The BANG file: a new kind of grid file

Michael Freeston

ECRC
European Computer-Industry Research Centre
Arabellastr 17
D-8000 Munchen 81
West Germany

Abstract

A new multi-dimensional file structure has been developed in the course of a project to devise ways of improving the support for interactive queries to databases and knowledge bases Christened the 'BANG' file - a Balanced And Nested Grid - the new structure is of the 'grid file' type, but is fundamentally different from previous grid file designs in that it does not share their common underlying properties It has a tree-structured directory which has the self-balancing property of a B-tree and which, in contrast to previous designs, always expands at the same rate as the data, whatever the form of the data distribution Its partitioning strategy both accurately reflects the clustering of points in the data space, and is flexible enough to adapt gracefully to changes in the distribution

1 Research directions

The eventual aim of the research reported here is to devise appropriate support structures for knowledge bases The immediate practical objective towards that end has been to find a more powerful and flexible replacement for the INGRES structures on which the logic programming/database system EDUCE [BOCC85] currently relies EDUCE in turn supports the KBMS Prolog-KB2 [WALL86] Of course, the specification of an 'ideal' knowledge base support structure will change as knowledge bases develop, but it seems clear that it must for the present include at least all the features of the 'ideal' structure for a conventional relational database

But the structures which support current relational databases are clearly not ideal Specifically, they are still almost without exception one-dimensional, so that access to a file via more than one key involves multiple indexes with associated update overheads Such structures may be adequate to support menu-driven user interfaces, for which the form of the allowed queries is entirely controllable and predictable, but they are a very poor match for an interactive query language This deficiency rapidly becomes clear to anyone who uses an interactive query language, and over the past ten years or so there has been a variety of attempts to improve the situation by the design of new multi-dimensional file structures and access mechanisms The guiding principle of all these designs has been that the response to a query should depend only on the complexity of the query, and not simply on the particular combination of attributes named in it

Research in this direction has followed three distinct lines

1 Tree structures a generalisation to n dimensions [BENT79 ROBI81 GARD83]

2 Multi-dimensional extendible hashing [FAGI79 OUKS83 OTOO85]

3 Grid files an explicitly geometric approach, representing n-tuples as points in a partitioned, n-dimensional hyperspace [NIEV81 BURK83 HINR85 OZKA85]

A detailed review of recent work in these areas [FREE86] was encouraging, to the extent that a number of novel approaches have been demonstrated to be effective for certain classes of application But all appeared to have some serious drawback as a general database support structure Of the various possibilities, the grid file approach seemed to have more potential than the others for further innovation and improvement

However, in the course of a subsequent study a new design emerged which, although of the grid file type, is sufficiently different in its characteristics and performance to distinguish itself as a new file structure It has been christened the BANG file a Balanced And Nested Grid file Its most important feature is that, in contrast to previous grid file designs, its directory always expands at the same rate as the data, whatever the data distribution As a structure for representing spatial objects, it has a partitioning strategy which accurately reflects the clustering of points in the data space while maintaining a high density of data storage

2. Principles of grid files

In order to appreciate the distinction between the BANG file and previous designs of the grid file type, it is first necessary to return to basic principles In a grid file, the tuples of an n-ary relation are represented as points in an n-dimensional hyperspace, the dimensions of which are the domains of the n attributes The representational problem is how to allocate the tuples in this hyperspace to a linear set of disk blocks In geometric terms the obvious thing to do is to divide up the dataspace into a set of hyperrectangles, each of which corresponds to a disk block Note however that there is no reason in principle why these subspaces should be rectangular But common sense (which we should remind ourselves is not always to be relied on) suggests that the subspaces should be bounded by planar surfaces, each of which is orthogonal to the axis of a domain This still does not mean that the subspaces must be rectangular They could contain concavities Nor does it follow that any of the planes in the division must bisect the entire data space (figure 2-1a)

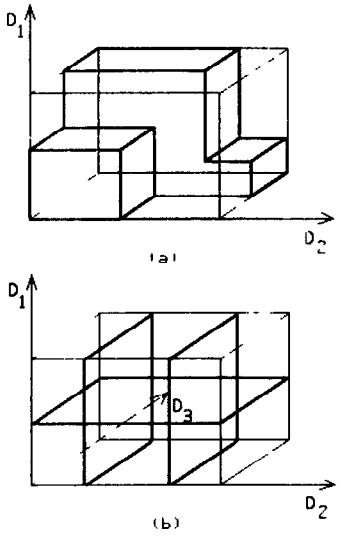

Figure 2-1 Partitioning the data space

Nevertheless, let us begin with the conceptually simpler assumption that the data space is to be divided into hyperrectangles by partitioning each dimension as shown in figure 2-1b The immediate questions which arise are how many partitions should be set up, and where should they be placed? If each hyperrectangle corresponds to one disk block, then clearly partitions must be so arranged that no hyperrectangle contains more tuples than can be contained in one disk block And if the search for a tuple is not to favour one key attribute over another, then the number of disk blocks to be searched should be the same, whichever dimension the key lies in This implies that the number of partitions in each dimension should be the same, if possible When the addition of a tuple would cause a disk block to overflow, then either the hyperrectangle corresponding to it must be divided, by adding an additional partition, or the previously defined partition boundaries must be moved, or both

2 1 The scale-based grid file

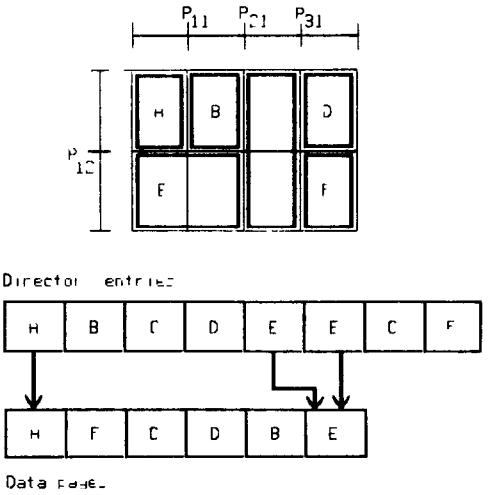

Figure 2-2 Structure of the SG grid file

The first and best known grid file structure was proposed by Nievergelt et al [NIEV81], and implemented by Hinrichs [HINR85] It is generally known as 'The Grid File', but since we need to distinguish it from subsequent grid file designs, it will be referred to here as the *scale-based grid file* or 'SG' file In the SG file a set of linear scales is used, one for each dimension, to define the positions of the *grid regions* or *grid partitions* of the data space (figure 2-2)

There is a directory entry for each grid region The directory is stored on a contiguous sequence of disk blocks, but is logically organised as a linear array of

pointers to corresponding disk buckets[1] in the data file The mapping between the set of scale values defining the co-ordinates of a grid region and the corresponding directory element is the familiar one from an n-dimensional to a one dimensional array, subsequently mapped on to the disk blocks

Each disk bucket in the data file contains the records lying within one grid region of the data space, or several adjacent regions In the latter case there will therefore be several directory elements pointing to the same bucket[2] (figure 2-2) This situation arises because, when a bucket containing only one grid region overflows, the region as well as the bucket must be split into two This means that a new partition must be introduced, which splits **all** the regions lying across its plane However, only the overflowing bucket is physically divided into two, with a directory pointer to each For all the others, the newly created region points to the same bucket as the region from which it was split

This is the fundamental weakness of the SG design As the data distribution becomes less uniform, the ratio of directory entries (grid regions) to data buckets increases and the directory expansion approaches an exponential rate Worse, most of these directory entries point to completely empty block regions (represented by nul pointers), and the problem is magnified by the number of dimensions of the file The introduction of a multi-level directory greatly improves the situation for non-uniform data distributions, provided that they do not involve highly correlated or functionally related data [HINR85, FREE86] In such cases however the expansion rate is still exponential

2 2 Interpolation-based grid files

Interpolation-based grid files avoid this particular problem by using a representation of the data space in which block regions are represented explicitly, so that there is always only *one directory entry* per *data bucket* To achieve this, the data space is divided into a hierarchy of sets of notional grid regions, each of which is identified by a unique number pair (r,l) where r is the region number, and l is the granularity or level number The domain of each dimension is divided into equal intervals by binary partitioning, and each level of the hierarchy of grid regions is generated from the previous higher level[3] by a further level of binary partitioning in one selected dimension It is essential that the dimensions are partitioned in a fixed (most simply cyclic) sequence For example, in two dimensions, the partition hierarchy could be as shown in figure 2-3

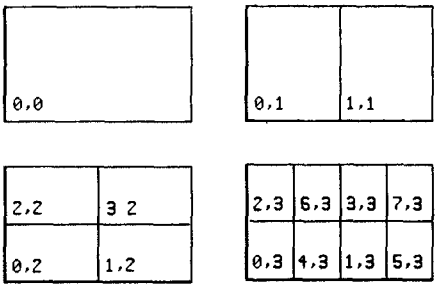

Figure 2-3 Numbering scheme for grid regions

Thus region (0,0) represents the whole data space, containing regions (0,1) and (1,1), and through subsequent levels the regions contained within them With this representation the data space can be partitioned explicitly at varying levels of granularity appropriate to the distribution of data points, and each region is a block region i e it corresponds to a single data bucket[4] As an example, figure 2-4 shows a space which has been partitioned to accomodate a non-uniform data distribution in six disk buckets

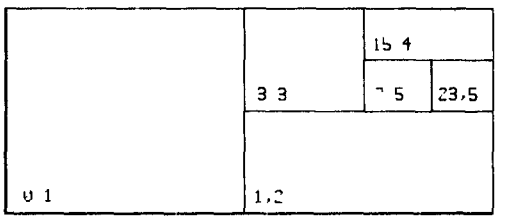

Figure 2-4 Block regions partitioning a non-uniform data distribution

The directory contains the set of region identifiers (number pairs) for the regions into which the data

[1]The word 'bucket' is used here rather than 'block' to emphasise that the contents of a disk bucket is *unordered*

[2]The union of all the regions represented in a disk bucket is termed a 'block region'

[3]Note that 'higher' or 'higher level' is taken to mean 'closer to the root' when referring to a hierarchy of region levels, but that the 'highest level' - the root - has the 'lowest level number' (zero) To try to avoid confusion in the following discussion, 'level numbers' are always referred to explicitly as such, not just as 'levels'

[4]In this context, a 'grid region' is simply a block region at the lowest partition level

space has been partitioned, together with pointers to the corresponding data buckets The representation outlined above was adopted by Ozkarahan and Ouksel [OZKA85] in their DYOP grid file (Dynamic and Order Preserving Partitioning), developed from the SG grid file and the work of Burkhard [BURK83] on interpolation-based hashing schemes Unfortunately there appear to be no published performance figures for the DYOP file The main advantage claimed for it over the SG grid file is that its directory is relatively well-behaved with non-uniform data distributions

However, when the data in a region overflows, there is no choice in the subsequent partitioning sequence, which may create empty regions For example, the partitions would be as shown in figure 2-4 even if all the data were confined to region (23,5) All the other regions would be empty, but would be represented by nul entries in the directory This problem is actually exacerbated by the directory structure proposed for the DYOP file - a tree structure in which each of the upper levels represents a partitioning of partitions - because a non-uniform data distribution will generate a non-uniform distribution of partitions and so on up the tree

Note also that the strategy for recombining DYOP regions is completely pre-determined Fundamentally, both the SG and DYOP files use the same strategy recombination is restricted to Buddies[5] But the DYOP scheme for numbering the regions dictates that a region can only recombine with the Buddy from which it was originally split - whereas a region in an n-dimensional SG file can in principle recombine with any of its n Buddies The adaptive ability of the DYOP file must therefore be much less than that of the SG file This will tend to reduce the average region/data bucket population of the DYOP file, and increase the size of its directory relative to a given set of data

These considerations suggest that the apparent advantage of the DYOP file over the SG file in representing block regions rather than grid regions will not generally be as great as first appears, and that in a dynamic environment of non-uniform data, its advantage could be lost altogether

[5]When a region is split by binary partitioning, the two resulting regions are the *Buddies* of each other

3 The BANG file

3.1 Axioms

The BANG file is an interpolation-based grid file, in that it partitions the data space into block regions by successive binary division, and uses the same numbering scheme for the block regions as the DYOP file But its representation of the data space is fundamentally different from that of the SG and DYOP files, which may appear quite different from each other, but are founded on the same set of axioms The BANG file differs in one crucial axiom In the SG and DYOP files

1 *The union of all the sub-spaces into which the data space has been partitioned must span the data space*

   i e there must be no *unrecorded* regions It follows that empty regions must be recorded in the directory This is clear in the SG directory structure, but not so obvious in the DYOP directory,

2 *No two sub-spaces into which the data space has been partitioned may intersect*

   In the SG file this means that a grid region can only be associated with one block region (although a block region may enclose many grid regions) In the DYOP case it means that the directory' can never contain more than one block region enclosing any point in the data space

The BANG file adopts axiom 1 above, but not axiom 2, which is replaced by the axiom that

> *If two sub-spaces into which the data space has been partitioned intersect, then one of these sub-spaces completely encloses the other*

In other words, *nested* block regions are allowed

This simple change is the foundation on which the exceptional performance characteristics of the BANG file rest It permits an algorithm for *balancing* the distribution of data points between the block regions *by redistributing the regions*, which is the key to achieving more compact data files with non-uniform data distributions The balancing algorithm guarantees that, although axiom 1 above still holds, there are never any empty block regions The partitioning and recombination algorithms are more flexible than the DYOP file to respond to changes in the data distribution, but do not incur the potentially heavy computational overheads of the SG file partitioning strategy and recombination deadlock checking algorithm

3 1 1 The grid mapping function

In interpolation-based grid files, a set of hash functions must be found which map the coordinates of a point in the data space to the number of the grid region in which it lies, at each of the defined levels of granulation However, before these hash functions can be applied, the set of n key values $(k_1,k_2,\quad,k_n)$ in the data record must be transformed to a set of n coordinates $(d_1,d_2,\quad,d_n)$ of the grid region enclosing its position in the data space This is straightforward, provided that it is possible to express each key value k_i as a fraction f_i of the domain D_i of the key variable

$$f_i = k_i/|D|$$

If the partial level of dimension i is l_i, then the corresponding key coordinate d_{i,l_i} is easily found as

$$d_{i,l_i} = \lfloor f_i 2^{l_i} \rfloor \qquad for\ 0 <= i <= n-1$$

and the coordinate at a higher partition level j_i is simply

$$d_{i,j_i} = d_{i,l_i} / 2^{(l_i - j_i)} \quad for\ 0 <= j_i <= l_i$$

Thus if the coordinates are computed for the lowest current partition level in each dimension, the source keys are not required again to compute the coordinates of the higher level regions

The mapping from the coordinates to the region number could in principle be performed in a variety of different ways, but the numbering scheme shown in figure 2-3 makes the mapping particularly simple It is based on the observations that

- level $(l+1)$ is generated from level l by dividing each grid region into two in some chosen dimension,
- the total number of grid regions at level l is 2^l,
- the grid region r at level l can be divided into two uniquely numbered regions r and $r+2^l$ at level $(l+1)$,
- the two numbers r and $r+2^l$ are the two possible extensions of the binary representation of r by one bit (the most significant),
- when the partial level of dimension i is increased from l_i to (l_i+1), the domain of coordinate i is doubled, by extending the binary representation of the coordinate by one bit (the least significant),

It follows from these considerations that

1 if the (single) grid region at level l=0 is assigned number r=0, then every grid region number at level l, *for all* $l > 0$, can be represented by l bits,

2 if $s_{min}(k)$ is a function giving the minimum number of bits required for the binary representation of integer k, then

$$s_{min}(d_{i,l_i}) = 1 \qquad for\ all\ l_i = 0$$

$$s_{min}(d_{i,l_i}) = l_i \qquad for\ all\ l_i > 0$$

$$(i=1,\quad,n)$$

However, it is more convenient for algorithmic purposes to define

$$s'_{min}(k) = s_{min}(k) \qquad for\ k > 0$$

$$s'_{min}(0) = 0$$

so that

$$s'_{min}(d_{i,l_i}) = l_i \qquad for\ all\ l_i \geq 0$$

$$(i=1,\quad,n)$$

and similarly

$$s'_{min}(r) = l \qquad for\ all\ l \geq 0$$

Then since

$$l = \sum_{i=1}^{n} l_i \qquad for\ all\ l_i \geq 0$$

it follows that

$$s'_{min}(r) = \sum_{i=1}^{n} s'_{min}(d_{i,l_i}) \qquad for\ all\ l_i \geq 0$$

This means that a mapping from a set of region coordinates to a unique corresponding region number can be obtained simply by concatenating the binary string representations of the coordinates in any predefined order of bits, but omitting coordinates at level 0 in any dimension

It is a prime objective, however, to select a representation in which the hierarchy of numbers of regions enclosing or enclosed by any region numbered r can be easily generated from r With this in mind, we note that it is conceptually and algorithmically simpler if the region numbers at level l remain as the "lower half" region numbers at level $l+1$ i e when a new partition is introduced in dimension i, each region is divided into an "upper half" region (odd coordinate $d_{i,(l+1)_i}$), and a "lower half" region (even coordinate

$d_{i,(l+1)_i}$) Referring back to the observations itemised earlier in this section, we see that this is easily achieved, by concatenating the least significant bit of the newly-formed coordinate in dimension i at level $l+1$ to the most significant bit of the corresponding region number at level l This generates one of two region numbers at level $l+1$ one is the same as that of the enclosing region at level l, the other is 2^l greater It follows that the region numbers of the hierarchy enclosing region r can be generated with equal ease by consecutively removing the most significant bit from r Figure 2-4 illustrates these relationships

It remains only to define the order in which the dimensions of the data space are partitioned, before a general mapping function can be constructed Assuming that there is no preferred attribute, then a cyclic partitioning through the dimensions is appropriate It may well be that in some dimensions the number of partitions will reach a limit dictated by the domain type, but this does not matter, provided that the next possible partition in the cyclic order is always selected, and the number of partitions l_i in each dimension i is recorded

Let $b_{i,j}$ be the j^{th} bit of coordinate d_i , taking the least significant bit as bit zero i e

$$l_i = \sum_{j=0}^{l_i-1} 2^j b_{i,j} \qquad (for\ l_i > 0)$$

$$d_i = 0 \qquad (for\ l_i = 0)$$

Then the mapping function $M(d,l)$ from a coordinate vector $d = <d_1,d_2, \quad ,d_n>$, in a data space partitioned $l = \sum_{i=1}^{n} l_i$ times, to the corresponding region number, is given by

$$r = M(d,l) = \sum_{i=1}^{n} \sum_{j=0}^{l_i-1} 2^x b_{i,j}$$

where

$$x = \sum_{k=1}^{n} \min\{l_k, p\}$$

$$such\ that\ p = j \qquad for\ k \leq i$$

$$p = j - 1 \qquad for\ k > i$$

The offset x is most easily understood in pictorial terms Figure 3-1 shows the equivalent mapping from a two-dimensional 'ragged' array of coordinate bit strings, to a one-dimensional array of bits

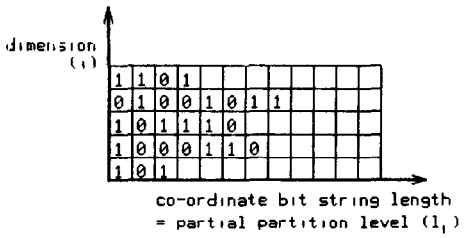

Figure 3-1. Mapping from coordinate vector to block region number

The corresponding mapping algorithm is particularly simple

```
r := 0,
offset := 1,
for k := 0 to (l - 1) do begin
   i := k mod n + 1,
   j := k div n,
   if l[i] > j then begin
      r := r * offset + b[i,j],
      offset := offset * 2
   end
end,
```

3 2 Representation of the data space logical regions

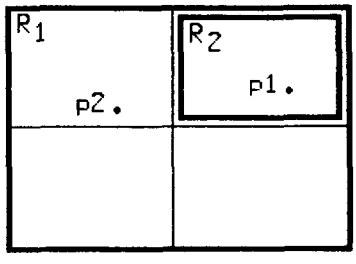

Figure 3-2 Nested partitions

Figure 3-2 shows a BANG file representation of a data space partitioned into two block regions R1 and R2 R1 encloses the entire data space, and R2 is enclosed or *nested* within R1 Such a configuration could not appear in an SG or DYOP representation, by axiom 2 of section 3 So what exactly does it mean in the BANG file?

For any space S partitioned into two block regions R1 and R2 such that R1 encloses R2, then R1 and R2 define two sub-spaces S1 and S2 S2 is the space enclosed by the boundaries of R2, S1 is the space enclosed by R1, *minus* S2 Therefore R2 alone defines S2, but *both* R1 and R2 are necessary to define S1 In general, every subspace of the BANG file is defined by a *set* of block regions One member of the set represents a rectangular space which encloses all the other members of the set These enclosed block regions represent subtractions from the enclosing block region

The sub-spaces into which the data space of the BANG file is partitioned are therefore defined by block regions, but are not necessarily block regions themselves In other words, *the sub-spaces do not have to be hyperrectangles* They can contain concavities, and have internal as well as external boundaries - contrary to the assumptions on which the designs of the SG and DYOP grid files are based It is even possible for a sub-space to be disjoint i e to be composed of subspaces which do not intersect and which have no common boundary

The sub-spaces and the sets representing them are however purely logical constructs (For this reason they are subsequently referred to as *logical regions*) Only block regions are represented explicitly in the directory, but each data bucket holds the tuples mapping to a logical region, not to a block region And the partitioning algorithm attempts to balance the distribution of tuples between logical regions Each directory page simply contains a sequence of block region identifiers, but because the identifiers are stored and searched in order of increasing partition level, no ambiguity arises in the association of tuples with logical regions and corresponding data buckets

For example, suppose that a tuple is to be added to the file whose current state is represented by figure 3-2, and that the tuple maps to point p1 in the data space Let R1 and R2 now be *logical* regions Then R2 is represented in the directory by the entry [3,2], and R1 is represented by the set { [0,0], [3,2] } Therefore the directory actually contains the two entries [3,2], [0,0] in that order (Note that, since R2 is at the current lowest level of partitioning, it also represents a block region and a grid region) When the grid mapping function is applied to the key attributes of the tuple, it generates the grid region identifier [3,2] From this, the identifiers of the hierarchy of all possible block regions enclosing [3,2] can easily be generated if required, as described in section 3 1 1 In this case they are [1,1] and [0,0]

The directory is now searched for the identifier of the smallest *recorded* block region which encloses p1 In this simple example the smallest recorded region *is* the grid region, so the search halts at the first directory entry, which contains the pointer to the data bucket for logical region R2 In the case of a tuple mapping to point p2, however, its grid region identifier is [2,2] which is not recorded in the directory So the search through the directory continues until the first entry at a a higher level is encountered ([0,0] is the only such entry in this example) The identifier of the block region enclosing p2 at this higher level is then computed from the grid region identifier, and the search is resumed *for the new identifier* This procedure is repeated whenever the partition level changes, until eventually a matching entry must be found even if, as in this case, it is [0,0], representing the outermost block region

This example demonstrates that, although the directory does not contain any explicit information about logical regions (as opposed to block regions), there is no confusion in the placement of tuples in the correct logical region, because of the order of searching in the directory Although p1 is enclosed by logical region R1 as well as R2, and a match would be found in the directory at the higher level, the first match at the lower level prevents this happening

Exactly the same algorithm is applied when searching the file for a tuple except that, once the corresponding data bucket has been found, a linear search must be made within the bucket because the tuples are not ordered in the bucket

3.3 Partitioning and recombination

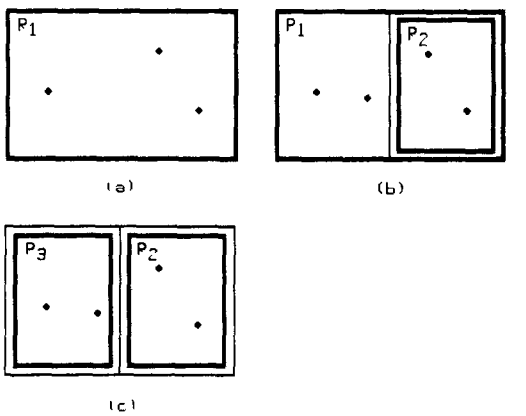

Figure 3-3: Creation of Buddies

The partitioning algorithm is invoked whenever the addition of a tuple causes a data bucket to overflow The algorithm essentially partitions the logical region corresponding to the overflowing bucket into two logical regions, one enclosing the other The procedure is recursive first the region is partitioned as shown in figure 3-3 (b), then the region containing more tuples than the other is progressively halved until the balance is reversed The best balance is then chosen If balance is achieved at the first level of division, then Buddy regions are created, (figure 3-3 (c)), as in the SG or DYOP files If not, then the external boundary of the partitioned logical region remains the same, but a new logical region is created within it

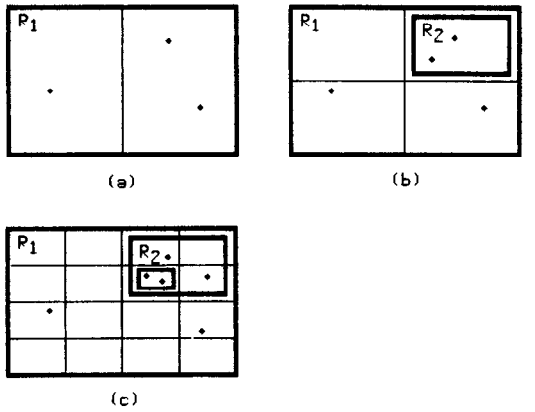

Figure 3-4: A sequence of partitions

Figure 3-4 shows the creation of a sequence of partitions as tuples are added Note that the partition configuration shown is not unique in general there are many possibilities Which one actually occurs depends on the *order* in which the tuples are added Figure 3-5 illustrates a refinement on the algorithm applied in figure 3-4 In figure 3-5 the splitting of an enclosed region is always treated as a continuation of a higher level split This sometimes allows a redistribution with the enclosing region (figure 3-5b), so that no new data bucket is created

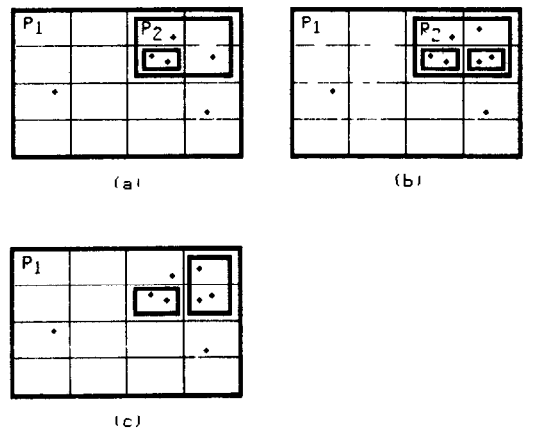

Figure 3-5 Redistribution of tuples

Recombination of logical regions proceeds on the principle of recombining smallest regions first, where possible If the population of a logical region sinks below some predefined minimum, an attempt is made to recombine it with one of the regions which it (immediately) encloses, starting with the smallest If this fails - either because recombination would cause overflow, or because there are no such regions - then an attempt is made to recombine the region with its Buddy If this fails, then a final attempt is made to recombine it with its (immediately) enclosing region which - if there is no Buddy - must always exist[6] The depleted region will be merged with the enclosed/Buddy/enclosing region provided this does not cause the population of the merged region to exceed some predefined maximum Note that no deadlock checking is required, and the BANG file does not have to recombine in the reverse order to partition, as does the DYOP file In particular, a region does not have to recombine with its Buddy It can therefore respond flexibly to changing data distributions

3 4 Directory structure

The complete BANG file directory is organised as a tree structure on the same principle as other grid files all levels above the lowest represent partitions of partitions When the first leaf node overflows, it is split into two according to exactly the same partitioning algorithm that is applied to the tuples That is, the block regions are treated as data points themselves, and are divided up into logical regions of the data space Each leaf node therefore contains the block regions which lie within a logical region of the directory space These logical regions can in turn be partitioned by logical regions at the next higher level in the tree, and so on to the root

The generalisation of the directory search algorithm to a tree structure is easy The same algorithm as was described in 3 2 for searching the leaf nodes is applied successively from the root down to the leaf Because the partitioning scheme is the same for the directory as for the data, the same balancing algorithm can be applied to all the nodes And since it is always updated from the leaf nodes upwards, it has a compact and balanced structure with very similar characteristics to the one-dimensional B-tree

3 5 Exact match, partial match and range queries

The general algorithm for performing searches in the BANG file is a modified depth-first traversal of those branches of the directory which satisfy the key Within each directory node the region identifiers are stored and searched in order of decreasing level number If an exact key does not match a directory entry at a particular level, then it cannot match any other entry at the same level, or lower Therefore, if a search succeeds at some level **L** but fails at some lower level, then if the search returns to level **L** it skips any subsequent entries of the same level number

Usually an exact match query will be answered in a single pass down the tree, but in the case of highly

[6] except of course when there is only one logical region

non-uniform data distributions a longer traversal is possible

For partial match queries, the same algorithm is applied, but using a key which contains a 'wild card' value in all the bit positions which correspond to unspecified attribute values In this case the query does not terminate until the tree traversal is complete

For range queries, the same tree traversal is followed, but the test for a matching directory entry is more complex, since it involves a combination of two keys, representing the top and bottom range values For a match between a specified range and a directory region, the two must intersect in every dimension of the data space The coordinates of the region are extracted from the region number by bit masking (effectively the reverse process to the original encoding of the region number), and compared with the range coordinates for intersection, one dimension at a time, until the intersection test fails in any dimension, or the match succeeds

3.6. Performance

On the basis of the figures published by Hinrichs [HINR85] for the SG file, it is possible to predict the performance of the BANG file and compare it with the SG file in similar circumstances Unfortunately there seem to be no figures available for correlated and functionally related data, where the advantage of the BANG file would show most clearly But a detailed analysis [FREE86] of the available figures suggests that it would under no circumstances be significantly inferior to the SG file On the other hand, Hinrich's case studies show that the BANG file becomes increasingly superior in terms of the rate of directory expansion and the efficiency of access operations as the data distribution becomes less uniform and/or the number of dimensions increases For example, it is argued in [FREE86] that the effect which a multi-level directory has on the SG file - of reducing the total directory size - is lost for correlated data It is therefore instructive to see how the SG and BANG directories compare as one-level structures in the case studies, even though the data is not correlated The analysis shows that the BANG directory is smaller by between one and two orders of magnitude

Acknowledgements

Thanks are due to Klaus Hinrichs for a valuable discussion of his implementation of the SG grid file, and for the use of results from his doctoral thesis Thanks to all at ECRC, but especially the KB group, for their encouragement and infectious enthusiasm The author is particularly grateful to Jean-Marie Nicolas for his patience and support during this research, and to Herve Gallaire for providing the opportunity to work in such a stimulating environment

References

BENT79a J L Bentley "Multidimensional Binary Search Trees in Database Applications" IEEE Trans on Soft Eng , Vol SE-5, No 4, July 1979

BENT79b J L Bentley, J H Friedman "Data Structures for Range Searching" ACM Computing Surveys, Vol 11, No 4, December 1979

BOCC85 J B Bocca "EDUCE - A Marriage of Convenience Prolog and a Relational DBMS" Proc of 3rd Symposium on Logic Programming, Salt Lake City, 1986

BURK83 W A Burkhard "Interpolation-Based Index Maintenance" Proc ACM SIGMOD-SIGACT Symposium, 1983

FAGI79 R Fagin, J Nievergelt, N Pippenger, H R Strong "Extendible Hashing A Fast Access Method for Dynamic Files" ACM-TODS, Vol 4, No 3, September 1979

FREE86 M W Freeston "Data Structures for Knowledge Bases Multi-Dimensional File Organisations" ECRC, Technical Report TR-KB-13, 1986

GARD83 G Gardarin, P Valduriez, Y Viemont "Les Arbres de Predicats" INRIA, Rapports de Recherche, No 203, April 1983

HINR85 K H Hinrichs "The grid file system implementation and case studies of applications" Doctoral Thesis Nr 7734, ETH Zurich, 1985

LEE 80 D T Lee, C K Wong "Quintary Trees A File Structure for Multidimensional Database Systems" ACM-TODS, Vol 5, No 3 September 1980

NIEV81 J Nievergelt, H Hintenberger, K C Sevcik "The Grid File an adaptable, symmetric multikey file structure" Internal Report No 46, Institut fuer Informatik, ETH Zurich, December 1981

OUKS83 M Ouksel, P Scheuermann "Storage Mapping for Multidimensional Linear Dynamic Hashing" Proc of 2nd Symposium on Principles of Database Systems, Atlanta, 1983

OTOO85 E J Otoo "A Multidimensional Digital Hashing Scheme for Files with Composite Keys" ACM 1985

OZKA85 E A Ozkarahan, M Ouksel "Dynamic and Order Preserving Data Partitioning for Database Machines" Proc of 11th Int Conf on Very Large Data Bases, Stockholm, August 1985

ROBI81 J T Robinson "The K-D-B-Tree A Search Structure for Large Multidimensional Dynamic Indexes" Proc ACM SIGMOD Conf 1981

WALL86 M G Wallace "KB2 A Knowledge Base System Embedded in Prolog" ECRC Technical Report TR-KB-12, August 1986