Overlapping Linear Quadtrees and Spatio-Temporal Query Processing[1]

THEODOROS TZOURAMANIS, MICHAEL VASSILAKOPOULOS AND YANNIS MANOLOPOULOS

Laboratory of Data Engineering, Department of Informatics, Aristotle University, 54006 Thessaloniki, Greece

Email: theo@delab.csd.auth.gr, mvass@computer.org and *manolopo@delab.csd.auth.gr*

In this paper, indexing in spatio-temporal databases by using the technique of *overlapping* is investigated. Overlapping has been previously applied in various access methods to combine consecutive structure instances into a single structure, without storing identical sub-structures. In this way, space is saved without sacrificing time performance. A new access method, overlapping linear quadtrees is introduced. This structure is able to store consecutive historical raster images, a database of evolving images. Moreover, it can be used to support query processing in such a database. Five such spatio-temporal queries along with the respective algorithms that take advantage of the properties of the new structure are introduced. The new access method was implemented and extensive experimental studies for space efficiency and query processing performance were conducted. A number of results of these experiments are presented. As far as space is concerned, these results indicate that, in the case of similar consecutive images, considerable storage is saved in comparison to independent linear quadtrees. In the case of query processing, the results indicate that the proposed algorithmic approaches outperform the respective straightforward algorithms, in most cases. The region data sets used in experiments were real images of meteorological satellite views and synthetic random images with specified aggregation.

Received 25 February 2000; revised 21 August 2000

1. INTRODUCTION

Spatial databases (SDBs) represent, store and manipulate data of spatial types, such as points, lines, surfaces, volumes and hyper-volumes in multi-dimensional space. There are numerous applications that require efficient retrieval of spatial objects: geographical information systems (GISs), image and multimedia databases, urban planning, computer-aided design (CAD), rule indexing in expert database systems, etc. The traditional indexing methods (B-trees [1], hashing methods, etc.) are not suitable for storing spatial data because of their inability to implement a total ordering of objects in space and preserve proximity, at the same time. Since the early 1980s several structures have been proposed for spatial data in the literature. These spatial access methods (SAMs) form the following classes.

- Methods that obey an embedding space hierarchy: a region containing data is split (when a certain criterion holds) to sub-regions in a regular fashion. A representative of this class is the *quadtree* and its variations [2, 3]. Quadtrees assume a quadrangular space (a square) which is recursively split into four subquadrants.
- Methods that obey a data space hierarchy: a region containing data is split (when, for example, a maximum capacity is exceeded) into sub-regions which depend on these data only (for example, each of two sub-regions contain half of the data). The R-tree [4] and its variations are the most widely used structures of this class. R-trees organize multidimensional data objects by making use of the minimum bounding rectangles (MBRs) of the objects.

SDBs have attracted increasing interest over the last two decades. References [5, 2] are extensive surveys with detailed methodology and algorithms of a plethora of techniques for spatial data.

On the other hand, *temporal databases* (TDBs) support the maintenance of time-varying data versions and specialized queries about them. Conventional databases are not suitable to handle continuously changing data, since they can store only one version of data, the one which is applicable at the *present time*. Therefore, whenever a piece of data is no longer valid, it is either deleted or updated, at the physical level.

Two concepts of time are usually considered in TDBs, *valid* and *transaction time*. According to [6] valid time is the time during which a fact is true in the real world. Transaction time is the time during which a piece of data is recorded in a relation. 'Transaction times are consistent with the

[1] Parts of this work have been presented at the 6th ACM Symposium on Advances in Geographic Information Systems (ACM-GIS'98) and at the 3rd East-European Conference on Advanced Databases and Information Systems (ADBIS'99).

serialization order of transactions and may be implemented' as a single value by 'using transactions commit times' [6]. In terms of data modeling, these two temporal aspects are orthogonal, in that each could be recorded independently. Each of them usually comprises a *start time point* and an *end time point* or, equivalently, an *interval* [*StartTime*, *EndTime*) and has specific properties associated with it. A TDB that handles only valid time is called *valid* or *historical*; when it handles only transaction time it is called *transaction* or *rollback*; one handling both of these notions of time is called *bi-temporal* [7]. A number of access methods for temporal data have been proposed up to now. Some of these methods achieve acceptable performance in real-life applications [8].

Although, in 1977 Thrift observed that time could be considered as an additional dimension in a two- or three-dimensional space [9], until recently the fields of temporal and spatial databases remained two separate worlds. However, modern applications (GIS, time-sequence analysis and forecasting, animation, etc.) demand the efficient manipulation of spatial objects that move and/or change their shapes and/or size over time and involve relationships among them. For instance, Worboys [10] provides a survey of (mainly GIS oriented) spatio-temporal applications on administrative areas, road networks and land ownership.

Spatio-temporal databases (STDBs) are SDBs in which data objects may change their spatial locations and/or their shapes at different time intervals. In these databases, special implementation techniques should be developed for efficient storage and access of spatial objects, their geometric representations and their time-varying characteristics. Reference [11] is an excellent survey on the advances made over the last few years, in STDB research.

According to the first attempt towards a specification and classification scheme for access methods suitable for STDBs [12] and up until now, several general spatio-temporal indexing methods have appeared in the literature: 3D R-tree [13], 2+3 R-tree [14], MR-tree and RT-tree [15] and HR-tree [16]. These approaches have the following characteristics:

- 3D R-tree and 2+3 R-tree treat time as another dimension using a state-of-the-art spatial indexing method, namely the R-tree,
- MR-tree and HR-tree use overlapping in R-trees to represent successive states of the database, and
- RT-tree couples time intervals with spatial ranges in each node of the tree structure by adopting ideas from R-tree and TSB-tree [17].

In addition, several methods have been proposed to index moving points and trajectories (see [18] for a survey on these specialized methods). All these above methods are extensions of the R-tree, which is based on the 'conservative approximation principle', i.e. spatial objects are indexed by considering their MBR. These methods are not suitable for representing regional data, in cases where many empty ('dead') space is introduced in the MBRs, since this fact decreases the index ability to prune space and objects during a top–bottom traversal.

The fundamental objective of the proposed study is to present an efficient spatio-temporal access method (STAM) for a sequence of images (regional data). Efficiency is considered in terms of space requirements and time performance while processing queries. The new indexing structure that is based on the transaction time is called *overlapping linear quadtrees* and it is a variant of the linear quadtree [19, 2]. It makes use of quadcodes that do not introduce dead space to decompose and represent image data. Moreover, it supports all the well-known spatial queries for quadtree-based SDBs (spatial joins, nearest-neighbor queries, similarity and spatial selection queries, etc.) without taking time into account. It can also support efficiently all the typical temporal queries for transaction time TDBs (most of which have been examined in [20]) without considering space at all. However, the major feature of overlapping linear quadtrees is that they can efficiently handle some special types of spatio-temporal window queries for quadtree-based STDBs, not previously mentioned in the literature. The rest of the paper is organized as follows. Section 2 provides a detailed description of the new implementation. Section 3 investigates query processing in overlapping linear quadtrees. Section 4 presents experimental results regarding space requirements and query performance. Finally, section 5 concludes the paper and, also, introduces ideas for further research.

2. THE NEW STRUCTURE

2.1. Framework and assumptions

In our discussion of STDBs we assume two-dimensional space, although the presented results can be easily expanded into higher dimensions in most cases. We assume that a sequence of evolving raster images is stored in the database. Each of them is represented as a $2^n \times 2^n$ array of pixels ordered by rows, where n is a positive integer. If the pixel colors are black and white only, the image is said to be *binary*, where one represents black and zero represents white. Each image has a unique timestamp[2] T_i, where $i = 1, 2, \ldots, N$, and N is the total number of images. This temporal attribute expresses the transaction time.

A transaction time STAM implicitly associates a time interval to each record representing a spatial object. When a new record is inserted at time T_1, this time interval is set equal to [T_1, *now*[3]). A 'real-world' deletion at time point T_2 is implemented as a *logical* deletion by changing the *EndTime* timestamp of the time interval from *now* to T_2. Alternatively, an STDB could be viewed as an SDB with time as an extra dimension. But, since this dimension behaves differently from other dimensions in most applications, it is not efficient to implement such a case by simply adapting methods for high-dimensional SDBs.

[2] 'A timestamp is a time value associated with some object, e.g. an attribute value or a tuple' [6].

[3] The term *now* is a special value in TDBs [21]. Its usage means that the respective object will be valid until some time point far in the future, that is not known beforehand.

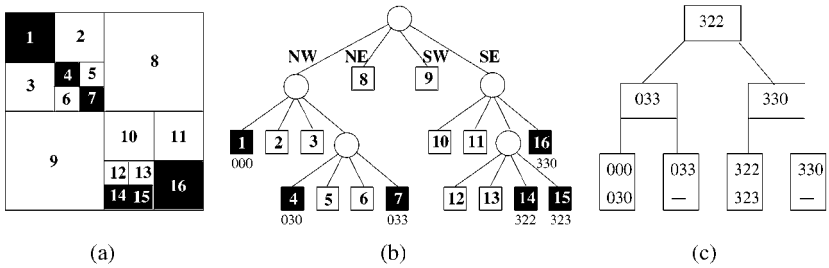

FIGURE 1. (a) A $2^3 \times 2^3$ raster image array of black and white pixels and its corresponding (b) region quadtree and (c) region linear quadtree.

2.2. Quadtrees and linear quadtrees for regional data

A *quadtree* is a term used to describe a class of hierarchical spatial data structures (first class of SAMs in the discussion of Section 1) whose common characteristic is that they are based on the principle of recursive space decomposition. A simpler example of a quadtree representation of data corresponds to the representation of two-dimensional binary raster images. It is called *a region quadtree* and is based on the successive decomposition of the images into four quadrants of $2^{n-1} \times 2^{n-1}$ pixels. If a part is not covered entirely by black or white, it is recursively subdivided into four subquadrants, until each subquadblock is entirely unicolor. The number of times that the decomposition process is applied depends on the input data, but always proceeds according to a regular scheme, until a quadblock of homogeneous color is reached. An example of a region quadtree is Figure 1b, which represents the $2^3 \times 2^3$ binary image of Figure 1a.

The root of the unbalanced tree of Figure 1b corresponds to the entire image array. Each child of a node represents a quadrant of the region corresponding to that node. The children, from left to right, correspond to the NW (Northwest), NE, SW and SE quadrants. Leaf nodes correspond to those quadblocks for which no further subdivision is necessary. Non-leaf nodes are said to be gray because their quadblocks contain both black and white pixels.

The region quadtree is a main memory structure. However, sometimes the represented image is very large and its quadtree cannot be stored in the main memory. In such a case, information on the leaf nodes that correspond to black quadblocks of the image array, can be inserted into a B^+-tree producing, thus, a pointer-less version of the quadtree. The latter method is called *linear region quadtree* (*linear quadtree* in the following). This is because its leaf nodes contain records that correspond to the linear list of the quadtree black nodes (nodes representing black quadblocks of the image).

Each black node of the quadtree is represented by a pair of numbers. The first number is the level of the quadtree at which the node is located (the root of the quadtree is said to be at level n and the pixels at level 0). The second number is termed a *locational code*. It is a base-4 number of n digits $(q_{n-1}, q_{n-2}, \ldots, q_0)$ whose values can be 0, 1, 2 or 3 corresponding to quadrants NW, NE, SW and SE, respectively. Each one of the n digits is a directional code that supports the traversal of the quadtree along a path from its root to the appropriate leaf. If the black node resides on level i, where $n \geq i \geq 0$, then the first n–i digits determine the path from the root to this node and the last i digits are all equal to 0 ('don't care').

This linear representation of the quadtree nodes is called an FD (Fixed length—Depth) linear implementation. The interested reader can find two other linear implementations in the literature: FL (Fixed Length) and VL (Variable Length), made of base-5 digits (see [3] for details). For reasons that will be explained in Section 2.4 the choice for this study was to use the FD linear implementation.

Figure 1c presents a linear quadtree which corresponds to the quadtree of Figure 1b and the binary raster image array of Figure 1a. In practice, a linear quadtree is created directly from the corresponding binary raster image. The algorithm OPTIMAL_BUILD [3] is used for converting the image to its linear FD representation. This algorithm processes the image array row by row (from top to bottom and from left to right) and produces the FD codes that are inserted one by one or in a batched manner [22] in an empty B^+-tree.

For simplicity, only the FD locational codes (*quadcodes* in the following) of the black nodes appear in the linear quadtree of Figure 1c, whereas the level at which the nodes are located is not shown. For example, node 7 (16) of the quadtree is represented by the pair of numbers 033/0 (330/1). The base-4 quadcode is 033 (330) and corresponds to the NW, SE and SE (SE, SE and 'don't care' for the last $i = 1$ digit) directions followed to reach this node from the root.

The linear quadtree reduces the necessity to store non-leaf and white leaf nodes of the pointer-based version of a quadtree. There are also variations of this representation where white nodes are also stored, or variations which are suitable for multicolored images.

2.3. The concept of overlapping

The technique of overlapping was initially presented by Burton *et al*. [23, 24]. The authors proposed the use of overlapping trees to manage the evolution of text files. Later the idea was generalized in [25] to manage temporally evolving B^+-trees. Recently, in [20], the idea was extended,

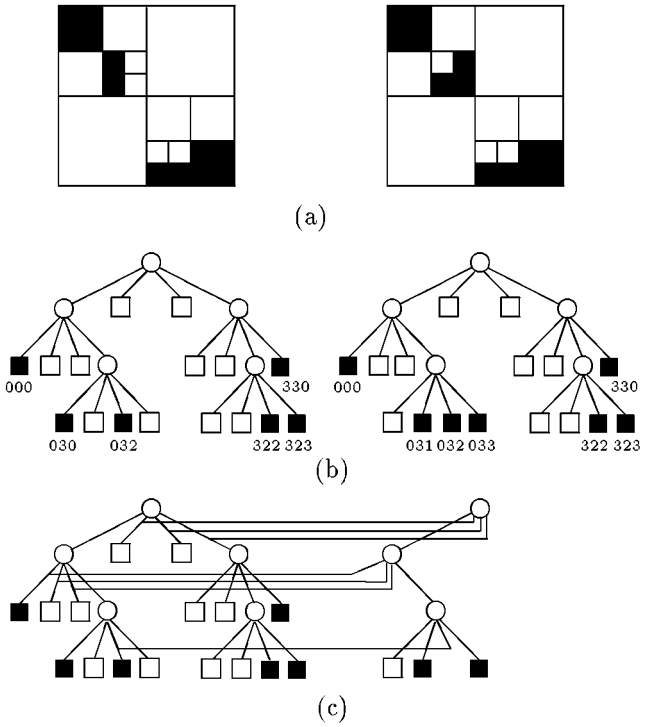

FIGURE 2. (a) Two similar binary images, (b) the respective quadtrees and (c) the corresponding overlapping quadtrees.

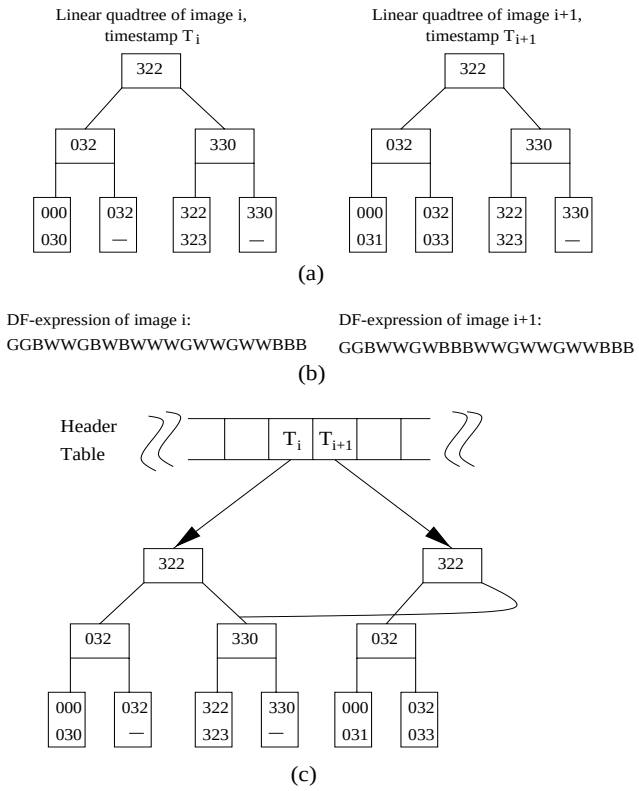

FIGURE 3. (a) Two linear quadtrees, (b) the corresponding depth-first (DF)-expressions and (c) their linear overlapping tree structure.

producing an efficient access method for transaction time databases in a two-dimensional key-time space. Also, in [15, 16] the original overlapping approach was extended from B^+-trees to R-trees, producing a spatio-temporal access structure suitable for moving spatial data.

The basic idea behind all these techniques is that, given two B^+-trees (or R-trees) where the second one is based on some changes upon the data set of the first one, the second B^+-tree (or R-tree, respectively) can be represented by registering only the modified branches of the first one. The subtrees that remain the same, are not replicated but simply re-used. Thus, each tree has a separate root and substantial space is saved.

In [26, 27] another implementation on the concept of overlapping trees is presented and analyzed. Considering a sequence of similar images, overlapping quadtrees can be used to represent this sequence and save significant main memory by not storing the common subtrees of consecutive separate quadtrees. Figure 2 demonstrates two similar binary images and the respective quadtrees. The result of applying overlapping to these trees is also shown.

2.4. Overlapping linear quadtrees

Overlapping linear quadtrees are based on linear region quadtrees. Let us assume that a sequence of N images is stored in the database and that each image has a unique timestamp T_i (for $i = 1, 2, \ldots, N$). If we use a single linear quadtree, then updates will overwrite old linear FD codes and only the last inserted image will be retained. In applications where spatial queries refer to past states of the structure, all the successive versions of the structure have to be accessible. The simplest method to store all these images is to construct one independent linear quadtree for each of them. Therefore, a search in such a database corresponds to spatial data accessing with different timestamps. However, in most cases, a large amount of spatial information remains stable between successive images and the corresponding linear quadtrees may differ only marginally. In such cases, instead of storing each complete image independently, the possibility of setting up a system for data-sharing should be explored to facilitate the storage and indexing of this data.

Overlapping linear quadtrees are a transaction time spatio-temporal indexing structure, implemented as a sequence of linear quadtrees. The first image at T_1 is stored in a linear quadtree and the image at time T_{i+1} is constructed based on the linear quadtree of T_i by sharing their common subtrees. As an example, consider the two consecutive linear quadtrees (with respect to their timestamps) of Figure 3a. They are the pointer-less versions of the quadtrees of images of Figure 2a. Since in the same quadtree two black nodes that are ancestor and descendant cannot co-exist, two linear FD codes that coincide at all the locational digits can also not exist. This means that the locational part of the FD codes is sufficient for building linear quadtrees at all the levels. At the leaf level, the depth of each black node should also be stored, so that images are accurately represented and that overlapping can be correctly applied. For simplicity, only

the FD-locational codes appear in the linear quadtrees of Figure 3a. (The locational code of each black quadtree node of the two depicted trees can be seen in Figure 2b.) The resulting overlapping version of the two linear quadtrees can be seen in Figure 3c.

Note also that the choice of the FD linear representation instead of the other two linear representations is not accidental. The FD linear representation is the only one that is made of base-4 digits and is thus easily handled using two bits for each digit, since its decoding process is much more simple [3]. Besides, the sorted sequence of FD linear codes is a depth-first traversal of the quadtree. Since internal and white nodes are omitted, sibling black nodes are stored consecutively in the linear quadtree and there is an increased probability for the same image part to reside in the same leaf between consecutive linear quadtrees.[4] This property maximizes the probability that a leaf will not change and will belong to consecutive linear quadtrees, since consecutive images have large identical parts. To make this probability even higher and keep the number of newly created paths as low as possible, the capacity in records of the linear quadtree leaf nodes is small (a few FD codes are very likely to remain unchanged).

In our implementation, the leaf node capacity was 10 FD codes, thus a disk page may host a number of consecutive leaf nodes.

Each node in the structure holds in a field called 'StartTime' the timestamp when it was created. This field is used to detect whether a node is being shared by other trees. We assign a value to StartTime during the creation of a node and there is no need for future modification of this field.

The structure of overlapping linear quadtrees is accompanied by the two following additional sub-structures:

Header table. This is built on top of the overlapping linear quadtrees in order to index transaction time values. Each record in this table is of the form $<T_i, P_i>$, where T_i is the timestamp when the respective image is recorded in the relation and P_i points to the root of the corresponding linear quadtree. For applications with a reasonable number of images (e.g. less than one billion), this sub-structure can be either a sequential array in ascending time order stored in the main memory, or a perfect hashing function, or even a B^+-tree with almost 100% utilization that has only its leaves stored in a secondary memory.

Depth-first expression. The depth-first (DF)-expression [28] of the last inserted image is kept and its use is to register all the black quadblocks of the last inserted image and to be able to know, without I/O cost, the black quadrants that are identical between this image and the next one. Thus, given a new image, we do know beforehand which exactly are the quadcode insertions, deletions and updates. The DF-expression is a compacted array that represents an image in a pointer-less form of the preorder traversal of its quadtree. It consists of the symbols 'B', 'W' and 'G' corresponding to black, white and gray nodes, respectively. It offers large compression, as each node type can be encoded with two bits. For example, assuming that sons are traversed in the order NW, NE, SW and SE, the DF-expression of the image of Figure 1 is: GGBWWGBWWBWWG-WWGWWBBB. For reasonable image sizes, it is small enough to be stored in the main memory (e.g. for images of size 2048×2048 pixels and the worst case where there is a completely full quadtree, the DF-expression is 1.33 Mbyte, which is quite reasonably available in modern workstations).

As an example to illustrate the whole data structure of the overlapping linear quadtrees, consider the two consecutive raster images of Figure 2a and their corresponding linear quadtrees of Figure 3a. Consider, also, that the image on the left half of Figure 2a is the last image i in a large sequence of similar overlapping images that have already been inserted in the overlapping linear quadtree of Figure 3c. We face the insertion of the new image i+1 (the image on the right half of Figure 2a) in two stages. The first stage is to sort the quadcodes of this image version and compare this sequence against the set of quadcodes of the last inserted image version, using the binary table of its DF-expression (see the left-hand side of Figure 3b). In the next stage, we use the header table of Figure 3c to locate the root of the last inserted image at timestamp T_i. Then, following the approach of [22], we build the new overlapping tree instance at time point T_{i+1} by carrying out all the FD code insertions, updates and deletions in a batched manner, instead of performing them one at a time. Simultaneously, the DF-expression of image $i+1$ is being constructed (see the right-hand side of Figure 3b), replacing, step-by-step, the DF-expression of image i.

A consequence of the technique for batch quadcode modifications (i.e. insertions, updates and deletions) is that a leaf may accept a number of quadcode insertions much greater than the number of available free slots. Thus, a specific leaf may split into more than two nodes. In a similar manner, more than two sibling leaves may merge during quadcode deletions.

2.5. Supplementary lists to improve query processing

In order to keep track of the image evolution (in other words, the evolution of FD codes) and efficiently satisfy spatio-temporal queries over the stored raster images, we embed some additional information in the leaf nodes of linear quadtrees. First, leaf nodes have one more extra field, called 'EndTime', to register the timestamp when a specific leaf becomes historical (no longer valid). As long as a leaf remains valid, its field EndTime is fixed to the special value *now*. Assuming that at some time point T_i at least one entry of the node changes, a new copy of that node has to be created (with <StartTime, EndTime> equal to $<T_i, now>$,

[4] Note that the depth-first (DF) traversal successively visits black quadtree nodes which correspond to potentially neighboring image parts, independently of their size. This is not usually the case for a breadth-first traversal.

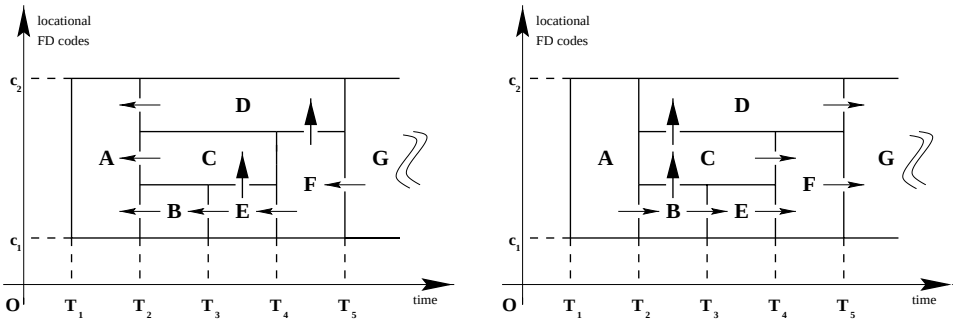

FIGURE 4. Backward (left) and forward (right) chaining for the efficient support of spatio-temporal queries.

respectively) to handle the new update. Besides, the state of the former node is changed to historical, by modifying the EndTime timestamp from *now* to T_i.

Assuming that A and B are two leaf nodes of the overlapping linear quadtrees, node A is a *temporal predecessor node* of B if the creation timestamp of node B is equal to the *logical* deletion timestamp of A and the quadcode range[5] of node B intersects the quadcode range of node A. Respectively, node B is called *a temporal successor node* of node A. As stated above, in overlapping linear quadtrees, the exact number of temporal successor nodes that a leaf node can have in the next timestamp is not fixed.

Figure 4 depicts a case where, at timestamp T_1, the leaf node A was created with quadcode range $[c_1, c_2]$. At timestamp T_2, the leaves B, C and D were created as the temporal successor nodes of node A. At timestamp T_3, leaf E was produced as the temporal successor node of leaf B. At timestamp T_4, the leaves E and C were merged, producing node F and, finally, at timestamp T_5 leaf nodes F and D were merged, producing node G.

As it appears in Figure 4 (fine and bold arrows), we incorporated some additional 'horizontal' pointers in the leaf nodes of linear quadtrees. This way there is no need to top–down traverse consecutive tree instances to search for a specific FD code and excessive page accesses are avoided. The tradeoff is a small overhead for insertions, deletions and updates. More specifically, we embed four pointers in every leaf to support spatio-temporal queries in an efficient way. The names and roles of these pointers are:

A *B-pointer* is used during a temporal query to traverse the structure backwards. When not-null, it points to a historical leaf from the previous tree instance. The node accommodating a not-null B-pointer is one of the temporal successor nodes of that historical node, after a merge/split/update (thin arrows on the left side of Figure 4).

A *BC-pointer* is also used during a temporal query to traverse the structure backwards. The node accommodating a not-null BC-pointer is always historical. The BC-pointer always refers to a historical leaf with the same *logical* deletion timestamp. This field is involved in the merge procedure (thick arrows on the left-hand side of Figure 4).

An *F-pointer* is used during a temporal query to traverse the structure forwards. The node which accommodates a not-null F-pointer is always historical. In such a case, the F-pointer points to a successive tree instant leaf created as one of the temporal successor nodes of this specific historical node, after a split/merge/update (thin arrows on the right-hand side of Figure 4).

An *FC-pointer* is also used during a temporal query to traverse the structure forwards. When this pointer is not-null, it points to a node with the same creation timestamp created after a split (thick arrows on the right-hand side of Figure 4).

[5]The *quadcode range* of a linear quadtree leaf node (a leaf node of the respective B^+-tree) is the set of quadcode values that are either contained within the specific leaf, or would be contained, if they had been inserted.

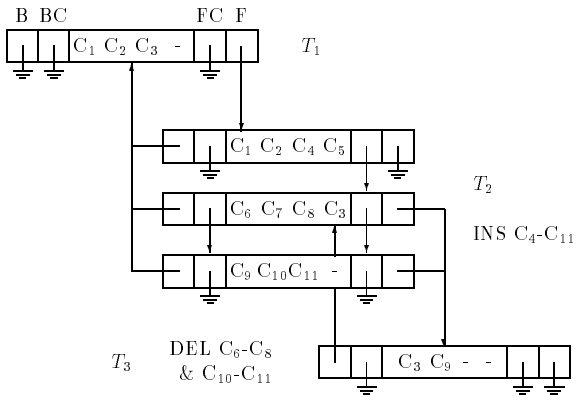

FIGURE 5. A real-word example for the forward and backward chaining of the leaf nodes of overlapping linear quadtrees.

Figure 5 shows a real-world example of how the leaves of three successive linear quadtrees can be forward and backward chained to efficiently support spatio-temporal queries. The leaf on the top left-hand corner of the figure corresponds to the first time instant, T_1, and contains the quadcodes $C_1 = 002/3$, $C_2 = 003/3$ and $C_3 = 302/3$ (the form of the codes is C/L, where C is the locational code and L is the level). Suppose that during time instant T_2, eight quadcodes with keys $C_4 = 030/3$, $C_5 = 031/3$,

C_6 = 032/3, C_7 = 200/2, C_8 = 211/3, C_9 = 330/3, C_{10} = 331/3 and C_{11} = 332/3 are inserted. In such a case, we have a node split. In other words, we have to allocate three new leaf nodes at time instant T_2, to accommodate the eleven quadcodes in total. The leaf of time instant T_1 is connected to the first of these three new nodes by using the F-pointer field, whereas the FC-pointer field is used to chain together the three new nodes. During time instant T_3, a set of five quadcodes is deleted, namely the quadcodes with key values C_6, C_7, C_8, C_{10} and C_{11}. Thus, two nodes of the tree corresponding to time instant T_2 are merged to produce a new node as depicted in the figure. B-pointer and BC-pointer fields are maintained accordingly.

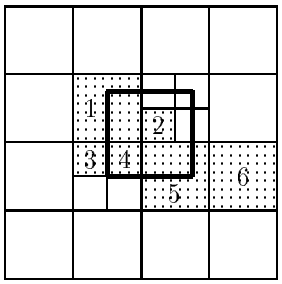

FIGURE 6. The quadblocks of a binary raster image and a query window (bold lines).

3. SPATIO-TEMPORAL WINDOW QUERY PROCESSING

As mentioned in the introduction, the structure of overlapping linear quadtrees supports all the well-known spatial queries for quadtree-based SDBs, without taking the notion of time into account. It also supports all the typical temporal queries for transaction time TDBs, without considering issues of space. A discussion on how to query only the spatial features of overlapping linear quadtrees (nearest-neighbor finding, similarity queries, spatial joins of various kinds, window queries, etc.) appears in the original papers of linear quadtree [19, 3], as no modifications to the original algorithms are required. The reader can also find interesting performance details in [20] of queries based in the temporal domain only, taking into consideration the stored quadcodes as ordinary numeric data (pure timeslice and range-timeslice queries, history queries, etc.). As already mentioned, the major advantage of the new STAM is that it can efficiently handle some special types of spatio-temporal window queries for quadtree-based STDBs, not previously mentioned in the literature.

Window queries have a primary importance since they are the basis of a number of operations that can be executed in an STDB. The basic function of window query processing is to divide the window into subwindows and therefore to decompose the query into a sequence of smaller queries onto these subwindows. Subwindows correspond to quadblocks of the leaf nodes in the quadtree that represents the interior part of the window region in the image space. These subwindows are called *maximal quadtree blocks* (*maximal quadblocks* for the rest of this paper) and the rationale for using them is to match the decomposition of the underlying quadtree-based database. A window-decomposition algorithm is given in [29]. It decomposes a two-dimensional window of $k \times k$ pixels, defined on an image of $2^n \times 2^n$ pixels into its $3(2k - \log k)$-5 maximal quadblocks in $O(k \log\log 2^n) = O(k \log n)$ time, in the worst case.

3.1. The strict containment window query

Given a $k \times k$ window and a sequence of N binary images stored in an STDB, each one associated with a unique timestamp T_i (where $i = 1, 2, \ldots, N$): find the black regions that completely fall inside the window (including those that touch a border of the window from inside), at each time point within the time interval $[T_1, T_N]$.

In Figure 6 is an example of a binary raster image (corresponding to a specific time point) partitioned in quadblocks and a query window. The strict containment window query for this time point would return quadblocks 2 and 4. The algorithm that processes this spatio-temporal window query is as follows.

Step 1. Break the $k \times k$ window into its $Q = O(k)$ maximal quadblocks, according to the algorithm described in [29].

Step 2. For each of these sub-windows[6] (sorted in lexicographical order, according to the locational part of their FD code) compute the quadcode of the pixels of their NW and SE corners. These are the smallest $Cmin_i$ and largest $Cmax_i$ locational FD codes, respectively, that may appear in the sub-window i (for $i = 1, \ldots, Q$). The range $[Cmin_i, Cmax_i]$ of these locational codes includes all the black quadblocks that are strictly included within the sub-window.

Step 3. Find the root of the linear quadtree of the very first time point using the header table.

Step 4. For each of the Q sub-windows, perform a respective range search in the first linear quadtree following a DF traversal of the tree and collect the leaves that either contain codes in the range $[Cmin_i, Cmax_i]$, or would contain them if they had been inserted. The codes that fall within the above range and appear in these nodes are the black quadblocks that are strictly contained within this sub-window for the specific time point.

Comment. The answer to the whole window query, for the first time point, is the union of all the answers generated by querying the underlying STDB with every maximal quadblock comprising the window.

Step 5. For each linear quadtree leaf node collected in the last step, following the F-pointer at the first step and the chain of FC-pointers at the second step, discover

[6] In the rest of the paper, the terms *sub-window* and *maximal quadblock* will be used interchangeably.

the leaves that are the temporal successor nodes of this leaf at the next time point.
Comment. Note that in the best case this specific leaf is common between the two successive linear quadtrees and no additional I/O access in the secondary memory is required.

Step 6. Discard from further consideration all leaves whose range does not intersect with the range $[Cmin_i, Cmax_i]$ of the respective sub-window i and collect all the remaining accessed leaves. The codes that fall within this range and appear in the remaining leaves are the black quadblocks that are strictly contained within the sub-window for the specific time point.
Comment. The answer to the whole window query for this time point, is the union of the answers generated by all the Q sub-windows.

Step 7. While the last time point of the time interval $[T_1, T_N]$ has not been reached, proceed to the next tree by repeating steps 5 and 6, for each leaf collected in the last execution of step 6.

Note that when we process the query for a tree of a specific time point, we keep in the main memory the nodes discovered and collected for this tree, as well as some of the accessed nodes of the tree of the preceding time point (only those that are common between the two successive trees). This holds for all the algorithms that are presented in this paper, except for that related to the cover window query.

3.2. The border intersect window query

Given a $k \times k$ window and a sequence of N binary images, each one associated with a unique timestamp T_i (where $i = 1, 2, \ldots, N$): find the black regions that intersect a border of the window (including those that touch the window borders from inside or outside), at each time point within the time interval $[T_1, T_N]$'.

The border intersect window query for the time point corresponding to Figure 6 would return quadblocks 1, 3, 4 and 5. The algorithm that processes this window query is as follows.

Step 1. Create a rectangular strip that is formed by keeping the pixels that make up the border of the window and the pixels outside the query window that touch its borders. For those sides of the query window that possibly touch the image borders, there are no pixels outside the window. Therefore, the resulting strip is up to two pixels thick. Break the strip into its Q maximal quadblocks. Notice that each maximal quadblock will be of size 2×2, or 1×1 pixels.

Steps 2 and 3. The same as steps 2 and 3 of the strict containment window query, respectively.

Step 4. For each of these maximal quadblocks, follow step 4 of the algorithm of the strict containment window query. The quadblocks discovered (if any) in this procedure are the black quadblocks that are strictly contained within the rectangular strip and thus intersect or touch a border of the $k \times k$ query window, for the first time point. If the search procedure for a sub-window returns no quadblocks, search for an ancestor of the specific maximal quadblock. The reason is that if such an ancestor exists, then the maximal quadblock under consideration belongs to a larger black region that intersects or touches a border of the query window. All the quadblocks discovered in step 4 compose the answer of the window query for the first time point.

Step 5. For each leaf collected during the previous step (following the respective step of strict containment window query), perform step 5 of the algorithm of the strict containment window query.

Step 6. The same as step 6 of the strict containment window query. In addition, if the search procedure for a sub-window returns no quadblocks, search for an ancestor of the specific maximal quadblock. All the quadblocks discovered (if any) in this step are the black quadblocks that intersect or touch a border of the original $k \times k$ window, for the specific time point.

Step 7. The same as step 7 of the strict containment window query.

Note that a search for an ancestor of a maximal quadblock with FD code C/L (in the form `locational_code/level`) is a search for the maximum locational code in the respective linear quadtree among the quadcodes $\leq C$. If such a quadcode exists and: (i) has a level $L' > L$ and (ii) their locational codes coincide in their first L' bits, then this FD code corresponds to an ancestor of the quadblock.

Such a search can be performed, in most cases, by accessing a very small number of extra disk pages. In more detail, the following sequence of actions is performed. In case the FD code under consideration is not the first in its node, we examine the presence of one of its ancestors in this node. Otherwise, if the previous leaf node (left sibling node) is among the nodes that reside in the main memory, we examine the presence of such an ancestor in this node. If it is not in the main memory, but the respective temporal predecessor node of the preceding tree is in main memory, we use F- and, possibly, FC-pointers to reach the previous node for the tree of interest, with very few disk accesses. If, however, none of the above holds, we have to perform a search in the tree of interest for the left sibling node, starting from the root. This search accesses a number of nodes which equals the height of the tree.

Note, also, that the ancestors of a maximal quadblock may be common to a number of subsequent maximal quadblocks (due to the order under which sub-windows are treated in steps 5 and 6). Thus, keeping in a variable the ancestor discovered (if any) for one maximal quadblock may help to avoid the repetition of the same disk accesses later in the processing of the same time point.

3.3. The general border intersect window query

Given a $k \times k$ window and a sequence of N binary images, each one associated with a unique timestamp T_i (where $i = 1, 2, \ldots, N$): find the black regions that completely

fall inside the window or intersect a border of the window (including those ones that touch a border of the window from inside or outside), at each time point within the time interval $[T_1, T_N]$.

The general border intersect window query for the time point corresponding to Figure 6 would return quadblocks 1, 2, 3, 4 and 5. It is obvious that the answer to this query is a combination of the answers generated by the previous two queries. However, the algorithmic handling of this query is not a combination of the previous two algorithms, in the sense that the prospective query processing cost is not the sum of the previous two. The algorithm that processes the general border intersect window query is as follows.

Step 1. Create a new extended window that is formed by adding to the $k \times k$ window of the query the pixels outside it that touch its border. For those sides of the query window that possibly touch the image borders there are no such outside pixels. Thus, the new extended window is of size $(k+2) \times (k+2)$ or $(k+1) \times (k+2)$ or $(k+2) \times (k+1)$ or $(k+1) \times (k+1)$ or $k \times k$ pixels. Break the new window into maximal quadblocks.

Steps 2 and 3. The same as steps 2 and 3 of the strict containment window query, respectively.

Step 4. For each of these maximal quadblocks, follow step 4 of the algorithm of the strict containment window query. The quadblocks discovered (if any) of this procedure are the black quadblocks that are strictly contained within the extended window query, and thus fall inside the original window query or intersect a border of it, for the first time point. If the search procedure for a sub-window returns no quadblocks, search for an ancestor of the specific maximal quadblock. All the quadblocks discovered in this step compose the answer of the window query for the first time point.

Steps 5–7. The same as steps 5–7 of the algorithm of the border intersect window query, respectively.

3.4. The cover window query

Given a $k \times k$ window and a sequence of N binary images, each one associated with a unique timestamp T_i (where $i = 1, 2, \ldots, N$): find out whether or not the window is completely covered by black regions, at each time point within the time interval $[T_1, T_N]$.

The cover window query returns YES/NO answers. For the time point corresponding to Figure 6, it would return NO. The algorithm that processes this kind of query is as follows.

Step 1. Break the window into maximal quadblocks.

Step 2. Find the root of the linear quadtree of the first time point, using the header table.

Step 3. For the first of the maximal quadblocks (according to the locational code of their NW corner), perform a search in the linear quadtree of the time point of interest and access (discover) the leaf that should contain the related FD code. If the code of the maximal quadblock is present in the leaf, continue. If not, then examine the FD codes in the same leaf that are before and after the code of the maximal quadblock (one of them at least exists). If these codes correspond to a sibling or a descendant of a sibling or a descendant of the quadtree node of the maximal quadblock, mark that the answer for the specific time point will be NO. The reason is that the sub-window cannot be completely covered, because the specific maximal quadblock corresponds to a quadtree node that is gray or white. However, continue processing for this time point to discover the leaves needed for the remaining time points. If the adjacent FD codes do not correspond to a sibling or a descendant, then search for an ancestor of this quadtree node. If such an ancestor does not exist, then mark NO.

Step 4. For the leaf discovered in step 3, following the F-pointer at the first step and the chain of FC-pointers at the second step, discover the leaf nodes that are the temporal successor nodes of this leaf at the next time point. Examine these leaves for the presence of the maximal quadblock of the respective sub-window, or any of its siblings, descendants or ancestors (in rare cases, a search from the root of the related tree may be needed to examine the presence of an ancestor). According to the FD codes discovered, NO may be marked for this time point.

Step 5. Repeat step 4, until you have reached the last image.

Step 6. For those images that have been marked with NO, such that all the images before or after them have also been marked with NO, the answer is definitely NO. There is no need to visit them in a subsequent stage and these images are excluded from further consideration. This means that the time interval gets smaller when we conclude that a number of subsequent images covering the left-hand or right-hand ends of the time interval $[T_1, T_N]$ have all been marked with NO.

Step 7. Next, using the header table, find the root of the first linear quadtree from the left-hand side of the time interval that has not been marked with NO until now.

Step 8. Follow step 3 again and handle the next unprocessed maximal quadblock for the remaining part of the initial time interval. The algorithm stops when, for every image that has not been excluded, all the maximal quadblocks have been processed. For those images that a NO has not been marked, the answer for the corresponding time points is YES.

Note the difference of policy from the previous three algorithms. In the cover window query we keep in the main memory only the leaves related to one maximal quadblock of the tree in process, as well as the respective leaf nodes of the preceding tree. It is evident that we must reserve a 1-bit space for holding the YES or NO answer of each image in the time interval. This approach is likely to produce NO answers for groups of images and not single images, while it avoids unnecessary disk accesses.

3.5. The fuzzy cover window query

This query can take two different forms. Given a $k \times k$ window and a sequence of N binary raster images, each one associated with a unique timestamp T_i (where $i = 1, 2, \ldots, N$):

(1) find out whether or not the percentage of the window area that is covered by black regions is larger than a given threshold, at each time point within the time interval $[T_1, T_N]$, or alternatively
(2) find out the percentage of the window area that is covered by black regions, at each time point within the time interval $[T_1, T_N]$.

The second kind of fuzzy cover window query for the time point corresponding to Figure 6 would return as answer 80%. The answer of the first kind would depend on comparison of 80% with the threshold given. The algorithm that processes this kind of query is similar to the algorithm for the general border intersect window query. The main difference is that it works on the original window (no expansion is necessary). Besides, for each quadblock discovered, the area of the part of the quadblock that falls within the window must be added to the area of the window covered by black regions.

The first form of the fuzzy cover window query returns answers YES/NO in a way analogous to the cover window query. YES or NO answers are also possibly produced in groups of images. In order to make this processing more efficient, without making extra I/O disk accesses, we can calculate incrementally both the white and the black percentages of the window. In certain cases, the calculation of the white percentage may lead us more rapidly to mark a NO answer.

Note that the cover window query may be considered as a case of the first form of the fuzzy cover window query, where the threshold equals 100%. Thus, we conclude that this type of query can also be processed with an algorithm similar to the cover window query. The drawback for using this algorithm is that during its execution, from one maximal block to the next, many previously accessed leaf nodes have to be accessed again and again, at different time points. We believe that this shortcoming could be minimized with the use of a carefully designed cache policy.

Note, also, that the second form of the fuzzy cover query may also be formulated as follows. Given a $k \times k$ window and a sequence of N binary images, each associated with a unique timestamp T_i, where $i = 1, 2, \ldots, N$: determine whether or not the black and/or white color exists inside the window, at each time point within the time interval $[T_1, T_N]$. Depending on the black percentage of the window area, only the black/white or both colors may exist inside the window for the corresponding time point.

3.6. General comment for window query algorithms

Alternative naive approaches for answering the above spatio-temporal queries are easy to devise. The respective algorithms would perform a suitable range search for all the trees that correspond to the given time interval as if all of them were separately stored, starting from the respective roots. These alternative approaches would not take into account the 'horizontal' pointers that link leaves of different trees and they are expected to have significantly worse I/O performance.

Moreover, all the presented sophisticated algorithms can be easily transformed to work backwards by starting from the end of the time interval and by using the B- and BC-pointers.

4. EXPERIMENTS

4.1. Preliminaries

The structure of overlapping linear quadtrees was implemented in the C++ language and all the experiments were performed on a Pentium II PC. As already mentioned in Section 2.4, in order to maximize overlapping, the capacity of leaf nodes was fixed to 10 FD codes. Thus, a disk page could host a number of consecutive linear quadtree leaves. We performed experiments for page sizes equal to 1 and 2 Kbytes (1K and 2K). For 1K (2K) page size, the capacity of internal nodes of linear quadtrees was 124 (252) keys and the size of each leaf was $\frac{1}{12}$ ($\frac{1}{24}$) of a page.

The evolving images were synthetic and real raster binary images of sizes: 256×256, 512×512 and 1024×1024 pixels. For the experiments with the synthetic (real) images, the number of evolving images was $N = 2$ ($N = 26$). Thus, the header table had a very small size and was stored in the main memory.

For every insertion of a new image (for converting it from raster to linear FD representation) in overlapping linear quadtrees, we used the algorithm OPTIMAL_BUILD described in [3]. At the start of every experiment, the FD codes of the first image were inserted in an empty B^+-tree. The codes were inserted one at a time, as they were produced by OPTIMAL_BUILD. Thus, we obtained the result of a typical linear quadtree with average storage utilization equal to $\ln 2$. The FD codes of the next image were inserted in the second linear quadtree, so that the identical subtrees between the two trees would overlap.

The FD codes of the second image (and of each of the remaining images) were firstly sorted in increasing order, so that an algorithm which performs batch modifications (i.e. insertions, deletions and updates) along the lines of [22] could be used. There was no I/O cost for black quadrants that were identical between the two consecutive images, since, by keeping the quadcodes of the last inserted image in a compact array as a DF-expression, we were able to sort out the respective identical FD codes. Note that the size of the DF-expression, for a 256×256 image is 21.3 Kbytes, for a 521×512 image 85.3 Kbytes, whereas for a 1024×1024 image it is 341.3 Kbytes in the worst case. Therefore, in any case it is small enough to be stored in the main memory.

4.2. Storage requirements

In the experiments of this subsection, the size of the images was fixed to 256×256 pixels. Every experiment was

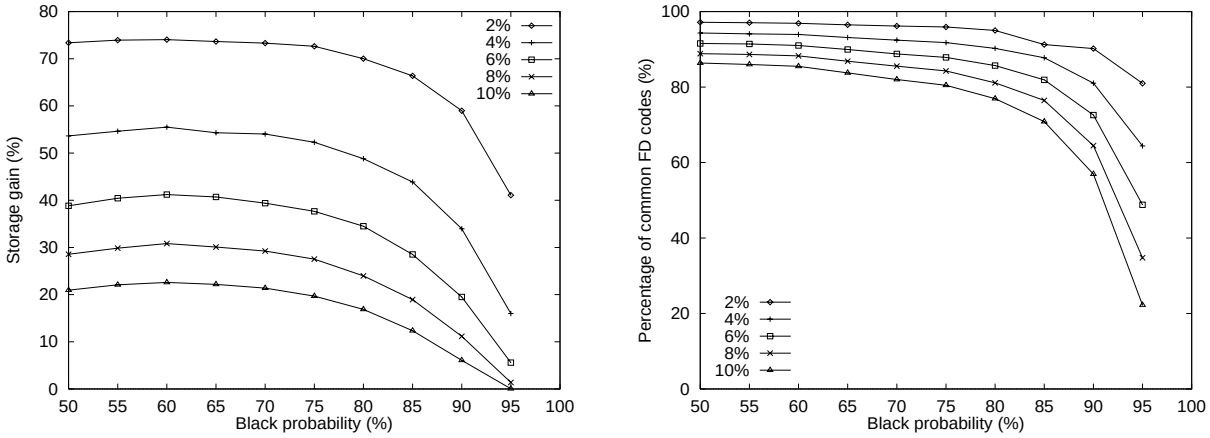

FIGURE 7. First experiment: the storage gain (left) and the percentage of common FD codes (right), as functions of the black probability, for various image differences.

repeated 50 times using a pair of similar synthetic random images. The main goal was to count the average storage gain measured as a ratio of the number of common linear quadtree nodes between the two images over the total number of the linear quadtree nodes of the second image, as if the latter image was physically stored. Due to the number of repetitions of every experiment, the results obtained converge to the average storage gain of a large sequence of evolving images (stored in the new structure). The parameters that varied in our experiments were as follows.

- *The black probability*, i.e. the percentage of the black pixels (from 50% up to 95%) for the creation of the first random image. Note that a random image is not considered very realistic when the black probability does not differ significantly from 50%, since the image created includes very small regions covered entirely by black or white and corresponds to an almost full quadtree.
- *The aggregation coefficient*, $agg(i)$ of an image i. This quantity was defined and studied in [30] and expresses the coherence of regions of homogeneous colors, of the image. Starting from a random image and using the algorithm presented in [30], an image with exactly the same black probability and higher aggregation (more realistic) can be created.
- *The image difference*, i.e. the percentage of pixels changing value from the first image to the next one (from 2% to 10%). Note that the random changing of single pixels is an extreme method of producing evolving images and the results produced by this policy should be seen as very pessimistic. In practice, much higher storage gains are expected.

4.2.1. First experiment

This experiment concerned random images of various black probabilities, the aggregation of which remained unchanged. Each image was randomly changed and overlapped with its changed version. The left-hand side of Figure 7 depicts the average storage gain as a function of the black probability of the first image of each pair, for various image differences. The right-hand side of this figure depicts the average percentage of common FD codes between the images of each pair. In most cases, the storage gain is significant and varies between 20% and over 70%, according to the image difference.

The abrupt decrease of the common disk space after the 85% black probability is explained by the fact that images with 85% black pixels and higher form many large and solid black spatial regions ('islands'). Thus, when we change the color of the 2% (for instance) of the pixels of this image in order to produce the second image, many of these islands are fragmented, producing different quadblocks. As a result, it is evident that all the curves of Figure 7 converge to point $(100, 0)$.

4.2.2. Second experiment

This experiment concerned random images of various black probabilities, the aggregation of which was increased by various amounts. Each synthetic image was randomly changed and overlapped with its changed version. Since a large number of results were produced, Figure 8 depicts, for two cases only (on the left for 60% black probability and on the right for 80% black probability), the storage gain as a function of aggregation for various image differences.

Figure 9 is based on the data produced by the same experiment. The difference in this figure is that in each curve the black probability was stable (while in Figure 8 in each curve the image difference was stable). Again, for two cases only (for image difference equal to 2%, left and to 8%, right), the storage gain as a function of aggregation for various black probabilities is depicted. In this experiment, the storage gain is also significant and, in most cases, varies between 15% and 70%, according to the image difference. The decrease of the percentage of the common disk-space inversely to the increase of the coefficient of the aggregation took place for the same reason as was described in the first experiment.

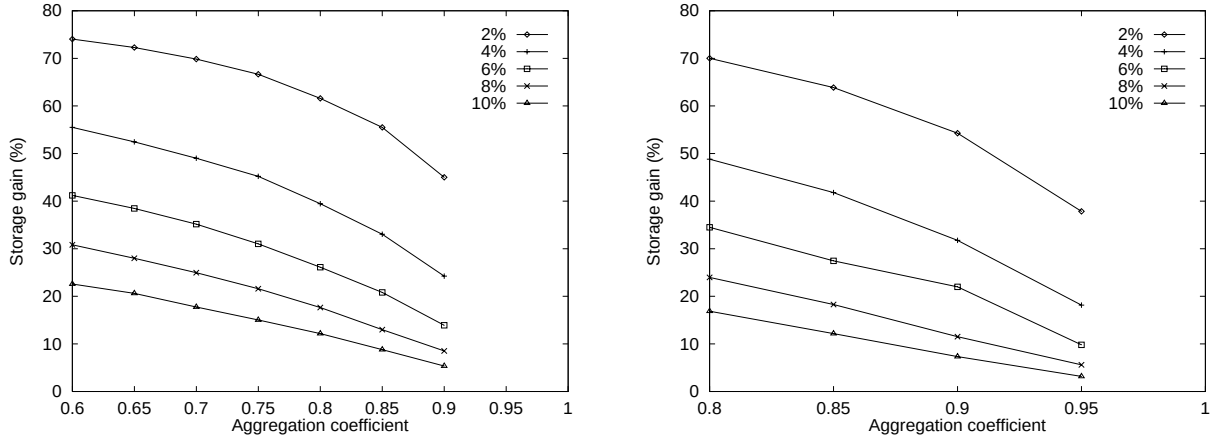

FIGURE 8. Second experiment: the storage gain as a function of aggregation, for various image differences. The images were 60% black (left) and 80% black (right).

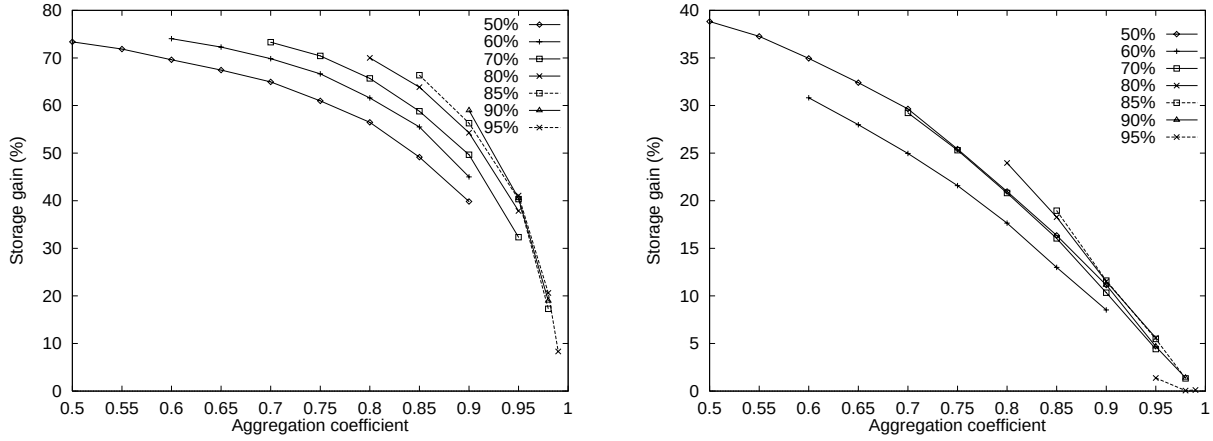

FIGURE 9. Second experiment: the storage gain as a function of aggregation, for various black probabilities. The image difference was 2% (left) and 8% (right).

4.3. Query processing

4.3.1. Experiments with synthetic data sets

The sizes of the binary random images were set to 256×256 or to 512×512 pixels. Every experiment was repeated 10 times using a pair of similar images. At the start, the first image was created with a specific black probability and an aggregation coefficient $agg()$ that was increased at various amounts. After the insertion of the first image, according to the procedure described in Section 4.1, a linear quadtree with an average node occupancy equal to $\ln 2$ was produced. This image represents the last image in a large sequence of overlapping images. Next, the second image was created again by randomly changing the color of a given percentage of the pixels of the first image. Finally, the FD codes of that image were compared with those of the previous image and inserted in the second linear quadtree which overlaps.

Windows of sizes equal to 32×32, to 64×64 or to 128×128 pixels were queried against the structure produced. Each of the five spatio-temporal window query algorithms was executed 10 times for a randomly positioned window on the image space. Thus, since every experiment was executed 10 times, each of the algorithms was run $10 \times 10 = 100$ times. Besides, for each window, the respective naive algorithms were executed (the algorithms that perform independent searches through roots, as if all linear quadtrees were independently stored). In each run, we kept track of the number of disk reads needed to perform the query for the linear quadtree corresponding to the second image. The reason why we excluded the query processing cost for the first image from the measurement is that in this linear quadtree both algorithms would perform the same range search, starting from its root and obtaining the same I/O processing cost. We are interested only in the I/O cost profit we can gain by the use of the horizontal pointers. In this case also, due to the number of repetitions of every experiment, the results obtained converge to the average I/O cost profit of a large sequence of evolving images (stored in the new structure). In the following, various experimental results are depicted by assuming that the image difference is 2%. Except for Figure 10, the black pixel probability of the first image is always fixed to 70%.

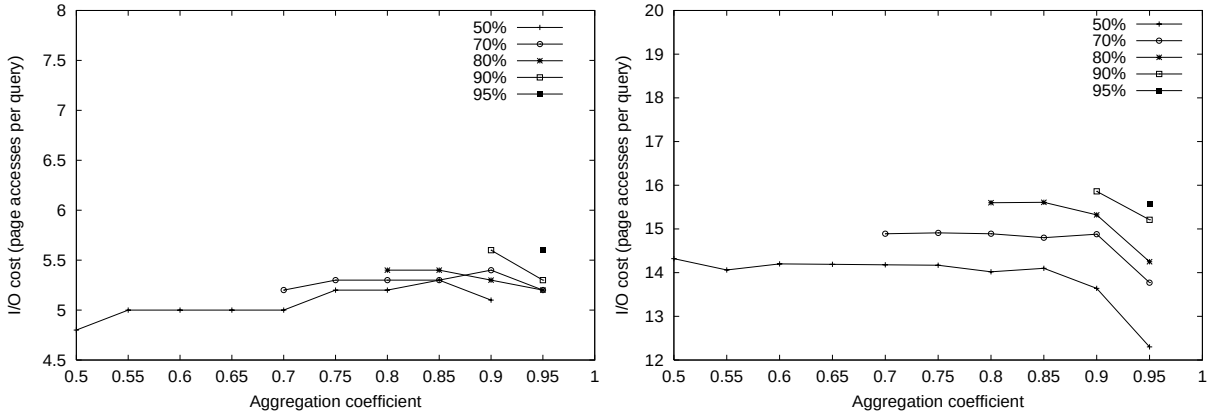

FIGURE 10. The I/O efficiency (for 1K pages) of the strict containment window query, as a function of aggregation, for various black probabilities.

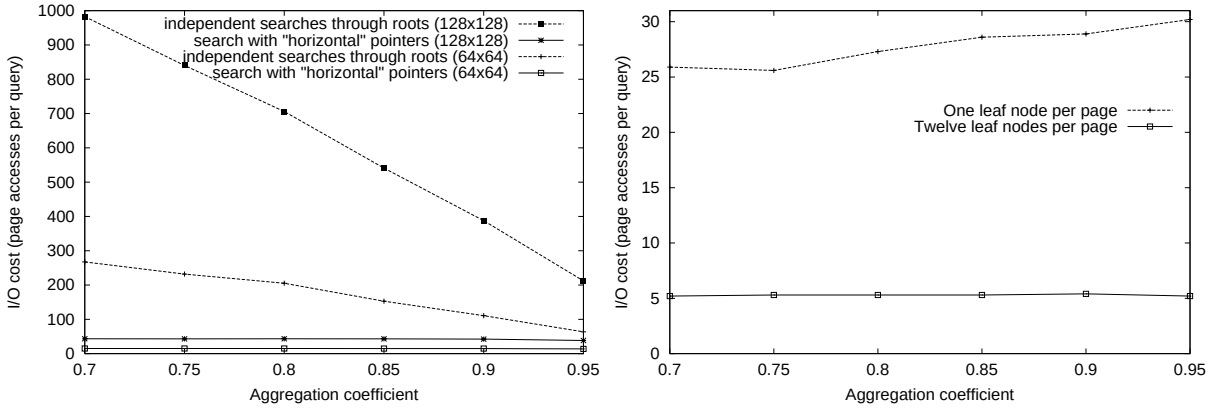

FIGURE 11. The I/O efficiency of the strict containment window query, as a function of aggregation, for two algorithmic approaches (left) and for one or 12 leaf nodes per disk page of size 1K (right).

In Figure 10, for the sophisticated algorithmic approach of the strict containment window query, one can see the number of page accesses as a function of aggregation for various black probabilities of the first image. In the left (right)-hand side of the figure, the image size is 256×256 (512×512), the window size is 32×32 (64×64) pixels and the page size is 1K. The decrease of the query I/O cost after the 0.90 aggregation coefficient is explained by the fact that images with the 0.90 aggregation coefficient and higher, form many large and solid black islands. Thus, increasing the aggregation coefficient, the size of the linear quadtrees of the produced images decreases and the query I/O cost also decreases. This phenomenon is more visible as the linear quadtrees are larger.

The left-hand side of Figure 11 shows, for the same window query and 512×512 image, the number of page accesses as a function of aggregation for the naive and the sophisticated algorithmic approaches. The black probability is 70% and the page size is again 1K. Results (different plots) for windows of the size of 64×64 and 128×128 pixels are depicted, showing a considerable improvement of the approach that uses the F- and FC-pointers to process the spatio-temporal query.

The right side of Figure 11 that concerns the sophisticated approach of the strict containment window query, shows that there is a significant improvement in the performance, when each 1K disk page holds 12 leaf nodes instead of a single one. However, this figure indicates that this I/O improvement cost is not directly proportional to 12, i.e. the number of leaf nodes that a 1K disk page may hold. This is because, as it was expected, not all the 12 stored leaves of an accessed page are useful for the answer of a query. It is also evident from the figure, that the query processing cost for the sophisticated approach is independent of the value of aggregation coefficient $agg()$ and thus independent of the coherence of regions of homogeneous colors in the images. This is the result of two reverse factors that neutralize each other. Both factors are related to the aggregation coefficient of the corresponding images. Increasing the aggregation coefficient of the first image, the solid black spatial regions in it become larger. Thus, as we also claimed earlier, the size of the linear quadtrees of the produced images is expected to decrease. On the other side, when we change the color of the 2% of the pixels of this image in order to produce the second image, many of these large black islands are fragmented, producing many different quadblocks and a larger linear

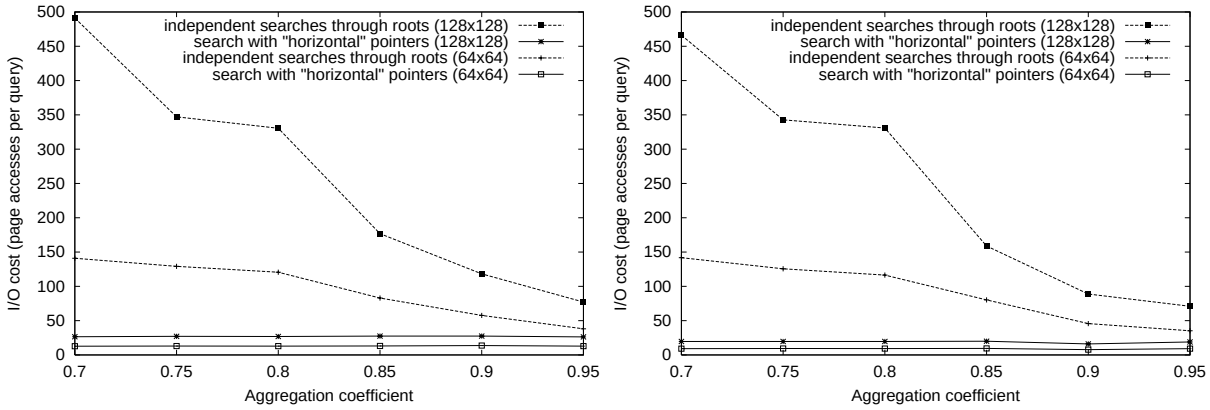

FIGURE 12. The I/O efficiency of the border intersect window query for page size of 1K (left) and 2K (right), as a function of aggregation.

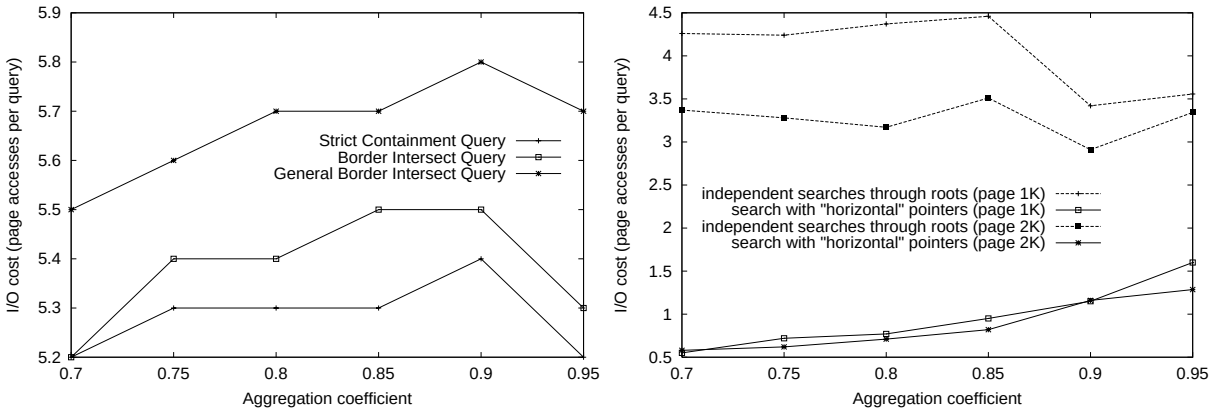

FIGURE 13. Analytical performance comparison of the strict containment, border, and general border intersect window queries for page size of 1K (left) and the I/O efficiency of the cover window query for two algorithmic approaches (right), as a function of aggregation.

quadtree for the second image. The result is that the size of the linear quadtree of the second image is finally expected to be almost the same for different values of the aggregation coefficient. Therefore, the I/O cost needed to perform the query for the second image is also expected to be about the same for different values of the aggregation coefficient. This conclusion holds for all the experiments we made with synthetic data sets. The image and the window sizes for the experiments concerning the right-hand side of Figure 11 were 256×256 and 32×32 pixels, respectively.

Figure 12 sets out the I/O processing cost for the border intersect window query and two algorithmic approaches (the naive and the sophisticated). The image size is 512×512 pixels whereas the page size equals 1K on the left-hand side and 2K on the right. It is apparent that the I/O cost difference between the two page sizes is very small even if the height of the corresponding linear quadtrees is not the same.

In the left-hand side of Figure 13 we compare the I/O efficiency of the sophisticated approach for the strict containment, border and general border intersect window queries for images with 256×256 pixels, query window with 32×32 pixels and 1K disk pages. It is shown that the I/O processing cost for the first two queries is almost the same, since the corresponding linear quadtrees are quite small and the respective algorithms access almost the same pages in each of them. The important conclusion of this figure is that, as expected and analyzed in Section 3.3, the I/O cost for the process of the general border intersect window query is not the sum of the other two ones, but only slightly greater than the greater of them.

The right-hand side of Figure 13 refers to the cover window query: one can see the number of page accesses as a function of aggregation for the naive and the sophisticated algorithmic approaches. The image and window sizes are 512×512 and 64×64 pixels, respectively. Results (different plots) for page sizes equal to 1K and 2K are depicted. Note that, since the cover window query is of YES/NO type and the intelligent exclusion of group of images from further consideration was used (see Section 3.4), the number of node accesses is extremely small.

A general remark from the diagrams in which the naive and the sophisticated approaches of Section 3 are compared is that the use of horizontal pointers leads to significantly higher I/O efficiency for all the five spatio-temporal window queries.

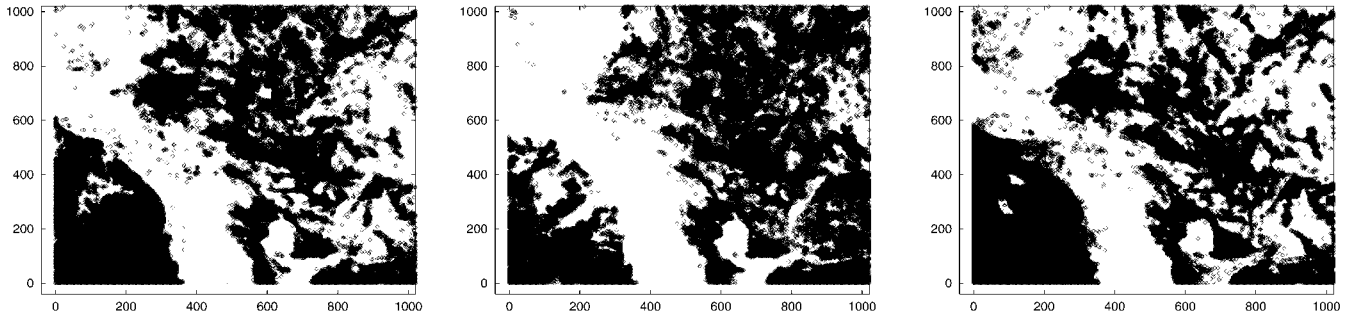

FIGURE 14. Three successive images (those with transaction time values of 24, 25 and 26, respectively) of the visible spectral channel for a 60% black pixel probability.

4.3.2. Experiments with real data sets

In the following we provide the results of some experiments based on real raster images that may be acquired via an anonymous FTP, from ftp://s2k-ftp.cs.berkeley.edu/pub/sequoia/benchmark/raster/. The images were meteorological views from the area of California, which were directly derived from the Sequoia satellite. They correspond to three different categories of spectral channels: visible, reflected infrared and emitted (thermal) infrared. Originally, each 8-bit pixel of these images represented a value in a scale of 256 tones of gray. We transformed each image to black and white, by choosing a threshold accordingly, so as to achieve a requested black probability. This probability ranged between 20% and 80%. The size of the evolving binary images was fixed to 1024×1024 pixels and the total number was $N = 26$ in every channel. Therefore, time point values varied from $T_1 = 1$ to $T_N = 26$.

Every group of N meteorological satellite images corresponds to a particular 2-week interval of the same area (almost one image every 13 h) and, thus, it is self-evident that there must appear many differences from image to image. Figure 14 depicts three successive images of the visible spectrum and confirms this fact. Table 1 shows that from the comparison of the 26 consecutive images of the spectral channel, the average percentage of pixels changing value from each image to the following one is from 12.5% to 21.2%, depending on the average black probability of the images. The same and even worse holds for the other two sequences of images, that of the reflected infrared and the thermal infrared spectral channels. Thus, it could be argued that the specific images are not the most suitable data to test the performance of overlapping linear quadtrees and the results produced should be seen as very pessimistic.

Note that in Section 4.2 (see Figures 8 and 9) the experiments with synthetic images showed clearly that if the image difference as a percentage of the pixels is more than 10%, whereas the average aggregation coefficient of the images is very high (like the image data in Table 1), then the average storage gain of storing these images in overlapping linear quadtrees is expected to be negligible. Indeed, the last row of Table 1 indicates that this is true and that the experiments with synthetic region data sets led to an exact prediction. In the case of inserting the sequence of these images into overlapping linear quadtrees the average storage gain from tree to tree is less than 0.53%. Thus, the N linear quadtrees are almost all physically stored and overlapping linear quadtrees cannot trade on their major advantage: that of sharing sub-trees whose spatial information remains unchanged between successive timestamps in order to obtain the smaller possible space occupancy and faster responses in queries that involve space and time. However, the query performance results we obtained with these real images were encouraging in such a worst-case environment. Much better query performance is expected in other, more suitable, spatio-temporal applications.

TABLE 1. Values of different parameters (in per cent) from the experiment of 26 consecutive images of the visible spectral channel. The average values in $N = 26$ images.

Black probability	20	40	60	80
Image difference	16.22	21.22	18.77	12.49
Aggregration coefficient	89.70	92.38	95.55	98.18
Storage gain	0.18	0.40	0.44	0.52

Real-world examples of such applications include the storage and manipulation of data of meteorological phenomena (e.g. atmospheric pressure zones or icebergs as they change and move over time), of faunal phenomena (e.g. movements of populations of animals/birds/fish), of the urban environment (e.g. areas covered by buildings as they change over time), of natural catastrophes (e.g. the evolution of fire or flood), etc. In every case, the data sets must include sequences of images where there are limited differences between consecutive images (e.g. successive images of digital video or satellite views of the same area, shot with small time intervals between them).

The page size in the following experiments was fixed at 1K since the height of the trees was rather low. Windows with sizes of 4×4, 16×16, 32×32, 64×64, 128×128 and 256×256 pixels were queried against the structure produced. Each of the five sophisticated spatio-temporal window query algorithms executed 50 times for each window size and in a random window position every time. For each window position, the respective naive algorithms were also executed.

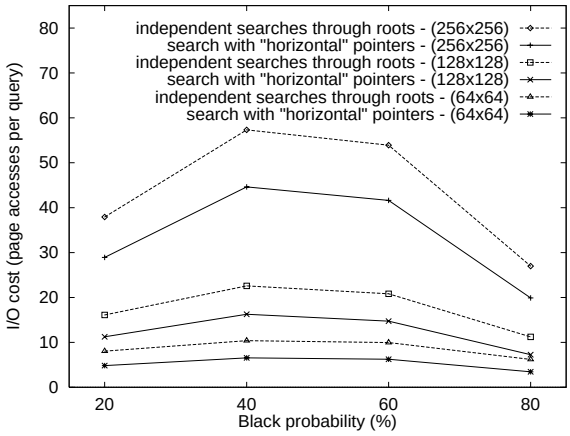

FIGURE 15. The I/O activity of the strict containment window query as a function of the black probability of the evolving images.

In each run, we kept track of the average number of disk reads needed to perform the query per linear quadtree. For a more effective comparison of the two different algorithmic approaches, we excluded from the measurement the number of disk reads spent for the very first image of the sequence of the N images. As already mentioned in the experiments with synthetic images, the reason is that for the first image, both algorithms would perform the same range search in the corresponding linear quadtree, starting from its root and, thus, accessing the same number of disk pages. It is important to highlight that we are only interested in the I/O cost profit we achieve by the use of horizontal pointers for the images 2 to 26.

Below, various experimental results are set out. In Figure 15, for the strict containment window query, one can see the average number of page accesses per linear quadtree as a function of black probability of the images for the naive and the sophisticated algorithmic approaches. The window sizes are 64×64, 128×128 and 256×256 pixels. The main conclusion that can be drawn from this figure is the following. Although Table 1 pointed out that the choice of the specific images to test the behavior of overlapping linear quadtrees was very bad, the sophisticated algorithm always outperforms the corresponding naive one in a percentage that is about 25% for 256×256 windows, 30% for 128×128 and approaches 45% for the 64×64 windows. This, of course, is due to the fact that the sophisticated algorithm does not access internal nodes in the linear quadtrees of images 2–26 (see Section 3.1). For the naive algorithm the following holds. The larger the window size, the smaller the ratio of the internal nodes accessed against the corresponding leaf nodes. This is the reason that the average gain of the algorithm that does not access internal nodes, decreases as the window becomes larger and larger. The graph shows also that the average I/O activity per tree is higher when the images are 40% or 60% black than when they are 20% or 80% black. The reason is that the average number of corresponding FD codes produced by the images in the first two cases is higher than in the other two.

The presentation of graphs for the border and general border intersect window queries is omitted from this subsection, since these experiments did not give plot behaviors that differed from those in strict containment window query. The left-hand side of Figure 16 illustrates the performance of overlapping linear quadtrees for the cover window query and the two algorithmic approaches (the naive and the corresponding sophisticated approach) for various window sizes and images with black pixel probability of 20%. It appears that the average query processing cost per tree is almost stable, regardless of the increase of the window size. This is because, even if the window is 32×32 or 256×256 pixels, the black probability is very low (20% on average). Thus, both algorithms conclude in answer NO and terminate, with very few accesses.

The right-hand side of Figure 16 for the same window query indicates that the performance of the sophisticated approach depends significantly on the black pixel probability and the aggregation coefficient (the corresponding values are shown in Table 1) of the images. In contrast, the corresponding naive algorithm is not, in practice, influenced by this. It accesses few pages every time, indicating a linear behavior. The reason that the naive algorithm for the cover window query outperforms the sophisticated algorithm is due to the way that these two algorithms work during the query process. The algorithm presented in Section 3.4, after the window decomposition into maximal quadblocks, searches from linear quadtree to linear quadtree (by the use of F- and FC-pointers) to process the query for this maximal quadblock for the whole time interval. After finishing this task, the process continues with the next maximal quadblock and follows the same procedure. This procedure will probably lead us to read many already accessed pages per linear quadtree, since most of the time two consecutive maximal quadblocks have neighboring FD codes. For this reason, for the experiments of Figure 16, we accommodated the sophisticated algorithm with a least-recently-used buffer of N–1 pages, in the sense that there was one buffered page for every linear quadtree in the interval $[T_2, T_N]$. A general conclusion for the cover window query is that when the linear quadtrees do not share much of their stored spatial information, then the naive algorithm outperforms the corresponding algorithm of Section 3.4 and, therefore, it is recommended to the database designer as the preferred solution.

The left-hand side of Figure 17 illustrates the performance of the overlapping linear quadtrees in the fuzzy cover window query when images are 40% black and the window size varies from 64×64 to 256×256 pixels. Recall that this query can take two different forms. The algorithm that uses the F- and FC- (or B- and BC-) pointers is exactly the same in both forms of the query, following the algorithmic behavior of the general intersect window query. However the corresponding naive algorithms differ. The naive algorithm of the second form will search, for every maximal quadblock produced by the query window, in all the linear quadtrees. However, the naive algorithm of the first form of the query will search only for those maximal quadblocks that are

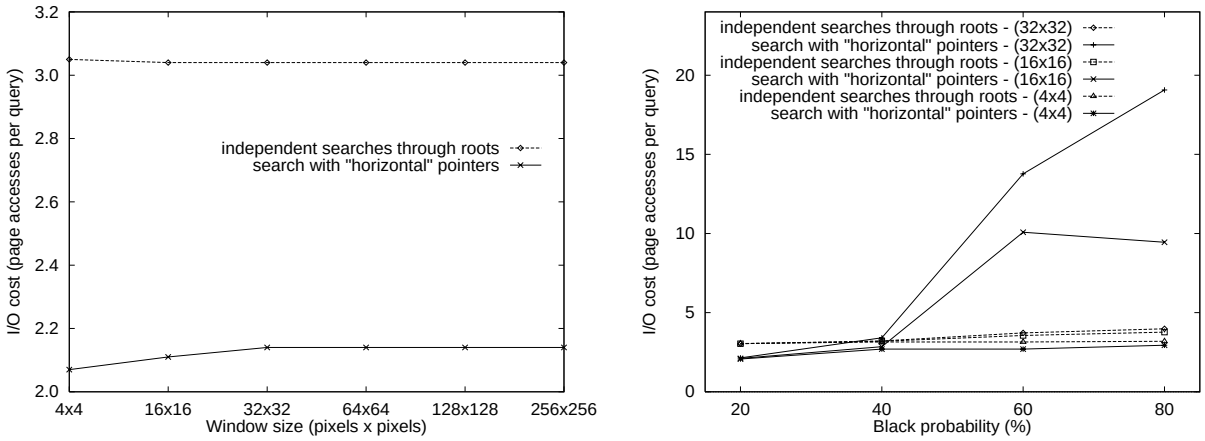

FIGURE 16. The I/O activity of the cover window query as a function of the window size for two different algorithmic approaches (left) and of the black probability of the evolving images (right).

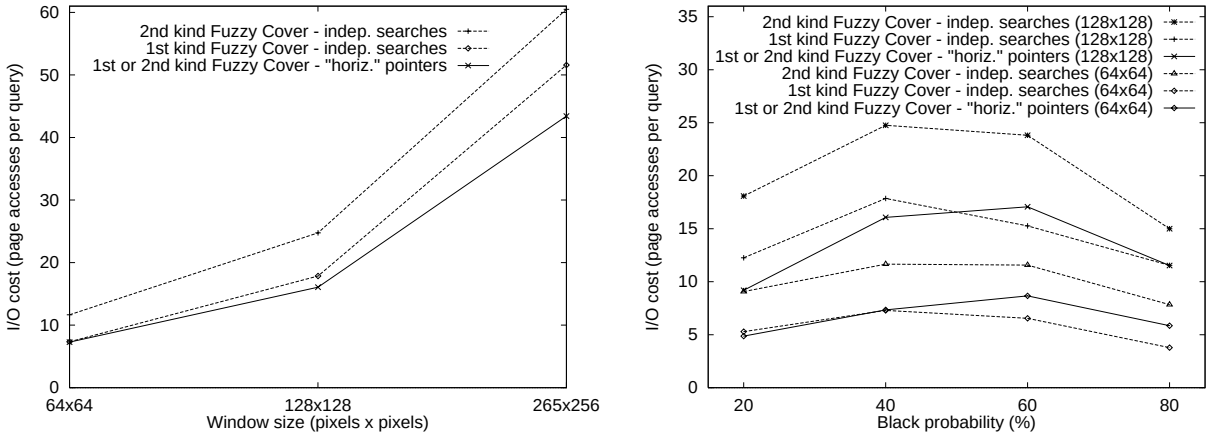

FIGURE 17. The I/O activity of the fuzzy cover window query as a function of the window size (left) and of the black probability of the evolving images (right), for different algorithmic approaches.

enough to conclude whether the answer is YES or NO for each linear quadtree, depending on the threshold given by the query. As derived from the figure, for the case of the first (second) form of the query, the sophisticated algorithmic approach is always superior of the naive one, in a percentage that varies from 1% to 10% (30% to 38%, respectively).

The right-hand side of Figure 17 demonstrates the performance of the above three algorithmic approaches (the two different naive approaches and the sophisticated one) in the fuzzy cover window query, for two different window sizes, as a function of the black probability. It is evident that for the case of the second form of the query, the sophisticated approach outperforms from 25% to 36% for the 128×128 window size and up to 52% for the 64×64 window size. For the first form of the query, in most cases this approach is also effective depending on the black probability of the evolving images. Therefore, for the first kind of fuzzy cover window query, it is a responsibility of the database designer to choose the algorithm that makes use of horizontal pointers or the corresponding naive approach.

5. CONCLUSIONS

In the present paper, we proposed a new spatio-temporal structure: overlapping linear quadtrees. This access method is based on transaction time and can be used as an index mechanism for a database of consecutive raster images. Experimentation with synthetic regional data revealed that considerable storage is saved in comparison to the independent linear quadtrees that are used to store these images. Five efficient algorithms for processing temporal window queries in an image database organized with overlapping linear quadtrees were also presented. It was thus demonstrated that this structure can be used in STDBs to support query processing of evolving images. More specifically, we introduced algorithms for processing the following spatio-temporal queries: strict containment, border intersect, general border intersect and cover and fuzzy cover window queries. In addition, we presented experiments performed to study the I/O efficiency of these algorithms. The latter experiments were based on real and synthetic sequences of evolving images. In general, our

experiments showed clearly that, thanks to the presence of 'horizontal' pointers in the leaf nodes of overlapping linear quadtrees, our sophisticated algorithms are very efficient in terms of disk activity.

In the future, we plan to develop algorithms for other new spatio-temporal queries (such as spatio-temporal joins, as well as spatio-temporal nearest-neighbor queries) in the context of overlapping linear quadtrees, and examine their performance. Along the same line, we plan to examine other quadtree-based STAMs and study their behavior. Preliminary results have been published in [31], where the structure of a multiversion B-tree (see [32]) is extended by accommodating quadcodes instead of ordinary numeric data. In addition, we believe it would be significant to extend the methods presented to grayscale and/or colored images. Another area of further research would be to perform experiments based on artificial similar images produced by shifts, rotations, scaling and other transformations, or experiments based on real images from various real-life applications.

ACKNOWLEDGEMENT

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

REFERENCES

[1] Bayer, R. and McCreight, E. (1972) Organization and maintenance of large ordered indexes. *Acta Informatica*, **1**, 173–189.

[2] Samet, H. (1990) *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA.

[3] Samet, H. (1990) *Applications of Spatial Data Structures*. Addison-Wesley, Reading, MA.

[4] Guttman, A. (1984) R-trees—a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf.*, Boston, MA, pp. 47–57. ACM Press, New York.

[5] Gaede, V. and Guenther, O. (1998) Multidimensional access methods. *ACM Comput. Surveys*, **30**, 123–169.

[6] Jensen, C. S. *et al.* (eds) (1994) A consensus glossary of temporal database concepts. *ACM SIGMOD Record*, **23**, 52–64.

[7] Kumar, A., Tsotras, V. J. and Faloutsos, C. (1998) Designing access methods for bi-temporal databases. *IEEE Trans. Knowledge Data Eng.*, **10**, 1–20.

[8] Saltzberg, B. and Tsotras, V. (1999) A comparison of access methods for time evolving data. *ACM Comput. Surveys*, **31**, 158–221.

[9] Thrift, N. (1977) *An Introduction to Time Geography*. Geo-Abstracts, London.

[10] Worboys, M. F. (1994) A unified model for spatial and temporal information. *Comput. J.*, **37**, 26–34.

[11] Abraham, T. and Roddick, J. F. (1999) Survey of spatio-temporal databases. *Geoinformatica*, **3**, 61–99.

[12] Theodoridis, Y., Sellis, T., Papadopoulos, A. and Manolopoulos, Y. (1998) Specifications for efficient indexing in spatiotemporal databases. *Proc. 7th Conf. on Statistical and Scientific Database Management Systems (SSDBM'98)*, Capri, Italy, pp. 123–132. IEEE Press.

[13] Theodoridis, Y., Vazirgiannis, M. and Sellis, T. (1996) Spatio-temporal indexing for large multimedia applications. In *Proc. 3rd IEEE Conf. on Multimedia Computing and Systems (ICMCS'96)*, pp. 441–448.

[14] Nascimento, M. A., Silva, J. R. O. and Theodoridis, Y. (1999) Evaluation for access structures for discretely moving points. In *Proc. Int. Workshop on Spatio-Temporal Database Management (STDBM'99)*, Edinburgh, UK, pp. 171–188. Springer-Verlag.

[15] Xu, X., Han, J. and Lu, W. (1990) RT-tree—an improved R-tree index structure for spatiotemporal databases. In *Proc. 4th Int. Symp. on Spatial Data Handling (SDH'90)*, Zurich, Switzerland, pp. 1040–1049.

[16] Nascimento, M. A. and Silva, J. R. O. (1998) Towards historical R-trees. In *Proc. ACM Symp. on Applied Computing (ACM-SAC'98)*, Atlanta, GA, pp. 235–240. ACM Press.

[17] Lomet, D. and Salzberg, B. (1993) Transaction time databases. In Tansel, A. *et al.* (eds), *Temporal Databases: Theory, Design and Implementation*, Redwood City, CA, pp. 388–417. Benjamin/Cummings.

[18] Sellis, T., Koubarakis, M. *et al*. (eds), *Spatiotemporal Databases: The Chorochronos Approach*. Book in preparation.

[19] Gargantini, I. (1982) An effective way to represent quadtrees *Commun. ACM*, **25**, 905–910.

[20] Tzouramanis, T., Manolopoulos, Y. and Lorentzos, N. (1999) Overlapping B^+-trees: an implementation of a temporal access method. *Data Knowledge Eng.*, **29**, 381–404.

[21] Clifford, J., Dyreson, C., Isakowitz, T., Jensen, C. S. and Snodgrass, R. T. (1997) On the semantics of 'NOW' in temporal databases. *ACM Trans. Database Syst.*, **22**, 171–214.

[22] Lang, S. D., Driscoll, J. R. and Jou, J. H. (1986) Improving the differential file technique via batch operations for tree structured file organizations. In *Proc. 2nd IEEE Int. Conf. on Data Engineering (ICDE'86)*, Los Angeles, CA, pp. 524–532. IEEE Press.

[23] Burton, F. W., Huntbach, M. W. and Kollias, J. (1985) Multiple generation text files using overlapping tree structures. *Comput. J.*, **28**, 414–416.

[24] Burton, F. W., Kollias, J. G., Kollias, V. G. and Matsakis, D. G. (1990) Implementation of overlapping B-trees for time and space efficient representation of collection of similar files. *Comput. J.*, **33**, 279–280.

[25] Manolopoulos, Y. and Kapetanakis, G. (1990) Overlapping B^+-trees for temporal data. In *Proc. 5th Jerusalem Conf. on Information Technology (JCIT'90)*, Jerusalem, Israel, pp. 491–498. IEEE Press.

[26] Vassilakopoulos, M., Manolopoulos, Y. and Economou, K. (1993) Overlapping quadtrees for the representation of similar images. *Image Vision Comput.*, **11**, 257–262.

[27] Vassilakopoulos, M., Manolopoulos, Y. and Kroell, B. (1995) Efficiency analysis of overlapped quadtrees. *Nordic J. Comput.*, **2**, 70–84.

[28] Kawaguchi, E. and Endo, T. (1980) On a method of binary picture representation and its application to data compression. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, **2**, 27–35.

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

[30] Manolopoulos, Y., Nardelli, E., Proietti, G. and Vassilakopoulos, M. (1995) On the generation of aggregated random spatial regions. In *Proc. 4th Int. Conf. on Information and Knowledge Management (CIKM'95)*, Washington, DC, pp. 318–325. ACM Press, New York.

[31] Tzouramanis, T., Vassilakopoulos, M. and Manolopoulos, Y. (2000) Multiversion linear quadtrees for spatio-temporal data. In *Proc. 4th East-European Conf. on Advanced Databases and Information Systems (ADBIS-DASFAA'2000)*, Prague, Czech Republic, pp. 279–292. Springer-Verlag.

[32] Becker, B., Gschwind, S., Ohler, T., Seeger, B. and Widmayer, P. (1996) An asymptotically optimal multiversion B-tree. *VLDB J.*, **5**, 264–275.