A Multidimensional Digital Hashing Scheme for Files With Composite Keys

Ekow J. Otoo

School of Computer Science
Carleton University
Ottawa, Canada

Abstract

A dynamic hashing method is presented for structuring files with multiple attribute keys The method is essentially the multidimensional analogue of linear hashing developed by Litwin and Larson Given a record of d attribute keys, the scheme called multidimensional digital hashing, applies the linear hashing technique independently to each of the attributes to derive d integer values These values form a d-tuple coordinate address of the home page of the record. A function, equivalent to the element allocation function of a d-dimensional extendible array of linear varying order and computable in time O(d), is used to map the d-tuple page address into a linear address space Algorithms for insertions, deletions and the processing of partial-match and range queries are presented

1. Introduction

Given a file of records where each record is uniquely identified by a d-attribute key $K = (k_1, k_2, \ldots, k_d)$, we address the problem of a direct access organization for the file under operations of insertions, deletions and query processing The types of queries considered are exact-match, partial-match and range queries. An exact-match query requests the retrieval of a record whose key values match some d specified values A partial-match query specifies values for some subset Q (where $Q \subseteq \{1, 2, \ldots d\}$), of the components of the key and requests the retrieval of all records that match the specified components A range query is similar to an exact-match query except that a range of values of the form $[a_j, b_j]$, is specified for $j \in Q$ The special case where $|Q| \leq d$ is referred to as a partial-range query We assume that the file is sufficiently large that it is resident on secondary storage and the unit of data access is a page

The obvious solution to the problem is a dynamic multidimensional order preserving hashing scheme Unfortunately there is no known multidimensional hashing scheme that presents a satisfactory solution to the above probem for all data distributions One difficulty is that if the natural order of each attribute field is to be preserved, the structure becomes dependent on the the distribution of the keys in the key space. Direct access file structures that efficiently process partial-match queries under high storage utilization sacrifice the efficiency to process range queries Some of the early proposed file structures for the problem are Partial-Match Retrieval methods [1, 18], Multiple Key Hashing [2,20], Interpolation-Based Index [3], Dynamic Multipaging [10,13,14], the Grid-File [12], Multidimensional Extendible Hashing [15], Multidimensional Linear Hashing [17] and the EXCELL Method [22].

These earlier methods may be classified as either static or dynamic We term a file organization as static if it takes into consideration knowledge of the file size to achieve a good design but whose performance eventually deteriorates with insertions and deletions. A file organization is dynamic (in the longitudinal sense) if its storage space grows and shrinks in consonance with the number of records in the file without performance deterioration The former type is exemplified by the approach in [1, 2, 20] . The latter type is illustrated by the design scheme in [3,15,17,22]

which we refer to as dynamic multidimensional organizations The method we propose falls in this latter class

The dynamic multidimensional hashing schemes are essentially different approaches to designing d-dimensional equivalents of two dynamic hashing schemes. linear hashing [6,7,8,9] and extendible hashing [4] Two design philosophies have been generally applied In the first approach, each key component k_j is considered as a string of binary digits which are then interlaced (or shuffled) to form a singly key K' The key K' is then used in either the linear hashing scheme or extendible hashing The use of shuffled bits in linear hashing is illustrated by the interpolation-based index method of Burkhard [3] The EXCELL method of Tamminen [22], illustrates the application of shuffled bits in Extendible hashing

In the second approach, each component value of the key is considered independently as participating in separate single key dynamic hashing scheme The integer index values generated form a d-tuple coordinate address of the bucket into which the record is assigned or looked up Such methods are illustrated by the structures in [12,15,17] Incidentally this approach requires a function for mapping the d-tuple addresses onto a linear consecutive address space The problem reduces essentially to that of defining an element allocation function for an extendible array that varies in some prescribed manner A detailed discussion on realizing extendible arrays is presented in [19] The multidimensional digital hashing we present adopts this approach In particular, the scheme we present has very close similarity with the scheme proposed by Ouksel and Scheuermann [12] Our design differs from theirs in the following respect Our scheme expands by groups of pages called a segment, while in [12] the file expands by one page a time In [12], the overflows are organized in a separate overflow while our scheme retains overflow records in the primary area The mapping function in [12] is computable in time O(log n) for a file of n pages The mapping function in our scheme is computable in time O(d) where d is the dimensionality of the key space

The basic idea in our scheme is as follows. Each component key, k_j, of the record to be stored is encoded into a string of binary digits called a pseudo-key component k'_j. Each k'_j value is then used in an independent linear hashing scheme, one on each dimension. The result is that we derive d integers which are formed into a d-tuple address of the primary page into which the record is stored Let the range of index value for each dimension be from 0 to $m_j - 1$, for $j = 1, 2, \ldots, d$, and suppose the linear page addresses are $\{0, 1, 2, \quad, n\}$, where $n = (\prod_{i=1}^{d} m_j) - 1$. For simplicity assume that $m_1 = m_2 = \ldots = n^{1/d}$. We allow the file to expand and contract by adding and deleting $(n^{1-1/d})$ pages at a time Rather than associating each primary page with a chain of overflow pages, the overflows are retained in underfilled primary pages, therefore giving us one single file to contend with

The multidimensional digital hashing method is similar to the design approach in [12, 17] in the sense that each key component is independently considered and the storage space expands and contracts with the size of the file Further, each of this scheme induces a space partitioning of the key space into rectangular cells The distinctive features of our method from the earlier proposed methods are

.

1 The function that maps the d-tuple page address to linear address space is computable in time O(d), which is independent of the size of the file

2 Instead of separate overflow area, overflow records are stored in the main file

3 The method can be tuned for either optimal partial-match or range searching

4 Let H_α be a predefined upper bound on the storage utilization α, defined as $\alpha = N/bn$, where N is the number of records in the file and b is the page capacity Then α is always close to H_α.

The outline of the rest of this paper is as follows. We present a summary of the concepts of linear hashing in the next section and introduce a variant of it which is appropriate for the multidimensional digital hashing scheme In section 3, we present the basic multidimensional hashing scheme and define the desired mapping from a d-coordinate

address space to a linear address space. The details of insertions, overflow handling and deletions algorithms are discussed in section 4. Section 5 addresses the problem of using the scheme for partial-match and range query processing. Some simulation results on the performance of the scheme are presented in section 6. We conclude in section 7, and compare the scheme with similar proposed data organization

2. Linear Hashing

2.1 Basic Concept.

The method of linear hashing (see [7,11] for details) involves a set of m_0, initial pages addressed logically from 0 to $m_0 - 1$, and a hash function g_0, which maps a key, K, of a record into $\{0, 1, \ldots, m_0 - 1\}$. Let the number of records stored in a page be b. Suppose after a sequence of insertions, a collision occurs, i.e., a page into which a record is to be inserted is found to be full. The technique applied is to expand the primary storage space by adding one page, in this case page m_0, and then split the records of some designated page so that about half of these are moved to the new allocated page. A pointer sp keeps track of which page is the next one to be split.

Suppose a collision is caused by a record r in page p The page sp is split irrespective of the page where the collision occurs, and r is inserted as an overflow record except perhaps if the splitting allows r to be stored in a primary page. The dynamic variation of the storage space requires the use of a set of dynamically created hash function, $g_0, g_1, g_2, \ldots, g_h$ called the split functions. The pointer sp is made to synchronize with the address of the new page to be created so that the keys in sp either get rehashed into sp or into the newly created page. The split functions have the characteristic that for any key K :

$$g_h : K \rightarrow \{0, 1, 2, \ldots, 2^h m_0 - 1\}, \qquad (i)$$

$$g_h(K) = \begin{cases} g_{h-1}(K), \\ g_{h-1}(K) + 2^{h-1} m_0 \end{cases} \qquad (ii)$$

The parameter h is called the file level and indicates the degree to which the file has been expanded from its initial allocated space

A specific design may be illustrated with the use of division hashing as follows. The initial values of sp, h, and m (the number of pages of the file) are respectively set to 0, 0, and m_0, and the split function is defined as

$$g(K, h, m) = \begin{cases} K \bmod 2^h m_0; & (a) \\ K \bmod 2^{h-1} m_0. & (b) \end{cases}$$

where the expression (b) is used if the address returned by (a) is greater than m The variables h and m are parameters of the hash function The split pointer sp, is updated by setting $sp \leftarrow (sp + 1) \bmod 2^{h-1} m_0$. Splitting the page designated by sp implies the extension of the file by adding the page $sp + 2^{h-1} m_0$ The file level h, is increased by 1 each time page 0 is to be split. Other split functions are defined in [9]. Instead of page splitting being carried out one at a time, Larson presents a variation of the basic linear hashing method called "Linear hashing with partial expansion" in [6], in which a group of designated pages are split at each expansion step.

In general some load control is enforced by specifying a lower and upper bound on the load factor α. Let these be denoted by L_α and H_α respectively. Then load control requires that the condition $L_\alpha \leq \alpha \leq H_\alpha$ be satisfied always. Other techniques for implementing linear hashing are discussed in [5, 7, 8, 11, 21].

2.2 Order Preserving Linear Hashing (OPLH).

The ideas of the preceding subsection can be easily modified to obtain a "weak" order preserving hashing By "weak order preserving" we mean that the pages can be retrieved in some order, such that if page p_i precedes p_j in the order, then for any two keys $K_i \in p_i$ and $K_j \in p_j$ we have $K_i < K_j$. We illustrate first, a method in which the preserved order is not necessarily the natural order of the keys

The technique is to apply the idea of pseudo-key generation proposed in extendible hashing [4]. We assume the existence of a transformation ψ that generates a w-bit binary value K' for each key K, i e., $K' = \psi(K)$. The value K' is called the pseudo-key. Let K' be represented by the binary sequence $\beta_1, \beta_2, \ldots, \beta_w$, and let m be the number of allocated pages addressed from 0 to m-1 where $2^{h-1} < m \leq 2^h$. Then we may define the split function, ϕ,

which takes on parameters K', h, and m as

function $\phi(k', h, m)$;
begin
  if h = 0 **then** $\phi \leftarrow 0$;
  else begin $\phi \leftarrow \sum_{r=1}^{h} \beta_r 2^{r-1}$
    if $\phi \geq m$ **then** $\phi \leftarrow \sum_{r=1}^{h-1} \beta_r 2^{r-1}$;
    end;
end;

The evaluation of the function ϕ is equivalent to taking the first h prefix bits of the pseudo-key, reversing the right to left ordering, and interpreting the result as a binary integer. By using the function ϕ in place of the split function g_h, we have a linear hashing function which preserves the order of the pseudo-keys in the sense explained earlier

To illustrate the idea of the order preserving linear hashing, we consider the insertions of the pseudo-keys shown in the Figure 2 1 in their order of occurrence. No load control is assumed hence we shall split the page sp whenever a collision occurs We use the term "key" generally, to mean a record uniquely identified by the specified key. Let the primary and overflow page capacity be denoted by b and b' respectively. In the example b = b' = 2. Figure 2 2 shows the configuration of the pages after inserting the first 7 keys The parameters h, sp, and m are 3, 1 and 5 respectively. The next pseudo-key to be inserted is k'_8 = '01001' This hashes to page 2 which is full. The key is therefore inserted in an overflow page, and the page 1 specified by sp, is split This split requires that we extend the file by the addition of page 5, and the pointer sp is advanced to point to page 2. The result of the split operation gives the new configuration of the pages shown in Figure 2.3 The number of pages now becomes 6. The rest of the keys are inserted with no further collisions and Figure 2.4 shows the contents of the final page configuration

Some properties of the hash function ϕ used in this manner are the following .

1 Let m be the number of pages in the file. Then to retrieve the keys in pseudo-key order, (i e , if page p_i is retrieved before p_j then for every $K'_i \in p_i$ and $K'_j \in p_j$ we have $K'_i < K'_j$), we simply retrieve the pages in the order $p_{i_0}, p_{i_1}, p_{i_2} \cdots, p_{i_{m-1}}$ where $i_j = \phi(\chi_j, h, m)$, and χ_j is the h-bit binary representation of j.

2 The pages can be ordered in time $O(m)$ so that consecutive page retrieval, except for overflow pages, corresponds to the weak ordered retrieval of the pseudo-keys.

3. The use of an order preserving pseudo-key encoding function ψ, in the sense that if $K_i < K_j$ then $K'_i = \psi(K_i) < K'_j = \psi(K_j)$, gives then a dynamic hashing scheme in which the keys can be retrieved in their natural order.

We depart from consideration of single attribute keys and address the design of a multidimensional equivalent of the latter variant of linear hashing.

3. Multidimensional Digital Hashing (MDH)

3.1. The Basic Organisation

Consider now that each key K is composed of d values, i e , $K = (k_1, k_2, \ldots, k_d)$. We encode each component, k_j, into a binary value k'_j of w_j bits using a transformation ψ_j. The value $k'_j = \psi_j(k_j)$, forms the j^{th} component of the pseudo-key K'. Each component k'_j is used in a separate order preserving linear hashing as described in the preceding section.

The scheme can be conceived as defining a d-dimensional address space which is partitioned linearly into $m_1 \times m_2 \times \quad m_d$ cells Each cell corresponds to a page addressed by a d-tuple integer coordinate address $(i_1, \ldots i_d)$ If n denotes the number of pages addressed then $n = \prod_{j=1}^{d} m_j$ Figure 3 1 shows the schematic address space of a digital hashing scheme In this respect it is similar to the grid-file design of Nievergelt et al [12]. Call the set of pages having a common index of any dimension a segment. The MDH scheme expands and shrinks by adding and deleting a segment of pages at a time

Since each pseudo-key component is used in a separate hashing scheme, we need to specify d of each parameter of the OPLH scheme We therefore have the following :

h_j · The file level for the j^{th} dimension We shall refer to this as the index level of dimension j.

i→	1	2	3	4	5	6	7	8	9	10
$k_i' = \psi(k_i)$	01101	01010	11100	11101	01110	10100	10001	01001	00010	00100

Figure 2.1 : Table of single valued pseudo-keys

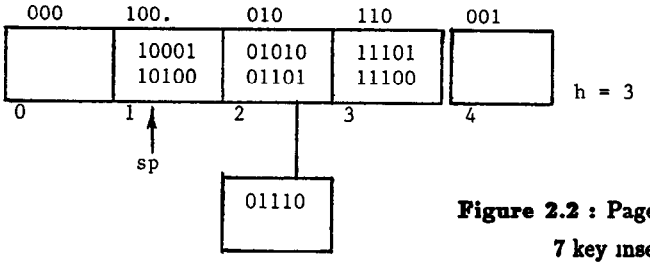

Figure 2.2 : Page contents after first 7 key insertions

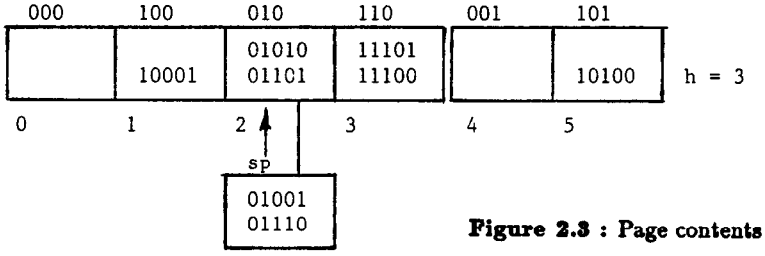

Figure 2.3 : Page contents after splitting page 1

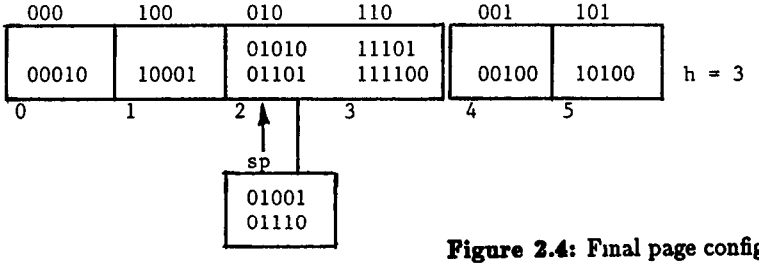

Figure 2.4: Final page configuration after all 10 key insertions.

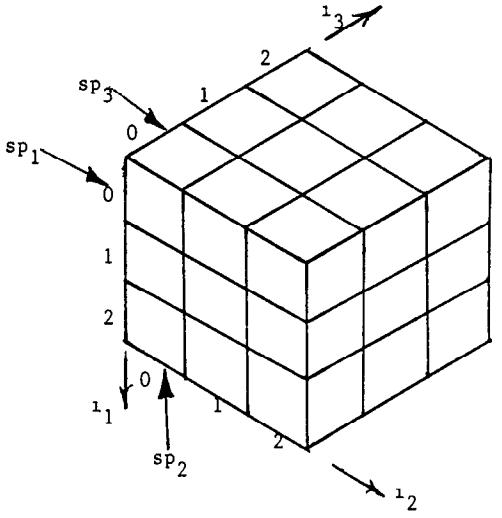

Figure 3.1 : The address space of the multidimensional digital hashing.

m_j The number of possible index values of dimension j where $2^{h_j-1} < m_j \leq 2^{h_j}$. values of the dimension j

sp_j . The pointer specifying the index of the segment of pages to split when dimension j is extended The pointer is advanced after a segment split by assigning $sp_j \leftarrow (sp_j + 1) \ mod \ 2^{h_j-1}$.

Two other parameters are required to indicate when to expand or contract the file, and which dimension is to be extended or decreased We denote the dimension to be extended next by y The value of y varies cyclically and skips over any dimensions that have exhausted the use of the bits of the pseudo-keys A parameter H_α specifies a threshold on the load factor α The file is expanded, if a collision occurs and the value of the load factor exceeds the threshold, by extending the dimension y. The load factor is computed as $\alpha = N/(b * n)$ where N is the number of records in the file, b is the page capacity and n is the number of pages of the file

At each expansion step, we adjoin the segment of pages addressed by $\langle i_1, \ , m_y, \ ., i_d \rangle$, where $i_j = 0, 1, \ ., m_j - 1$, $j = 1, \ , d$, $j \neq y$ Each page in the segment implied by sp_y is split The addresses of these pages are given by $\langle i_1, \ . \ ., sp_y, \ . \ , i_d \rangle$, where $i_j = 0, 1, \ . \ ., m_{j_1} - 1$, and $j \neq y$ The value of m_y is always related to sp_y by the expression $m_y = sp_y + 2^{h_y-1}$

3.2. The Mapping function for MDH.

Given the d-tuple address $\langle i_1, \ , i_d \rangle$ of a page, we desire a mapping function $\Re_\ell$ that maps the integer coordinates $\langle i_1, \ , i_d \rangle$ one to one onto the set of consecutive addresses $\{0, 1, 2, \ . \ ., n-1\}$ corresponding to the linear address space An extension of the index range of dimension y from m_y to $m_y + 1$ induces the extension of the address space by $\prod_{\substack{j=1 \\ j \neq y}}^{d} m_j$ pages

It turns out that the desired mapping function is equivalent to the element allocation function of a uniform extendible array of linear varying order, which we abbreviate as UXAL Such an array is formally presented in [13]. We only state its definition here, and give the required element allocation function

Definition 3.1. *Let I denote the set of non-negative integers, and let $U_j \in I$, be some predefined limit on $u_j \in I$, for $0 \leq u_j \leq U_j$, $j = 1, \ , d$, and $d > 0$ A d-dimensional rectangular array $A[0 \ \ u_1, 0 \ \ u_2, \ \ , 0 : u_d]$ is said to be uniform extendible and of linear varying order iff*

1 *for any two indices j, k, where $1 \leq j, k \leq d$, we have $u_j = u_k$ whenever $u_j \leq U_j$ and $u_k \leq U_k$*

2 *u_j varies by a constant size ℓ, from 0 to U_j, for $j = 1, \ , d$*

For our purpose it is sufficient to consider only the case $\ell = 1$ Consider a UXAL $A[0 \ \ u_1, \ \ , 0 \ . \ u_d]$ Suppose we extend the index range of dimension j from u_j to $u_j + 1$ This implies that we add (d-1)-dimensional subarray implied by the index $u_j + 1$ If we consider that the elements within this subarray block are allocated in column major order, then we can address every element of the subarray, once we know the starting address of the first element of this adjoined subarray, i e , the address of the element $\langle 0, \ ., u_j+1, \ \ , 0 \rangle$ This is the basis of the element allocation function stated in the Proposition 3 1 below The uniformity condition requires that at each expansion step, we

increase every index range that has not reached the maximum index value by one. At the expansion step then, we adjoin subarray blocks implied by $u_1, u_2, \ldots, u_d$, in that order, skipping over any dimensions that have attained their maximum values. That is we skip any dimension t for which $u_t = U_t$. The desired mapping function is stated below.

Proposition 3.1. *Suppose $A[0 : u_1, \ldots, 0 : u_d]$ denotes a uniform extendible array of linear varying order, where $0 \leq u_j \leq U_j$, and U_j specifies the maximum index value attainable for dimension j. Let the elements of A be allocated in consecutive storage locations such that at each expansion step, the blocks implied by $u_1, u_2, \ldots, u_d$ are adjoined in that order with exclusion of those dimensions j for which $u_j = U_j$. Within each subarray block let the elements be allocated in column major order. Then each coordinate index $(i_1, \ldots, i_d)$ is mapped one-to-one onto $\{0, 1, \ldots, n-1\}$, where $n = \prod_{j=1}^{d} (u_j + 1)$, by the mapping function $\mathcal{R}_\ell$ which is defined as*

$$\mathcal{R}_\ell(i_1, \ldots, i_d) = \prod_{j=1}^{d} J_j + \sum_{\substack{j=1 \\ j \neq z}}^{d} c_j * i_j, \qquad (3.1)$$

where z = highest subscript such that $i_z = \max_j(i_j)$,

$$J_j = \begin{cases} \min(i_z + 1, U_j + 1), & \text{if } j < z, \\ \min(i_z, U_j + 1) & \text{otherwise,} \end{cases}$$

and $c_j = \prod_{\substack{r=1 \\ r \neq z}}^{j-1} J_r$

The proof of Proposition 3.1 is given in [13]. A schematic storage layout for the elements of a 2-dimensional extendible array A[0 3,0 4] is shown in Figure 3.2. Note that the first dimension has attained its maximum size while the second can still be expanded.

The allocation function $\mathcal{R}_\ell$, is efficient since storage utilization is 100% in the sense that N elements utilize N locations and element address is computable in time $O(d)$. The function $\mathcal{R}_\ell$ defines our desired mapping function for computing the linear page addresses from the d-tuple coordinate addresses. The equivalence is established by recognizing that m_j corresponds to u_j, $2^{w_j - 1}$ corresponds to U_j, and the cyclic order of expansions in MDH is the same order in which the dimensions of the UXAL are extended.

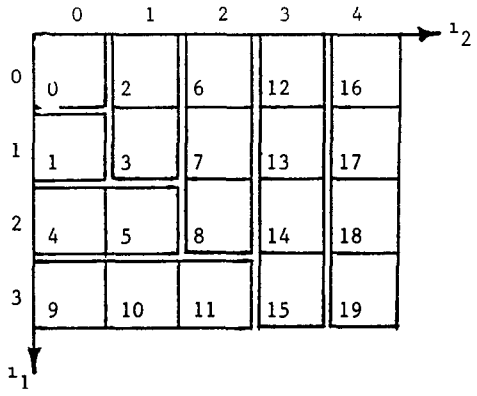

Figure 3.2 : The schematic storage layout of a uniform extendible array of linear varying order A[0 3, 0 4]

Insertions, Overflow Organisation and Deletions

4.1 Insertions.

The insertion of a record with key $K = (k_1, \ldots, k_d)$, is carried out as follows. For each key component k_j, we compute the index $i_j = \phi(\psi(k_j), h_j, m_j)$ and generate the d-tuple coordinate index $(i_1, i_2, \ldots, i_d)$. The address p of the page to insert the record is given by $p = \mathcal{R}_\ell(i_1, \ldots, i_d)$. If there is room in the page p, the record is inserted. However if an overflow occurs and expanding the file will not violate the constraint on the load factor we expand the file, otherwise the record is stored as an overflow record of page p. After a file expansion, a second attempt is made to insert the record in a primary page. If it fails, we insert it as an overflow record.

Overflows may be handled by the separate chaining method. However we choose to organize overflows using an open addressing method so that only a single file is manipulated. Let n be the number of pages in the file. Suppose each page can contain b records. Whenever a collision occurs, we compute the load factor $\alpha = (N + 1)/(n * b)$, where N is the number of records already in the file. If $\alpha > H_\alpha$, we expand the file by extending the index range of dimension y by 1.

The expansion of the file along this dimensions is carried out as follows. First we add the pages whose

coordinate addresses are $\langle i_1, \quad , m_y, \quad , i_d \rangle$, for $i_j = 0, 1, . \quad , m_j - 1, j = 1, \quad , d, j \neq y$, and then set $m_y \leftarrow m_y + 1$ If $sp_y = 0$ we set $h_y \leftarrow h_y + 1$ All records of the pages addressed by the d-tuple coordinate address $\langle i_1, \quad , sp_y, \quad , i_d \rangle$ for $i_j = 0, 1, . \, , m_j - 1, j = 1, \quad ., d, j \neq y$, are rehashed Finally we set $sp_y \leftarrow (sp_y + 1) \ mod \ 2^{h_y - 1}$ and $y \leftarrow (y \ mod \ d) + 1$ The page where the collision occurs may not necessarily be split during this expansion phase and even if it does, it could still be full The offending record must now be stored as an overflow record

Overflow Organization

The overflows are organized by an open addressing method which scans the pages in a shell-by-shell manner Resolving overflows by open addressing requires specifying a rule for the probe sequence, i e., the sequence of pages which must be inspected whenever an overflow record is to be inserted or looked up Given an integer coordinate address $\langle i_1, \quad , i_d \rangle$ of a page p, the probe sequence we define is a shell-by-shell traversal of the pages enclosing p We define first the concept of the k^{th} shell of a page p with coordinate address $\langle i_1, \quad , i_d \rangle$

Definition 4.1. *The k^{th} shell of a page p with coordinate address $\langle i_1 \quad , i_d \rangle$ is the set of pages having the addresses $\langle i'_1, \quad , i'_d \rangle$ such that there exist at least one subscript j, $1 \leq j \leq d$, for which $|i'_j - i_j| = k$*

The notation "$|x|$" denotes the absolute value of x The maximum number of pages in the k^{th} shell is $(2k+1)^d - (2k-1)^d$ The shell-by-shell open addressing method involves inspecting all valid pages of the first shell, followed by all valid pages of the second shell and so on. We need to inspect each page of a shell only once An algorithm for the shell-by-shell traversal of the pages enclosing a given page is fully described in [16] During an insertion of an overflow record, we terminate a search the first time we encounter an underfilled page Unlike open addressing in a static table, we cannot terminate a search during a look up when the first underfilled page is found in the probe sequence since after an insertion, the segment split could have created an underfilled page along the path traced by the probe sequence The problem is resolved by maintaining the count of the number of records addressed to a primary page in the primary page itself During a look up of a record addressed to a page p say, we continue the search until we have located as many records belonging to page p as indicated by the count value

Call the record stored in a page p, but belonging to some other page q, a foreign record If a record r hashes to a page which is found to be full, but a foreign record r' exists, then r' is replaced by r and r' gets allocated in a page in the probe sequence path of its primary page Each time a page splits all overflow records belonging to the page are regrouped Any subsequent overflows are eventually stored in pages within the immediate logical neighbourhood of the primary page. The numbr of page accesses required to gather all overflows of a particular page is equal to the number of page accesses for an unsuccessful search of a record addressed to that page An overflow record stored in page other than its primary page is not chained to its primary page It is always identified by the fact that it hashes into an address different from the address at which it currently resides By storing overflows in this way, the pages accessed during processing of partial-match, and range queries are most likely to contain foreign records that satisfy the query as well and as such reduce the number of physical page accesses in a paging environment

4.3 An Example

The MDH scheme is illustrated by tracing the expansion of the file as the 2-dimensional pseudo-keys, shown in the Table 4 1, are inserted The first value of each key is assumed to be encoded into 4 bits, while the second is encoded into 3 bits In the example the page capacity b = 4, and the value of $H_\alpha = 0\ 75$ The file is initialized with 4 pages as in Figure 4 1a. The parameters of the scheme are set initially as follows $h_1 = h_2 = 1$, $sp_1 = sp_2 = 0$, $m_1 = m_2 = 2$, y = 1 and n = 4

The first 5 keys are inserted with no collisions The first collision occurs in the attempt to insert the key $k'_6 = (0010, 100)$ into page 2, and since $\alpha = 6/8$, is not greater than 0 75, it is inserted as an overflow key in page 0 The next collision occurs in inserting the key $k'_8 = (0111, 100)$

Figure 4.1 : A set of 2-dimensional pseudo-keys $K' = (k'_1, k'_2)$

$K' = (k'_1, k'_2)$

$K'_1 = (1110, 010)$
$K'_2 = (1011, 101)$
$K'_3 = (0101, 101)$
$K'_4 = (1100, 101)$
$K'_5 = (0001, 111)$
$K'_6 = (0010, 100)$
$K'_7 = (0100, 010)$
$K'_8 = (0111, 100)$
$K'_9 = (0001, 001)$
$K'_{10} = (0110, 010)$
$K'_{11} = (1000, 110)$
$K'_{12} = (0111, 001)$
$K'_{13} = (0011, 000)$
$K'_{14} = (1100, 000)$
$K'_{15} = (1001, 011)$
$K'_{16} = (1101, 001)$
$K'_{17} = (0011, 100)$
$K'_{18} = (1110, 011)$
$K'_{19} = (0011, 011)$
$K'_{20} = (0001, 010)$
$K'_{21} = (1001, 001)$
$K'_{22} = (0110, 111)$

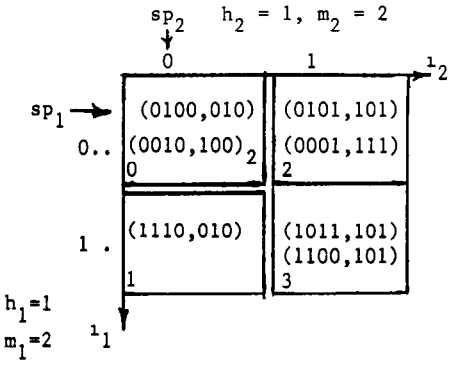

Figure 4.1a : Page contents after first 7 key insertions.

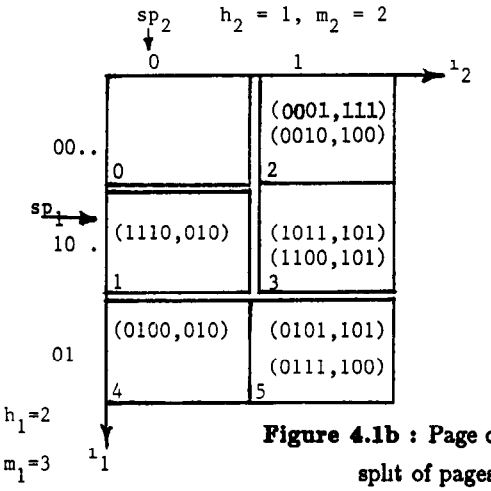

Figure 4.1b : Page contents after the split of pages 0 and 2

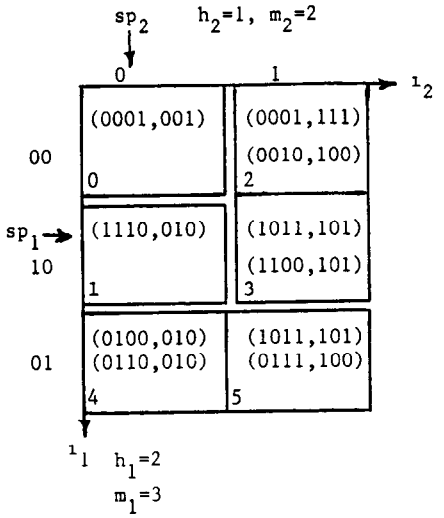

Figure 4.2a : Page contents before splitting

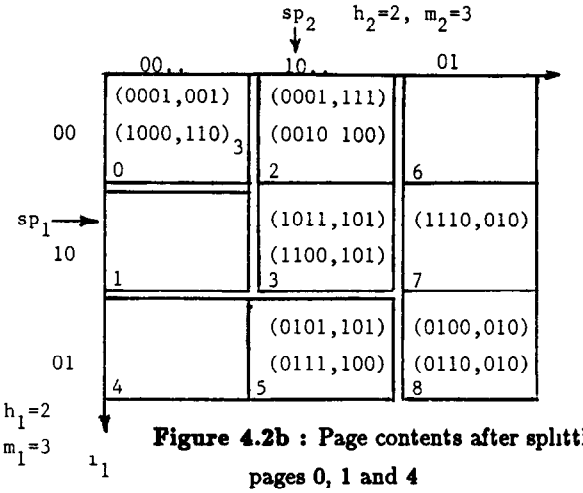

Figure 4.2b : Page contents after splitting pages 0, 1 and 4

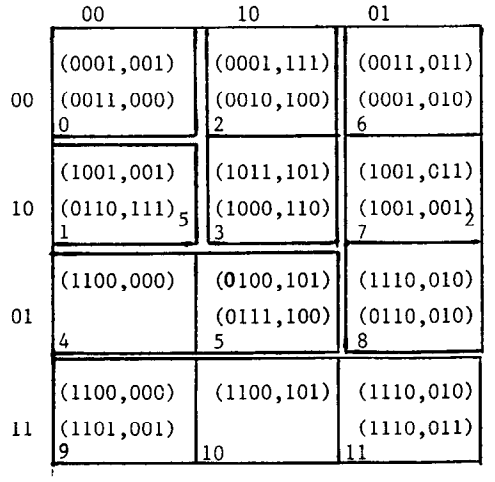

Figure 4.3a : The final page configuration after all 22 key insertions

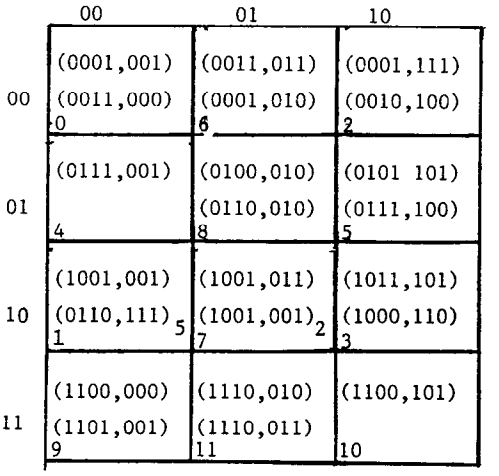

Figure 4.3b : The storage space representation showing the logical adjacency of the pages

into page 2. But now the load factor $\alpha > 0.75$, therefore we split the pages of the segment $sp_y = sp_1$. We set $h_1 = 2, m_1 = 3$, allocate the new pages 4 and 5, and rehash the keys of the pages 0 and 2. After this, we set $sp_y = 1$, and $y = 2$, so that the next time we expand the file, the dimension 2 will be extended. The result of this expansion is shown in Figure 4.1b

Subsequent insertions cause no overflows until we try to insert the key $k'_{11} = (1000, 110)$ Since the load factor exceeds 0 75, we extend the file by adding the pages 6, 7 and 8 and update m_2 to 3, and h_2 to 2 The pages of the segment sp_2 that are split are 0, 1 and 4 The value of sp_2 is set to 1, and y is set to 1 The Figure 4 2a shows the schematic layout of the pages just before the expansion and Figure 4 2b shows the pages after the expansion Continuing the insertion procedure, we obtain the final configuration of the pages in Figure 4 3a The key $k'_{21} = (1001, 001)$ of page 2 is stored as a foreign key in page 7 and similarly key $k'_{22} = (0110, 111)$ of page 5 is stored as a foreign key in page 1

Splitting segments of pages at each expansions step is justified for the following reasons

i In creating the n pages of the file, where $n = \prod_j^d m_j$, the same number of page splits occurs whether we split the pages one at a time or one whole segment at a time. The average number of page accesses required to build the file is of the same order.

ii The page splits occur in groups in an analogous manner to linear hashing with partial expansion [7].

Proposition 4.1. *Let d be the dimensionality of a multidimensional digital hashing scheme of n pages. Then the worst case number of page accesses to insert a record is* $O(n^{1-1/d})$, *and the average number of page accesses per record insertion is* $O(1)$.

Proof

Let λ'_s be the average number of page accesses for an unsuccessful search which is a small integer constant for an appropriate chosen H_α and page size b The worst case number of page accesses occurs when a segment is split. The number of pages accessed is $C\lambda'_s n/(\max_j(m_j))$, where the m_j's are the sizes of the respective dimensions, and C is a small integer constant that accounts for the writing and reading of pages during the split But we have that

$$\frac{C\lambda'_s n}{\max_j(m_j)} \leq \frac{C\lambda'_s n}{n^{1/d}} = C\lambda'_s(n^{1-1/d}),$$

$$= O(n^{1-1/d})$$

Hence the worst case number odf page accesses to insert a record $= O(n^{1-1/d})$

Suppose there are n pages of the file When the dimension j is extended from m_j to $m_j + 1$ the number of pages added is n/m_j. The number of page accesses is $C\lambda'_s n/m_j$ Before the next expansion, the number of records inserted is $bH_\alpha n/m_j$. But these insertions must have made $\lambda'_s bH_\alpha n/m_j$ page accesses This process occurs periodically as the file grows, hence the average number of page accesses per record inserion is given by

$$E(\mu) = \frac{C\lambda'_s n/m_j + \lambda'_s bH_\alpha n/m_j}{bH_\alpha n/m_j},$$

$$= \frac{\lambda'_s}{bH_\alpha}(C + bH_\alpha),$$

$$= O(1)$$

for appropriately chosen values of b, and H_α.

4.4 Deletion

Deleting a record with key $K = (k_1, \ldots, k_d)$, involves essentially performing an exact-match search to locate the record and then deleting it to free the space. If the deleted record is from a home page but overflow records belonging to that page exists, we simply replace the deleted record with an overflow record. Of interest is the action taken to decrease the size of the file after a series of deletions.

Suppose m_l implies the segment of pages added during the last file expansion If the deleted record is from a home page and no overflow record exists, we compute the load factor as $\alpha = m_l N/((m_l - 1)bn)$ If $\alpha < H_\alpha$ we shrink the file by deleting the pages of the last segment m_l The deletion is carried out basically as the reverse of the insertion process. The cost of a deletion is, therefore, of the same order as the insertion The worst case deletion cost from a file of n pages in d dimensions is $O(n^{1-1/d})$, while the average deletion cost is $O(1)$.

We conclude this section by stating algorithms for insertions and deletions A record will be referred to simply by its key K We assume the existence of a function Find(K, p) and a procedure Get_Page(p) The function "Find" takes a record of key K, a page address p, and returns the boolean value true or false according to whether K is found in page p or not. The search examines all overflow records of page p The procedure "Get_Page" retrieves the page p into memory A page p is assumed to have a field "p windx' which specifies the number of records written into the page First we specify the algorithm for computing the page address of a record with key K

Algorithm Address(K = $(k_1, \ldots k_d)$),

1 Set $t_j \leftarrow \phi(\psi_j(k_j), h_j, m_j)$ for $j = 1, 2, \ldots,$ d,

2 $p \leftarrow \Re_l(t_1, \quad, t_d)$; Get_Page(p);

3 Return(p),

Algorithm Insert (A record with key K = $(k_1, \ldots, k_d)$);

I1 $p \leftarrow$ Address(K); Get_Page(p);

I2 If Find(K,p) then "write error message" and return, If p windx < b then store K in p and return

I3 If a foreign record with key K_o is in p then interchange records K_o and K.

I4 Set $\alpha \leftarrow (N + 1)/(n * b)$; if $\alpha < H_\alpha$ goto I7. Add the pages addressed by $\langle t_1 \quad, m_y, \quad, t_d, \rangle$, $t_j = 0, 1, \ldots, m_j - 1$, $j \neq y$, Set $m_y \leftarrow m_y + 1$; if $sp_y = 0$ then set $h_y \leftarrow h_y + 1$. Initialize a temporary file F to empty

I5 For each page q addressed by $\langle t_1, \ldots, sp_y, \quad, t_d \rangle$, $t_j = 0, 1, \ldots, m_j - 1, j \neq y$, do the following. Copy keys in q to a temporary location Q Set page q to empty For each record $t \in Q$ and any overflows of q, set $p \leftarrow$Address(t) Get_Page(p) If p windx < b store t in p, else add t to the temporary file F.

I6 Set $sp_y \leftarrow (sp_y + 1)\ mod\ 2^{h_j - 1}$, and $y \leftarrow y\ mod\ d + 1$. For each record t in F, set $p \leftarrow$ Address(t). Follow collision resolution method to insert t

I7 Set $p \leftarrow Address(K)$; Get_Page(p); if p.windx < b, store record K in p, otherwise follow collision resolution method to store K Return.

Algorithm Delete(Given a key $K = (k_1, \ldots, k_d)$;

D1 Set $p \leftarrow$ Address(K); Get_Page(p);

D2 If not Find(K,p) then "print error message" and return; otherwise free space occupied by record K. If deleted record is an overflow record return If deleted record is from primary page p and an overflow record exists, move an overflow record to replace K and return

D3 Set $z \leftarrow y - 1$, if $z < 0$ then $z \leftarrow z + 1$; $N \leftarrow N - 1$, { N is the number of records in the file } Compute $\alpha = m_z N/(bn(m_z - 1))$, if $\alpha > H_\alpha$ return

D4 Set $y \leftarrow z$, if $m_y = 0$ return. Otherwise set $m_y \leftarrow m_y - 1$, $sp_y \leftarrow sp_y - 1$, if $sp_y < 0$, $sp_y \leftarrow sp_y + 2^{h_y - 1}$;

if $sp_y = 0$, then $h_y \leftarrow h_y - 1$

D5 For each page p addressed by $\langle i_1, \ldots, m_y, \quad, i_d\rangle$, $i_j = 0, 1, \quad, m_j - 1, j \neq y$, do the following For each record $t \in q$ compute $p \leftarrow$ Address(t), Get_Page(p), if p windx < b then store record in p else follow overflow method to insert t

D6 Return

5. Query Processing

The operations of retrieving a record or a set of records given full or partial information are easily handled by the MDH scheme We address, specifically, the problem of partial-match queries and show how algorithms for processing such queries are easily extended to cover orthogonal range queries An algorithm for exact-match queries is easily derived by combining the algorithms "Address", "Get_page" and "Find".

Let Q be a subset of $\{1, 2, \ldots, d\}$, i.e , $Q \subseteq \{1, 2, \ldots d\}$, whose cardinality $|Q| = s$. A partial-match query requests the retrieval of all records with key $K = (k_1, k_2, \quad, k_d)$, where k_j is specified only if $j \in Q$. Let the unspecified key components be denoted by '*' Then we have the following algorithm.

Algorithm Partial-Match

, {Given a set of key components k_j, for $j \in Q \subseteq \{1, 2, \ldots, d\}$ and $d - |Q|$ unspecified values '*' Each dimension j, of the organization has an index range $[0, m_j - 1]$, and an index level h_j }

1. Set $k'_j \leftarrow l_j \leftarrow u_j \leftarrow \psi(k_j)$ if $j \in Q$, otherwise set $k'_j \leftarrow l_j \leftarrow$ '000 0', and $u_j \leftarrow$ '111.. 1', for $j = 1, 2, \ldots d$,
2. Set $i_j \leftarrow \phi(k'_j, h_j, m_j)$ for $j = 1, 2, \ldots d$, $p \leftarrow \Re_\ell(i_1, i_2, \quad, i_d)$ and Get_Page(p),
3. For all records t belonging to p do the following If components values of t match the specified values then process(t) Set $j \leftarrow 1$;
4. If j > d return, Increase first $h_j - bits$ of k'_j by 1, if $k'_j > u_j$ then set $k'_j \leftarrow l_j$, $j \leftarrow j + 1$; and repeat 4 Otherwise goto 2

The MDH scheme may be tuned for optimal-partial match design using the method in [1] Suppose the number of bits into which the j^{th} values are to be encoded is w_j and we have $w = \sum_{j=1}^{d} w_j$ We choose w to be of some fixed value (say w = 32) in the design Assuming that P_j is the probability that the j^{th} key value is specified in a query, and is independent of all others, it is shown in [1] that the optimal partial-match is achieved by assigning the w_j's according to

$$w_j = \frac{w - \sum_{k=1}^{d} \log_2(\frac{P_k}{1-P_k})}{k} + \log_2(\frac{P_j}{1-P_j}) \qquad (5\ 1)$$

We need then to select the w_j's according to (5 1) Under this assumption we show that the expected number of page accesses for a partial-match query specifying s out of d key values is $O(n^{1-s/d})$

Proposition 5.1. *Given a multidimensional digital hashing scheme of n pages in d dimensions, the expected number of page accesses for a partial-match query specifying s out of d key values is* $O(n^{1-s/d})$

Proof

A query Q that specifies values for exactly s key components can be conceived of as an s-element subset of $\{1, 2, \quad d\}$ Let the probability that the j^{th} component value is specified in a query Q be P_j, and let this be independent of all other components Let S denote the set of all s-element subsets of $\{1, 2, \ldots, d\}$. Then the probability P_Q that the query Q is specified is

$$P_Q = \frac{\prod_{j \in Q} P_j * \prod_{j \notin Q}(1 - P_j)}{\sum_{Q \in S}(\prod_{j \in Q} P_j * \prod_{j \notin Q}(1 - P_j))} \qquad (5\ 2)$$

Let λ_s denote the expected number of pages retrieved to locate all records of any one page. The expected number of page accesses made for any query specifying s components is

$$\begin{aligned} &\text{E(page accesses} \cdot \text{—Q— = s)} \\ &= \lambda_s \sum_{Q \in S} \left[\frac{\prod_{j \in Q} P_j * \prod_{j \notin Q}(1 - P_j)}{\sum_{Q \in S}(\prod_{j \in Q} P_j * \prod_{j \notin Q}(1 - P_j))} \right] \frac{n}{\prod_{j \in Q} m_j}, \\ &= \frac{\lambda_s n}{\sum_{Q \in S} \prod_{j \in Q}(\frac{P_j}{1-P_j})} \sum_{Q \in S} \left(\prod_{j \in Q} (\frac{P_j}{1-P_j}) \frac{1}{m_j} \right), \qquad (5\ 3). \end{aligned}$$

If all the bits in each dimension are used in creating the n pages, the optimality condition in (5.1) can be shown to be equivalent to

$$\left(\frac{P_j}{1-P_j}\right) \frac{1}{m_j} = \theta \text{ (some constant) for } j = 1, \quad, d, \quad (5\ 4)$$

The relationship (5.4) is derived in [13] Substituting (5 4) in (5 3) we have

$$E(\text{page accesses} \mid |Q| = s) = \frac{\lambda_s n}{\sum_{Q \in S} \theta^s \prod_{j \in Q} m_j} \binom{d}{s} \theta^s,$$
$$\leq \frac{\lambda_s n \binom{d}{s} \theta^s}{\theta^s \binom{d}{s} n^{s/d}},$$
$$= \lambda_s (n^{1-s/d}) = O(n^{1-s/d})$$

The algorithm stated for partial-match query can be easily modified for range and partial range queries. We show this for range queries. We assume that the transformations $\psi_j's$ are order preserving Since ranges of values of the form $[a_j, b_j]$ are specified, we modify the statement (1) to read as follows;

1 Set $k_j' \leftarrow l_j \leftarrow \psi_j(a_j)$, and $u_j \leftarrow \psi_j(b_j)$, for $j = 1, 2, \quad , d$

The rest of the algorithm applies with no modifications The ranges of values specified for each dimension define a hyper-rectangular region in d-space corresponding to the response region Let R be the number of pages that logically cover this region Then the algorithm retrieves only the pages that cover the response region plus the overflow pages storing any records of these pages. The number of page accesses then is $O(R)$.

6. Experimental Results

The multidimensional extendible hashing scheme has been implemented for experimentation. These preliminary simulations are aimed at studying the effect of such parameters as the page size b, the limiting load factor H_α, and the number of dimensions on the performance characteristic in a paged memory envioronment. The following characteristics are observed :

μ · the number of page accesses required for a record insertion,

λ_s the average number of page accesses for a successful search,

λ_s' · the average number of page accesses for an unsuccessful search, and

α : the average load factor

We retain 16 page frames in memory A page fault occurs if a referenced page is not resident in memory and is to be fetched from secondary storage Our unit of access is the physical read/write to secondary storage made We apply the "least recently used" policy for the page replacement algorithm The key values used are such that

a) for any d, the size of each pseudo-key field $w_j = 24/d$

b) each key component is generated as a pseudo-random number integer uniform in $[0, 2^{w_j} - 1]$,

Each run of the simulation consists of inserting N = 10000 keys for a fixed value of d, b and H_α. After every 1000 insertions, the average values of $\mu, \lambda_s, \lambda_s'$, and α are computed as the file grows from N = 0 to 10000. The averages of each of the corresponding recorded statistics is the summary results for each run. These results are computed for H_α = 0 6, 0.7, 0 8, 0 9 ; b = 10, 20, 40 ; and d = 2, 3, 4. The Tables 6 1, 6.2 and 6 3 in the Appendix summarize the results of the experiment.

The results show the average performance of the MDH scheme for a few combinations of b, α, H_α, and d Clearly an increase in the page size gives better performance, i e high load factor and low average time for successful and unsuccessful search. Increasing the number of dimensions, improves the search times, but lowers the average load factor. This is a result of the increase in the number of pages per segment at each expansion step when the number of dimensions is increased.

We have concentrated on studying λ_s and λ_s', in these simulations. Not only do these give a measure of the retrieval efficiency of the scheme but also they appear as the unknown factors in the analytical results derived for other classes of searching. The results suggest that a recommended upper bound on the load factor is between 0.7 and 0 8 with a page size greater than 10.

7. Conclusion

We have presented in this paper a generalization of the basic linear hashing scheme to files with composite keys The extra complexity required for the generalization is a definition of a function for mapping a d-tuple page address onto a linear address space This function has been shown to be equivalent to that for realizing a uniform extendible array of linear varying order

The method, called digital hashing, is proposed in conjunction with an open addressing collision resolution method which favours partial-match and range searching

In this way we achieve a single file version of the MDH scheme in the spirit of the methods in [7, 8, 11]. This by no means excludes it from being used in conjunction with other overflow record organization such as separate chaining

The MDH scheme has all the inherent properties of linear hashing, in that the file grows and shrinks with insertions and deletions Further it can be tuned for either optimal partial-match retrieval or range query processing by defining each encoding transformation ψ_j to be order preserving Algorithms for such query processing are made simple The idea of considering each key component independently makes it comparable with earlier proposed methods such as the grid-file [12] and a multidimensional linear hashing proposed in [17] In comparison with the grid-file organization, the MDH scheme and the grid-file both induce a grid-like partition of the attribute space However the directory growth in the grid-file is exponential and can be very large if the key distribution in the key space is not reasonably uniform. This situation arises when the grid-file is organized for range searching The space in the MDH scheme grows linearly Expensive searches that might result when the data is not reasonably uniform can usually be offset by specifying a lower value of H_a. The grid-file has a guaranteed two disk access to records but has an average storage load factor of 0 7. At this load factor the access efficiency of the MDH scheme is comparable

Compared with the method in [21], our method gives a method of computing the storage mapping function in time $O(d)$ as opposed to $O(log(n))$ for the multidimensional linear hashing given a file of n pages with d attribute keys.

We remark that this method of independent treatment of each component of the key is desirable as a first step towards achieving complete dynamic file organization. Dynamic file organizations currently consider the situation where the performance of the file is to remain relatively stable with insetions and deletions Such files can be considered as dynamic only in the longitudinal sense. A complete dynamic file needs to allow growth of the file both longitudinally and laterally where the number of attributes is allowed to increase as well.

Acknowledgments

The use of the VAX/780 of the department of Systems and Computer Engineering for our simulations is very much appreciated We wish to thank M Atkinson, John Oommen, and Nicola Santoro for their helpful discussions and comments This research is supported in part from NSERC grant No 102A

References

[1] Aho, A V and Ullman, J D. Optimal partial match retrieval when fields are independently specified. *ACM Trans on Database Syst 4, 2 (Sept 1979), 168 - 179*

[2] Bolour, A Optimality properties of multiply key hashing function *J ACM 26,2 (Apr 1979), 41-62*

[3] Burkhard, W A Interpolation based index maintenance *Proc 2nd Int'l Symp on Principles of Database Syst (1983), 76-89*

[4] Fagin, R , Nievergelt, J , Pippenger, N and Strong, H R Extendible hashing a fast access method for dynamic files *ACM Trans on Database Syst 4, 3 (Sept 1979), 315-344*

[5] Kjellberg, P and Zahle, T U Cascade hashing *Proc 10th Int'l Conf on Very Large Databses, Singapore (Aug 1984), 481-492*

[6] Larson, P Linear hashing with partial expansion *Proc 6th Int'l Conf on Very Large Databases, Montreal (1980), 224-232*

[7] Larson, P A single file version of linear hashing with partial expansion *Proc 8th Int'l Conf on Very Large Databases, Mexico City (1982), 300-309*

[8] Larson, P Linear hashing with linear probing *Technical Report No CS-83-38, University of Waterloo, (Jan. 1984)*

[9] Litwin, W Linear hashing a new tool for files and table addressing *Proc 6th Int'l Conf on Very Large Databases Montreal (Oct 1980), 212-223*

[10] Merrett, T H and Otoo, E J Dynamic multipaging a storage structure for large shared data banks *Proc Int'l Conf on Databases Improving Usability and Responsiveness P Scheurermann ed , Academic Press (1982), 237-256*

[11] Mullin, J K Tightly controlled linear hashing without separate overflow storage. *Bit 21 (1981), 390-400.*

[12] Nievergelt, J , Hinterberger, H , Sevcik, K C , The grid file an adaptable symmetric multikey file structure *ACM Trans on Database Syst 9, 1 (Mar 1984), 38-71*

[13] Otoo, E J Low level structures in the implementation of the relation algebra *Ph D Thesis, School of Computer Science, McGill University, (Aug 1983)*

[14] Otoo, E J and Merrett, T H Dynamic multipaging a storage structure for fast associative searching *Technical Report No SCS-TR-56, School of Computer Scienc, Carleton University, (Submitted to Act Informatica)*

[15] Otoo, E J A mapping function for the directory of a multidimensional extendible hashing *Proc 10th Int'l Conf on Very Large Databases, Singapore (Aug 1984), 493-506*

[16] Otoo, E J A method for associative searching using index tries *Technical Report, School of Computer Science, Carleton University, Ottawa,*

[17] Ouksel, M and Scheurmann, P Storage mapping for multidimensional linear hashing *Proc of 2nd Symp on Principles of Database Syst , Atlanta, Georgia (1983), 90-105*

[18] Rivest, R L Partial-match retrieval algorithms. *SIAM J of Comput 5, 1 (1976), 19-50*

[19] Rosenberg, A L Managing storage for extendible arrays *SIAM J Comput 4, 3 (Sept 1975), 287-306*

[20] Rothnie, J B and Lozano, T Attribute-based file organization in paged memory environment *Comm ACM 17, 2 (Feb, 1974), 63-69*

[21] Scholl, M A new file organization based on dynamic hashing *ACM Trans on Database Syst 6, 1 (1981), 194-211*

[22] Tamminen, M The EXCELL method for efficient geometric access to data *Acta Polytechnica Scandinavica, Mathematics and Computer Science Series No 34, Helsinki, (1981)*

Appendix

		load factor α		# page accesses per insertion μ		# page accesses for successful search λ_s		# page accesses for unsuccessful search λ'_s	
	b	min	avg	avg	max	avg	max	avg	max
$H_\alpha = 0.6$	10	0.585	0.590	3.419	4.586	1.389	3.968	0.963	3.794
	20	0.585	0.587	2.470	4.223	1.774	4.212	1.648	4.454
	40	0.555	0.583	1.091	1.317	0.600	0.929	0.2644	1.042
$H_\alpha = 0.7$	10	0.632	0.673	5.311	10.43	4.849	11.145	3.988	8.543
	20	0.609	0.665	1.938	2.575	1.001	2.087	0.658	1.827
	40	0.609	0.673	1.178	2.261	0.967	1.840	0.984	2.121
$H_\alpha = 0.8$	10	0.762	0.790	9.490	14.29	5.415	11.67	5.128	10.26
	20	0.714	0.776	5.243	11.95	4.489	10.95	3.968	9.029
	40	0.714	0.769	2.500	4.053	1.448	2.984	1.742	3.655
$H_\alpha = 0.9$	10	0.877	0.882	15.57	25.50	12.64	24.00	12.36	21.60
	20	0.833	0.866	9.152	19.32	9.990	25.60	7.990	20.50
	40	0.800	0.852	4.540	8.000	3.174	6.758	3.758	6.326

Table 6.1 : Performance for uniform distributed keys for d = 2

	b	load factor α min	load factor α avg	# page accesses per insertion μ avg	# page accesses per insertion μ max	#page accesses for successful search λ_s avg	#page accesses for successful search λ_s max	#page accesses for unsuccessful search λ'_s avg	#page accesses for unsuccessful search λ'_s max
	10	0 533	0 559	2 368	4.161	1 947	4 846	1 152	3 876
$H_\alpha = 0\,6$	20	0 533	0 558	1 536	1 822	0 623	0 948	0 331	0 953
	40	0 533	0 563	1 038	1 450	0 632	0 862	0 452	0 778
	10	0 634	0.659	3 063	5 489	2 803	7 496	1 698	5 680
$H_\alpha = 0\,7$	20	0.625	0 651	2.143	2 743	0 916	1 587	0 796	1 854
	40	0 555	0 633	1 272	2 039	0 888	1 456	0 845	1 648
	10	0 699	0 763	4 566	8 354	4 399	13 88	2 845	9 674
$H_\alpha = 0\,8$	20	0 625	0 737	3 467	4 606	1 572	3 556	1 653	3 576
	40	0.625	0.754	1.898	3.584	1 647	2 844	1 676	3 373
	10	0 800	0 856	8 846	14 93	7 489	20 87	5 687	16 91
$H_\alpha = 0\,9$	20	0 800	0 826	6 758	9 601	3 570	6 271	4 157	6 924
	40	0 740	0 801	3 468	6 987	2 685	5 401	3 457	6 498

Table 6.2: Performance of MDH for uniform distributed keys when d = 3

	b	load factor α min	load factor α avg	# page accesses per insertion μ avg	# page accesses per insertion μ max	#page accesses for successful search λ_s avg	#page accesses for successful search λ_s max	#page accesses for unsuccessful search λ'_s avg	#page accesses for unsuccessful search λ'_s max
	10	0 500	0.543	2.477	3 969	1 534	2 805	1 240	2 474
$H_\alpha = 0\,6$	20	0 468	0 529	1 342	1 964	0 881	1 806	0 479	1 563
	40	0 468	0 525	0 853	1 029	0 456	0 523	0 409	0 520
	10	0 555	0 619	3 510	5 869	2 500	4 153	2 111	4 178
$H_\alpha = 0\,7$	20	0 493	0 608	1 726	2 653	1 232	2 635	0 834	2 364
	40	0 493	0 597	1 030	1 265	0 494	0 640	0 226	0 730
	10	0 625	0 748	5 513	9 487	4 416	10 41	3 672	7 083
$H_\alpha = 0\,8$	20	0 625	0 728	3 022	4 627	2 094	5 586	1 906	4 851
	40	0 555	0 693	1 698	2 357	0 631	1 056	0 738	2 158
	10	0 740	0 813	10 99	18 90	8 260	19 18	8 010	14 98
$H_\alpha = 0\,9$	20	0 720	0 780	5 399	8 751	3 528	7 975	4 136	10 56
	40	0 694	0 763	3 855	5 252	1 186	2 554	2 249	4 993

Table 6.3: Performance of MDH for uniform distributed keys when d = 4