MULTI-DIMENSIONAL DIRECTORY FOR RETRIEVAL ON SECONDARY KEYS

by

Jenn-Hann Liou

Memorandum No. ERL-M503

26 February 1975

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

MULTI-DIMENSIONAL DIRECTORY FOR RETRIEVAL ON SECONDARY KEYS

by

Jenn-Hann Liou

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

ABSTRACT

This paper describes a new technique, called multi-dimensional directory (MDD) method, for record retrieval on secondary keys. The shortcomings of multiple key hashing (MKH) method [6,7] are discussed. Then the MDD method is introduced, whose retrieval performance is expected to be at least as good as the MKH method for static files.[1] Following that, the techniques used in B-tree, which is a special data structure for one-dimensional directory, are applied to the maintenance of an MDD. This makes MDD particularly suitable for dynamic files

Research sponsored by the Naval Electronics Systems Command, Contract N00039-75-C-0034.

[1]A file is dynamic if it is subject to deletion and insertion of records, and its size is subject to growth and shrinkage; otherwise it is static.

1. INTRODUCTION

Automated information systems are requiring more efficient strategies for retrieval on secondary keys. The relationships between objects such as hierarchy and association, which have traditionally been represented by pointers in the hierarchical and network models, are now indicated by secondary keys in the relational model [1,2]. Query languages [3,4] have been proposed to provide the user the freedom of retrieving records on any combination of any keys. Thus efficiency of retrieval on secondary keys has become one of the most important issues in the design of a modern information system.

Quite a few schemes have been introduced to facilitate the retrieval on secondary keys. Among them [5], file inversion, index combination [8], and multiple key hashing [6,7] have drawn the most attention.

Files are almost always organized on their primary keys. They are sorted or hashed on their primary keys. When retrieval on secondary keys is necessary, inverted files or combined indices are created to expedite it. But there are several serious drawbacks in them. (1) They need extra storage spaces that are comparable to the main file. (2) All inverted files and combined indices have to be updated whenever a record is deleted from or inserted into the file. (3) They do not minimize page access: The secondary key has the property that when a query specifies a condition on this key, there are many records that satisfy this condition.[2]

[2] If the key has values from a discrete domain, each value usually occurs many times. If the key has values from a continuous domain, then only range quieries are meaningful. In either case, a condition on this key will hit many records.

These target records distribute randomly over the whole file and the page accesses is about as many as the number of the target records. Rothnie has calculated that, if a file has 64 pages and if 20 target records are randomly distributed over the 64 pages, then the expected number of page accesses is 17.3.

If the retrieval on secondary keys is a nontrivial part of the file activity, then why not structure the file on the secondary keys instead of the primary key so that the target records are not randomly distributed with respect to secondary keys? An immediate idea that arises is to somehow sort the file on the secondary keys and create a directory[3] for it. A simple way to do it is to concatenate the secondary keys and treat the concatenation as a single key. An example is shown in Fig. 1. The keys A_2, A_3 and A_4 are concatenated. The file is sorted on this concatenation and a directory is created to facilitate the searches. It is obvious that this scheme is only helpful for queries conditioned on the first key (A_2) and is useless for queries conditioned on other keys (A_3 or A_4) alone.

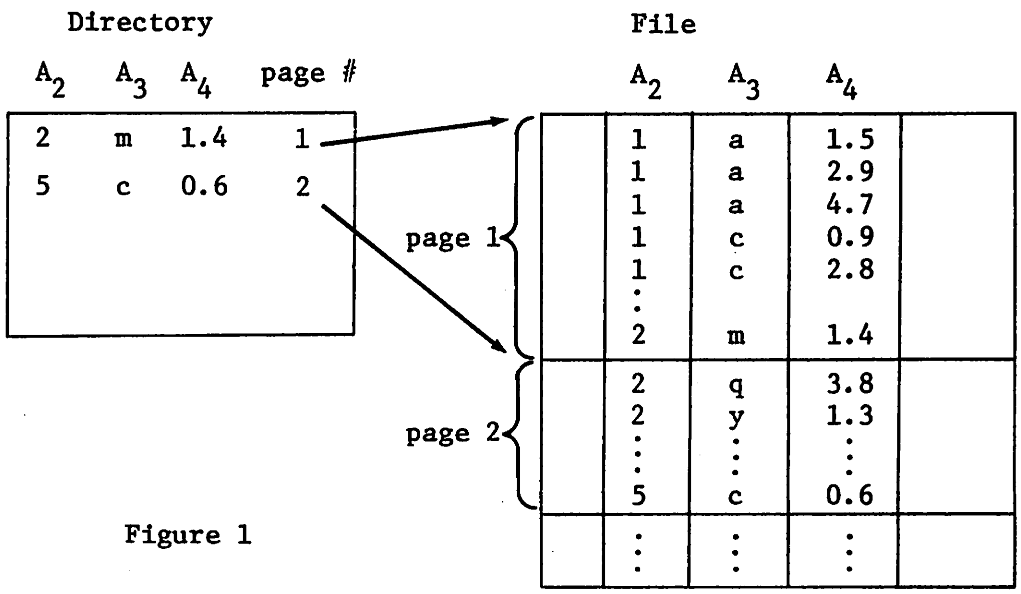


Figure 1

[3]In this paper we distinguish a directory from an index by insisting that 'directory' is used only when the file is structured on the key in the directory. Thus ISAM is a directory, while an inverted file is an index.

Another way of structuring the file is to hash every secondary key independently (multiple key hashing) and store the records that have the same set of hashed values into the same page. The secondary keys are treated symmetrically and the overall page access is expected to be minimized. This idea has been discussed by Rothnie [6] and Rivest [7] and will be explained in more detail in the next season.

The multi-dimensional directory is a more sophisticated way to structure a file and construct a directory for it. It has the merits of MKH (symmetrical treatment of keys and overall minimization of page access) and it gets rid of the shortcomings of MKH. Furthermore, the application of B-tree techniques to its maintenance makes it even more attractive when the file is dynamic.

The following table shows the status of MDD in relation to other access methods:

	hashing	indexing
retrieval on single key	BDAM	ISAM, B-TREE
retrieval on multiple keys	MKH	MDD

2. MKH METHOD

Assume a file has k secondary keys $a_1,a_2,\dots,a_k$ and the corresponding domains are $D_1,D_2,\dots,D_k$ respectively.

(1) Select a hash function h_i for each domain D_i such that $h_i : D_i \to \{1,2,\dots,m_i\}$; where $m_1 \times m_2 \times \dots \times m_k = N$. the total number of pages needed for the file.

(2) Associate with each tuple $<z_1,z_2,\dots,z_k>$ a page in the secondary memory, where z_i is an integer and $1 \le z_i \le m_i$.

(3) If the keys $a_1,a_2,\dots,a_k$ of record r have values $v_1,v_2,\dots,v_k$ then store r in the page associated with $<h_1(v_1), h_2(v_2),\dots,h_k(v_k)>$,

where $v_1 \in D_1$, $v_2 \in D_2, \ldots, v_k \in D_k$.

In this scheme, the retrieval of records with $a_i = v_i$ requires N/m_i page accesses, and retrieval of records with keys $a_1 = v_1 \wedge a_2 = v_2 \wedge \wedge \ldots \wedge a_k = v_k$ requires 1 page access. The algorithms for retrieval, insertion and deletion are simple. The keys are created symmetrically, and the overall page access is considerably reduced.[4]

However, MKH has its shortcomings. We will discuss them by first classifying the hash functions into two types: type I hash functions do not preserve ordering, while type II hash functions do. That is, $v_1 \leq v_2 \not\Rightarrow h_1(v_1) \leq h_1(v_2)$ and $v_1 \leq v_2 \Rightarrow h_2(v_1) \leq h_2(v_2)$, where h_1 belongs to type I and h_2 belongs to type II.

If we use type I hash functions in MKH then

(1) To answer a range query, a search through the whole file is necessary.

(2) If the domain has discrete values, some values may occur more frequently than others. Then even after very sophisticated hashing the record distribution can still be far from uniform. This may result in high collision and poor performance.

The type II hash function h for domain D is obtained by dividing domain D into m intervals $I_1, I_2, \ldots, I_m$ and letting $h(v) = j$ if $v \in I_j$. Figure 2 shows a two dimensional example. Domain A is partitioned into 4 intervals and domain B into 5 intervals. The two hash functions partition the domain space into 20 cubes. A page is assigned to each cube. A record can be represented by a point in the domain space and a

[4]The values of $m_1, m_2, \ldots,$ and m_k are determined by the relative frequencies of different queries. If query with condition a_i = (some constant) occurs more frequently than others, we should make m_i large so that the page access for this query $(\frac{N}{m_i})$ is small. For overall optimality see [6] or section 5 of this paper.

file can be represented by a set of points in the domain space.[5] Obviously, a range query can now be answered without searching through the whole file, and a cube can be made smaller where the record density is high. But this type of hash functions has its own drawbacks.

(1) If there is some relationship between two domains, the storage utility can be very low. For example, in case a domain pair (D_1,D_2) is (AGE, SALARY) or (WEIGHT, HEIGHT) then values of key 1 and key 2 tend to be proportional and record distribution will show a high density around the diagonal line (Fig. 3). The pages assigned to cubes off the diagonal

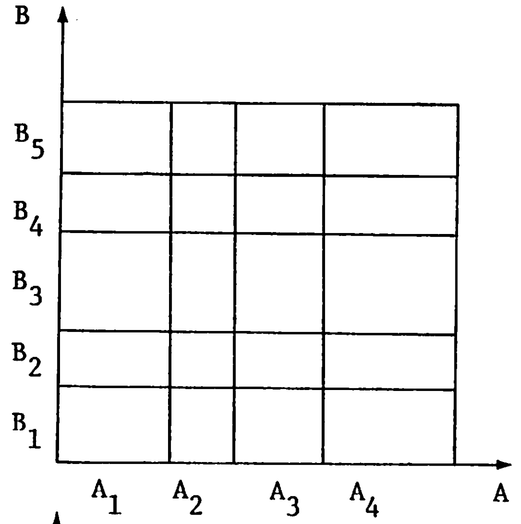


Figure 2

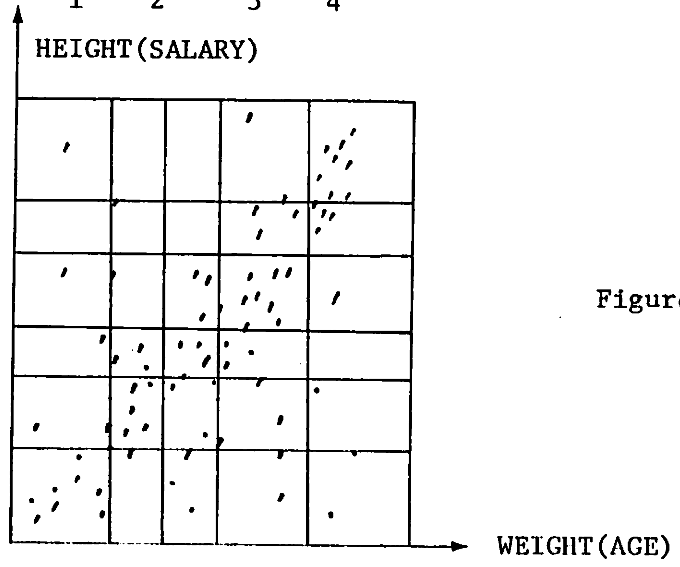


Figure 3

[5]A point in the domain space may correspond to more than one record occurrence in the file.

line have very few records in them. When there are more domains with some relationships among them, the storage utility can be extremely low. (2) If the partitions are made blindly, the performance can be very bad. The common practice is that a hash function of this type is used only when the record distribution is known before designing the hash function. If the record distribution changes (due to deletions and insertions) the performance can degrade very fast and reorganization will be necessary.

3. ONE-LEVEL MULTI-DIMENSIONAL DIRECTORY

In the following, a two-dimensional example is used. The generalization of the algorithms to any number of dimensions is straightforward. Let us assume we want to construct a two-dimensional directory for a file on two attributes whose domains are A and B respectively. In MDD the domain space is partitioned into small cubes (Fig. 4a) in a different way than is done by the type II hash functions in MKH, and the boundaries of the cubes are free to shift as we will see in Section 6. The cubes are obtained from the following procedure. Let us denote the total number of records in the file by pN, where p is the number of records in a page and N is the total number of pages in the file.

(1) Divide the domain A into m_1 intervals $A_1, A_2, \ldots, A_{m_1}$ such that each subspace $A_i \times B (i=1,2,\ldots,m_1)$ contains approximately pN/m_1 records (Fig. 4b). Let us call each $A_i \times B$ subspace a <u>1st-degree cube</u> and denote it by C_i.
(2) For each 1st-degree cube $A_i \times B$ divide domain B into m_2 intervals $B^i_1, B^i_2, \ldots, B^i_{m_2}$ (Fig. 4c) such that each subspace $A_i \times B^i_j$ contains approximately $pN/(m_1 \times m_2)$ records. Let us call each $A_i \times B^i_j$ subspace a <u>2nd-degree cube</u> and denote it by C_{ij}. Also let us call the whole domain space a <u>0th-degree cube</u> and denote it by C. C has C_i, $i=1,2,\ldots, m_1$ as

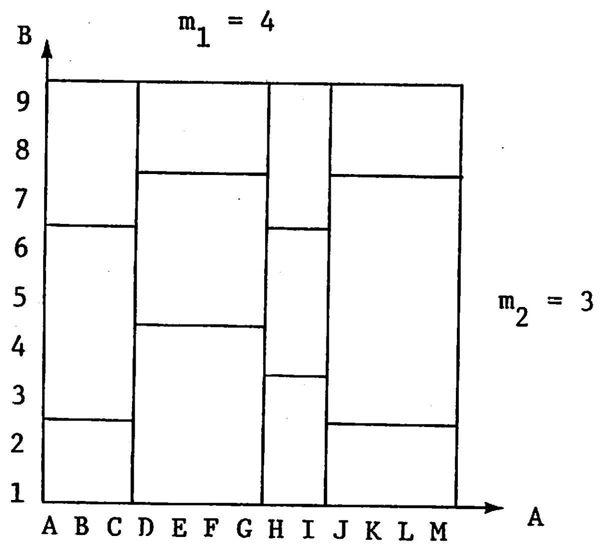


Figure 4a

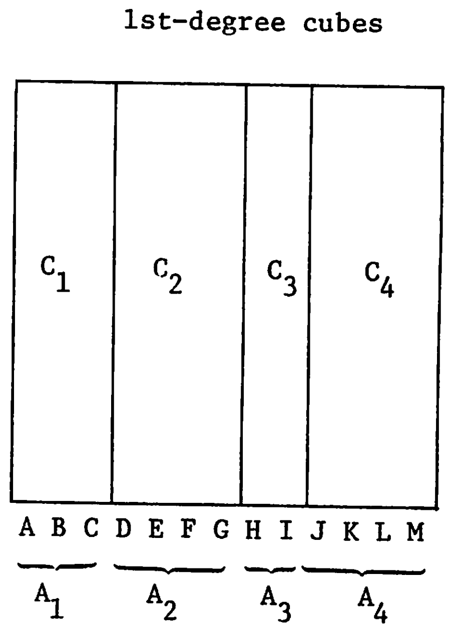


Figure 4b

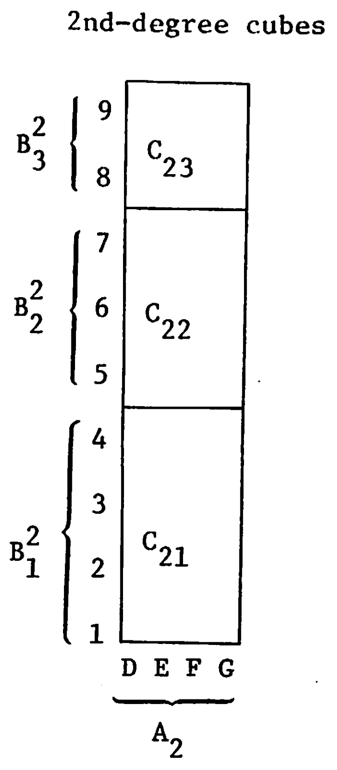


Figure 4c

MDD

A_L	A_H	B_L	B_H	Page #	# of records
A	C	1	2	(1.1)	
A	C	3	6	(1.2)	
A	C	7	9	(1.3)	
D	G	1	4	(2.1)	
D	G	5	7	(2.2)	
D	G	8	9	(2.3)	
H	I	1	3	(3.1)	
H	I	4	6	(3.2)	
H	I	7	9	(3.3)	
J	M	1	2	(4.1)	
J	M	3	7	(4.2)	
J	M	8	9	(4.3)	

Figure 4d

its subcubes and C_i has C_{ij}, $j = 1,2,\dots,m_2$ as its subcubes. If y is a subcube of x then x is a supercube of y.

(3) Assign a page of secondary memory to each 2nd-degree cube, and construct a directory (Fig. 4d). There is an entry in the directory for each 2nd-degree cube. The entry contains the high and low keys of every dimension of the cube and a pointer to the page assigned to the cube.

(4) All records occurring in a cube are stored in the page assigned to the cube. The total number of records stored in the page is indicated in the last column of the directory.

To answer a user query, a system query is issued to search the MDD to find the pages that store the target records. The following are a few examples expressed in language ALPHA [3]:

(i) user query : GET W x : x.B = 4

system query : GET Z y. page # : $y . B_L \leq 4 \wedge y . B_H \geq 4$ (1)

result : $Z \leftarrow (1.2), (2.1), (3.2), (4.2)$

(ii) user query : GET W x : $x . A = F \wedge x . B = 4$

system query : GET Z y . page # : $y . A_L \leq F \wedge y . A_H \geq F \wedge$
$y . B_L \leq 4 \wedge y . B_H \geq 4$

result : $Z \leftarrow (2.1)$

(iii) user query : GET W x : $4 \leq x . B \leq 5$

system query : GET Z y : $(y . B_L \leq 5 \wedge y . B_H \geq 4) \vee$
$(y . B_L \leq 5 \wedge y . B_H \geq 4)$

result : $Z \leftarrow (1.2), (2.1), (2.2), (3.2), (4.2)$

If we want to construct a k-dimensional directory, it is only necessary to generate the k-th-degree cubes. Figure 5 depicts a 3-dimensional example with $m_1 = 4$, $m_2 = 3$, $m_3 = 3$. A page is assigned

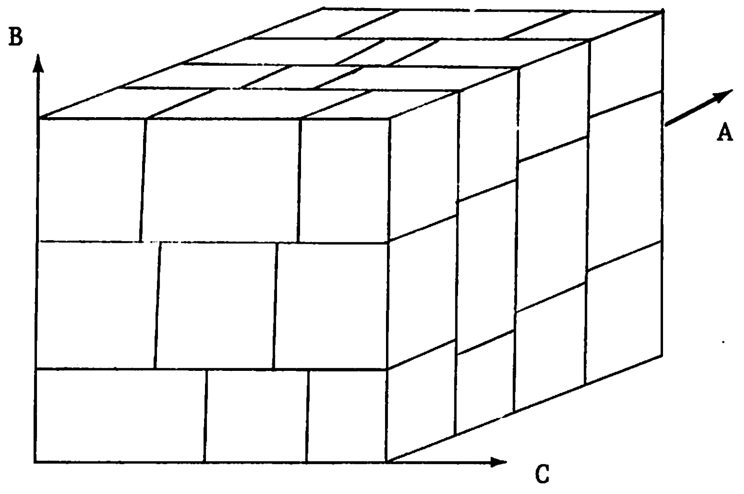


Figure 5

to each 3rd-degree cube and there is an entry in the MDD for each 3rd-degree cube, which contains the high and low keys of all dimensions of the cube. Now $N = m_1 \times m_2 \times m_3 = 36$. The number of page accesses for GET $x : x . A = a$ is N/m_1; for GET $x : x . A = a \wedge x . B = b$ is N/m_1m_2; and for GET $x : x . A = a \wedge x . B = b \wedge x . C = C$ is $N/m_1m_2m_3$. It is approximately equivalent to the MKH method with $h_1 : A \rightarrow \{1,...,m_1\}$, $h_2 : B \rightarrow \{1,...,m_2\}$, $h_3 : C \rightarrow \{1,...,m_3\}$. However, there is an asymmetry in the fact that domain A is partitioned first, B next and C last. Which domain is partitioned first does not affect the retrieval performance. However, when records are deleted or inserted and a cube split or collapse has to be done, it seems that dimensions should be ordered in decreasing m value (see section 6). That is, the dimension with largest m value should be partitioned first as we have done in last two examples. This will minimize the number of pages involved in a split or collapse.

The way we partition the domain space in MDD (instead of partitioning all dimensions at the same time, we partition them in sequence) not only makes the performance unaffected by the record distribution in the domain space (because all cubes of same degree contain the same number of records), but also makes it convenient to modify the partition (see section 6). Obviously the MDD scheme is as efficient as the MKH scheme as far as retrieval is concerned. It is efficient for range query and extra storage required is negligible when compared to the main file. Although we have assumed the knowledge of record distribution in explaining the MDD, we will see this is not necessary if we use B-tree techniques to construct and maintain the MDD.

4. B-TREE TECHNIQUES

The basic techniques used in B-tree [9] can be summarized by the following:

(1) At the insertion of an entry (a record in the case of a leaf page; a key and a pointer in the case of any non-leaf page) a page may become overloaded, then

```
(if)    a brother on either side is not maximally loaded
(then)  overflow an entry(s) to that brother
(else)  get a free page, split the overloaded page in the middle,
        move half of the entries to the new page, and add one
        entry to the father page.
```

(2) At the deletion of an entry, a page may become underloaded, then

```
(if)    a brother on either side is not minimally loaded
(then)  underflow an entry(s) from that brother
(else)  collapse the page with one brother and delete an entry
        from the father page.
```

5. PRELIMINARIES

We can determine m_1 and m_2 so that the overall number of page accesses is minimal by considering the relative frequencies of different types of queries. The following is a list of each query type, its relative frequency of occurrence and the page accesses it needs:

query[5] type	relative frequency	page accesses needed
$x . A = a_1$	p_1	m_2
$x . A \in [a_2,a_3]$	q_1	$k_1 m_2$
$x . B = b_1$	p_2	m_1
$x . B \in [b_2,b_3]$	q_2	$k_2 m_1$
$x . A = a_1 \wedge x . B = b_1$	p_{12}	1
$x . A \in [a_2,a_3] \wedge x . B \in [b_2,b_3]$	q_{12}	k_3

The constants p_1, q_1, p_2, q_2, p_{12}, q_{12}, k_1, k_2, k_3 can be obtained by a user estimate or system monitoring. Then the average number of page access (npa) for any single query is

$$E[npa] = p_1\, m_2 + q_1\, k\, m_2 + p_2\, m_1 + q_2\, k_2\, m_1 + p_{12} + q_{12}\, k_3$$

$$= P_1\, m_2 + P_2\, m_1 + P_{12},$$

where

$$P_1 = p_1 + q_1\, k_1,$$
$$P_2 = p_2 + q_2\, k_2,$$
$$P_{12} = p_{12} + q_{12}\, k_3.$$

Also $m_1 \times m_2 = N(t)$, the total number of pages in the file, which

[5a]Only 1-(tuple) variable queries are considered because a complicated query can always be decomposed into a sequence of 1-variable queries [10,11].

[5b]The queries are classified by the qualification part.

[5c]The query of type $x . A = a_1 \vee x . B = b_1$ can be considered as two separate queries $x . A = a_1$ and $x . B = b_1$.

is a function of time.

Minimizing E[npa], we get

$$m_1 = \sqrt{\frac{P_1}{P_2} N(t)} \quad , \qquad m_2 = \sqrt{\frac{P_2}{P_1} N(t)}$$

and $\frac{m_1}{m_2} = \frac{P_1}{P_2}$.

The ratio of m_1 to m_2 is independent of the file size. In general case, we have $\frac{m_1}{P_1} = \frac{m_2}{P_2} = \ldots = \frac{m_k}{P_k}$. Therefore, we should keep the ratio of $m_1 : m_2 : \ldots : m_k$ as close as possible to the ratio of $P_1 : P_2 : \ldots : P_k$ in order to have optimal overall performance. In two-dimensional case if $\frac{m_1}{m_2} = \frac{P_1}{P_2} \doteq \frac{5}{3}$ then when N increases, m_1 and m_2 should increase as the arrows in the following diagram indicate:

$$\begin{array}{lllllll} m_1 & 1 \rightarrow 2 \rightarrow 3 & 3 \rightarrow 4 & 4 \rightarrow 5 \rightarrow 6 & \ldots \\ m_2 & 1 \quad\;\; 1 \quad\;\; 1 \rightarrow 2 & 2 \rightarrow 3 & 3 \quad\;\; 3 & \ldots \end{array}$$

We will call this <u>inter-dimensional constraint</u>.

Let us define $|c_{i_1 \ldots i_k}|$ = number of subcubes of cube $c_{i_1 \ldots i_k}$. We have assumed $|c| = m_1$ and $|c_i| = m_2$ for all $i = 1,2,\ldots,m_1$. We will now let $m_2 = \max_i |c_i|$, $m_2' = \min_i |C_i|$ and allow $m_2 - m_2' \leq 1$. This is called <u>intra-dimensional constraint</u>. This constraint says that for any two cubes of same degree their numbers of subcubes should not differ by more than one.

In a k-dimensional case,

$$m_j \triangleq \max_{\substack{i_1 = 1,2,\ldots,m_1 \\ \vdots \\ i_{j-1} = 1,2,\ldots,m_{j-1}}} |c_{i_1 \ldots i_{j-1}}| \quad , \qquad m_j' \triangleq \min_{\substack{i_1 = 1,2,\ldots,m_1 \\ \vdots \\ i_{j-1} = 1,2,\ldots,m_{j-1}}} |c_{i_1 \ldots i_{j-1}}|$$

and the intra-dimensional constraint is $m_j - m'_j \leq 1$ for all $j = 2,\ldots,k$.

When a file grows or shrinks, these constraints should be observed so that the domain space is always optimally partitioned with respect to overall page access.

6. APPLYING B-TREE TECHNIQUES TO MDD

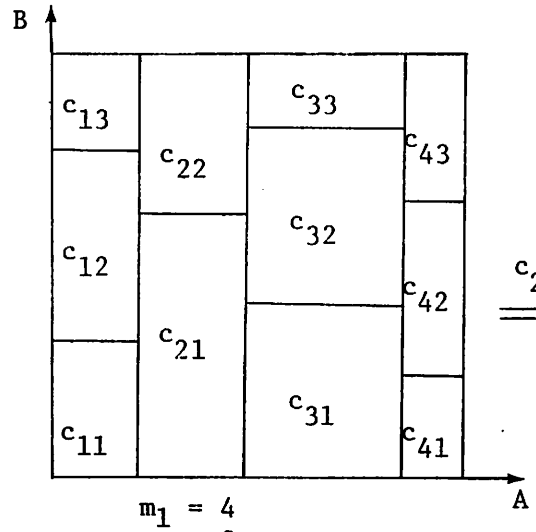
In one-dimensional directory, only one key is associated with each entry, and there is only one way to split the content of a page into two equal parts. In the multi-dimensional case, each entry has as many keys as there are dimensions. When a page becomes overloaded it is the last dimension that is considered first (recall that we have ordered the dimensions according to their m values). Again let us use the two-dimensional example to illustrate the idea. When a 2nd-degree cube is overloaded it can split into two 2nd-degree cubes or overflow some records to other 2nd-degree cubes. All these activities should be within the supercube of the overloaded cube. If neither split nor overflow is possible then the supercube (a 1st-degree cube) is considered. It can split into two 1st-degree cubes or overflow some records to other 1st-degree cubes. Under all circumstances the inter- and intra- dimensional constraints should be observed.

For example, if $P_1/P_2 = 5/3$ and the domain space is currently partitioned as in Fig. 6a. The inter-dimensional constraint is repeated here and the current state indicated by an arrow:

$$\begin{array}{llllllllllll}
m_1 & 1 & 2 & 3 & 3 & 4 & & 4 & 5 & 6 & 6 & \ldots \\
m_2 & 1 & 1 & 1 & 2 & 2 & \leftrightarrow & 3 & 3 & 3 & 4 & \ldots
\end{array}$$

If c_{21} becomes overloaded, it can split (Fig. 6b). This means that half of the records in the page assigned to c_{21} are moved to a new page.

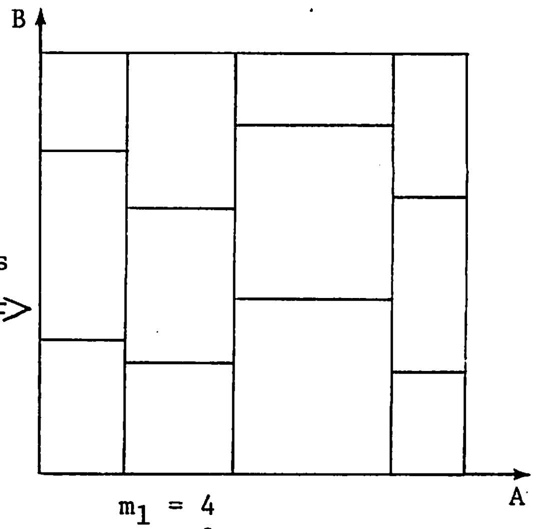
In the directory the entry for c_{21} is modified and an entry for the new page is inserted. But if c_{31} becomes overloaded, it cannot split. Because if it splits then m_2 (= $\max|c_i| = |c_3|$) becomes 4 and m_1 remains 4. This violates the inter-dimensional constraint. Also $m_2 - m_2' = 2$ and the intra-dimensional constraint is violated. Therefore, if c_{31} becomes overloaded, records should be overflowed to c_{32}, which if in turn becomes overloaded should overflow to c_{33} (Fig. 6c). The boundaries between c_{31} and c_{22}, and c_{32} and c_{33} are shifted downward because of the record migration. The entries in the directory are correspondently modified. If both c_{32} and c_{33} are maximally loaded and again c_{31} becomes overloaded then cube c_3 (the supercube of c_{31}) is considered. We can overflow c_3 to c_2 (or c_4) and a subcube of c_2 may split (Fig. 6d). If later c_1 becomes overloaded, it is free to split (no violation of either constraint) (Fig. 6e). The algorithm for a K-dimensional case is shown in Fig. 7. Note that whenever a cube is overloaded, the algorithm is used to try to solve the problem within the supercube of the overloaded cube. If the problem cannot be solved within the supercube, the algorithm is used again with the supercube as the overloaded cube. The algorithm for underload is shown in Fig. 8.



$m_1 = 4$
$m_2 = 3$
$m_2' = 2$

Figure 6a

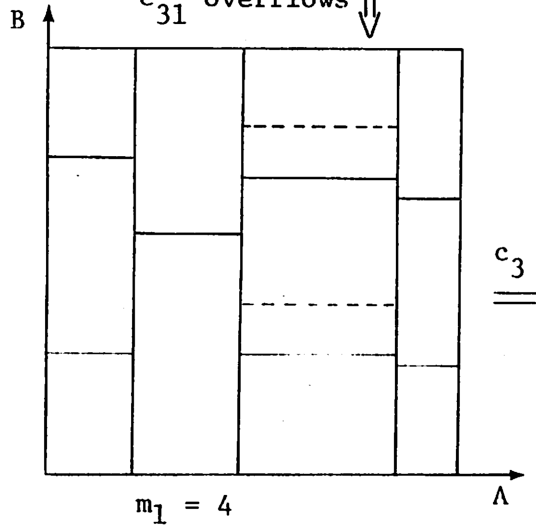
c_{21} splits



$m_1 = 4$
$m_2 = 3$
$m_2' = 3$

Figure 6b

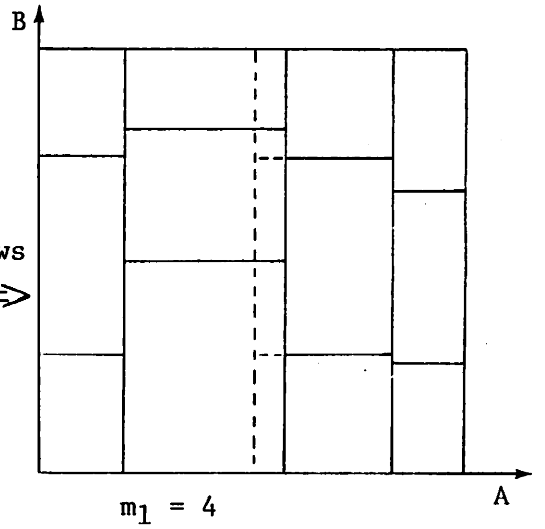
c_{31} overflows



$m_1 = 4$
$m_2 = 3$
$m_2' = 2$

Figure 6c

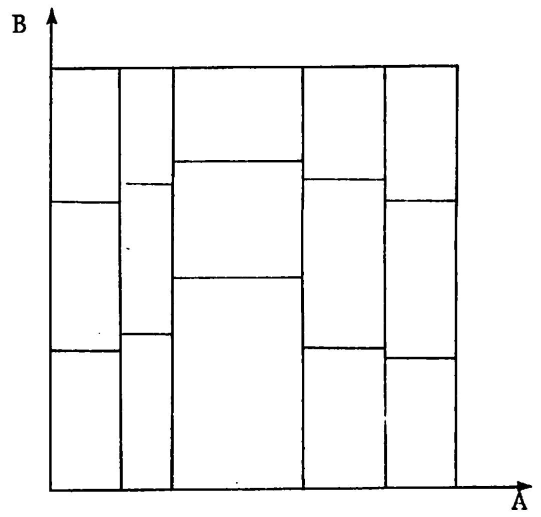
c_3 overflows



$m_1 = 4$
$m_2 = 3$
$m_2' = 2$

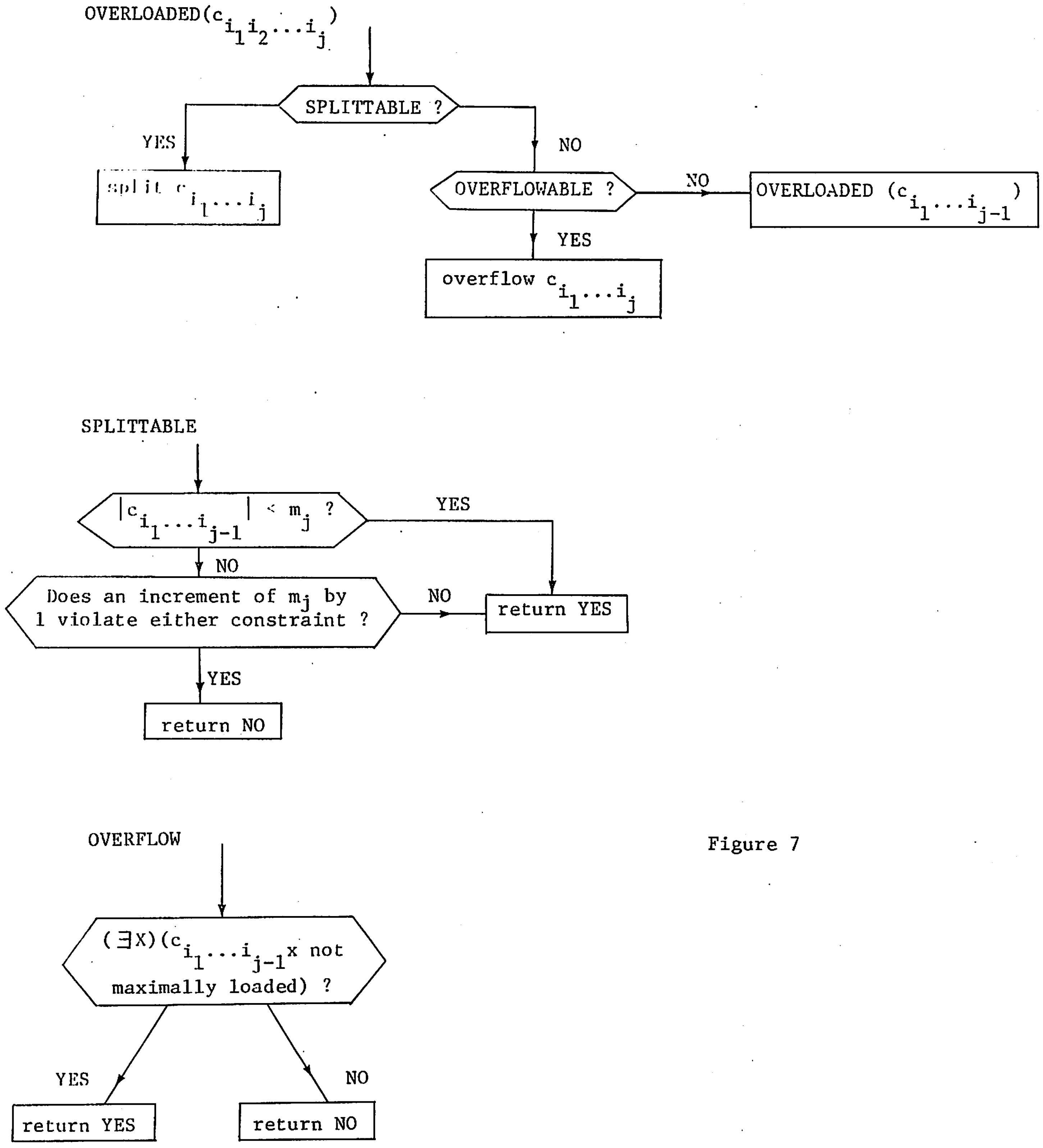
Figure 6d

c_1 splits



$m_1 = 5$
$m_2 = 3$
$m_2' = 3$

Figure 6e



OVERLOADED($c_{i_1 i_2 \ldots i_j}$)
SPLITTABLE ?
YES
NO
split $c_{i_1 \ldots i_j}$
OVERFLOWABLE ?
NO
OVERLOADED ($c_{i_1 \ldots i_{j-1}}$)
YES
overflow $c_{i_1 \ldots i_j}$
SPLITTABLE
$|c_{i_1 \ldots i_{j-1}}| < m_j$?
YES
NO
Does an increment of m_j by 1 violate either constraint ?
NO
return YES
YES
return NO
OVERFLOW
$(\exists X)(c_{i_1 \ldots i_{j-1}} x$ not maximally loaded) ?
YES
NO
return YES
return NO

Figure 7

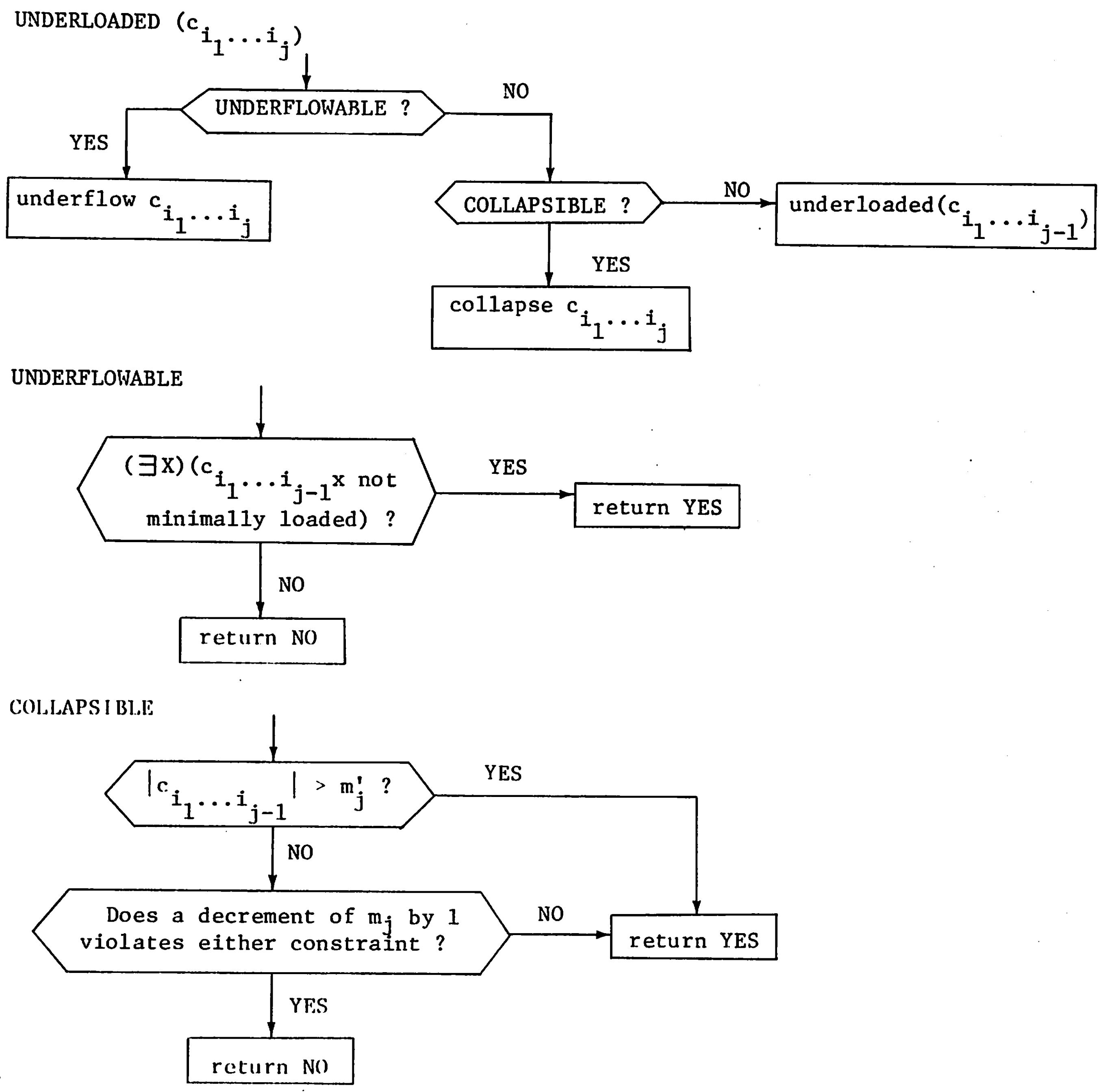

UNDERLOADED ($c_{i_1 \ldots i_j}$)
UNDERFLOWABLE ?
NO
YES
underflow $c_{i_1 \ldots i_j}$
COLLAPSIBLE ?
NO
underloaded($c_{i_1 \ldots i_{j-1}}$)
YES
collapse $c_{i_1 \ldots i_j}$
UNDERFLOWABLE
$(\exists X)(c_{i_1 \ldots i_{j-1}} x$ not minimally loaded) ?
YES
return YES
NO
return NO
COLLAPSIBLE
$|c_{i_1 \ldots i_{j-1}}| > m'_j$?
YES
NO
Does a decrement of m_j by 1 violates either constraint ?
NO
return YES
YES
return NO

Figure 8

In these algorithms, an overload (underload) of a k-th degree cube requires 2 to m_k page accesses, and an overload of a (k-1)st degree cube requires (2 to m_{k-1}) * m_k page accesses, and so on. The overload of high degree cubes has much higher frequency of occurrence than the overload of lower degree cubes. The average page access for each insertion (deletion) should still be low. Also we can see now if we make $m_1 \geq m_2 \geq m_3 \geq \cdots \geq m_k$, then the overload of higher degree cubes (which happens more frequently) requires less page accesses. This is why we want to order the dimensions in decreasing m values.

7. MDD OF MORE LEVELS

When the file is large, the directory itself may occupy more than one page, and we need a directory for the directory. A two level MDD is shown in Fig. 9. Now there are two levels of cubes. The cubes in the second level are separated by double lines. A 2nd-degree cube in the second level is a 0th-degree cube in the first level. Let us denote the cubes in the second level by c_i (1st-degree) and c_{ij} (2nd-degree), and denote the cubes in the first level by $c_{ij;k}$ (1st-degree) and $c_{ij;k\ell}$(2nd-degree).

A page (leaf page) is assigned to each $c_{ij;k\ell}$ for all i,j,k,ℓ; the page stores all records that occur in this cube. A page (first-level MDD page) is assigned to each cube c_{ij} for all i,j; this page contains an entry for each cube $c_{ij;k\ell}$; the entry contains the high and low keys of every dimension in cube $c_{ij;k\ell}$, and a pointer to the page assigned to cube $c_{ij;k\ell}$. A page (second-level MDD page) is assigned to cube c, it contains an entry for each cube c_{ij}, for all i,j; the entry contains the high and low keys of every dimension in cube c_{ij} and a pointer to the

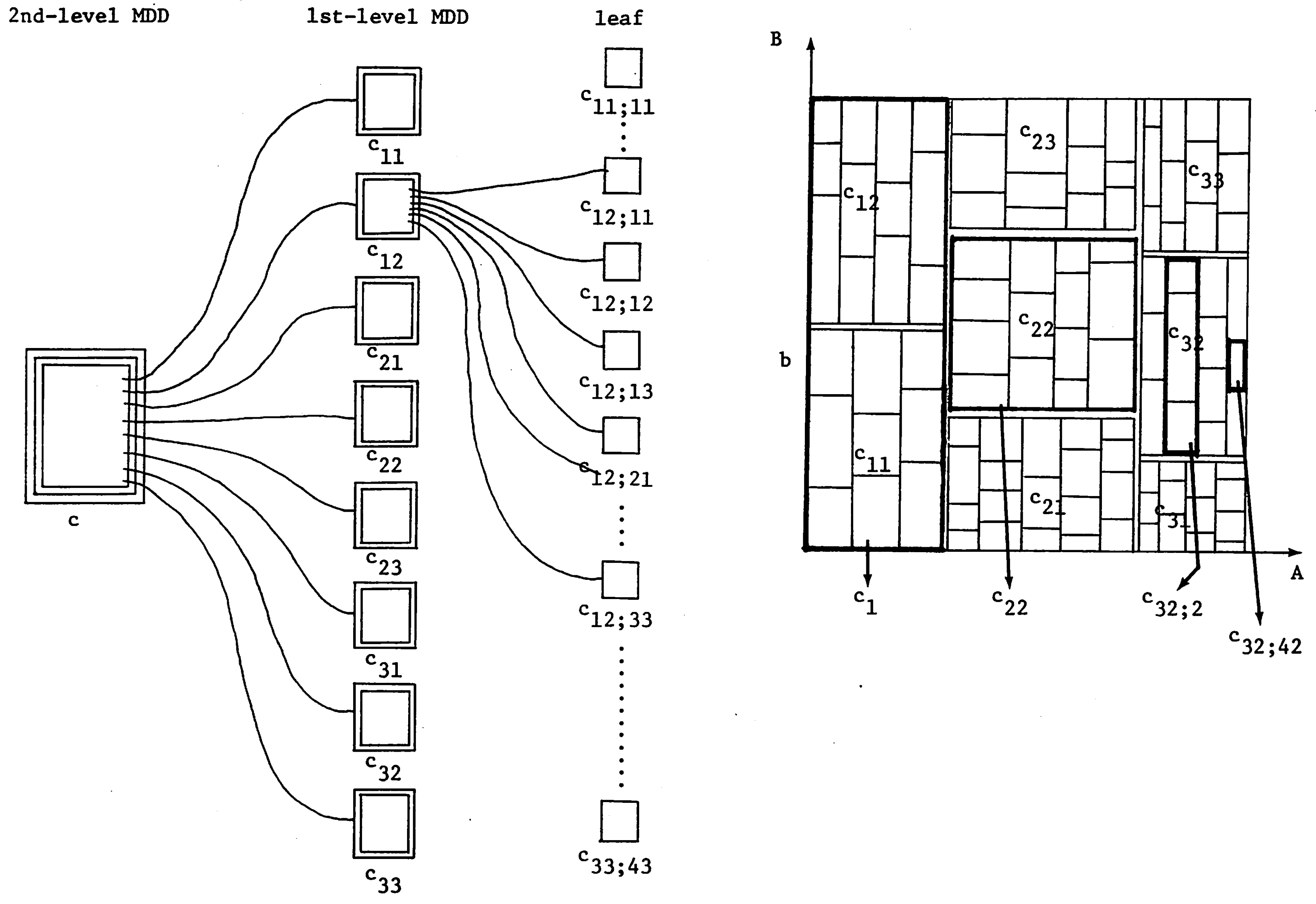

2nd-level MDD
1st-level MDD
leaf
c
c_{11}
c_{12}
c_{21}
c_{22}
c_{23}
c_{31}
c_{32}
c_{33}
$c_{11;11}$
$c_{12;11}$
$c_{12;12}$
$c_{12;13}$
$c_{12;21}$
$c_{12;33}$
$c_{33;43}$
B
b
A
c_{12}
c_{11}
c_{23}
c_{22}
c_{21}
c_{33}
c_{32}
c_{31}
c_1
c_{22}
$c_{32;2}$
$c_{32;42}$

Figure 9

page assigned to cube c_{ij}.

When a user query GET w x : x.B = b_1 is processed, a system query like (1) in section 3 is generated, page c is searched and page addresses of c_{11}, c_{22}, c_{32} are returned. Then the same query is issued to pages c_{11}, c_{22} and c_{32} and the page addresses of all (leaf) pages that contain the target records are returned.

When overload (underload) occurs at an insertion, split and overflow (collapse and underflow) are tried in the first level. If this fails, the second level is considered and split and overflow are tried in the second level.

8. A VARIATION

The algorithms in section 6 have been purposely primitive in order to make the basic ideas clear and the retrieval cost has been minimized at the expense of insertion and deletion efficiency. When insertion and deletion are frequent, it is advisable to modify the algorithms so that the overall cost is minimal. In the appendix (section 11) we show that suppressing split and collapse until the last moment is not a good idea and a simple modification can appreciably improve the B-tree algorithms. We will extend this idea to MDD.

Again let us use the two-dimensional example. The main modification is to loosen the intra-dimensional constraint $m_2 - m_2' \leq 1$ and replace it by $\frac{2}{3} m_2 \leq |c_i| \leq \frac{4}{3} m_2$ for all $i = 1,2,\ldots,m_1$, where m_2 is determined by the inter-dimensional constraint. In section 6 c_i is maximally loaded if $|c_i| = m_2$ and c_{ij} is maximally loaded for all $j = 1,2,\ldots,m_1$. Let us now determine the fullness of c_i solely by the number of its subcubes disregarding the fullness of its subcubes. That is cube c_i is overloaded if $|c_i| > \frac{4}{3} m_2$; underloaded if $|c_i| < \frac{2}{3} m_2$; maximally loaded if $|c_i| = \frac{4}{3} m_2$; and minimally loaded if $|c_i| = \frac{2}{3} m_2$. If we use the principles in the appendix and the new constraint then the algorithms read as follows:

```
OVERLOAD ($c_{ij}$)
    (if)  a brother of $c_{ij}$ is not nearly maximally loaded
      (then) overflow to that brother
      (else) if $|c_i| = \frac{4}{3} m_2$ then OVERLOADED'($c_i$)
             else split $c_{ij}$ and its most loaded adjacent brother into 3 cubes.

OVERLOADED'($c_i$)
    (if)  a brother of $c_i$ is not nearly maximally loaded
      (then) overflow to that brother
      (else) if $m_1 \leftarrow m_1+1$ does not violate the inter-dimensional
                constraint
             then split $c_i$ and its most loaded adjacent brother into 3 cubes
                and increment $m_1$ by 1
             else split $c_{ij}$ and its most loaded adjacent brother into 3 cubes
                and increment $m_2$ by 1

UNDERLOADED ($c_{ij}$)
    (if)  a brother is not nearly minimally loaded
      (then) underflow from that brother
      (else) if $|c_i| = \frac{2}{3} m_2$ then UNDERLOADED'($c_i$)
             else collapse $c_{ij}$ and its adjacent brothers into 2 cubes

UNDERLOADED'($c_i$)
    (if)  a brother is not nearly minimally loaded
      (then) underflow from that brother
```

```
(else) if $m_1 \leftarrow m_1-1$ does not violate the inter-dimensional
           constraint
       then collapse $c_i$ and its adjacent brothers into 2 cubes
           and decrement $m_1$ by 1
       else collapse $c_{ij}$ and its adjacent brothers into 2 cubes
           and decrement $m_2$ by 1
```

In these algorithms, when a cube is overloaded (underloaded), the overflow (underflow) involves only the two adjacent brothers, and it does not propagate to non-adjacent brothers. The looser constraint allows the overload and underload problems to be solved locally most of the time; but on the other hand, it degrades the retrieval efficiency. The tradeoff between retrieval cost and insertion-deletion cost is obvious.

9. TREATMENT OF THE PRIMARY KEY

The primary key K of a file is by definition [12] a combination of attributes (possibly a single attribute) of the file with properties p_1 and p_2.

p_1. In each record of the file, K uniquely identifies that record.

p_2. No attribute in K can be discarded without destroying property p_1.

A primary key is simple if it consists of only one attribute and is compound if it consists of two or more attributes.

When the MDD method is used to structure the file, we can consider each attribute in the primary key as a secondary key and incorporate it in the MDD. In case the primary key is compound, each attribute does not identify a record and therefore possesses the property of a secondary key mentioned in section 1 note 2. Incorporation of these attributes as secondary keys in the MDD is natural and profitable. On the other hand, if the primary key is simple, it should be excluded from the MDD. We can create an inverted file for it and store the inverted file in a B-tree, in which a record consists of a primary key and a pointer to the record identified by the primary key in the main file. A transaction[6] then needs 1 to 4 page accesses (depending on the size of the file; because of the large fan-out of a B-tree, its height is unlikely to exceed 4) for the search of the B-tree and 1 page access to the page that contains the target record. This is only 1 more access than if the file is structured by the primary key and stored in a B-tree. (The extra access is due to the separation of the main file and the inverted file). And obviously this is much more economical than to incorporate the primary key in the MDD. On first thought since the file is not structured by the primary key, the batch[7] processing may be difficult. Actually a technique similar to "batch random" [13] easily solves this problem. The keys in the batch are sorted, then the B-tree of the inverted file is searched (probably sequentially) and returns a set of primary key and pointer pairs. The set of the primary key and pointer pairs are then sorted on the pointers and finally they are sequentially processed.

10. SUMMARY

The new concept of a single directory for more than one key has

[6]A transaction is the access of a single record (given its primary key) at a request.

[7]A batch is the access of a set of records (given their primary keys) at a request.

been presented. The directory treats all keys symmetrically, and helps the search of records on any combination of any number of these keys. The retrieval efficiency is expected to be as good as the MKH method.

Also introduced are the basic ideas of the algorithms for the construction and maintenance of the directory. Attempts have been made to employ the B-tree techniques in these algorithms. The insertion and deletion cost are believed to be lower than those of the file inversion scheme (note that every insertion and deletion causes an update on every inverted file, while most of the time nothing has to be done to the MDD).

11. APPENDIX

If we follow the algorithm given in section 4, overflow, underflow, split and collapse can be very frequent in some situations. When adjacent pages are nearly maximally (minimally)loaded and entries keep coming in, frequency of overflow (underflow) will become higher and higher until a split (collapse) occurs. Also, because newly split pages tend to collapse and a newly collapsed page tends to split, when insertions and deletions are mixed, the frequencies of split and collapse can be very high [9]. A simple modification that adopts early split and early collapse easily gets around these undesirable situations.

Assume x, y, z are adjacent pages and each page can store 100 entries. Let $|w|$ denote the number of entries stored in page w and let

$$M = \begin{cases} x & \text{if} \quad |x| \geq |z| \\ z & \text{if} \quad |x| < |z| \end{cases}$$

$$N = \begin{cases} x & \text{if} \quad |x| \leq |z| \\ z & \text{if} \quad |x| > |z| \end{cases}$$

Also let w' denote page w denote page w after an underflow or overflow.

```
(1) y becomes overload ($|y| > 100$)
    (If) $|y| + |N| \geq 190$
        (then) split y and M into three pages, each containing
            $(|y| + |M|)/3$ entries
        (else) overflow entries from y to N so that
            $|y'| = |N'| = (|y| + |N|)/2$
(2) y becomes underloaded ($|y| < 50$)
    (If) $|y| + |x| + |z| \leq 180$
        (then) collapse x, y, z into 2 pages, each containing
            $(|y| + |x| + |z|)/2$ entries
        (else) underflow entries from M to y so that
            $|y'| = |M'| = (|y| + |M|)/2$
```

The thresholds 190, 50 and 180 are arbitrarily selected here. Obviously, in this scheme, pages are more evenly loaded and frequency of overflow, underflow, split and collapse are considerably reduced.

ACKNOWLEDGEMENT

The author is grateful to Professor L. A. Zadeh and the attendants of his weekly seminar for their helpful discussions. Thanks are also due to Mr. Rowland Johnson for his careful reading of the draft of this paper.

REFERENCES

[1] Whitney, V. K. M., "Relational Data Management Implementation Techniques," ACM SIGMOD, 1974.

[2] Date, C. J. and Codd, E. F., "The Relational and Network Approaches: Comparison of the Application Programming Interfaces," IBM Research Laboratory, San Jose, California, 1974.

[3] Codd, E. F., "A Data Base Sublanguage Founded on the Relational Calculus," IBM Research Laboratory, San Jose, California, 1971.

[4] Chamberlin, D. D. and Boyce, R. F., "SEQUEL: A Structural English Query Language," ACM SIGMOD, 1974.

[5] Knuth, D. E. The Art of Programming, Vol. 3, Section 6.5, 1973.

[6] Rothnie, J. B. and Lozano, T., "Attribute Based File Organization in a Page Memory Environment," CACM, Vol. 17, No. 2, Feb. 1974.

[7] Rivest, R. L., "Analysis of Associative Retrieval Algorithms," Ph.D. Thesis, Computer Science Dept., Stanford University, 1974.

[8] Lum, V. Y., "Multi-attribute Retrieval with Combined Indexes," CACM, Vol. 13, No. 11, Nov. 1970.

[9] Bayer, R. and McCreight, E., "Organization and Maintenance of Large Ordered Indices," Boeing Scientific Research Laboratory, July 1970.

[10] Stonebraker, M., et al., "Preliminary Design of INGRES," ERL-M435 Electronics Research Laboratory, University of California, Berkeley, 1974.

[11] Rothnie, J. B., "An Approach to Implementing a Relational Data Management System," ACM SIGMOD, 1974.

[12] Codd, E. F., "Further Normalization of the Data Base Relational Model," IBM Research Laboratory, San Jose, California, 1971.

[13] Nijssen, G. M., "Indexed Sequential Versus Random," IAG Journal, Vol. 4, March 1971.