PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 17/30 — A1

(11) International Publication Number: WO 95/02222

(43) International Publication Date: 19 January 1995 (19.01.95)

(21) International Application Number: PCT/EP94/02166

(22) International Filing Date: 1 July 1994 (01.07.94)

(30) Priority Data:
93110856.7 7 July 1993 (07.07.93) EP
(34) Countries for which the regional or international application was filed: DE et al.

(71) Applicant *(for all designated States except US)*: EUROPEAN COMPUTER-INDUSTRY RESEARCH CENTRE GMBH [DE/DE]; Arabellastrasse 17, D-81925 Munich (DE).

(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: FREESTON, Michael, William [GB/DE]; Elektrastrasse 30a, D-81925 Munich (DE).

(74) Agent: DUPUY, Susan, Mary; Intellectual Property Dept., International Computers Limited, Cavendish Road, Stevenage, Herts SG1 2DY (GB).

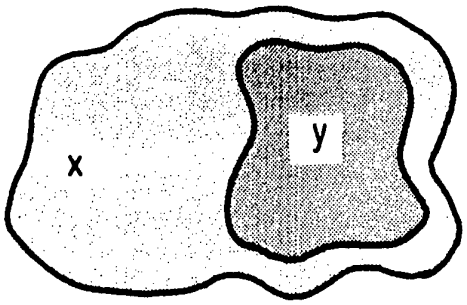
(81) Designated States: AU, JP, US, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

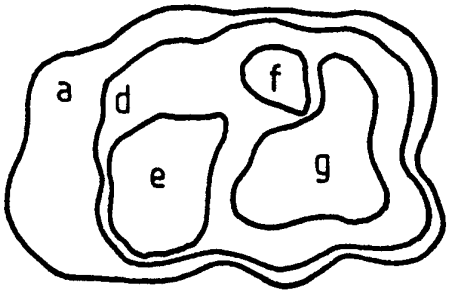
Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

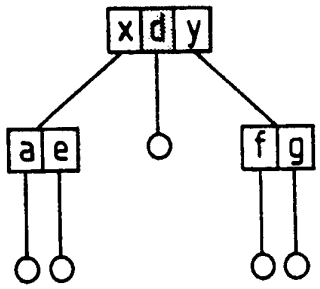
(54) Title: DATABASE STRUCTURES

(57) Abstract

A hierarchical index structure, for a multi-dimensional information database, is such that at least one index entry partition (d) may be disposed at a higher index tree level than the hierarchical level to which it corresponds. This change of level (promotion) occurs when a data distribution changes or the pattern of insertions/deletions is very skewed to particular areas of the dataspace. Whilst this makes the index tree unbalanced, it facilitates the addition of information to and the searching of such a database.







FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	**GB**	United Kingdom	**MR**	Mauritania
AU	Australia	**GE**	Georgia	**MW**	Malawi
BB	Barbados	**GN**	Guinea	**NE**	Niger
BE	Belgium	**GR**	Greece	**NL**	Netherlands
BF	Burkina Faso	**HU**	Hungary	**NO**	Norway
BG	Bulgaria	**IE**	Ireland	**NZ**	New Zealand
BJ	Benin	**IT**	Italy	**PL**	Poland
BR	Brazil	**JP**	Japan	**PT**	Portugal
BY	Belarus	**KE**	Kenya	**RO**	Romania
CA	Canada	**KG**	Kyrgystan	**RU**	Russian Federation
CF	Central African Republic	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CG	Congo			**SE**	Sweden
CH	Switzerland	**KR**	Republic of Korea	**SI**	Slovenia
CI	Côte d'Ivoire	**KZ**	Kazakhstan	**SK**	Slovakia
CM	Cameroon	**LI**	Liechtenstein	**SN**	Senegal
CN	China	**LK**	Sri Lanka	**TD**	Chad
CS	Czechoslovakia	**LU**	Luxembourg	**TG**	Togo
CZ	Czech Republic	**LV**	Latvia	**TJ**	Tajikistan
DE	Germany	**MC**	Monaco	**TT**	Trinidad and Tobago
DK	Denmark	**MD**	Republic of Moldova	**UA**	Ukraine
ES	Spain	**MG**	Madagascar	**US**	United States of America
FI	Finland	**ML**	Mali	**UZ**	Uzbekistan
FR	France	**MN**	Mongolia	**VN**	Viet Nam
GA	Gabon				

DATABASE STRUCTURES

Technical Field

This invention relates to database structures and in particular to hierarchical index structures for use therein, methods for use in indexing a dataspace and methods of searching database structures. The invention also relates to the electronic storage of an n-dimensional entity in the form of an n-dimensional data space in a one-dimensional memory of a computer, and/or the transfer and/retrieval of said n-dimensional entity to or from the memory of the computer.

Background Art

Spatial information can be stored in the memory of a computer. Spatial information consists essentially of points in an n-dimensional data space. For example, the points may be coordinates of the centres of objects on a map, in two dimensions, or the locations of aircraft in three-dimensional air-space. Associated with this positional information may be additional information describing the object at the location - such as the type of a map object, or the call-sign and velocity of an aircraft.

Classically, data structures in database systems have been restricted to fixed-structure records or tuples. The structure of a tuple is a set of fields, or attributes. The types of the attributes have been restricted to a few simple types, such as real, integer, and string.

In order to build an index to a set of records, it is assumed that all the members of the set have the same structure. A key value must be associated with each. This value may or may not be unique to the record. In principle, it may be the direct value of a single attribute, or of several attributes, or it may be generated by some conversion operation on one or more attributes. The unit of memory allocation for the data and the index, in main memory or secondary storage, is a page, and this page is almost invariably of fixed size.

The invention is concerned with a hierarchical index, which takes the form of a tree structure. In general, a tree structure is composed of a root node, branch nodes and leaf nodes. By convention, the tree is represented in inverted form ie with the root at the top. A traversal path through the tree is defined by the sequence of nodes encountered along the path. The height of the tree is the length of the longest direct path traversed from root to leaf. The fan-out ratio is the number of branches leading from a node in the direction of the leaves. This ratio usually has a range of allowed values, depending on the details of the design and implementation. The limits of this range are the same for all the index nodes.

The best known and most widely used hierarchical structure for dynamically indexing a set of records in a database is the B-tree. The B-tree takes the value of a single attribute in a record, or the lexical concatenation of several attributes, as the index key. Each index node corresponds to a page of memory, and contains an ordered set of index keys. The index is constructed as a hierarchy of index keys: at any particular level of the tree, each node contains an ordered set of key values and, associated with each key,

a pointer to a node at the index level below. Each key represents an upper (or lower) bound to the key values stored in the node to which it points. At the lowest index level, the keys point to data pages containing records within the ranges defined by the lowest level index keys.

When the insertion of an additional record causes a data page to overflow:

1. the page is split into two pages about the median value of the index key attribute(s).

2. a copy of the median key attribute value, together with an additional pointer for the new page, is inserted in the index leaf node which holds the pointer to the original page. An index node thus consists of [key, pointer] pairs, stored in key value order. [Plus one additional pointer, for the extreme upper or lower range partition].

If an insertion in an index node causes it to overflow, then the index node is similarly split about its median key value, a copy of which is posted upwards, together with a pointer to the newly created index page.

In the worst case, a single insertion of a data record can trigger a chain of overflows and insertions up to and including the root of the index tree. When the root splits, a new root is generated and the height of the index tree increases by one. In this way a tree-structured index grows upwards (ie an inverted tree) as the number of data pages increases.

If all index pages are the same size, and the maximum number of key entries in each index page is F (the

fan-out ratio), then each level of the index has maximally F times as many index pages as the level above, and there will be maximally F times as many data pages as pages in the lowest index level. Since both data and index pages are split at their median values, the worst-case occupancy is 50%, but the statistical average for the entire index is approximately 69%.

The B-tree structure has a number of attractive properties, which is why it has been almost universally adopted for dynamic indexing in database systems. In particular:

1. the tree is always perfectly balanced ie for a given size of tree, every direct path from root to leaf is the same length;

2. if the full key of any individual record is specified (an exact-match query), the record can always be accessed via a direct path from root to leaf. Thus the access time for a single record is constant and predictable, for a given tree size;

3. the length of the direct path from root to leaf increases only logarithmically in the total number of indexed records so that, typically, a five-level index is sufficient to index several Terabytes of data;

4. the maximum update time for the insertion or deletion of a single record is also logarithmic in the total number of indexed records;

5. there is a guaranteed minimum occupancy (50%) of the data and index pages;

6. the maximum size of the index is always directly proportional to the quantity of the data;

7. all the above properties are preserved under deletion as well as insertion, ie the performance of the structure does not deteriorate with usage - it is fully dynamic.

For many years researchers have tried to find a structure which generalises the properties of the B-tree to n dimensions ie 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 individuals 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 partition this data space into sub-spaces or regions in such a way that the properties of the B-tree as listed above are preserved. Specifically, each region corresponds to a data page, and the index represents a recursive partitioning of the data regions into higher level index regions.

In order to achieve symmetry in the n dimensions, in the sense defined above, the following additional property

is needed:

8. the number of recursive partitions of the data space required to define the boundaries of each data region should be (as nearly as possible) the same in each dimension.

It is further important to the efficiency of the index that the following properties are maintained:

9. the spatial distribution of the data regions should correspond as closely as possible to the spatial distribution of the data points;

10. the spatial distribution of the index regions at each level of the index tree should correspond as closely as possible to the spatial distribution at the next lower level, and to that of the data regions;

11. the representation of empty space should be minimised.

Unfortunately, it has proved extremely difficult to reconcile all these requirements. Considerable progress has been made, as apparent from the substantial number of proposals put forward over the last few years, but there remains an underlying inflexibility in current designs, and prior to this invention there was no solution which was provably resistant to pathological cases.

One of the previous proposals is a multi-dimensional file structure known as the BANG file (Balanced And Nested Grid file). See for example "The BANG file: a new kind of grid file" M W Freeston. Proc. ACM SIGMOD

Conf. San Francisco, June 1987, or "Advances in the design of the BANG file" M W Freeston, Third International Conference on Foundations of Data Organisation and Algorithms, Paris, June 1989. The BANG file has a tree-structured directory which has the self-balancing property of a B-tree and which allows data space and index regions to be nested, one inside the other. The BANG file adapts well to non-uniform data distributions but it cannot maintain guaranteed performance characteristics in all cases.

The problem is how to allocate the tuples represented by the points in an n-dimensional data space to a set of data blocks (or pages) in a file. In geometric terms the obvious thing to do is to divide up the data space into a set of hyperrectangles or block regions, each of which corresponds to a disk page. However, in order to maximise the storage efficiency of the file, the number of such regions and the positions of their boundaries, have to be arranged so that each corresponding disk page has a high data occupancy.

In the BANG file, data pages are accessed via a multi-level directory of block regions, stored in the same file as the data. When the insertion of a tuple into the file causes a disk block to overflow, then the disk page and the corresponding region are split into two and the directory is updated. The partitioning strategy is required to obtain a "best balance", ie nearest possible balance, between the occupancies of the two resulting regions. The BANG file partitions the data space by a sequence of binary divisions in which the dimension selected for the next division is chosen according to a pre-defined order of cycling through all the dimensions. Whilst the same partitioning sequence is applied throughout the data space, the partitioning

operation is applied selectively to individual block regions. The partitions can, in principle, be any shape within the restrictions of the following axioms:

Axiom 1: no partition boundary may intersect itself;

Axiom 2: no two partition boundaries may intersect.

In particular, one partition can enclose another ie they can be "nested".

It is an object of the present invention to provide a database structure, in particular a hierarchical index structure for use therein, which enables n-dimensional information to be mapped into the one-dimensional memory of a computer (ie a memory consisting of a linear sequence of pages) and conversely which enables efficient retrieval of information from the memory relating to a specified point in the n-dimensional data space.

Disclosure of Invention

According to one aspect of the present invention there is provided a hierarchical index structure characterised in that, in the event of data distribution changes or skewing of insertion/deletions to particular areas of a dataspace, at least one index entry partition (d) is disposed at a higher index tree level than the hierarchical level to which it corresponds.

According to another aspect of the present invention there is provided a database structure for multi-dimensional information including a hierarchical index structure and wherein a data space can be recursively partitioned into subspaces with

corresponding index entry nodes arranged in an index tree, characterised in that, in the event of data distribution changes or skewing of insertion/deletions to particular areas of a dataspace, at least one index entry partition is disposed at a higher index tree level than the hierarchical level to which it corresponds.

According to a further aspect of the present invention there is provided a method for use in indexing a dataspace, the dataspace comprising a database for multi-dimensional information, the indexing having a hierarchical structure and the dataspace being recursively partitioned into subspaces with corresponding index entry nodes arranged in an index tree, characterised in that upon the introduction of extra information causing an index entry node to overflow, the index entry node is split into two split partitions, which are disposed at a respective level of the index tree, and a respective index entry partition whose boundary directly encloses a said split partition is disposed at a higher level in the index tree than the hierarchical level to which it corresponds.

According to yet another aspect of the present invention there is provided a method of searching a database structure containing multi-dimensional information in a dataspace and including a hierarchical index structure, the dataspace being recursively partitioned into subspaces with corresponding index entry nodes arranged in an index tree, wherein at least one index entry partition may be disposed at a higher index tree level than the hierarchical level to which it corresponds as a result of data distribution changes or skewing of insertions/deletions to particular areas of the dataspace, characterised by searching iteratively the index tree hierarchy for the smallest partition in which

an item of information sought is located, the search beginning at a root entry node and examining each hierarchical level in turn, and the search involving backtracking between levels in the case of index entry partitions not disposed at the index tree level corresponding to their hierarchical level, the number of different nodes visited along the search path for the item of information sought being constant for a constant total number of items of information contained in the dataspace.

According to a still further aspect of the invention there is provided a computer including a memory employing a hierarchical index structure characterised in that the hierarchical index structure is such that, in the event of data distribution changes or skewing of insertion/deletions to particular areas of a dataspace, at least one index entry partition (d) is disposed at a higher index tree level then the hierarchical level to which it corresponds.

According to a further aspect of the invention there is provided a computer database system including a memory employing a hierarchical index structure, characterised in that the hierarchical index structure is such that, in the event of data distribution changes or skewing of insertion/deletions to particular areas of a dataspace, at least one index entry partition (d) is disposed at a higher index tree level than the hierarchical level to which it corresponds.

According to another aspect of the invention there is provided a method for electronic storage and/or transfer of an n-dimensional entity represented in an n-dimensional data space into one-dimensional memory of a computer comprising the steps of partitioning the data

space into sub-spaces or regions, splitting a first region into the first region and at least one further region such that the first region encloses the further region(s) if the number of points inserted into said region exceeds a predetermined number,
storing the regions represented by data notes into one or more pages in the memory of the computer,
attributing an index identifying a region to each region,
storing the indices into one or more index pages in the memory of the computer,
splitting an index page if the number of indices representing the regions exceeds a predetermined number,
connecting said split index page by pointers to the several index pages and
storing said several index pages in the memory of the computer.

Brief Description of the Drawings

Embodiments of the invention will now be described with reference to the accompanying drawings, in which:

Figures 1 to 3 illustrated a very simple 2-level index;

Figure 4 illustrates the corresponding BV-tree;

Figure 5 illustrates the organisation and content of the corresponding memory pages;

Figures 6-15 illustrate various definitions and axioms;

Figures 16a-16e show a sequence of partitionings of a data space (left) and the corresponding BV-index structure (right);

Figure 16f shows the splitting of a promoted region, and

Figures 17 and 18 illustrate two examples of exact-match searching, and

Figure 19 illustrates, schematically, a computer system.

Modes for Carrying Out the Invention

The apparent intractability of the underlying problem, namely generalising the B-tree to n dimensions, is clearly related to the increased topological complexity of the n-dimensional case, and will be discussed in greater detail hereinafter.

Reference will, however, first be made to Figures 1 to 3, which whilst giving an example of a very simple 2-level index according to the invention, also illustrates the problem in view of the way in which Figure 2 is drawn. An n-dimensional data space is an n-dimensional space in which the domain of values of each coordinate corresponds to the domain of some data attribute. For example, in the case of an employee record: employee (id_number, age), this can be represented as a point in 2-dimensional space with coordinates representing id_number and age. The idea of any hierarchical index is to iteratively narrow down the search space within the whole data space. Each level of the index partitions the data space (or subspace of the data space) into smaller partitions. Whilst the subspaces can have any shape, in practice they should have simple shapes, such as rectangles, in order to be able to find a simple and compact representation for them.

In Figure 1-3 the subspaces of a two-dimensional

dataspace are generated by strict binary partitioning ie by repeatedly dividing the space exactly in half in alternate dimensions. If the left hand or lower half of a partition is represented by a zero, and the right hand or upper half by a 1, then every subspace can be identified by the sequence of 0's and 1's corresponding to the partition sequence which generated it. Subspace f, for example, has the unique representation 10110.

For clarity it assumed that the coordinates of only three points can be stored in each memory page. Thus, for the data distribution shown in Figure 3, the dataspace must partitioned, as show in Figure 2, into five subspaces, since each subspace corresponds to a memory page. If we also assume that an index page cannot store the representation of more than four subspaces in this case, then a two-level index is needed. The dataspace partitioning corresponding to the upper level index page (root node of the hierarchy) is shown in Figure 1. This page partitions the dataspace into two sets of two partitions at the lower index level. Figure 2 shows all the partitions together at the lower level. Figure 2 should actually be drawn as two separate diagrams representing two separate nodes of the index: one containing a and e, and the other containing f and g. This highlights the problem. Which one of the two should d go in? It is partly in both. In the BANG file, the partition d had to be split into two parts along the boundary of partition y. All other previously proposed designs have done something similar.

The trouble with splitting partition d into two parts is that the whole principle of "balanced" partitioning is lost. As discussed above, when a page representing a partition overflows, it must be divided into two so that both resulting partitions contain the same number

of entries, as far as possible. In the worst case a ration of 1:2 can always be achieved, as described below. If this principle cannot be guaranteed under all circumstances, then it is possible to devise arrangements where most of the partitions contain no more than a single entry. The number of pages in the index then explodes and is no longer proportional to the number of entries in the index. Hence the principle is crucial for fully predictable performance.

In the example of Figure 3, if d is "forced" to split along the boundary of y, then there is no guarantee that each of the resulting split partitions will contain at least one third of the entries, which is a requirement for a practical multi-dimensional arrangement.

The tree structure proposed herein and referred to as the BV-tree (from the German word "Begriffsverzeichnis" (concept index)) solves the problem simply by moving partition d one level up the index hierarchy, ie to the branch node below which d would have to be split otherwise.

Figure 4 shows the conceptual structure of the corresponding BV-tree. Each of the labelled boxes corresponds to a memory page, and an index branch corresponds to the address of the page to which the branch points.

Note, however, a particular feature of the BV-tree: subspace d - which directly represents a subspace of the dataspace at the lower partition level - has nevertheless been promoted to the higher level in the index. The reasons for this are explained later.

Figure 5 shows the actual organisation and content of

the corresponding memory pages.

In order to define precisely what is meant by partitioning a dataspace into subspaces or regions it is necessary to make a number of definitions. Referring firstly to Figures 6a and 6b.

Definition 1. A point X lies outside a closed boundary B in a dataspace D iff every straight line of infinite length which passes through X intersects B either zero or an even number of times on each side of X.

Definition 2. A point X lies inside a closed boundary B in a dataspace D iff every straight line of infinite length which passes through X intersects B an odd number of times on each side of X.

Referring next to Figures 7a and 7b.

Definition 3. Two points X and Y lie on opposite sides of a closed boundary B iff a straight line joining X and Y intersects B an odd number of times.

Definition 4. Two points X and Y lie on the same side of a closed boundary B iff a straight line joining X and Y intersects B either zero or an even number of times.

The axioms referred to above, are illustrated in Figures 8a and 8b.

Axiom 1. No boundary may intersect itself.

Axiom 2. No two boundaries may intersect.

Definition 5. A closed boundary B encloses a subspace D_S of a dataspace D iff every point X in D_S lies inside

boundary B.

Definition 6. A closed boundary B exclusively encloses a subspace D_s of a dataspace D iff B encloses D_s and B encloses no point which is not in D_s.

Definition 7. A partition region P of a data space D is any subspace D_s which is exclusively enclosed by a finite or infinite closed boundary B(P).

The data space D is itself a partition region, whose closed boundary B(D) is delineated by the finite or infinite upper and lower limits of its data domains or ranges.

Definition 8. A partition region P_a encloses a partition region P_b iff every point in P_b also lies within P_a. Thus: $P_a \cap P_b = P_b$ and $P_a \cup P_b = P_a$. (See Figure 9).

Definition 9. A partition region P_a directly encloses a partition region P_b iff P_a encloses P_b and there exists no other partition region P_c which encloses P_b and which is enclosed by P_a. (See Figure 10).

Definition 10. A partition region P_a strictly encloses a partition region P_b if P_c encloses P_b and there exists no point which is common to the boundaries of both P_a and P_b. (See Figure 11a).

Definition 11. A partition region P_a tangentially encloses a partition region P_b if P_a encloses P_b but P_a does not strictly enclose P_b ie there exists at least one point which is common to $B(P_a)$ and $B(P_b)$. (See Figure 11b).

It follows from axiom 2 that no two partition regions P_a and P_b may intersect, unless one encloses the other: $P_a \bigcap P_b \in \{\emptyset, P_a, P_b\}$.

Definition 12. A logical region L_a is the difference between a partition region P_a and the set of partition regions $S = \{P_{b1}, P_{b2}, \ldots, P_{bn}\}$ which it directly encloses:

$$P_{bu} \bigcap P_{bv} = \emptyset \text{ for } u,v = 1,\ldots,n; u \neq v$$
$$P_a \bigcup P_{bu} = P_a \text{ for } u = 1 \ldots n$$
$$L_a = P_a - \bigcup_{u=1}^{n} P_{bu}$$

It follows from this definition that a logical region is composed of one or more disjoint sub regions, each of which may have one or more internal boundaries as well as an external boundary (see Figures 12, 13a and 13b):

Definition 13. The external boundary $B_e(L_a)$ of a logical region L_a is the set S_B of boundaries which exclusively enclose the m disjoint sub regions L_{au}, u = 1....,m which comprises L_a ie:

$$L_a = \bigcup_{u=1}^{m} L_{au};\ B_e = \{B(L_{a1}), B(L_{a2}), \ldots, B(L_{am})\} \quad (1)$$

Definition 14. The internal boundary of a logical region L_a is the set of boundaries of the union of the partition regions which P_a directly and strictly encloses.

Note that, according to these definitions, a logical region always spans the space which it encloses ie no point which is enclosed by a logical region can at the same time be enclosed by another logical region. In contrast, a partition region does not, in general, span

the space which it encloses. Note also that, given two partition regions P_a and P_b such that P_a encloses P_b, it does not necessarily follow that logical region $L_a \equiv P_a - P_b$ will enclose $L_b \equiv P_b$.

Now consider partitioning the partition regions. Referring to Figures 14a and 14b, suppose that a data space D is partitioned into a set of n partition regions $Sp = \{P_1, P_2, \ldots, P_n\}$ (Figure 14a) according to the definitions and axioms above:

Suppose further that the set S_p is itself to be partitioned into two sets S_{pa} and S_{pb}, such that S_{pa} and S_{pb} are enclosed within partition regions P_a and P_b respectively ie:

$$P_a \cup P_b \subseteq D; \quad S_{pa} \cup S_{pb} = S_p; \quad S_{La} \cap S_{Lb} = \emptyset \quad (2)$$

If all the partition regions are disjoint, then it is always possible to divide the set of partition regions into two halves ie to partition S_p so that the cardinalities of S_{pa} and S_{pb} do not differ by more than 1 (Figure 14b). If, however, some partition regions enclose others, then the situation is not so straightforward. Intuitively, one can see that, the more disjoint partition regions here are, the better the chance of achieving a 1:1 split. The worst case arises when there are no disjoint partition regions ie every partition region either encloses or is enclosed by another (Figure 15a). Since, according to axiom 1, region boundaries may not intersect, the set S_p must be partitioned into sets, one of which enclose the other (Figure 15B) ie:

$$P_a \cup P_b = P_a; \quad P_a \cap P_b = P_b;$$
$$\forall P_i \in S_{pa}, \forall P_j \in S_{pb}, \; P_i \cup P_j = P_i \quad (3)$$

In this case, the algorithm for finding the best-balance partition proceeds as follows: suppose that, in general, La and L_b contain fractions $f(L_a)$ and $f(L_b)$ respectively of the total number of partition regions in the dataspace ie $f(L_a) + f(L_b) = 1$. Initially, however L_a encloses only the outermost member of S_p, with the remainder enclosed by L_b. If more than one member of this remainder enclosed by L_b (ie P_b directly encloses a number of disjoint partition regions), and if one of these enclosed members P_i itself encloses more than half of the total membership of S_p, then the set S_p is repositioned such P_b is directly enclosed by P_i.

If none of the partition regions directly enclosed by P_b enclose more than half of the total membership of S_p, then it may be possible to obtain a better balance between L_a and L_b by a further repositioning of P_b which has the effect of transferring one or more members of the set from L_b to L_a. The complexity of such a balancing algorithm is, however, exponential in the size of the set. But the partitioning ratio need never fall outside a worst-case ration of 2:1. The proof below also provides the framework of an algorithm to maintain this ratio:

Given that $f(L_a) < 1/3$, then $f(L_b) > 2/3$. Suppose now that P_b is notionally replaced by two arbitrary partition regions P_b' and P_b''. Then two distinct cases arise:

(a) $f(L_b') \leq 2/3$ and $f(L_b'') \leq 2/3$

(b) $f(L_b') > 2/3$ or $f(L_b'') > 2/3$

In both cases, P_b is repositioned to coincide with whichever of the two (notional) partition regions P_b' and

P_b' encloses the larger fraction. Since $f(L_b') + f(L_b'') > 2/3$, this larger fraction must be greater than 1/3. Therefore, in case (a), $1/3 < f(L_b) \leq 2/3$. In case (b), it remains true that $f(L_b) > 2/3$. However, P_b must still directly enclose more than one member of S, because no member which P_b directly encloses can itself comprise a fraction greater than 1/2. It is therefore possible to repeat the (notional) repartition operation on P_b until case (a) becomes true - which it must eventually do, if only one member remains directly enclosed by P_b.

This is by no means a new result. It is emphasised here that it is a consequence of topology and it is quite independent of any particular data model. The proof may not be transparently clear that the result is almost intuitively obvious, This intuition relies on topological observations: a superimposed data model usually serves only to obscure.

Now consider partitioning logical regions. If a partition region P_a encloses partition region P_b, and a further partition boundary is introduced between those of the other two, it has no effect on the extent of the dataspaces enclosed by P_a and P_b, because partition regions are completely defined by their (external) boundaries.

In contrast, the logical region L_a is changed, since it has an internal boundary which is defined by any partition boundaries which it directly encloses. (L_b therefore remains unchanged). In general, if a new partition region P_b is introduced directly inside P_a, then L_a will be split into two logical regions L_a' and L_b, where $L_a = L_a' \cup L_b$. The ultimate objective - the construction of an index - requires partitioning of a

dataspace into a set of non-intersecting subspaces, each of which contains as nearly as possible an equal number of data points. It is therefore clear that, with the definitions given above, these subspaces must be logical regions, since logical regions do not intersect. Note, however, that logical regions are defined in terms of partition regions. It is therefore possible to represent the subspaces of a dataspace either directly, as logical regions, or indirectly as a set of partition regions: a data point which lies in a particular logical region also lies unambiguously in the partition region which directly encloses the point. Thus, in the latter case, a search for a target data point simply looks for the smallest partition region which encloses the point.

The problem of representing and indexing a set of objects with spatial extension has not been considered. However, a fundamental problem which this raises cannot be avoided when considering a hierarchy of subspaces, since the subspaces themselves have spatial extension.

Now consider the construction of a hierarchical index to the logical regions of the dataspace. Each branch and leaf node in the tree-structured index represents a logical region, and contains a representation of the set of logical regions which it encloses. Associated with each region is a pointer to an index node at the next level below in the tree structure. The branch nodes contain sets of regions. The pointer associated with a logical region in a leaf node points to a data node which contains a set of data points lying within that logical region of the data space. (If a partition region representation is chosen, then each pointer in the tree is associated with a partition region rather than a logical region, but the index nodes still represent logical regions). No other restrictions are

imposed on the representation of the index, except that the number of entries in each tree node and data node (the fan-out ratio) has a fixed upper limit F.

Thus, as with a B-tree, the search for a data point in the data space proceeds downwards from the root of the tree, searching at each index level for the logical region which encloses the point, and following the associated pointer down to the next level.

Also, as with a B-tree, the index grows upwards as the data nodes at the bottom of the index hierarchy overflow and split. Provided that all the data points are unique, a partition can always be found which divides an overflowing data node into two half-full nodes. When an index node overflows, it is always possible to partition the logical regions which it contains in a ratio not worse than 1:2, as proved above. This last property is the general n-dimensional case, contrasting with the 1:1 splitting ratio achieved by the B-tree in the special, one-dimensional case.

However, a fundamental problem associated with the splitting of the index nodes has so far been ignored. Although the recursive partitioning procedure described above does not violate any of the axioms or definitions laid down earlier, it nevertheless introduces an ambiguity (the spanning problem) when interpreting a logical or partition region as a continuous subspace.

Suppose that logical region L is to be partitioned into two regions L_a and L_b, such that $L_a \bigcup L_b = L_a$. If L corresponds to a data node, then the data points lying within L_b can be transferred to the new node representing L_b. But if L is a logical region at a higher index level, then it contains a set of logical

regions S_L of the next lower level. In this case, the boundary of P_b cannot be located arbitrarily, since it may easily cross a boundary of one or more of the members of S_L, thereby violating the constraint that no two boundaries may intersect. Nevertheless it is known that it is always possible to find a position for the boundary of P_b such that the sizes of S_{La} and S_{Lb} do not differ by a ratio great than 2:1.

But there is no guarantee that the boundary of P_b will not be enclosed by one of the members P_{ai}, of S_{Pa}. (For example, the boundaries of all the members of S_p may be nested one within another). P_{ai} must then itself be partitioned along the boundary of P_b. If P_{ai} itself corresponds to an index node, then the same situation may be repeated at the next lower index level, and so on until a data node is reached - which can always be split without invoking any further chain reaction of partitioning.

Note that the partition boundary of the initially invoked partition of L is the boundary used for all subsequent partitions at the lower index levels ie only the position of the initial partition can be freely chosen to maintain the 2:1 partition ratio. It can be shown that the basic axioms will not be violated, that is the partition boundary will never intersect boundaries of regions at the lower index levels. But severe partitioning imbalances may occur at the lower levels. Worse, a lower level node split may trigger overflow in an index node above, which in turn may trigger a further cascade of downward splits.

This is a condition which haunts all multi-dimensional index methods which are based on balanced tree structures. Most of the complexity and ingenuity in

their designs is aimed at circumventing or minimising this problem.

When logical region is L is partitioned, the index node which contains the representation of L is modified: the representation of L is replaced by the representation of L_a and L_b. There is ab initio no inconsistency within region L_{ai}, or within any of the regions which it encloses at lower index levels. It only needs to be split in order to ensure the correct behaviour of the search path through the new index branch created by L_b. But there is an alternative strategy: to leave L_{ai}, intact, and modify the search path.

Suppose that L_a and L_b contain the sets of logical regions S_a and S_b respectively. If L_{ai} is split into L'_{ai} and L''_{ai} along the boundary of P_b, then:

$$L_a = \bigcup_{u=1}^{n} L_{au} \bigcup L'_{ai},\ 1 \leq i \leq n,\ u \neq i \qquad (4)$$

$$L_b = \bigcup_{v=1}^{m} L_{bu} \bigcup L''_{ai},\ 1 \leq i \leq n \qquad (5)$$

But since $L_a \bigcap L''_a = \emptyset$ and $L_b \bigcap L'_{ai} = \emptyset$ then:

$$L_a = \bigcup_{u=1}^{n} L_{au} \bigcup L'_{ai} \bigcup L''_{ai},\ 1 \leq i \leq n,\ u \neq i \qquad (6)$$

$$= \bigcup_{u=1}^{n} L_{au} \bigcup L_{ai},\ 1 \leq i \leq n,\ u \neq i$$

$$L_b = \bigcup_{v=1}^{m} L_{bu} \bigcup L''_{ai} \bigcup L'_{ai},\ 1 \leq i \leq n \qquad (7)$$

$$= \bigcup_{v=1}^{m} L_{bu} \bigcup L_{ai},\ 1 \leq i \leq n$$

This shows that, rather than split L_{ai}, it can be left intact if the search path can be changed so that L_{ai} is always searched if the search has failed in L_a or L_b respectively. All that is required is that, before an index node is split according to the partitioning algorithm, the node is searched for an entry which represents a partition boundary P_{ai} which directly encloses the splitting boundary P_b. If P_{ai} exists, then it is not split, but promoted to the index level above, along with the entries for P_a and P_b. The subtree of P_{ai} is automatically promoted with it. A marker is attached to the promoted entry, to distinguish it from an entry created by direct partitioning.

The consequences of this promotion policy on the construction of and access to a BV-tree are developed and demonstrated in the following example.

Figures 16a to 16e above show a sequence of partitionings of a data space (left) and the corresponding BV-index structures (right). 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 be intersect. The data space may finite or infinite.

Initially, there is a single partition region, which is the whole data space. Conceptually, a number of points are inserted into this region (Figure 16a). (In practice these points are correspond to data records). No assumptions are made about how the points are represented, only that there is a maximum number P which can be inserted into any one partition region.

On the right of Figure 16a is a data node, which represents the partition region containing the points.

(In practice this corresponds to one or more pages of computer memory, in which the records are stored).

When the number of points inserted into a partition region exceeds P, the region must be split into two. It can be shown that 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 16b shows a data space after the first overflow and split. An index node has been created which contains two entries, each of which is a unique key identifying one of the partition regions. Associated with each index node entry is a pointer to the data node which represents the partition region and its enclosed data points. Again, no assumptions are made about the way in which the identifying keys of the partition regions are generated or represented. It is assumed 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 partition region keys, and the efficiency of operations on this representation. For clarity, the data points themselves are not shown in subsequent figures.

Assume, without loss of generality, that the fan-out ratio F of the index nodes is four. Figure 16c shows the situation after the creation of three additional partition regions with keys e, f, g. This causes the index node of Figure 16b to overflow. The regions represented in the node with keys a, d, e, f and g are then partitioned into two regions of regions x and y,

which form the root node of a new, two-level index. The left part of Figure 16c shows the partition regions represented at the upper and lower levels of the index.

The right part of Figure 16c also shows that the index split has been accompanied by the promotion of entry d to the level above. Whenever the boundary of a split (in this case the boundary of region y) does not coincide with an existing region boundary, but is itself enclosed by one or more region boundaries, then that region whose boundary directly encloses the boundary of the split (in this case region d) is promoted to the index level above. If d is the root of a subtree, then the subtree is automatically promoted with it.

During traversal operations on the tree structure, promoted entries are treated differently from non-promoted entries. Thus promoted entries are marked as such. This marking is indicated in the diagram (Figure 16c right) by the shading of the entry for region d.

Figure 16d shows the result of a second overflow and split. Region x has overflowed and split into x and v. This has caused region e - which directly encloses v - to be promoted.

Finally, Figure 16e shows the result after several further splits, which have finally caused the creation of a third directory level. In particular, the creation of the new root node has been accompanied by the promotion of both d and y from the level below. This illustrates the general rule that, when a region p splits into regions p and q, such that p encloses q, then any entries at the index level below - whether already promoted or unpromoted - which represent regions

which directly enclose q, are promoted to the level of the entries for p and q. [NB Two regions a and b can both directly enclose a third if a and b are from different index levels].

A promoted region p in node n can also overflow and split into two regions p^1 and p^{11}. Both resulting regions are then treated in the same way. Thus if p now represents either p^1 or p^{11} then:

if p directly encloses an unpromoted region in index node n, then it remains in the node (unless the insertion of an additional entry causes the node to overflow). Otherwise, a search is made within node n for the entry e - of any promoted level lower than that of p, or unpromoted - which directly encloses p. p is then inserted, according to the same algorithm, in the node to which entry e points ie if it does not enclose an unpromoted region then it will be further demoted, until it reaches its own original index level.

For example, suppose region d in Figure 16e overflows and splits as shown in Figure 16f. In the root index node, entry d^{11} directly encloses unpromoted entry q, and so d^{11} is inserted in the root node. d^1 encloses no entries in the root node, so an attempt is made to insert it into the node to which p points, since p directly encloses d^{11}.

The algorithm is then repeated: d^1 encloses v, but not directly, since e encloses v directly. Therefore d^1 is inserted in the node to which x points, since x directly enclosed d^1. The algorithm now terminates, because the unpromoted level of entry d^1 has now been reached.

Now consider an exact-match search. It requires an

algorithm which consists of a combination of a forward and backward search of the index tree. The forward search starts at the root of the tree. A search is made in the root node for the smallest region which encloses the specified data point (ie matches the search key). If such a region is found, then the associated pointer is followed down to the next level of the index tree, and so on, until a data node is reached. The data node is then searched for a data point which matches the specified data point. It a match is found, the search succeeds. Otherwise it fails. In either case, the search then terminates.

Superimposed on this forward search is a search which backtracks over the forward search path. When a matching entry is found at any index level l during the forward search, the search backtracks to the root, searching at each higher level i($1 \leq i < L$) for a promote region from level L which is directly enclosed by the non-promoted matching entry level i, and which is a better match (ie smaller region) than any previously found matching region at any level. If no such better match is found during backtracking, the forward search resumes from the point at which the backtrack was initiated ie at the node (at level L+1) pointed to by the last non-promoted entry found at level L. Otherwise, the forward search is restarted at the node pointed to by the best-match entry. Note that this node will also be at level L+1.

If the forward search fails at a branch node at level L, then the backward search is again initiated, but this time a search is made at each higher level i($1 \leq i < L$) for a promoted region from level L which directly encloses the non-promoted matching entry at level i. The backtrack search halts as soon as the first such

matching entry is found, and the forward search resumes at the node to which it points. Again, this node will be at level L+1. If this backtrack search fails to find any matching entry at any level, then it exits from the root and the search fails and terminates.

Two examples of exact-match searching are given in the following.

Referring firstly to Figure 17, suppose that there is a data point located at position P in a data space partitioned by a three-level index as shown in Figure 17. A search for P then proceeds as follows:

Starting from the root, the key generated from data point P matches entry p - and only entry p - in the root node. The search then follows the pointer associated with entry p down to index level 2. At this level, the search key matches only x.

At this point, the backtrack search is invoked: a search is made at level 1 among the entries promoted from level 2. y is the only candidate in this case. y is directly enclosed by p, but it does not enclose point P. The backtrack therefore fails at this level, and since there are no higher levels, it fails altogether, and the forward search resumes at level 3, at the node pointed to by x.

a is the only matching entry for P at level 3. Backtracking is again invoked: a search is made at level 2 among the entries promoted from level 3. e is the only candidate. It is directly enclosed by x, it encloses P, and it is smaller than a - the previous best match. e thus becomes the current best-match. Backtracking however continues. At level 1, d is the

only candidate. But since it is not smaller than e, e remains the best match. Having reached the root, the backtrack search stops and the forward search resumes at the node pointed to be e. This is a data node, and so the search succeeds when the point P is found in this node.

Referring now to Figure 18, suppose point P is now located as shown in Figure 18. The search for P proceeds as follows:

Starting from the root, the key matches entry q. But at level 2 the search fails, because P does not lie in either z or w. This causes backtracking to be invoked. At level 1, y is the only candidate entry promoted from level 2. It also directly encloses q, the matching unpromoted entry in level 1. The backtracking thus halts, and y becomes the new starting point for the forward search. But the search fails again at level 3, since P does not lie in either f or n. Backtracking is thus again invoked. At level 1, d is the only candidate entry promoted from level 3. It directly encloses q, and it encloses P. Backtracking again halts, and the forward search continues from d. Since the node pointed to d, is a data node, the search succeeds when the point P is found in this node.

There is a price to be paid for this solution: the index tree is no longer balanced. At first sight, this violates the most sacred principle of hierarchical indexing: without a fixed length search path from root to leaf, the worst-case behaviour for exact-match queries becomes uncontrollable. However, what is really important is not so much the path length as the number of different tree branch nodes traversed. These two will be different if a search path involves

back-tracking. The total path length must have a predictable upper limit, but it is a reasonable practical assumption that, with standard caching techniques, there will be no extra retrieval cost in re-visiting a node which lies along the direct path from root to leaf.

Consider the downward and upward parts of as exact-match search path, according to the search algorithm described above. A downward search will fail if the target location lies within a promoted logical region, and the failure will occur at the level from which the region was originally promoted. But the sum of all the nodes visited during the downward traversal phases of an exact-match search will remain constant for a given tree size. Likewise, although a promoted subtree may be further promoted to any higher level in the tree, and a subtree of a subtree may also be promoted, the total height through which promotion of any logical region can occur cannot be greater than the height of the tree. Therefore, the number of different branch nodes visited in an exact-match search is always equal to the height of the tree.

If any subtree could be arbitrarily promoted to a high level, a crucially important property of a balanced tree structure would be lost: the logarithmic relationship between the number of data nodes and the path length from root to leaf of the index. This relationship first of all requires that the direct path length from root to leaf is fixed, for a given state of the index tree ie that the tree is perfectly balanced. It also requires that the fan-out ratio in every index node (except the root) must never fall below a fixed minimum (which must be at least an order of magnitude greater than 1). The linear relationship between the number of index nodes

and the number of data nodes is a further consequence of these requirements.

However, the promotion of a subtree only occurs under the circumstances described above, and only when an index node overflows and splits. Therefore, since every index node except the first is created by splitting, there can never be, overall, more promoted index entries than unpromoted entries, unless through the splitting of an entry which is already promoted, since this will create a second entry at the same (promoted) level.

It could also cause a further promotion from the level below. Fortunately, it can be proved that it is always possible in such cases to return all but one of the entries resulting from such a split to their originating (unpromoted) positions in the index tree. It is also possible to prove that this requires no more node accesses than would have been necessary if the entries had not been promoted ie if the tree had been balanced.

This does not however means that there are never more promoted entries than unpromoted entries in an index node. It was explained above that, when a region p splits into regions p and q, such that p encloses q, then any entries at the index level below - whether already promoted or unpromoted - which represent regions which directly enclose q, are promoted to the level of the entries for p and q. This effect can thus become cumulative as entries are promoted up the index tree. In general, the maximum possible number of promoted entries in a node at level L of an index tree of maximum depth d is n(d - L), where n is the number of unpromoted entries in the node.

So, 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 index nodes at level L of a BV-tree need to be increased in size by a factor of (d - L + 1). (Note however that the occupancy of the index nodes will still not fall below 1/3). 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 1/3, compared to 1/2 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. The worst-case occupancy of a data node remains at 1/2 in the n-dimensional case, provided that for any given attribute of a tuple, no two tuples have the same attribute value. Otherwise, the worst case occupancy falls to 1/3.

The BV-tree (hierarchical index of memory pages) presented above organises the contents of the memory pages, and the access paths between them, in such a way that the following properties are satisfied and guaranteed under all circumstances.

1. each page corresponds to a representation of a unique subspace of the data space;

2. the time taken to access the page which holds the

data representing a single, specified point in the corresponding data space depends only on the total number of pages used to represent the whole space, and increases only logarithmically with this number. In particular, the time is independent of the point chosen;

3. the data occupancy of every page never falls below a fixed and acceptable minimum (1/3);

4. the distribution of points in the data space is reflected closely in the distribution of the data in the memory pages ie adjacent points in the data space are represented within a single memory page, as far as possible;

5. the time taken to insert, delete or modify a specified point in the data space in the corresponding memory pages depends logarithmically on the total number of pages;

6. the performance of access, insertion, deletion and modification operations does not degrade as a consequence of the passage of time and/or degree of usage.

The BV-tree represents the recursive partitioning of the data space into a hierarchy of subspaces. A feature of the BV-tree is that some entries in the index tree can be "promoted" to a higher level of the tree, so that the length of a direct path from top to bottom of the tree is not necessarily independent of the path chosen. Nevertheless, the number of different nodes of the index hierarchy which are visited along the search path for an individual point in the data space is always constant, for a constant total number of points.

In order to be able to detect which index entries have been promoted in this way, each index entry is "labelled" with its "level number". This is the level of the index tree at which the entry was "created". Since an expanding tree grows by splitting the root, the entries in the lowest (inverted) tree nodes are labelled level 1, and the entries in the root node have the highest level so far created.

In essence, the search algorithm to find a point in the index is unchanged from the classical exact-match tree search, as in the B-tree. A search is made iteratively at each level of the hierarchy, from the root downwards, for the "smallest" partition in which the point lies. The critical difference from every other such design in that the index "tree" no longer directly corresponds to the index "hierarchy".

In Figure 17, for example, only those entries labelled as level 3 of the index hierarchy are initially examined at the top level of the index tree. In this example these are entries p and q. Entries d (level 1) and y (level 2) are ignored. Suppose a match is found with entry p. The search then iterates to the next lower hierarchical level. The entries at the next lower "hierarchical" level are x, v and y. That is all the entries traversed so far which are labelled level 2, despite the fact that they are not all at the same index "tree" level. The problem involved with splitting y is avoided by not splitting y. Instead y is included in the search, whichever side of the index tree the search descends. The fact that d has actually been promoted through the levels in Fig 17 is a consequence of recursively applying the idea to the whole index. If, for example, the target point lies in partition v at hierarchical level 2, then the partitions searched at

level 1 are s, r, e and d, i.e. all of the entries of level 1 in the nodes traversed during descent of the index tree. It should be noted that if a match is found with partition v, it will also match partition x which encloses v. This ambiguity is resolved, however, because the "smallest" matching entry in the index is the one taken i.e. the most deeply nested one.

The solution to the multidimensional problem, with which the invention is concerned, provided by the BV-tree is a surprising one. It had always been assumed that the index tree must be exactly balanced i.e. that the direct path from root to leaf must be the same for all possible paths. This was to ensure that the number of tree nodes visited was the same for all direct paths. It was always assumed that the two things automatically went together. The BV-tree shows that they do not. It does not matter how unbalanced the tree structure is, if the search path up and down the tree is so arranged that the number of different nodes visited is always the same.

The positions of the partitions, their shapes and the way in which they are represented does not affect the principle of the invention in any way. The only restrictions on the partitions are given by axioms 1 and 2. These axioms apply to all the partitions in the index, taken together, whatever level of the hierarchy they may belong to. The strange-shaped partitions of Fig 16 were deliberately chosen to emphasise the above. Usually, index designs do not distinguish between the "principles" of the design and a specific "implementation" of it. This is usually because the design is heavily implementation dependent. In practice partition shapes like those in Fig 16 are impractical because it is difficult to find a compact and precise representation of such shapes. Hence rectangular boxes

are usually chosen. These can be represented in many ways. The most obvious is by specifying the coordinates of the bottom left-hand and upper right-hand corners. This way a box of any size and rectangular shape can be represented. An alternative representation is shown in Figs 1-4. The data space there is partitioned by progressively dividing it up precisely in half in alternate dimensions (binary division). This generates a unique binary identifier for each partition as shown.

For the sake of completeness the differences between the BANG file and the BV-tree will now be repeated. It should be understood, however, that the invention is not simply an extension of the BANG file. The feature of promoting index entries upon partition splitting is equally applicable to other indexes, with corresponding beneficial effects.

In principle the index hierarchy is the same for both the BANG file and the BV-tree. The index tree is, however, different. The binary partitions of the data space used in the BANG file, and the BANG file's representation of these binary partitions as variable-length strings, could also be used in one possible representation of the BV-tree. As is apparent from the foregoing the partitioning does not have to be binary, as evidenced by the strange-shaped partitions of Fig 16 etc. The BV-tree can also be described, for example, in terms of a range-based representation in which the partitions are described by an arbitrary sequence of divisions at any point in the range of a dimension of the data space. There are many possibilities. The binary domain-based partition representation of the BANG file is not the only way of representing a BV-tree.

The iterative exact-match index search algorithm down the tree is also essentially the same in the BANG file and the BV-tree. The basic difference is that the index entries of a hierarchical level L in the BANG file are all stored at level L in the tree, whereas in the BV-tree some entries of hierarchical level L may appear at index tree levels higher than L.

In the BANG file every exact-match search follows the shortest path from the root to one of the leaves of the index tree. The number of tree nodes visited is the same for all such paths, and is equal to the number of levels in the tree hierarchy, i.e. the height of the tree, which is fully balanced. In the BV-tree, however, every exact-match search visits the same number of different tree nodes, and this number is equal to the longest direct path from root to leaf of the index tree, the maximum height of the tree being h.

In the BANG file and the BV-tree, it is always possible to split an overflowing partition, at any index tree level, so that each of the resulting partitions contains at least one third of the contents (data or partitions) of the original partition. However, in the BANG file it is not in general possible to do this without also splitting at least one other partition at the lower index level, and in this case it is not in general possible to maintain the above splitting balance. The BV-tree avoids this problem by promoting partitions.

The BV-tree provides the following advantages:

1. In circumstances where the pattern of (relational) database queries is unpredictable, the average response time can be an order of magnitude faster than with a conventional 1-dimensional index. (The

n-dimensional index can still be used as a 1-dimensional index where there is a fixed query pattern).

2. It allows n-dimensional indexing in 0(n/B), exact-match query I/O time 0(log n base B) and worst-case update time 0(log n base B), where n is the total number of indexed data points and B is the maximum number of index entries per page.

The hierarchical index structure, database structure, indexing and searching techniques described above are capable of being used in any computer system including a memory, in particular a one-dimensional memory, ie a memory consisting of a linear sequence of pages. The computer system may be as illustrated schematically in Figure 19, which comprises a computer (processor) 1, with a memory device 2, an input device 3 and a display device 4.

CLAIMS

1. A hierarchical index structure characterised in that, in the event of data distribution changes or skewing of insertion/deletions to particular areas of a dataspace, at least one index entry partition (d) is disposed at a higher index tree level than the hierarchical level to which it corresponds.

2. A structure as claimed in claim 1 characterised in that index entry partitions are labelled with the level at which they were created whereby to facilitate detection of higher-index-tree-level-disposed partitions.

3. A structure as claimed in claim 1 or claim 2 characterised in that a said higher-index-tree-disposed partition corresponds to a partition whose boundary directly encloses a split partition created when an index node was caused to overflow upon the introduction of extra information to a database, rather than the split partition itself.

4. A database structure for multi-dimensional information including a hierarchical index structure and wherein a data space can be recursively partitioned into subspaces with corresponding index entry nodes arranged in an index tree, characterised in that, in the event of data distribution changes or skewing of insertion/deletions to particular areas of a dataspace, at least one index entry partition (d) is disposed at a higher index tree level than the hierarchical level to which it corresponds.

5. A database structure as claimed in claim 4 characterised in that index entry partitions are labelled with the level at which they were created whereby to facilitate detection of higher-index-tree-level-disposed partitions.

6. A database structure as claimed in claim 4 or claim 5 characterised in that a said higher-index-tree-level-disposed partition corresponds to a partition whose boundary directly encloses a split partition created when an index node was caused to overflow upon the introduction of extra information to the database, rather than the split partition itself.

7. A method for use in indexing a dataspace, the dataspace comprising a database for multi-dimensional information, the indexing having a hierarchical structure and the dataspace being recursively partitioned into subspaces with corresponding index entry nodes arranged in an index tree, characterised in that upon the introduction of extra information causing an index entry node to overflow, the index entry node is split into two split partitions, which are disposed at a respective level of the index tree, and a respective index entry partition (d) whose boundary directly encloses a said split partition is disposed at a higher level in the index tree than the hierarchical level to which it corresponds.

8. A method as claimed in claim 7 characterised by labelling index entry partitions with the level of the index tree at which they were created whereby to facilitate the detection of index entry partitions disposed at higher levels in the index

tree than the hierarchical level to which they correspond.

9. A method of searching a database structure containing multi-dimensional information in a dataspace and including a hierarchical index structure, the dataspace being recursively partitioned into subspaces with corresponding index entry nodes arranged in an index tree, wherein at least one index entry partition (d) may be disposed at a higher index tree level than the hierarchical level to which it corresponds as a result of data distribution changes or skewing of insertions/deletions to particular areas of the dataspace, characterised by searching iteratively the index tree hierarchy for the smallest partition in which an item of information sought is located, the search beginning at a root entry node and examining each hierarchical level in turn, and the search involving backtracking between levels in the case of index entry partitions not disposed at the index tree level corresponding to their hierarchical level, the number of different nodes visited along the search path for the item of information sought being constant for a constant total number of items of information contained in the dataspace.

10. A computer including a memory employing a hierarchical index structure characterised in that the hierarchical index structure is such that, in the event of data distribution changes or skewing of insertion/deletions to particular areas of a dataspace, at least one index entry partition (d) is disposed at a higher index tree level then the hierarchical level to which it corresponds.

11. A computer database system including a memory employing a hierarchical index structure, characterised in that the hierarchical index structure is such that, in the event of data distribution changes or skewing of insertion/deletions to particular areas of a dataspace, at least one index entry partition (d) is disposed at a higher index tree level than the hierarchical level to which it corresponds.

12. Method for electronic storage and/or transfer of an n-dimensional entity represented in an n-dimensional data space into one-dimensional memory of a computer comprising the steps of partitioning the data space into sub-spaces or regions, splitting a first region into the first region and at least one further region such that the first region encloses the further region(s) if the number of points inserted into said region exceeds a predetermined number,
storing the regions represented by data notes into one or more pages in the memory of the computer,
attributing an index identifying a region to each region,
storing the indices into one or more index pages in the memory of the computer,
splitting an index page if the number of indices representing the regions exceeds a predetermined number,
connecting said split index page by pointers to the several index pages and
storing said several index pages in the memory of the computer.

13. Method of retrieving an n-dimensional entity stored in the form of an n-dimensional data space in a

one-dimensional memory of a computer, according to the method described in claim 12.

14. Apparatus for electronically storing and/or transferring an n-dimensional entity in a one-dimensional memory of a computer working according to the method of claim 12.

15. Apparatus for retrieving an n-dimensional entity stored in the form of an n-dimensional data space in a one-dimensional memory of a computer working according to the method of claim 13.

Fig.1.

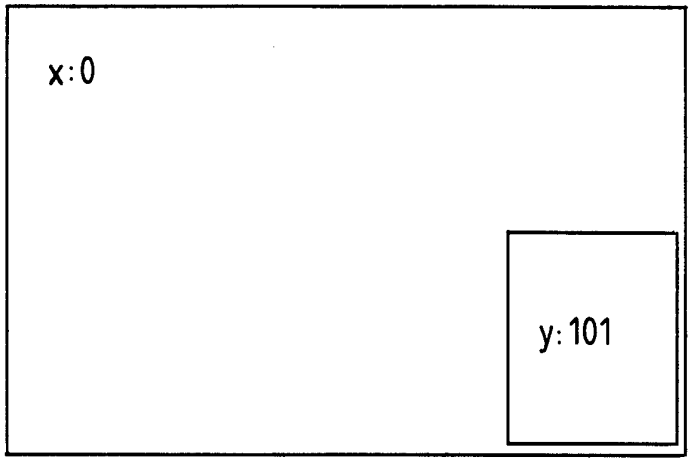


Fig.2.

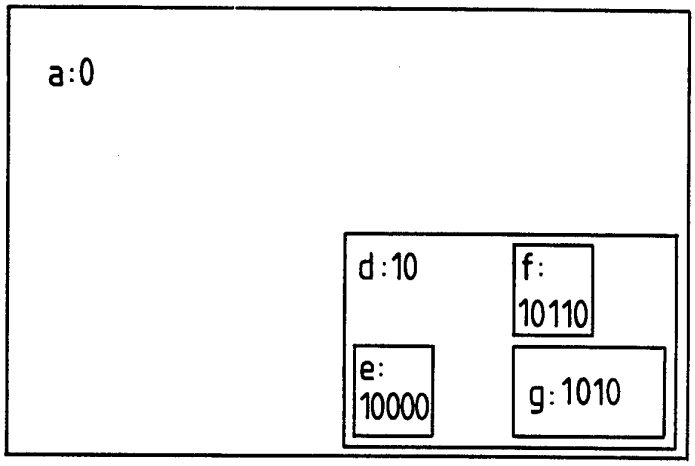


Fig.3.

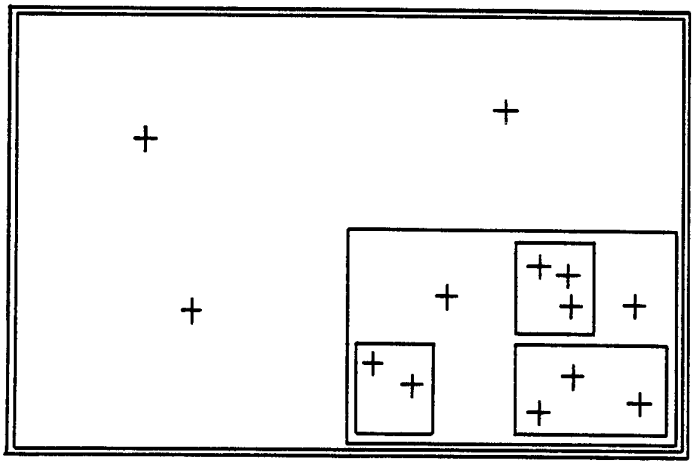


Fig.4.

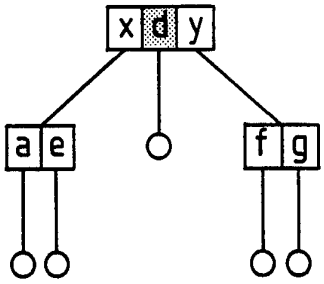


Fig.5.

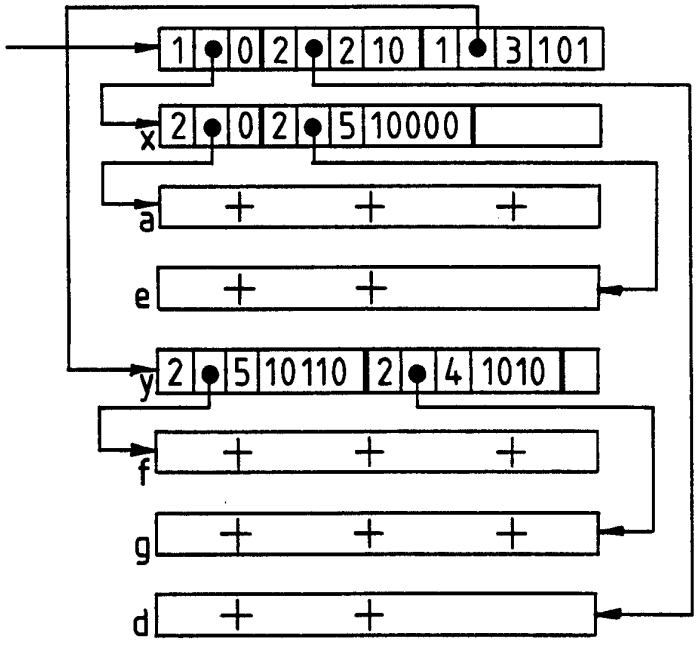


Fig.6a.

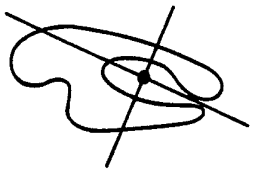


Fig.6b.

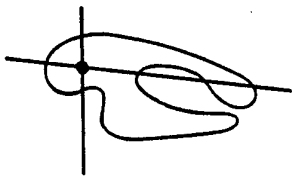


Fig.7a.

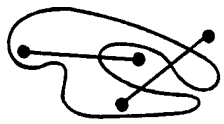


Fig.7b.

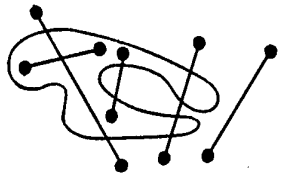


Fig.8a.

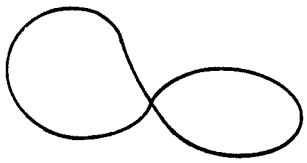


Fig.8b.

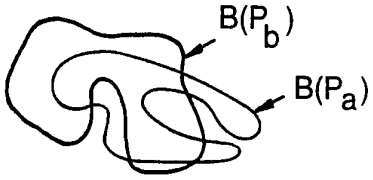


Fig.9.

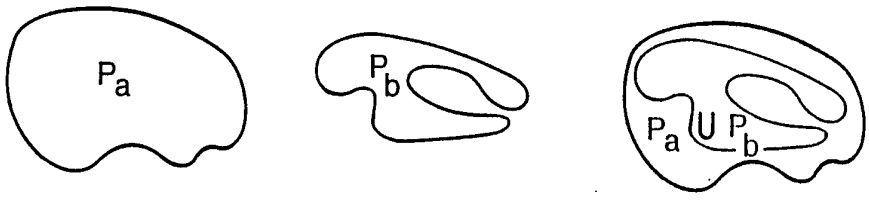


Fig.10.

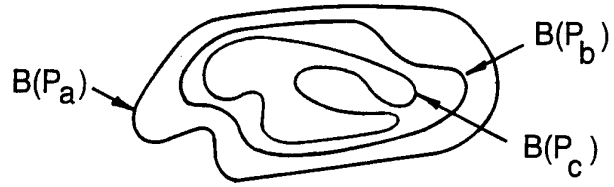


Fig.11a.

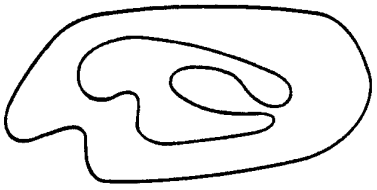


Fig.11b.

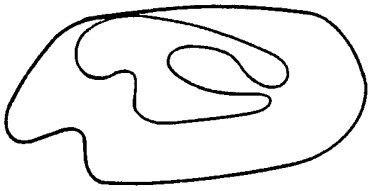


Fig.12.



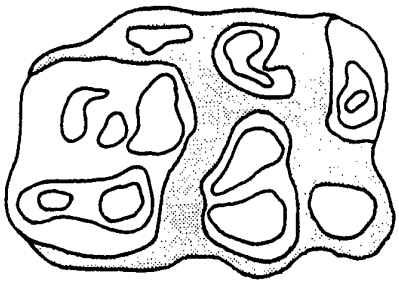
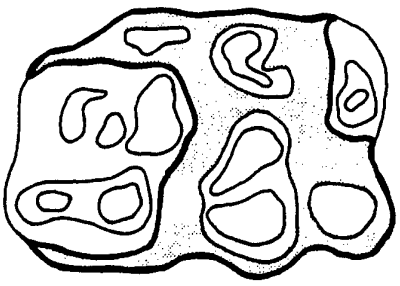
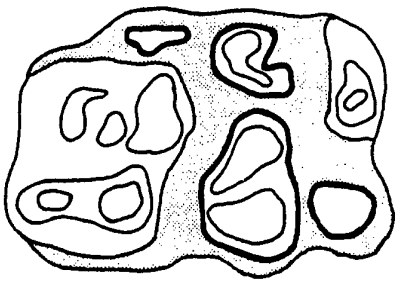


Fig.13a.



External boundary of Logical region L_a

Fig.13b.



Internal boundary of Logical region L_a

Fig.14a.

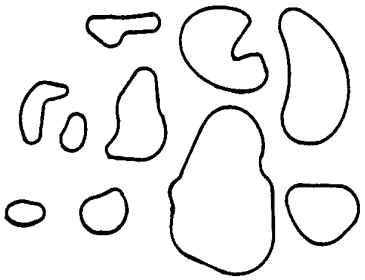


Fig.14b.

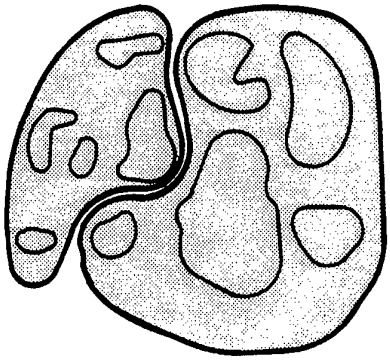


Fig.15a.

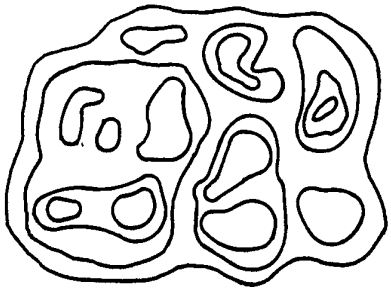


Fig.15b.

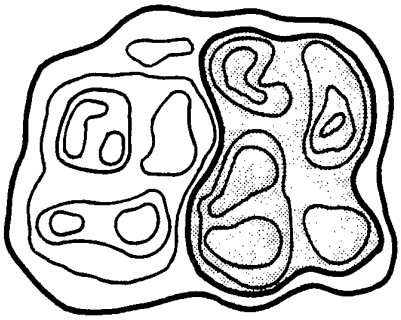


Fig.16a.

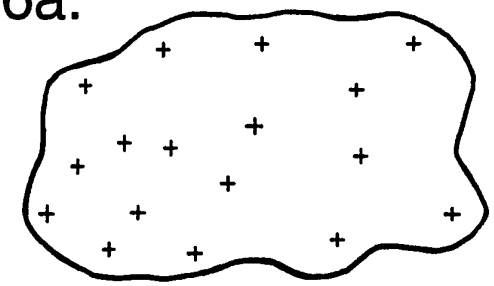
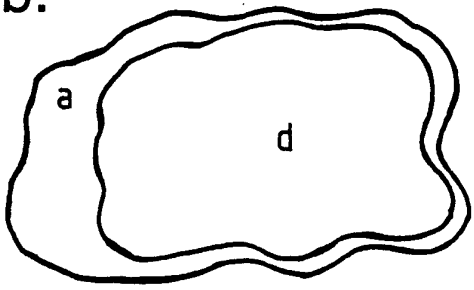




Fig.16b.



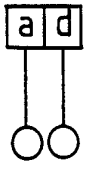


Fig.16c.

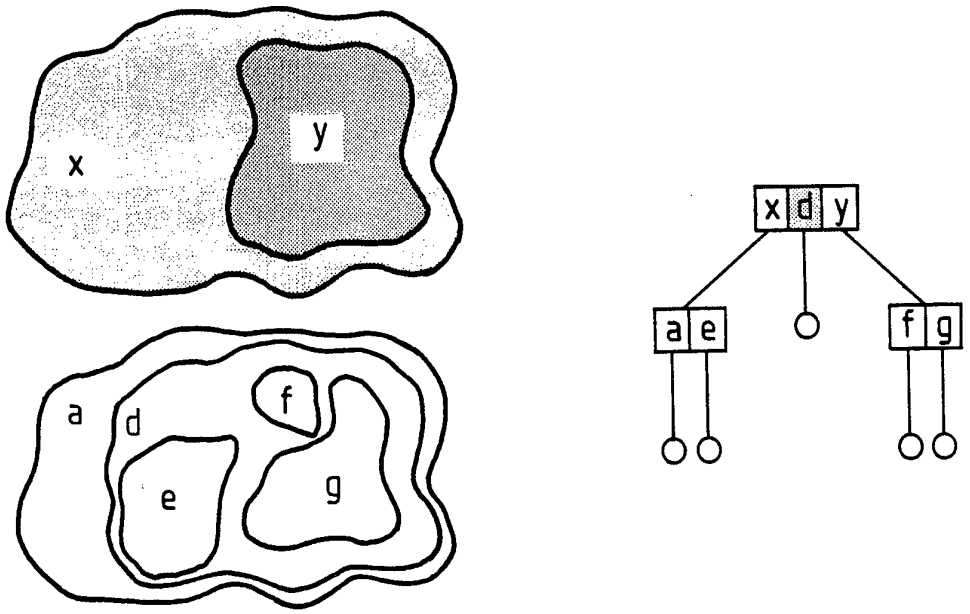


Fig.16d.

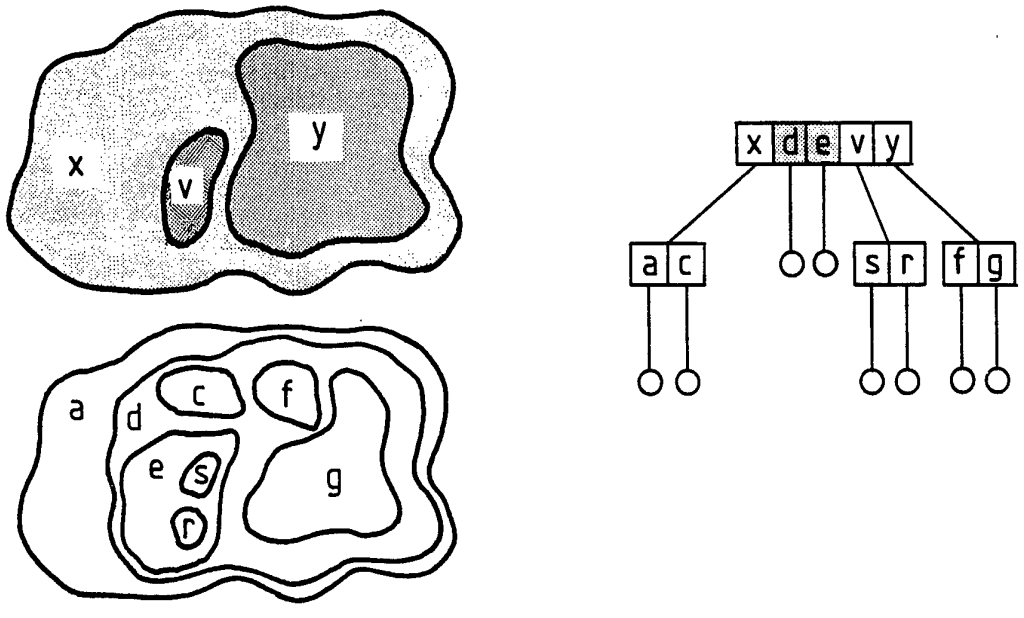
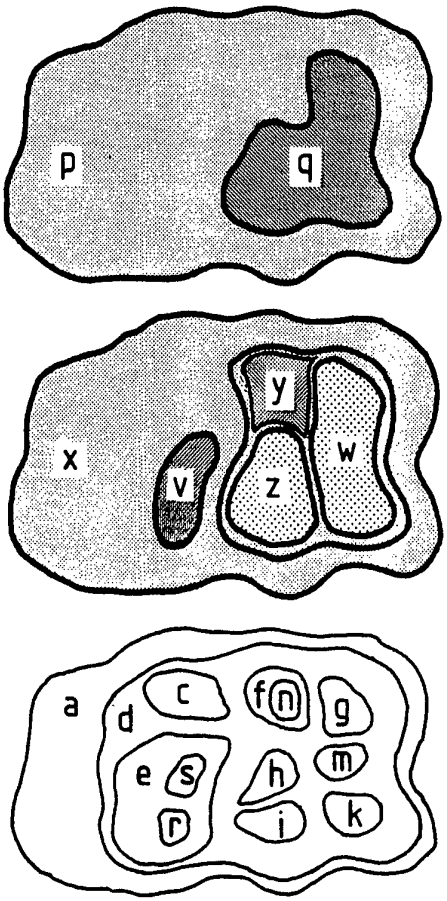


Fig.16e.



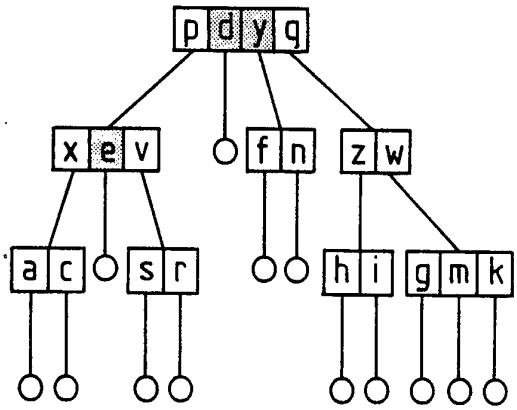
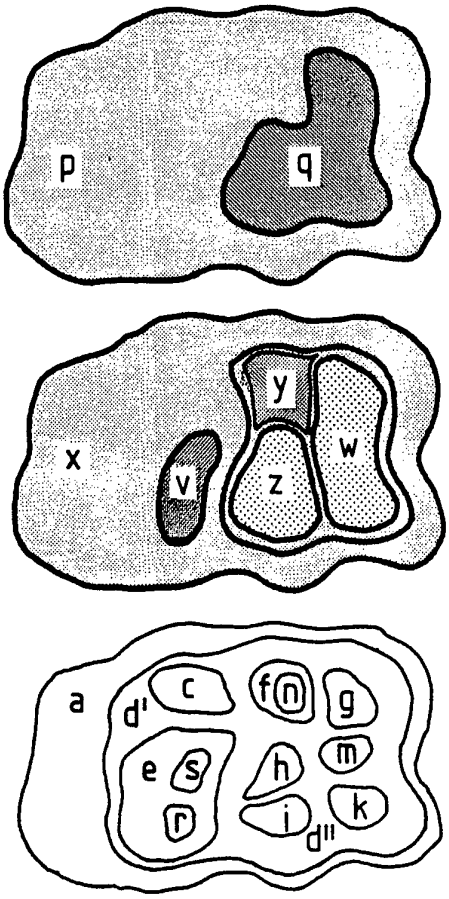


Fig.16f.



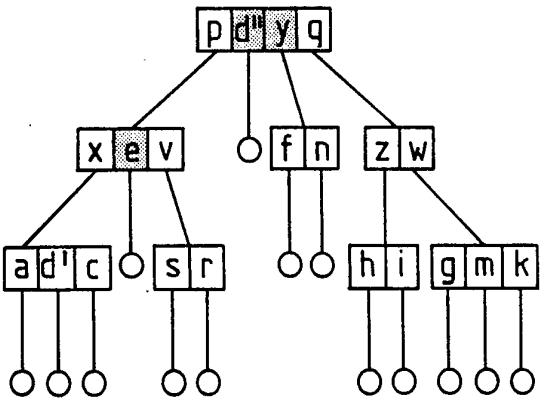


Fig.17.

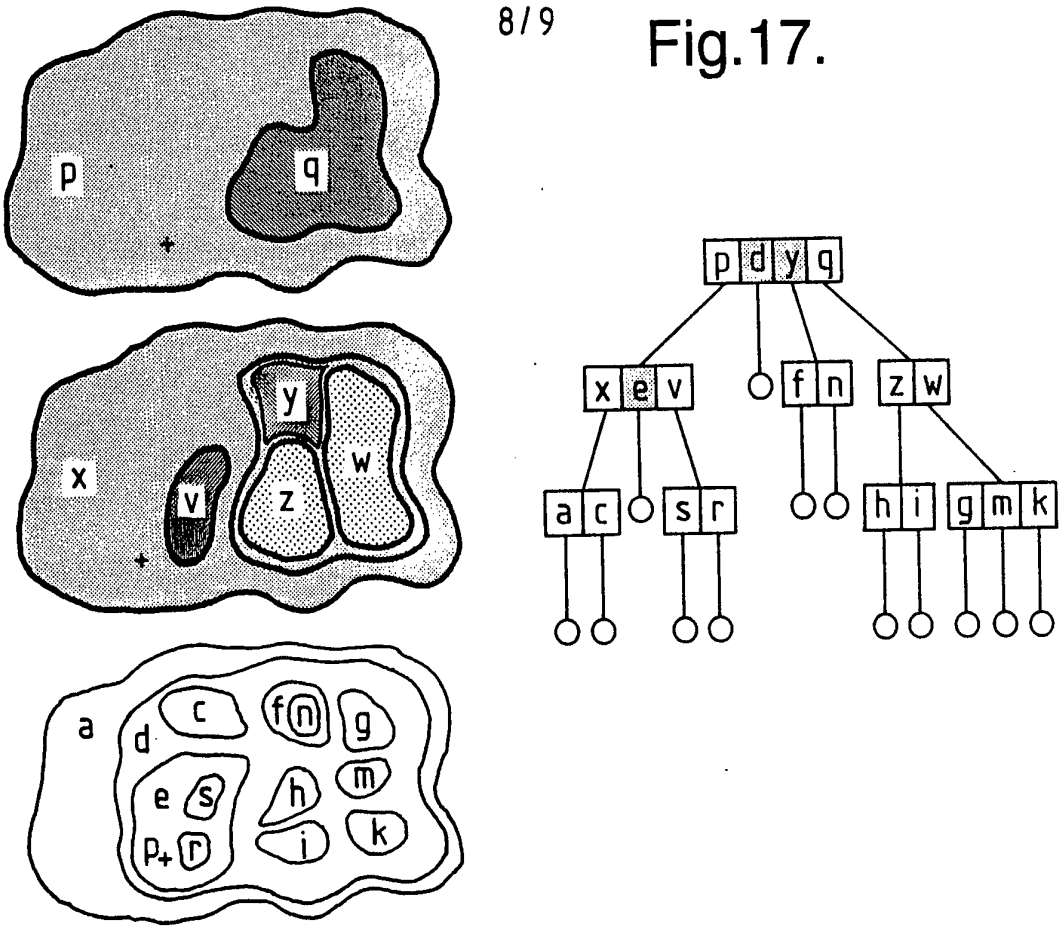
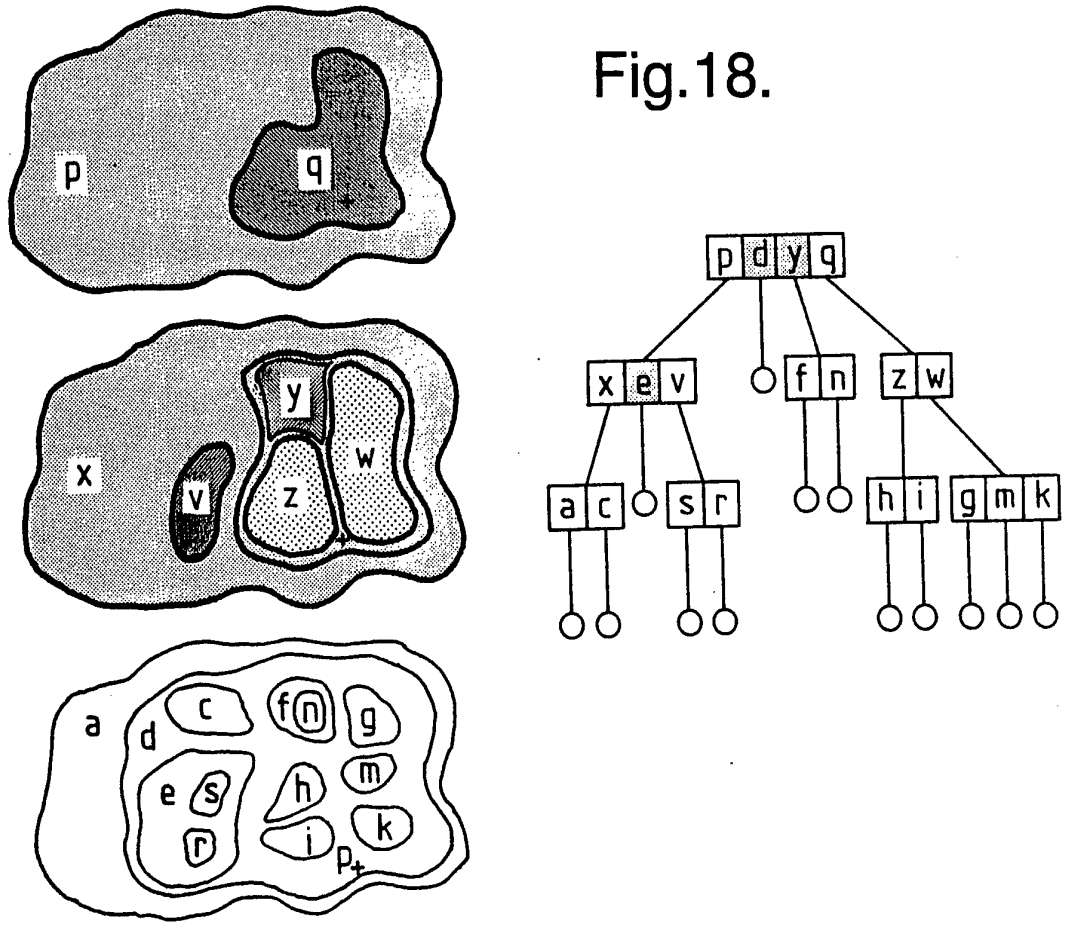

p
q
x
y
w
v
z
a
d
c
f n
g
e
s
h
m
p+
r
i
k
p d y q
x e v
f n
z w
a c
s r
h i
g m k

Fig.18.


p
q
x
y
w
v
z
a
d
c
f n
g
e
s
h
m
r
i
k
p+
p d y q
x e v
f n
z w
a c
s r
h i
g m k

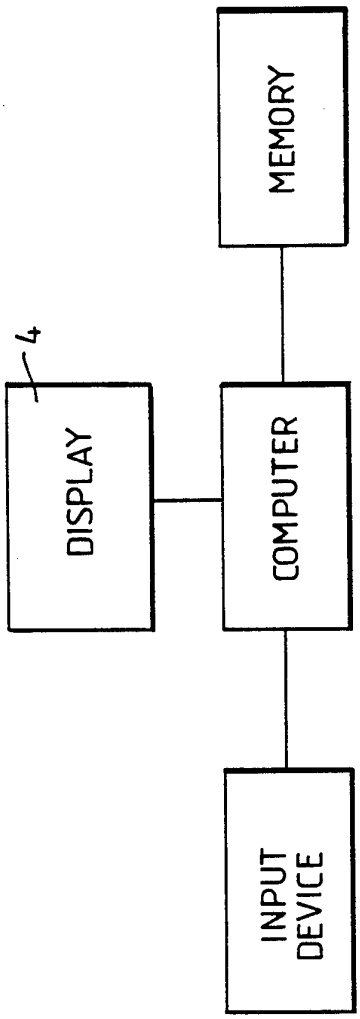


Fig.19.

INTERNATIONAL SEARCH REPORT

International application No.
PCT/EP 94/02166

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F17/30

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	3RD CONFERENCE ON FOUNDATIONS OF DATA ORGANIZATION AND ALGORITHMS, 21 June 1989, PARIS, FR pages 322 - 338 M. FREESTON : 'Advances in the design of the BANG file' cited in the application see the whole document ---	1-15
A	1ST SYMPOSIUM ON DESIGN AND IMPLEMENTATION OF LARGE SPATIAL DATABASES, 17 July 1990, BERLIN, GERMANY pages 287 - 300 M. FREESTON : 'A weel-behaved file structure for the storage of spatial objects' see page 292, line 10 - page 298, line 4; figures 3-6 --- -/--	1-15

[X] Further documents are listed in the continuation of box C.

[ ] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

1

Date of the actual completion of the international search	Date of mailing of the international search report
21 November 1994	01.12.94

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Fournier, C

Form PCT/ISA/210 (second sheet) (July 1992)

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	5TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 8 February 1989, LOS ANGELES, US pages 606 - 615 D. GREENE : 'An implementation and performance analysis of spatial data access methods' see the whole document	1,4,7, 9-15
A	6TH INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 5 February 1990, LOS ANGELES, US pages 296 - 303 Y. OHSAWA & M. SAKAUCHI : 'A new tree type data structure with homogeneous nodes suitable for a very large spatial database' see the whole document	1,4,7, 9-15
A	16TH INTERNATIONAL CONFERENCE ON VERY LARGE DATABASES, 13 August 1990, BRISBANE, AUSTRALIA pages 590 - 601 B. SEEGER ET AL : 'The buddy-tree: an efficient and robust access method for spatial data base systems' see the whole document	1,4,7, 9-15