A PERFORMANCE COMPARISON OF QUADTREE-BASED ACCESS METHODS FOR THEMATIC MAPS *

Eleni Tousidou
Data Engineering Lab
Department of Informatics
Aristotle University
54006 Thessaloniki, Greece
eleni@delab.csd.auth.gr

Yannis Manolopoulos
Data Engineering Lab
Department of Informatics
Aristotle University
54006 Thessaloniki, Greece
manolopo@delab.csd.auth.gr

ABSTRACT

In this paper, the efficient manipulation of thematic maps that contain multiple non-overlapping features is investigated. New methods based on Linear quadtrees are proposed and their performance is compared to that of similar structures. More specifically, window queries involving multiple features are described and tested having the number of disk accesses as a performance measure. Experimentally, it is shown that the proposed methods have a stable behavior and, in general, outperform the previous structures with respect to time and space complexity.

Keywords

spatial databases, region quadtrees, multiple features, superimposed bitstrings, window queries

1. INTRODUCTION

Today, the manipulation of large volume two-dimensional data representing multiple features is of great interest for a variety of applications (e.g. in image databases, geographical information systems (GISs), scientific visualization, 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. The interested reader can refer to [4; 6; 13] for interesting surveys on the topic.

The quadtree is a spatial access method based on the hierarchical image decomposition. Each image is regularly and successively decomposed into four quadrants until a *homogeneous* maximal block, with respect to the contained feature, is reached. Quadtrees can be implemented either in main memory or in secondary memory as pointer-based or pointerless structures, respectively. As far as secondary storage is concerned, two types of quadtree representations have been presented:

- by extracting the collection of homogeneous (black) leaves, which evidently carry semantic information [5], and
- by traversing the quadtree in preorder and forming a string, which is called DF-expression [8].

Since the need is on the random access of the quadtree leaves, the focus will be on the first representation.

In this work the focus is in the manipulation of raster thematic maps that contain multiple non-overlapping features, i.e. maps where each pixel contains one and only one feature and where pixels of the same color are aggregated into patches. For example, such thematic maps can be widely met in GISs. Such maps represent distinct thematic layers where each layer, as its name states, has a distinct theme (or subject). This theme could be the geology of the land, the elevation of the area, or the soil type found in the depicted area. For example, each type of soil occupies a certain space of the area and, apparently, no other type of soil can co-exist at the same space. This way, a map of non-overlapping categories is obtained. In the following, the words features, colors and categories will be used interchangeably.

Some of the most important types of queries applied to spatial data are window queries, since they allow extracting only the needed information from the whole image. More specifically, the window query types under examination are the following:

- exist query (w, f_i, f_j, ..., f_k): check whether one or more features exist inside the window w.
- report query (w): report all features that are found inside the window w.
- select query (w, f_i, f_j, ..., f_k): select all homogeneous blocks inside the window w containing feature f_i, f_j, ... or f_k.

The efficient processing of window queries has already been studied by Nardelli and Proietti, who proposed adjusting region Linear quadtrees to manipulate the feature information [10]. The Hybrid Linear quadtree (HL-tree) was introduced as an enhancement to the previous method [11], whereas the MOF-tree, which was based on the HL-tree, was presented

*Research performed under the European Union's TMR Chorochronos project, contract number ERBFMRX-CT96-0056 (DG12-BDCN).

SAC'00 March 19-21 Como, Italy
(c) 2000 ACM 1-58113-239-5/00/003>...>$5.00

as a structure to efficiently manipulate images with multiple overlapping features [9]. Apart from these structures, Tanimoto and Pavlidis introduced a multi-resolution representation of images, the Pyramid data structure [14], two variations of which were later proposed by Aref and Samet [1], and Nardelli and Proietti [12]. Finally, in [7] a different in philosophy structure was presented, the S^+trees, which are based on DF-expressions. The latter structure basically manipulates black-and-white images and though it can be adjusted to manipulate multiple non-overlapping features, this would be performed in a less efficient way due to possible large space waste.

In this paper, a quadtree-based approach will be described and examined aiming to the efficient handling of thematic layers with multiple non-overlapping features. The focus will be on efficient processing of queries, which involve *both* the features and the spatial object locations as well. In the sequel, only the pointerless representation of this structure will be examined, although the same method could be also invariably applied to the pointer-based representation. By comparing the new structure to the previously proposed methods that were also based on Linear quadtrees, it will be shown that there is considerable gain achieved both in terms of storage space and time complexity.

The rest of the paper is organized as follows. In Section 2, some of the quadtree-based access methods that have already been presented in the past are reviewed. Also, the points which motivated in introducing the new method will be mentioned. In Section 3, the new structure will be described in detail, along with some new algorithms for the efficient performance of window queries. In Section 4, some representative results that were derived from the conducted experiments will be shown. Section 5 contains concluding remarks and some directions of possible future work.

2. RELATED WORK AND MOTIVATION

As already mentioned, in this work the focus lies on supporting raster thematic maps containing multiple non-overlapping features. In our representation of maps, each distinct feature is represented by a different color. This means that each pixel of the map will contain one and only one color in contrast to the maps with overlapping features, where a block of the map could contain 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.

2.1 Simple Linear Quadtree

A first step toward better exploitation of thematic maps was the use of simple Linear quadtrees (SL-trees) [10]. In fact, the latter structure is the original Linear quadtree [5], 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:

1. the *locational key*, whose digits reflect successive quadrant subdivision,
2. the *value field*, 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.

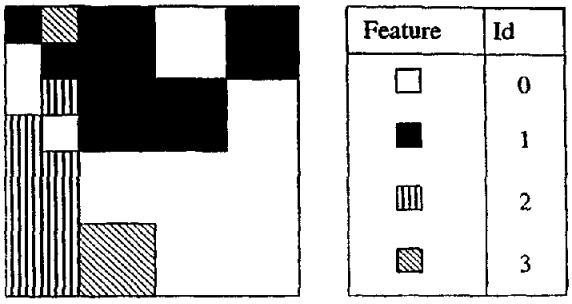

Figure 1: An 8x8 image and the feature-id table.

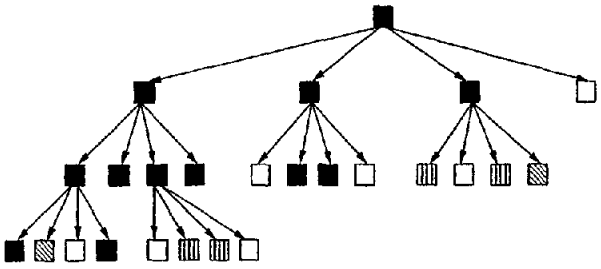

Figure 2: The Quadtree representing the image of Fig. 1.

In Figure 1, an 8x8 image is depicted which contains four non-overlapping features. The feature id's are listed in the table in the right part of Figure 1. In Figure 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 is depicted. For example, the leaf with locational code 132 has a value field equal to 2, since the feature contained in the corresponding subimage is the one having id=2.

(111,1), (112,3), (113,0), (114,1), (120,1), (131,0),
(132,2), (133,2), (134,0), (140,1), (210,0), (220,1),
(230,1), (240,0), (310,2), (320,0), (330,2), (340,3), (400,0)

2.2 Hybrid Linear Quadtree

Another approach has been proposed in [11], where apart from the quadtree leaves, internal nodes are also registered in the B^+tree. However, internal nodes are heterogeneous blocks since they contain more than one feature. Internal nodes are coded with a locational key, whereas to represent the feature information each quadtree node will accommodate a bitstring of size equal to the number of features existing in the thematic map. A specific bit of the bitstring is set to 1, if and only if the respective feature exists in the represented quadrant.

Evidently, this approach has a storage overhead due to the storage of the internal nodes and the corresponding bitstring. In [11] it is explained that this overhead is not significant since the number of internal nodes are not larger than the 1/3 of the number of leaves and, consequently, the asymptotic space occupancy will remain the same. Also, since computer systems are based on a 32-bit architecture,

space can always be saved for at least 16 features, which is a reasonable number of features.
Applying the method of HL-trees on the thematic layer of Figure 1, the following list of nodes, either internal or external, will be created and stored in the B$^+$tree leaves. As can be seen, each locational key is accompanied by a bitstring of size four, since four are the features existing in the represented thematic map. Nodes accompanied by a bitstring that has only one bit set to 1 are homogeneous quadtree leaves covered by the respective feature. For example, the locational code 110 that corresponds to an internal node is related to bitmap 1101, since all features exist in the represented quadrant apart from the feature with id=2.

(000,1111), (100,1111), (110,1101), (111,0100), (112,0001), (113,1000), (114,0100), (120,0100), (130,1010), (131,1000), (132,0010), (133,0010), (134,1000), (140,0100), (200,1100), (210,1000), (220,0100), (230,0100), (240,1000), (300,1011), (310,0010), (320,1000), (330,0010), (340,0001), (400,1000)

2.3 Independent Linear Quadtrees

A straightforward approach adopted for comparison purposes is to use Independent Linear quadtrees (IL-trees). As its name states, a separate Linear quadtree is used for each feature resulting in as many Linear quadtrees as the number of features in the thematic map. 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 because of the need to traverse and join the results from multiple indices.

2.4 Query Manipulation and Motivation

The original Linear quadtree was firstly proposed for black and white images, where the only information stored were the addresses of black quadrants [5]. This means that the structure worked fine for window queries, since no feature filtering was necessary, whereas the only expectation was the good performance of the B$^+$tree when queried with the spatial location of objects based on the locational quadtree codes. The next step towards the adjustment of the original method to the efficient manipulation of multiple features (i.e. thematic maps) was confined to the maintenance of feature information in the quadtree leaves.
As already seen, the bottom line in all previous methods is that this information will be stored in the B$^+$tree leaves making it impossible to take further advantage of the features as a spatial filter. For example, though the HL-tree retains information for internal and external quadtree nodes, both of them are stored only at the B$^+$tree leaf level. Consequently, we cannot take advantage of it in higher B$^+$tree levels to avoid traversing some branches for queries based on feature information. In the IL-trees, there is no need for filtering but instead several indexes have to be traversed to answer queries involving multiple features.
In the present paper, a method aiming at achieving better exploitation of feature information in combination with spatial location is proposed. This is not always trivial since feature information and spatial location are two orthogonal issues that have no relation with each other.

3. PROPOSED METHOD

In spatial data processing, for example in GISs 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 deduced to searching for the specific color with which this category is represented in the map. The efficient processing of queries which are based both on the feature as well as on the spatial object location is pursued.
Assume a user query for information from a thematic map of size $T \times T$ that contains k non-overlapping features, where $T=2^m$ and m is a positive integer. Since thematic maps contain more than one feature, the regular space decomposition process will stop when a homogeneous block, i.e. a maximal block that is fully covered by one feature only, is reached. As already seen in the previous examples, for the sake of uniformity the image background is treated as a separate feature.

3.1 Description of the New Method

Here, the pointerless representation of the new method will be described since the pointer-based one can be developed in a similar way. The Linear quadtree uses the B$^+$tree as a storage medium for the locational keys of the thematic map, and as an efficient structure for the fast retrieval of the represented information. The proposed method is based on a restructuring of the B$^+$tree nodes.
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 needed information will be stored in the upper levels of the B$^+$tree.
In the following it will be made obvious that the proposed method of posting to the parent node a second-order bitstring can be applied to the SL-tree and the HL-tree as well. The structures derived by such a use of the bitstring are named BSL-tree and BHL-tree, respectively. According to the new proposal each entry in the B$^+$tree leaves will consist of:

- a locational key of the represented quadrant,
- a value field containing the color of the specific quadrant or (when applied to the HL-tree) a bitstring of size equal to the number of features. This bitstring is encoded so that the i-th bit is set to 1, if and only if the feature with id=i exists in the respective quadrant.

However, according to the new method, a second-order bitstring is introduced in the entries of the internal nodes. More specifically, the bitstrings (or value fields) of 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 i-th 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 in the 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.

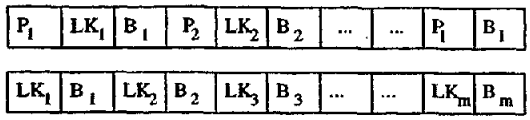

Figure 3: Layout of internal and leaf nodes.

The layout of internal and leaf nodes is illustrated in the upper and lower part of Figure 3, respectively. Basically, leaves comprise of m pairs (locational key, value field or bitstring), whereas internal nodes comprise of l triplets (tree pointer, locational key, bitstring), where l is the tree fanout[1]. The size of a tree pointer is 4 bytes (32 bits). The value field of the leaves is used in the case of the BSL-tree since only the color of the homogeneous quadrant needs to be stored. The bitstring is used in the case of the BHL-tree since, in case of a heterogeneous quadrant, it might be needed to store more than one feature.

Regarding the internal nodes of the tree, since their average space requirements 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. On the other hand, for images of size 1024×1024 or 2048×2048 pixels, the quadtree depth is 10 or 11 levels and the length of the locational key is 24 or 26 bits, respectively. Assuming that the locational key is represented by an integer, in the case of the BSL-tree, as far as the leaves are concerned, the remaining one or two bytes can be used in order to store the color.

As stated in the previous, this is the reason for ignoring the space occupied by the value field when calculating the introduced storage overhead in the case of the BSL-tree. Also, there is no change in the case of the BHL-tree since the leaves of the original HL-tree already contained those bitstrings.

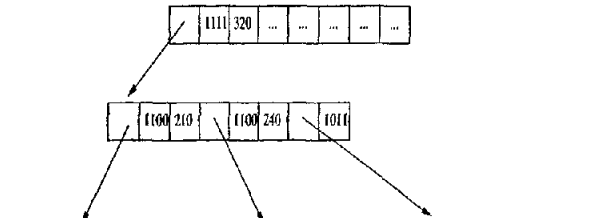

Figure 4: Branch of the B^+tree.

As expected, the B^+tree root and some entries at the level below the root will probably contain bitstrings with many positions set to 1 since the respective subtrees will contain almost all features. More specifically, in the case of widely spread features, the number of bitstring positions set to 1 will be large, resulting in visiting all queried maximal blocks inside the query window. It should be noticed though, that for such thematic maps, whatever the method used, they will result in a great number of disk accesses since the queried features will be spread in almost all B^+tree leaves. On the other hand, if the queried maps contain features which are concentrated in specific areas, this would result in only a few positions set to 1 in the B^+tree leaves, allowing to avoid visiting some irrelevant tree branches. Figure 4 illustrates an example of a BHL-tree, where m=3 and l=3.

[1]To be more precise, internal nodes store l bitstrings and tree pointers toward the l leaves, and l–1 locational keys which are sufficient to traverse the tree.

3.2 Algorithm Description

3.2.1 *The Creation Algorithm*

The first step to build the proposed structure (either BSL-trees or BHL-trees) is the image decomposition, which will result in a list of maximal blocks. In the case of BSL-trees, this list will contain only homogeneous blocks, while in the case of the BHL-tree this list will also contain the internal quadtree nodes, that is the non-homogeneous blocks. Each entry in the list will consist of the locational code of the represented quadrant followed by the bitstring standing for the feature(s) that is(are) 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 of the stored quadrants to the upper B^+tree levels; i.e. a bitstring for each entry in all B^+tree nodes is extracted. At the leaf level this bitstring is identical to the original (in the sense of HL-trees) bitstring that was generated in the quadtree list. For the internal B^+tree levels though, the superimposition method (OR-ing) is used to propagate the feature information to higher tree levels. Thus, by superimposing the bitstrings which exist 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.

3.2.2 *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 comprises a maximal block of the image inside the window [2]. In the following, it is explained how these queries proceed according to the proposed method.

The Exist Query

Consider a query over a specified window, where a search for the existence of features f_i, f_j, ... or 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, only then the respective subtree is followed; otherwise we skip to the next entry of the node. The same procedure is followed at the remaining tree levels; searching stops only when the leaf level is reached. However, in case of the 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 the 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, one more disk access may be needed to retrieve the previous or the next page of the reached leaf, where the ancestor or the descendant of the desired locational key may be stored, respectively. In case of the BHL-tree, it is certain that the searched maximal block will be located since all quadrants are stored, and by looking at its bitstring the question can be answered immediately gaining one disk access.

The Report Query

In a report query, the user asks for all the features that comprise the queried window. In this kind of query the bitstring will play no role, since only when reaching the leaf level its bitstring is searched to return the features whose corresponding positions in the bitstring are set to 1. At the leaf level, the BHL-tree will work in exactly the same way as described in [11], i.e. similarly to the exist query.

The Select Query

The last window query is the selection query where the user asks for the blocks of the map in the queried window where the wanted features are found. As in the case of the exist query, 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 the queried maximal block is searched. As described in the previous subsection, if this searching is not successful, then we first try to see if its descendants exist in the tree. In such a case, only those descendants that are homogeneous with respect to one of the queried features are returned. If the descendants are not stored in the tree, then we have to look for its ancestor. Only then, it can be verified whether the ancestor is covered by one of the queried features or not. This procedure operates in exactly the same way as in the HL-tree, since we need to report the exact blocks where the features exist and in this case searching cannot be avoided with the use of the quadtree's internal nodes.

4. PERFORMANCE EVALUATION

Detailed experiments were performed to compare the SL-tree, the HL-tree and the IL-trees against the new proposed technique applied to both the SL-tree (BSL-tree) and on the HL-tree (BHL-tree). All kinds of queries were tested, however the results of the report query were only used to show the drawbacks of the IL-trees, as it will be shortly explained in the following section. The exist and selection query were considered as the most important ones, since they can extensively show how the bitstring can accelerate processing in queries involving features. The selection of the queried features was based on their frequencies. Suppose that i features are to be selected out of j ones. First, the features were sorted according to decreasing frequency and, then, the 1st, the $\lfloor \frac{j}{i} \rfloor$-th, $\lfloor \frac{2j}{i} \rfloor$-th, ..., $\lfloor \frac{(i-1)j}{i} \rfloor$-th feature was selected. For instance, if i=4 and j=64, then the 1st, the 16th, the 32nd and the 48th feature should be selected.
All structures were implemented in C++ programming language under Windows NT and the experiments run on a Pentium II workstation. For a thematic map of size 256×256, 512×512 or 1024×1024 pixels, the B^+tree that will be created will have a height of at most 4 levels. The window queries of size 25×25, 50×50 and 100×100 pixels were executed on 256×256 and 512×512 images containing 8, 16 and 64 features. The page size used was 1K for smaller maps and 2K for larger maps leading to a fanout of 84 and 169 entries, respectively. The number of maximal blocks created for each thematic map ranges approximately from 40,000 to 240,000. The first group of thematic layers (i.e. 256×256 images) was downloaded from the GRASS site, a public domain GIS system[2], while the second group of maps (i.e. 512×512 maps) were meteorological satellite views of Europe and Asian regions from Meteosat Imagery site[3]. The measurements were based on the number of disk accesses where only the accessed leaves were counted. For each thematic map, 50 queries were performed for four different window sizes and the results were averaged. Due to space limitations, the results for the 512×512 images only are shown since the conclusions are similar for all cases.

4.1 Space Overhead

In the first set of experiments the space overhead involved in each method was measured. As explained in Section 3.1, no space overhead due to the use of the bitstring in the B^+tree nodes for the BSL-tree and the BHL-tree was considered. As a result, both SL and BSL nodes on one hand, and HL and BHL nodes on the other hand will have exactly the same number of entries for a given page size. This is the reason why in the experiments performed measuring the space overhead, only the three columns of the IL-trees, the BSL-trees and the BHL-trees are shown. The column of the SL-tree would be exactly the same to that of the BSL-tree, as the column of the HL-tree would be exactly the same to that of the BHL-tree.

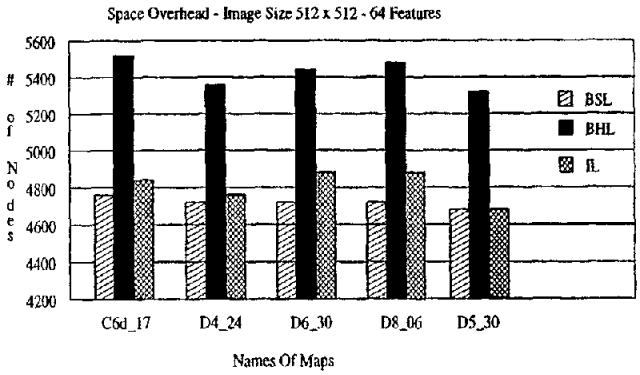

Figure 5: Space Overhead involved in five different 512×512 images containing 64 features.

As can be seen in Figure 5, the (B)HL-tree is the worst method regarding the space overhead, due to the storage of internal quadtree nodes in the B^+tree leaves and upwards. 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 also noticed that the (B)SL-trees are almost always the best ones, whereas the IL-trees are close with respect to the storage overhead.

[2]http://moon.cecer.army.mil
[3]http://www.nottingham.ac.uk/~cczsteve/graphif.shtml

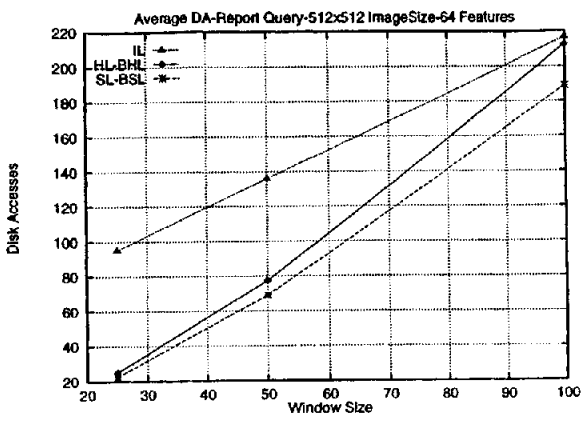

Figure 6: Averaged results of a report query on images of 512×512 size containing 64 features.

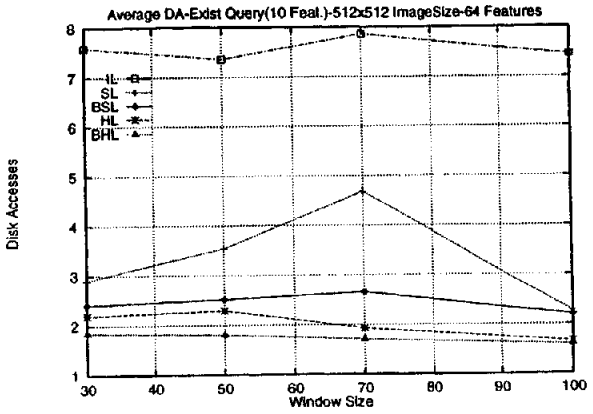

Figure 9: Averaged results for an exist query where 10 features were queried, image size 512×512, 64 features.

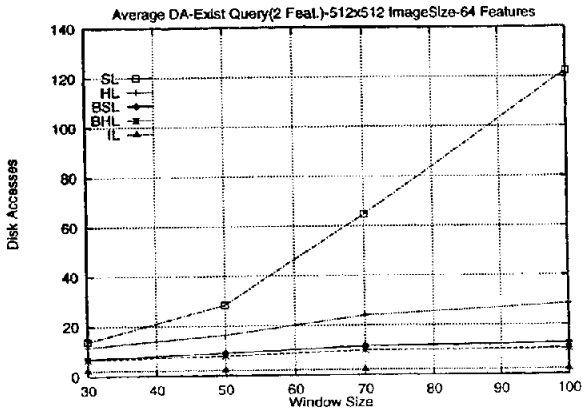

Figure 7: Averaged results for an exist query where 2 features were queried, image size 512×512, 64 features.

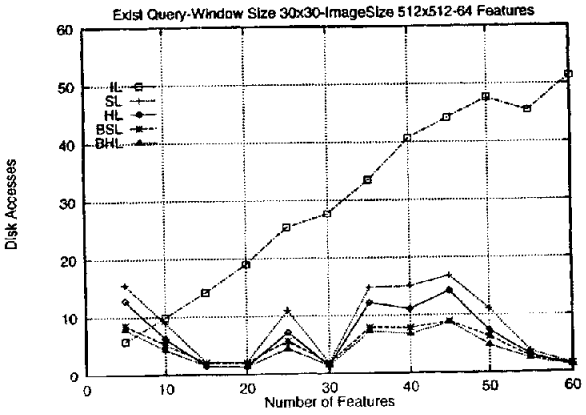

Figure 10: Averaged results for an exist query for a varying number of queried features, image size 512×512, 64 features, query window 30×30.

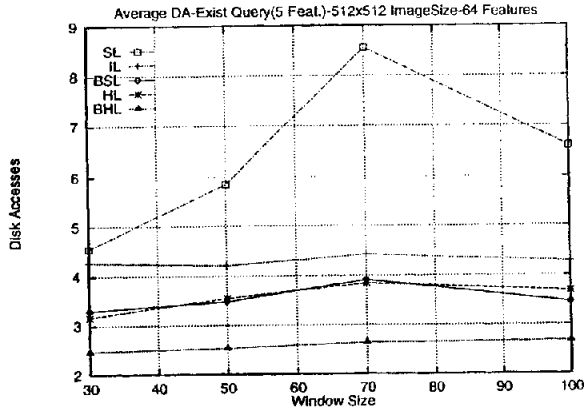

Figure 8: Averaged results for an exist query where 5 features were queried, image size 512×512, 64 features.

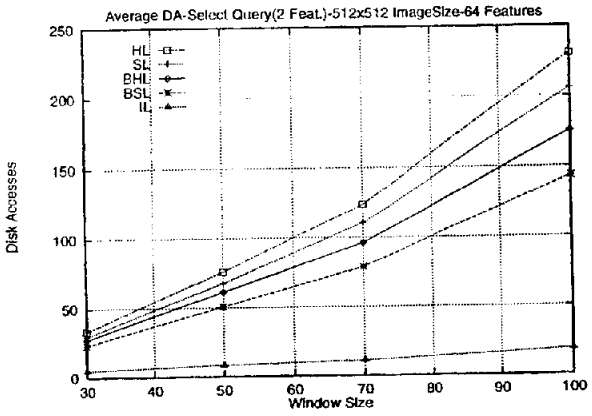

Figure 11: Averaged results for a select query where 2 features were queried, image size 512×512, 64 features.

4.2 Report

As described previously, the report query returns the features that exist inside the queried window. As far as the IL-trees are concerned, this means that for our data set, 64 indices should be visited to check whether the corresponding feature exists inside the window. This explains the reason why IL-trees have by far the worst performance, as depicted in Figure 6. The SL- and BSL-tree have exactly the same performance, since the bitstring is not taken into any consideration. The same applies for the case of HL- and the BHL-tree. The BSL-tree is the best access method, while the slightly worse performance of the BHL-tree can be explained by considering the larger number of leaves.

4.3 Exist

As far as the exist query is concerned, the performance difference of each examined method can be seen clearly. For each queried window, the existence of a number of features was searched. More specifically, the focus was in finding out whether at least one of the queried features existed inside the queried window. As soon as one of them was met, processing stopped.

In Figures 7, 8, 9 the results, when 2, 5 and 10 features are queried respectively, are illustrated. A first observation is that the methods using the bitstring always perform better than their counterparts which do not use a bitstring. In addition, the BHL-tree outperforms all methods for more than 2 features. This is explained by considering two facts:

- the HL-tree stores locational keys for internal quadtree nodes as well. As explained in Section 3.2.2, this can lead to less disk accesses since very soon it can be identified whether the region that it is looked for is homogeneous or not, and which features it contains.
- at upper levels, the use of the bitstring can help skipping those tree portions that do not contain any of the queried features.

The bad performance of the SL-tree is explained by the fact that the specific quadblocks of which, the region that is searched is comprised, have to be found out in order to check the contained features. As far as the IL-trees are concerned, all the feature-indices have to be processed always without knowing whether the specific feature is contained inside the queried window or not. In addition, the order of accessing the independent indices is random and there is no means to bypass non useful indices. Figure 10 depicts how the IL-trees' behavior worsens with respect to the number of features.

4.4 Select

Regarding the select query, in the first set of experiments for each queried window the blocks, where the queried features were found, were searched. The results can be seen in Figures 11, 12 and 13. As observed before, the methods where the new bitstring is embedded always perform much better than the respective ones without the bitstring since we can avoid to visit tree branches where the queried features do not exist.

More specifically, in Figures 11, 12 and 13 where we search for 2, 5 and 10 features respectively, it can be observed that the IL-trees seem to have a good performance in comparison to the other methods. However, their performance is not stable since it will always depend on the number of queried features (as demonstarted in Figure 6 where all features are queried). The explanation is that the IL-trees' performance is tightly connected to the number of features queried, as well as to their occurrence frequency. More specifically, sparse features with low occurrence will create very small trees, possibly of even one node only, while the tree of a more frequent feature will be bigger and will affect the method's performance substantially. In general, the IL-trees seem to work very well for a few features only, which are rarely met in the thematic map.

In the previous graphs, it is shown that with an increasing number of queried features, the performance of the IL-trees becomes worse and this tendency is apparent in the second set of experiments. In this second set of experiments, the methods' performance was tested for two fixed window sizes, where the number of the queried features was increasing.

As can be seen in Figure 14, the performance of the IL-trees decreases progressively with the number of queried features resulting in a quite bad performance when a large number of features is queried, whereas on the contrary the BHL- and BSL-tree show a comparatively stable behavior. As expected, on one hand SL- and BSL-trees, and on the other hand HL- and BHL-trees behave similarly when a large number of features is queried and that is why the graphs of the BSL- and BHL-tree only are shown.

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 thematic maps because they produce a greater number of locational keys and, therefore, the height of the B^+tree will be bigger. Thus, 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,
- thematic maps 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 technique has been introduced for use in quadtree-based access methods to improve previously proposed methods and efficiently process window queries in thematic maps with multiple non-overlapping features. A variation of the Linear quadtree has been presented and algorithms to process window queries have been described. The results of the conducted experiments concerning the performance of the new method when applied to the window queries have been discussed. It has been shown that this new method has a stable behavior in all cases, performing either the best or close to the best.

Future work may focus on a more efficient clustering of quadrants with similar features. This could be achieved either with the use of signature trees (S-trees) [3] or two-dimensional R-trees, where the first dimension would be dedicated for feature information and the second one would be the locational code.

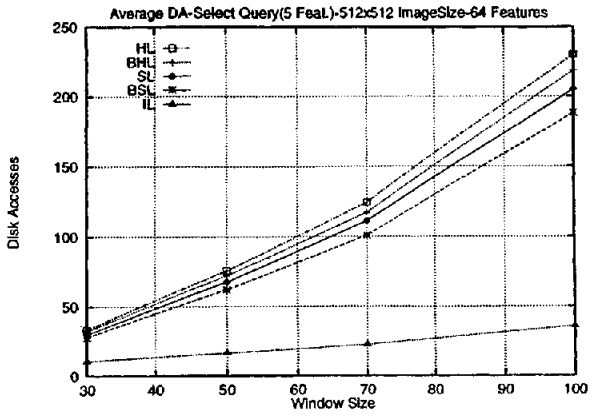

Figure 12: Averaged results for a select query where 5 features were queried, image size 512×512, 64 features.

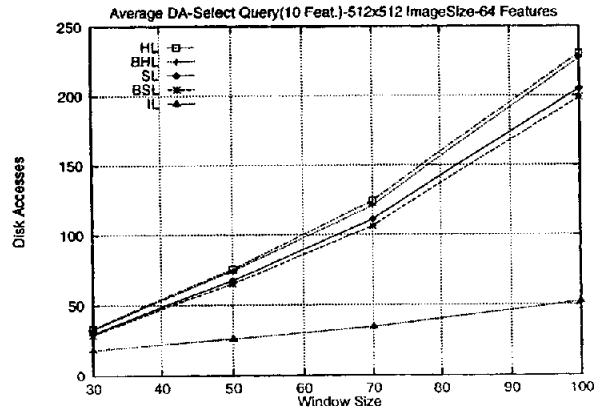

Figure 13: Averaged results for a select query where 10 features were queried, image size 512×512, 64 features.

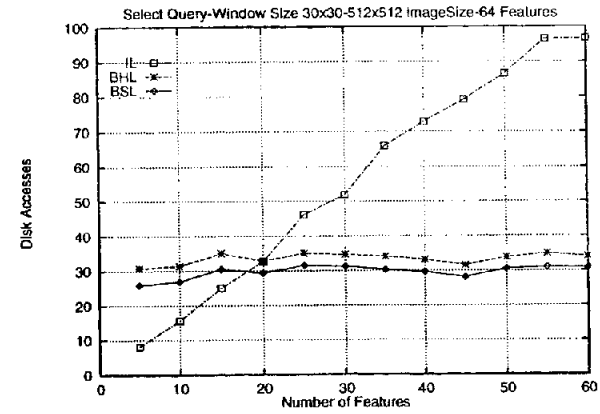

Figure 14: Select query for a varying number of queried features, image size 512×512, window size 30×30.

6. ACKNOWLEDGEMENTS

The authors would like to thank Mr. Alexandros Nanopoulos for his helpful comments and suggestions.

7. REFERENCES

[1] W. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *Proceedings 9th ACM PODS Conference*, pages 265–272, 1990.

[2] W. Aref and H. Samet. Decomposing a window into maximal quadtree blocks. *Acta Informatica*, 30:425–439, 1993.

[3] U. Deppisch. S-tree: a dynamic balanced signature index for office retrieval. In *Proceedings of the ACM SIGIR Conference*, pages 77–87, 1986.

[4] V. Gaede. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[5] I. Gargantini. An effective way to represent quadtree. *Communications of the ACM*, 25(12):905–910, 1982.

[6] O. Guenther. *Efficient Structures for Geometric Data Management.* LNCS 337, Springer Verlag, 1988.

[7] W. D. Jonge, P. Scheuermann, and A. Schijf. S$^+$trees: an efficient structure for the representation of large pictures. *Computer Vision,Graphics and Image Processing: Image Understanding*, 59(3):265–280, 1994.

[8] E. Kawaguchi, T. Endo, and M. Yokota. Depth-first expression viewed from digital picture processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 373–384, 1983.

[9] Y. Manolopoulos, E. Nardelli, A. Papadopoulos, and G. Proietti. MOF-tree: a spatial access method to manipulate multiple overlapping features. *Information Systems*, 22(8):465–481, 1997.

[10] E. Nardelli and G. Proietti. Efficient secondary memory processing of window queries on spatial data. *Information Sciences*, 80:1–17, 1994.

[11] E. Nardelli and G. Proietti. A hybrid pointerless representation of quadtrees for efficient processing of window queries. In *Proceedings International Workshop on Advanced Research in GIS (IGIS)*, 1994.

[12] E. Nardelli and G. Proietti. An optimal resolution sensitive pyramid representation for hierarchical memory models. *Journal of Computing and Information*, 1:385–402, 1994.

[13] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley Publishing Company, Reading, MA, 1990.

[14] S. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, 1975.

[15] M. Vassilakopoulos and Y. Manolopoulos. Analytical comparisons of two spatial data structures. *Information Systems*, 19(7):569–582, 1994.