Quadtree representations for storage and manipulation of clusters of images

Maude Manouvrier, Marta Rukoz, Geneviève Jomier

Quadtree representations for storage and manipulation of clusters of images ⋆

Maude Manouvrier [a] , Marta Rukoz [b] , Geneviève Jomier [a]

[a] *LAMSADE, Paris Dauphine University, Place du Mar. de Lattre de Tassigny, 75775 Paris Cedex 16 France, e-mail: {jomier,manouvrier}@lamsade.dauphine.fr*

[b] *Universidad Central de Venezuela, Centro de Computación Paralela y Distribuida, Escuela de Computación, Ciudad Universitaria, Av. Los Ilustres, 1041 Caracas, e-mail: mrukoz@kuaimare.ciens.ucv.ve*

Abstract

This article gives an overview of different approaches proposed for the storage and the manipulation of clusters of images. Clustering images consists in grouping together images having a defined relationship. In this article, images are represented by quadtrees implemented in a hierarchical or in a linear way. The discussion, presented at the end of the article, allows selecting a quadtree-based representation well-adapted to a specific area of applications or to the characteristics of the manipulated images.

Key words: hierarchical and linear quadtree, similarity of quadtrees, operations on image quadtrees, classification of approaches

1 Introduction

The design of efficient image representations that reduce the storage space of images and the time required for image manipulations has been widely studied [1,2,11,19,30,37,42,43]. The quadtree [9,30] is one of the most used structures for image representation. This structure is efficient to store 2D images and has been frequently used in the field of computer graphics [35], content-based image retrieval [21,22], image processing [19,33,43], Geographical Information Systems (GIS) [2,15,28,34] and image databases [13,38,41].

Clustering images means grouping them together according to some of their features. For instance, images may be clustered according to their semantics, their physical characteristics (e.g. color or texture) or their history. In the applications mentioned above, it is useful to cluster images having similarities. In [36], images with the same semantics belong to the same cluster in a GIS. The clusters, in that case, preserve the similarity in semantics, creating, for example, water, grass and agriculture clusters of images. Without any semantic clustering, irrelevant images may be returned by a query, because semantically not similar images may have a very small distance between their feature vectors (i.e., the numerical representations of the image features, e.g. wavelet transforms [16]). In the image processing application presented in [13], images are generated as the result of an image processing operation or a sequence of operations applied on an initial medical image. The purpose is to make some image elements appear more clearly, to show some characteristics or salient features or to emphasize differences. Grouping the result images in a cluster allows to analyze the modifications done by the image processing.

⋆ This work was supported by the CNRS in France and by CONICIT (accord numbers 8680 and 10058) and CDCH in Venezuela.

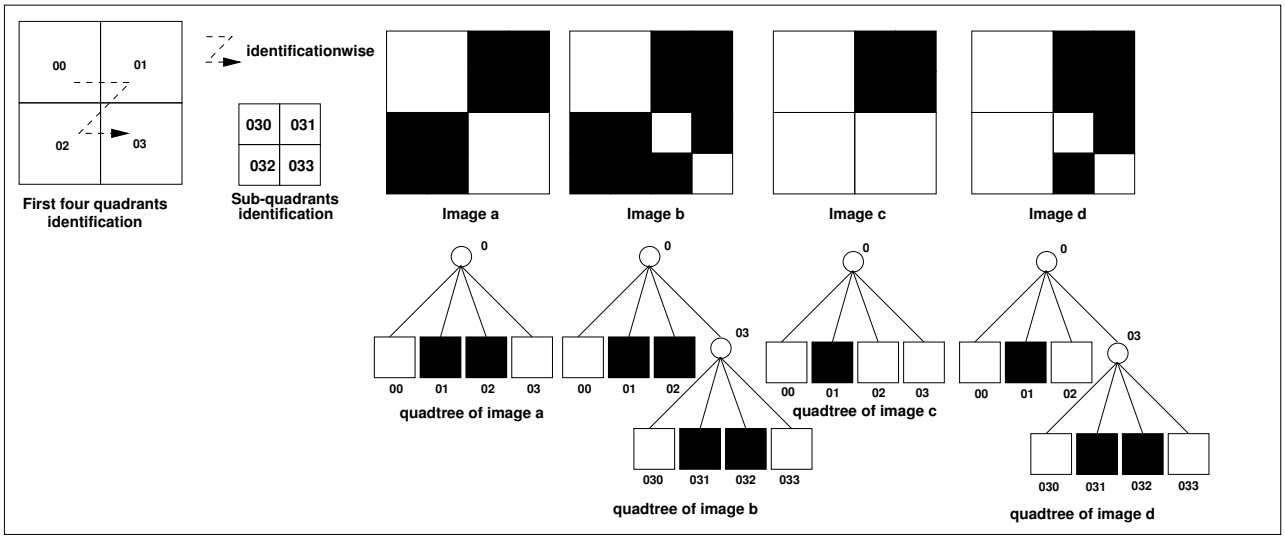

Fig. 1. Four binary images and their quadtree representation.

In this article, a *cluster of images* is defined as a data structure used to store a set of related images. Different approaches based on quadtrees and proposed for the representation and the manipulation of clusters of images have been described [7,13,18,38,40,41]. The main contribution of this article is a qualitative comparison of these approaches according to the storage space and the application of operations on image quadtrees. It allows selecting a quadtree-based representation well-adapted to an application or to the kind of images.

This article is organized in the following way. Section 2 briefly reminds the principles of quadtree representations and of operations applied to quadtrees. Section 3 describes several approaches which minimize the memory space used by image quadtrees encoded in a linear form. Section 4 presents five approaches for managing clusters of images organized in quadtrees. Section 5 compares all the different approaches considered in this article. Finally, some conclusions are addressed in section 6.

2 Representing and manipulating images with quadtrees

A quadtree is built by recursive divisions of the space in four disjoint quadrants. Quadtrees are used for different types of data, like curves, surfaces or volumes [12,31]. The most widely known quadtree allows cutting an image in regions or quadrants according to a given criterion.

2.1 Image quadtree

To represent an image by a quadtree the image is recursively cut in four disjoint quadrants or squares of the same size so that a node of the quadtree represents a quadrant. The root node represents the initial quadrant containing the whole image. In the presentation below, to make the explanations clear, we use a particular case where the cutting of image quadrants stops when quadrants are homogeneous according to the pixel color, black or white. Other criteria, like the same texture [33,36] for instance, may be used to cut images in quadrants.

If an image is not homogeneous (according to a particular criterion), the quadtree root has four descendant nodes representing the four first level image quadrants: northwestern (NW), northeastern (NE), southwestern (SW) and southeastern (SE). A node is a leaf when its corresponding image quadrant is homogeneous; otherwise the node is internal or non-terminal. Internal nodes are called *gray* (G) in case of *binary images*, i.e. containing *black* (B) and *white* (W) pixels. In general, the leaf nodes of a quadtree are not all at the same level, i.e. the quadtree is unbalanced. Figure 1 presents an example of four binary im-

ages and their quadtree representations.

Different encoding methods [1,11,13,29,30,39] are used to associate an identifier (also called *locational code*, *quadcode* or *locational key*) with a quadtree node: a quadtree node and the corresponding image quadrant use the same identifier. In this paper, we use a *Z ordering* [29], following the NW, NE, SW, SE directions, as shown on figure 1. However, any other encoding method can be used. For example, in figure 1, the numeral 0 identifies the initial quadrant representing the whole image. Numerals 0, 1, 2 or 3, following their parent node identifier 0, identify the four first level image quadrants. Recursively, sub-quadrants of an image quadrant n are identified by nk where $k \in [0, 3]$.

2.2 Operations on image quadtrees

To compare the manipulation of images in the quadtree-based approaches presented in the following sections, we apply the same set of operations on image quadtrees. We analyze how the operations should be performed even if they have not been defined by the authors.

Operations on images organized in quadtrees may be considered as operations on quadtrees. In this section, we briefly present a non exhaustive set of operations, divided into three classes: operations applied to one image quadtree, operations applied to two quadtrees and operations applied to a set or a cluster of images represented by quadtrees.

2.2.1 Operations applied to one quadtree

Operations applied to a quadtree consist in reading or modifying an image quadtree. For instance, the complement operation consists in accessing each leaf node of the quadtree and changing the node value. That is, this operation changes the value p of each pixel by the value $(255 - p)$, when a total of 256 different intensity levels are used to represent a pixel, $p \in [0, 255]$. In the case of binary images, black and white values are interchanged.

2.2.2 Operations applied to two images

We define *homologous nodes* as nodes of different quadtrees having the same identifier. The basic operations (union, intersection, difference and comparison) [30] applied to two image quadtrees are applied to the homologous nodes of the corresponding quadtrees. Table 1 shows the result of these operations applied to a pair of homologous nodes in the case of binary images where each pixel has only two values: *black* and *white*. In the following sections, we just describe the union and the comparison of images. The intersection and the difference operations are performed through a similar process. More details can be found in [30,35].

Quadtree comparison needs some explanation. Comparing two image quadtrees means the comparison of the values of each pair of homologous nodes. To compare the descendant nodes of a node n which is internal in a quadtree i but is a leaf in a quadtree j, either the descendant nodes of node n in quadtree j have a special value called *does not exist*, or node n in quadtree j becomes internal and has four descendant leaf nodes having the same value. In the case of binary images, the values of two nodes are equal (marked *true* in Table 1) if both nodes are black (B), white (W) or gray (G). In the case of colored or gray-scale images, a similarity measure on image features (e.g. color or texture) [10,22,20,27,33,36] can be used to define the distance between the values of homologous nodes. Two node values may be considered as similar if the similarity distance between them is below a given threshold. The comparison of two image quadtrees returns two sets of node identifiers: (1) a set corresponds to homologous nodes having the same or a similar value in both quadtrees, (2) another set corresponds to the homologous nodes having different values in both quadtrees. In order to obtain a value of the comparison, a similarity between image quadtrees can be defined.

The *Q-similarity distance* [13,23] is a distance between quadtrees that is defined as the number of pairs of homologous nodes having different values, divided by the cardinal of the union of node identifiers.

Table 1
Results of the operations applied to a pair of homologous nodes

Value of the first node	W	W	W	B	B	B	G	G	G
Value of the second node	W	B	G	W	B	G	W	B	G
Result of the union	W	B	G	B	B	B	G	B	G
Result of the intersection	W	W	W	W	B	G	W	G	G
Result of the difference	W	B	G	B	W	G	G	G	G
Result of the comparison	*true*	*false*	*false*	*false*	*true*	*false*	*false*	*false*	*true*

More formally, let $S(i,j)$ be the set of pairs of homologous nodes having different values in quadtrees of images i and j. Let $U(i,j)$ be the union of the set of node identifiers existing in the quadtree of image i and the set of node identifiers existing in the quadtree of image j. Let $Card(S(i,j))$ (resp. $Card(U(i,j))$) be the cardinal of $S(i,j)$ (resp. of $U(i,j)$, $Card(U(i,j)) \neq 0$).

The Q-similarity distance between image quadtrees i and j, $d(i,j)$, is computed according to definition (1).

$$d(i,j) = \frac{Card(S(i,j))}{Card(U(i,j))} \qquad (1)$$

$d(i,j) \in [0,1]$, that is, it defines a metric distance function, i.e. one satisfying the symmetry, non-negativity and the triangle inequality properties [23]. $d(i,j) = 0$ means "all homologous nodes have the same values in both quadtrees i and j" and $d(i,j) = 1$ means "no homologous node has the same value in both quadtrees i and j ".

For example, the Q-similarity distance between image quadtrees a and b, represented in figure 1, is equal to: $d(a,b) = \frac{Card(S(a,b))}{Card(U(a,b))} = 5/9$. $Card(S(a,b)) = 5$ because homologous nodes 03, 030 to 033 have different values in the quadtrees of image a and image b. Nodes 030 to 033 have the *doesn't exist* value in the quadtree of image a. $Card(U(a,b)) = 9$ because nodes 0, 00 to 03 and 030 to 033 appear in the union of node identifiers of both quadtrees. If node 03 in quadtree of image a is cut in four white descendant nodes, then $d(a,b) = 2/9$, because only homologous nodes 031 and 032 have different values in both quadtrees.

2.2.3 Operations applied to a cluster

Insertion or deletion of images are operations on a cluster of images modifying the cluster organization but not the images already stored in the cluster.

Because images are organized in a cluster, before performing any operation on the images or on the image regions, like performing the same modification on different images, images must be reached or read. Thus, in the following sections, we consider how reading an image depending on the quadtree-based structure used to organize the cluster.

Access to image regions consists in reaching the same region in different images. It allows extracting either similar or different regions in the images of the cluster [13]. Using quadtree representation of images, this operation is performed by reaching the sub-trees with the same root identifier and by comparing all homologous nodes in the sub-trees.

3 Improving the memory space used by image quadtrees

The hierarchical implementation of quadtree uses pointers to nodes and is costly in memory space [32]. To avoid this problem, the linear storage of quadtree has been proposed [1,11,14,18,19,43].

A linear representation of a quad tree is a list of values, which stores the hierarchical tree structure. Node values are encoded following a pre-order (depth-first order) [1,11,14,43] or a breadth first order [18,19]. This encoding is generally used for binary images. When only black leaf nodes are encoded [11], a structure like

B^+tree can be used to store the linear quadtree [1]. This section presents several linear codings and shows how operations, can be applied to these quadtree representations.

3.1 Linear quadtree approaches

3.1.1 Encoding only black leaf nodes

The approaches [1,11] represent a quadtree as a list of its black nodes. A black node is identified by a unique key derived from its ordered list of ancestors and the list of node identifiers is sequenced by keys [2].

In [11], each black node is encoded in a weighted quaternary code with digits 0 (for NW), 1 (for NE), 2 (for SW), 3 (for SE) in base 4, where each successive digit represents the quadrant subdivision from which it originates according a depth-first traversing. All quaternary codes have the same number n of digits, where n is the number of levels of the quadtree. If a black node is at level i, $i < n$, then the traversing process obtains only $(n - i)$ digits and appends i times the symbol X to the code [18]. For example, the quadtrees of images a and b, represented on figure 1, are respectively coded by the following sequences: $\{1, 2\}$ and $\{1X, 2X, 31, 32\}$.

In [1], the SW, NW, SE, NE directions are respectively coded by 1, 2, 3, 4 (following a N-ordering). The key values are given in base 5 and begin by a 1, representing the root node. The fill character is a 0: if a black node is at level i, $i < n$, then the key value contains i zeros. For example, the quadtrees of images a and b, represented on figure 1, are respectively coded by the following sequences: $(11, 14)$ and $(110, 131, 134, 140)$. Moreover, all keys are stored in a B^+tree.

3.1.2 Encoding all leaf and internal nodes

We now consider three different approaches where all nodes are coded.

The DF-expression [14]: The *Depth-First Expression* is a linear implementation where a left parenthesis represents a gray node. Symbol 1 represents a black leaf node and symbol 0 represents a white leaf node. The quadtree is coded according to a depth-first ordering. For example, the quadtrees of images a and b, represented on figure 1, are respectively coded by the following sequences: (0110 and (011(0110.

The Compact Improved Quadtree [43]: The *Compact-IQ* codes a white leaf node by 0, a black leaf node by 1 and a gray node by 2. The entire image is encoded by a list $P =< P_1, P_2, ..., P_n >$ where n is the number of gray nodes in the quadtree of the image (except the root node which is not coded). The image quadtree is read using a pre-order. Each gray node G_i, $i \in [1, n]$, is encoded using the coded color (whose value is 0, 1 or 2) of its four descendants. More formally, the coding value of G_i, noted P_i, is given by: $P_i = \sum_{j=0}^{3} C_j(G_i) * 3^j$, where C_j is the coded color of the j-direction child of G_i, $j \in [0, 3]$ denoting directions NW, NE, SW and SE. For example, the first quadtree of figure 1 is represented by the list $< 12 >$, because $0 * 3^0 + 1 * 3^1 + 1 * 3^2 + 0 * 3^3 = 12$. The second quadtree of figure 1 is represented by the list $< 66, 12 >$, because $0*3^0+1*3^1+1*3^2+2*3^3 = 66$ and $0 * 3^0 + 1 * 3^1 + 1 * 3^2 + 0 * 3^3 = 12$. Furthermore, in order to reduce the number of bits required for each coding value P_i, the authors of [43] use an entropy coding. Each P_i value is represented by an Huffman code depending on the level of the coded gray node in the quadtree. We do not give more details in this article about the entropy coding - for details we refer to [43].

The CBLQ code [19]: In the CBLQ (*Constant Bit-length Linear Quadtree*) code, each black leaf node is coded by 1 and each white leaf node by 0. The numeral 2 codes an internal node only if one of its descendants is internal. Otherwise, if all its descendants are leaves, a 3 codes the node. Unlike the encoding of [14,43], a quadtree is represented by a CBLQ code following a breadth first order. The root node is not coded. For example, the quadtrees of image a and b, represented on figure 1, are respectively represented by the following CBLQ codes: 0110 and 0113 0110.

3.2 Operations on linear quadtrees

This section describes how the operations defined in section 2.2 are applied to linear quadtrees.

Image Reading: Reading (displaying) an image represented by a linear quadtree consists in uncoding the image. For example, in the Compact-IQ [43], the uncoding procedure gives the code value P_i of each gray node G_i from the entropy coding of P_i. The color of the j-direction child of G_i is unpacked by the function: $(P_i/3^j)$ mod 3, for $j \in [0, 3]$.

Complement operation: In [1,11], where only black nodes are encoded, the complement operation totally changes the linear sequence representing the quadtree: the black nodes identifiers are replaced by the white ones. In the DF-Expression [14] and in the CBLQ code [19], the complement operation only changes the value 0 to 1 and vice versa. In the Compact-IQ [43], the same operation modifies the coding value of the gray nodes (P_i) and thus modifies the Compact-IQ representation (P). Consequently, each P_i value must be uncoded before the application of the complement operation and coded again afterward.

Quadtree Union: When only black nodes are encoded [1,11], the union operation is performed as a merge of two input quadtrees, with the nodes from the merge passed serially for compaction. The compaction, also called condensation in [11], consists in reducing the list of black node identifiers to a minimal list: if a node identifier appears in the list, then the identifiers of its descendants must be removed from the list and if four descendants appear in the list, then the four identifiers are replaced by the father node identifier. Performing the union operation is more complex either using the encoding method of the CBLQ code [19] or the Compact-IQ [43]. In these approaches, a table is given and it contains: the encoding of homologous nodes in two image quadtrees; the result of their union and a process (copy or deletion of a sub-tree) to be performed later. A process is performed in the case that the union creates four black (or four white) descendants, changing the father node from internal to leaf node. See [19,43] for more details.

Quadtree comparison: Comparing two images coded by a linear representation amounts to the comparison of the values of homologous coded nodes. However, in the Compact-IQ [43], an image quadtree must be uncoded before any comparison.

Operations applied to a cluster: Using a linear coding, images in a cluster are stored independently: any image can be inserted (or deleted) in the cluster. Applying operations to the images of the cluster using this representation requires: (1) the retrieval of each image in the cluster and (2) the reading of the entire linear coding of each image. Moreover, in the Compact-IQ [43], images must be uncoded before any operation. The following section presents structures managing cluster of images.

4 Representing and managing a cluster of images

The representation of cluster of images by data structures has been considered in different articles. Several approaches have been proposed to represent the content of images using quadtrees [21,22,24,33]. However, these approaches focus on content-based image retrieval and do not consider storage optimization of cluster of images or operations on images, which is the focus of interest of this work.

The quadtree-based approaches of [7,13,18,38–41] focus on the optimization of the storage of cluster of images. Their goal is to maximize the sharing of common parts between quadtrees. The approaches of [18,38–40] (see sections 4.1 and 4.2) are based on extensions of overlapping between successive data structures of the same type [3,4]. The approach of [13] (see section 4.4) is based on the Database Version approach of [6].

4.1 Overlapping of hierarchical quadtrees

The technique of overlapping trees was initially presented in [3,4] to manage the evolution of text files. The mechanism of overlapping has been extended to sequences of a given data structure, like B-trees [5,25], R-trees [26] or quadtrees implemented with the hierarchical [40] or the linear form [18,38]. The authors of [40] propose a technique of overlapping to represent sequences of similar binary raster images using hierarchical quadtrees. This section presents this structure,

explains how parts of quadtrees are shared and shows how operations can be applied.

4.1.1 Principles

To store images in sequence, the authors of [40] use the overlapping mechanism: when a new image i is inserted, its quadtree overlaps the last quadtree of the sequence, i.e. the quadtree of image $(i-1)$, if parts of both image quadtrees have the same value. The quadtrees representing image $(i-1)$ and image i share identical parts (homologous nodes with the same value) in both quadtrees. The common nodes are referenced in the quadtree of $(i-1)$ from the quadtree of i. When a leaf node has different values in two successive quadtrees, all the nodes appearing in the path from the root to the modified node are copied in the quadtree of image i. Each quadtree node has a counter reference, which contains the number of pointers currently referencing the node. All nodes with a reference counter greater than 1, together with all descendants of such nodes, constitute shared information. This counter allows to perform deletion of a particular quadtree from the overlapped family [40]. An example of overlapped quadtrees is presented in figure 2. The parts of quadtrees representing the images a (in the left of the figure 2.A) and b (in the right), which are different, are represented in gray in the quadtree of b. Both quadtrees share nodes 00 to 02, because they appear with the same value in both of them. On the other hand, node 03 has different values in both quadtrees: a new sub-tree, whose root is node 03, is created in the quadtree representing image b. In figure 2.B, dotted lines represent references to shared nodes when the quadtrees overlap and the pair of numbers represented under each quadtree node corresponds to the pair of *(node identifier, reference counter value)*.

4.1.2 Operations

Image reading: Reading an image i in a sequence of overlapped quadtrees consists of two steps: (1) retrieving the root of the image quadtree and (2) reaching and reading nodes of the quadtree from its root node. Reading an image is as much time consuming using overlapped quadtrees [40] as using independent quadtrees, i.e., one quadtree for each image.

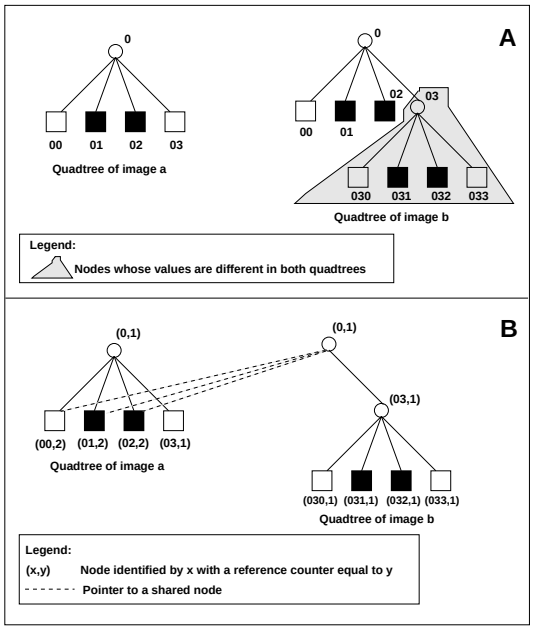

Fig. 2. Overlapping Quadtrees.

Quadtree modification: The overlapping structure is *partially persistent*: all quadtrees can be reached, but only the last one can be modified [8]. Thus, a modification or a group of modifications performed on a quadtree i, which is not the last one of the sequence, creates a new quadtree j. If the new quadtree j is inserted in the sequence, then it is appended at the end of the sequence in order not to modify other overlapped quadtrees. There is no overlapping between quadtrees i and j, if $j \neq i+1$. Overlapping can exist between quadtree j and the preceding quadtree in the sequence, $(j-1)$. It is worth noticing that the approach of [40] is proposed to store a linear sequence of images. However, as the authors of [5] show, the overlapping mechanism, applied to a hierarchical structure, can be used for a non-linear sequence, i.e. using a tree ordering. Consequently, if we use the Q-similarity distance to organize a tree of overlapped quadtrees then a new image quadtree i is inserted as overlapping an image quadtree j when the distance Q-similarity distance between i and j has the smallest value among all the Q-similarity distances between i and the other images of the cluster. The smaller the Q-similarity distance, the greater the sharing of nodes.

Quadtree complement: The complement op-

eration is the same with or without the overlapping mechanisms. It creates a new quadtree i which can be inserted in the sequence by overlapping a quadtree of the sequence (the last one or using the Q-similarity distance).

Quadtree union: The union operation is performed by reading two image quadtrees in the sequence of overlapped quadtrees. If the resulting quadtree of the union is inserted in the cluster, it overlaps a quadtree of the sequence.

Quadtree comparison: Comparing the values of two homologous nodes in two quadtrees is performed by reading two paths from the root in both quadtrees. If the node values are the same or if a similarity distance between node values is below a given threshold, then *true* is returned, otherwise $false$ is returned. The result of the comparison of two images returns two sets of node identifiers or a numerical value if the Q-similarity distance is used.

Operations applied to a cluster: In the Overlapping Quadtrees [40], any image can be inserted, but only at the end of the sequence, whereas any image can be deleted from the sequence because of the reference counters. Because the structure is partially persistent, a modification or a group of modifications on an image quadtree, internal to the sequence, generate a new overlapped quadtree. Hence, performing the same update on p quadtrees, internal to the sequence, requires the creation of p new quadtrees.

Access to image regions: For Overlapping Quadtrees, the access must be performed from the root node of each image. Users have access to each image independently of the other images. If the purpose of the access is the comparison of the same region in different images, then the comparison of the value of node n belonging to different overlapped image quadtrees must be explicitly computed, because no information on the sharing (i.e. knowledge about the quadtrees currently sharing a node) is directly available from a node. Only the number of quadtrees sharing a node is known. For example, if the node n of quadtree i has a reference counter equal to x, then to know which quadtrees share node n with quadtree i at most x quadtrees preceding quadtree i in the sequence and x quadtrees following quadtree i in the sequence must be read. Knowing the sharing implies several readings. The sharing could be known directly from a node n by adding the identifier of the first quadtree sharing node n to the reference counter of n.

The approach of [40] has been extended to linear representations in [38,39]. In this extension, the black node identifiers are stored in a B^+tree like in [1], and the overlapping mechanism is applied to the B^+trees, each B^+tree representing a linear quadtree linear. Thus, in [38,39], B^+trees are subject to the overlapping mechanism similar to the Overlapping Quadtrees. As the overlapping mechanism is the same, we will not consider further this approach.

4.2 Linear overlapped quadtrees

Among the linear representations presented in section 3, only Lin proposes a procedure, called *overlapped CBLQ code*, for coding a sequence of images by linear quadtrees [18], i.e. a totally ordered sequence, with the goal of an efficient management of sequences of video images. All the other approaches [1,11,14,43] can also be used to store a cluster of images, by coding each image quadtree independently. This section presents the *overlapped CBLQ code* of [18] and shows how parts of image quadtrees are shared and how operations are applied to overlapped linear quadtrees.

4.2.1 Principles

In [18], the first image and the last one in the sequence are coded using the CBLQ code (see section 3.1.2). Then, all images in the sequence, except the first one, are coded using an overlapped CBLQ representation. This representation codifies the differences between two successive images.

The difference between two successive images i and $(i-1)$ is obtained through a two-step procedure. Step 1: the first four nodes of quadtree of i and $(i-1)$, are compared. If a node has the same value in both quadtrees, then it is coded by A in the overlapped coding of i. If homologous nodes are different, two cases appear: if one node

Table 2
Coding of a quadtree node by the overlapped CBLQ representation [18]

Value of a node in the first quadtree:	White	White	Gray	Gray	Black	Black
Value of a node in the second quadtree:	Gray	Black	Black	White	White	Gray
Overlapped code:	C	D	C	D	C	D

is gray, then the code is B, otherwise the code is C or D according to the black or white value of each node (see table 2). Step 2: a letter A, C or D codes each descendant of a node coded by B in step 1, according to their respective values in quadtrees of $(i-1)$ and i (see table 2). If a node n coded by a B in step 1 is not subdivided in quadtree $(i-1)$ but is subdivided in quadtree i, then each value of nodes nk, $k \in [0,3]$, in quadtree of i is compared with the value of node n in the quadtree of $(i-1)$.

As an example the coding of the four images represented in figure 1 is:

0110 ⟵ CBLQ code of image *a*

AAAB ADDA ⟵ Overlapped CBLQ of image *b*. It codes the differences between *a* and *b*

AACB ACCA ⟵ Overlapped CBLQ of image *c*. It codes the differences between *b* and *c*.

AAAB ADDA ⟵ Overlapped CBLQ of image *d*. It codes the differences between *c* and *d*.

0103 0110 ⟵ CBLQ code of image *d*

4.2.2 Operations

Image reading: The following operations are defined, in [18], for managing an image in a linear sequence: *un_overlap* (resp. *re_overlap*) recursively computes the CBLQ coding of an image, according to the previous (resp. following) image. Function *un_overlap* (resp. *re_overlap*) is recursively applied from the first (resp. last) image of the sequence: $i = un\_overlap(i-1, O_i)$ and $i = re\_overlap(i+1, O_{i+1})$ with O_i representing the overlapped CBLQ code of image i i.e. the difference between image $(i-1)$ and image i. Reading an image from a sequence coded by the overlapped CBLQ code consists in recursively using *un_overlap* (resp. *re_overlap*) from the first (resp. the last) image. Then, when the CBLQ encoding of the image is obtained, the image can be finally computed. In order to speed up image reading, the author of [18] proposes to store the CBLQ code (not only the overlapped one) of some intermediate images of the sequence. This allows the application of functions *un_overlap* (resp. *re_overlap*) from intermediate images of the sequence.

Quadtree modification: In [19], operations on image quadtrees coded by the CBLQ code are defined, but they are not defined on overlapped CBLQ codes in [18]. Thus, an image quadtree may be uncoded by using *un_overlap* or *re_overlap* functions before any modification. A modification or a group of modifications cannot be directly applied to a quadtree i included in a sequence, because of the change of the CBLQ code of i and therefore the overlapped CBLQ codes between i and the following quadtrees in the sequence. Such a structure is partially persistent.

Quadtree complement: The effect of the complement operation in the CBLQ representation of an image exchanges labels 0 and 1, after the image uncoding. This operation creates a new image quadtree which can be inserted at the end of the sequence.

Quadtree union: As mentioned above, operations on images coded by an overlapped CBLQ code are not defined in [18]. Thus, performing the union of two image quadtrees i and j requires the uncoding of both images, using *un_overlap* or *re_overlap* functions, in order to obtain the CBLQ code of each image for which the union operation is defined [19].

Quadtree comparison: If the compared quadtrees are successive in the sequence, reading the overlapped CBLQ code of the second quadtree performs the comparison, because the overlapped CBLQ representation codes the difference between a quadtree and the previous one in the sequence. However, if the compared quadtrees are not successive in the sequence, they must be uncoded in order to be compared.

Operations applied to a cluster: Images can be inserted in the cluster but only at the end of the sequence, and no image can be deleted without rebuilding all the sequence of overlapped linear codes. The overlapped linear quadtree is a partially persistent structure. As a consequence, a modification creates a new overlapped quadtree. Then, performing the same update on different images requires the creation of new images or the rebuilding of the linear representation of the cluster from the first updated image.

Access to image regions: As explained above, each image must be rebuilt before any access.

4.3 Two approaches to Inverted Quadtrees

The approaches of [7] and [41] propose an *Inverted Quadtree*, called *Fully* (*FI-Quadtree*) in [7] and *Dynamic* (*DI-Quadtree*) in [41]. In those structures, a set of binary images is encoded in a single quadtree. In a classical representation, an image identifier is associated with a set of prefixes encoding the corresponding bitmap (e.g. a set of black node identifiers). In the inverted representation, a prefix is associated with a set of image identifiers. An Inverted Quadtree representing the four images of figure 1 appears in figure 3. Each node n in the Inverted Quadtree is associated with either a set of image identifiers, whose quadtree contains a black node n, or with an empty set ($\emptyset$) if no image quadtree contains a black node n. This section presents the principles of both approaches and shows how image operations can be applied.

4.3.1 Principles

The FI-Quadtree [7] consists of a full quadtree, i.e. a quadtree where each node has four children except for the level-0 nodes, which are all leaves. Each node holds a bit string of maximum length (the maximum number of images in the database). Each bit designates a separate image. A black node in the quadtree of any image is identified by a 1 in the bit corresponding to the image in the node string. This structure is static: it can hold a predefined number of images. This number can be increased after a total reorganization of the structure. On the other hand, in the DI-quadtree [41], each node of the full quadtree points to a list containing only identifiers of the images that have the corresponding black node in their quadtree. The list is implemented by chained segments so that the structure is dynamic: any number of images can be added. Both approaches [7,41] are suitable for answering queries based on image content (e.g., exact and fuzzy image pattern searching).

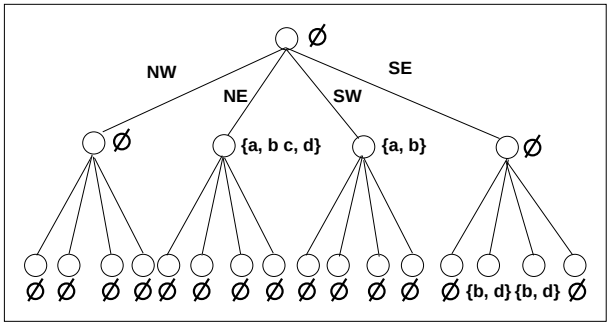

Fig. 3. An example of an Inverted Quadtree.

4.3.2 Operations

Image reading: To read an image from an Inverted Quadtree, the structure is read from the root node. If the image identifier appears in the list associated with a node n, then the corresponding quadrant is black in the image. Otherwise, the corresponding quadrant is white or gray. To determine the color (white or gray) of quadrant n, all descendant nodes of node n, until leaf nodes, are read and, if the image identifier does not appear in any list associated with a descendant node, then the corresponding quadrant is white.

Quadtree modification: Modifying an image quadtree stored as an Inverted Quadtree consists in a set of additions or deletions of the image identifier in the lists associated with the quadtree nodes. The Inverted Quadtree is called *fully persistent* structure because every image quadtree can be reached and modified [8].

Quadtree complement: A quadtree complement is performed by reading all nodes of the Inverted Quadtree and by adding or deleting the corresponding image identifier in the lists associated with the quadtree nodes. For each node n from the root of the Inverted Quadtree, if the image identifier appears in a list associated with

node n, then the identifier is deleted (node n was black and becomes white after the complement operation). If the image identifier does not appear in the list associated with n, then it is added only if it is not associated with any node of the sub-tree whose root is n (node n was white before the operation). Otherwise node n is gray and the complement operation does not change its value but the values of its descendants.

Quadtree union: As shown in Table 1, the result of the union applied to two homologous nodes is black when one of the node is black. Thus, in the Inverted Quadtree, if at least one image identifier appears in the list associated with a node n, it means that the result of the union is black. Otherwise, all descendants of node n, until reaching the leaf nodes, are read to determine if the homologous nodes n are white or gray. The image resulting from the union can be inserted in the Inverted Quadtree.

Quadtree comparison: Comparing the value of two homologous nodes is performed by reading the corresponding node in the Inverted Quadtree. For each node n, if the identifiers of the compared images appear in the list associated with the node, then the result of the comparison is *true*. Otherwise the descendants of n in the Inverted Quadtree are read in order to determine if the node is white or gray in both image quadtrees.

Operations applied to a cluster: The structure is fully persistent, thus any image quadtree can be inserted or deleted. Performing the same update in different image regions is computed by reading each image and by performing the update to each image quadtree. The structure of the Inverted Quadtree facilitates this operation.

Access to image regions: Accessing a region in different images is performed by accessing the corresponding nodes in the image quadtrees, i.e. by accessing the corresponding nodes in the Inverted Quadtree. Given the structure of an Inverted Quadtree it is easy to compare homologous nodes n in different quadtrees because all the informations are contained in node n of the Inverted Quadtree. However, when some homologous nodes n are not black, some additional descendant nodes must be read in order to know if homologous nodes n are white or gray.

4.4 Generic Quadtree

This section presents the Generic Quadtree [13], an inverted structure which expands the range of tractable images in other inverted quadtrees approaches by allowing the management of binary, gray-scale or color images.

4.4.1 Principles

The Generic Quadtree is a quadtree-based structure whose nodes are called *generic nodes*. For each node appearing in an image quadtree, there is a node with the same identifier in the Generic Quadtree. A generic node n represents all nodes n of the quadtrees of images belonging to the cluster. It contains the whole information necessary to rebuild the value of the node with the same identifier n in each image quadtree. Each generic node may be seen as a table with two columns and one or several lines. For example generic node 02 in figure 4 contains two lines, while generic node 01 has only one line. Each line l of a generic node n contains a list of image identifiers and a value v of quadtree node. v is the value of node n in each image quadtree whose identifier i appears in line l. A generic node can take any value in the following logical OR-sequence: $\perp$, meaning that the node *does not exist* in the quadtrees of images appearing in the corresponding line (see generic node 030 in figure 4); OR I, meaning that the node is internal - it has four descendants - (see generic node 0 in figure 4); OR *black* if it is a black leaf, OR *white* if it is a white leaf, etc.

The Generic Quadtree is based on two principles of sharing of quadrant values between images: *explicit* and *implicit*. If a quadrant q has the same value in several images, this value is stored only once and is associated with the list of image identifiers. In this case, the sharing is called *explicit*, because the identifier of each image sharing the value is explicitly present in the list. The *implicit* sharing is based on the following rule: *except if the identifier of an image i is explicitly associated with another value v, image i shares the value with its parent image*. To use

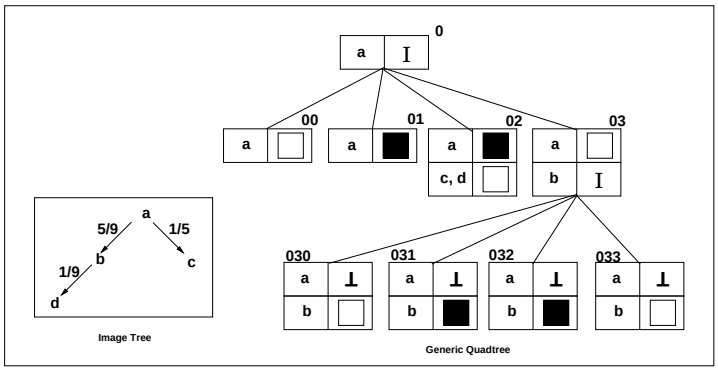

Fig. 4. The Image Tree and the Generic Quadtree of the images represented in figure 1.

this rule, all the images of the cluster are organized in a tree structure called *Image Tree*. As the Q-similarity is a distance between quadtrees, in order to increase the implicit sharing of node values and consequently to diminish the size of generic nodes, an image j is inserted in the Image Tree as a descendant of an image i if the Q-similarity distance between the quadtrees of i and j has the smallest value among all other distance values (see on the left of figure 4 - the Q-similarity distance between each pair of quadtrees is represented in the tree labels). Image identifiers, inside the Image Tree, are built in such a way that knowing the identifier of an image in the Image Tree, all its ancestor identifiers can immediately be deduced [6,17]. Applying the sharing rule, all the nodes n of the quadtrees representing images descendant of image i implicitly share the value v; this implicit sharing is stopped by a descendant image identifier appearing in another line of generic node n, i.e. associated with another value v' (See generic node 02 in figure 4. Only image quadtree b implicitly shares the value *black* with image quadtree a).

4.4.2 Operations

Image reading: Reading an image i from a Generic Quadtree is performed by reading the structure from its generic root node, identified by 0. The value of a node n, for an image quadtree i, is determined according to the sharing rule presented above. If the value of node n is I for image quadtree i, then the node is internal in the quadtree of i and the generic nodes, descendants of n, are read. Otherwise, the node n is a leaf of image quadtree i and the obtained value is its color.

Quadtree modification: The Generic Quadtree allows any modification. It is a fully persistent structure. Modifying an image updates its quadtree. When the modification preserves the image quadtree structure (i.e. when only values of leaf nodes are changed), performing a modification of value of quadtree node n generates changes in generic node n. The new value $v.new$ of node n for the modified image quadtree i must be implicitly or explicitly associated with i. However, to avoid the propagation of the modification in the possible descendant images of i, the implicit sharing between i and its image descendants must be cut. Thus each image descendant of i, not explicitly associated with a value in generic node n before the updating, must be explicitly associated with the old value $v.old$ that it was implicitly sharing with i before the modification. When the modification alters the image quadtree structure (nodes are divided, merged, created or deleted), performing the modification implies changes in the modified nodes and in their ancestor or descendant nodes in the Generic Quadtree. For example, if after the modification, four brother nodes $n0$, $n1$, $n2$ and $n3$ have the same color v in quadtree i, then they do not exist anymore in quadtree i (value $\perp$ is associated with the image identifier i in the corresponding generic nodes) and their father n takes the value v (value v is associated with the image identifier i in generic node n). On the contrary, if after the modification, a node n is divided in four descendant nodes in quadtree i, then node n is internal in quadtree i (value I is associated with the image identifier i in generic node n) and nodes $n0$, $n1$, $n2$ and $n3$ exist in

quadtree i (a node value is associated with identifier i in each corresponding generic nodes).

Quadtree complement: The complement operation applied to a quadtree i changes the value of all leaf nodes of the quadtree. This operation preserves the structure of the image quadtree.

Quadtree union: If both image identifiers are associated with the same value v of a generic node n, then the result of the union of homologous nodes is v. Table 1 displays the possible values of the node union operation, when the image identifiers are not associated with the same value. The implicit or explicit sharing between image resulting from the union and other images of the cluster depends on the position of the resulting image in the Image Tree.

Quadtree comparison: Comparing the values of homologous nodes in the Generic Quadtree is performed by reading the corresponding generic node. If the identifiers of the compared images are associated with a similar node value, the comparison between the two homologous nodes is *true*.

Operations applied to a cluster: This structure allows performing the same modification directly on several image quadtrees. For example, simultaneously updating the node n of image quadtree i and all quadtrees the images descendant of image i consists in modifying the value of generic node n associated with image i (for example from white to black), and in deleting the identifiers of the descendants of image i in generic node n. Thus all descendants of image i implicitly share the value of node n with image i. However, the consistency of the quadtrees of all modified images must be checked: the updating may lead to node creation, node deletion or node modification in the modified images (see above in image modification).

Access to image regions: Each generic node can be reached according to two dimensions. One dimension allows hierarchical navigation inside the quadtree of an image. The other dimension, across images, allows jumping from one image to another using generic nodes. Because of the sharing, the comparison of node n values in different image quadtrees is immediate.

5 Comparison and discussion

In this section, the different methodologies are qualitatively compared regarding memory space and operations on image clusters. We intend to show weaknesses and strengths of each approach and underscore the areas of applications where each method could fit better. This comparison may be seen as a starting point for benchmark tests. Subsection 5.1 compares image representation. Then the following subsections compare the approaches according to: (5.2) image organization, (5.3) image memory space, (5.4) data access and (5.5) operations. Table 3 (at the end of the article) sums up the comparison.

5.1 Classification of approaches

The approaches presented in this article may be classified in two classes: (1) methods based on hierarchical quadtrees [7,13,40,41] and (2) methods based on linear quadtrees [1,11,14,18,19,43]. All of these methodologies, except the Overlapping Quadtrees of [40] and the Generic Quadtree of [13], manage only binary (white and black) images.

The hierarchical representations of [7,13,40,41] (see subsections 4.1, 4.3 and 4.4) are defined to store clusters of images. On the other hand, among the linearly coded representations (see section 3), only the overlapped CBLQ code [18] is designed to store a cluster of images (see section 4.2). The other linear approaches [1,11,14,43] allow the representation of a cluster by representing independently each image without storage sharing.

The Overlapping Quadtrees of [40] and the overlapped CBLQ code of [18] use the mechanism of overlapping in order to share homologous parts of images having the same value. These approaches are oriented towards the storage of image sequences, as in video applications where direct image manipulation is rare.

In the Fully or Dynamic Inverted Quadtree [7,41] and in the Generic Quadtree [13], all images are

stored in a single quadtree as opposed to the overlapping methods [40,18] where a quadtree is associated with each image. These approaches are oriented toward content-based retrieval or bookkeeping the changes in image processing. The Fully or Dynamic Inverted Quadtrees [7,41] only manage binary image: each node n of the Inverted Quadtree contains the set of quadtree identifiers whose node n value is black. The Generic Quadtree [13] manages gray-scale or colored images: each generic node n contains several lines, each line containing a node value v associated with the set of quadtree identifiers whose node n value is v. The Generic Quadtree [13] can be seen as a generalization of the Dynamic Inverted Quadtrees [41] to the gray-scale or colored images with a specific mechanism of implicit sharing.

5.2 Image organization and value sharing

In the hierarchical [40] and linear [18] overlapping methods, a cluster of images is stored in a sequence, whereas it is stored according to a tree-order in the Generic Quadtree [13] and without any order in the Inverted Quadtrees [7,41].

However, in the hierarchical overlapping method [40], images can be organized not only in sequence but also in a tree-order (see section 4.1.2). When a quadtree i is inserted in a cluster of images organized in a tree order, it overlaps a quadtree j which is a leaf of the structure representing the tree order and which is, for example, chosen according to the minimization of the Q-similarity distance. Thus, quadtree i shares homologous parts with quadtree j and possibly shares identical parts with the ancestors of quadtree j. However, a quadtree cannot share parts with any quadtree but only with the quadtrees of its lineage (its ancestors or its descendants in the tree order). For instance if images a to d are organized in the tree order presented on the left of figure 4 (see section 4.4), then the quadtree of image d overlaps the quadtree of image b and quadtree b overlaps quadtree a. Quadtrees a, b and d share identical parts (nodes 00, 01 03 and $03x$, $x \in (0,3]$). However, quadtree of image d cannot share the value of node 02 with quadtree of image c because image c is not an ancestor of image b in the tree order. Thus, in overlapping methods, there is no sharing when a quadrant or a part of a quadtree has repeatedly the same value in different quadtrees, non successive in the sequence or in the tree order: for instance, in case of periodic or pseudo-periodic occurrences.

On the contrary, the Fully or Dynamic Inverted Quadtrees of [7,41] and the Generic Quadtree [13] maximize the sharing of quadrant values. In these approaches, the information on the sharing is stored in each node of the structure, respectively called inverted or generic node (see sections 4.3.1 and 4.4.1). In the Inverted Quadtrees, any quadtree of a binary image can share the black value of node n with any other image quadtree: the image quadtrees, whose identifiers appear in the list associated with node n, share the black value of node n. For example, in figure 3, the quadtree of image d shares the black value of node 031 and 032 with the quadtree of image b. Given that identifiers c and d do not appear in the list associated with node 02 (SW) and because the lists associated with descendant nodes of 02 are empty, the value white of node 02 is shared by the quadtrees of images c and d. In the same way, in the Generic Quadtree, a common value v is stored only once and is shared by as many image quadtrees as necessary when the same value v appears in the node n of several image quadtrees: value v is automatically shared, implicitly or explicitly. For instance, in figure 4, the quadtree of image d explicitly shares value *white* of node 02 with the quadtree of image c and implicitly shares values of node 03 and its descendants with the quadtree of image b. Reading a node in such structures (Inverted or Generic) allows to know which image quadtrees share the node value, whereas in the Overlapping Quadtrees [40], each quadtree node only contains a counter of references indicating the number of pointers currently referencing the node. This counter does not directly indicate which quadtrees are currently sharing the node. In the absence of special features, discovering this sharing in the Overlapping Quadtrees requires an explicit comparison of the image quadtrees.

5.3 Image memory space

5.3.1 Comparison of linear representations

In the case of binary images, linear approaches [14,18,19,43] do not manipulate pointers as opposed to other approaches [7,13,40,41]. Using experiments, Lin [18] shows that the space storage required by the overlapped CBLQ code is less than for the Overlapping Quadtrees [40].

Compared to CBLQ code of [19] or to the DF-expression of [14], the authors of [43] argue that the Compact-IQ improves the compaction of images by using an entropy coding (see [43] for experimental results). However, the images of the cluster are coded independently as opposed to the overlapped CBLQ code [18]. From the point of view of [18], the space reduction from linear quadtrees to overlapped linear quadtrees is equal to the overlapping percentage of images.

5.3.2 Comparison of hierarchical approaches

In the Overlapping Quadtrees approach of [40], the memory space may be modified in case of creation and deletion of an image quadtree. The former leads to adding a new overlapped quadtree whereas the later removes the quadtree nodes whose reference counter is equal to 1 (otherwise nodes are shared by other quadtrees and must not be deleted). In the Overlapping Quadtrees, images are stored in a partially persistent structure thus modifying an image is not allowed when it is inside the sequence and this operation creates a new image quadtree. After an quadtree insertion, at least one new path from the root of the quadtree to a non shared leaf node is inserted in the set of the overlapped quadtrees. The order of size of the mean depth of a quadtree is $log_4(N)$, where N is the number of nodes in the quadtree. Thus, the minimum order of size of the mean additional space occupied by a structure of k overlapped quadtrees is $k * log_4(N)$ nodes.

In the Generic Quadtree, the memory space may be modified in case of creation, modification or deletion of an image. It must be noticed that, in the table representing a generic node, the maximum number of lines is the number of possible values for a leaf node plus 2 (I for internal node and $\perp$ for *does not exist*). For instance in the black-and-white case, each generic node has a maximum of 4 lines. As a consequence, the storage space used by the Generic Quadtree depends on the number of different node values in the different quadtrees representing the images of the cluster.

In the Inverted Quadtrees [7,41], the memory space may be modified in case of creation, modification or deletion of an image. All these operations add or delete image identifiers in the lists associated with the nodes of the Inverted Quadtree. The number of image identifiers stored in the Inverted Quadtree is directly proportional to the number of black quadrants in the image of the cluster. Moreover, among the Inverted Quadtrees, the Dynamic one [41] is more compact than the Fully one [7] (see [41] for an analytical comparison between both inverted structures).

When updates are performed in the images of the cluster, the Generic Quadtree [13] and the Fully or Dynamic Inverted Quadtrees [7,41] require less memory space than the Overlapping Quadtrees [40]. The justification for this last statement is based on two considerations (1) Let k be the number of elements of an initial set of images stored in the cluster. After a number u of update groups performed on images, the number of images stored in the Generic or the Inverted Quadtrees is still k, while it is $k' = k+u$ in the Overlapping Quadtrees. This is because modifications on one image, internal to the sequence, always create a new image and then a new quadtree in the Overlapping Quadtrees. (2) In the Overlapping Quadtrees, an image shares a node value only with the precedent images in the sequence, whereas in the Generic and Inverted Quadtrees, all the stored image quadtrees can potentially share a node value.

Moreover, it must be noticed that, the Fully and the Dynamic Inverted Quadtrees [7,41] do not store any specific value, like the value *does not exist* managed in the Generic Quadtree [13], or any additional path, like in the Overlapping approach [40].

5.4 Image reading

5.4.1 Reading the entire image

When images are coded independently, finding and displaying an image need no additional operations, except an uncoding for linear representations [14,19,43]. On the contrary, the overlapped CBLQ code [18] is proposed to store and read images in sequence. Rebuilding an image from the sequence in the CBLQ code is complex because of the recursive use of *un_overlap* or *re_overlap*, depending on the choice of beginning the computing by the first or the last image (see section 4.2).

In the Overlapping Quadtrees [40], images are stored in such a way that the storage do not interfere with the reading of images. In the Generic Quadtree [13], reading an image i is performed by reading all generic nodes containing the image i identifier or the identifiers of image i ancestors. The reading of generic nodes stops when all leaf nodes of the quadtree i are found. On the contrary, in the Inverted Quadtrees [7,41], the number of inverted nodes read is higher because nothing indicates if a node is white or gray: the color of a non-black node is only known after the reading of all its descendants.

Each inverted or generic node is explored using an algorithm (see sections 4.3.2 and 4.4.2) whose complexity is $O(k)$, k being the number of stored images in the cluster. On the other hand, the retrieval of image i in the Overlapping Quadtrees [40] consists of two steps: (1) retrieving the root of the image quadtree ; the complexity of this operation is $O(k)$ (2) reaching and reading nodes of the quadtree from its root node. Thus, for the same number of images stored in the cluster (if there is no image updates), an entire image reading is faster in the Overlapping Quadtrees [40] than any other approach.

5.4.2 Accessing regions in the image

In the approaches of [18,40,43], each image is read independently of the others and the comparison of node n values in different image quadtrees must be explicitly computed. In the overlapped CBLQ code of [18] and in the Compact-IQ [43], the image must be uncoded before reaching any image region.

For Overlapping Quadtrees [40], the access must be performed from the root node of each image. On the other hand, using the Inverted Quadtree [7,41] or the Generic Quadtree [13], it is easy to have access simultaneously to the same nodes in different image quadtrees because the nodes of these structures allow jumping from one image to another. The comparison of the value of node n in different image quadtrees is immediate, except for the reading of some additional descendant nodes (in order to determine if a node is gray or white) in the Inverted Quadtrees.

5.5 Operations

The overlapping methods [18,40] have been proposed to store sequences of images. Thus, the only operations allowed on images stored in overlapped quadtrees are reading an image, inserting a new image (at the end of the sequence) or modifying an image whose quadtree is not overlapped (the last image of the sequence).

In the Overlapping Quadtrees [40], an image quadtree can be deleted but only non shared nodes are removed (see 5.3.2). On the other hand, the deletion is not allowed in the overlapped CBLQ code, because deleting an image quadtree requires the rebuilding of the end of the sequence. To operate on image quadtrees coded by the overlapped CBLQ code of [18], image quadtrees must be uncoded. Similarly, operating on image quadtrees represented by the Compact-IQ [43] implies the uncoding of each gray node (value P_j, see section 3.1.2).

On the contrary, for image quadtrees stored in the Fully or Dynamic Inverted Quadtrees [7,41] or in the Generic Quadtree [13], there is no limitation on allowed operations (insertion, modification and deletion). There is a great difference between these approaches and the other methods. These approaches allow many operations which do not explicitly exist for the other approaches: computing the difference between two images or parts of images, following the evolution of an area across images, etc. As a consequence, they are efficient for working on one

image and simultaneously on a set of images. It must be stressed that the Inverted Quadtrees are limited to binary image, whereas the Generic Quadtree can manage any kind of image (binary, gray-scale, color).

6 Conclusions

This paper presents a survey of quadtree-based structures [7,13,18,40,41,43] to give an overview of pros and cons of a variety of approaches for the storage and the manipulation of clusters of images organized in quadtrees. Among the proposed structures, the linear approaches [18,19,43,38] and the Inverted Quadtrees [7,41] are limited to binary images. On the other hand, the Overlapping Quadtrees [40] and the Generic Quadtree [13] allow the manipulation of gray-scale or color images. Some approaches are based on the same mechanisms: the linear representation of quadtree [18,19,38,43], the overlapping mechanisms [19,38,40], or an inverted structure [7,13,41]. The comparison of these approaches is summed-up in table 3.

Each approach, presented in this article, has been proposed for specific applications. The CBLQ code [18,19] and the Compact-IQ [43] compact binary images with a perspective of image manipulation. The Inverted Quadtrees [7,41] compact binary images as well but with the aim of researching pattern in images. Compaction and delivery of video sequence are the main areas of applications of the Overlapping Quadtrees [40] and the overlapped CBLQ code [18], whereas it is the application of image processing operations and the bookkeeping of the changes in the processed images for the Generic Quadtree [13]. As a consequence, none of these structures efficiently supports all operations or presents the best compromise solution between compaction and image manipulation. This qualitative comparison presents the advantages and the disadvantages of each approach allowing the reader to be able to choose the "best" structure according to its application needs.

Acknowledgments: The authors would like to thank the referees of this article and V. Mujica for their helpful comments and suggestions.

References

[1] D.J. Abel, A B+tree Structure for Large Quadtrees, Computer Vision, Graphics and Image Processing 27(1):19-31, 1984

[2] D.J. Abel. Some Elemental Operations on linear Quadtrees for Geographic Information Systems, The Computer J., 28(1):73–77, 1985

[3] F.W. Burton, M.M. Huntbach and J.Y.G. Kollias, Multiple Generation Text Files Using Overlapping Tree Structures, The Computer J. 28(4):414-416, 1985

[4] F.W. Burton, J.G.Kollias, D.G. Matsakis and V.G. Kollias, Implementation of overlapping B-trees for time and space efficient representation of collections of similar files, The Computer J. 33(3):279-280, 1990

[5] M.J. Carey, D.J. DeWitt, J.E. Richardson and E. J. Shekita, Chapter 14 - Storage Management for Objects in EXODUS in: Object-Oriented Concepts, Databases, and Applications, Addison Wesley, 341-369, 1989

[6] W. Cellary and G. Jomier, Consistency of Versions in Object-Oriented Databases, Proc. 16th Int. Conf. on Very Large DataBases (VLDB), Brisbane (Australia), 1990

[7] J.P. Cheiney and A. Touir, FI-Quadtree: a New Data Structure for Content-Oriented Retrieval and Fuzzy Search, Proc. 2nd Symposium on Spatial Databases (SSD'91), 1991

[8] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making Data Structures Persistent, Journal of Computer and System Sciences, 38:86-124, 1989

[9] R.A. Finkel and J.L Bentley, Quadtrees: A Data Structure for Retrieval on Composite Keys, Acta Informatica 4(1): 1-9, 1974

[10] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, et al., Query by Image and Video Content: The QBIC System, Computer 28(9):23-32, 1995

[11] I. Gargantini, An Effective Way to Represent Quadtrees, Comm. of ACM 25(12):905-910, 1982

[12] V. Gaede and O. Günther, Multidimensional Access Methods, ACM Comp. Survey 30(2):170-231, 1998

[13] G. Jomier, M. Manouvrier and M. Rukoz, Storage and Management of Similar Images, Journal of the Brazilian Computer Society (JBCS), 3(6):13-26, 2000

[14] E. Kawagushi and T. Endo, On a method of binary picture representation and its application to data compression, IEEE Trans. Pattern Anal. Mach. Intell. 2(1):27-35, 1980

[15] M-A. Kang, S. Servigne, K-J. Li and R. Laurini, Indexing Field Values in Field Oriented Systems: Interval Quadtree, Int. Conf. on Information Knowledge Management (CIKM), Kansas City, 1999

[16] O. Kao and I. la Tendresse, CLIMS - A System for Image Retrieval by Using Colour and Wavelet Features, Proc of 1st Inter. Conf. on Advances in Information Systems (ADVIS), Izmir (Turkey), pp 238-247, 2000

[17] A. Keller and J. Ullman, A Version Numbering Scheme with a Useful Lexicographic Ordering, Proc. of 11th Int. Conf. on Data Engineering (ICDE), Taipei (Taiwan), pp 240-248, 1995

[18] Tsong-Wuu Lin, Compressed quadtree representations for storing similar images, Image and Vision Computing 15:833-843, 1997

[19] Tsong-Wuu Lin, Set operations on Constant Bit-Lenght linear Quadtrees, Pattern Recognition 30(7):1239-1249, 1997

[20] Guojun Lu, Multimedia Database Management Systems, Artech House Publishers, 1999

[21] H. Lu, B-C. Ooi and K-L. Tan, Efficient Image Retrieval by Color Contents, Int. Conf. on Applications of Database (ADB), Vadstena (Sweden), 1994, Lecture Notes in Computer Sciences - 819 - Springer Verlag

[22] S. Lin, M. Tamer Özsu, V. Oria, and R. Ng. An Extensible Hash for Multi-Precision Similarity Querying of Image Databases. In Proc. of the 27th Int. Conf. on Very Large DataBase (VLDB'2001), Roma (Italy), 2001

[23] M. Manouvrier, Objets Similaires de Grande Taille dans les Bases de Données, PhD Thesis, Paris IX Dauphine Univ. (France), Jan. 2000 http://www.lamsade.dauphine.fr/~manouvri/thesis_subject.html

[24] J. Malki, N. Boujemaa, C. Nastar and A. Winter, Region Queries without Segmentation for Image Retrieval by Content, 3rd Int. Conf. on Visual Information Sys. (Visual), Amsterdam, 1999

[25] Y. Manolopoulos and G. Kapetanakis, Overlapping B+-Trees for Temporal Data, Proc. of Jerusalem Conf. on Inf. Technology (JCIT), Israël, 491-499, 1990

[26] M.A. Nascimento and J.R.O. Silva, Toward Historical R-trees, Proc. of ACM Symposium on Applied Computing (SAC), Atlanta (USA), 235-240, 1998

[27] E.G.M. Petrakis and C. Faloutsos, Similarity Searching in Medical Image Databases, IEEE Trans. on Know. and Data Engineering 9(3): 435-447, 1997

[28] J-P. Peloux, G. Reynal de St Michel and M. Scholl, Evaluation of Spatial Indices Implemented with the O_2, Ingénierie des Systèmes d'Information, 3(4), 1995

[29] P. Rigaux, M. Scholl, and A. Voisard. Spatial Databases With Application to GIS, chapter 6 - Spatial Access Methods, pages 201–266. Morgan Kaufmann Pub., 2002

[30] H. Samet, The Quadtree and Related Hierarchical Structures, Comp. Surveys 16(2):187-260, 1984

[31] H. Samet, The Design and Analysis of Spatial Data Structures, Addison Wesley, 1989

[32] I.P. Stewart, Quadtrees: Storage and Scan Conversion, The Computer J. 29(1):60-75, 1986

[33] J.R. Smith and S-F. Chang, Quad-Tree Segmentation for Texture-Based Image Query, Proc. of 2nd Annual ACM Multimedia Conf., 1994

[34] C. A. Shaffer, H. Samet and R. C. Nelson, QUILT: a geographic information system based on quadtrees, Int. Journal on Geographical Information Systems, 4(2): 103-131, 1990

[35] H. Samet and R. E. Webber, Hierarchical Data Structures and Algorithms for Computer Graphics. Part I: Fundamentals, IEEE Computer Graphics and Applications 8(5):48-68, 1988

[36] G. Sheikholeslami, A. Zhang and L. Bian, A Multi-Resolution Content-Based Retrieval Approach for Geographic Images, GeoInformatica, 3(2): 109-139, 1999

[37] Y-H. Tsai and K-L. Chung, Some image operations on S-tree-related spatial data structure, Image and Vision Computing, 17: 897-904, 1999

[38] T. Tzouramanis, M. Vassilakopoulos and Y. Manolopoulos, Overlapping Linear Quadtrees: a Spatio-Temporal Access Method, ACM GIS'98, Washington D.C., 1998

[39] T. Tzouramanis, M. Vassilakopoulos and Y. Manolopoulos, Overlapping Linear Quadtrees and Spatio-Temporal Query Processing, The Computer J. 43(4):325-343, 2000

[40] M. Vassilakopoulos, Y. Manolopoulos and K. Economou, Overlapping Quadtrees for the Representation of Similar Images, Image and Vision Computing 11(5):257-262, 1993

[41] M. Vassilakopoulos and Y. Manolopoulos, Dynamic Inverted Quadtree - a Structure for Pictorial Databases, Information Systems, Special Issue on Multimedia Information Systems 20(6):483-500, 1995

[42] J-G. Wu and K-L. Chung, The Logical Representation of Bincode and its applications in Manipulating Binary Images, Real-Time Imaging, 5: 397-408, 1999

[43] Yuh-Horng Yang, Kuo-Liang Chung and Yao-Hong Tsai, A compact improved quadtree representation with image manipulations, Image and Vision Computing 18: 223-231, 2000

Table 3
Summing-up and comparison of quadtree-based approaches storing and managing clusters of images

Linear structures	**Linear quadtrees [1,2,11], DF-Expression [14], CBLQ code [19], Compact-IQ [43]**
Purposes:	Compacting images within linear quadtrees without using pointers.
Main areas of application:	Transfer through a network and manipulation of binary images being too large to be represented wholly in the primary memory.
Limitation:	Only binary images are managed, reading an image represented by a linear quadtree consists in uncoding the image and a cluster of images is store by coding each image independently.
Differences:	Only black nodes are coded in [1,2,11], whereas other encoding methods [14,19,43] code all leaf and internal nodes.
	Operations (union, intersection and difference) on coded image quadtrees are only proposed for the approaches of [2,19,43].
	According to the authors of [43], the Compact-IQ improves the compaction of images compared to other linear approaches.
Overlapping structures	**Overlapping Quadtrees [40] and Overlapped CBLQ code [18].**
Purposes:	Compacting a linear sequence of images within a structure of quadtrees.
Main areas of application:	Compaction and delivery of video.
Limitation:	Modifications are not allowed on quadtrees which are internal to the sequence: new quadtrees are created and added to the end of the sequence.
Differences:	The approach of [40] is based on hierachical quadtrees and the approach of [18], which is limited to black and white images, is based on linear quadtrees.
	In [40], reading an image is as much time consuming using overlapped quadtrees as using independent quadtrees, i.e., one quadtree for each image.
	In [18], before applying any operation on an image, the CBLQ code of the image must be rebuilt and, if the image is internal to a sequence, several preceding images must be rebuilt. Moreover, before any reading, the image must be computed from its CBLQ code.
	The compaction of images is improved by [18] in comparison with [40] because no pointer is manipulated.
Inverted structures	**Fully [7] and Dynamic [41] Inverted Quadtree and Generic Quadtree [13].**
Purposes:	Compacting a cluster of images within a single quadtree structure, allowing the querying of images by fuzzy search pattern for [7,41] and allowing the manipulation of images and the comparison of the homologous regions in different images for [13].
Main areas of application:	Content-based image retrieval for [7,41] and bookkeeping of the changes in the image processing [13].
Limitation:	Reading an image is more time consuming than using independent quadtrees.
Differences:	The structure of [7] is static ; it must be totally reorganized when the number of stored images increases.
	The approaches of [7,41] are limited to black and white images.
	Comparing homologous regions in different images is more time consuming in [7,41] than in [13] because more quadtree nodes must be read.
	In [13], additional node values (*internal* and $\perp$) are managed.
Differences between all structures	Linear approaches [11,1,14,18,19,43] do not uses pointers as opposed to the others approaches [7,13,40,41] based on hierachical quadtrees.
	The approaches based on overlapping mechanisms [18,40] manage sequences of images whereas in the other approaches managing clusters [7,13,41], there is no predefined order between images.
	Among approaches managing cluster of images, the structures proposed by [18,40] are partially persistent, whereas the structures proposed by [7,13,41] are fully persistent.
	Comparing the same node in different image quadtrees is faster using inverted structures [7,13,41] than using overlapping approaches [18,40].
	The Fully and Dynamic Inverted [7,41] Quadtrees and the Generic Quadtree [13] approaches allow many operations which do not explicitly exist for the other approaches: computing the difference between two images or parts of images, following the evolution of an area across images, etc.