ML-Quadtree: The Design of an Efficient Access Method for Spatial Database Systems

A. Touir
Department of Computer Science, College of Computer and Information Sciences
King Saud University, Riyadh, Saudi Arabia

Abstract. The aim of this paper is to present a new indexing technique that provides an efficient support for retrieving and handling spatial data. Traditionally, the mapping between layers (in a thematic point of view) and index structures is one to one. Each layer is associated with an index structure. In some previous work, we have presented a data structure, the FI-Quadtree that handles a set of images using only one index structure. This handling is a raster-oriented format. In this paper, we focus on the processing of these objects from the vector oriented format point of view. The Multi-Layer Quadtree (ML-Quadtree) is a new data structure that allows the storage and processing of several layers at the same time. This structure is based on the PM-Quadtree, which allows the storage of only a single-layer map. The aim of the ML-Quadtree is to be able to manage, store and perform queries among multiple layers simultaneously. The design and the manipulation of the proposed structure is presented in this paper whereas the implementation and the experimentation result will be treated in a subsequent paper.

1. Introduction

Spatial Information systems require an efficient spatial data handling [1, 2]. The large amount of information and the complexity of the characteristics of data have given rise to new problems of storage and manipulation. One of the important problems is how to store this kind of complex data for efficient search and retrieval operations. Convenient data organization for spatial databases is still a problem to be solved [3, 4].

Data structures for storing objects in a vector format should satisfy various characteristics. In fact, there is a trade-off between retrieval capabilities and storage or memory requirement, and this is an important issue for spatial data handling. The organization of spatial data objects requires the ability to cluster them together according

to their spatial location. The number of required disk accesses is usually used to measure the efficiency of the operations. The first approach adopted in most reported research on spatial access methods considered that free-form objects could be approximated by their minimum bounding rectangles [5] to simplify the complexity of the search. R-trees [6] and their extensions R+trees [7], R*trees [8], and other structures such as buddy tree [9] are examples of such structures. Another approach is to consider the object as it is without any approximation. In this case, the space is subdivided according to certain rules. A commonly used data structure that fits this approach is the quadtree [10,11]. An other approach is to use the SP-GiST index structure [12]. This latter aims at partitioning unbalanced trees where it can behave as a quadtree, or any of its variants. Another similar approach [13], the GL/GiST were proposed to deal with spatial index based on granular locking technique. Quadtrees provide an interesting technique to code images either in a raster format or in a vector format. In this paper we will be dealing with the vector format using the quadtree approach.

The quadtree is a hierarchical data structure used to organize an object space. An object can be a point, a line segment, a polyline, etc. This data structure has been widely used in computer vision, geographic information systems and geometric modelling [14]. Its main advantage is its compactness and regularity. Consider that we have a binary image; the principle of this structure consists in partitioning each object into homogeneous quadrants and labelling each of them. A homogeneous quadrant could be either white or black, and it is associated with a leaf (terminal) node of the quadtree. A non-homogeneous quadrant is considered as a grey quadrant and it is associated with a non-terminal node of the quadtree. Recursive subdivision is applied to the binary image: a quadrant is subdivided into four equal parts until a homogeneous quadrant or pixel is reached. The Morton order [15] can be used to organize and sort the squares that aggregate the space. Fig.1 shows how squares are ordered and labelled.
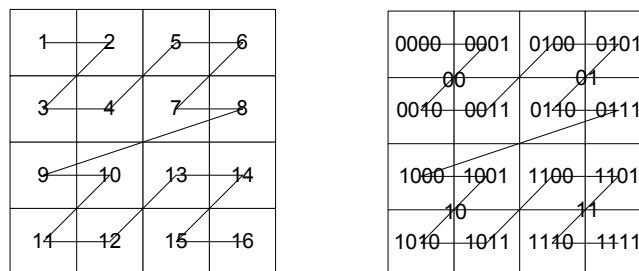

Fig. 1. Ordering squares using Morton order.

The Morton code of a node [16] is built by interleaving the bits of x and y co-ordinates of the upper left corner of the quadrant that corresponds to the node. The label of a quadrant, which we call a prefix, may have several representations. In this paper, we

suppose that a prefix can be defined either by its binary label, or by its length (number of bits that compose it) and its decimal value. The length of each binary label depends on the image definition. Within a 2^N x 2^N bitmap image, the maximum length is 2N.

Some varieties of quadtrees have been proposed. Each one is more or less adapted to manipulate a specific type of data. Linear quadtrees are used to code and store black quadrants. The PM-Quadtree [11] represents line-segments data. Section 2.1 explains this structure in more detail. Hjaltason improved the bulk-loading PMR quadtrees [17,18]. Its principle consists in assuming that the quadtree is implemented using a linear quadtree, a disk-resident representation that stores objects contained in the leaf nodes of the quadtree is in a linear index ordered on the basis of a space-filling curve as shown above. The PR-Quadtree [19] is used to code point and region data. Another kind of quadtree is the MX-CIF-Quadtree [20]. It is used to represent data of type rectangle. The principle of the MX-CIF-Quadtree is to associate with each rectangle, the quadtree node corresponding to the smallest quadrant that contains it. The decomposition of quadrants is recursively carried out until no quadrant contains any rectangle. The last two types of quadtrees accept more than one data in a node. Moreover, using the MX-CIF-Quadtree, data can be in any node (root, terminal or non-terminal nodes), whereas the use of the other kind of quadtrees allows data to be stored only in leaves (terminal nodes).

All the presented spatial structures allow the storage and indexing of a single object whatever this object is: a map, an image, etc. One quadtree is built for one object. On the contrary, the FI-Quadtree [21,22] allows the storage of a set of images into a single quadtree without any substantial difference from the complexity viewpoint. The DI-Quadtree [23] is an improved version of the FI-Quadtree. Its main differences with the FI-Quadtree reside in the labeling order of the nodes and in the storage mechanism. Given the number of layers, the FI-Quadtree computes in advance the necessary space needed to store them. If this number should be modified, the FI-Quadtree is reorganized to create more rooms for those additional layers. On the contrary, the DI-Quadtree is defined independently of the number of layers and it is used as a front structure. Both structures are raster-oriented. The MOF-tree [24] is also a raster–oriented structure. Its aim is to support images with multiple overlapping features. Its principle consists in recursive decomposition of the image into four equal-sized quadrants until each quadrant is fully covered by features covering it. Same is the case [25] with Multiversion Linear Quadtree, a spatio-temporal access method based on Multiversion B-trees [26]. The structure may be used as an index mechanism for storing and accessing evolving raster images.

Unfortunately, all these quadtrees are not suitable when a given object, say a map, is composed of several layers where layers are in vector-oriented format. Layers are used for the purpose of thematic approach. Layers may represent districts, parcels, or a network of roads, rivers, etc., one layer per theme. Several index structures are used to

process queries like: *"Retrieve all the water pipes and the electrical cables situated in a given region"*. The use of a single index structure to perform such query is an attractive track. In this paper, we focus on this type of problem and show that our proposed data structure is suitable for this kind of queries. The use of the Multi-Layer-Quadtree (ML-Quadtree or MLQ for short) considerably reduces the complexity of the process of this type of queries in terms of I/O, time consumption and storage. Its flexibility resides in the fact that only one index structure is used to manage several layers. Another application of this structure is to manage multiple versions or temporal evolution of an object in a spatial databases context.

The paper is organized as follows. In Section 2, we review the definition of the PM-Quadtree and introduce the Multi-Layer-Quadtree: its definition, characteristics, and properties. This section also deals with the proposed index structure. Section 3 gives some details of the multiple layers search within the proposed structure; it introduces the principle of the Insert and Delete operations and discusses the capabilities provided by the ML-Quadtree in querying several layers. Finally, conclusions are given in Section 4.

2. The Multi-Layer-Quadtree Data Structure

2.1 Principles of the PM-Quadtree

The PM-Quadtree (PMQ) [11] represents objects of polygonal shapes. The representation of objects is neither approximated nor based on digitization. Different kinds of arithmetic and geometric operations could be performed using this type of structure without any distortions of the objects. This is in contrast with the bitmap/raster representation, where zooming in on, or rotating, an object may change its original shape. In addition, the PMQ allows each entity of the stored object to have a semantic meaning such as lake, hotel, road, etc.

Different kinds of PM-Quadtrees are used. Each of these PMQs consists in subdividing each region into four equal-sized quadrants until we obtain quadrants that satisfy the following rules [12]:

PM1-Quadtree (Fig.2.a): At most, one polygon vertex can lie in a region represented by a quadtree leaf node. If a quadtree leaf node region contains a vertex, then any edge of this region must contain that vertex. If a quadtree leaf node region does not contain vertices, it can contain at most one edge. Each region quadtree leaf node is maximal and contains at most one vertex.

PM2-Quadtree (Fig.2.b): It differs from PM1-Quadtree in that a region is decomposed into four equal-sized quadrants as long as a quadrant contains more than one line segment unless the line segments are all incident to the same vertex.

PM3-Quadtree (Fig.2.c): The decomposition depends only on the vertices. A region is decomposed into four equal-sized quadrants as long as it contains more than one vertex.

Bucket PM-Quadtree (Fig.2.d): Recursively decomposes a region into four equal-sized quadrants as long as a quadrant contains more than BC line segments. BC is called the bucket capacity of the PM-Quadtree. Table 1 reflects the prefixes and the data generated from each kind of PM-Quadtree shown in Fig. 2.
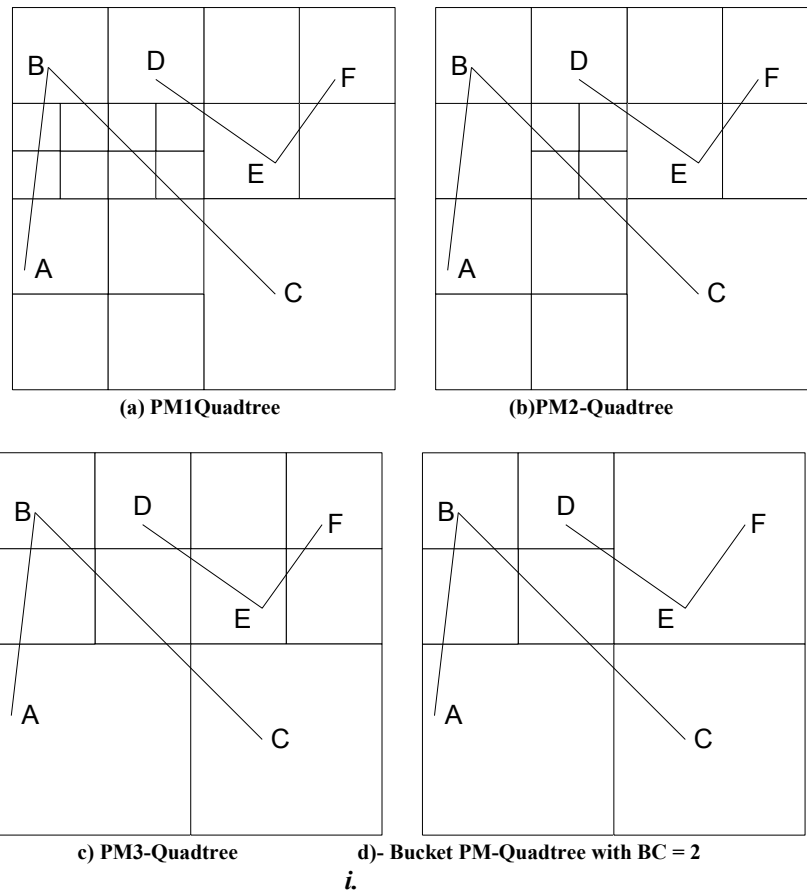

(a) PM1Quadtree   (b)PM2-Quadtree

c) PM3-Quadtree   d)- Bucket PM-Quadtree with BC = 2

i.

Fig. 2. Principle of subdividing the space using different PM-Quadtrees.

Table 1. List of prefixes and their related data generated from the example of Fig. 2 according to the type of PMQ used

Quadtree type	List of prefixes and their related data
PM1Q	{(0000, B),(0001, D),(001000, AB),(001001, BC),(001010, AB),(001100, BC),(001101, DE), (001110, BC),(001111, BC),(0101, F),(0110, E),(0111, EF),(1000, A),(1001, BC),(11, C)}
PM2Q	{(0000, B),(0001, D),(0010, [AB, BC]),(001100, BC),(001101, DE), (001110, BC),(001111, BC),(0101, F),(0110, E),(0111, EF), (1000, A),(1001, BC),(11, C)}
PM3Q	{(0000, B),(0001, D),(0010, [AB, BC]),(0011, [BC,DE]) ,(0101, F),(0110, E), (0111, EF), (10,A),(11,C)}
Bucket-PMQ	{(0000, B),(0001, D),(0010, [AB, BC]),(0011, [BC,DE]),(01, [E,F]), (10,A), (11,C)}

As we can notice, with the same data, a generated PMQ is more or less complex depending on its definition (i.e. the number of items that could be contained in a quadrant). PM1Q generates much more nodes than the other PMQs. A PM1Q leaf node contains no more than one item (a vertex or a line segment). On the contrary, the other PM-Quadtrees allow a leaf node to contain several items and, so, they need additional processing.

2.2 Basic Idea

We consider a layer as a set of spatial objects and a map as a set of layers. A spatial object may be part of several layers. In addition, queries may trigger the processing of several layers, in search for the requested spatial objects. This multi-layer processing is a complex operation. This complexity is due to the fact that the data set of the spatial objects is very large. It is, therefore, crucial to support indexing techniques to facilitate the retrieval of any portion of an object. In order to overcome this problem, we present a Multi-Layer-Quadtree that allows for the processing of components of different layers in the same structure. This processing could be considered as a multiple join operations or a multiple criteria selection. We define MLQ as being based on the PM1Quadtree. This choice of PM1Quadtree is due to the fact that the number of nodes of the MLQ grows with respect to the inserted layers. So, nodes that are not generated from one layer may be generated from others. As indicated, earlier, all PMQs but PM1Q need additional processing to determine the right vertices; for this reason, we have chosen PM1Q as the basic structure for the MLQ to avoid the extra processing. Applying the MLQ based on the other PMQs may be presented as future work.

2.3 The Multi-Layer-Quadtree: Definition and Characteristics

Similar to the DI-Quadtree, the ML-Quadtree is defined as a pure index structure, where each node is a pointer to a structure that contains the data belonging to that node (Fig. 3). The maximum number of nodes that constitute an MLQ is $\left(\frac{4^{N+1}-1}{3}\right)$ nodes. For N = 10 we have a total of 1398101 nodes. The fact that the MLQ is used as a pure index structure allows us to manipulate it as a main memory-oriented structure. Each node addresses a rear structure defined as follows:

- Each rear structure is composed of a set of components $\{c_1, \ldots, c_k\}$.
- Each component c_i is defined as being a triplet (layer$_i$, object$_j$, element$_k$), where element$_k$ (e.g. a line) belongs to object$_j$ (e.g. a polyline), which in turn belongs to layer$_i$.
- As it is described in [23], after several insertions in a given rear structure (say RS), the latter may be full; in which case a new rear structure is created and logically linked to RS.
- Figure 3 shows how a new layer is inserted and handled with the ML-Quadtree and its rear structure.
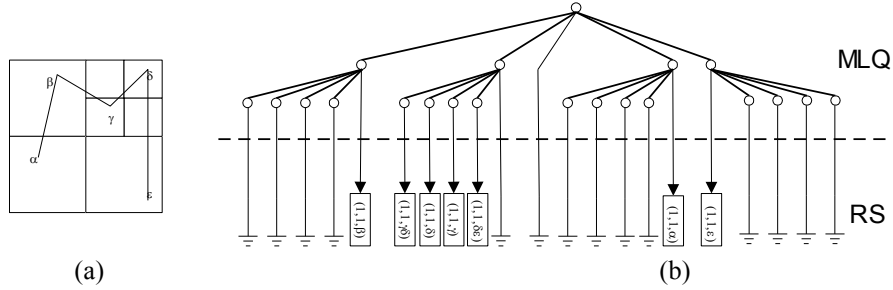

Fig. 3. Inserting a first layer in the ML-Quadtree.

Table 2. List of symbols and definitions

Symbol	Definition
N	Number of decompositions of the ML-Quadtree
Q_N	Workspace of size $2^N \times 2^N$, consisting of all quadrants of a $2^N \times 2^N$ image
P_N	The set of prefixes associated with Q_N
W_N	A set of integers defined as: $W_N = \{w / \exists p \in P_N, w = H_N(p)\}$
H_N	A mapping function between P_N and W_N
q	A quadrant of Q_N
p	An element of P_N associated with a quadrant q
L_p	The number of bits that compose p
V_p	The decimal value of p
b	Bit value

w	A node value. w belongs to W_N
L_{max}	The maximum number of layers in the MLQ
S_{cap}	The maximum number of components that can fit in a segment of the rear structure

We denote $nd=(w, \{c_1, \ldots, c_k\})$ a node of an ML-Quadtree, where w is a node label and $\{c_1, \ldots, c_k\}$ is the set of components indexed from that node. A triplet (L_i, O_j, E_k) defines a component c_i, where E_k is the element of the object O_j belonging to the layer L_i.

An ML-Quadtree is a quaternary tree that satisfies the following:

i) Data of an ML-Quadtree is indexed from any node. Three components c_1, c_2, c_3 could be distributed on three nodes nd_1, nd_2, nd_3 of three different levels, where $nd_1=(w_1,\{c_1\})$ is a parent of $nd_2=(w_2,\{c_2\})$, which is in turn a parent of $nd_3=(w_3,\{c_3\})$.

ii) if $(w_1,\{c_1, \ldots, c_n\})$ and $(w_2, \{d_1, \ldots, d_m\})$ are in the same path, c_i and d_j could not belong to the same layer, for any i= 1, ..., n and j = 1, ..., m; i.e two components of the same layer must not be in the same path unless they are in the same node.

iii) The Morton code is used to label the upper left corner of the quadrant that corresponds to a node. In this paper, we suppose that a prefix can be defined either by its binary representation p, or by its length L_p (number of bits that compose it) and its decimal value

$$V_p = \sum_{i=1}^{L_p} b_i * 2^{(Lp-i)}$$. Thus for any prefix $p = b_1b_2b_3\ldots b_{Lp}= (L_p, V_p)$,

where $b_i \in \{0,1\}$.

In the sequel and by notation abuse, we use any of the two representation (p or (L_p,V_p)).

As defined in (Touir, 1991),

$$H_N(p) = \frac{L_p}{2} + \sum_{i=1}^{L_p} b_i * \Omega_i ,$$

where $\Omega = \begin{cases} \dfrac{4^{(N-\frac{i}{2}+1)} - 1}{3} & \textit{if i is even} \\ 2*\Omega_{i+1} & \textit{if i is odd} \end{cases}$.

This function offers a total order of labeling between P_N and W_N. It reflects a pre-order traversal of the ML-Quadtree (Fig. 4).

Note that H_N is a 1-1 onto mapping. Thus, it could be used to index nodes in a unique way.

Suppose that we have already inserted a layer (say X) represented in Fig. 3, and we would like to insert a new one (say Y) represented in Fig. 5. By applying the above defined rules (i-iv), we obtain a virtual layer (say Z) Z=X⊕Y (Fig. 6). Z is the superimposition of X and Y.
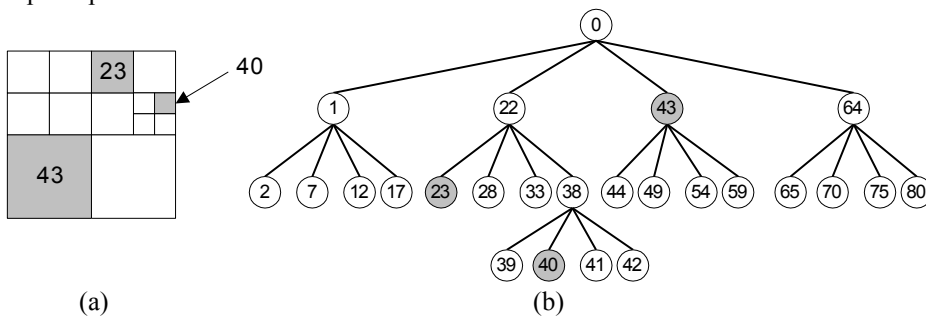

(a)   (b)

Fig.4. The labelling of quadrants in a quadtree (decimal values-pre-order traversal).

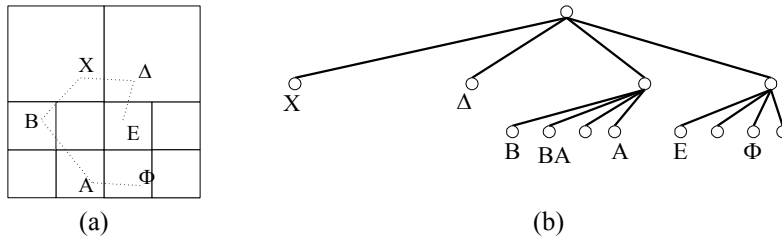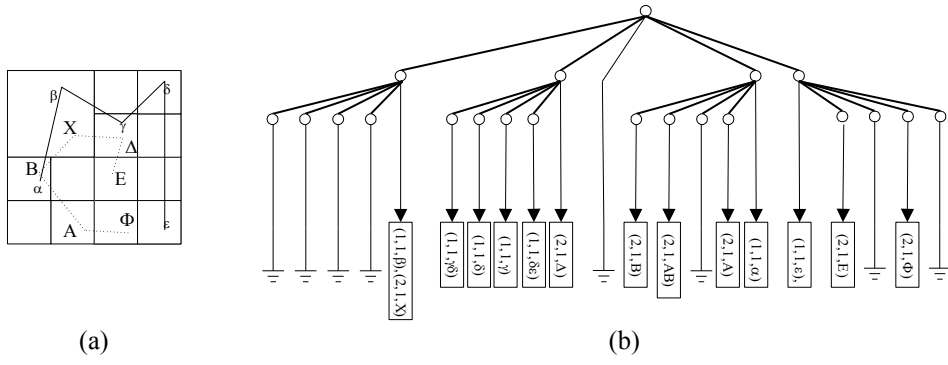(a)   (b)

Fig. 5. Building a second layer component.

By applying (i), we see that Z has components distributed on many levels: $(2,1,\Delta)$ is a component in a node of level 1. This node is a parent of $\{(1,1,\gamma\delta), (1,1,\delta), (1,1,\gamma), (1,1,\delta\varepsilon)\}$.

By applying (ii), we see that if a component of X (resp. Y) is in a given node, all its sub-nodes (children) do not contain any component of X (resp. Y).

A. Touir

(a)　　　　　　　　　　　　　　　(b)

Fig. 6. How the ML-Quadtree changes when a new layer is inserted.

2.4 Indexing Technique

As it was mentioned, the maximum number of nodes of an MLQ is $\left(\frac{4^{N+1}-1}{3}\right)$ nodes. We intend to use the MLQ as a pure main memory oriented index structure. For this reason, we choose an index whose size is small enough to give the needed information and to keep the MLQ reasonably small so that it can fit in the main memory. As an illustration, consider the MLQ in Figure 4.a, where N=3, q_1, q_2 and q_3 three quadrant regions, then their corresponding prefixes are respectively p_1=10=(2,2), p_2=0100=(4,4) and p3=011101=(6,29); Using H_N and Ω_i where i=1...6, we obtain: w_1= H_N (p_1)= 1+Ω_1= 43, w_2 = H_N (p_2)= 2+Ω_2= 23 and w_3 = H_N (p_3) = 3+Ω_2+Ω_3+Ω_4+Ω_6= 40.

By the definition of w and p, it can be seen that the size of w is smaller than the size of p. Consequently using w as an index will reduce considerably the size of MLQ. To use w as an index, we need to introduce the notion of H_N^{-1}, which is the inverse of H_N. We denote H_N^{-1} with G_N (i.e. G_N=H_N^{-1}). The intention is to determine $p \in P_N$ for every w in W_N. That is $\forall w \in W_N$, $\exists p \in P_N$ / $p=G_N(w)$. With this in mind, we have developed the algorithm below to compute $p \in P_N$ / $p= G_N(w)$ for each given $w \in W_N$:

G_N:

```
input : w a value that belongs to W_N, N
output : p a value that belongs to P_N
length_of_p = 0
value_of_p = 0
for i=2N downto 1 do
        if w> Ω_2N-i+1 then
                value_of_p = value_of_p<<1
                X =w- Ω_2N-i+1
                if (2N-i+1> 2X) then
                        value_of_p = value_of_p+1
                        w=X
                endif
        endif
        else if value_of_p >0 then value_of_p = value_of_p<<1
endfor
length_of_p = (w<<1)
value_of_p = value_of_p >>(2N-(length_of_p))
```

Where the expression "a<<1" allows to shift 1-bit to the left all bit-values of a.

3. Manipulation of Multi-Layer-Quadtrees

This section discusses the fundamental operations on the MLQ structure namely: the insertion, selection and deletion.

3.1 The Insertion

The following algorithm details the steps of the insertion operation. It should be noted that the conditions i-ii laid out in section 2.3 hold. Thus the algorithm makes sure that the stated conditions are met for any inserted object. This is can be deduced from the bold statement in the proposed algorithm below.

Insert:

> *input: component to be inserted;*
> > *output: the updated ML Quadtree*
>
> *Clipping the component to be inserted against the squares corresponding to the nodes. We process this operation to avoid looking at areas where the component is not to be inserted.*
> > *if the result of the clipping is null then we have nothing to insert*
> > *else we have new components (say NC) resulting from the clipping against the current square*

* if the current node has some data C that belongs to the same layer as NC
> **Merge C and NC**
> **Compute newComp = C$\cup$NC**
> **if newComp does not verify PM Quadtree rules**
> > **if the current node is a leaf**
> > > **split it into four nodes**
> > > **recall insert procedure for each new node to insert NewComp**
> >
> > **else recall insert procedure for each Son of the current node to insert NewComp**
>
> **else   {if newComp verifies PM Quadtree rules against the current node}**
> > **memorize that node say ND**
> > **go down in depth to look for data that belong to the same layer as NC**
> > **if data (say C) is found goto ***
> > **else insert NC in ND**

> end.

The resulting MLQ in Fig. 6 as insertion of a second layer in the MLQ and illustrated the working of the above algorithm, whereas Fig. 3 shows the insertion of a first layer.

Remarks:
i.   The first part of this algorithm is similar to the insert algorithm of a PMQ. The statement "go down in depth…" takes into account more complex cases. Indeed, after locating the right place, (say N) where a new component will be inserted, non-empty nodes rooted at N will be visited until leaf nodes are reached, or data

from the same layer is met. This supplementary analysis is due to the lack of information about layers stored in each node. In the example of Fig. 5, ϕ is inserted in the second level. In the proposed algorithm, ϕ will be stored in a specific node (say ND). But before inserting it, we have to go down in depth to check if there is some data of the same layer other than ϕ. In this example the result is an empty set and leaf nodes are reached. At this point the insertion is performed. This step ensures that no data of the same layer exists in two different levels of the same path.

ii.   In fact, the proposed algorithm is too complex and it generates several unneeded I/O. One of the implementations that we carried out concerning the insertion, to overcome the I/O complexity, is: first build the PMQ of the layer to be inserted; second update the MLQ according to the result of the obtained PMQ. This method avoids testing each time the "go down in depth …" but it adds the copying process of the PMQ to the MLQ.

3.2 The Selection

Queries can be of two types: point-based queries and region/window-based queries. The MLQ is flexible enough to carry out those types of queries. In addition, those queries can be performed on:

a.   objects of a specified layer such as: *select all the objects of layer X that are located inside a region R*

b.   objects of several layers such as: : *select all the objects of layers X_1, X_2,…, X_n that are located inside a region R*

c.   objects of specific layer with more complex condition: *select all the objects of layer X that are located inside a region R and intersect objects of layers X_1 and/or X_2.*

We can summarise those cases in the following :

Select <list of objects>
From <list of layers>
Where <list of conditions>

This type of queries are processed in two steps:

1)   Select the set of candidate objects using the select, from clauses.
2)   Perform the user-conditions on those candidates using the where clause.

The following algorithm shows how to search an object according to a given component. It mainly focuses on step 1.

Select Component:
  input : ML Quadtree ; RL : a list of layers' information (say names) to which the selected components belong ; location ;
  output: list LC of components, their owner objects and their layers ;

if location intersects the square of the current node
 if the current node is not a leaf
  fetch components of the desired layers
  for each found component
   remove its layer name from RL
   add the found triplet to LC
  if RL is empty return
  else for each son of the current node recall SelectComponent with RL and the
  current location
 else fetch components of the remaining layers RL and add them to LC
end.

To illustrate the working of the above algorithm, suppose that we would like to select all layers that contain objects that intersect the object that contains the vertex (γ). As a first step, all objects that are in the search path are collected, which are in this case (2, 1, Δ) and (1, 1, γ). Then a second process on those objects is performed (to satisfy the user's criteria). This means that the intersection of object 1 of layer 2 and object 1 of layer 1 is computed (Fig. 7). The result is the empty set. The reader can imagine the wide range of queries that can be performed using this structure without being obliged to use as many index structures as the involved layers.
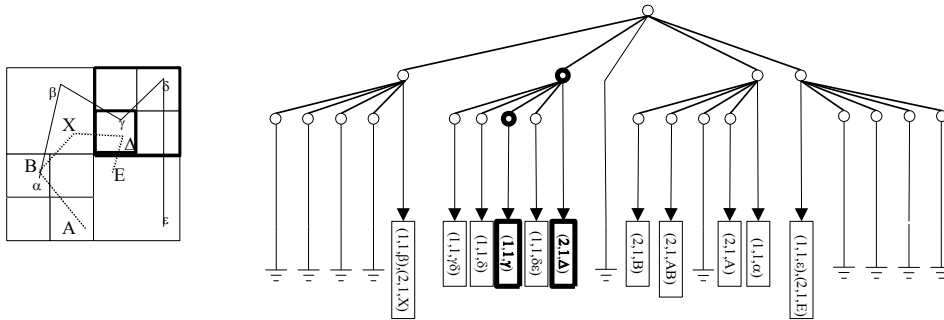

Fig. 7. Selection of objects that are candidate to intersect the object containing γ.

3.3 The Deletion

   The deletion offers the same flexibility as shown in the previous section, when we would like to delete objects that belong to one layer or more. The deletion may be point-based or region/window-based queries. Cases a, b and c of section 3.2 are also valid. We can summarise this in the following form:
   Delete
   From <list of layers>

Where <list of conditions>

The following algorithm describes the main tasks of this operation:

Delete:

input : ML-Quadtree; RL :a list of layers names from which these components will be

deleted; location and list of conditions;

output: Multi Layers Quadtree

if location intersects the square of the current node

if the current node is not a leaf

fetch components of the desired layers RL

for each found component C of a given layer do

if the user conditions are verified delete C

remove its layer name from RL

try Moving data one level up

if RL is empty return

else for each son of the current node recall Delete with RL and the current location

else fetch components of the remaining layers RL

if the user conditions are verified delete it

check the possibility of moving data one level up

end.

Suppose that we would like to delete the line (AΦ) from the second layer (Fig. 6). The resulting data is shown in Figure 8. We notice that two important actions are performed:

a.   (2,1,E) is moved one level up (from level 2 to level 1) where (1,1,ε) resides.

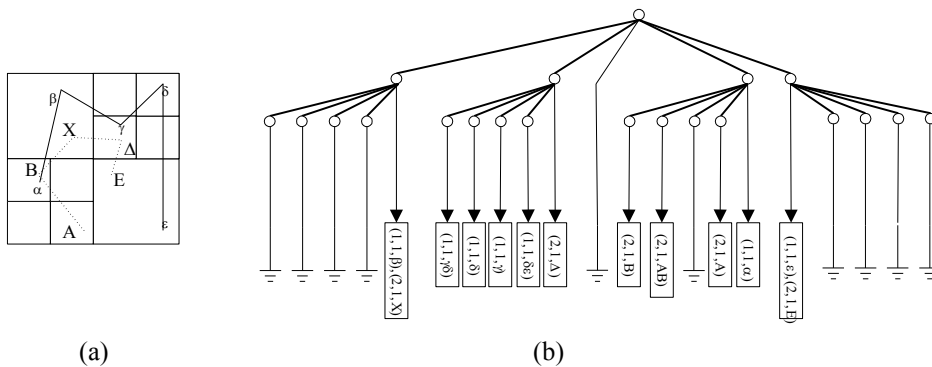b.   Its new location has changed from a node to a leaf node.

(a)   (b)

Fig. 8. Deletion of the line (AΦ) from the Multi Layer Quadtree.

The deletion of line ($\delta\varepsilon$) (Fig. 9), generates the update of two nodes.
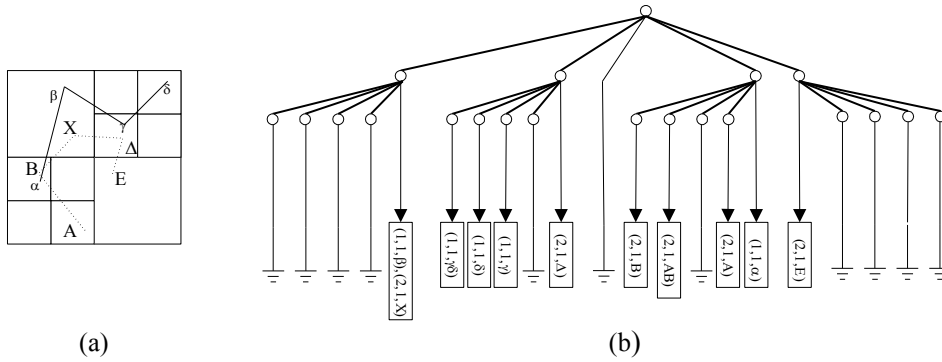

(a)   (b)

Fig. 9. Deletion of the line ($\delta\varepsilon$) from the Multi Layer Quadtree.

4. Conclusion

In this paper we have proposed and analyzed a new Quadtree-based data structure: the ML-Quadtree. This structure allows the simultaneous manipulation of several Layers with the same index structure. The manipulation could be of a different level of complexity. We have investigated different types of manipulations (insertion, deletion, searching) within this structure, and we have shown that it is well adapted for multi-criteria retrieval. We have analyzed this structure with respect to several insertions of layers of different types and sizes. In a forthcoming work, some tracks are to be taken into account:

The investigation of the parallel processing of such structure, where the behavior of this spatial access method will be analyzed.

- The use of such a structure in the multi-version maps and map-history contexts.
- Labelling principle and the use of B-trees as carrier of such a structure are also tracks to be investigated.
- Evaluation of the MLQ based on the other type of PMQ and comparing both the complexity and flexibility of the operations using the generated MLQ.

References

[1] Chang, S. K., Jungert, E. and Li, Y. "The Design of Pictorial Database Based Upon the Theory of Symbolic Projections". In: *Proceedings 1st International Symposium on Large Spatial Databases*, Santa Barbara, USA (1989), 303-323.

[2] Nardelli, E. and Projetti, G. "Time and Space Efficient Secondary Memory Representation of Quadtrees". In: *Information Systems*, 22, No. 1 (1997), 25-37.

[3] Erwig, M. and Guting, R. H. "Explicit Graphs in a Functional Model for Spatial Databases". In: *IEEE Transaction on Knowledge and Data Engineering*, 6, No. 5 (1994), 787-804.

[4] Kriegel, H. P., Fahldiek, A. and Mysliwitz, N. "Query Processing of Geometric Objects with Free Form Boundaries in Spatial Databases". In: *Proceedings DEXA'93, International Conference on Database and Expert Systems Applications*, Prague (1993), 349-360.

[5] Brinkhoff, T., Kriegel, H.P. and Seeger, B. "Efficient Processing of Spatial Join Using R-Trees". In: *Proceedings ACM SIGMOD'93 International Conference on Management,* (1993), 237-246.

[6] Guttman, A. "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *Proceedings ACM SIGMOD* (1984), 47-57.

[7] Sellis, T., Roussopoulous, N. and Faloutos, C. "The R+Tree: A Dynamic Index for Multidimensional Objects". In: *Proceedings 13th International Conference on Very Large Database,* (1987), 507-518.

[8] Beckmann, N., Kriegel, H.P., Schneider, R. and Seeger, B. "The R*Tree: An Efficient and Robust Access Method for Points and Rectangles". In: *Proceedings ACM SIGMOD'90 International Conference on Management of Data,* (1990), 322-331.

[9] Seeger, B. and Kriegel, H. P. "The Buddy-tree: An Efficient End Robust Access Method for Spatial Database Systems". In: *Proceedings 16th International Conference on Very Large Database.* (1990), 590-601.

[10] Samet, H. "The Quadtree and Related Hierarchical Data Structures". In: *ACM Computing Surveys*, 16, No, 2 (1984), 187-260.

[11] Samet, H. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[12] Aref, W. G. and Ilyas, I. F. "An Extensible Index for Spatial Databases". In: *Proceedings the 13th International Conference on Statistical and Scientific Databases*, Virginia. (2001), 49-58.

[13] Chakrabarti, K. and Mehrotra, S. "Efficient Concurrency Control in Multidimensional Access Methods". In: *Proceedings ACM SIGMOD*, Philadephia, Pennsylvania,USA (1999), 25–36.

[14] Samet, H. *Applications of Spatial Data Structures.* Addison-Wesley, 1990.

[15] Laurini, R. "Graphics Databases Built on Peano Space-filling Curves". *Proceedings of the Eurographics Conference*, Nice (1985), 327-338.

[16] Morton, G. M. "A Computer Oriented Geodetic Database and a New Technique in File Sequencing". IBM Ltd., Ottawa, Canada (1966).

[17] Hjaltason, G. R. and Samet, H. "Improved Bulk-loading Algorithms for Quadtrees". In: *Proceedings, 7th International Symposium on GIS (ACM GIS '99)*, Kansas City, MO. (1999), 110-115.

[18] Hjaltason, G. R. and Samet, H. "Speeding up Construction of Quadtrees for Spatial Indexing". *Comp. Sci. Dep. TR-4033*, Univ. of Maryland, College Park, MD (1999).

[19] Ang, C. H. and Samet, H. "Node Distribution in a PR Quadtree". In: *Proceedings 1st International Symposium on Large Spatial Databases*, Santa Barbara, USA. (1989), 233-252.

[20] Kedem, G. "The Quadtree-CIF Tree: A Data Structure for Hierarchical On-line Algorithms". In: *Proceedings 19th Design Automation Conference*, Las Vegas, USA (1983), 352-357

[21] Cheiney, J. P. and Touir, A. " FI Quadtree: A New Data Structure for Content Oriented Retrieval and Fuzzy Search". In: *Proceedings* SSD, *2nd International Symposium on Advances in Spatial Database*, Zurich, Switzerland (1991), 23-32.

[22] Touir, A. "The Design of an Eficient Data Structure in an Image Database System". In: *Proceedings DEXA International Conference on Database and Expert Systems Applications*, Berlin, Germany (1991.), 191-196.

[23] Vassilakopoulos, M. and Manolopoulos, Y. "Dynamic Inverted Quadtree: A Structure for Pictoral Databases". In: *Information Systems*, 20, No. 6 (1995), 483-500.

[24] Manolopoulos, Y. and Nardelli, E. "MOF-TREE: A Spatial Access Method to Manipulate Multiple Overlapping Features". *Information Systems,* 22, No. 8 (1997), 456-481.

[25] Tzouramanis, T., Vassilakopoulos, M. and Manolopoulos, Y. "Multiversion Linear Quadtree for Spatio-temporal Data". Stuller, J. *et al.* (Eds.): ADBIS-DASFAA 2000, LNCS 1884 (2001), 279-292.

[26] Becker, B., Gschwind, S., Ohler, T., Seeger B. and Widmayer, P. " An Asymptotically Optimal Multiversion B-Tree". *The VLDB Journal*, 5, No.4 (1996), 264-275.

الشجرة الرباعية المتعددة الطبقات (ML-Quadtree): تصميم هيكل فهرسي فاعل متعدد الطبقات لنظم المعلومات الفضائية

عامر الطوير

قسم علوم الحاسب، كلية علوم الحاسب والمعلومات

جامعة الملك سعود، الرياض

ملخص البحث. يتناول هذا البحث تقديم تقنية فهرسية جديدة لمساندة العملية الاستعلامية والتحليلية للمعلومات الفضائية. إن تركيبة الخريطة الجغرافية هي عبارة عن عدة طبقات، كل طبقة تمثل نوعاً معيناً من المعلومات مثل طبقة الأنهار وطبقة الطرق ... الخ. كما أن الطريقة المتعارف عليها والمتعامل بها لمعالجة الطبقات هي إنشاء هيكل فهرسي لكل طبقة، يتم من خلالها الوصول إلى بيانات تلك الطبقة وتنفيذ عملية المعالجة التي يريدها المستخدم. فلو أردنا أن نعرف مثلاً ما هي نقاط التقاطع بين شبكة المياه وشبكة الكهرباء في منطقة معينة فيجب استخدام الهيكل الفهرسي التابع لطبقة شبكة المياه والهيكل الفهرسي التابع لطبقة شبكة الكهرباء. وتعتبر هذه الطريقة مكلفة جداً مما يجعل من الصعوبة على أنظمة المعلومات الجغرافية توفير هذه الخدمة. إننا ومن خلال هذا البحث نقترح تقنية فهرسية جديدة نستطيع من خلالها تنفيذ أي استعلام أو معالجة لعدة طبقات باستخدام هيكل فهرسي موحد. نستعرض من خلال هذه الورقة تعريف وتوصيف هذا الهيكل وكيفية إنشائه وتعديله والاستعلام من خلاله.