Dynamic Multipaging in Even Less Space

T. H. Merrett and Louis Hamel

School of Computer Science

McGill University

We present an improvement to the directory structure of dynamic multipaging which reduces the overall storage requirement from The outstanding advantage of multipaging is that it provides multi-attribute access to a database file using directories of negligible size. The present improvement makes the directory size even more negligible and the addressing algorithm easier.

Keywords disk storage, file organization, multiattribute search, orthogonal range queries

Introduction

Multipaging, or multi-dimensional paging, was proposed by Merrett [1978] and refined by Merrett and Otoo [1982]. It provides symmetric access to all $2^d - 1$ subsets of d attributes of a relation or of a fixed-length file, while requiring storage only for the data itself and for d small "axial directories" of size proportional to the dth root of n, the number of data pages. The axial directories are so called because the data pages are perceived as occupying a d-dimensional hyper-rectangle, to which access is given through the d axes of the space. This organization of data permits orthogonal range queries (including exact-match, partial-match and range queries - all of the practically important cases) and more general geometrical queries to be answered with minimum accesses to the data. Multipaging is thus a complete solution to the problem of indexing secondary keys.

The early work on multipaging solved the problem of allocating pages for static data, for which the distribution of the data in the multi-dimensional space is known in advance. Merrett and Otoo [1982] subsequently showed how to allocate pages dynamically for a growing file. The major sub-problem is that of how to extend the axes of a multidimensional array, a problem which produced a number of papers [Rosenberg 1974, 1975; Rosenberg and Stockmeyer 1977a, 1977b] and was solved by Otoo and Merrett [1983] by admitting d axial directories of size $O(dn^{\frac{1}{d}})$. It is a refinement of this solution which we discuss in the present paper.

Multipaging has also been termed "grid files" [Nievergelt, Hinterberger and Sevcik 1984].

Dynamic Multipaging

As with other dynamic storage schemes (e.g., B-trees [Bayer and McCreight 1972] and linear hashing [Litwin 1980]), dynamic multipaging is based on a page-splitting technique. The important trick is to allow the file to grow without damaging the access

structure. (Here B-trees and linear hashing succeed where, say, I.B.M.'s indexed sequential access method, ISAM, fails.) For multipaging, access is through the spatial axes, which is possible because the pages are bounded by hyperplanes orthogonal to the axes. Thus we must split all pages bounded by a hyperplane if we wish to split one. So dynamic multipaging allows a file of n pages to grow by adding a slab, or "segment" of $n^{1-\frac{1}{d}}$ new pages. Since we create a new segment all at once, its pages can have addresses which are numbered sequentially, and the segment can be identified by its lowest page address. The axial directory must hold, at least, this lowest page address and an appropriate value (say the lowest or highest) of the corresponding attribute, for each segment orthogonal to the axis. Thus, the axial directory is normally used to return the address of a segment containing a given value of the attribute.

With this structure, the segment addresses need not be in the same order as the attribute values: the axial directory can permute the segments from their ordering, in the data space, by the values of the attribute. Thus, when we come to split some arbitrary segment in the middle of the data space, we can always consider that the new segment is created on the outer face of the hyper-rectangle, that is, at the face which is furthest along the axis from the origin.

The multipaged hyper-rectangle thus grows dynamically always at its outer faces, and the problem of how the data space grows is equivalent to the problem of how to extend a d-dimensional array. For this, it is sufficient to have axial directories which record the lowest page address of each element as it is created. The above is a summary. A textbook introduction is given by Merrett [1984]. Each segment is a (d-1)-dimensional array, and, in order to address pages within the segment, we can use the usual row-major or column-major addressing formulas. For either, we need to know (d-2) of the dimensions of the segment. Merrett and Otoo [1982] proposed storing these (d-2) values in the axial directory, requiring $O(d^2 n^{\frac{1}{d}})$ storage for the d directories. We show here how the segment dimensions can be deduced from the lowest page addresses of the segments that preceeded it chronologically in the growth of the file.

Extensible Arrays

We now give and discuss the algorithm to compute the address of an element, $(i,j,k,..)$, of an extensible array. The algorithm has four steps.

1. Find the base address of the segment containing the element. This is done as described in earlier publications, using the base addresses stored in the axial directories: the trick is to recognize that of all d possible segments which could contain the element (i.e., one orthogonal to each axis), the correct segment will be the one with the largest base address, since this is the latest segment that could have been created containing the element.
2. Find the dimensions of the segment with this base address. This is the new part, and the insight required is that, when the segment was created, its dimensions were set equal to those of the extensible array on all axes except the one orthogonal to the segment.
3. Use the dimensions and a suitable array-addressing rule, such as column-major order, to compute the offset address of the element within the segment.

4. The address of the element is the base plus the offset.

We can give the algorithm fully using Aldat notation (see chapters 2.1 and 3.1 of [Merrett 1984]). This is much more convenient than subscript notations for this problem, and for readers unfamiliar with it, we work an example fully.

Algorithm Extensible Array Address

We are given the relations AXDIR(AXIS,INDEX,SEGBASE), containing all axial directories, and SEARCH(AXIS,INDEX), containing the coordinates of the element required. Element $(i,j,k,..)$ corresponds to AXIS=1 **and** INDEX=i, AXIS=2 **and** INDEX=j, etc. SEGBASE contains the base address of the segment at position INDEX orthogonal to AXIS. Indices start from 0.

1. *Base Address*
   let AXIS′ **be** AXIS <<rename>>
   let BASE **be red max of** SEGBASE
   <<find max. base for segment sought>>
   A ← AXIS′,BASE **where** SEGBASE=BASE **in** (AXDIR **ijoin** SEARCH)
2. *Dimensions*
   let SUBMAX **be equiv max of**
   (**if** SEGBASE≤BASE **then** SEGBASE **else** -infinity) **by** AXIS
   <<find last segments added to each other axis prior to segment sought>>
   B ← AXIS′,AXIS,INDEX,SUBMAX **in** (AXDIR **ijoin** A)
   <<cartesian product>>
   let DIM **be if** AXIS=AXIS' **then** 1 **else** INDEX+1
   <<dimensions are positions of these other segments>>
   C ← AXIS,DIM **where** AXIS≠AXIS′ **in** B
3. *Offset*
   <<using rule $i + d_i j + d_i d_j k + ..$, where $d_i d_j$,.. are the dimensions>>
   let CO **be fcn** × **of** DIM **order** AXIS <<product of dimensions>>
   let COEF **be if** AXIS<(**red max of** AXIS) **then** CO **else** 1
   let COEFF **be fcn pred of** COEF **order** AXIS <<cyclic permute>>
   let OFFSET **be red** + **of** INDEX × COEFF
   D ← OFFSET **in** C
4. *Address*
   let ADDRESS **be** BASE + OFFSET
   RESULT ← ADDRESS **in** (A **ijoin** D) <<cartesian product>>

This algorithm can be shortened, but it is spelled out here so that the data in Figure 2, below, can be used to follow it in detail. Figure 1 shows the extensible array, with segments enclosed in rectangles and with all element addresses shown. The rectangles are also numbered in order of creation for the reader's benefit, although the order of creation can be seen from the sizes and positions of the rectangles, and the numbering is redundant. Because the array is three-dimensional, some of the segments stick out of the page: connections across the two values, k=0 and k=1, are shown. The axial directories are enclosed in dashed rectangles: pointers from the directories to the main array are not shown explicitly.

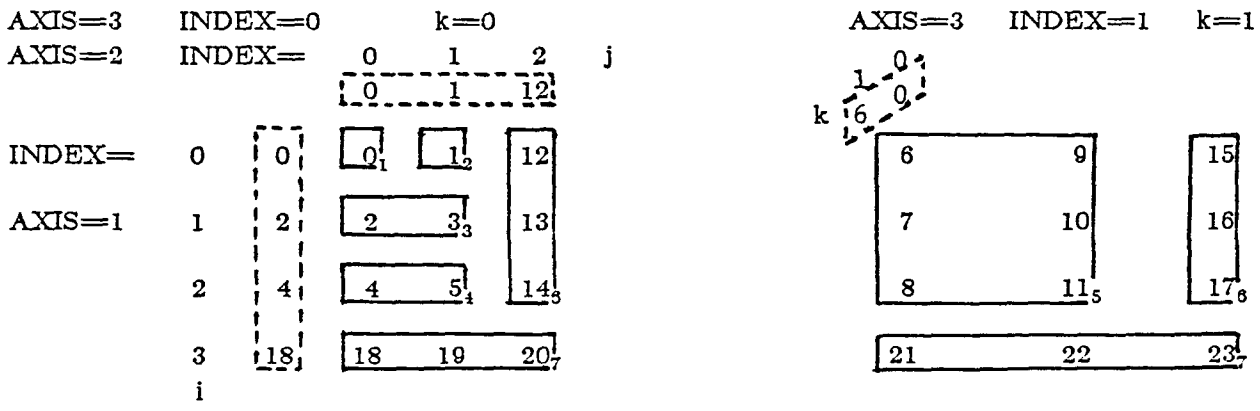


Figure 1. Extensible Array with Axial Directories

Figure 2 shows the relation AXDIR corresponding to the axial directories of Figure 1. It also shows SEARCH for the element $(i,j,k)=(1,2,1)$ and all the relations and attributes calculated in the algorithm. Domain algebra attributes are shown outside the relation parentheses to show virtual values before they are actualized. The final offset addressing rule is $i+3k$.

AXDIR

(AXIS	INDEX	SEGBASE)
1	0	0
1	1	2
1	2	4
1	3	18
2	0	0
2	1	1
2	2	12
3	0	0
3	1	6

SEARCH

(AXIS	INDEX)
1	1
2	2
3	1

A

(AXIS'	BASE)
2	12

B

(AXIS'	AXIS	INDEX	SUBMAX)	DIM
2	1	2	4	3
2	2	2	12	1
2	3	1	6	2

C

(AXIS	DIM)	CO	COEF	COEFF
1	3	3	3	1
3	2	6	1	3

D

(OFFSET)
4

RESULT

(ADDRESS)
16

Figure 2. The Relations Given to and Calculated by the Algorithm

We now walk through the algorithm step by step, using this example. As Figure 1 shows, the extensible array is assumed to have been built up as follows

the segment containing page 0	was added 1st giving a 1×1×1 array
the segment containing page 1	was added 2nd giving a 1×2×1 array
the segment containing pages 2,3	was added 3rd giving a 2×2×1 array
the segment containing pages 4,5	was added 4th giving a 3×2×1 array
the segment containing pages 6-11	was added 5th giving a 3×2×2 array
the segment containing pages 12-17	was added 6th giving a 3×3×2 array
the segment containing pages 18-23	was added 7th giving a 4×3×2 array

The final result has three axial directories, of lengths 4, 3 and 2, respectively, containing the base addresses of the seven segments (the first segment has base address 0, which is stored as the first element of all three axial directories). These directories are stored together as the nine tuples of the relation AXDIR in Figure 2.

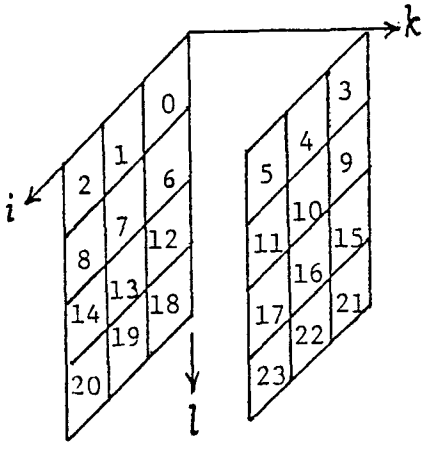
The algorithm computes the address (page number) of a page with given coordinates, which, in this example, are taken to be (1,2,1). These are found in the relation SEARCH. Step 1 is to compute the base address of the segment containing the page at position (1,2,1). This is the largest of the three axial directory entries at i=1, j=2 and k=3, respectively. It is computed as the largest entry (red max of SEGBASE) in the natural join (ijoin) of AXDIR with SEARCH. The result is 12, and corresponds to AXIS (renamed to AXIS') 2, i.e., j.

Step 2 is to calculate what the dimensions of the extensible array were when the segment at j=2 (the 5th segment) was formed. Say these dimensions are $d_i \times d_j \times d_k$. The dimensions of the segment are then $d_i \times 1 \times d_k$. The computation is done by finding, for each AXIS, the largest directory entry less than the BASE of 12 we found in the last step. (equiv max of (if SEGBASE $\leq$ BASE then SEGBASE else -infinity) by AXIS). To do the comparison with BASE, it must be appended to the relation AXDIR by cartesian product (ijoin with no common attributes). This gives values of 4 for i=2, 6 for k=1 and, of course, 12 for j=2, which can be found in relation B. Relation C simply shows the dimensions of the segment as 1 for j and 1 + the indices found for i and k: 3 × 1 × 2.

Step 3 evaluates the multinomial which combines the dimensions and the search indices to calculate the offset address within the segment. CO is the cumulative product of the dimensions, COEF is the same as CO except for the last axis (k), where it is 1, and COEFF is the same as COEF, but cyclically permuted among the axes. Finally, offset is the multinomial (red + of INDEX × COEFF). This is all evaluated over relation C and the result, 4, assigned to relation D.

Step 4 finally adds the base and the offset to compute the address, 16. Figure 1 confirms that the page at position (1,2,1) is indeed page 16 (offset 4 within the segment based at page 12).

In Figure 3 we show the offset calculation for a three-dimensional segment, since the two-dimensional 2×3 segment (6) in Figures 1 and 2 is too simple to show the effect of the algorithm.



	(AXIS	DIM)	CO	COEF	COEFF
i	1	3	3	3	1
k	3	2	6	6	3
l	4	4	24	1	6

offset $= i + 3k + 3 \times 2l$

Figure 3. Three-dimensional Offset

Conclusions

We have improved the segment addressing techniques for extensible multidimensional arrays, making them still more compact and simplifying the calculation of offset addresses. This result is directly applicable to dynamic multipaging.

Acknowledgements

This work was supported under grants A4356 from the Natural Science and Engineering Research Council of Canada and EQ2561 of the Fonds formation de chercheurs et l'aide à la recherche of the Province of Quebec.

References

1. R. Bayer & E. M. McCreight, 1972. Organization and maintenance of large ordered indices. Acta Informatica 1 3, 173-89.
2. W. Litwin, 1980. Linear hashing: a new tool for file and table addressing. Proc. VLDB6, 212-23.
3. T. H. Merrett, 1978. Multidimensional paging for efficient database querying. Proc. ICMOD '78, Internat. Conf. on Data Base Management Systems, FAST, Milano, Italy (29-30 June, 1978) 277-90.
4. T. H. Merrett, 1984. *Relational Information Systems.* Reston Publishing Co., Reston, VA.
5. T. H. Merrett & E. J. Otoo, 1982. Dynamic multipaging: a storage structure for large shared data banks in Peter Scheuermann ed. *Improving Database Usability and Responsiveness,* Academic Press, 237-55.
6. J Nievergelt, H. Hinterberger & K. C. Sevcik, 1984. The grid file: an adaptable symmetric multikey file structure. ACM Trans. on Database Syst. 9 1, 38-78.
7. E. J. Otoo & T. H. Merrett, 1983. A storage scheme for extendible arrays. Computing 21. 1-9.
8. A.L. Rosenberg, 1974. Allocating storage for extendible arrays. J. ACM 21 4,652-70.
9. A.L. Rosenberg, 1975. Managing storage for extendible arrays. SIAM Journal of Comp. 4 5, 287-306.
10. A.L. Rosenberg & L.J. Stockmeyer, 1977a. Hashing schemes for extendible arrays. J. ACM 24 2, 199-221.
11. A.L. Rosenberg & L.J. Stockmeyer, 1976b. Storage schemes for boundedly extendible arrays. Acta Informatica 7 3, 289-303.