A New Generic Indexing Technology

Michael Freeston
CCSE/Alexandria, University of California
Santa Barbara CA93106
freeston@alexandria.sdc.ucsb.edu
tel: +1-805-893-8589
fax: +1-805-893-3045

Abstract

There has been no fundamental change in the dynamic indexing methods supporting database systems since the invention of the B-tree twenty-five years ago. And yet the whole classical approach to dynamic database indexing has long since become inappropriate and increasingly inadequate. We are moving rapidly from the conventional one-dimensional world of fixed-structure text and numbers to a multi-dimensional world of variable structures, objects and images, in space and time.

But, even before leaving the confines of conventional database indexing, the situation is highly unsatisfactory. In fact, our research has led us to question the basic assumptions of conventional database indexing. We have spent the past ten years studying the properties of multi-dimensional indexing methods, and in this paper we draw the strands of a number of developments together - some quite old, some very new, to show how we now have the basis for a new *generic* indexing technology for the next generation of database systems.

Multi-dimensional indexing

There has been no fundamental change in the dynamic indexing methods supporting database systems since the invention of the B-tree twenty-five years ago. And yet the whole classical approach to dynamic database indexing has long since become inappropriate and increasingly inadequate. We are moving rapidly from the conventional one-dimensional world of fixed-structure text and numbers to a multi-dimensional world of variable structures, objects and images, in space and time.

But, even before leaving the confines of conventional (i.e. relational) database indexing, the situation is highly unsatisfactory. In fact, our research has lead us to question the basic assumptions of conventional database indexing. The concept of a set of one-dimensional primary and secondary indexes, tuned to a particular query pattern, emerged in the days of menu-driven applications. This concept has not evolved in any way to match the flexibility of 4GL query languages. And today two new factors make such tuning increasingly impractical: internal system inferencing - where even the system itself cannot predict the progression of queries - and an increasing number of very large scale applications where the overhead of periodic re-indexing is unacceptable .

Of course, menu-driven applications remain, but there are many conventional database applications today where much more flexibility is needed. Users who browse through a database in an unpredictable way find system performance equally unpredictable. We believe that:

1. index design should be guided by a natural principle which is intuitively understood by users: the more information which is given to the system to guide a query, the faster should be the response.

2. the *average* response time to the set of all possible instantiations of a query is a much better guide to user acceptability than the *fastest* time obtainable with one particular instantiation pattern.

To achieve this kind of performance flexibility without incurring massive additional update overheads and increased software complexity requires some form of multi-dimensional indexing. Most recent research on multi-dimensional indexing has been motivated by the need for better spatial access methods. This has tended to obscure the much more general attraction of multi-dimensional indexing, which was clearly appreciated by earlier researchers in the field [1] [2].

The original inventors of the B-tree [3] already had the next step in mind: an index which would generalize 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. The attraction of such an index is that it narrows the search space evenly over all the indexed attributes, and so behaves as if there were an index on each attribute - while avoiding all the update complication of secondary indexes. It also greatly reduces the need to re-index - or multiply-index - large data sets on different attribute combinations.

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 generalization of the B-tree must recursively partition this data space into sub-spaces 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, should be logarithmic in the total data occupancy of the tree; and the occupancy of each data or index region should not fall below a fixed minimum.

The challenge has therefore been to generalize the B-tree to *n* dimensions *while preserving all of its worst-case characteristics*. But a solution has proved very elusive. Alternative

approaches of every kind have been proposed (see, for example, [4] [5]), but all have been vulnerable to pathological cases.

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 these same qualities before it has any hope of replacing the B-tree. For these reasons, and a number of others - even more compelling - which we give below, we believe that our recent discovery of a general solution of the *n*-dimensional B-tree problem [6] is a major breakthrough in dynamic indexing techniques.

Principles of a new generic indexing technology

As a result of ten years research in this area, we have come to appreciate the fundamental importance of a solution of the n-dimensional B-tree problem. This is essentially because *any* data structure which can be expressed as a tree can be represented as a point in an *n*-dimensional data space at the deepest level of a set of recursively nested data spaces i.e. data spaces containing points which represent data spaces at the next deeper level of recursion.

For example, conventional relational *n*-tuples can be represented as points in a single, *n*-dimensional data space. But if the relation contains a nested attribute, then the discriminator which distinguishes between instances of the nested attribute is the index key to the data space of nested attribute instances. This observation has been the basis of our long-standing hypothesis that an indexing system could be developed of sufficiently wide applicability to deserve being called *generic*. It has been the prime motivation for our attempts to develop a multi-dimensional index technique based on the following principles:

1. the index should be fully dynamic i.e. performance should not degrade with time;

2. performance should be fully independent of the data distribution;

3. the exact-match access path length, and times for single insertion, update and deletion should be at worst logarithmic in the size of the data set;

4. there should be a guaranteed fixed minimum in the ratio of the size of the data set to the size of the memory needed to store it;

5. there should be a fixed ratio between the maximum size of the index and the size of the data set (and this ratio should be << 1);

6. the index should represent a recursive partitioning of the data space (and *not* a recursive partitioning of the points in the space);

7. the partitions should be regular binary partitions of the *domains* of the data space;

8. the partitioning scheme should allow one partition region of the data space to *enclose* another;

9. the representation of the partitioning scheme should allow variable-length, binary keys;

10. the index keys should be generated dynamically.

The first five of these principles are satisfied by a B*-tree. Although it has never been proved that these principles cannot be satisfied by some data structure other than a tree, no such structure has yet been devised (with the possible exception of the hB-tree [7] which, strictly speaking, is not a tree, but a directed acyclic graph). So, in practice, we assume that these principles dictate that the basic indexing structure should be a 'grow and post tree' [8].

Principle six is not satisfied by a B-tree, which recursively partitions the *points* in the data space, not the data space itself. This choice may not be immediately obvious, since there are simple and effective techniques for mapping an n-dimensional point on to a linear order - using, for example, Z (Morton, or Peano) order [9] . Then an ordinary B-tree can be used, and the worst-case characteristics of the B-tree are automatically inherited. There is the additional practical advantage that it can be immediately applied in any system which supports B-trees.

But this approach is too restricted for a generic technology, which must be able to efficiently access extended spatial objects as well as points. The rectangular cover of an *n*-dimensional spatial object can always be mapped to a *2n*-dimensional point, but neighborhood relationships are lost in such a transformation. An alternative approach [9] resolves the object cover into a set of subspaces, each represented by a unique (e.g. Peano) code. This allows a recursive linear partitioning of these subspaces, in the same way as for points in a conventional B-tree. But the method violates principle three because, in general, the update of a single object cover requires the update of all its constituent parts. This introduces the uncontrollable access and update characteristics we want to avoid.

(Other well-known methods such as the R-tree and the R+-tree suffer from similar problems).

There are two other major reasons for the decision to partition the data space itself, rather than the objects within it. The first is that it is otherwise not possible to contract the representation of a sparsely occupied dataspace to a set of occupied subspaces. Comparative studies by [5] have clearly shown this to be a very significant factor in the efficiency of spatial range queries.

The second is that, in a data space representation, each of the subspaces in the recursive sequence enclosing a point or spatial object can be defined by a key which is a prefix of that of the next subspace in the sequence. This is the motivation for principle seven: by adopting regular (or *strict*) binary partitioning of the data space, each subspace can be assigned an extremely compact binary key with this prefix property. In fact, a much more compact representation than this is possible, since each subspace can be uniquely represented by a key which defines it *relative* to the subspace which encloses it at the next higher recursion level.

It is therefore possible to map the recursive partitioning of the data space to a tree-structured index in which:

(a) index tree level *l* corresponds to recursion level *l*.

(b) each branch node represents a subspace *s*, and the set of keys in the node represents the set of subspaces into which it is partitioned, *relative* to *s*.

(c) the leaf nodes of the tree represent the subspaces into which the data space is directly partitioned. The full key corresponding to any one of these subspaces is formed by concatenating the keys of all the subspaces which enclose it along the path from root to leaf of the tree.

If this tree can be made to have the properties of a ‘grow and post’ tree, then it will have further valuable properties:

(d) the full key defining a leaf node (subspace) does not need to be any longer than is necessary to differentiate it from the keys of all other leaf nodes.

(e) when a leaf node overflows and splits, as the tree grows, then the full key of at least one of the resulting nodes will grow in length.

(f) as the tree grows, common key prefixes will be promoted upwards through all levels of the tree.

The choice of regular binary partitions of the domains of the data space results in a partitioning scheme very closely related to that of quad-trees [4] . A criticism which has often been leveled at quad-trees is that they are not 'grow and post' trees, and therefore do not have guaranteed logarithmic worst-case access characteristics when mapped to secondary storage. But a quad-tree can always be mapped to a binary tree, and a 'grow and post' tree is a binary tree mapped to an *n*-ary tree. So it is natural to ask why it is not possible to map a quad-tree to a 'grow and post' tree?

In fact, a mapping between a quad-tree and a B-tree has been proposed [10] . But the spatial proximity relationships of the components of the quad-tree are lost in the mapping from two dimensions to one. To preserve these relationships, a mapping is needed from a quad-tree to a two-dimensional structure with the properties of a 'grow and post' tree. The problem lies not with the mapping, but in devising the two-dimensional 'grow and post' tree. Once this problem is solved, it becomes possible to support large-scale, persistent quad-trees, and the very large body of computational techniques now associated with them, within a generic, multi-dimensional indexing framework.

The adoption of regular binary partitioning of the data space also brings a particularly attractive feature of the quad-tree to the whole supporting framework: the precise registration (coincidence) of data space partition boundaries in overlay and spatial join operations - notably in GIS (Geographic Information System) applications. This is extremely important for the efficient and accurate execution of such operations. And since this property is a feature of the whole framework, it applies equally to raster and vector representations of spatial data, and conversions between the two.

It has sometimes been claimed that regular binary partitioning leads to the inclusion of too much empty space in the representation of skewed data distributions, particularly in the direct representation of extended spatial objects. We have found this not to be the case [11] . The basic reason for this is that, using a recursive partitioning scheme in a dynamic environment, the positions of the partitions at any level are moved very rarely compared to those at the level below. They are therefore effectively fixed, and in a dynamic environment their position becomes almost arbitrary. What is much more important is that, as already emphasized, it is possible to contract the representation to a set of occupied subspaces. It is also important that regular binary partitioning does not necessarily imply a strict cyclic order in the choice of dimension/domain for the partitions.

A consequence of properties (d) and (e) is that, in contrast to conventional indexing methods, the binary key for an indexed object can be generated *dynamically* during the descent of the tree. Its value and length will no longer depend solely on the indexed value of the object - it will also depends on its *degree of similarity* to other objects. When a leaf

node overflows and splits, the keys of the two resulting nodes will be extended just as far as is necessary to differentiate them.

This incremental approach to index key generation opens the way to the indexing of data entities with different structures in the same index. In conventional indexing, it is assumed that all the elements in an indexed data set have the same structure. In a relational database, this structure is stored in the schema, and is used to interpret the contents of the data elements in the corresponding relation.

New data models, however - specifically nested relational, object-oriented and logic - all allow structural variation within the elements of a relation, class or predicate respectively. In order to apply conventional indexing techniques to nested relations or objects, their structures must be resolved into simpler, fixed structures connected by physical or logical links. There is as yet no mechanism for a *single* index on the whole nested tuple or complex object instance.

However, it is always possible to generate a unique key from a (unique) instance of any complex structure, by including the structural information as well as the value information in the key. Then property (d) makes it possible to index very complex structures efficiently without in general using correspondingly long keys.

Further, a consequence of property (f) is that, if a set of points or objects in the data space generates a set of keys which are all different within the first few bits, then only these very short keys need be stored in the index. If, on the other hand, all the keys share a long, identical prefix, then this prefix will migrate to a single instance in the root of the index tree. Either way, the index representation becomes extremely compact.

All these properties, however, rest on the assumption that a data structure can be devised which complies with the ten principles listed above - which themselves require a solution of the *n*-dimensional B-tree problem. This is why a solution of this problem is such a key result.

BANG indexing

The original motivation for this research was to provide symmetric indexing support for rules and facts in a deductive database system based on logic [12] [13] [14] }. This proved to be a very difficult problem so, following the classical precept, we began by trying to solve a simpler problem: the multi-dimensional indexing of fixed-structure tuples. The result was the BANG file [15] [16] - a **B**alanced **A**nd **N**ested **G**rid file. With hindsight, this was a misnomer: a BANG index is in fact a balanced tree structure like a B-tree. What makes it different from a B-tree, and what it shares with the Grid file [17] [18] , is that it

partitions the data space itself, rather than the values within the space i.e. BANG index entries are not data values, but encoded representations of subspaces of the data space.

Apart from the B-tree, the other main source of inspiration for the BANG index design was linear hashing [19] . This led to the incorporation of two essential elements: variable length and bit-wise extensible index keys, and the dynamic generation of search keys.

The BANG file represents a set of *n*-tuples as points in an *n*-dimensional space. The index key associated with each point is a Peano code, generated by interleaving the bits of its *n* attribute values, from most to least significant, cycling through the *n* dimensions. However, the full key is not stored with the tuple. In fact it is not stored at all: it is generated dynamically during a tuple search, and during data space partitioning. Further, it is rare that the full key is generated: usually only a short prefix is necessary.

The data pages containing the *n*-tuples correspond to subspaces of the data space. These subspaces are also uniquely identified by Peano codes, generated by a sequence of regular binary partitions of the domains of the data space, cycling through the dimensions in the same sequence as for the index keys. The Peano code of any subspace is a prefix of the index key of every data point which it encloses.

The index itself is a balanced, multiply-branched tree structure in which every level corresponds to a recursive partitioning of the space represented at the level below. Each branch node represents a subspace which is defined by a Peano code in the index entry which points to it from the index level above (with the exception of the root node, which by definition represents the whole data space). Conversely, each index entry within a branch node represents a subspace which is enclosed by the subspace represented by the branch node itself.

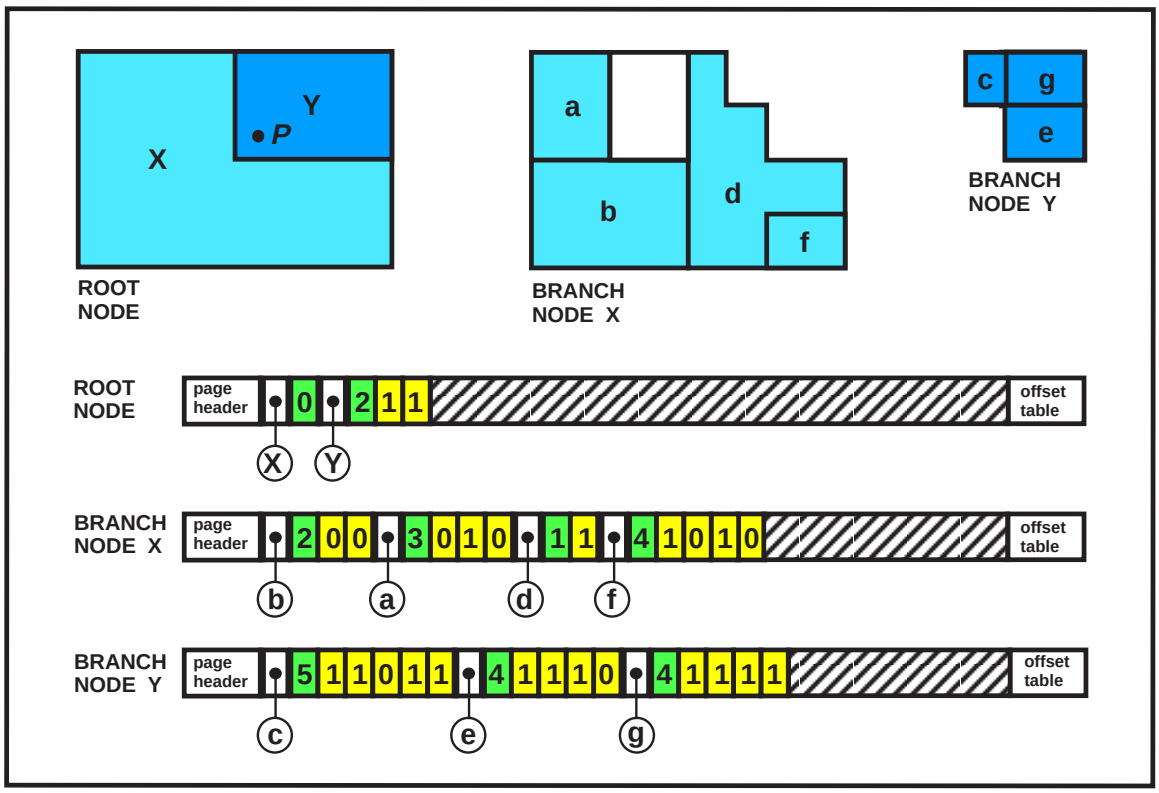

Figure 1: BANG indexing

Because of the regular binary partitioning sequence, the boundaries of partitions never intersect—either within the same index level, or between levels (although they may partially or fully coincide). However, there is nothing to prevent the creation of a subspace which fully encloses another at the same index level.

An example of a simple two-level partitioning of a data space, and the BANG index structure corresponding to it, is given in figure 1. Note that each index entry consists of three components: a pointer to the level below, an integer giving the length of the Peano code in bits, and the Peano code itself. Note also that the Peano codes defining subspaces at the lower index level are *relative* to the codes at the level above. In general, the full Peano code defining a subspace at index level l is the concatenation of all the Peano codes encountered in the direct traversal from the root to level l. This leads to an extremely compact index. (In practice, trailing code bits beyond the penultimate byte boundary are repeated at the next lower level, since it is much more efficient to perform code comparison and update operations if the alignment of bits in the byte is maintained).

Exact-match tuple search proceeds downwards from the root in the familiar B-tree manner. But within an index node, a bit-wise binary search is made for a Peano code which matches the corresponding bits of the search key. The key is dynamically extended during this search: only as many key bits are generated as the length of the longest Peano code

encountered. Note in the example in figure 1 how only very short keys need to be generated to locate the target data node, however long the actual tuple attributes may be.

Within a data node, tuples are stored in Peano code order. The target tuple can then be located by bit-wise binary search, dynamically generating both search and target keys. (Alternatively, each tuple can be stored with its full Peano code as an additional attribute).

It will be seen from figure 1 that there is a potential location ambiguity which arises when one subspace encloses another. This ambiguity is, however, simply resolved by always choosing the index entry with the longest Peano code which matches the search key: this must represent the innermost of any nested subspaces. Figure 1 also demonstrates a critically important feature of domain indexing: it is not in general necessary to represent the whole data space in the index, if large tracts of the space are empty. In the example of figure 1, some exact-match queries will fail at the root index level. In general, therefore, the *average path length* traversed to answer an exact-match query *will be less than the height of the index tree*. The more highly correlated the data, the greater the advantage of this feature compared to the conventional range-based indexing of the B-tree.

This advantage is reinforced by the extremely compact index - typically 1%-2% of the size of the data set for 1K index and data pages - due to the (on average) very short index entries. This leads to high fan-out ratios and hence a shorter index tree height and consequent faster access for a given data set.

But the greatest advantage of all is the multi-dimensional flexibility of this design. It can be used in conventional ways as a one-dimensional primary or secondary index, providing a smooth upgrade path from B-tree indexing, or it can be used multi-dimensionally when the query pattern is unpredictable. Figure 2 shows the results of two sets of two small experiments with A BANG index. The middle table shows the number of data pages accessed in response to all possible combinations of an exact-match/partial-match query on an employee relation of three attributes. Identity no distinguishes a tuple uniquely, the other two attributes do not.

Employee relation:

1-d: identity no, surname, forename
3-d: identity no, surname, forename

10,000 tuples
615 data pages

20,000 tuples
1246 data pages

Form of query	1-d	3-d	1-d	3-d
(100, smith, john)	1	1	1	1
(100, ? , ?)	1	42	1	60
(? , smith, ?)	615	84	1246	164
(? , ? , john)	615	145	1246	230
(100, smith, ?)	1	7	1	9
(? , smith, john)	615	21	1246	28
(100 , ? , john)	1	15	1	22
Total:	**1849**	**315**	**3742**	**514**
Av. per query:	**264**	**45**	**535**	**73**
	Ratio	**5.9 : 1**	**Ratio**	**7.3 : 1**

Figure 2: 1-d versus 3-d indexing

The 1-d column shows the number of accesses when the relation is indexed one-dimensionally on the *identity no*. attribute (which is a unique key). The 3-d column show the number of accesses when the relation is indexed multi-dimensionally on all three attributes. (the *surname* and *forename* attribute values are not unique). Assuming that each query is equally probable, the average number of data page accesses per query is shown at the bottom of the table. It will be seen that, *on average*, the single 3-d index is almost six times faster than a single 1-d index. The right-hand table shows the experiment repeated with a relation of twice the size. The 3-d index is now over seven times faster than the 1-d index. This is very good news: the larger the relation, the greater the relative performance improvement. In this example, the improvement is approaching an order of magnitude.

It must be said at once that this is a very crude test, and a number of factors - notably in the data distribution - could substantially change the outcome. Nevertheless, the performance advantage of the multi-dimensional approach in this example is so great that it is surely hard to dismiss. A better testimony is that BANG indexing has been used for several years now as the only index method supporting persistent data in the ECLiPSe Common Logic Programming System [20], which is now in use by over 300 sites around the world. So far, no request has ever been received for an alternative indexing method.

At this point the reader may justifiably ask: if multi-dimensional indexing is really so much better, why does the B-tree still exist? One answer is that, in order to take full advantage of multi-dimensional indexing, practically everything else in the database system must also be changed, from the setting up of indexes in the schema definition to the query optimizer and the fundamental relational operators - notably joins. There is therefore an enormous legacy problem. But the failure of all multi-dimensional indexing methods to date (including BANG indexing) to guarantee acceptable worst-case performance characteristics must also have been a major contributing factor. This weakness manifests itself in every multi-dimensional design in one of two ways: either there is no guaranteed minimum occupancy of data and index pages, or there is no guaranteed maximum path length for (single) search and update operations. Although this risk may be acceptable in many conventional business applications, it is certainly not so in the increasing number of real-time, mission-critical applications.

The BV-tree

In [6] we showed that this universal problem is a consequence of the basic topological properties of a data space, or rather of previous erroneous assumptions about the mapping of a recursively partitioned data space to a tree structure. The solution to the problem, surprisingly, is an *unbalanced* rather than a balanced tree.

Briefly, the problem arises when an index node overflows and splits along a boundary which does not coincide with any node boundary at the next lower index (or data) level. The choice is then either to choose a different splitting boundary - thereby losing any minimum occupancy guarantee for the two resulting nodes; or to force a split along the upper level boundary in one or more lower level nodes - which may propagate recursively to the bottom of the tree, so that no certain upper limit can be placed on the number of node splits resulting from a single update.

The nature of the problem is illustrated in figure 1. The root node of the two level index in figure 1 is created when the original single index node overflows into two nodes *X* and *Y*. The boundary between the subspaces represented by *X* and *Y* is chosen so that the occupancy of both nodes is as equal as possible. But it will be seen that this boundary cuts through region *d* at the index level below. In the figure, region *d* has been assigned to node *X*, with the result that a key search for a point *P* will fail, since there is no entry corresponding to this point in node *Y*.

In the original BANG index design [15], the problem was side-stepped by backtracking up the tree to the next longest entry key (i.e. the next smallest subspace - if such exists) matching the search key. However, in the worst case this could involve searching the entire index tree. In [16] forced recursive splitting was suggested as a better performance compromise, together with algorithms designed to minimize its worst effects. The maximum number of resulting split nodes in the worst case could then be exactly predicted, but the price for this was the loss of the minimum node occupancy guarantee. A careful examination will reveal the same trade-off in every other multi-dimensional index design.

The solution offered in [6] to this impasse is not in itself an index method, but a general result concerning the representation of a recursively partitioned data space by a tree structure. The basic idea is very simple: instead of forcing a node to split, promote it to the index level above. At first sight this may seem illogical, since it destroys the one-to-one correspondence between the levels of recursive data space partitioning and the levels of the index tree which represents this partitioning. However, all that it really destroys is the prejudice that such a static correspondence is necessary.

Instead, the correspondence is reconstructed *dynamically*, while traversing the tree structure. By promoting a node instead of splitting it, its index entry is moved to the root of the subtree which contains the pointers to the two nodes which would otherwise contain the split node entries. Thus any index tree search which visits this subtree root can then pick up the promoted entry - if the search key matches - and follow one of these pointers back down to the next lower index level. The entry is then included in the set of index entries to be searched at this level, exactly as if it had been split rather than promoted.

The result, as illustrated in figure 3, may seem paradoxical: an unbalanced tree (a BV-tree) with the operational properties of a balanced tree. But this is simply because all search and update operations are effectively carried out on a balanced tree reconstituted from the unbalanced tree. In figure 3 the subspace boundaries are deliberately chosen to be randomly shaped, to emphasize that this is a general technique, not a specific index method. A more detailed account is given in [6] , but the main point is that the structure makes it possible to combine a minimum node occupancy guarantee with guaranteed maximum search and update path lengths.

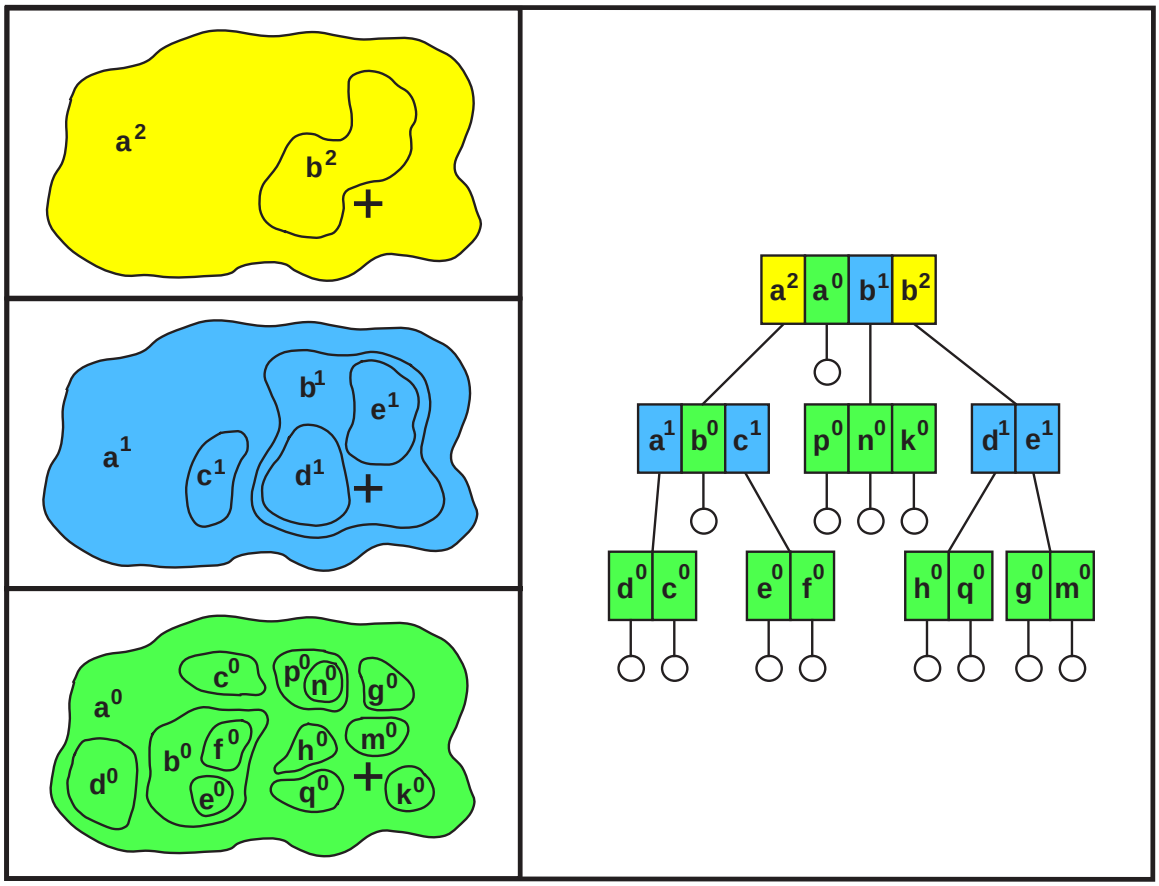

Figure 3: The BV-tree

The worst-case node occupancy is 33% for the BV-tree, compared to 50% for the B-tree, but this is the maximum which is topologically possible in more than one dimension. In practice, as with the B-tree, the worst case can always be improved upon by redistributing the contents of nodes whose occupancy levels fall below some arbitrary value. The average occupancy is the same as that of the B-tree i.e. around 69%.

The number of nodes visited in an exact-match search, and the maximum number of nodes *split* as the result of a single update, is logarithmic in the size of the data set, as in a B-tree. The number of nodes *visited* during a single update can however be greater than logarithmic: in the worst case it is $h(h+1)/2$, where h is the maximum height of the tree.

[N.B. This case was overlooked in [6], where it was claimed that updates can always be performed in logarithmic time. This is indeed true except in one case, which arises when a promoted node overflows, and when one of the resulting split nodes can be demoted. The demotion must be carried out immediately, in order to maintain the dynamic behavior of the tree. We believe - although have not proved - that this, like the 33% minimum page occupancy, is a topological inevitability. In practice, this will be such a rare occurrence that

the average update time will hardly deviate from logarithmic. More important, the worst-case update time remains entirely predictable].

It is also possible that, if a fixed page size is used, the height of the index tree may be greater than that of the equivalent balanced tree (see [6] for an analysis). This tendency may however be more than compensated by the fact that the average path length for an exact-match search in domain-based indexing is less than the height of the tree.

So if we are to judge the acceptability of multi-dimensional indexing by comparison with the worst-case characteristics of one-dimensional B-tree indexing, there is clearly now very little difference between them. The important point is that multi-dimensional indexing can now safely support mission-critical applications with absolutely predictable worst-case performance. Best of all, there is *no* price to pay for the extra flexibility and average performance which multi-dimensional indexing gives.

A BANG representation of a BV-tree

We have so far made the implicit assumption that the general properties of the BV-tree can be implemented in a specific index representation. And if we are to achieve our ultimate goal of an indexing technology which satisfies all the principles and properties listed above, we also have to combine this implementation with all the principles applied in BANG indexing. So the question remains: can such a design be implemented efficiently?

There is not space here to set out the full details, but we hope to show enough to demonstrate that it is possible to build an implementation which is both efficient and elegant, with relatively low software complexity. The appendix contains the pseudo-pascal code for a procedure to perform a key-search within an index node of a BANG index representation of a BV-tree. Figure 4 illustrates both the modified form of a BANG index node, and the key-search operation on it. The contents of the ten index entries are set out vertically rather than horizontally as in figure 1, but otherwise the only difference in their structure is the addition of the *partition level* of the entry. This is the index level at which the entry was originally created, counting the index leaf node level as level 0. The index node shown is located at level 2, since the highest partition level shown is 2, but the node also contains two entries promoted from level 1, and one entry promoted from level 0. (In figure 2, the equivalent would be the promotion of the entry for subspace *d* from node *X* at level 0 to the root node at level 1). We frequently refer to promoted entries as guards.

Before the tree search begins, an array is initialized, which we call the *guard set*, having one element for each partition level of the tree, such that array element *l* corresponds to partition level *l*. Each element consists of a record of two fields: the relative length of an index entry key (i.e. the length relative to the page in which it lies); and the pointer associated with that index entry. The array is used to hold this information on the longest entry key of each partition level so far encountered in the traversal of the tree.

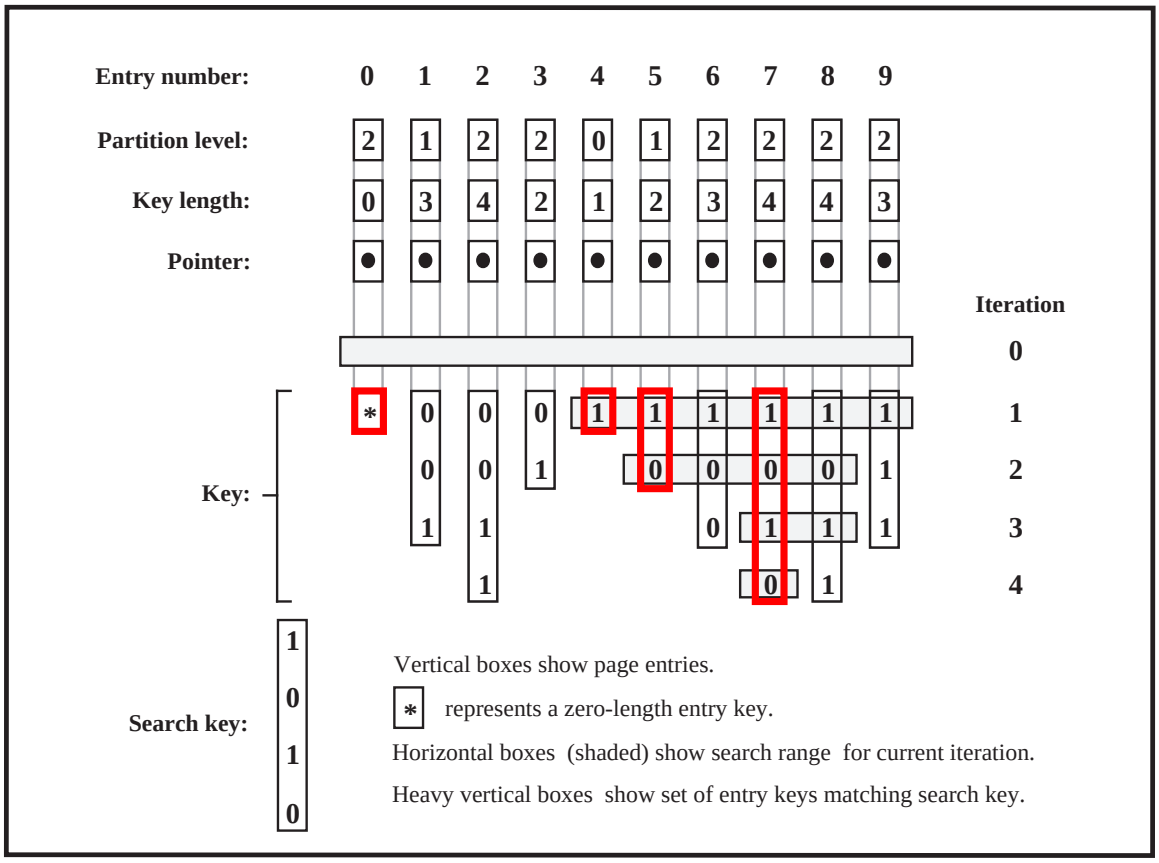

Figure 4: Key search in a BANG implementation of a BV-tree

Regardless of their promotion level, index entries are stored in lexical order of their entry keys, so that the longest key matching the search key can be found by bit-wise binary search of the index node (entry 7 in figure 4). The range of each iteration of this binary search is shown by the shaded horizontal bars in figure 4, and it will be seen that the next matching prefix of the search key is always the first entry in the range. At each iteration, the relative key length of this entry is compared with the length stored in the array element corresponding to the same partition level. If the new entry is the longer, its length and associated pointer are stored in this array element, overwriting its previous contents.

When the binary search within a node at index level *l* is complete, the next node to be searched is that pointed to by array element *l*. Note that array element *l* may at this stage simply hold the key length and pointer of the longest matching entry in the node just searched at level *l*, or it may hold values carried down from a promoted entry at level *(l+1)* - if this entry has a longer key. And such an entry may also have been carried down from higher levels in the course of the tree search.

An example

The effect of this simple algorithm is to reconstitute a balanced tree by moving the relevant promoted entries back to the index level at which they were created. As an example, consider a search for the point marked + in figure 3. At the root index level (level 2) entries $\mathbf{a}^2$, $\mathbf{b}^1$ and $\mathbf{a}^0$ represent the smallest subspaces enclosing the target point at each partition level, so the lengths of these entries and their associated pointers are stored in the guard set. We then follow the pointer in element 2 of the guard set down to the next index level. Here the only matching entry is $\mathbf{a}^1$. But there already exists in array element 1 of the guard set another entry - $\mathbf{b}^1$ - carried down from the level above, and which plainly encloses the target point more closely than $\mathbf{a}^1$ i.e. in a BANG representation, it has a longer key.

So we now follow the pointer associated with $\mathbf{b}^1$. Note that this causes the tree traversal to effectively backtrack up the tree and down another branch, although no previously visited nodes are revisited. Finally, we see that there are no matching entries at level 0 in the node pointed to by $\mathbf{b}^1$, so we pick up the existing pointer in array element 0, which is the pointer from entry $\mathbf{a}^0$ carried down from the root. Thus a second 'virtual backtrack' occurs. Note however that the total number of nodes actually visited is equal to the maximum direct path length from root to leaf of the tree. This is true for all exact-match search paths in a BV-tree, thus guaranteeing its logarithmic search behavior.

Of course, the complete search and update code is considerably longer and more complex than that shown in the appendix, but the tree traversal and dynamic tree reconstruction techniques shown recur throughout.

Conclusion

We can now reasonably claim to have fulfilled all the principles and properties laid down in our initial list of design requirements. Multi-dimensional indexing really is now a more flexible alternative to B-tree indexing. But this is not the end of the matter - it is only the beginning. As we began by pointing out, our true objective has not been to find a replacement for the B-tree: that is just a spin-off. The main aim has been to develop a basis for a new *generic* indexing technology which is capable of supporting much more generalized data types than those of conventional business applications, and in a scalable way for large applications. We are confident that, with the technology which we have assembled above, we can now develop this generalized support. But it is clear that there is a lot of research still to be done, and whole new areas to be investigated.

References

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

2. Robinson, J.T. *The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes*. ACM SIGMOD Conf. 1981.

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

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

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

6. Freeston, M. *A General Solution of the n-dimensional B-tree Problem*. ACM SIGMOD Conf., San Jose, CA, May 1995.

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

8. Lomet, D. Grow and Post trees.

9. Orenstein, J. *Spatial Query Processing in an Object-Oriented Database System*. ACM SIGMOD Conf. 1986.

10. de Jonge, W., Scheuermann, P. and Schijf, A. *Encoding and Manipulating Pictorial Data with S^+-trees.* 2nd Symposium on Large Spatial Databases, Zurich, August 1991.

11. Freeston, M. *The Comparative Performance of BANG Indexing for Spatial Objects*. 5th International Symposium on Spatial Data Handling, Charleston SC, August 1992.

12. Bocca, J., Dahmen, M. and Freeston, M. *MegaLog—A Platform for Developing KnowledgeBase Management Systems.* 3rd Russian Conference on Logic Programming and Automated Reasoning, St. Petersburg, Russia, July 1992.

13. Bocca. J. *MegaLog—A Platform for Developing Knowledge Base Management Systems*. Int. Symposium on Database Systems for Advanced Applications, Tokyo, Japan, April 1991.

14. Vieille, L., Bayer, P., Kuechenhoff, V. and Lefebvre, A. *EKS-V1, A Short Overview*. Proc. AAAI-90 Workshop on Knowledge Base Management Systems, Boston, July 1991.

15. Freeston, M. *The BANG file: a new kind of grid file*. ACM SIGMOD Conf., San Francisco, May 1987.

16. Freeston, M. *Advances in the Design of the BANG File*. 3rd International Conference on Foundations of Data Organisation and Algorithms (FODO), Paris, June 1989.

17. Nievergelt, J., Hintenberger, H. and Sevcik, K. *The Grid File: An Adaptable, Symmetric Multikey File Structure*. ACM Trans. on Database Systems, Vol.9, No.1, 1984.

18. Hinrichs, K. *The Grid File System: Implementation and Case Studies of Applications*. Doctoral Thesis Nr. 7734, ETH Zurich, 1985.

19. Litwin, W. *Linear Hashing: a New Tool for File and Table Addressing*. Proc. 6th Int. Conf. on Very Large Databases, Montreal, Canada, 1980.

20. Eclipse 3.5 *User Manual*. European Computer-Industry Research Centre (ECRC), 1996. URL http://www.ecrc.de/eclipse/html/umsroot/umsroot.html.

Appendix

procedure search_index_page(page, bottom, top, search_key, guard_set, insertion_point)

{ *Initialize the byte offset from the start of the (absolute) search key to the start byte of the relative search key i.e. to the byte corresponding to the first key byte of the p age entries* }

relative_search_key := search_key + page_level div BYTESIZE;

{ *Initialize the bit offset of the start of each entry key in the page, relative to the start of the first key byte*}

start_bit_offset := page_level mod BYTESIZE;

{ *Initialize comparison bit number relative to the start of the first byte of each entry key. The most significant bit in a byte is numbered as bit 0.*

Initially, the comparison_bit_number *is set to one less than that of the actual start position, so that any initial entries of zero length will be correctly treated* }

```
comparison_bit_number := start_bit_offset - 1;

while bottom <= top do begin

 if length(key(page_entry(page, bottom)))=comparison_bit_number-start_bit_offset +
 1
 then begin
   { This is a matching entry i.e. the complete key of the entry matches a prefix of the
     insertion key. Note that there may be several matching entries with identical keys,
     provided that they differ in their index node level.

    Note also that the presence of a guard of node level l-x in a page of node level l
    does not imply that the page must also contain guards of all higher levels l-x+n,
    where 0 < n < x }

   { Record the matching entry in the guard set }
   level := index_node_level(page_entry(page, bottom));
   guard_set.entry[level] := bottom;
   bottom := bottom + 1
 end
 else
   if length(relative_search_key) = comparison_bit_number - start_bit_offset + 1
   then { No further matching entries possible: the search key (region) encloses all
   remaining entries in the current search set.
   Note that there must be at least one entry in this set, since this point in the code is
   only reached if there exists at least one entry in the current search set whose key is at
   least one bit longer than the current position of the comparison bit }
   begin
       insertion_point := bottom;
     { The insertion point lies immediately before the entry numbered bottom in the page
       entry table }
       return
   end
   else begin
     { Establish range of entries matching search key as far as the next comparison bit}


     { Advance comparison bit }
       comparison_bit_number := comparison_bit_number + 1;

       if value(comparison_bit(key(page_entry(page, top)))) = 0
       then begin
         if value(comparison_bit(relative_search_key)) = 1
         then bottom := top + 1 { No further matching entries possible }
       { else all entries in the current search range must match as far as the current
         comparison bit: the range thus remains unchanged }
       end
       else if value(comparison_bit(key(page_entry(page, bottom)))) = 1
```

```
        then  begin
          if  value(comparison_bit(relative_search_key)) = 0
          then top := bottom - 1 { No further matching entries possible }
        { else all entries in the current search range must match as far as the current
          comparison bit: the range thus remains unchanged }
        end
        else  begin

        { Binary search for first entry with current bit set to 1. If this section of code is
          entered, there must be at least two entries in the current search range. This
          follows from the fact that the program execution must drop through the
          preceding code to arrive at this point. This can only occur if the value of the
          comparison bit for the top entry is 1, and that for the bottom entry is 0. Hence
          these two entries cannot be the same. It also follows that there must be at least
          one entry in the new search range }

          top_limit := top;
          bottom_limit := bottom;
          repeat
            middle := (top_limit + bottom_limit) div 2;
            if comparison_bit(key(page_entry(page, middle))) = 1
            then top_limit := middle - 1
            else bottom_limit := middle + 1
          until top_limit < bottom_limit; { end of binary search }


        { Reset lower or upper limit of search range: bottom_limit is first entry with
          comparison bit value 1; top_limit is last entry with comparison bit value 0}

          if  value(comparison_bit(relative_search_key)) = 1
          then bottom := bottom_limit
          else top := top_limit
        end { Binary search ...}
      end { Establish range of entries matching search key... }
    end { while bottom <= top }

    insertion_point := bottom
  { The insertion point lies immediately before the entry numbered bottom in the page
    entry table }

end {procedure search_index_page}
```