Multiversion Linear Quadtree for Spatio-Temporal Data ⋆

Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos

Data Engineering Lab
Department of Informatics, Aristotle University
54006 Thessaloniki, Greece
theo@delab.csd.auth.gr, mvass@computer.org, manolopo@csd.auth.gr

Abstract. Research in spatio-temporal databases has largely focused on extensions of access methods for the proper handling of time changing spatial information. In this paper, we present the Multiversion Linear Quadtree (MVLQ), a spatio-temporal access method based on Multiversion B-trees (MVBT) [2], embedding ideas from Linear Region Quadtrees [4]. More specifically, instead of storing independent numerical data having a different transaction-time each, for every consecutive image we store a group of codewords that share the same transaction-time, whereas each codeword represents a spatial subregion. Thus, the new structure may be used as an index mechanism for storing and accessing evolving raster images. We also conducted a thorough experimentation using sequences of real and synthetic raster images. In particular, we examined the time performance of temporal window queries, and provide results for a variety of parameter settings.

1 Introduction

Spatial Databases (SDBs) represent, store and manipulate spatial data, such as points, lines, surfaces, volumes and hyper-volumes in multi-dimensional space. Numerous applications require efficient retrieval of spatial objects: geographical information systems (GIS), image and multimedia databases, urban planning, computer-aided design (CAD), rule indexing in expert database systems, etc. The traditional indexing methods are not suitable to store spatial data because of their inability to implement a total ordering of objects in space and preserve proximity, at the same time. References [5,10] 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 and specialized queries on them. Conventional databases are not suitable to handle continuously changing data, since they can store only one version of data, the one applicable at *present time*. Therefore, whenever a piece of data is not valid any longer, it is either deleted or updated, at the physical level.

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

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. Each of these two notions of time is comprised by a *start time point* and an *end time point* or, equivalently, an *interval* [*StartTime*, *EndTime*) and has specific properties associated with it. A TDB handling valid time only is called *valid* or *historical.* When it handles transaction time only, it is then called *transaction* or *rollback.* When handling both of these notions of time at once it is called *bi-temporal.* 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 [12].

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 information that change over time. *Spatio-temporal Databases* (STDBs) are spatial databases 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 [1] is an excellent survey on the advances made during the last years, in spatio-temporal database research.

The fundamental objective of the proposed study is to present an efficient spatio-temporal access method (STAM) for storing and accessing evolving raster images (regional data). Efficiency is considered in terms of space requirements and time performance during query processing. The new indexing structure that is based on transaction time is called *Multiversion Linear Quadtree* (MVLQ).

The motivation for devising this new spatio-temporal access method is the Multiversion B-tree (MVBT) [2], however the proposed method differs for a number of reasons. Instead of storing independent numerical data having a different transaction-time each, for every consecutive image MVLQ stores a group of codewords that share the same transaction-time, whereas each codeword represents a spatial subregion. As a consequence, the algorithms of insertion, deletion and update processes in MVLQ are significantly different from the corresponding algorithms in MVBT. This is due to fact that after a batch operation with many insertions, deletes and updates of data records at a specific transaction-time, we may have significant different policies in node splitting and merging.

MVLQ has analogous functionality to another significantly different structure proposed by the authors, Overlapping Linear Quadtrees (OLQs) [14,15,17]. Both structures have the same origin, Linear Region Quadtree (LRQ) [4]. However, MVLQ stores the codewords present in LRQs in a modified MVBT, while OLQs apply the technique of overlapping in a sequence of LRQs. The purpose of this article is to present the MVLQ along with an initial experimental study of the time performance of temporal window queries, and provide results for a variety of parameter settings. We conducted a thorough experimentation using sequences of real and synthetic raster images. A comparison with OLQs is a research activity in progress.

The rest of the paper is organized as follows. Section 2 provides a description of the new structure. Section 3 discusses query processing in MVLQ. Section 4 presents experimental results regarding space requirements and query performance. Finally, Section 5 concludes the paper introducing, also, ideas for further research.

2 The New Structure

2.1 Framework and Assumptions

In our discussion of STDBs we assume that a sequence of evolving raster images is stored in the database. Each of these images 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 a *binary* one, where 1 stands for black and 0 for white color. Each image has a unique timestamp[1] T_i, where i=1, 2, ..., N, and N is the total number of images. This temporal attribute expresses 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 $[T_1, *)$ [2]. 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 * to T_2.

2.2 Quadtrees and Linear Quadtrees for Regional Data

The *region Quadtree* is based on the successive decomposition of two-dimensional binary raster images into four quadrants of $2^{n-1} \times 2^{n-1}$ pixels. If a part of an image is not covered entirely by black or white, it is recursively subdivided into four subquadrants, until each subquadblock is entirely unicolor. An example of binary raster image arrays of pixels and their corresponding region Quadtrees appears in Fig. 1.

The region Quadtree is a main memory structure. However, the represented image may be very large and its Quadtree can not be stored in main memory. In such a case, information about the leaves that correspond to black quadblocks of the image array, can be inserted into a B$^+$-tree producing, thus, a pointerless version of the Quadtree. The latter method is called *Linear region Quadtree* (*Linear Quadtree* in the sequel, [4,10]).

Each black Quadtree node is represented by a pair of numbers (C, L). The first number C is termed *a locational code* and denotes the correct path to this node, traversing the Quadtree from its root till the appropriate leaf. Each one of the n digits of C can be 0,1,2 or 3 corresponding to quadrants NW, NE, SW and

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

[2] The symbol '*' refers to *now* which is a special value in TDBs. Its usage means that the respective object will be valid until some time point far in the future, that is not known beforehand.

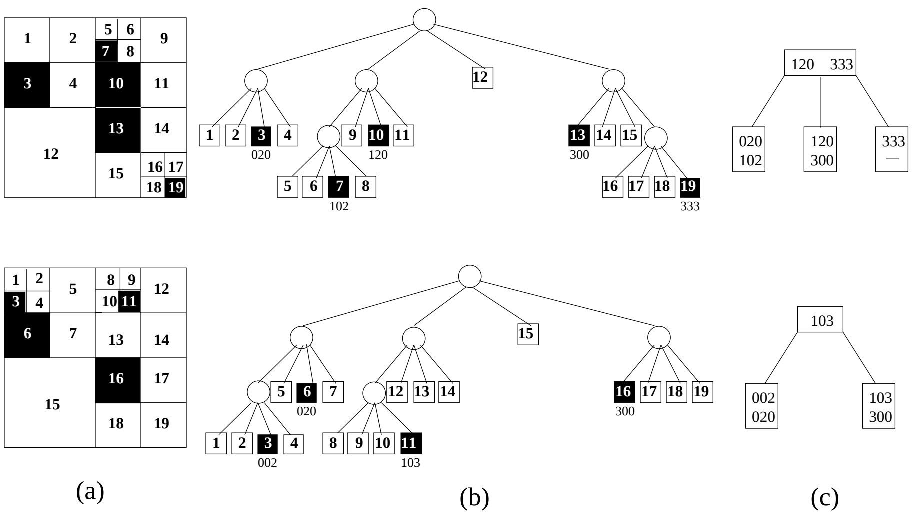

Fig. 1. Two similar binary $2^3 \times 2^3$ raster images (left) and their corresponding region Quadtrees (middle) and Linear region Quadtrees (right).

SE, respectively. The second number L of the pair is the Quadtree level where the node is located.

This linear representation of the Quadtree nodes, is called 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) (see [11] for details). For reasons that are explained in [14], the choice of the linear representation for the black Quadtree nodes was the FD (Fixed length - Depth) implementation. The right part of Fig. 1 presents two different Linear Quadtrees that can be obtained from the corresponding Quadtrees in the middle of the same figure. For simplicity, only the FD locational codes (*quadcodes* in the sequel) of the black nodes appear in the Linear Quadtrees, whereas the levels of the nodes are not shown.

2.3 Multiversion Linear Quadtree

If a sequence of N images has to be stored in a Linear Quadtree, each image labeled with a unique timestamp T_i (for i=1, 2, ..., N), then updates will overwrite old versions and only the last inserted image will be retained. In applications where spatial queries refer to the past, all the successive versions of the structure need to be accessible. MVLQ converts the ephemeral Linear Quadtree to a *persistent data structure* [3], where past states are also maintained.

MVLQ couples time intervals with spatial objects in each node. Data records residing in leaves contain records of the form $<(C, L), T>$ where (C, L) is the FD code of a black node of the region Quadtree and T represents the time interval when this black node appears in the image sequence. Nonleaf nodes contain entries of the form $<C', T', Ptr>$, where Ptr is a pointer to a descendent node,

C' is the smallest C recorded in that descendent node and T' is the time interval that expresses the lifespan of the latter node.

In each MVLQ node, we added a new field, called "*StartTime*", to hold the time instant when it was created. This field is used during the modification processes and will be examined further, later. Moreover, in each leaf we added one more extra field called "*EndTime*", to register the transaction time when a specific leaf changes and becomes historical. The structure of the MVLQ is accompanied by two additional main memory sub-structures:

– the *root* table*: it is built on top of MVLQ. MVLQ hosts a number of version trees and has a number of roots in such a way that each root stands for a time/version interval $T''=[T_i, T_j)$, where $i, j \in \{1, 2, ..., N\}$ and $i < j$. Each record in the root* table represents a root of MVLQ and obeys the form $< T'', Ptr' >$, where T'' is the lifespan of that root and Ptr' is a pointer to its physical disk address.
– the *Depth First-expression* (DF-expression, [7]) of the last inserted image: its usage is to keep track of all the black quadblocks of the last inserted image, and to be able to know at no I/O cost, the black quadrants that are identical between this image and the one that will appear next. Thus, given a new image, we do know beforehand which exactly are the FD code insertions, deletions and updates. The DF-expression is a compacted array that represents an image in the preorder traversal of its Quadtree.

As we claimed earlier the basis for the new access method is the MVBT. However, its algorithms of insertion, deletion and update processes are significantly different from the corresponding algorithms in the MVBT.

Insertion
If during a quadcode insertion, at time point T_i, the target leaf is already full, a *node overflow* occurs. Depending on the StartTime field of the leaf, the structural change may be triggered in two ways.

– If *StartTime*$=T_i$ then *a key split* occurs and the leaf splits. Assuming that b is the node capacity, after the key split the first $\lceil b/2 \rceil$ entries of the original node are kept in this node and the rest are moved to a new leaf.
– Otherwise, if *StartTime* $< T_i$, a copy of the original leaf must first be allocated, since it is not acceptable to change past states of the spatio-temporal structure. In this case, we remove all non-present (past) versions of quadcodes from the copy node. This operation is called *version split* [2] and the number of present versions of quadcodes after the version split must be in the range from $(1+e)d$ to $(k–e)d$, where k is a constant integer, $d=b/k$ and $e > 0$. If a version split leads to less than $(1+e)d$ quadcodes, then a merge is attempted with a sibling or a copy of that sibling containing only its present versions of quadcodes (the choice depends on the StartTime field of the sibling). If a version split leads to more than $(k–e)d$ quadcodes in a node, then a key split is performed.

Deletion

Given a “real world” deletion of a quadcode at time point T_j, its implementation depends on the StartTime field of the corresponding leaf.

- If $StartTime{=}T_j$ then the appropriate entry of the form $< C, L, T >$ is removed from the leaf. After this *physical* deletion, the leaf is checked whether it holds enough entries. If the number of entries is above d, then the deletion is completed. If the latter number is below that threshold, then the *node underflow* is handled as in the classical B$^+$-tree, with one difference that if a sibling exists (preferably the right one) then we have to check its StartTime field before proceeding to a merge or a key redistribution.
- Otherwise, if $StartTime < T_j$ then the quadcode deletion is handled as a *logical* deletion, by updating the temporal information T of the appropriate entry from $T{=}[T_i, *)$ to $T{=}[T_i, T_j)$, where T_i is the insertion time of that quadcode. If an entry is logically deleted in a leaf with exactly d present quadcode versions, then a *version underflow* [2] occurs that causes a version split of the node, copying the present versions of its quadcodes into a new node. Evidently, the number of present versions of quadcodes after the version split is below $(1{+}e)d$ and a merge is attempted with a sibling or a copy of that sibling.

Update

Updating (i.e. changing the value of the level L of) an FD code leaf entry at time point T_j is implemented by (i) the logical deletion of the entry and (ii) the insertion of a new version of that entry; this new version of the entry has the same quadcode C but a new level value L'.

Example

Consider the two consecutive images (with respect to their timestamps $T_1{=}1$ and $T_2{=}2$) on the left of Fig. 1. The MVLQ structure after the insertion of the first image is given in Fig. 2a. At the MVLQ leaves, the level L of each quadcode should also be stored but for simplicity only the FD-locational codes appear. The structure consists of three nodes: a root R and two leaves A and B. The node capacity b equals 4 and the parameters k, d and e equal 2, 2 and 0.5, respectively. The second version of the structure is constructed based on the first one, by inserting the FD code $< 002, 0 >$ (in the form $< C, L >$), the deletion of $< 102, 0 >$, the insertion of $< 103, 0 >$ and the deletion of FD codes $< 120, 1 >$ and $< 333, 0 >$.

Figure 2b shows the intermediate result of the insertion of FD code $< 002, 0 >$, the deletion of $< 102, 0 >$ and the insertion of FD code $< 103, 0 >$. When we attempt to insert the quadcode 103 in the leaf A of Fig. 2b, the leaf overflows and a new leaf C is created after a version split. All present versions of quadcodes of leaf A are copied into leaf C and the parent R is updated for the structural change. Leaf C holds now more than $(k{-}e)d{=}3$ entries and a key split is performed producing a new leaf D. Again, the parent R is updated.

The final status of MVLQ after the insertion of the second image is illustrated in Fig. 2c. The quadcode 120 is deleted from leaf D of Fig. 2b and a node underflow occurs (the number of entries is above d), which is resolved by merging

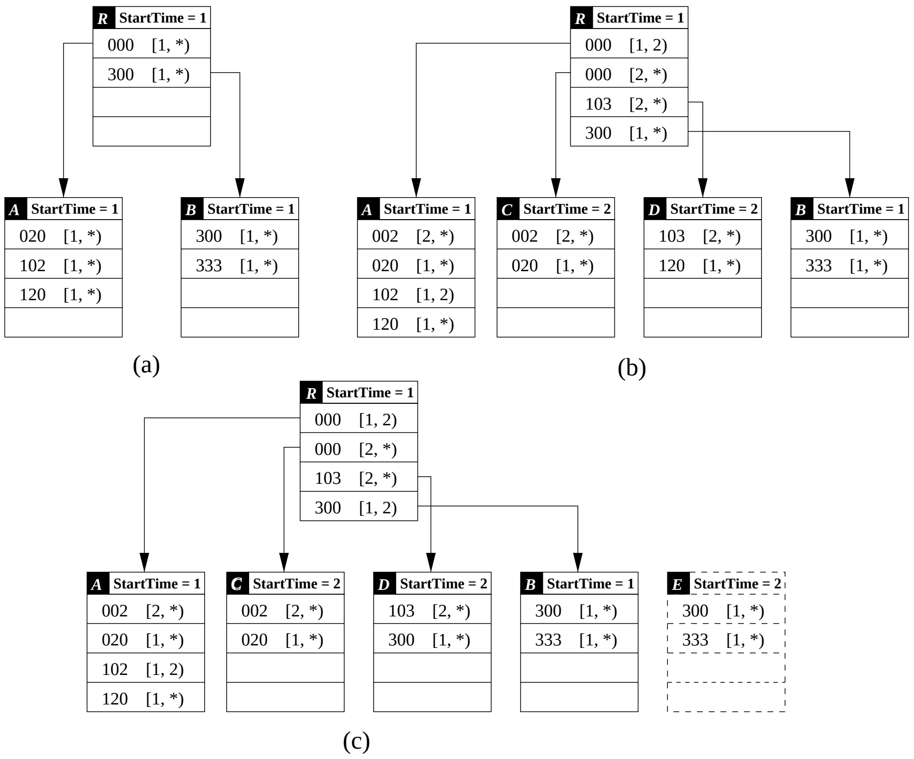

Fig. 2. (a) The MVLQ structure after the insertion of the first image, (b) a preliminary result during the insertion of the second image, and (c) the final result after the insertion of the second image.

this node with its right sibling B or a copy of it, containing only its present versions of quadcodes. After finding out that the StartTime field of leaf B is smaller than T_2, a version split on that leaf is performed, which is followed by a merge of the new (but temporary) leaf E and leaf D, in leaf D. The process terminates after the physical deletion of quadcode 333 from leaf D. The final number of entries in leaf D equals d. Both versions of MVLQ (Fig. 2a and Fig. 2c) have the same root R, although in general, more than one roots may exist.

Generally, we face the insertion of a new image in two stages. The first stage is to sort the quadcodes of the new image and compare this sequence against the set of quadcodes of the last inserted image, using the binary table of its DF-expression. Thus, there is no I/O cost for black quadrants that are identical between the two successive images. During the next stage we use of the root* table to locate the root that corresponds to the last inserted image. Then, following ideas of the approach of [8], we build the new tree version by performing all the quadcode insertions, updates and deletions in a batched manner, instead of performing them one at a time. (We did not follow this approach in the example of Fig. 2 for simplicity reasons). It is obvious that after a batch operation with insertions, deletions and updates at a specific time point, we may have conceptual node splittings and mergings. Thus, a specific leaf may split in more than two nodes and in a similar manner, more than two sibling leaves may merge during FD code deletions.

3 Spatio-Temporal Window Query Processing

The MVLQ structure is based on transaction time and is an extension of MVBT and Linear Quadtree for spatio-temporal data. It supports all the well-known spatial queries for quadtree-based spatial databases (spatial joins, nearest neighbor queries, similarity and spatial selection queries, etc.) without taking into account the notion of time. However, the major feature of the new STAM, is that it can efficiently handle all the special types of spatio-temporal window queries for quadtree-based databases, described in detail in [15,17].

Window queries have a primary importance since they are the basis of a number of operations that can be executed in a STDB. Given a $k \times k$ window and a sequence of N binary images stored in a STDB, each one associated with an unique timestamp T_i (where i=1, 2, ..., N), we considered the satisfaction of the following queries by the use of MVLQ:

- The Strict Containment Window Query
- The Border Intersect Window Query
- The General Border Intersect Window Query
- The Cover Window Query
- The Fuzzy Cover Window Query

Definitions and algorithms for the processing of the above queries were described in [15,17] and can be applied to MVLQ with slight modifications. For brevity, the description of these modifications is not included in this report.

In order to improve spatio-temporal query processing on raster images, we added four “horizontal” pointers in every MVLQ leaf. The use of these pointers was first introduced in [13] and later it was adapted to spatiotemporal data in [14]. This way there is no need to top-down traverse consecutive tree instances to search for the history of a specific FD code and excessive page accesses are avoided. The names of these pointers are: B-pointer, BC-pointer, F-pointer and FC-pointer. Their roles and functions are described in [14].

Alternative naive approaches for answering the above spatio-temporal queries are easy to devise. The respective algorithms would perform a suitable range search for every MVLQ version that corresponds to the given time interval as if each one of them was separately stored in an LRQ, starting from the respective MVLQ roots. These alternative approaches would not take into account the horizontal pointers resulting in significantly worse I/O performance.

4 Experiments

4.1 Preliminaries

The MVLQ structure was implemented in C++ and all the experiments were performed by using the following parameter values. Assuming that the page size is 1K, the size of a time interval is 8 bytes, the size of an FD locational code as well as the size of a pointer are 4 bytes each, and the size of the level of an FD code is 1 byte, we conclude that the internal nodes of the MVLQ can accommodate 60 $< C', T', Ptr >$ entries, whereas leaves contain 75 records of the format $< (C, L), T >$. For a given node capacity b, it is useful for the time complexity to choose a large d, whereas k should be as small as possible [2]. To guarantee a good space utilization it is also useful to choose a large e. It has been proved that the maximum value of e is equal to e=1–1/d, whereas for the parameter k, the following inequality should hold [2]:

$$k \geq 2 + 3e - \frac{1}{d} \tag{1}$$

Thus, for a leaf (internal) node capacity equal to b=75 (b'=60), the values used in experimentation for the above parameters were: d=15 (d'=12), k=5, and e=0.933 (e'=0.916).

The evolving images were synthetic and real raster binary images of sizes: 512×512 and 1024×1024 pixels. For the experiments with synthetic (real) images, the number of evolving images was N=2 (N=26). The size of the DF-expression, for a 512×512 image is 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 main memory. For every insertion of a new image (for converting it from raster to linear FD representation) in the MVLQ, we used the algorithm OPTIMAL_BUILD described in [10].

We performed an extensive experimentation with respect to the storage performance. For brevity, we do not include any such results. However, the interested reader may find additional details and results in [16].

4.2 Query Processing

Each sophisticated algorithm for the five spatio-temporal window queries was executed several times for different window sizes and in a random window position every time. Besides, the respective naive algorithms were executed by performing independent searches through multiple MVLQ roots. In each run, we kept track of the average number of disk reads needed to perform the query per time point. 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. The reason is that both algorithms would perform the same range search in the corresponding tree instance, starting from its root and, thus, accessing the same number of disk pages. We are interested only in the I/O cost profit we can succeed by the use of the horizontal pointers.

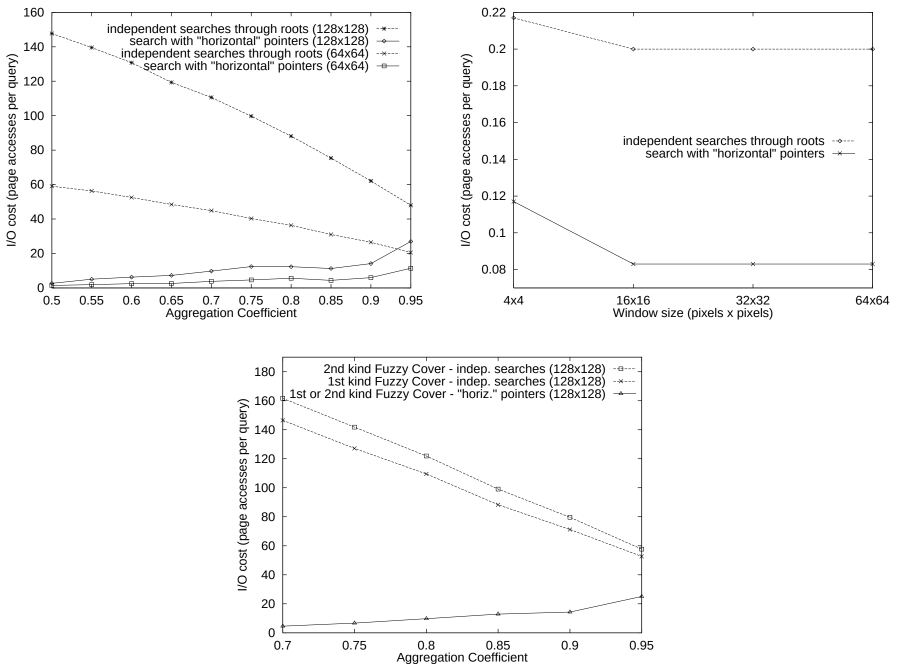

Fig. 3. The I/O efficiency of the Strict Containment (upper left) of the Cover (upper right) and Fuzzy Cover Window Query (lower middle).

Experiments with Synthetic Data Sets

Every experiment was repeated 10 times using a pair of similar images. In the beginning, the first image was created with a specific black/white analogy and an aggregation coefficient $agg()$ that was increased at various amounts. The quantity $agg()$ has been defined in [9] and expresses the coherence of regions of homogeneous colors in an image. Starting from a random image and using the algorithm presented in [9], an image with exactly the same black/white analogy and higher aggregation (more realistic, including larger regions covered entirely by black or white) can be created. After the insertion of the first image, the second image was created by randomly changing the color of a given percentage (2%) 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 MVLQ. Note that the random changing of single pixels is an extreme method of producing evolving images and the results derived by this policy should be seen as very pessimistic. In practice, much higher performance gains are expected. Windows of sizes ranging from 4×4 to 128×128 pixels were queried 10 times each against the structure produced. Thus, every algorithm was executed 10×10 times.

The performance of the sophisticated and the naive algorithms is illustrated in Fig. 3. The upper left (upper right) part shows the I/O cost of the Strict

Containment (Cover) Window Query as a function of the aggregation coefficient (of the window size) for 50% black images (70% black images and aggregation coefficient equal to 0.7). The lower middle part shows the I/O cost of the Fuzzy Cover Window Query for 80% threshold and 70% black images, as a function of the aggregation coefficient. The linear decrease of the I/O cost for the naive algorithms of the Strict and Fuzzy Cover Window Queries is explained by the fact that images with larger aggregation form larger and solid black spatial regions ("islands") and thus the corresponding Linear Quadtree holds less number of FD codes. An interesting remark is that the use of horizontal pointers leads to a remarkably high and stable I/O performance for all the spatio-temporal window queries examined.

Experiments with Real Data Sets

In the sequel, we provide the results of some experiments based on real raster images, which were meteorological views of California, and may be acquired via anonymous FTP from `ftp://s2k-ftp.cs.berkeley.edu/pub/sequoia/benchmark/raster/`. These images correspond to three different categories of spectral channels: visible, reflected infrared, and emitted (thermal) infrared. Originally, each 8-bit image pixel represented a value in a scale of 256 tones of gray. We transformed each image to a binary one, by choosing a threshold accordingly, so as to achieve a black analogy ranging between 20% and 80%. The total number of evolving binary images was N=26 in every channel, and, therefore, the time point values varied from T_1=1 to T_N=26.

Figure 4 depicts three successive images of the visible spectrum. Table 1 shows that from the comparison of the 26 consecutive images of the visible 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 percentage of the black/white analogy. Thus, many differences appear from image to image (Fig. 4 confirms also this fact) and it could be argued that the specific images are not the most suitable data to test the performance of MVLQ and the results produced should be seen as very pessimistic.

It is self-evident that the larger the image difference in percentage of pixels, the worse the query time performance of the sophisticated algorithmic approa-

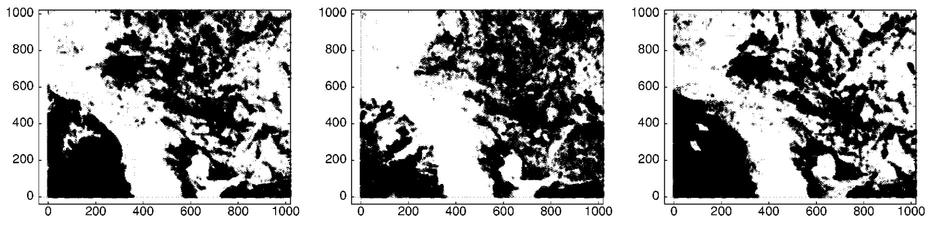

Fig. 4. Three successive 60% black images.

Table 1. Values of different parameters came up from the experimentation with the 26 consecutive images of the visible spectral channel.

	Average values in N=26 images			
Black analogy	20%	40%	60%	80%
Difference	16.22%	21.22%	18.77%	12.49%
Aggr. coefficient	89.70%	92.38%	95.55%	98.18%

ches. However, the query performance results we obtained with these real images were encouraging in such a worst case environment.

Windows of sizes ranging from 4×4 to 256×256 pixels were queried 50 times each against the structure produced. 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.

The upper left (upper right) part of Fig. 5 depicts the time performance of the Strict Containment (Fuzzy Cover) Window Query as a function of black analogy (and threshold 80%), for two different window sizes. The lower middle part presents a general performance comparison of the I/O cost of the sophisticated algorithms for the four different window queries and window size 128×128. Again, a general remark from the diagrams, is that the use of horizontal pointers leads to significantly higher I/O efficiency for all the spatio-temporal window queries examined.

5 Conclusions

In the present paper, we proposed a new spatio-temporal structure: Multi-Version Linear Quadtree. This access method is based on transaction time and can be used as an index mechanism for consecutive raster images. Five efficient algorithms for processing temporal window queries were also adapted to an image database organized with MVLQ. It was demonstrated that this structure can be used in spatio-temporal databases to support query processing of evolving images. More specifically, we studied algorithms for processing the following spatio-temporal queries: Strict Containment, Border Intersect, General Border Intersect, Cover and Fuzzy Cover Window Queries. Besides, we presented experiments performed for studying 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 "horizontal" pointers in the MVNQ leaves, our algorithms are very efficient in terms of disk activity.

In the future, we plan to compare the space and time performance of MVLQ to those of OLQs. We also plan to develop algorithms for other new spatio-temporal queries that take advantage of MVLQ, OLQs and other Quadtree-based STAMs and study their behavior. Moreover, we plan to investigate the possibility of analyzing the performance of such algorithms. Also, it is considered important

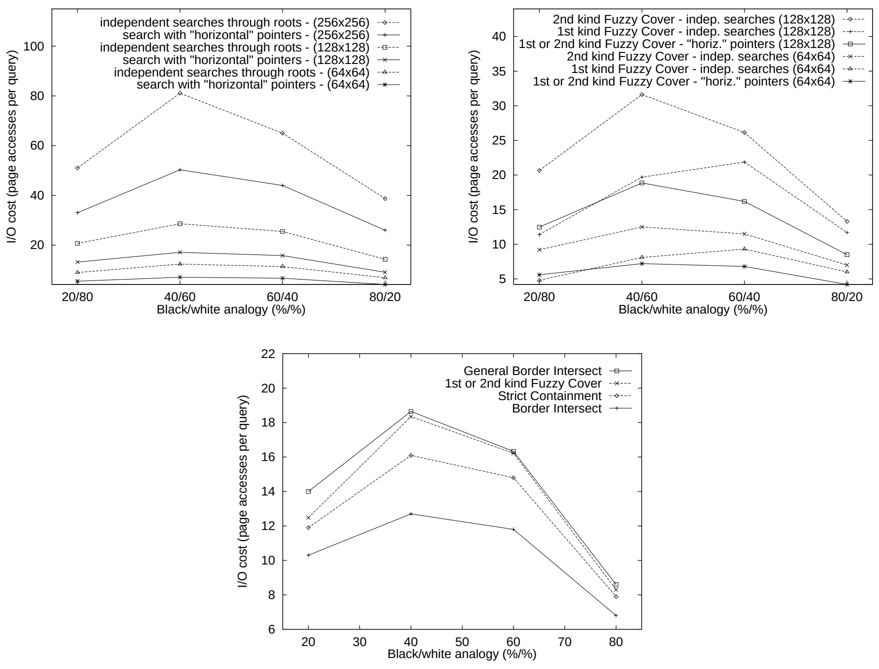

Fig. 5. The I/O efficiency of the Strict Containment (upper left), the Fuzzy Cover Window Query (upper right) and performance comparison of four different window queries (lower middle) as a function of the black/white analogy of the evolving images.

to examine the performance of MVLQ and OLQs in the context of various spatio-temporal operations, such as spatio-temporal joins, as well as spatio-temporal nearest neighbor queries [18].

Acknowledgments We would like to thank Prof. Bernhard Seeger of the University of Marburg for kindly providing the MVBT code.

References

1. T. Abraham and J.F. Roddick: Survey of Spatio-Temporal Databases, *Geoinformatica*, Vol.3, No.1, pp.61-99, 1999.
2. B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer: An Asymptotically Optimal Multiversion B-tree, *The VLDB Journal*, Vol.5, No.4, pp.264-275, 1996.
3. J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan: Making Data Structures Persistent, *Journal of Computer and System Sciences*, Vol.38, pp.86-124, 1989.
4. I. Gargantini: An Effective Way to Represent Quadtrees, *Communications of the ACM*, Vol.25, No.12, pp.905-910, 1982.
5. V. Gaede and O. Guenther: Multidimensional Access Methods, *ACM Computer Surveys*, Vol.30, No.2, pp.123-169, 1998.

6. C.S. Jensen et al.: A Consensus Glossary of Temporal Database Concepts, *ACM SIGMOD Record*, Vol.23, No.1, pp.52-64, 1994.
7. E. Kawaguchi and T. Endo: On a Method of Binary Picture Representation and its Application to Data Compression, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.2, No.1, pp.27-35, 1980.
8. S.D. Lang and J.R. Driscoll: Improving the Differential File Technique via Batch Operations for Tree Structured File Organizations, *Proceedings IEEE International Conference on Data Engineering (ICDE'86)*, Los Angeles, CA, 1986.
9. Y. Manolopoulos, E. Nardelli, G. Proietti and M. Vassilakopoulos: On the Generation of Aggregated Random Spatial Regions, *Proceedings 4th International Conference on Information and Knowledge Management (CIKM'95)*, pp.318-325, Washington DC, 1995.
10. H. Samet: *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading MA, 1990.
11. H. Samet: *Applications of Spatial Data Structures*, Addison-Wesley, Reading MA, 1990.
12. B. Saltzberg and V. Tsotras: A Comparison of Access Methods for Time Evolving Data, *ACM Computing Surveys*, Vol.31, No.2, pp.158-221, 1999.
13. T. Tzouramanis, Y. Manolopoulos and N. Lorentzos: Overlapping B^+-trees: an Implementation of a Temporal Access Method', *Data and Knowledge Engineering*, Vol.29, No.3, pp.381-404, 1999.
14. T. Tzouramanis, M.Vassilakopoulos and Y. Manolopoulos: Overlapping Linear Quadtrees: a Spatio-temporal Access Method, *Proceedings 6th ACM Symposium on Advances in Geographic Information Systems (ACM-GIS'98)*, pp.1-7, Bethesda MD, November 1998.
15. T. Tzouramanis, M. Vassilakopoulos and Y. Manolopoulos: Processing of Spatio-Temporal Queries in Image Databases, *Proceedings 3rd East-European Conference on Advances in Databases and Information Systems (ADBIS'99)*, pp. 85-97, Maribor, September 1999.
16. T. Tzouramanis, M. Vassilakopoulos and Y. Manolopoulos: Multiversion Linear Quadtree for Spatio-Temporal Data, Technical Report, Data Engineering Lab, Department of Informatics, Aristotle University of Thessaloniki. Address for downloading: `http://delab.csd.auth.gr/~theo/TechnicalReports`
17. T. Tzouramanis, M. Vassilakopoulos and Y. Manolopoulos: Overlapping Linear Quadtrees and Window Query Processing in Spatio-Temporal Databases, submitted.
18. Y. Theodoridis, T. Sellis, A. Papadopoulos and Y. Manolopoulos: Specifications for Efficient Indexing in Spatiotemporal Databases, *Proceedings of the 7th Conference on Statistical and Scientific Database Management Systems (SSDBM'98)*, pp.123-132, Capri, Italy, 1998.