilized (virtually empty) primary pages.

It is the doubling of primary nodes and the absorption of data from secondary nodes, both primary data and secondary data, that permit DL-trees to maintain their excellent performance in the face of arbitrary key distributions and file growth. These are the crucial ideas that make DL-trees work well.

Page Organization in Primary Nodes

There are multiple demands on primary pages of DL-trees. Such pages can contain primary data, an index to secondary nodes, and the secondary node of some "alien" primary page. The data structures used in the primary page must enable such information to be readily distinguished and manipulated. Figure 2

illustrates schematically how primary pages are organized within the overall structure of a DL-tree.

The first distinction should be made between information native to the page and secondary node data that is "alien" to the page. The primary reason for this is to ease the problems of concurrency control. Logically, alien data should not interfere with the reading and updating of the native data. Changes in alien data should leave native information unchanged, both in content and in location and vice versa. A further source of contention is that native updates may compete with alien updates for unused space. One way to deal with this is to completely partition the space of the page into native and alien parts. An alternative, which permits more flexible use of the space of the page, is to synchronize explicitly when additional space in the page is required. That is, the free list of such space must be locked prior to its being used, and when the appropriate space has been acquired, this lock must then be released.

The native information consists of an index and, perhaps, primary data, i.e., data that can satisfy search requests without the need to access secondary nodes. For algorithmic simplicity, it is useful to access all native data via the index. The primary data could then appear in the index like any secondary node with the exception that no additional disk access would be required to reach it. While the exact form of the index is left open, the pointers in the index merit some discussion. There are two forms of such pointers. Either the pointer references its primary page or it references a secondary node. The primary page "pointer" need merely indicate that the data is stored in the same page. To access a secondary node contained in a remote primary page, it is useful to avoid using a disk address, even one in the form of a relative address. The reason for this is that when the primary node containing that remote primary page doubles, this page will be relocated, and hence, so will the secondary node that it contains. This will invalidate all pointers in the form of disk addresses. What is needed is a form of reference that can continue to find a secondary node even after the page in which it is contained is relocated due to primary node doubling.

Recall that primary pages are only one disk access away from the root. Thus, given a key prefix, the primary page holding data with that key prefix can be reached in a single disk access. If that page contains a secondary node, then the data of that node is likewise reachable in one disk access. This suggests that the key prefix for the containing primary page be used to reference a secondary node. This key prefix will, with a trivial transformation, continue to locate the correct page even after the primary node has doubled and been relocated. How this relocation is done is discussed in section III.

Because of the multiple roles played by pages of primary nodes, several cases arise when one of these pages overflows. If the page contains an alien secondary node, then we must either split this secondary node or move it elsewhere in its entirety, regardless of the event that triggered the page overflow. This will require accessing the primary page of this alien node, and this is an extra access if the triggering event involved a native update. If no alien secondary node is present and primary data is contained within the node, this primary data must either be split or moved elsewhere in its entirety. Finally, if only index terms exist on a primary page, then page overflow forces a doubling of the primary node. In the above cases where data entries are either split or moved in their entirety, the determining factor in making this choice is how much of the primary page remains available for use by the data in question. Should the remaining space be less than some threshold, e.g., 50%, then all the data in question is moved. Otherwise it is split.

Secondary Nodes

In DB-trees, not only were index nodes searched digitally, but data nodes were split digitally as well. The reason for this was to avoid the problem of having data nodes that contained entries, some being accessible only via one index page, with other data entries being accessible only via a second such page. Such a data node we call a boundary node. Digital splitting of data nodes usually avoids the occurence of boundary nodes but there is a price paid for this in terms of storage utilization. Analysis in [8] showed how utilization is affected by changes in split factor(s) that might result from digital splitting. Long term utilization is .693, when the data entries divide evenly, i.e. when $s = .5$, which is the same value as that for B-trees. As the value of s increases, i.e. as more data entries are placed in one of the resulting pages than in the other, long term utilization declines quite slowly at first, reaching .5 when $s = .773$ and then declines thereafter quite rapidly, going to zero as the split factor approaches one. Utilization is almost certainly unacceptable if split factors are consistently greater than $s = .8$, which indicates that .8 of the original data entries are placed on one page while only .2 of the data entries are placed in the other. In a simulation study, we found that utilization was fifty percent if the split factor was uniformly distributed between *.5* and *1.0*. This may be acceptable but it is clear that a higher value is to be desired. Even using digital splitting, however, should an under utilized index page reference only one data node, the splitting of this page would force us to either split the data node or take some other action so as to keep all the data accessible.

If we are prepared to cope with boundary secondary nodes, another strategy is possible. Instead of digital splitting, B-tree type splitting can be used in

which an equal number of entries is placed in each new secondary node. The result is a DL-tree in which digital searching is used in going from root to primary node and comparison searching is used to go from primary node to secondary node.

The frequency of occurence of boundary pages can be reduced if one follows the strategy suggested in [2] of always attempting to use the shortest separator, i.e. the shortest key space value (prefix) that divides the entries of a splitting data page, as the index term. If we do not insist on a split factor of *.5*, but rather let it be perhaps as large as *.75*, then there is a good chance that this separator will be the index term chosen by a digital split, and boundary nodes will then be avoided in these cases. Only when the digital split factor is larger than this would B-tree splitting be necessary that would result at times in boundary nodes. Not only might some boundary pages be prevented from forming by using this approach, but boundary nodes that do arise might subsequently by eliminated by using this shortest separator approach to node splitting. The decreased utilization of the data nodes that is caused by permitting this larger split factor is small [8].

The Root

The form of the root entry of a DL-tree is quite similar to the root entry of a DB-tree. That is, the entry contains a key fragment, a size field that holds the logarithm base two of the size of the primary node, and a pointer that refers to the beginning of the primary node. It is this pointer that needs some further elaboration. In [9], we suggested that the buddy storage allocation method was an ideal way of managing disk storage since nodes were an integral power of two in size. This holds for DL-trees as well. In addition, such a buddy scheme would naturally provide start addresses for primary nodes that were "aligned". That is, such addresses would always have trailing zeros greater than or equal in number to the value of the size field. This is important as it makes these bit positions of the pointer field available for another purpose. The primary node start address can always be recovered by masking out the information stored in these bits.

The decision as to when to double a primary node, and which node to double requires certain node specific information. So long as this information does not exceed the space available due to pointer alignment of primary nodes, the bit positions made available by node alignment can be used to store the information. In particular, it is possible to use these bits to count the number of pages of the primary node which possess a given property, since this number will fit into the available space. In section IV, we show how this space is used. This is important as it means that some node wide information can be maintained in the root with no space penalty, thus avoiding an extra disk access were the information to be stored elsewhere.

III. Primary Node Doubling

Conceptually, the doubling of a primary node is quite simple. The index and data entries, for each page of the node, are divided between the two new pages of the doubled node based on key space splitting. Thus, if all entries on some primary page have a key prefix 'x', then these entries are divided between the new pages such that those with prefix 'x' | | '0' are placed on the low order (0th) successor page and those with prefix 'x' | | '1' are placed on the high order (1th) successor page. This process is complicated, however, by the DL-tree characteristic of absorbing data entries from secondary nodes during the doubling process so as to improve both primary node utilization and search performance. These complications are dealt with below.

Boundary Nodes

As indicated in section II, doubling a primary node can give rise to boundary secondary nodes when digital splitting of secondary nodes is not universally used. There are only a small number of ways of dealing with boundary nodes and they are discussed below.

1. Ignore them during node reorganization. During a search operation, if the entry sought has no index entry in the primary page and it requires an index entry lower than (or higher than) all other such entries on the page, go to the adjacent page of the node and continue the search there. This approach results in an occassional extra access when the above situation occurs.
2. Put index entries for the boundary node in both primary pages so that its data entries will be accessible from either page. Extra algorithmic complexity arises when boundary node splitting occurs as extra care must be taken to assure the correct updating of primary node index entries. Extra complexity and care is required as well to insure locking consistency during concurrent operations.
3. Split all boundary nodes whenever they arise during primary node doubling so that a pure tree always exists and a pure tree search can be used. The cost involved is potential extra I/O to perform the split and a potential decline in storage utilization of these secondary nodes.

The absorption of secondary nodes already requires us to access some secondary nodes during the process of doubling a primary node. If we make these accessed nodes also be the boundary nodes, when such nodes exist, then there is a good chance that approach 3. above can be followed without cost that is much in

excess of the cost needed for the absorption process. It is this possibility that is explored below.

Absorbing Primary Data

Absorption of secondary nodes that will become primary data should always be done at the time that doubling occurs, and not delayed. While this makes the doubling operation more costly, there is no saving to be achieved by waiting. Delaying absorption means that the next access to those entries that would have been absorbed will cost the extra disk access. In addition, to absorb the secondary node then would require an extra write of the primary page. This extra write is avoided at doubling time by absorbing the secondary node when the primary page has to be written in any event.

The doubling process can be simplified by having the primary page always contain the low order data entries referenced by the index. Thus, whenever the primary page overflows, it is the high order data entries that are placed into the newly created secondary node, the low order entries remaining in the primary page. This reduces the complexity in terms of the number of cases that must be dealt with by the doubling algorithm as discussed below. It also means that in all likelihood only one of the new primary pages will need to absorb a secondary page since the low order page will inherit the data entries of the original page.

Because of the proposed organization, only five cases needed be distinguished. They are explained below. The entries in the original page are all prefixed by the string 'x'.

(a)All the data entries contained in the subtree rooted at the original page have keys with prefix 'x'||'0'. Thus, the original page is simply copied into the 0th page of the pair of new pages, the 1th page being left empty.

(b)There is a secondary node Aj that is a boundary node. Then the 0th page contains the original data entries plus index terms up to and including the term for Aj. The 1th page absorbs the data entries from Aj that have a prefix 'x'||'1' and contains the remainder of the index terms for the secondary nodes.

(c)There is no boundary page and the secondary pages divide between the 0th and 1th page after secondary page Aj. Then the 0th page contains the original data entries plus the index terms up to and including the term for Aj. The 1th page absorbs all or part of the data entries from Aj+1 and contains the remainder of the index terms for the secondary nodes.

(d)The data entries of the primary page form the boundary page. Then the 0th page contains the original data entries that have keys with prefix 'x'||'0'. The 1th page "absorbs" the data entries from the original page that have keys with prefix 'x'||'1' and contains the entire index for the secondary nodes.

(e)There are no entries, data or index, in the original page that have keys with prefix 'x'||'0'. The entire original page is copied into the 1th page, the 0th page being empty.

In cases (b), and (d), should the 1th page of any resulting pair of primary pages be under utilized, all or part of the secondary node contiguous to its data entries can also be absorbed when such an adjacent page exists. It is this sometimes necessary absorption of an extra secondary node that is the cost of eliminating boundary pages.

Note that absorption of primary data, whether it is from a boundary page or not, always requires an extra read. This extra read cost is recouped as soon as data is requested that would have been on a secondary node but is now primary data. No extra writes are required for the secondary nodes. If these nodes are completely absorbed, their storage can be freed. If they are only partially absorbed, such nodes can continue to contain redundant and even erroneous data until such time as they have to be rewritten. This is possible because the requests for these data entries are intercepted at the primary node, the redundant entries not being reachable in any search.

Dealing With Secondary Data

There are two issues that must be dealt with in handling secondary nodes and these are discussed below.

1. Relocating secondary nodes: Recall that a secondary node is referenced from its primary page index by means of the key prefix that specifies the primary page in which it is contained. When the containing primary node doubles, where one page existed previously, now there are two pages, and each subsequent node doubling doubles the number of pages. Thus, we need to decide in which one of these resulting pages to place the secondary node, and we need to know how to locate it. We choose, by convention, to place the secondary node in the 0th descendent page (as opposed to the 1th descendent page). In using a key prefix to locate a secondary node, should the prefix identify a block of pages instead of a single unique page, then the key prefix is extended with sufficient zero bits so as to select the 0th page of this block. By relocating the secondary node as described above, this referencing method will find the correct page.
2. Constructing the free list of "empty" pages: There are three sources of empty pages. One source is the primary pages vacated by the secondary nodes that are absorbed to become primary data. The second source of free pages is the virtually empty pages of

the doubled node itself. The third is pages of absorbed secondary nodes that were allocated independently of primary nodes. All these pages must be placed on the free list during the doubling process. Only the pages of the doubling node itself must all be accessed and written during the doubling process. The part of the free list with these pages can be maintained in the pages themselves since this does not involve extra disk accesses. The remainder of the free list can be maintained in the root to avoid having to write these empty pages. The list pointers themselves can be disk addresses and will not require any relocation because all the virtually empty pages from any primary nodes will be consumed before any primary node doubles again. This is explained below.

IV. Which Node to Double and When

Choosing the Node

In choosing a node to double, we wish to maximize the performance gain per new page added to the file. We have no direct way of computing the performance improvement that will result if a node is doubled. However, under the assumption that all entries are equally likely to be accessed, we can approximate the search performance of the file, in disk accesses, by

$$DA_{search} = 1 + \frac{E_s}{E_s + E_p} = 1 + \frac{E_s}{E_t}$$

where E_s is the number of data entries in secondary nodes, E_p is the number of primary data entries, and $E_t = E_s + E_p$ is the total number of data entries, i.e. records. Thus

$$\Delta DA_{search} = \left(1 + \frac{E_s}{E_t}\right) - \left(1 + \frac{E_{s\_new}}{E_t}\right)$$

or

$$\Delta DA_{search} = \frac{E_s - E_{s\_new}}{E_t} = \frac{E_{p\_new} - E_p}{E_t} = \frac{\Delta E_p}{E_t}$$

It is undesirable to try to maintain precise information as to the value of ΔE_p because of the large number of such entries and hence the amount of space consumed to store it. However, ΔE_p can be estimated in a reasonable fashion from quantities that are readily maintained. The accuracy of this estimate depends on the average utilization of primary pages absorbed during doubling being approximately equal to the average utilization of all secondary nodes in the file. This is usually a reasonable assumption. Thus we have

$$\Delta E_p \cong \Delta Primary\left(\frac{E_s}{Pages_s}\right)$$

where $Pages_s$ is the number of pages on which secondary nodes are stored, and $\Delta Primary$ is the number of secondary nodes of a primary node that would be absorbed so that their data becomes primary data of the node were the node doubled.

The node that we wish to double is then the primary node with the highest value of

$$\frac{\Delta DA_{search}}{2^N} = \frac{\Delta E_p}{2^N E_T} \cong \frac{\Delta Primary}{2^N}\left(\frac{E_s}{E_t Pages_s}\right)$$

where 2^N is the current size of the node and is the number of new pages added to the file were this node to be doubled. Note here that E_s, E_t and $Pages_s$ are properties of the file that are the same for all nodes. Hence, the ranking of the nodes depends solely on the node specific values of $\Delta Primary$ and node size (2^N).

The strategy for choosing the best node for doubling boils down to keeping a short list (perhaps of length about ten) of primary nodes with the best values of $\Delta Primary/2^N$. When additional empty pages are needed, the first node in the list, i.e. that node with the highest such value, is doubled. To maintain the list, whenever $\Delta Primary$ changes for a node, it must be compare to the current entries on the list, being inserted in the correct location if it qualifies, and bumping the lowest node off the list. Doubling always makes available about $\Delta Primary$ new empty pages since this new primary data must be absorbed from other primary pages, usually leaving virtually empty pages behind.

Choosing the Time

From the point of view of search performance, it is advantageous to double primary nodes so as to keep all data as primary data and not permit the formation of secondary nodes. Given a uniform key distribution, one can sometimes approach this ideal situation without sacrificing much in utilization and hence without wasting large amounts of disk space. With skewed distributions, the disk space penalty can become extreme. What is needed is a simple and efficient method of identifying at as early a point as possible when primary node doubling can occur without bringing with it disasterous utilization. Since we intend that secondary nodes usually be contained in virtually empty primary pages, the timing of the doubling process usually is driven by the availability of these pages. When these pages are in short supply, as they might be with a uniform distribution of keys, premature doubling of primary nodes can occur which drives down utilization in an effort to suppy the empty pages.

There is a way to solve this problem which simultaneously adds flexibility to the DL-tree approach. Rather than automatically doubling a primary node when the set of available virtually empty pages is con-

sumed, we suggest that, for each file, a doubling goal be established. This goal is the value that must be met or exceeded by file utilization before a primary node is doubled. If this goal is not reached at the time that the set of virtually empty pages is consumed, then a new page is independently allocated to accommodate the secondary node. A big advantage of the doubling goal technique is that the decision of when to allocate new independent overflow storage as the file grows is taken care of automatically. Such independent allocation continues until the file utilization is sufficiently high. At this point, primary node doubling would resume.

The existence of independently allocated secondary nodes complicates slightly the management of the free list of available pages. Previously, only pages from primary nodes were on the free list. Since no primary node would double prior to all free pages being consumed, there was no need to relocate free pages. We wish to retain this property. Thus, we propose to keep two separate free lists, one of available pages from primary nodes, and the other available pages from the independently allocated pool. All pages on the list comprising pages from primary nodes must be consumed prior to a node doubling. However, unused pages from the collection of independently allocated pages can exist at the time of primary node doubling.

With the existence of independently allocated secondary nodes and the file utilization doubling goal, it becomes possible to adjust the doubling point for primary nodes so as to control the trade-off between performance and storage. The higher the doubling goal, the better the utilization of storage since independently allocated secondary nodes have good utilization, and the worse the performance since the fraction of primary entries is reduced. In this way, however, while keeping performance at less than two disk accesses per search request, storage utilization should cease to be a problem, regardless of the size of the file and the size of the primary nodes. All but a very small number of pages are more than 15% full since the free list of virtually empty pages is always quite short, typically being less than halve the size of the most recently doubled primary node.

V. Controlling the Structure of DL-Trees

The preceding three sections have described the structure of DL-trees, how they can grow by node doubling, and how to determine when a node should be doubled. Unmentioned in all this is how the overall structure of the DL-tree is arrived at. This boils down to how one achieves the best performance given the limited number of root entries that can be accommodated in main memory. Put another way, it becomes the problem of determining the circumstances under which new primary nodes are introduced or previous primary nodes consolidated so as to best optimize performance.

There are two problems to be solved here. The ongoing problem is one of deciding when an existing node should be split into two or more nodes, thus consuming additional root entries, and when two existing nodes should be consolidated, thus saving a root entry. The second problem is the construction of the initial DL-tree from a potentially large collection of existing data. This is the file loading problem and frequently requires special measures. Both of these problems are discussed below.

For optimal results, we need a cost function in terms of performance and space such that, when we are given a certain number of root entries, we attempt to structure the DL-tree so as to maximize this function's value. This is a very hard problem. Indeed, we know of no method of solving this problem short of extensive search involving the examination of many possible DL-trees. The number of such trees, assuming we fix the number of root entries, is finite, but examining all these trees imposes an enormous computational cost. Further, since files grow and shrink, it is difficult to keep the DL-tree in an optimal configuration. Thus, what we need are heuristic methods that produces acceptable but not necessarily optimal results.

Initial File Growth

The method that we choose for initial file growth is to attempt to have the sub-trees associated with each primary node have approximately the same numbers of data entries. The justification for this approach is that (i) it limits the fraction of the file that is in the largest sub-tree and hence keeps the cost of maintaining its primary node within reasonable bounds; and (ii) each root entry, in the absence of detailed information about key distribution within each sub-tree, should have a reasonably equal chance to have and retain acceptable performance.

While we attempt to equalize the number of entries accessible via each primary node, we can in no way insist on near equality. If the nodes are all within a factor of four, or even eight, of the average node, that should be all that we expect or require. With key space splitting, it is almost impossible to do much better than that. In fact, key space splitting can spawn a series of small nodes in an attempt to limit the size of any primary node that contains a heavily utilized portion of the key space but is surrounded by underutilized regions.

Unlike the case with B-trees, where arbitrary re-balancing operations can be performed, DL-trees are restricted at the primary node level to key space splitting and cannot be re-balanced at all. Thus, it is very difficult to produce a reasonable initial DL-tree without using information about the distribution of keys. This information can either be determined by pre-examining the keys of data entries that are to be loaded initially or assumptions can be made based on a priori knowledge of the key distribution. We consider two approaches to initial file growth.

1. With advance knowledge of the overall key distribution, we can pre-allocate the root entries for the file so as to attempt to equalize the number of data entries that will be covered by each primary node. Initially, all pre-allocated root entries can reference a shared primary node that is one page in size and contains data entries for the entire file. When this page overflows, if all the entries are covered by a single root entry, then we double the node, having the appropriate root entry reference the doubled node. The remaining root entries that previously referenced the page now share "null" nodes as permitted by the key space division. If the entries of an overflowing page come from several root entries, we split the primary node, associating the descendent primary nodes with the appropriate root entries. Again, root entries can share primary nodes until such time as all root entries have their own primary nodes. After that point, further file growth is accomplished primarily by node doubling. We do not elaborate on this here but the general idea is to minimize the number of primary pages into which data is stored while the file is very small so as to maximize utilization and make it possible for some of the data pages to be retained in main memory.
2. If we can be confident that the the keys of the early data entries are a reasonably valid sample from the entire collection of keys of the file, then we can incrementally generate the root entries. We begin with one root entry that covers the entire key space. When its node overflows, the node is split by key space, with an additional root entry (or entries) referring to the newly generated pages. Whenever these one page nodes overflow, such splitting continues until the permitted number of root entries is reached. Thus, these early insertions generate root entries, each of which covers an approximately equal number of data entries. Again, subsequent file growth is accomplished primarily by doubling. An early valid sample of data entries can by assured by associating each data entry with a random number and then sorting the data entries on this number. This sorting by random number might arise naturally should a primary file be organized as a hashed file. The hashed keys would then be the random numbers and the sorting would occur in the construction of the hashed file organization. Secondary indices could then be generated, assuming the valid sample condition, by reading the hashed file in hashed key order.

Simple Node Splitting

After the initial growth phase, further growth of the file should not result in the need for rapid changes in the overall structure of the DL-tree. Nonetheless, it is reasonable to anticipate that some changes could be made over time that would result in improving the performance of the file. The availability of additional root entries is always helpful and a reasonable strategy is to slowly add root entries as the file grows. Our problem then becomes one of deciding how best to use these root entries. In this section, we consider only the simple case of determining which nodes it is most useful to split into two nodes, i.e. which of the primary nodes, when split, will lead to the largest performance improvement. As with the discussion on initial growth, we do not insist on optimality in this regard but merely look for a method that does well.

Primary nodes that are good candidates for splitting have two attributes. (i) If split into two nodes, one of the new nodes will double much earlier than the other. (ii) The node is fairly large, so that its $\Delta Primary$ is large and the improvement in file performance per additional root entry consumed is large. What we need is an inexpensive method of identifying nodes with these attributes and, from among the good candidates, making sure that we choose the best candidates and split them in a timely fashion.

The process of identifying the good candidate nodes involves accumulating node wide information and can be expensive because information must be drawn from each page. Evaluating nodes for splitting during node doubling seems quite attractive since doubling requires us to read and write all pages of the node. Thus while we have a page in main memory, the information needed to evaluate the node as a splitting candidate is available to us. Its accumulation during the doubling process requires no extra disk accesses. So long as the distribution of keys of the file changes only slowly over time, examining nodes for possible splitting during node doubling represents an inexpensive and effective approach.

We want to select nodes which, if split, will result in two nodes, one of which will double a good deal sooner than the other. Thus we need to estimate the future value of $\Delta Primary$ for each half of the node. We use the current occupancy of the pages of the node as the predictor of its future inclusion in $\Delta Primary$. We use the notation $Pages_{xx}$ to indicate the number of pages in one half of the node that have an occupancy of greater than xx%. When $Pages_{50}$ of one half of the node is significantly larger than $Pages_{35}$ of the other half of the node, then we can reasonably

predict that, if the node is split, that one of its resulting nodes will double much earlier than the other. Significantly larger probably should mean a factor of more than two but experimentation is needed in this area.

The purpose of splitting is to improve file performance. Our goal here is to produce as large an improvement in performance as possible for each additional root entry consumed. Thus, splitting candidates would consist only of large nodes since only large nodes would possess future $\Delta Primary$ values which would justify the consumption of the root entry. Only nodes that are at least twice as large as the average node need be evaluated since we want the new nodes, when doubled, to each have $\Delta Primary$ values that are larger than average.

After evaluating a node, we need to know, from among the good candidates, which nodes should actually be doubled. There are two ways to proceed which can be used separately or in combination. The simplest thing is to establish a threshold size for predicted future $\Delta Primary$ values for new nodes. Any good splitting candidate that is sufficiently large would be split immediately. The alternative is to keep a short list of the best candidates, ordered by predicted future $\Delta Primary$ values (these predictions can be based on $Pages_{50}$) and to periodically take the best one from the top of the list and split it.

Further Re-structurings

There are a vast number of other node re-structurings. Only the most likely merit evaluation. These are the ones that produce only small changes in the number of root entries. The simple splitting of a node, as discussed above, should be the most common case and requires only one additional root entry. Splitting off one quarter of a primary node requires at least two additional root entries. It is probably not worth examining the splitting the node into eighths, which requires up to seven additional root entries. In the case of node consolidation, a key space "buddy" of the node must be found. If the key space spanned by the doubling node has a key prefix of the form y = 'x'||'0' ('x'||'1'), then its key space buddy will have a key space with a key prefix of the form z = 'x'||'1' ('x'||'0'). If such a buddy does not exist, then the amount of re-structuring needed to eliminate a root entry is probably excessive. Limiting consideration to these re-structurings limits the number of alternatives to be evaluated as well as limiting the cost of evaluating them and the cost of performing the indicated re-structuring. These more complicated node re-structurings should probably be performed 'off-line', when the system is otherwise underutilized. At such times a background program could search the file, examining its nodes in some detail, and attempting to improve the file performance without adding an excessive number of new root entries. The discussion of such a program is beyond the scope of this paper. Indeed it may well prove a worthwhile subject for several papers.

VI. Improving Key Distributions

As we have noted previously, digital search trees are sensitive to the distribution of the keys. Much can be done with the keys themselves prior to building DL-trees that will improve the characteristics of the resulting file. A word of caution is in order, however. If key sequential access is required, then the key transformation must be one in which the transformed keys have the same ordering as the original keys. The suggested key transformations below will satisfy this requirement. If key sequential access is not required, then hashing should usually be used. It has the effect of balancing the tree and keeping the disk access cost of retrieval at close to one. There is no point in losing key sequentiality without realizing the gains that hashing will provide in terms of performance.

Key Space Gap Removal

The most important key transformation is what we call key space gap removal. To define key space gaps, let us begin by defining key space.

Key Space: The key space of a file is the range of binary numbers between the largest possible key value that can be represented and the smallest possible key value that can be represented, when their bits are interpreted as binary numbers.

We can then define a key space gap as follows:

Key Space Gap: A key space gap is a subrange of binary numbers within the key space such that no possible key will fall into that subrange.

The reason that key space gaps cause a problem for digital searching, and for DL-trees in particular, is that when they are encountered during node doubling or splitting, they result in a split factor $s = 1$. The effect of such a split factor is that all primary entries in one page, that we would expect to be split between two pages as a result of node doubling or splitting in fact will fall into only one of the new pages. Thus, a key space gap will result in the introduction of empty pages at such places. These empty pages will further multiply with each additional doubling of the node. Only when such empty pages can be split from the non-empty pages by node splitting, which is sometimes, but not always, possible is this difficulty avoided. These empty pages will, of course, usually be filled by means of the absorption of alien secondary nodes. This should avoid the potential utilization disaster but access performance will be degraded as the

fraction of data that is primary data, and hence only a single disk access from the root, will be reduced.

Let us illustrate how key space gaps might arise with an example. Suppose that the keys of a file are the names of employees and that these names are encoded in a conventional binary code, e.g. ASCII or EBCDIC. In either of these codes, many bit patterns exist, interspersed between the alphabetic characters, which would never occur in the collection of employee names, and hence would lead to key space gaps. The result for DL-trees would be a large number of primary node pages what would be forever empty of primary data, leading to greatly reduced search performance. Fortunately, with some knowledge of the nature of the keys, and their encoding, key space gaps can almost always be removed.

In our example, simply encoding the alphabetic characters as binary numbers between 0 and 31, i.e., as five bit binary numbers, removes most of the gaps. The 26 alphabetic characters, the blank character, and perhaps comma and period, consume 29 out of the 32 bit patterns. There are still some key space gaps but occassional empty pages can be tolerated. The letters and other characters should be spread among the bit patterns so as to minimize split factor in an attempt to equalize key occupancy among the pages of potential primary nodes. No matter how this is done, however, the result, once primary node size is 32, is for there to be some empty pages. At this point each page will contain entries proportional to the frequency of the letter whose entries it is holding. Nonetheless, this is a substantial improvement over ASCII or EBCDIC encoding of the alphabet.

It is possible to entirely remove key space gaps, assuming that one knows where they occur. The characters (digits, bytes, bits) in any key can be interpreted as digits of a mixed radix number. The radix is simply the number of possible characters in any give position of the key. That number can then be converted to a binary number between zero and some maximum value. This range of binary numbers will have no key space gaps though the maximum value might not be of the form 2^k-1. This is not very important since primary nodes need not be provided for the non-existent key values between the maximum key value and the nearest k such that $max\_value \leq 2^k-1$. While mixed radix conversion is feasible, we suggest below a method that usually produces better results and is also more efficient.

Key Compression

It is possible to find a much better encoding than the ones suggested above by using a variable length encoding of the letters. Order preserving key compression using the Hu-Tucker algorithm, as described in Knuth [5] can be used. It has two advantages. First, the average length of keys is reduced. The first encoding above required 5.0 bits per letter while the Hu-Tucker encoding requires, on average, only 4.2 bits per letter. Second, the split factor of each bit is reduced. This results in the relative frequencies of data entries per page of a multi-page primary node being more uniform and produces better access performance as a higher fraction of the data entries will then be primary entries.

The encoding (and subsequent decoding) of keys can be done very efficiently by means of a lookup in tables with modest numbers of entries. For the Hu-tucker algorithm, operating on EBCDIC characters, a table of 256 entries suffices for the encoding, one entry for each 8 bit byte of the code. For decoding, one needs a table of size

$$2^{\max\ (length(encoded\_letter))}$$

This turns out to be a 256 entry table for the decoding as well. The lengths of the encoded characters must be maintained in these tables as well as the encoded and decoded values. The lengths are used to construct the encoded forms and to do the decoding. Even this extra complexity is trivial to deal with and adds very little to the cost of these operations. The cost of encoding (and decoding) will be a very small fraction of the disk access cost and will not significantly affect performance. Further, decoding will not usually be necessary. Only when answers to a range search are requested, and not a request for a particular key, will decoding be required. Thus, most of the time, only the encoding step will be needed. Further, even for the range search, the decoding can be avoided if unencoded key values are stored with the data.

More elaborate versions of key compression are possible. At the expense of much larger tables, one can encode and decode character pairs or triples, etc. Another possibility is some form of context sensitive compression in which a different encoding is used depending on either the position of a character in the key or by which character or characters precede the current one. While these more complex compression techniques might be more expensive, the result of using one of them might be a significant improvement in the uniformity of key distribution. Further, one would still expect the computation required for compression to be modest compared with the computational cost of a disk access.

Hashing

Hashing serves no useful purpose in file access methods if the keys of a file are already uniformly distributed. It is the uniformity of distribution introduced by hashing that makes the structures used in a hashing access method suitable and which results in

their high performance. Thus, what must be demonstrated with respect to DL-trees is the impact that hashing has on disk access performance. Hashing can then be viewed as an optional step that can be employed if three conditions are met. (i) The original keys are distributed non-uniformly. (ii) Key sequential access is not a requirement. (Remember that hashing destroys this property.) (iii) Search performance of close to one disk access is required.

The key to understanding the implications of hashing on DL-tree performance is to view the primary nodes as the data nodes and the secondary nodes as overflow pages. Considered in this fashion, DL-trees then closely resemble in structure the organization of BEH (bounded index exponential hashing) files[9]. Given an approximately uniform key distribution within a primary node, no page of the node will reference more than a very small number of secondary or overflow pages at the time when doubling should occur. This point should be reached when the number of pages that have overflowed is approximately one half of the pages in the node. The analysis in [9] clearly demonstrates that at this point, a reasonable trade-off between performance and storage utilization exists.

Files with uniform distributions will have a scarcity of virtually empty pages, and hence, we should not be surprised that independently allocated secondary nodes are required. For exponentially hashed files, the amount of independently allocatable space that would be needed to produce a reasonable trade-off between storage utilization and search performance is of the order of 5% or so. Each user could, however, by adjusting the value of the file utilization goal, shift the trade-off in any direction desired.

It is possible to use hashing as an option with DL-trees. It is also possible to use an exponential hash function so as to achieve constant performance as the file grows or shrinks [9,11]. Further, there is no reason to limit the root size of DL-trees to only a single page. Like the bounded index exponential hashing access method (BEH), the larger the root size, the smaller can be the primary nodes. The only essential difference between DL-trees and BEH organized files is in the manner in which the overflow of data entries out of primary pages is handled. In the research report on which this paper is based[10], the performance implications of the DL-tree form of overflow technique are considered. Generally speaking, the BEH method has marginally better performance at the expense of losing some flexibility and hence robustness.

VII. Discussion

We have described DL-trees and attempted to show that files organized as DL-trees have good access performance and utilization, regardless of key distribution. They clearly perform better than B-trees once files are fairly large. Typically, for large files, B-trees require close to three disk accesses to find a data entry. With DL-trees, such search performance is always less than two disk accesses, and may even approach one disk access if the keys are uniformly distributed. File utilization is probably lower in DL-trees than with B-trees, but should be in the acceptable range, due to the storing of secondary nodes in under utilized primary pages. DL-trees out perform DB-trees on both utilization and search performance measures.

In the case of hashed files, where uniformity can be virtually guaranteed, DL-trees perform better than most extendible hashing techniques. DL-trees approach the search performance and utilization of the best methods. Further, due to their ability to handle arbitrary key distributions in an effective way, DL-trees provide insurance against the possible, though unlikely, disaster that could occur if the distribution of hashed keys were skewed.

A significant advantage enjoyed by DL-trees over all competing file organizations is that DL-trees can be used for all files in a data base. Thus, only one file organization need be implemented, debugged, and fine tuned. This universality is a compelling reason for preferring DL-trees.

The largest uncertainty associated with using DL-trees is the global structuring of the tree. In section V, we suggested ways in which DL-trees might be structured initially and re-structured as the files grow. These methods were justified only by plausibility arguments. Analysis and/or simulation may provide reassurance that the methods are truly effective, or suggest other approaches that produce better results, and these could well be the subject of a number of papers. Unlike B-trees, which are fully defined as to the structuring method, DL-trees are not specified completely. This is due to the greater complexity of the structuring and to the increased opportunities for tuning DL-trees to application requirements.

Another interesting research area suggested by the DL-tree organization is that of order preserving key compression. Interestingly enough, this is not due so much to the space saving potential of such compression methods as it is to their property of making the distribution of the transformed keys more uniform. The result of this greater uniformity is that DL-tree search performance will be improved, perhaps substantially. Key transformations, aside from hashing, have not previously had much of an impact on file perform-

ance. Again, many useful papers might result from the investigation of such transformations.

Acknowledgements

The ideas in this paper come from many sources. Digital searching has a long history but was first suggested as a file search method with the papers on extendible hashing [4,6,7]. The use of digital searching in an explicit tree structure for arbitrary distributions of keys was, to our knowledge, first proposed in our own paper on digital B-trees [8] as was the method of node doubling. The notion of slowly converting data nodes with overflow pages into index pages originated with Diehr [3]. The ideas of absorbing seondary nodes into newly doubled primary nodes, either to become primary data or to become alien secondary nodes, are, we believe, new with this paper.

Bibliography

1. Bayer, R. and McCreight, E. M. Organization and maintenance of large ordered indices. Acta Informatica 1,3(1972), 173-189.
2. Bayer, R. and Unterauer, K. Prefix B-trees. ACM Trans. Database Syst. 2,1(March, 1977), 11-26.
3. Diehr, G. Extendible hashing extended. Unpublished manuscript, Graduate School of Business Administration, U. of Washington, Seattle, WA.
4. Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. Extendible hashing - a fast access method for dynamic files. ACM Trans. Database Syst. 4,3 (Sept. 1979), 315-344.
5. Knuth, D. *The Art of Computer Programming, vol. 1, Fundamental Algorithms.* Addison-Wesley Pub. Co., Reading, Mass. (1973).
6. Larson, P. Dynamic hashing. BIT 18 (1978), 184-201.
7. Litwin, W. Linear virtual hashing: a new tool for files and tables implementation. Proc. 6th Int'l Conf. on Very Large Data Bases, Montreal, 1980.
8. Lomet, D. Digital B-trees. Proc. 7th Conf. on Very Large Data Bases, Cannes, France, 1981, 333-343.
9. Lomet, D. Bounded index exponential hashing. ACM Trans. Database Syst. 8,1 (March 1983), 136-165.
10. Lomet, D. A high performance, universal, key associative access method. IBM Research Report RC9638, Yorktown Heights, New York (Oct. 1982)
11. Martin, G. Spiral storage: incrementally augmentable hash addressed storage. Theory of Computation Rpt. 27, U. of Warwick, Coventry, England, March 1979.

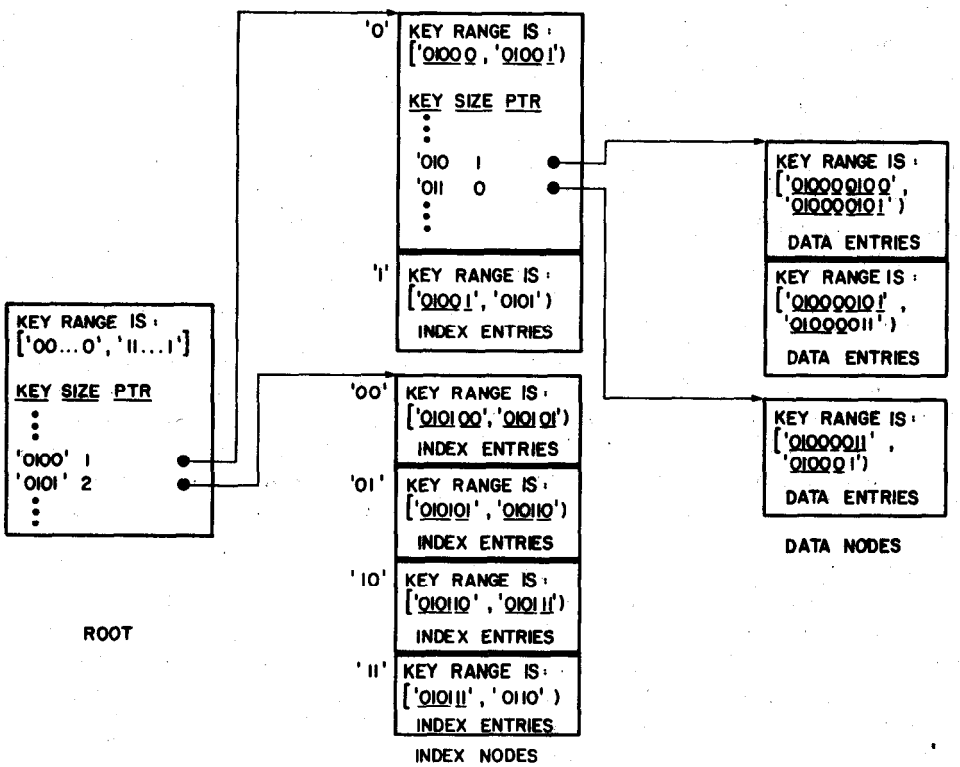

Figure 1: A schematic illustration of a fragment of a DB-tree.

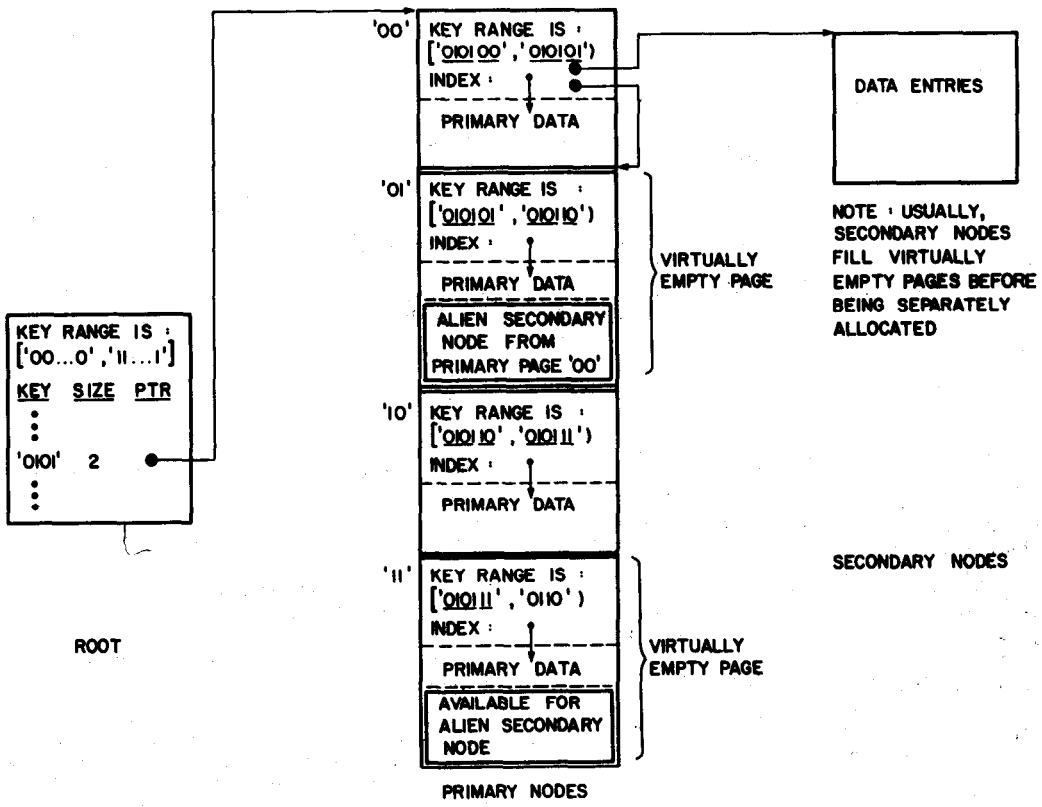

Figure 2: A schematic illustration of a fragment of a DL-tree.