ELSEVIER

Data & Knowledge Engineering 37 (2001) 1–23

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

A generalized comparison of linear representations of thematic layers ☆

Y. Manolopoulos [a], E. Nardelli [b,c], G. Proietti [b,c], E. Tousidou [a,*]

[a] Department of Informatics, Data Engineering Laboratory, Aristole University of Thessaloniki, 54006 Thessaloniki, Greece

[b] Department of Pure and Applied Mathematics, University of L'Aquila, 67100 L'Aquila, Italy

[c] IASI, National Research Council, 00185 Roma, Italy

Received 11 March 1999; received in revised form 21 June 2000; accepted 29 August 2000

Abstract

In this paper, the efficient encoding and manipulation of large sets of two-dimensional data that represent multiple non-overlapping features is investigated. Both linear structures are examined, i.e., leafcode and treecode representations of thematic layers. A new technique is presented and evaluated by applying it to already proposed access methods that are based on the previous representations. More specifically, we experiment with window queries that involve multiple features having as a performance measure the number of disk accesses. Experimentally it is shown that treecode representations outperform the previous structures with respect to time and space complexity. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Spatial databases; Region quadtrees; Leafcode and treecode representations; Multiple features; Window queries; Superimposition

1. Introduction

Today, we are often confronted with the task of handling large volumes of two-dimensional data representing multiple features. Examples could be found in applications such as image databases, geographical information systems (GISs), scientific visualization and computer-aided design. So far, a number of different approaches have been presented to manipulate specific classes of spatial data (i.e., points, lines, rectangles, volumes and hyper-volumes), the most popular of which are quadtrees, bintrees [15], R-trees, the cell tree and the grid file [8]. The interested reader can refer to [2,4,13] for detailed surveys on the topic.

In this work, we are interested in the manipulation of raster thematic maps comprised of multiple non-overlapping features, i.e., maps where each pixel contains one and only one feature. For example, such maps can be widely met in GISs where they represent distinct thematic layers.

☆ Research performed under the European Union's TMR Chorochronos project, contract number ERBFMRXCT96-0056 (DG12-BDCN).

* Corresponding author.

0169-023X/01/$ - see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S0169-023X(00)00045-8

Each layer, as its name states, has a distinct theme (or subject). This theme could be the land geology, the area elevation, or the soil type found in the depicted area. In the following, the words *features*, *colors* and *categories* will be used interchangeably.

Apparently, such data have the characteristics of images containing region data, and specifically the most prominent one, that is the aggregation of pixels of a given color into patches. This allows to assume that the number of features (i.e., colors) involved in the represented picture will be limited (e.g., in the range from 8 to 64), and most important, it calls for the application of hierarchical methods of image representation to save space and time.

Regarding hierarchical data decomposition, the most popular ones are the binary and quaternary decomposition generating the region bintree and the region quadtree, respectively. Each image is regularly and successively decomposed into two or four equal sized subimages for each case, until a *homogeneous* maximal block, with respect to the contained feature, is reached. Referring to the bintree, the image is split into two equal parts alternating a horizontal and a vertical subdivision.

Both kinds of trees can be implemented either in main memory or in secondary memory as pointer-based or pointerless structures, respectively. Since for very large images in database applications we are forced to rely on secondary memory implementations, the focus will be in this case only. Two types of representations appear in the literature:

- by extracting the collection of homogeneous leaves, which evidently carry semantic information (leafcode representation) [3];
- by traversing the tree in preorder and forming a string, which is called DF-expression (treecode representation) [6].

In this paper, both leafcode and treecode representations will be investigated. Some of the most important types of queries applied on spatial data are window queries, since they allow the extraction of only the desired information from the whole image. More specifically, the window query types on which we are going to experiment with are the following:

- exist$(w, f_i, f_j, \dots, f_k)$: check whether or not at least one of the features $f_i, f_j, \dots, f_k$ exists inside the window w;
- report(w): report all features that are found inside the window w;
- select$(w, f_i, f_j, \dots, f_k)$: select all homogeneous blocks inside the window w containing features $f_i, f_j, \dots, f_k$.

The efficient processing of window queries has already been studied by Nardelli and Proietti [9], who proposed adjusting region linear quadtrees to manipulate the feature information. The hybrid linear quadtree (HL-tree) was introduced as an enhancement to the previous method [10], whereas the MOF-tree, which was based on the HL-tree, was presented as a structure to efficiently manipulate images with multiple overlapping features [7].

Regarding the treecode approach, it is asymptotically more compact than the leafcode one but it has suffered for a long time the lacking of a paged version able to support the access to a given element without being forced to scan the entire database, in the worst case. This difficulty has been overcome by de Jonge et al. [5], who developed the S^+-tree, a spatial access method combining the advantages of treecode and leafcode representations, essentially by indexing through locational codes the space-compact DF-expression.

The S^+-tree was tailored to binary images, hence causing large space waste and consequently a performance degradation when trying to manipulate colored images. To overcome this drawback,

Nardelli and Proietti [11] enhanced the S^{+}-tree and proposed the S^{*}-tree. By using the S^{*}-tree for colored images they succeeded in saving up to 25% of space and in performing asymptotically the same with the S^{+}-tree as far as disk accesses for solving window queries were concerned.

In this paper, we describe a technique which will be applied on both kinds of representations, aiming to the efficient handling of thematic layers with multiple non-overlapping features. The focus will lie on the efficient processing of queries, which involve both the features and the spatial object locations as well. As already mentioned, in the sequel, only the pointerless representation of these structures will be examined, although the same technique could be applied to the pointer-based representation in a straightforward manner. By applying the proposed technique to some quadtree based access methods, the advantages derived by using this technique are shown from the standpoint of saving space and reducing the execution time in window queries as the number of disk accesses will reveal.

The rest of this paper is organized as follows. In Section 2, most quadtree-based access methods that have already been proposed for maps with thematic layers are reviewed. In Section 3, the new structure will be motivated and described in detail, whereas some new algorithms for performing window queries more efficiently will be presented. In Section 4, some representative results which were derived from the conducted experiments will be shown. Section 5 contains concluding remarks and directions of possible future work.

2. Related work

As already mentioned, in this work the focus lies in querying raster images containing multiple non-overlapping features. In our representation of images, each distinct feature is represented by a different color. This means that each pixel of the map will carry one and only one color, in contrast to the images with overlapping features, where a pixel of the map may be assigned more than one color representing different map categories. It must be mentioned that, since multiple features have to be handled, the hierarchical image decomposition will stop only when a maximal block of space that contains a single feature, i.e., a homogeneous block is, reached. In the rest of the paper, m will denote the image resolution, whereas n will denote the number of contained features.

2.1. Independent linear quadtrees

A first straightforward method for representing colored images, named *independent linear quadtrees* (IL-trees) and mainly adopted for comparison purposes, is that of using a separate linear quadtree [3] for each feature, resulting in as many linear quadtrees as the number of features in the image. This approach could present a substantial space overhead since multiple indices have to be stored. This fact is also a weak point during concurrent manipulation of multiple features due to the need to traverse and join the results from multiple indices.

2.2. Simple linear quadtree

A first step towards the better exploitation of thematic layers was the use of *simple linear quadtrees* (SL-trees) [9]. In fact, the latter structure is the original linear quadtree, enriched with feature information. During the procedure of successive decomposition, once a homogeneous block is reached, the information about the particular feature that was found in this block is

retained together with the corresponding leaf quadcode. More specifically, now each quadtree leaf will be characterized by two fields:

- the *locational key*, whose digits reflect successive quadrant subdivision;
- the *feature identifier*, which contains the id of the feature that exists in the specific node.

Then, the entries for all quadtree leaves will be inserted in a B$^+$-tree where the locational key will serve in traversing the latter structure.

In Fig. 1, an 8×8 image is depicted which contains four non-overlapping features. The feature ids are listed in the table in the right-hand side of Fig. 1. In Fig. 2, the homogeneous leaves of the corresponding quadtree are shown, whereas internal nodes are represented with gray color. Next, the list of generated locational codes, in the form of FD codes [14], is depicted. For example, the locational code 3-021 represents the leaf node at depth 3 which has a feature identifer equal to 2, since the feature contained in the corresponding subimage is the one having id $=$ 2. Similarly, the locational code 2-010 represents the leaf node at depth 2 and has feature identifier equal to 1, since this is the feature that the corresponding quadrant contains

$$(3\text{-}000, 1),\ (3\text{-}001, 3),\ (3\text{-}002, 0),\ (3\text{-}003, 1),\ (2\text{-}010, 1),\ (3\text{-}020, 0),\ (3\text{-}021, 2),\ (3\text{-}022, 2),$$
$$(3\text{-}023, 0),\ (2\text{-}030, 1),\ (2\text{-}100, 0),\ (2\text{-}110, 1),\ (2\text{-}120, 1),\ (2\text{-}130, 0),\ (2\text{-}200, 2),\ (2\text{-}210, 0),$$
$$(2\text{-}220, 2),\ (2\text{-}230, 3),\ (1\text{-}300, 0).$$

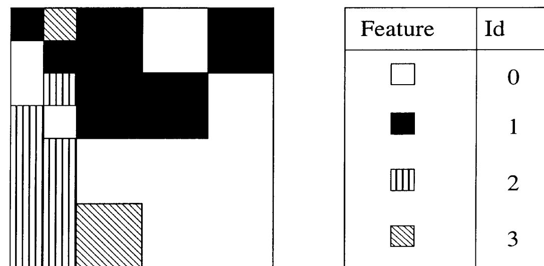

Fig. 1. An 8×8 image and the feature-Id table.

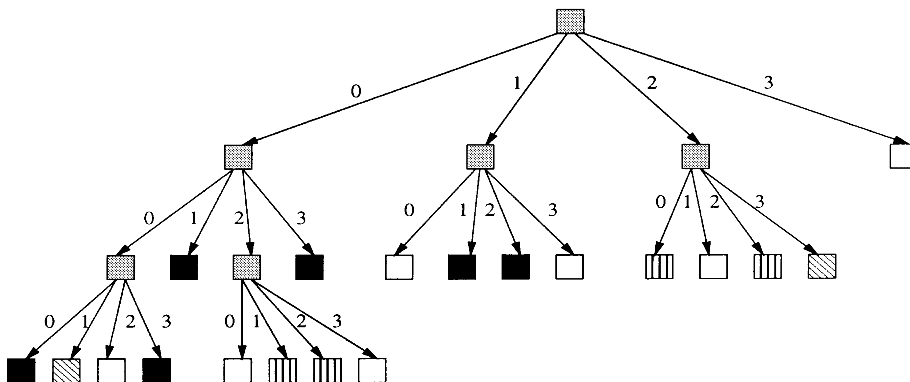

Fig. 2. The quadtree representing the image of Fig. 1.

2.3. Hybrid linear quadtree

The *hybrid linear quadtree* (HL-tree) has been proposed in [10] to improve the performance of SL-trees. The main idea behind HL-trees is that, apart from the quadtree leaves, internal nodes are also registered in the B^+-tree. The drive was the fact that searching could terminate when reaching an internal node, since it would be able to provide information about the features of its children. Therefore, in case the queried features did not exist in the corresponding subimage, no additional disk accesses would be required and the retrieval cost would be decreased.

However, since an internal node is associated with an heterogeneous block, and therefore it contains more than one feature, a different structure of the representing record is required. Basically, the authors suggest replacing the feature identifier with a so-called *features bitstring*, that is a bitstring of size equal to the number of features existing in the image. A specific bit of the features bitstring is set to 1, if and only if the respective feature exists in the represented quadrant. Concerning records associated with leaf nodes, they continue to have the same structure as for the linear quadtree. To distinguish between the two kinds of nodes, an additional *leaf bit* is associated with each record, whose value is 1 if and only if the corresponding node is a leaf. Summarizing, each record in the HL-quadtree has the following structure, depending on whether it is associated with a non-leaf or with a leaf node:

- (*l-key* $\{3m$ bits$\}$, *features bitstring* $\{n$ bits$\}$, *leaf bit* $\{1$ bit$\}$),
- (*l-key* $\{3m$ bits$\}$, *feature identifier* $\{\lceil \log n \rceil$ bits$\}$, *leaf bit* $\{1$ bit$\}$),

respectively. Evidently, this approach has a space overhead due to the storage of the internal nodes and the corresponding feature bitstring. However, in [10] it is explained that this overhead is not significant since the number of internal nodes is not larger than the 1/3 of the number of leaves and, consequently, the asymptotic space overhead will remain the same.

Applying the method of HL-trees on the image of Fig. 1, the following list of nodes will be created and stored in the B^+-trees leaves. To improve readability, leaf nodes are in italic. As can be seen, internal nodes contain a bitstring of size 4, since 4 are the features existing in the represented image. For example, the locational code 2-000 that corresponds to an internal node is related to bitstring 1101, since all features exist in the represented quadrant apart from the feature with id $= 2$

(0-000, 1111, 0), (1-000, 1111, 0), (2-000, 1101, 0), (*3-000*, *1*, *1*), (*3-001*, *3*, *1*), (*3-002*, *0*, *1*), (*3-003*, *1*, *1*), (*2-010*, *1*, *1*), (2-020, 1010, 0), (*3-020*, *0*, *1*), (*3-021*, *2*, *1*), (*3-022*, *2*, *1*), (*3-023*, *0*, *1*), (*2-030*, *1*, *1*), (1-100, 1100, 0), (*2-100*, *0*, *1*), (*2-110*, *1*, *1*), (*2-120*, *1*, *1*), (*2-130*, *0*, *1*), (1-200, 1011, 0), (*2-200*, *2*, *1*), (*2-210*, *0*, *1*), (*2-220*, *2*, *1*), (*2-230*, *3*, *1*), (*1-300*, *0*, *1*).

2.4. S^+-tree

A first step towards the integration of leafcode and treecode representations has been done by de Jonge et al. [5] who defined a secondary memory implementation of binary images named

S^+*-tree*. This tree was originally described by using the leafcodes generated by a bintree, though a quadtree could be used equally well.

Actually, the S^+-tree comprises the paged version of a main memory structure called, S-tree. During the creation of an S-tree, a preorder traversal of the binary tree is performed, producing a '0' ('1') whenever an internal (leaf, respectively) node is encountered. The outcome is a bitstring that corresponds to the *linear bintree*. At the same time, during this traversal the colors of the leaves are stored in an additional bitstring, called *color table*, where a '0' ('1') represents a white (respectively, black) leaf. The two bitstrings thus obtained comprises the S-tree. In [5] it is described how this tree can be created by using the list of locational codes of the black leaves.

Searching for an area's features begins by traversing the linear bintree where, whenever a leaf is reached, its color is retrieved by visiting the respective position of the leaf in the color table. An example of a binary image with its respective binary tree can be seen in Fig. 3. The corresponding S-tree would be

linear bintree	0000110110110000111100111
color table	0101100100010

However, the need to manipulate large pictures led to the creation of a paged version of this structure which could be stored in secondary devices, and therefore the S^+-tree was introduced. The S^+-tree is built by storing the original tree into a list of data pages where each page contains a segmented and augmented S-tree representation of the image. These data pages will be indexed by a prefix B-tree [1] with the use of separators in place of keys. This way, each data page will constitute a self-contained local S-tree that can be searched independently.

In particular, a data page consists of the linear bintree (growing from the beginning of the page) along with the corresponding portion of the color table (growing from the tail of the page). The two bitstrings fill the page as much as possible under the constraint that the last node stored in a page must always be a leaf in order to avoid duplicating information. Therefore, due to this constraint, some unused space might be left. Moreover, at the very beginning of the page there is a *linear prefix* which can be regarded as the summary of all the data pages preceding the actual one. This linear prefix is constructed in the following way.

When a data page becomes full during the building process, a new page is created and a *separator* between the pages is stored in the index. The separator is built by encoding the path

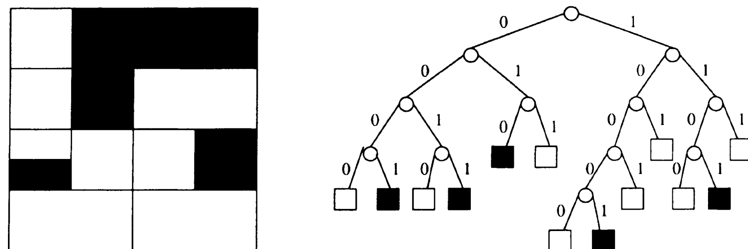

Fig. 3. A binary image (left) and its representing binary tree (right).

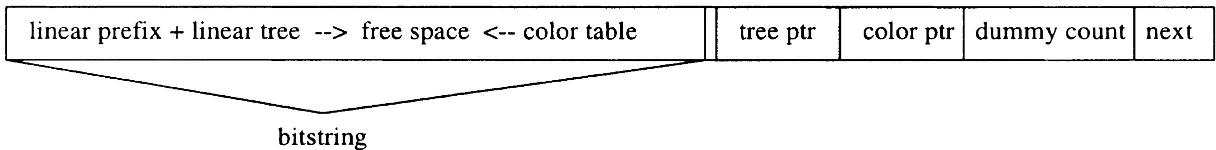

Fig. 4. The layout of a data page of the S^+-tree.

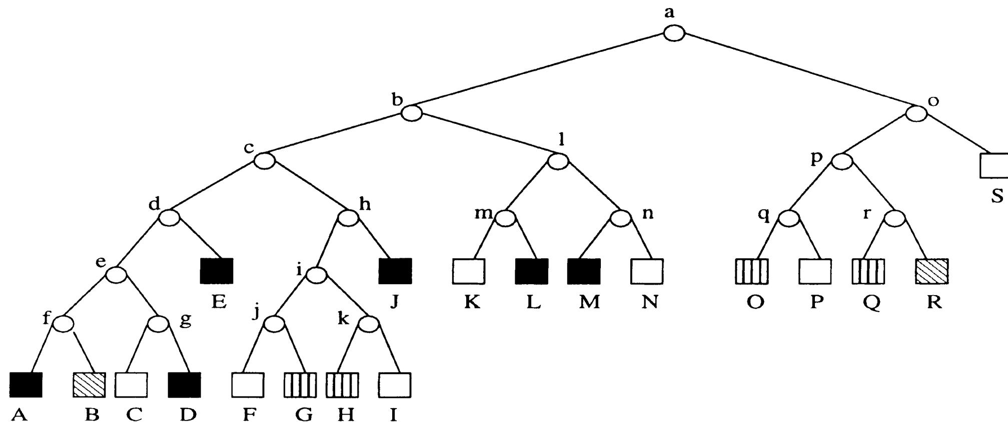

Fig. 5. The binary tree which will be produced by the image of Fig. 1.

from the bintree root to the first node stored in the next page, producing a '0' when moving to the left, and a '1' otherwise. The linear prefix is built by encoding with a '0' a 0 in the separator, and with a '01' a 1 in the separator. This way, the linear prefix provides the information needed to retrieve a node in a page, since it resembles the whole bintree preceding the nodes in such a page by condensing all the left subtrees in leaves. The structure of an S^+-tree node can be seen in Fig. 4. The *tree pointer* points to the next available position in the linear tree string, the *color pointer* points to the next available position in the color table, *next* is a pointer to the next page in the sequence set, whereas *dummy count* indicates where the linear prefix ends and the linear tree starts.

Since in this paper, the focus is in thematic layers (which contain more than two features) and not in binary images, we will describe shortly how the S^+-tree corresponding to the thematic layer of Fig. 1 will be created. Firstly, in Fig. 5, the bintree representing the previous image is shown. The S-tree that will be produced from this image is

linear bintree	0000001101110001101110011011000110111
color table	01 11 00 01 01 00 10 10 00 01 00 01 01 00 10 00 10 11 00

As it can be seen, we encoded each feature with a binary number in order to store it in the color table. Therefore, feature 0 has been encoded to 00, feature 1 to 01, feature 2 to 10 and feature 3 to 11. Trying to create the paged version, the following pages are produced where each page has a capacity of 20 bits.

	Local linear bintree	Empty space	Color table	Separator
page A	00000011011	x	01 00 11 01	0001
page B	00001100011	xxx	10 00 01	00101
page C	00010010111	xxx	01 00 10	01
page D	0010011011	xx	00 01 01 00	1
page E	0100011011	xx	11 10 00 10	11
page F	01011	xxxxxxxxxxxxx	00	

In the second column, the underlined bits represent the part of the real tree that is stored in a page, while the remaining bits represent the linear prefix, that is the condensed part of the tree which has been already stored in previous pages. The column 'Separator' is not a part of the page and contains the separators that are created after each page is filled and which will be inserted in the prefix B-tree. The specific nodes of the tree that are stored in each page can be seen in Figs. 6–8.

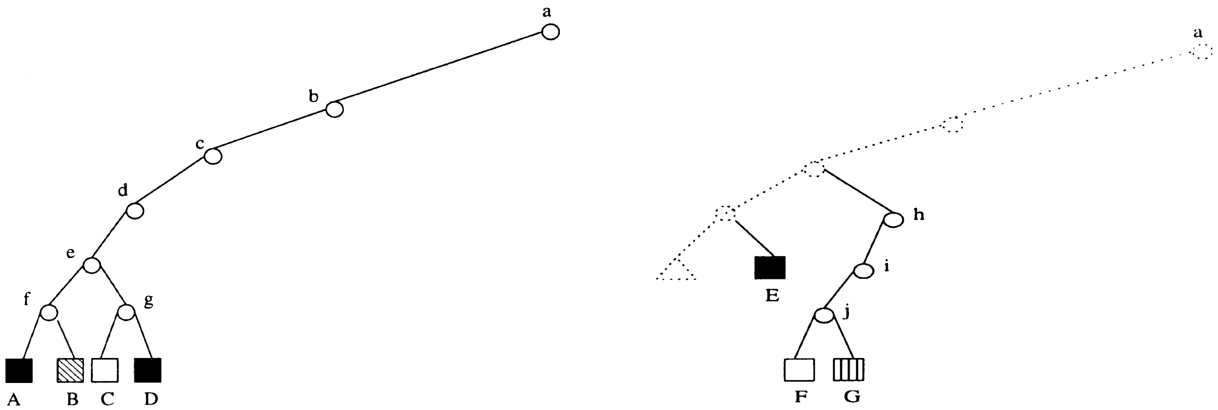

Fig. 6. Page A (left), page B (right).

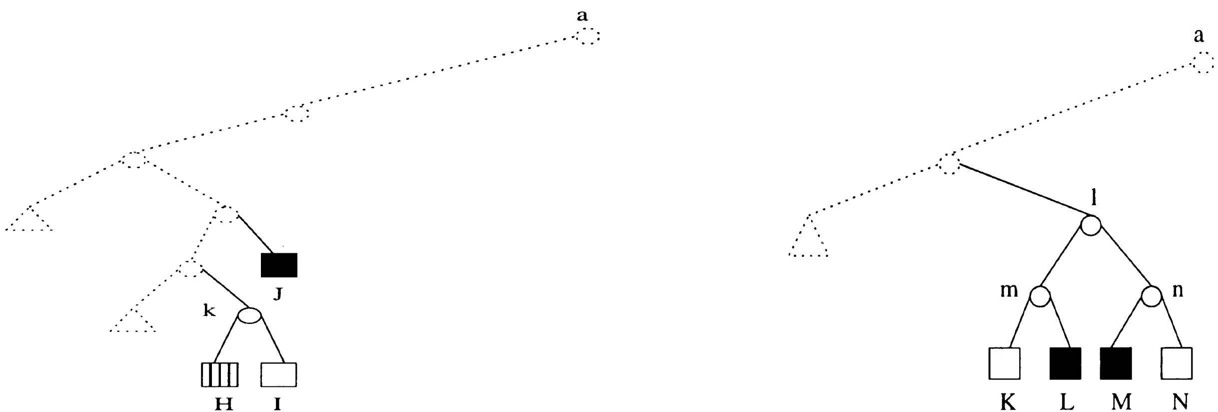

Fig. 7. Page C (left), page D (right).

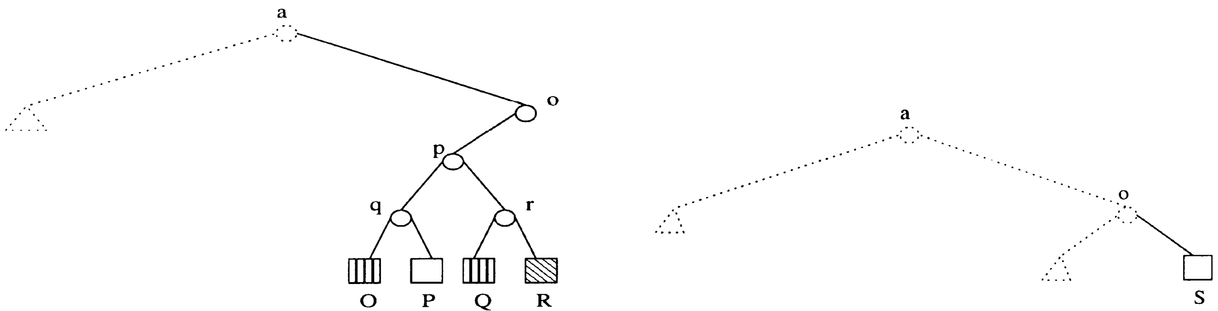

Fig. 8. Page E (left), page F (right).

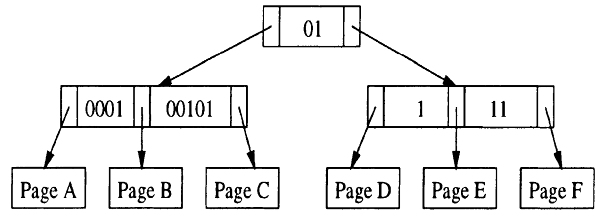

Fig. 9. The prefix B-tree that contains the separators of the bintree shown in Fig. 4.

Dotted lines represent portions of the tree stored in previous pages, as they are represented by the linear prefix.

Trying to see how a separator is produced, an example should be followed. In page A, all nodes until leaf *D* are stored. Therefore, the next node to be inserted in page B is the leaf node *E* which has the code 0001 and, consequently, this is the first separator to be inserted in the B-tree. As it can be seen in the right-hand side of Fig. 6, the branch leading to the small triangle in page B represents the portion of the tree that has been stored in page A. Similarly, after page C is filled, the last node which is stored in it is node *J*. The node which will be the first to store in page D is the internal node *l* which has code 01 and which is also the third separator. The rest of the separators are shown in the last column of the previous table. The prefix B-tree that will be created is shown in Fig. 9.

Notice that building the S^+-tree by using the leafcodes of a quadtree instead of a bintree, as is the case for the rest of this paper, leads to a somewhat different creation of the separator and the linear prefix. The path from the quadtree root to the node that caused the filling of the page is encoded by producing a '0' when moving towards the first child (NW), and a '1', '2', or '3' when moving towards the second (NE), third (SW) or fourth (SE) child, respectively. Consequently, the linear prefix would be built by encoding with a '0' a 0 in the separator (preceding internal node or root of a subtree), with a '01' a 1 (preceding root of the subtree and the NW child), with a '011' a 2 (preceding root, NW and NE child) and with a '0111' a 3 in the separator (preceding root, NW, NE and SW) child.

This structure and the characteristics of S^+-trees, in particular the property that each data page represents a self-contained local S-tree that can be searched independently, is its great advantage

in window queries. As already mentioned, it provides a very compact data representation, while the index behaves like B^+-trees and permits easy sequential and random access at the same time. Using a picture's leafcodes or its binary pixel array as input, the corresponding S^+-tree can be constructed in such a way that the sequence set pages are generated from left to right, which allows for almost 100% storage utilization (in these data pages). Subsequent insertions and deletions will degrade the storage utilization, but only down to 69% which is the typical storage utilization of B^+-trees [5].

2.5. S^*-tree

The *S^*-tree* by Nardelli and Proietti [11] is a refinement of the S^+-tree and was introduced in order to manipulate efficiently colored images. When describing the contents of an S^+-tree node it was mentioned that, by requiring the last node stored in a page to be a leaf, some unused space between the bitstring and the color table might be left. Actually, as described in [11], this unused space is negligible for binary images since no information is associated with internal nodes (they are simply gray). Therefore, if m is the image resolution, there can be at most $2m - 1$ unused bits per page, which means at most 1 byte for typical m values. However, this is not the case for colored images. In fact, to fully exploit the S^+-tree capabilities, each external node should be associated with a feature identifier of $\lceil \log n \rceil$ bits, (as shown in the example of Section 2.4) and a features bitstring of n bits should be associated with each internal node, similarly to the strategy adopted for HL-trees. In fact, associating a color string with internal nodes greatly improves the performance in executing several spatial operations. However, as soon as n increases, the constraint that the last node stored in a page must be a leaf would result in a large space waste, since a large number of internal nodes, which could potentially be stored in a page, are shifted to the next one. Therefore, such a constraint has to be relaxed for colored images. In [11], it is also proved that this can be done efficiently without modifying the separators, i.e., without augmenting the space occupied by the index. However, rarely a small time penalty has to be paid when a search to a given node is performed.

The actual layout of an S^*-tree page is given in Fig. 10. Note that the free space will be at most n bits (i.e., the length of a features bitstring), instead of $(n + 1) \times (m - 2) + \lceil \log n \rceil$ bits of the corresponding S^+-tree (this is the case when the next leaf that should be stored lies at the end of a path of $m - 2$ internal nodes that have not yet been visited). Instead of using the prefix B-tree, the separators were inserted in a B^+-tree with similar results. Additionally, to make the index manipulation easier and to save space, the linear prefix was eliminated from the pages since it can be easily recomputed from the separators in the B^+-tree. This way, the space overhead is reduced and the update operations of the B^+-tree are performed in a more straightforward way. Finally, the

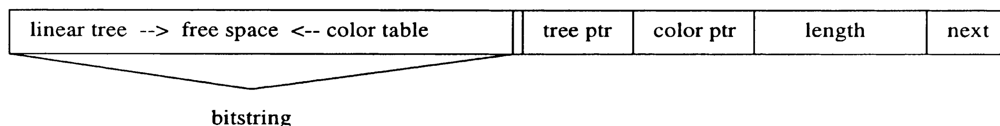

Fig. 10. The layout of a data page of the S^*-tree.

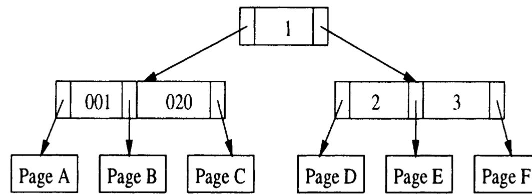

Fig. 11. The B^+-tree where the separators created by the S^*-tree are stored.

space occupied by the field dummy count in the S^+-tree has been replaced by the field *length*, which stores the length of the separator associated with the page.

In the following table, we show how each page of the S^*-tree will be formed for the quadtree shown in Fig. 2, where pages have a capacity of 18 bits. The B^+-tree where the separators are inserted, can be seen in Fig. 11.

	Local linear bintree	Empty space	Color table	Separator
page A	0001		01 1101 1111 1111	001
page B	11110	x	1010 01 01 00 11	020
page C	11111	xxx	01 00 10 10 00	1
page D	01111	x	00 01 01 00 1100	2
page E	01111	x	11 10 00 10 1011	3
page F	1	xxxxxxxxxxxxxxx	00	

3. Proposed technique

Due to the increasing interest of the database community towards the development of efficient management systems for geographical data, where each map constitutes a specific thematic layer with its own non-overlapping categories, the need for fast retrieval of all or some of the categories that exist in a given region is emerged. In simple words, searching for a category is reduced to searching for the specific color by which this category is represented in the map. In this section, the motivation is stated and a new technique which addresses this task efficiently is presented.

3.1. Query manipulation and motivation

The original linear quadtree as well as the original S^+-tree were proposed for the efficient manipulation of binary images where all the information was carried out by the black pixels. This information was basically handled by a B^+-tree, adopting all the benefits of such an index (random access, fast retrieval and good space utilization), and it comprised the target of every query.

As far as thematic layers were concerned, the previous methods lacked any support for such images, where the information is carried out by all pixels that represent a specific area. Refinements introduced by the HL-tree, whose target is the more adequate manipulation of thematic layers, were confined to the maintenance of feature information within the records associated with the nodes of the corresponding quadtree, with no exploitation of such information to the higher

levels of the B^+-tree index. The same holds for the refinement of the S^+-tree, the S^*-tree. Consequently, it is not possible to take advantage of the feature information in higher B^+-tree levels and avoid traversing unqualified branches for queries based on features. In the IL-trees, there is no need for filtering but instead several indexes have to be traversed to answer queries involving multiple features.

Our study is focused on the better manipulation of the color information without degrading the performance of location-based queries. Internal B^+-tree nodes will be used to avoid searching in image areas where it is certain that the sought feature is not to be found.

3.2. Description of the new technique

All the previously described methods depend on the ability of the B^+-tree to retrieve the desired information that lies in its branches. The proposed technique enhances B^+-trees by enriching them with the color information not only in their leaves but also in their internal nodes. Enriching the upper B^+-tree levels with color information avoids traversing branches where the queried categories do not exist.

Evidently, the original B^+-tree traversal is based on the locational code which is the B^+-tree key. To be able to efficiently perform window queries that search for certain features inside a given window, it is important to know whether the B^+-tree sub-structure that is about to be traversed contains at least one of the desired features. In case it does not, this sub-structure can be skipped resulting in fewer disk accesses. Since this traversal has to be additionally constrained with the feature information, a bitstring representing the kind of desired information will be stored in the upper B^+-tree levels. In the following, it will be made obvious that the proposed technique of posting to the parent node a second-order bitstring can be applied to the SL-tree, the HL-tree and the S^*-tree as well. The structures derived by such a use of the bitstring are named BSL-tree, BHL-tree and BS^*-tree, respectively.

According to the new technique, the entries at the B^+-tree leaves are left unchanged for each one of the access methods in consideration, while a second-order bitstring is introduced in the entries of internal nodes. More specifically, all the features appearing in the entries of a specific B^+-tree leaf are superimposed (OR-ed), thus forming a new second-order bitstring which represents the feature information of the respective leaf in a condensed/abstract way. In essence, for each leaf a new bitstring is encoded so that the ith bit is set to 1, if and only if the feature with id $= i$ exists in the entries of the specific leaf. Then, this second-order bitstring is posted to the parent node of the leaf. The idea of producing second-order bitstrings can be generalized for all B^+-tree levels. Thus finally, each entry of internal nodes will be accompanied by this second-order bitstring which will have 1s only at positions where the respective features exist in the corresponding B^+-tree sub-structure. Henceforth, internal B^+-tree nodes consist of at most r triplets (tree pointer, locational key, features bitstring), where r is the tree fanout. [1]

Notice that this way, only the internal B^+-tree nodes will be affected, and since their average space requirements for real images would be about 30 or 40 MB, it can be safely assumed that they can be easily accommodated in the main memory of modern computers. This allows for

[1] To be more precise, internal nodes store r bitstrings and tree pointers towards the r children, and $r - 1$ locational keys which are sufficient to traverse the tree.

ignoring the space occupied by the features bitstrings when calculating the introduced storage overhead for the proposed technique.

3.3. Algorithm description

3.3.1. The creation algorithm

The first step in building the proposed structure (either BSL-, BHL- or BS*-tree) is the image decomposition, which will result in a list of maximal blocks.

In BSL-trees, this list will contain only homogeneous blocks, where each entry will consist of the locational code of the represented quadrant followed by the feature identifier standing for the feature that is found in this quadrant. All entries of this list will be inserted in a B^+-tree and will be stored at its leaves. A bottom-up procedure is then followed to post the feature information from the leaves to the upper B^+-tree levels; i.e., a bitstring from each B^+-tree node is extracted. At the leaf level this bitstring results from the features stored in the feature identifiers stored in each leaf node while for the internal B^+- tree levels the superimposition method (OR-ing) is used to propagate the feature information to higher tree levels. Thus, by superimposing the bitstrings that exists in a node, a new second-order bitstring is produced and stored in the entry that is the ancestor of this node at the next higher level. This procedure propagates upwards until the root is reached. Entries at the tree root will, most probably, have many bits set to 1 since they will correspond to sub-images with many features. An example of a BSL-tree can be seen in Fig. 12. Entries at the leaf level are in the form (locational code, feature indentifier), while internal nodes have the form (pointer, features bitstring, locational code, ..., pointer, features bitstring).

As far as BHL-trees are concerned, the list of blocks will consist of the locational codes of both external and internal quadtree nodes, that is, homogeneous and non-homogeneous blocks, followed by the feature indentifier and features bitstring, respectively, indicating the features that are found in the represented quadrant. Similarly to the previous case, all these entries will be inserted in a B^+-tree and stored at its leaves while the bottom-up procedure of building the feature bitstrings will follow. In Fig. 13 an example of such a tree can be seen where the only difference from the previous case is that entries at the leaf level can be either (locational code, feature identifier, leaf bit) or (locational code, feature bitstring, leaf bit) depending on the represented quadrant. The field leaf bit denotes whether the respective entry is an internal ('0') or external ('1') node.

However, in the case of BS*-trees, the list of maximal blocks of the homogeneous quadrants will only help build the S*-tree pages whereas the separators will be stored in the B^+-tree internal nodes. The leaves of the B^+-tree will actually contain the S*-tree pages. The superimposed bitmap that will be produced by the entries at the leaf level will result from the color table and will contain

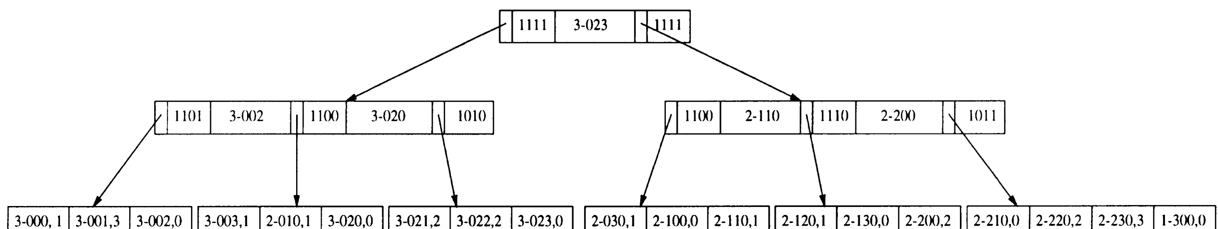

Fig. 12. The BSL tree which represents the image of Fig. 1.

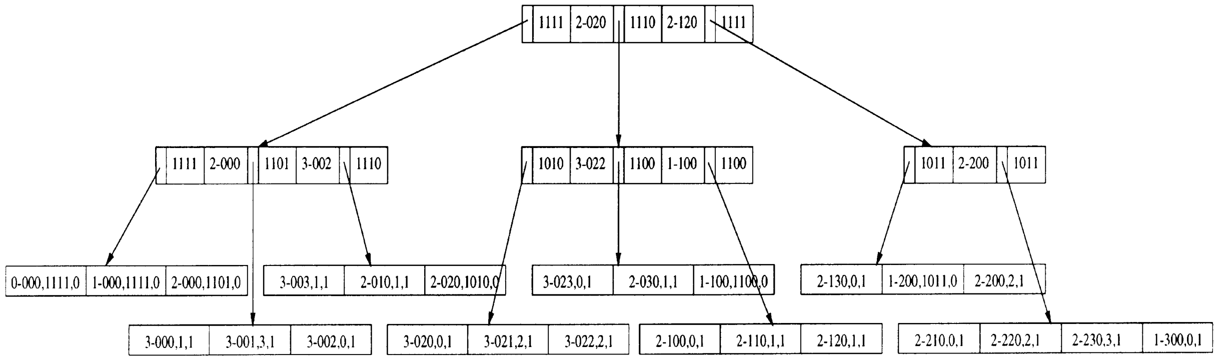

Fig. 13. The BHL tree which represents the image of Fig. 1.

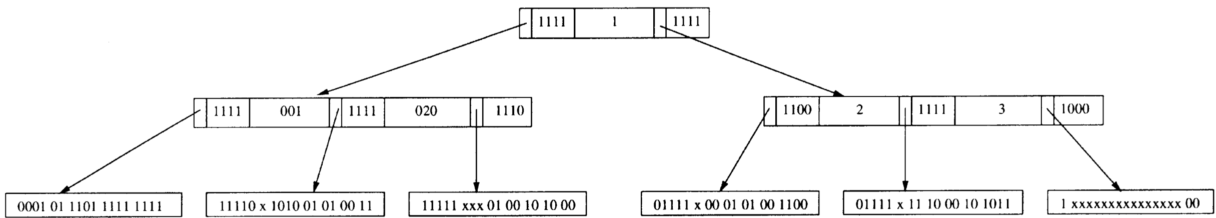

Fig. 14. The BS* tree which represents the image of Fig. 1.

all the features that exist in the respective part of the tree. In Fig. 14 an example of a BS*-tree can be seen, where each page at the leaf level corresponds to a page of the S*-tree, while entries at internal nodes are similar to the previous methods. For simplicity purposes, only the string of the linear bintree and the color table is depicted at the leaf nodes.

3.3.2. Window queries

In case of window queries, the basic approach of decomposing the window query into a sequence of smaller queries is followed, where each smaller query comprise a maximal block of the image inside the window [12]. In the following, it is explained how these queries proceed according to the proposed technique.

The exist query. Consider a query over a specified window, where a search for the existence of features $f_i, f_j, \ldots, f_k$ has to be performed. For each maximal block, searching starts from the B^+-tree root. Before descending the tree levels, each entry's bitstring is examined. If at least one bit corresponding to one of the queried features is set to 1, then the respective subtree is followed; otherwise we skip to the next node entry. The same procedure is followed at the remaining tree levels; searching stops only when the leaf level is reached.

In the case of BSL-trees, it should be emphasized that two possibilities may arise when the leaf level is reached:

- the search is successful and the desired locational key (maximal block) has been located;
- the search is unsuccessful, but searching continues for a homogeneous ancestor or the descendant of the desired locational key, since they correspond to larger or smaller maximal blocks containing or contained in the desired maximal block.

In the latter case, looking for the ancestor, one more disk access on the previous page may be needed, while retrieving all the descendants might be very expensive (a maximal block might contain a number of descendants which is quadratic on its side).

In case of BHL-trees, the algorithm is similar to that for BSL-trees. The difference is that now it is certain that either the searched maximal block or a homogeneous ancestor of it will be located, since all quadrants are stored. This way, the query is satisfied without looking to the descendants of the maximal block.

Finally, regarding the BS*-tree, reaching the leaf level means that we reached one of the S*-tree pages, namely we reached part of the corresponding quadtree. Three situations might arise for the searched maximal block:

- it is a leaf in the corresponding quadtree, and therefore its color can be immediately found from the color table;
- it is contained in a leaf, and therefore its color can be found by looking to its ancestor, with at most an additional access on the previous page;
- it is an internal non-homogeneous node, and therefore all the contained features can be immediately found from the color table.

The report query. In a report query, the user asks for all the features contained in the queried window. In this kind of query the bitstring will play no role, since only when reaching the leaf level we look up at its bitstring to return the features whose corresponding positions in the bitstring are set to 1. The difference comes only from the fact that in BHL and BS* trees it is not necessary to reach the leaf level since internal nodes, along with their contained features, are stored as well and can therefore provide promptly the needed information.

The select query. The last window query is the selection query where the user asks for the blocks of the map inside the queried window where the wanted features are found. As in exist queries, for each maximal block searching starts by examining the entries at B^+-tree root. As already described, only the branches where the respective entry's bitstring has at least one position of the queried features set to 1, are followed. Once the leaf level is reached, then we search for the queried maximal block. As described in the previous subsection, if this searching is not successful, then we first try to see if an homogeneous ancestor exists in the tree. In such a case, the searched maximal block is output if the ancestor is homogeneous with respect to one of the queried features. If a homogeneous ancestor does not exist, then we look to all the descendants, and all those that are homogeneous with respect to one of the queried features are returned.

4. Performance evaluation

Attempting an evaluation at a conceptual level, by combining the advantages inherent in a structure such as the B^+-tree with the feature information which accompanies every thematic layer, we aimed at a better manipulation of the latter and at an improved performance in processing window queries. As it will be shortly shown, the main drive behind this reformation of B^+-tree entries, was the fact that the use of bitstrings through the whole structure of the index would provide the means to take advantage of the feature information in higher B^+-tree levels, thus avoiding traversing unqualified branches for queries based on features. Still, as experiments verified, the performance in location-based queries remained unaffected.

In order to show the advantages of the treecode representations of thematic layers in combination with the storage of the feature information in upper index levels, a comprehensive experimental evaluation is performed using data from various satellite images. More specifically, we experimented with three groups of data. The first group of images (i.e., 256×256 images) was downloaded from the GRASS site, a public domain GIS system, [2] while the second and third group of images (i.e., 512×512 and 1024×1024 images) were meteorological satellite views of European, Asian and North American regions. Specifically, the second group was from the Meteosat Imagery site, [3] while the third one was from the CNN site that concerned the weather forecast. [4]

All structures were implemented in C++ programming language under Windows NT and the experiments run on a Pentium II workstation. For an image of size 512×512 or 1024×1024 pixels, the corresponding B^+-tree will have a height of at most three levels. The window queries were performed on images of size 256×256, 512×512 and 1024×1024 containing 8, 16 and 64 features, respectively. The query windows were 1%, 5%, 10% and 25% of the image area. The page size was 1 Kbyte for smaller images and 2 Kbyte for larger ones, leading to a fanout of 84 and 169 entries, respectively. The emphasis was given in the exist and select query, since using this bitstring would not have any effect in the report query. For each image, 50 queries were performed for the four different window sizes and the results were averaged. Due to space limitations, only the results for the 1024×1024 images containing 64 features are shown, since the results are similar for all cases.

The selection of the queried features was based on their frequencies. Suppose that i features are to be selected out of n ones. First, the features are sorted according to decreasing frequency, and then, the 1st, the $\lfloor n/i \rfloor$th, $\lfloor 2n/i \rfloor$th, $\ldots$, $\lfloor (i-1)n/i \rfloor$th feature are selected. For instance, if $i = 4$ and $n = 64$, then we should select the 1st, 16th, 32nd and the 48th feature.

Our measurements taken in consideration the number of disk accesses, where we only counted the accessed leaves based on the fact that buffering techniques can eliminate the repeated traversal of internal nodes.

4.1. Space overhead

In the first set of experiments, the space overhead that was involved in each access method was measured. As explained in Section 3.2, no space overhead due to the bitstring in the B^+-tree nodes for the BSL-, BHL- and the BS^*-tree was considered. As a matter of fact, in the respective graph only the four columns of the IL-, BSL-, BHL- and the BS^*-trees counting the number of leaves are shown.

As it can be seen in Fig. 15, the (B)HL-tree is the worst method with respect to the space overhead, due to the storage of internal quadtree nodes in the B^+-tree leaves. The space occupied by the IL-trees was found by summing up the space size occupied for each one of the feature indices involved, i.e., the 64 indices of our experiment. It is obvious that the BS^*-tree has by far

[2] http://www.moon.cecer.army.mil.

[3] http://www.nottingham.ac.uk/~cczsteve/graphif.shtml.

[4] http://www.cnn.com/WEATHER/images.html.

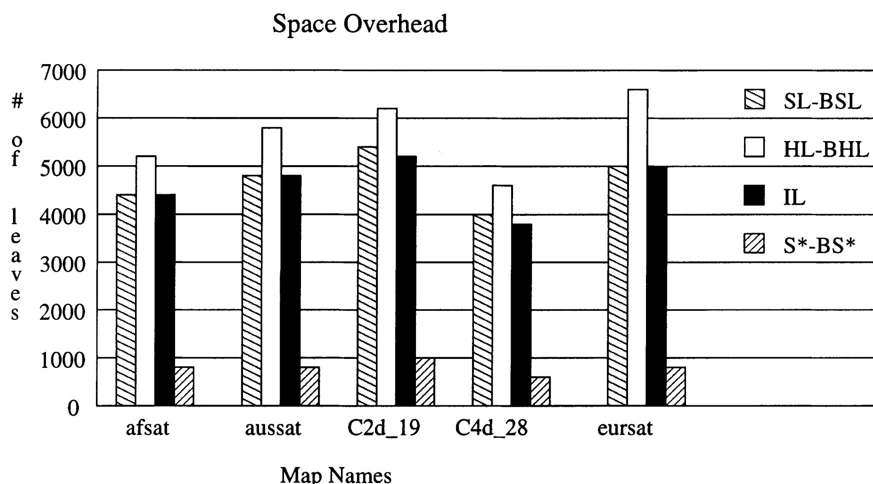

Fig. 15. Space overhead involved in five different 1024 × 1024 images containing 64 features.

less space overhead when compared to the other methods. As it was already explained, this was expected due to the very compact nature of treecode representations.

4.2. Exist query

Regarding the exist query, it exploits the outmost of the new technique. In the experiments performed, we searched for the existence of a varying number of features. Specifically, we were interested to find out whether one of the queried features existed inside the queried window. As soon as one of them was found, processing stopped.

All access methods using the proposed enhancement could quite early eliminate the traversal of index branches, where the desired information did not exist. Therefore, they

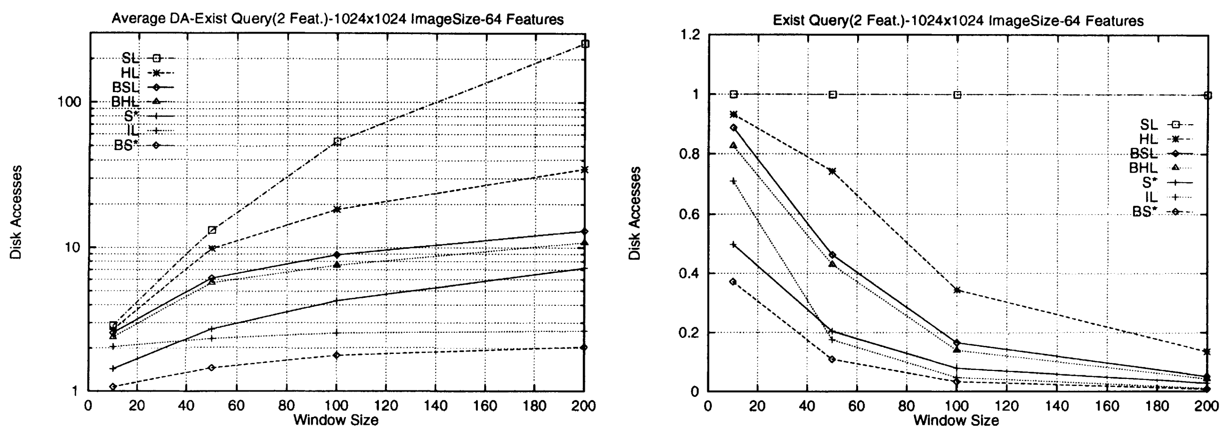

Fig. 16. Exist query where two features were queried, image size 1024 × 1024, 64 features: (left) averaged results, (right) normalized results.

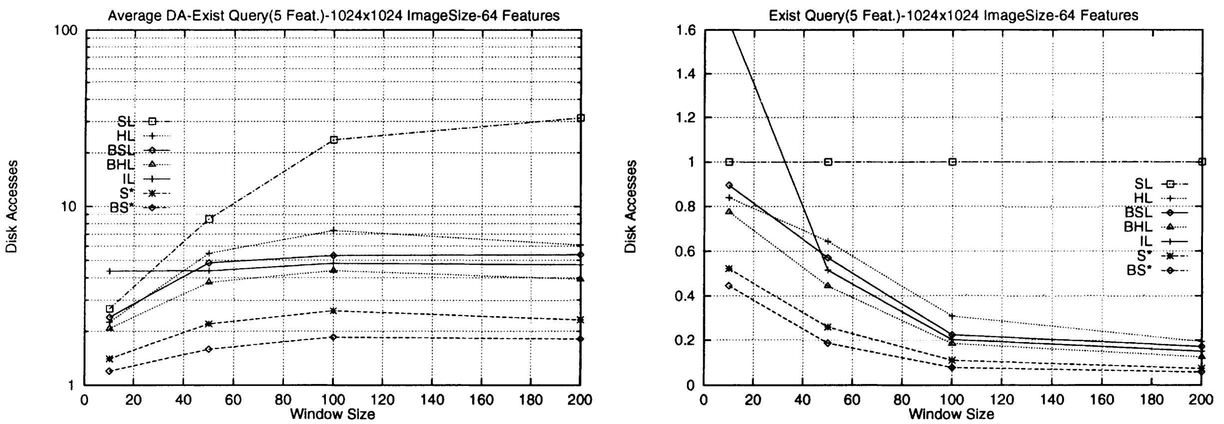

Fig. 17. Exist query where five features were queried, image size 1024×1024, 64 features: (left) averaged results, (right) normalized results.

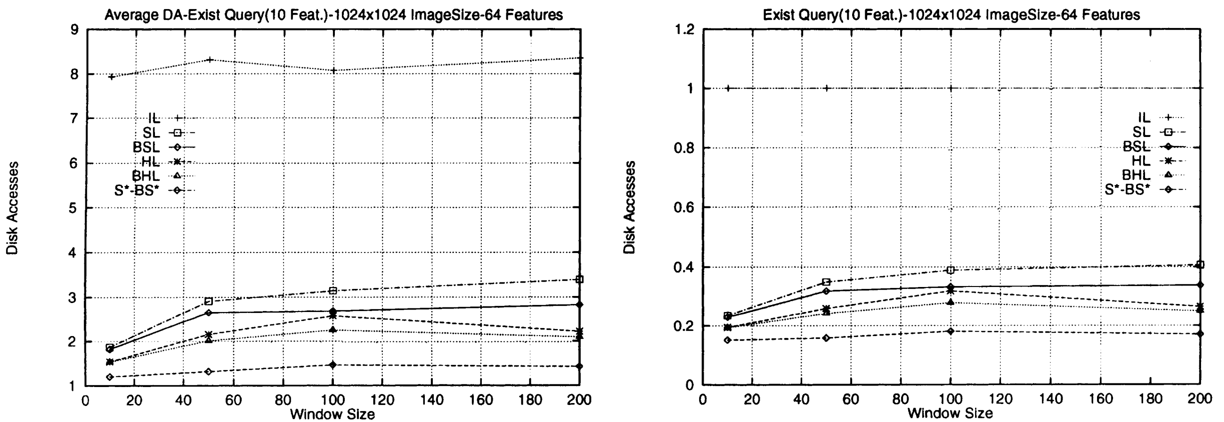

Fig. 18. Exist query where 10 features were queried, image size 1024×1024, 64 features: (left) averaged results, (right) normalized results.

always performed better than their counterparts that made no use of this information, as it can be seen in Figs. 16–18. On the right-hand side, the normalized results with respect to the worst method are shown. From these figures, it is easy to verify that with an increasing number of features the performance of IL-trees deteriorates. This outcome can be seen more clearly in Fig. 19 where we experiment with a fixed window size, while increasing the number of queried features. This was expected since we cannot predict the absence of some of the queried features and, therefore, avoid the traversal of their respective indices.

In all previous graphs, it is obvious that the BS*-tree outperforms all other structures, showing a constant behavior independent of the number of features searched. On the other hand, the bad performance of SL-trees is explained by the fact that we have to find the quadblocks comprising the specific region and check the contained features.

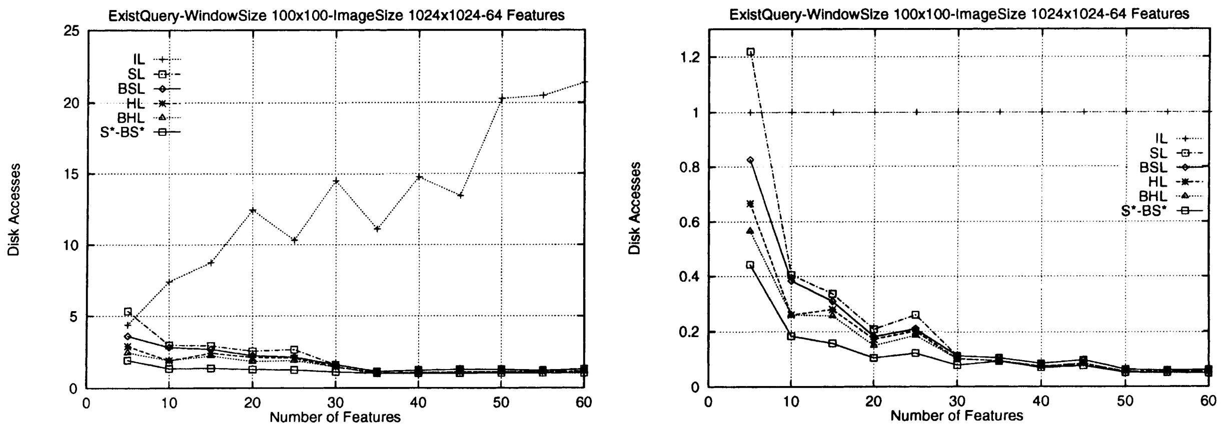

Fig. 19. Exist query for a varying number of queried features, image size 1024 × 1024, 64 features, query window 100 × 100: (left) averaged results, (right) normalized results.

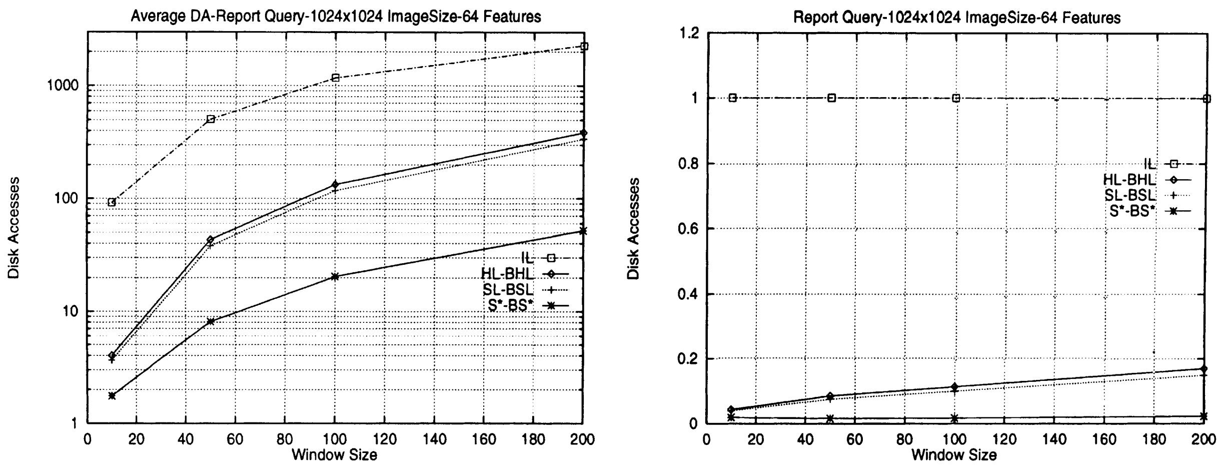

Fig. 20. Report query on images of 1024 × 1024 size containing 64 features: (left) averaged results, (right) normalized averaged results.

4.3. Report query

Although our new technique has no effect for the report query, since all the features that exist inside the queried window have to be returned, we chose to experiment with this query because it can clearly show two issues. Firstly, the extremely bad performance of the IL-trees due to the fact that all independent trees have to be searched to check whether the respective feature exists in the queried window. Secondly, it is a first indication of the very good performance of the treecode representations, as it can be clearly seen in Fig. 20. On the right-hand side of this figure, the normalised results with respect to the IL-trees, as the worst method, are depicted.

4.4. Select query

Regarding the select query, in the first set of experiments, for each queried window we searched for the blocks where the queried features were found. The results can be seen in Figs. 21–23.

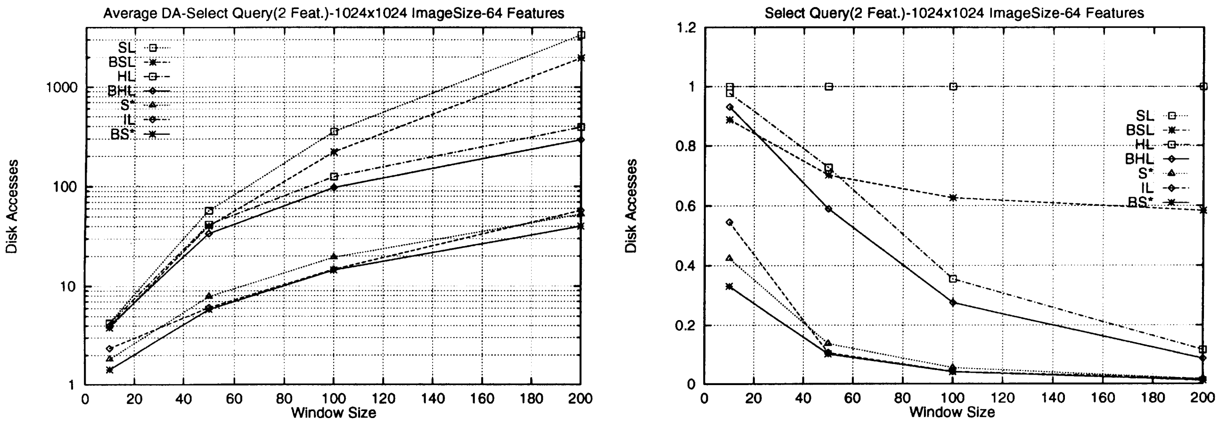

Fig. 21. Select query where two features were queried, image size 1024×1024, 64 features: (left) averaged results, (right) normalized results.

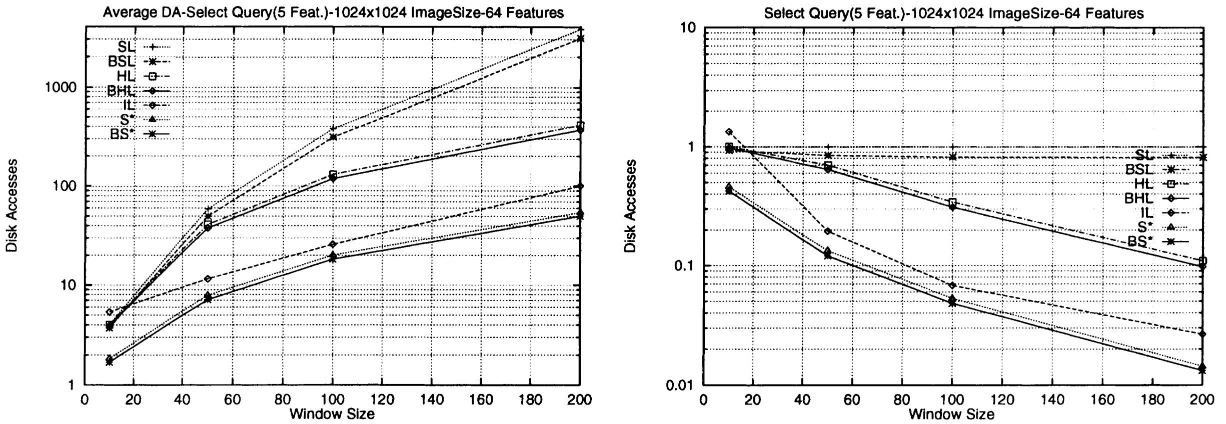

Fig. 22. Select query where five features were queried, image size 1024×1024, 64 features: (left) averaged results, (right) normalized results.

Again, on the right-hand side of the graphs the normalized results with respect to the worst method are shown. As observed before, also in these graphs, the access methods where the new bitstring was embedded always outperformed the alternative ones without the bitstring.

More specifically, in Figs. 21–23 the methods' performance is shown for images of size 1024×1024, when 2, 5 and 10 features out of 64 are queried, respectively. Here, it is observed that the IL-trees seem to have a good performance in comparison to the other leafcode based methods. However, they do not show a stable behavior since their performance will always depend on the number of queried features as well as on their occurrence frequency. More specifically, sparse features with low probability occurrence will create very small trees, possibly of even one node only, while the tree of more frequent feature will be bigger and will affect the method's performance substantially. The performance of IL-trees becomes worse with an increasing number of features, a tendency that is more apparent in Fig. 24, where for a fixed query window the number of queried features is increased.

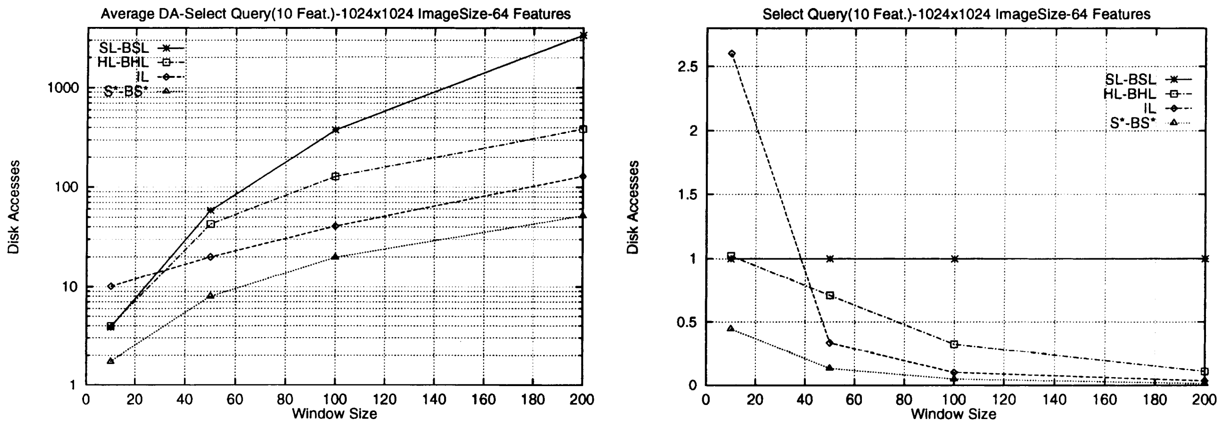

Fig. 23. Select query where 10 features were queried, image size 1024 × 1024, 64 features: (left) averaged results, (right) normalized results.

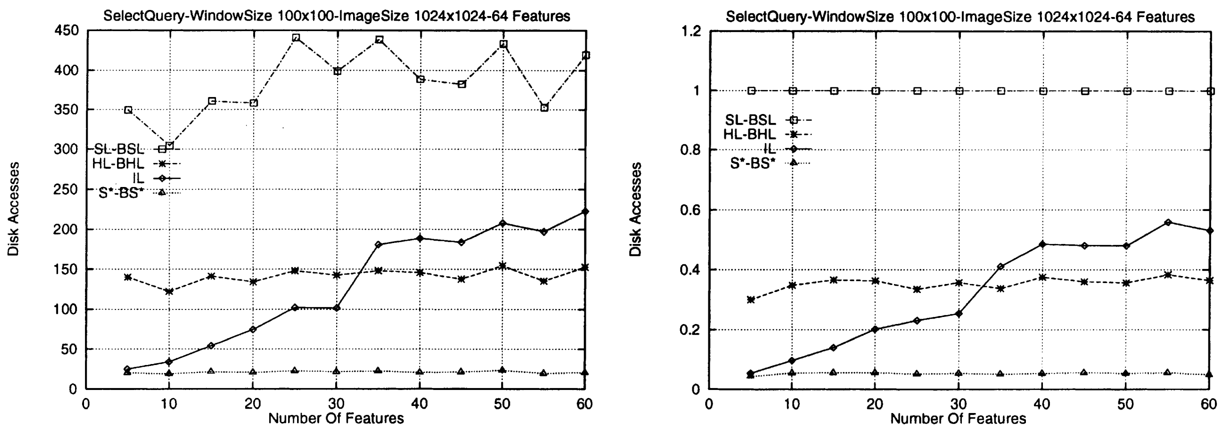

Fig. 24. Select query for a varying number of queried features, image size 1024 × 1024, window size 100 × 100: (left) averaged results, (right) normalized results.

On the other hand, the S*-tree shows a similar performance to the IL-trees for a small number of features, while, due to its compact nature, its behavior is improved when the number of features is increasing. The BS*-tree shows constantly the best behavior and outperforms the IL-trees especially for a large number of features.

As a conclusion, the previous experiments show that the technique of posting feature information by means of a bitstring to upper levels of a linear-type quadtree results in superior performance for:

- Large images, because they produce a greater number of locational keys and, therefore, the B^+-tree height will be bigger. Hence, filtering based on queried features can be applied at higher tree levels and, thus, traversing some B^+-tree branches can be avoided.
- The exist query, because at a very early stage the bitstring helps responding to the user query.

- Images with concentrated features, such as typical GIS raster images. Otherwise, the features will be spread to various locations leading to non-homogeneous leaf nodes, as far as the features are concerned, and consequently to bitstrings with many positions set to 1 that allow no filtering.

5. Conclusions

In this paper, a collection of secondary storage spatial access methods for representing large sets of two-dimensional data containing multiple non-overlapping features is reviewed. Most of these techniques use a B^+-tree to index a list of pages containing the data. A new coding technique for the index of such a B^+-tree is presented, which allows one to improve the performance of the various proposed approaches. Our experiments on a gallery of images, consisting of window queries that involve multiple features, show that the number of disk accesses is usually lowered by 15%.

Our future work will be in the direction of applying our new technique to spatial access methods used to represent images containing multiple overlapping features.

References

[1] R. Bayer, K. Unterauer, Prefix-B trees, ACM Transactions on Database Systems 2 (1) (1977) 11–26.
[2] V. Gaede, O. Güenther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998) 170–231.
[3] I. Gargantini, An effective way to represent quadtree, Communications of the ACM 25 (12) (1982) 905–910.
[4] O. Güenther, Efficient structures for geometric data management, Lecture Notes in Computer Science, vol. 337, Springer, Berlin, 1988.
[5] W. de Jonge, P. Scheuermann, A. Schijf, S^+-trees: an efficient structure for the representation of large pictures, Computer Vision, Graphics and Image Processing: Image Understanding 59 (3) (1994) 265–280.
[6] E. Kawaguchi, T. Endo, M. Yokota, Depth-first expression viewed from digital picture processing, IEEE Transactions on Pattern Analysis and Machine Intelligence (1983) 373–384.
[7] Y. Manolopoulos, E. Nardelli, A. Papadopoulos, G. Proietti, MOF-tree: a spatial access method to manipulate multiple overlapping features, Information Systems 22 (8) (1997) 465–481.
[8] Y. Manolopoulos, Y. Theodoridis, V. Tsotras, Advanced Database Indexing, Kluwer Academic Publishers, Dordrecht, 1999.
[9] E. Nardelli, G. Proietti, Efficient secondary memory processing of window queries on spatial data, Information Sciences 80 (1994) 1–17.
[10] E. Nardelli, G. Proietti, Time and space efficient secondary memory representation of quadtrees, Information Systems 22 (1) (1997) 25–37.
[11] E. Nardelli, G. Proietti, S^*-tree: an improved S^+-tree for colored images, in: Proceedings of the Third East European Conference on Advances in Databases and Information Systems (ADBIS'99), vol. 1691, Springer, LNCS, Maribor, Slovenia, 1999, pp. 156–167.
[12] G. Proietti, An optimal algorithm for decomposing a window into maximal quadtree blocks, Acta Informatica 36 (4) (1999) 257–266.
[13] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990.
[14] H. Samet, Applications of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990.
[15] M. Vassilakopoulos, Y. Manolopoulos, Analytical comparisons of two spatial data structures, Information Systems 19 (7) (1994) 569–582.

Eleni Tousidou received a B.Sc. degree from the Department of Informatics of the Aristotle University of Thessaloniki in 1996. She is currently a Ph.D. student at the same institution. Her research interests include query processing and access methods in object-oriented databases and spatial databases.

Yannis Manolopoulos received a B.Eng (1981) in Electrical Eng. and a PhD (1986) in Computer Eng., both from the Aristotle Univ. of Thessaloniki. He has been with the Department of Computer Science of the Univ. of Toronto, the Department of Computer Science of the Univ. of Maryland at College Park and the Department of Electrical and Computer Eng. of the Aristotle Univ. of Thessaloniki. Currently, he is Professor at the Department of Informatics of the latter university. He has published over 100 papers in refereed scientific journals and conference proceedings. He is co-author of a book on ''Advanced Database Indexing'' by Kluwer. He is also author of two textbooks on data/file structures, which are recommended in the vast majority of the computer science/engineering departments in Greece. His research interests include spatiotemporal databases, databases for web, data mining, data/file structures and algorithms, and performance evaluation of secondary and tertiary storage systems.

Guido Proietti received his degree of Doctor in Mathematics from the University of Rome 'La Sapienza' in 1990. In 1996 he joined the Department of Pure and Applied Mathematics at University of L'Aquila, where he currently works as Assistant Professor of Computer Science. In 1998 he was a visiting scientist at the Department of Computer Science of Carnegie Mellon University, Pittsburgh, PA. His current activity includes the study of efficient spatial access methods, query processing and evaluation, and the design of algorithms for managing transient failures in networks. He is also interested in graph algorithms, computational geometry and computer vision. He has published more than 30 refereed articles.

Enrico Nardelli is full professor of Computer Science at the University of L'Aquila and invited research scientist at the Institute for System Analysis and Computer Science of the Italian National Research Council in Rome. He carries out research in various field of theoretical and applied Computer Science, namely the design and analysis of algorithms and data structures, the definition and analysis of data models, the definition and design of tools and environments for interaction with information systems. He has authored more than 50 refereed papers in the fields of interest, published in the most reputed international scientific journals. He has been Team Leader and Principal Investigator since 1989 in more than 10 national and international Research and Development Projects in Computer Science.