A General Solution of the n-dimensional B-tree Problem

Michael Freeston

European Computer-Industry Research Centre (ECRC)
Arabellastr. 17, D-81925 Munich, Germany

freeston@ecrc.de

Abstract

We present a generic solution to a problem which lies at the heart of the unpredictable worst-case performance characteristics of a wide class of multi-dimensional index designs: those which employ a recursive partitioning of the data space. We then show how this solution can produce modified designs with fully predictable and controllable worst-case characteristics. In particular, we show how the recursive partitioning of an n-dimensional dataspace can be represented in such a way that the characteristics of the one-dimensional B-tree are preserved in n dimensions, as far as is topologically possible i.e. a representation guaranteeing logarithmic access and update time, while also guaranteeing a one-third minimum occupancy of both data and index nodes.

1 Introduction

For twenty years researchers have tried to find a structure which generalises the properties of the B-tree to n dimensions i.e. an index on n attributes of a record instead of one. Ideally, such an index should have the property that, if values are specified for m out of n key attributes (a *partial match* query), then the time taken to find all the records matching this combination should be the same, whichever combination of m from n is chosen. To achieve this, the index must be symmetrical in n dimensions. There is no longer a directly defined ordering between the individual records according to their (single key) attribute values. Each record must be viewed as a point in an n-dimensional data space, which is the Cartesian product of the domains of the n index attributes. An n-dimensional generalisation of the B-tree must recursively partition this data space into subspaces or *regions* in such a way that the properties of the B-tree are preserved, as far as is topologically possible.

Specifically, the number of nodes encountered in an exact-match search of the tree (assuming no duplicates), or a single update, must be logarithmic in the total data occupancy of the tree; and the occupancy of each data or index region must not fall below a fixed minimum.

Unfortunately, it has proved extremely difficult to achieve this - apparently simple - objective. Considerable progress has been made, as shown by the substantial number of designs developed in the last few years. (For a review, see [Sam89]). But there remains an underlying inflexibility in current designs, and there is still no solution which is provably resistant to pathological cases i.e. which has totally predictable search and update characteristics in all circumstances.

In the past this was largely an academic issue. But now it is becoming a matter of life and death. Large databases control mission-critical and non-stop applications of every kind - from fly-by-wire aircraft control systems to electricity supply load balancing. There is no room any more for systems which work very well most of the time. They *must* be completely predictable *all* the time. This consideration provides justification enough for commercial database companies to cling to the B-tree. It may be inflexible, but it has the essential qualities of being totally predictable and fully dynamic. Whatever advantages an alternative may have, it must demonstrate the same qualities before it has any hope of replacing the B-tree.

A way around the problem is to map the co-ordinates of the points in a data space in to a linear order such as Z (or Morton) ordering [Ore86]. Then an ordinary B-tree can be used, and the worst-case characteristics of the B-tree are au-

SIGMOD ' 95,San Jose , CA USA

tomatically inherited. This approach has the additional practical advantage that it can be immediately applied in any system which supports B-trees. But there are also disadvantages. Possibly the most serious is that the method requires the representation of the whole data space i.e. there is no means of contracting the representation to a set of occupied subspaces. Comparative studies by [KSS+90] have clearly shown this to be a very significant factor in the efficiency of range queries.

Here, however, we are more concerned with a fundamental limitation of the method: it cannot support the direct representation of extended spatial objects (more precisely - the rectangular covers of the objects), without dividing an object into several parts. This introduces the uncontrollable update characteristics we are trying to avoid (and which, for example, the R^+ tree also shows). We showed in [Fre89b] how a dual point and object representation could overcome the worst-case access and update problems of the R-tree [Gut84] and R^+ tree [SRF87]. But the method we proposed can only guarantee its own worst-case characteristics if it rests on a solution of the n-dimensional B-tree problem.

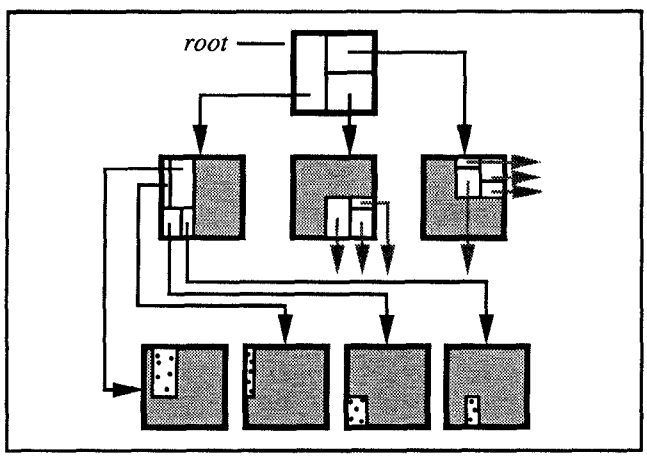

Figure 1-1: The K-D-B tree (after Robinson)

The apparent intractability of the underlying problem is clearly related to the increased topological complexity of the n-dimensional case. The approach described below therefore re-attacks the problem from a topological perspective. This clarifies the basic nature of the problem and, by so doing, reveals how it is possible to overcome a fundamental barrier to the achievement of acceptable and guaranteed performance characteristics. We present here only the basic ideas on which the treatment is founded. A full theoretical treatment is given in [Fre94].

The essence of the problem is how to recursively partition an n-dimensional data space so that every subspace maintains a certain minimum occupancy. Figure 1-1 shows an example of an early attempt: the K-D-B tree [Rob81]. The data space is recursively partitioned into rectangular subspaces, each of which corresponds to a memory page. The *k* co-ordinates of each point in a data page correspond to the *k* key attributes of a tuple in the page. Robinson did not specify how subspaces were to be represented in the index pages.

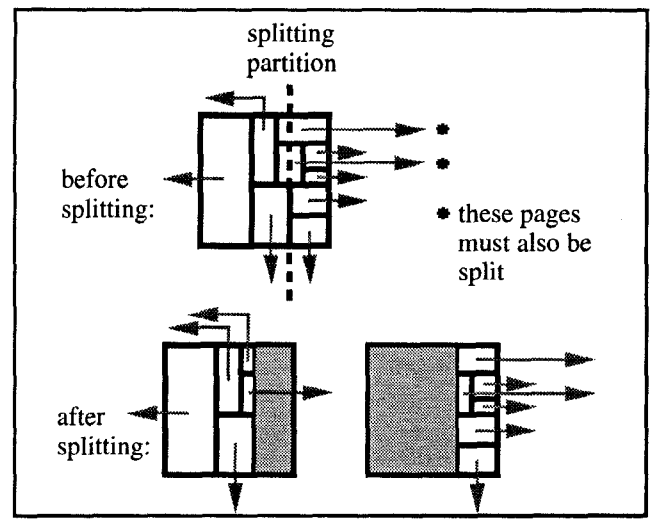

Figure 1-2: Partitioning a directory node of a K-D-B tree (after Robinson)

He used a sledge-hammer approach (fig 1-2), splitting both data and directory pages about an arbitrarily chosen point along an arbitrarily chosen partition axis. But figure 1-2 shows that it is in general not possible to split a directory page about a single partition, without also splitting entries within the page. This effect then cascades downwards through every lower level of the tree to the leaves. The consequences of a single insertion are thus wholly unpredictable. And since there is no choice in the position of the partition in any of the cascade of splits except the first, there is no way of maintaining any balance in the occupancy of the resulting pages, and hence no possibility of guaranteeing any minimal overall page occupancy. This in turn makes it impossible to predict the height of the directory tree for a given size of data set in the worst case. Deletion and redistribution also prove impractical.

The cascade splitting problem can only be avoided if every partition at any upper index level coincides with a partition at every level below. This severely restricts the parti-

tioning flexibility at the upper levels, whatever partitioning strategy is adopted at the data page level. Assuming, however, that some binary partitioning strategy is adopted for the data pages - which is universally the case in existing multi-dimensional indexing designs - then the same partitioning sequence can be followed in the higher index levels.

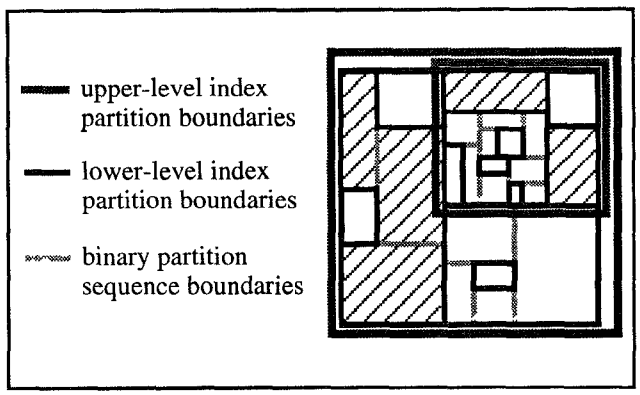

Figure 1-3: Enclosed subspaces through partition sequences

It is well known and easily proved (see, for example, [LS89]) that, by following such a binary partition sequence, it is always possible to partition a subspace into two parts such that each is at least 1/3 occupied. In general, however, such a partitioning sequence results in a configuration as shown in figure 1-3a i.e. one resulting subspace may *enclose* another . So it is necessary to choose a representation of subspaces which allows this. It is curious therefore that so few existing multi-dimensional index designs do so.

For example, the LSD tree [HSW89] and the Buddy tree [SK90] both avoid the cascade splitting problem by always splitting a directory page by the first partition in the binary splitting sequence - which is the only single partition about which the page can always be split. (The LSD tree is a little more flexible in the root page). But this is achieved at the price of abandoning all control over the occupancy of the resulting split index pages, and so both the average index page occupancy and the consequent height of the directory tree become unpredictable. In principle, the hB-tree [LS89] satisfies the representational requirement but, because it is actually a directed acyclic graph and not a tree, this leads to other problems which we consider separately elsewhere [Fre94].

The BANG file [Fre87, Fre89a] does allow the representation of enclosure, using a sequence of regular binary partitions to split both data and directory pages. However, having got that far, it then runs into a more fundamental paradox: although directory-level splitting follows the sequence of partitions from the levels below, the boundary of a directory page split may not coincide with any actual subspace boundaries at the level below. Figure 1-3 illustrates the problem. A set of subspaces, each corresponding to a data page, is to be partitioned into two subspaces at the index level above (i.e. the two subspaces with the widest boundary lines in the figure). But one member of this set - the subspace represented by the light shading - lies partly in *both* higher-level subspaces. (Note also that this subspace, whose outer boundary coincides with that of the whole data space, is composed of three disjoint parts - a consequence of allowing one subspace to enclose another).

So we seem to find ourselves back where we began with Robinson: if we force a split on the best-balance boundary of the higher index level, then we will be forced into a cascade of splits on the same boundary at the levels below. The real essence of the problem is thus not so much the maintenance of a minimum page occupancy, but the construction of a recursive set of subregions such that no subregion boundaries intersect, either within an index level, *or between levels*. If this can be achieved, then the 1/3 minimum page occupancy will follow.

2 A generic solution: the BV-tree

It has always been assumed that the generalisation of the B-tree to n dimensions would be a balanced tree. Otherwise, how could the access and update characteristics of the B-tree be preserved? But the solution which we present here is based on the realisation that such reasoning is false. The objective is to move from the particular to the general, and not vice-versa: we are looking for a structure which *specialises* to a B-tree in the one-dimensional case. There is no reason in principle why the generalised structure should be balanced.

To this insight we add a reminder: that, when we talk of the properties of a data structure, we mean the combined properties of the structure itself and a defined set of operations upon it. If the implementation of a set of operations on an unbalanced tree yields the same performance characteristics as a different implementation of the same set of operations on a balanced tree, then it does not matter that one of the trees is unbalanced. However, in order to claim a solution to the n-dimensional B-tree problem with an unbalanced tree, it must maintain the characteristics of the B-tree in n dimensions, and it must degenerate to a balanced tree in the one-dimensional case. We describe below a tree structure

which has this property. And the unbalanced structure of the general case makes the impossible possible, by creating the *effect* of splitting a region without *actually* splitting it. Since every new structure must have a name by which to refer to it, we have called this structure *the BV-tree.*

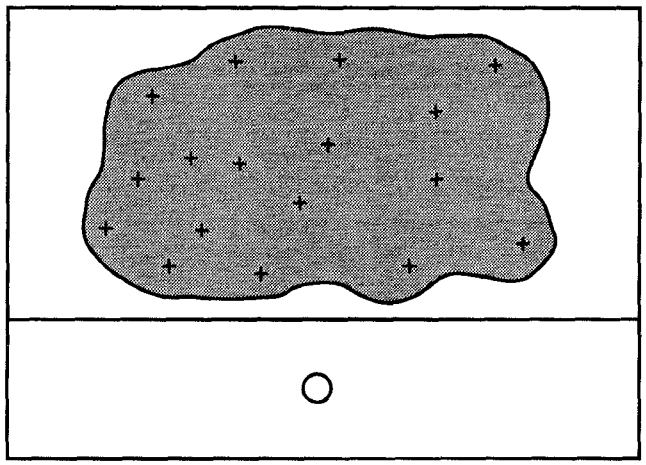

Figure 2-1a

Figures 2-1a to 2-1d show a sequence of recursive partitionings of a data space, and the corresponding BV-index structures. (The upper sections of each figure show the recursive partitions; the bottom section shows the corresponding index structure). No assumptions are made about the shape of the data space, or any of the subspaces into which it is partitioned, except that partition boundaries may not intersect. The data space may be finite or infinite. Initially, there is a single subspace or *region*, which is the whole data space. Conceptually, a number of points are inserted into this region (figure 2-1a). We make no assumptions about how the points are represented, only that there is a maximum number P which can be inserted into any one region. The single data page in the corresponding physical data structure is represented by a circle in the figure.

When the number of points inserted into a region exceeds P, the region must be split into two. As we have already seen, it is always possible to replace an overflowing region by two new regions, each of which contains at least 1/3 of the maximum number of points P allowed in a single region. Figure 2-1b shows a data space after the first overflow and split. An *index node* has been created which contains two entries a_0 and d_0, each of which is a unique key identifying one of the two resulting data regions. Each entry in the index is labelled with its *partition level number* (which we may from now on just call its *level*). This is because, in the BV-tree, there is no longer an automatic correspondence between the *partition hierarchy* and the *index tree hierarchy* which represents it. (From now on we will call a level in the index tree hierarchy an *index level* or *directory level*). The lowest level of the partition hierarchy - that of the data regions - is labelled as (partition) level 0. The lowest level of the index tree hierarchy (i.e. the leaf nodes of the index) is taken as index level 1. Thus a node at index level x represents a region at level x in the partition hierarchy. We will denote a region r at (partition) level x by r_x.

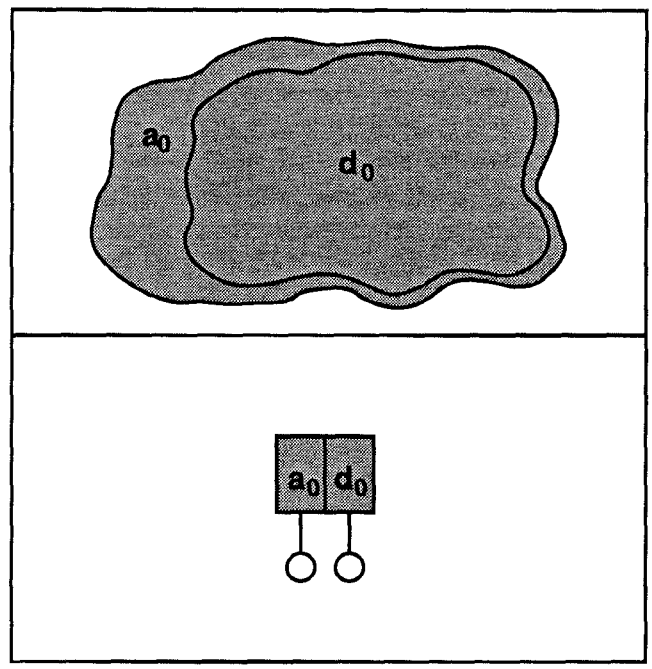

Figure 2-1b

We do not make any assumptions about the way in which the identifying keys of the regions are generated or represented. We assume only that there is an upper limit F (the *fan-out ratio*) to the number of entries which can be recorded in a single index node. The order shown for the index entries has no significance: in practice, the particular order chosen depends on the form of the representation of the index keys, and the efficiency of operations on this representation. Associated with each entry in the index (leaf) node is a pointer to the corresponding data node. For clarity, the data points themselves are not shown in subsequent figures.

Let us assume, without loss of generality, that the fan-out ratio F of the index nodes is four. Figure 2-1c shows the situation after the creation of three additional data regions. This causes the index node to overflow. The regions represented in the node are then partitioned into two *regions of regions* a_1 and b_1, which form the root node of a new, two-level index.

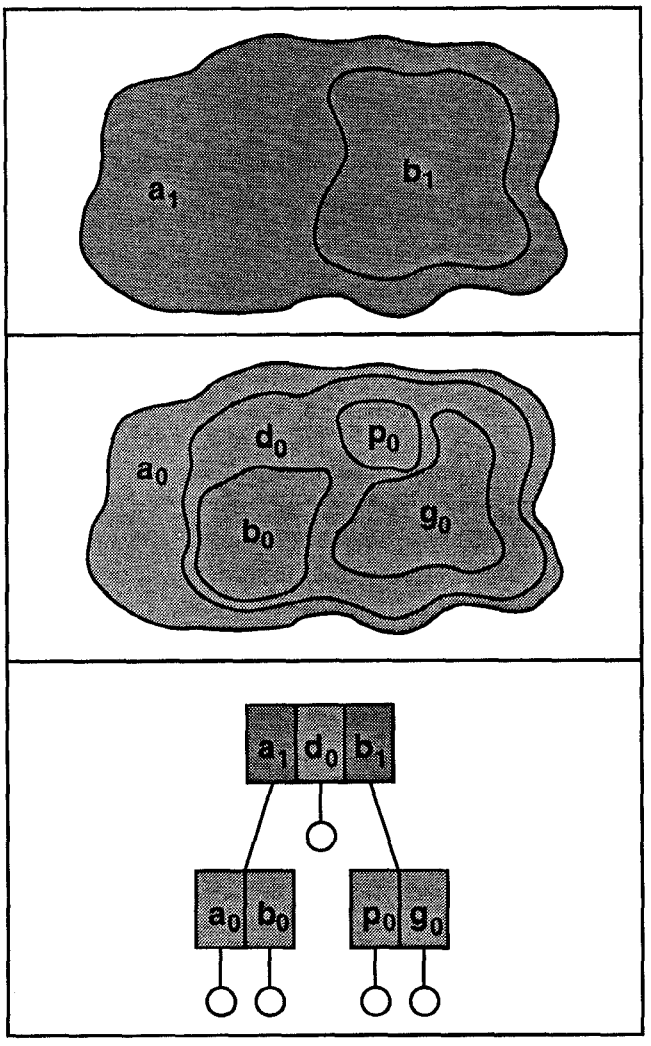

Figure 2-1c

The upper sections of figure 2-1c shows the regions represented at the upper and lower levels of the index. The bottom section of figure 2-1c shows that the index split has been accompanied by the *promotion* of entry d_0 to the index level above. This illustrates the general principle that, if there exists some region r_{x-1} whose boundary directly encloses the boundary of a region r_x resulting from a split, but r_{x-1} and r_x do not coincide, then r_{x-1} must be promoted to level x.

[N.B. region r_x *directly* encloses region s_y (where $x = y$ or $x < y$) if there exists no other region t_x such that $r_x \supset t_x \supset s_y$].

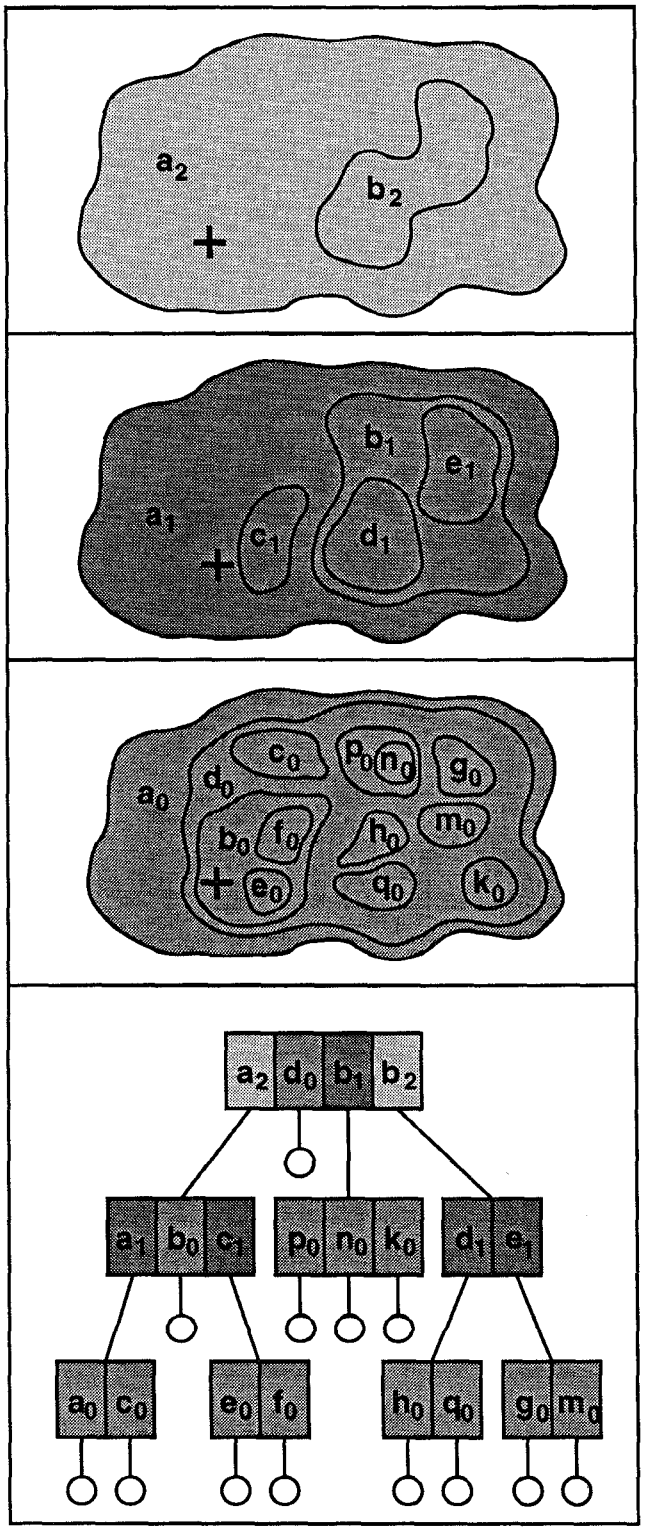

Figure 2-1d

This procedure sounds complex but is easy to implement. d_0 is the region which, if it were not promoted, we would be forced to split along the boundary of b_1. By avoiding such splits, we also avoid a possible further cascade of splits lower down the tree. By promoting d_0, we place it at the next higher branch point in the index tree (or higher - see below), so that it will be visited during any search of the index tree for a point stored on either side of this branch - *exactly as if it had been split into two at its original index level.* We term d_0 the *guard* of b_1. There is no confusion between guards and guarded within an index node, because every entry is labelled with its partition level. Note that, if a region d_x, $(x > 0)$ is promoted, then its subtree is automatically promoted with it.

Finally, figure 2-1d shows the result after several further splits, which have caused the creation of a third index level. In particular, the creation of the new root node has been accompanied by the promotion of both d_0 and b_1 from the level below. This illustrates a generalisation of the promotion principle given above, to regions already promoted. Suppose a region r_x has been promoted from level x to level $(x+1)$ and that, following a split at level $(x+1)$, r_x directly encloses one of the split regions. Then region r_x must be promoted further to level $(x+2)$. (see [Fre94] for proof). Thus, in the extreme case, a tree node at height x could contain $(x-1)$ entries of promoted guards for each unpromoted (level x) entry.

3 The exact-match search algorithm

All operations on the index tree are completely conventional and easily understood, provided it is borne in mind that the operations are to be performed on the partition hierarchy, and not the index hierarchy. The relevant index entries for a search of any partition level x are no longer all found at the corresponding index level $(x+1)$ but are, in general, accumulated during the descent to level $(x+1)$ of the index tree. As an illustration of this, we consider the exact-match search algorithm. In principle, this search simply descends the index tree from root to leaf in the usual way. What is unconventional is that the correspondence between the partition hierarchy and the index hierarchy is dynamically reconstituted as the search descends the tree i.e. the relevant promoted regions are effectively put back into their original, unpromoted positions in the partition hierarchy.

At the root of an index of height h, the set of all entries - both unpromoted and promoted - is searched for the best match with the search key. If the best match entry is unpromoted, then it may have a set of up to $(h-1)$ guards. If so, this *guard set* is temporarily stored, awaiting inclusion of its members in later searches further down the tree at the level from which they were originally promoted. Otherwise, if the best match entry is a guard, then the best-match unpromoted entry is that unpromoted entry which directly encloses this guard. Such an unpromoted entry must exist (see [Fre94]), and it may also directly enclose matching guards of other levels. These, plus the guard which is the best-match entry, become the guard set in this case.

The search then descends to level $(h-1)$, following the pointer from the best-match unpromoted entry. The search procedure within level $(h-1)$ is repeated as before, except that the best match in the search of the unpromoted entries of level $(h-1)$ is compared with (if it exists) the guard of level $(h-1)$ in the guard set brought down from the level above. If this guard proves to be the best match at level $(h-1)$, then its pointer is followed down to the next level.

If the guard of level $(h-1)$ in the guard set is not the best match at level $(h-1)$, then the pointer from the unpromoted best match at level $(h-1)$ is followed to the index level below. The guard set for level $(h-2)$ is created by merging the matching guards found at level $(h-1)$ with the set brought down from level h, discarding the member of level $(h-1)$ - if it exists. Two guards of the same level are merged by discarding the poorer match. Recall that, at level x of the index hierarchy, the set of matching guards contains at most $(x-1)$ members.

3.1 Example

Suppose that there is a data point located at position + in a data space partitioned by a three-level index as shown in figure 2-1d. A search to locate point + then proceeds as follows:

starting from the root, the best match entry for the key generated from data point + is guard d_0. The root contains no other guards which enclose the point + so, in this case, d_0 constitutes the entire guard set carried down to the next index level. The best match unpromoted entry is a_2. Following the pointer down from a_2, guard b_0 is the best match at index level 2 and, since it is a better match than d_0, the latter is discarded. The only matching unpromoted entry at index level 2 is a_1. Following the pointer down from a_1, the only match among the entries at index level 1 is a_0. (There can be no guards at index level 1). But the

guard b_0 - the only remaining member of the guard set - has now returned to its original index level , and must now be included in the search of that level. b_0 is a better match than a_0, and so the search finally succeeds in the region pointed to by b_0. Note that this represents a notional backtrack up the index tree, although no backtrack up the partition hierarchy has occurred, and no nodes of the index tree have been revisited.

4 Splitting of promoted regions

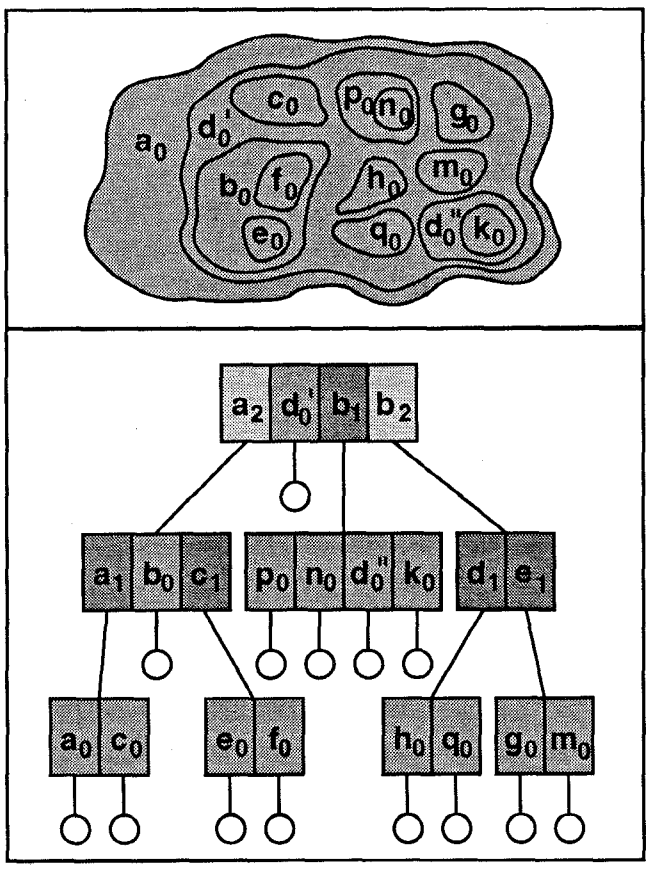

Figure 4-1

When a promoted region overflows and splits, it may be that either one or both of the resulting regions can be demoted by one or more index levels. Figure 4-1 illustrates the case where d_0 splits into d_0' and d_0'' , where $d_0' \supset d_0''$. (N.B. only the regions at the data level are illustrated in figure 4-1. The regions at the higher levels remain unchanged from those of figure 2-1d). d_0' stays promoted to the root index level, because it still directly encloses regions b_1 and b_2. d_0'' however does not enclose any higher level region - either directly or indirectly. It can therefore be demoted to its unpromoted position in the index tree as illustrated, using the exact-match search algorithm.

In general, however, it may be that a newly-split region d_x'' itself directly encloses a higher level region. So, during the exact-match search to insert d_x'', a check is made at each node visited to establish if there exists a higher level region r_{x+k} which is directly enclosed by d_x''. If so, then d_x'' is inserted in the node containing r_{x+k} . If there is already a promoted region r_x guarding r_{x+k} in the node then, if $r_x \supset d_x''$, d_x'' becomes the new guard and the old guard becomes the candidate for demotion. The demotion/insertion procedure thus continues down the tree with the new candidate. A special case of this arises when a region d_x splits into d_x' and d_x'', such that $d_x' \supset d_x''$ and such that d_x'' replaces d_x as the guard. d_x' is then demoted as detailed above. Note that this procedure requires only a single direct descent of the index tree. If the eventual insertion of a demoted region causes overflow in the index page, then this overflow may propagate upwards in the usual way.

We have, however, assumed that the split of a promoted region results in two regions, one of which encloses the other. This can always be taken to be the case, even if it is possible to reduce each of the resulting regions to a much smaller volume, where one region does *not* enclose the other. But if this is done, the situation could arise where *neither* of the regions into which a guard splits are themselves guards. Two regions must then be reinserted, and further reinsertions may be triggered while descending the tree during the reinsertion procedure. This is the kind of uncontrollable behaviour we are striving to avoid.

We therefore avoid the situation by always representing the result of an overflow as a partition in which one split region encloses the other. On any subsequent visit to a node, a check can be made to see if it is of minimal size. If it is not, it can be reduced. This may cause it to be demoted, but since it is a demotion without a split, there is only one region to demote, and thus only one descent of the tree to make.

5 Redistribution and deletion

Truly dynamic behaviour of multi-dimensional index structures has been as difficult to achieve as the combination of an acceptable lower limit to data and directory occupancy together with a logarithmic access and update guarantee. The

reason for this is that, fundamentally, the problem is the same. Redistribution of the contents of two data or directory pages - when one of them underflows - implies joining their contents together and then splitting them again. There are thus two problems: finding a region with which an underflowing region can be merged or redistributed; and finding a way of splitting the resulting merged region again.

The first problem essentially involves reversing the binary partitioning sequence. Many index designs have resolved this problem without too much complication, by imposing some restrictions on the sequence of both splitting and merging partitions. If enclosure of regions is allowed, then it is always easy to find a region r_x with which to merge an underflowing region s_x : if there exists an r_x which directly encloses s_x, then r_x and s_x can merge. If no region encloses s_x , then any region adjacent to s_x , which is also not enclosed by any other region, can be merged with s_x . (This is treated in detail in [Fre94]).

Thus the merge operation alone does not present a major difficulty. It is the possible subsequent need for redistribution which is the problem. And if redistribution cannot achieve a minimum page occupancy in both redistributed pages, then there is little point in attempting a merge, until no subsequent redistribution is necessary. But this means, of course, that no limit can be set on the minimum acceptable occupancy before underflow. Therefore the resolution of the splitting problem also clears the way for truly dynamic deletion in multi-dimensional indexes.

6 An unbalanced balanced tree

The BV-tree resolves a paradox: how to maintain a fixed direct path length from root to leaf of a tree in which the path length from root to leaf is not fixed. It is easy to see, from the exact-match search algorithm above, that the search always proceeds downwards in the partition hierarchy, although it may appear to backtrack through the index tree. The length of every exact-match search path from root to leaf of the index tree is therefore always equal to the height of the partition hierarchy. Actual backtracking through the index tree is avoided by using the guard set: the effect of this is to provide direct pointers to backtracking points at the roots of promoted subtrees, rather than physically backtracking through the index tree.

Although the occupancy of both index and data nodes can be maintained at a minimum of 33% in the BV-tree, this does not fix the height of the tree for a given size of data set. As we have shown, there can be, in the worst case, *(x-1)* promoted entries for every unpromoted entry in a branch node at index level *x*. Each of these *(x-1)* promoted entries is the root of a subtree of a different height, so we need to know what will be the maximum height of the index tree for a given size of data set in this worst case. We analyse this relationship in the next section.

In order to guarantee that the exact-match search path from root to leaf in the BV-tree will be no longer than that of a balanced tree with the same number of data nodes, the maximum allowed fan-out ratio of an index node at level *x* needs to be increased in size by a factor of *x*. This implies increasing the page size accordingly for higher level index nodes. (Note however that the occupancy of the index nodes will still not fall below 33%). It may well not be considered worthwhile to make allowance for this effect in practice. But at least it is possible to do so in a controlled manner, with predictable consequences, if necessary.

This is the only price which has to be paid for obtaining the characteristics of the B-tree in n-dimensions. The worst-case index node occupancy is only 33%, compared to 50% for the B-tree. But this is a consequence of topological considerations, rather than a limitation of a particular index design. More important is that this occupancy level is guaranteed for every index node under all circumstances, thus guaranteeing a minimum occupancy for the entire index.

7 Analysis of worst-case performance

7.1 Index with single page size: best case

We initially assume that all the pages representing the nodes of the index are of a fixed size *B* bytes, and that the individual keys within a page are of fixed size *b* bytes. The maximum number of keys *F* per page (the *fan-out ratio*) is thus: $F = B / b$. If there are no promoted subtrees then every node at index level x $(h \geq x > 1)$ of a tree of height *h* contains a maximum of *F* pointers to index nodes at the next index level below; and every node at index level *1* contains a maximum of *F* pointers to data nodes. Then, as in a B-tree, the maximum number of data nodes $td_{(h)}$ in a tree of height *h* is:

$$td_{(h)} = F^h \qquad (1)$$

... and the maximum number of index nodes is:

$$ti_{(h)} = \sum_{k=0}^{h-1} F^{k} = \frac{(F^{h} - 1)}{(F - 1)}$$

$$\approx F^{h-1} \ldots for\ F \gg 1 \qquad (2)$$

$$\frac{ti_{(h)}}{td_{(h)}} \approx \frac{1}{F} \quad \ldots for\ F \gg 1 \qquad (3)$$

7.2 Index with single page size: worst case

Let us now consider the other extreme, when the tree contains as many promoted subtrees as possible at all levels. In this case, every node at index level x contains entries pointing to F sons (i.e. immediate descendant subtrees): a proportion F/x of these sons are unpromoted subtrees of height $(x-1)$; and for each of these there are $(x-2)$ entries pointing to promoted sub-trees of heights 1 to $(x-2)$ .The remaining fraction F/x of the entries point directly to data nodes. (i.e. There is a full sequence of guards for each unpromoted entry). The total number of data nodes can then be expressed recursively in terms of the number of data nodes which are sons of the root i.e. :

the number $td_{(h)}$ of data nodes reachable from the root of a tree $T_{(h)}$ of height h is:

the number $td_{(1)}$ of data nodes which are sons of the root of $T_{(h)}$

+ the sum of the number $td_{(1)}$ of data nodes reachable from the root of each son of height (1) of $T_{(h)}$

+ .....

+ the sum of the number $td_{(h-1)}$ of data nodes reachable from the root of each sub-tree $T_{(h-1)}$ of height $(h-1)$ of $T_{(h)}$.

Clearly, this expression will only be exact if F/x is integer at all index levels x , $1 \le x \le h$. For example, the smallest fan-out ratio which will yield a tree with the largest possible data capacity for a tree of height 5 in the worst case is 60.

Thus, for a tree with fan-out ratio F :

$$td_{(h)} = \frac{F}{h} \cdot \{ 1 + \sum_{k=1}^{h-1} td_{(k)} \} \qquad (4)$$

$$td_{(1)} = F; \qquad td_{(2)} = \frac{F}{2} \cdot \{F+1\}$$

In general:

$$td_{(h)} = \frac{(F+h-1)!}{(F-1)!\ h!} \approx \frac{F^{h}}{h!} \quad \ldots for\ F \gg h \qquad (5)$$

Thus, in the worst case, the number of data nodes in a BV-index of height h , and fan-out ratio F at all tree levels, is reduced by a factor of $h!$ compared to the best case.

The maximum number of index nodes in the worst case can be calculated in the same way i.e.

the number of index nodes $ti_{(h)}$ in a tree $T_{(h)}$ of height h is:

the root

+ the sum of the number of index nodes $ti_{(1)}$ in each son of height (1) of $T_{(h)}$

+ .....

+ the sum of the number of index nodes $ti_{(h-1)}$ in each son of height $(h-1)$ of $T_{(h)}$.

Thus, for a tree with fan-out ratio F :

$$ti_{(h)} = 1 + \frac{F}{h} \cdot \sum_{k=1}^{h-1} ti_{(k)} \qquad (6)$$

We can derive an approximate closed formula for $ti_{(h)}$ by neglecting the root index node in the initial recursive sum for $ti_{(h)}$ in equation (6) i.e.:

$$\mathrm{ti(h)} \approx \frac{\mathrm{F}}{\mathrm{h}} \cdot \sum_{\mathrm{k=1}}^{\mathrm{h-1}} \mathrm{ti(k)} \qquad (7)$$

Then: $ti_{(1)} = 1;\quad ti_{(2)} = \frac{F}{2};\quad ti_{(3)} = \frac{F}{3!}(F+2)$

... and in general:

$$ti_{(h)} \approx \frac{F}{h!} \cdot \frac{(F+h-1)!}{(F+1)!} \qquad (8)$$

Hence from (5) and (8) we have:

$$\frac{ti_{(h)}}{td_{(h)}} \approx \frac{F}{h!} \cdot \frac{(F+h-1)!}{(F+1)!} \cdot \frac{(F-1)!\ h!}{(F+h-1)!}$$

$$\approx \frac{1}{F} \quad \text{... for } F \gg 1 \qquad (9)$$

Equations (3) and (9) thus show that the ratio of index size to data size is effectively constant for all index configurations between the best and worst case.

It follows from equation (5) that, in the worst case, the number of index nodes is reduced by a factor of $h!$ (i.e. the same factor as the data nodes), compared to the best case. Figures 7-1 and 7-2 illustrate this reduction factor for data nodes. For each value of h, the height of the shaded area shows the difference between $log_F(td_{(h)})$ in the best and worst cases i.e. $log_F(h!)$

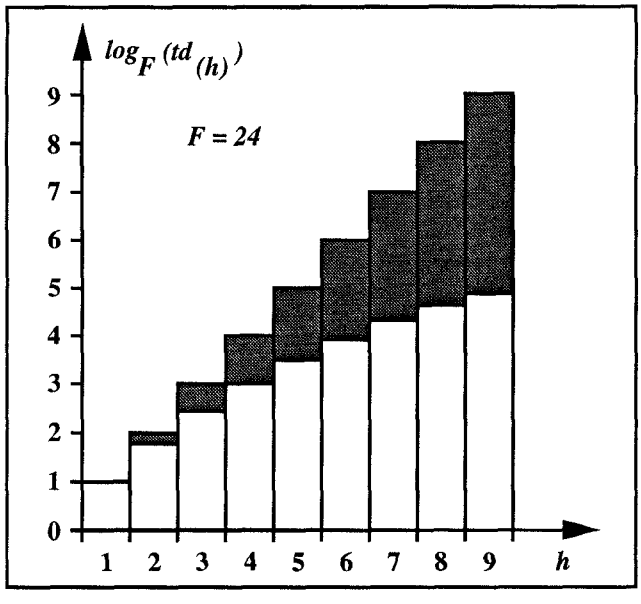

Figure 7-1: a comparison of best and worst-case performance of a BV-tree with uniform index page size; fan-out ratio F = 24

In figure 7-1, it will be seen that, with a fan-out ratio of 24, a best-case three-level index will have to grow to height 4 in order to maintain the same number of data nodes in the worst case. A best-case tree of height 4 will have to grow to height 6, and a best-case tree of height 5 will have to grow to height 10.

With a higher fan-out ratio (figure 7-2), this effect is less marked: a tree of height 4 need only grow to height 5, and a tree of height 6 need only grow to a height between 8 and 9. If the data pages are 1 Kbyte each, the latter corresponds to a 3 Petabyte file ($3x10^{15}$ bytes). For more modest-sized files - up to 200 Gigabytes - the index tree only has to grow by a maximum of 1 level to accommodate the worst case.

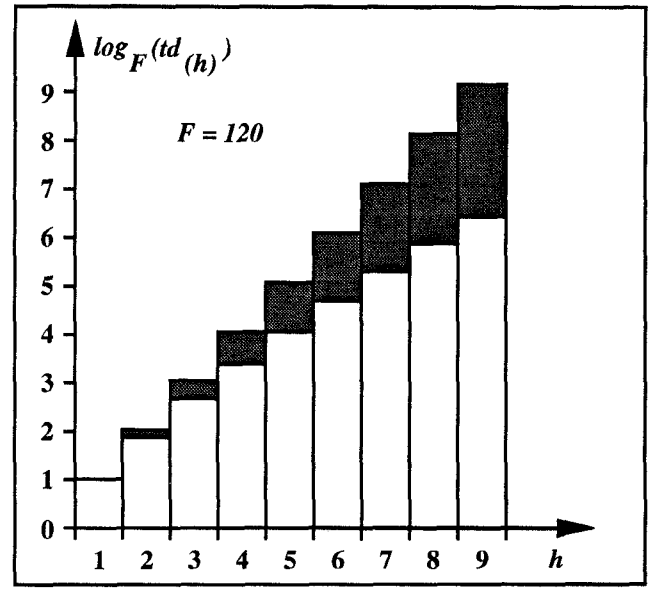

Figure 7-2: a comparison of best and worst-case performance of a BV-tree with uniform index page size; fan-out ratio F = 120

There remains some uncertainty however. In order to predict best and worst-case performance precisely - in terms of index tree height - the fan-out ratio must be known accurately. This is easy in range-based indexes of classical relations with fixed-sized attribute values as index keys. But if the attribute values are of variable length, or if the index is composed of variable-length computed keys, as in the BANG file, then the effective fan-out ratio will vary - if only slightly - according to the data and data distribution. In this case, the fan-out ratio can be monitored *dynamically*, by maintaining a record of the current total number of index and data nodes.

This will allow the worst-case performance to be precisely predicted at any time, but it will still not guarantee that the performance remains within a pre-determined, fixed limit. This can only be achieved if some upper limit can be set on the length of index keys, and if this upper limit allows a large enough fan-out ratio within available page-size limitations. If the gap between best and worst-case performance cannot be sufficiently narrowed within these parameters, then the problem can be eliminated entirely by introducing multiple-sized index pages.

7.3 Index with multiple page sizes

In the foregoing analysis, we assumed that all the pages representing the nodes of the index are of a fixed size B bytes, and that the individual keys within a page are of fixed

size b bytes. Let us now assume that every page at index level x is of size $B.x$ .This means that, at any level x, every node is just large enough to accommodate F unpromoted entries, and $F(x-1)$ promoted entries (guards) in the worst case. We can calculate the total number of data nodes in the worst case by modifying equation (4) to:

$$td_{(h)} = F[\,1 + \sum_{k=1}^{h-1} td_{(k)}\,] \qquad (10)$$

... from which we can derive the recursion formula:

$$td_{(h+1)} = td_{(h)}(F + 1) \qquad td_{(1)} = F \qquad (11)$$

... which leads to the closed formula for $td_{(h)}$:

$$td_{(h)} = F(F + 1)^{(h-1)} \approx F^{\,h} \;\ldots for\; F \qquad (12)$$

Comparing equation (12) with equation (1), we see that, for practical sizes of fan-out ratio, the maximum number of data nodes in the worst case of an index of height h with page size $B.x$ at index level x is the same as for the best case of an index of the same height with fixed page size B .

The corresponding maximum number of index nodes in the worst case is given by:

$$ti_{(h)} = 1 + F.\sum_{k=1}^{h-1} ti_{(k)} \qquad (13)$$

... which leads to the closed formula for $ti_{(h)}$:

$$ti_{(h)} = (F + 1)^{(h-1)} \approx F^{\,(h-1)} \;\ldots for\; F \gg 1 \qquad (14)$$

From equations (12) and (14) the ratio of index nodes to data nodes is given by:

$$\frac{ti_{(h)}}{td_{(h)}} = \frac{(F + 1)^{(h-1)}}{F(F + 1)^{(h-1)}} = \frac{1}{F} \qquad (15)$$

... which is the same as in the best case. Therefore the ratio is independent of the index configuration.

However, the index nodes are not all the same size. Given that the nodes at level x of the tree are of size $B.x$ bytes, the total size $si_{(h)}$ of an index with $ti_{(h)}$ nodes is given by:

$$si_{(h)} = B.h + F.\sum_{k=1}^{h-1} si_{(k)} \qquad (16)$$

... from which we can derive the recursion formula:

$$si_{(h+1)} = si_{(h)}(F + 1) + B \qquad si_{(1)} = B \qquad (17)$$

If $F \gg 1$ then $si_{(h)}(F + 1) \gg B$ so that:

$$si_{(h+1)} \approx si_{(h)}.F$$

... giving the approximate closed formula for $si_{(h)}$:

$$si_{(h)} = B.F^{(h-1)} \;\ldots for\; F \gg 1 \qquad (18)$$

Comparing equations (18) and (14), it will be seen that the increased size of the upper level nodes of the index has negligible effect on the overall index size. This is simply because, for $F \gg 1$, the number of nodes at level 1 of the index is at least an order of magnitude larger than at any higher level.

In summary: for a BV-tree with uniform index page size, a fan-out ratio of 24 and a data page size of 1Kbyte, the height of the index tree will increase by not more than two levels in the worst case index configuration compared to the best case, up to a data set size of order 100 Megabytes. For a fan-out ratio of 120, this size increases to order 25 Terabytes.

The degradation in performance due to promoted subtrees can however be completely avoided by introducing multiple page sizes. This has no significant effect on the overall index size.

8 Conclusion

We have shown how it is possible to recursively partition an n-dimensional dataspace so that the characteristics of the one-dimensional B-tree are preserved, so far as is topologically possible i.e. a minimum occupancy of 33% for both data and index nodes can be guaranteed,while also maintaining logarithmic access and update time. The only price which has to be paid for this performance is multiple-page sizes in the index. Even this is not likely to be necessary in practice except for very large files.

A preliminary modified version of the BANG file, supported by a BV-tree, confirms the anticipated performance characteristics. Future research will apply the principle of the BV-tree to other approaches to indexing. In particular, it was shown in [Fre89b] how to overcome the worst-case performance characteristics of the R-tree [Gut84] and the R+ tree [SRF87] for indexing extended spatial objects, by using a dual representation. The technique nevertheless depended on the BANG file, which itself remained vulnerable to pathological cases. However, by building the dual representation on the BV-tree, it should now be possible to create, for the first time, a direct method of indexing extended spatial objects, with the worst-case performance characteristics of the B-tree.

References

[Ben75] J.L.Bentley. *Multidimensional binary search trees used for associative searching.* Comm. ACM, Vol. 18, No. 9, 1975.

[Ben79] J.L.Bentley. *Multidimensional Binary Search Trees in Database Applications.* IEEE Trans. on Soft. Eng., Vol. SE-5, No. 4, July 1979.

[BM72] R.Bayer and E.McCreight. *Organisation and maintenance of large ordered indexes. Acta Informatica Vol. 1, No. 3, 1972.*

[BU77] R.Bayer and K.Unterauer. *Prefix B-trees.* ACM-TODS, Vol. 2, No. 1, March 1977.

[Com79] D.Comer. *The ubiquitous B-tree.* ACM Computing Surveys Vol. 11, No. 2, 1979.

[Fre87] M.Freeston. *The BANG File: a New Kind of Grid File.* Proc. ACM SIGMOD Conf., San Francisco, May 1987.

[Fre89a] M.Freeston. *Advances in the design of the BANG File.* 3rd Int. Conf. on Foundations of Data Organization and Algoritms (FODO), Paris, June 1989.

[Fre89b] M.Freeston. *A Well-Behaved File Structure for the Storage of Spatial Objects.* Symp. on the Design and Implementation of Large Spatial Databases, Santa Barbara, California, July 1989. Lecture Notes in Computer Science No. 409, Springer-Verlag, 1989.

[Fre94] M.Freeston. *A general solution of the n-dimensional B-tree problem.* ECRC Technical Report No. ECRC-94-40.

[Gut84] A.Guttman. *R-trees: a dynamic index structure for spatial searching.* Proceedings ACM SIGMOD Conf., Boston, 1984.

[HSW89] A.Henrich, H.-W.Six and P.Widmayer. *The LSD-tree: Spatial Access to Multidimensional Point and Non-point Objects.* 15th Int. Conf. on Very Large Data Bases (VLDB), 1989.

[KSS+90] H.P.Kriegel, M.Schiwietz, R.Schneider and B.Seeger. *A Performance Comparison of Multidimensional Point and Spatial Access Methods.* Proc. 1st Symp.on the Design of Large Spatial Databases, Santa Barbara CA, 1989. Lecture Notes in Computer Science No. 409, Pub. Springer-Verlag, 1990.

[LS89] D.B.Lomet and B.Salzberg. *The hB-tree: a Robust Multi-Attribute Indexing Method.* ACM Trans. on Database Systems, Vol. 15, No. 4, 1989.

[Ore86] J.A.Orenstein. *Spatial Query Processing in an Object-Oriented Database System.* Proc. ACM SIGMOD Conf., 1986.

[Rob81] J.T.Robinson.*The K-D-B-Tree: A Search Structure for Large Multi-dimensional Dynamic Indexes.* Proc. ACM SIGMOD Conf. 1981.

[Sam89] H.Samet. *The Design and Analysis of Spatial Data Structures*, Pub. Addison Wesley, 1989.

[SK90] B.Seeger and H.P.Kriegel. *The Buddy-tree: an Efficient and Robust Access Method for Spatial Data Base Systems.* Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, 1990.

[SRF87] T.Sellis, N.Roussopoulos and C.Faloutsos. *The R+ Tree: a Dynamic Index for Multi-dimensional Objects.* Proceeding 13th Int. Conf. on Very Large Data Bases, Brighton, 1987.