Processing of Spatiotemporal Queries in Image Databases*

Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos

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

Abstract. Overlapping Linear Quadtrees is a structure suitable for storing consecutive raster images according to transaction time (a database of evolving images). This structure saves considerable space without sacrificing time performance in accessing every single image. Moreover, it can be used for answering efficiently window queries for a number of consecutive images (spatio-temporal queries). In this paper, we present three such temporal window queries: strict containment, border intersect and cover. Besides, based on a method of producing synthetic pairs of evolving images (random images with specified aggregation) we present empirical results on the I/O performance of these queries.

Keywords: Spatio-temporal databases and queries, transaction time, access methods, indexing, B$^+$trees, linear region quadtrees, overlapping, time performance.

1 Introduction

Several spatial access methods have been proposed in the literature, for storing multi-dimensional objects (e.g. points, line segments, areas, volumes, and hypervolumes) without considering the notion of time. These methods are classified in one of the following two categories according to the principle guiding the hierarchical decomposition of data regions in each method: data space hierarchy and embedding space hierarchy. The book by Samet [10] and the recent survey by Gaede and Guenther [3] provide excellent information sources for the interested reader.

On the other hand, temporal access methods have been proposed to index data varying over time without considering space at all. The notion of time may be of two types: transaction time (i.e., time when the fact is current in the database and may be retrieved) and valid time (i.e., time when the fact is true in the modeled reality) [4]. A temporal DBMS would support at least one of these

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

two types of time. A wide range of access methods has been proposed to support multi-version/temporal data by keeping track of data evolution over time. For excellent recent surveys on temporal access methods see [7,9].

Until recently the field of temporal databases and spatial databases remained two separate worlds. However, modern applications (e.g. geographical information systems, multimedia systems, scientific and statistical databases, such as medical, meteorological, astrophysics oriented databases) involve the efficient manipulation of moving spatial objects, and the relationships among them. Therefore, there is an emerging growing need to study the case of "spatio-temporal databases". According to the first attempt towards a specification and classification scheme for spatio-temporal access methods [13], up until the time it was written, only four spatio-temporal indexing methods had appeared in the literature: 3D R-trees [12], MR-trees and RT-trees [18], and HR-trees [8]. All these methods are extensions of the R-tree, which is based on the "conservative approximation principle", i.e. spatial objects are indexed by considering their minimum bounding rectangle (MBR). These methods are not suitable for representing regional data, in cases where a lot of 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.

In [15], a different paradigm was followed, that of quadtrees. Quadcodes were used to decompose image data in an exact (i.e. non-rough) manner. As a result, a new spatio-temporal structure was presented, named Overlapping Linear Quadtrees, suitable for storing consecutive raster images according to transaction time (a database of evolving images). This structure is based on linear quadtrees which are enhanced by using the overlapping technique in order to avoid storing identical sub-quadrants of successive instances of image data evolving over transaction time. In [15] it is shown by experimentation with synthetic regional data that the new structure saves considerable space, without sacrificing time performance in accessing every single image. Moreover, in [15] an abstract algorithm for answering temporal window queries with Overlapping Linear Quadtrees is presented.

In the present paper, we elaborate on spatio-temporal queries that can be answered efficiently with this structure. More specifically, we present three temporal window query processing algorithms: strict containment, border intersect and cover. We also report on I/O efficiency results of experiments performed with these algorithms. The experiments were based on synthetic pairs of evolving images. The first image of each pair was formed according to the model of random images with specified aggregation of black regions [6]. The second image of each pair was formed by random change of pixels, a rather pessimistic method of image changing. In real situations it is expected that the above algorithms will perform even better.

The rest of the paper is organized as follows. Section 2 describes the building blocks of the implementation of Overlapping Linear Quadtrees. Section 3 provides a detailed description of the three algorithms that use the new structure and answer spatio-temporal queries. Section 4 presents the experimental setting and reports on the I/O performance of these algorithms in terms of the number of disk assesses. Section 5 provides conclusions and suggestions for future work directions.

2 The Spatiotemporal Structure

The notion of overlapping consecutive instances of access methods has been mentioned in the previous section. Except for the cases of MR-trees and HR-trees, overlapping has been also used in a number of occasions, where successive data snapshots are similar. For example, it has been used as a technique to compress similar text files [1], B-trees and B^+trees [2,5,14], as well as main-memory quadtrees [16,17]. In this section, first we make a short presentation of region quadtrees, and second we describe the application of overlapping to secondary memory quadtree variations.

2.1 Region Quadtrees

The region quadtree is the most popular member in the family of quadtree-based access methods. It is used for the representation of binary images. More precisely, it is a degree-four tree. Each node corresponds to a square array of pixels (the root corresponds to the whole image). If all of them have the same color (black or white), then the node is a leaf of that color. Otherwise, the node is colored gray and has four children. Each of these children corresponds to one of the four square sub-arrays to which the array of that node is partitioned. For more details regarding quadtrees see [10]. Figure 1 shows an 8 × 8 pixel array and the corresponding quadtree. Note that black (white) squares represent black (white) leaves, whereas circles represent gray nodes.

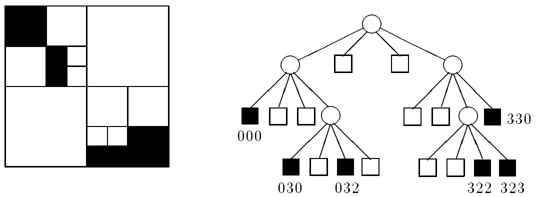


Fig. 1. An image, its Quadtree and the linear codes of black nodes

2.2 Overlapping Linear Quadtrees

Variations of region quadtrees have been developed for secondary memory. Linear region quadtrees are the ones used most extensively. A linear quadtree representation consists of a list of values where there is one value for each black node of the pointer-based quadtree. The value of a node is an address describing the position and size of the corresponding block in the image. These addresses can be stored in an efficient structure for secondary memory (such as a B-tree or one of its variations). The most popular linear implementations are the FL (Fixed Length), the FD (Fixed length – Depth) and the VL (Variable length) linear implementations [11]. In the FD implementation, the address of a black quadtree node has two fixed size parts: the first part denotes the path (directional code) to this node (starting from the root) and the second part the depth of this node. In Figure 1, one can see the directional code of each black node of the depicted tree.

Each quadtree, in a sequence of quadtrees modeling time evolving images, can be represented in secondary memory by storing the linear FD codes of its leaves in a B$^+$tree. The structure of Overlapping Linear Quadtrees is formed by overlapping consecutive B$^+$trees, that is by storing the common subtrees of the two trees only once (for more details regarding this structure, see [15]). Since in the same quadtree two black nodes that are ancestor and descendant cannot co-exist, two FD linear codes that coincide at all the directional digits cannot exist neither. This means that the directional part of the FD codes is sufficient for building B$^+$trees 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. The above part of Figure 2 depicts the B$^+$trees that correspond to two region Quadtrees and the below part depicts the resulting overlapped linear structure. Note that in Overlapping Linear Quadtrees there is no extra cost for accesses in a specific linear quadtree.

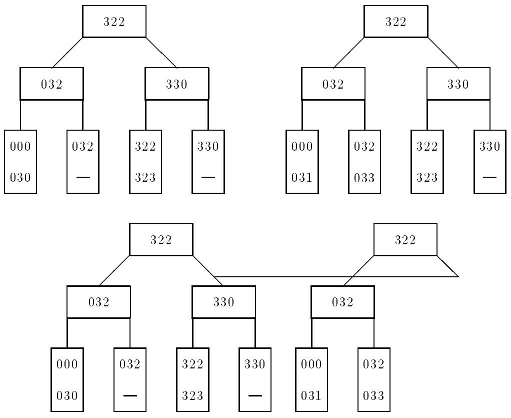


Fig. 2. Two B$^+$trees storing linear quadtree codes and the corresponding linear overlapped structure.

All nodes of Overlapping Linear Quadtrees have an extra field, called "StartTime", that can be 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. Moreover, leaf-nodes have one more extra field, called "EndTime", that is used to register the transaction time when a specific leaf changes and becomes historical.

In order to keep track of the image evolution (in other words, the evolution of quadcodes) and efficiently satisfy spatio-temporal queries over the stored raster images, we embed some additional horizontal pointers in the B$^+$tree leaves. This way there will be no need to top-down traverse consecutive tree instances to search for a specific quadcode, thus avoiding excess page accesses. More specifically, we embed two forward and two backward pointers in every B$^+$tree leaf to support spatio-temporal queries. The F-pointer of a node points to the first of a group of leaves that belong in a successive tree and have been created from this node after a split/merge/update. The FC-pointers chain this group of leaves together. The B and BC pointers play analogous roles when traversing the structure backwards.

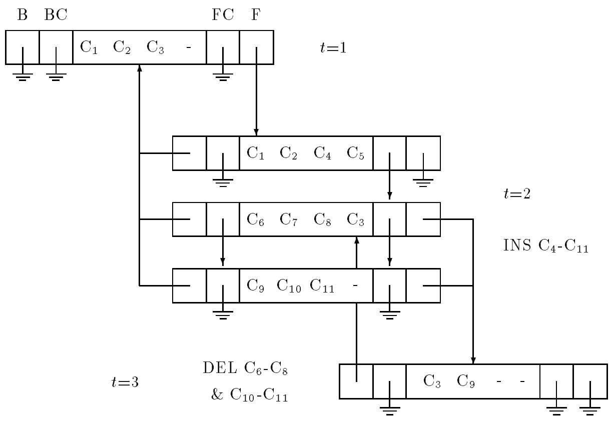


Fig. 3. Forward and backward chaining for the support of temporal queries.

Figure 3 shows how the leaves of three successive B$^+$trees can be forward- and backward-chained to support temporal queries. The leaf on the left-top corner of the figure corresponds to the first time instant, t=1, and contains the 3 quadcodes. Suppose that during time instant t=2, 8 quadcodes are inserted. In such a case, we have a node split. During time instant t=3, a set of 5 quadcodes is deleted. Thus, two nodes of the tree corresponding to time instant t=2 are merged to produce a new node as depicted in the figure.

3 Temporal Window Query Processing

In Spatial Databases and Geographical Information Systems there exists the need for processing a significant number of different spatial queries. For example, nearest neighbor finding, similarity queries, spatial joins of various kinds, window queries, etc. In this section we provide algorithms for the solution of various temporal window queries for evolving regional data. Given a window belonging in the area covered by our images and a time interval the following spatio-temporal queries may be expressed:

3.1 The Strict Containment Query

– Find the black regions that totally fall inside the window (including the ones that touch the window borders from inside) at each time point within the time interval.

In Figure 4 an example of a raster image corresponding to a specific time point, partitioned in quadblocks and a query window are depicted. The Strict Containment window query for this time point would return quadblocks 2 and 4. The algorithm that processes such temporal window queries is as follows:

1. Break the window into maximal sub-quadrants, as if it were a black region represented by a region quadtree.
2. For each of these sub-windows (in order according to the directional code of their North-West corner), compute the smallest and largest directional codes that may appear in the sub-window. The range of these codes includes all the codes (black sub-quadrants) that are strictly included within the sub-window. Perform a respective range search in the B^+tree of the first time point and discover the leaves that either contain such codes 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 sub-quadrants that are strictly contained within the window for the specific time point.
3. For each leaf discovered in step 2, following the F-pointer at first step and the chain of FC-pointers at second step, discover the leaves that *evolve* from this leaf at the next time point. Discard from further consideration the leaves, the range of which does not intersect with the respective range specified in step 2. The codes that fall within this range and appear in the remaining leaves are the black sub-quadrants that are strictly contained within the sub-window for the specific time point. Proceed to the tree for the next time point by repeating step 3 for each remaining leaf. Stop when the last time point of the time interval is reached.

Note, that when we process the query for a tree of a specific time point, we keep in main memory the nodes discovered for this tree, as well as some of the nodes of the tree of the preceding time point (only those that may lead us to nonaccessed nodes of the tree of the current time point). This holds for all the algorithms presented, except for the one related to the Cover query.

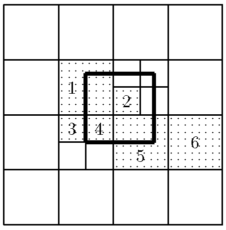


Fig. 4. The quadblocks of a raster image and a query window (thick lines).

3.2 The Border Intersect Query

– Find the black regions that intersect a border of the window (including the ones that touch a border of the window from inside or outside) at each time point within the time interval.

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

1. Create a rectangular strip that is formed by keeping the pixels that make up the border of the query 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 such outside pixels. Therefore, the resulting strip is up to 2 pixels thick. Break the strip into maximal sub-quadrants (of size 2×2, or 1×1).
2. For each of these sub-quadrants (in order according to the directional code of their North-West corner), compute the smallest and largest directional codes that may appear in the sub-quadrant. Then, perform a range search in the B^+tree of the first time point and discover a number of leaves (as in step 2 of the previous algorithm). If such a search returns no quadblocks, search for an ancestor of the sub-quadrant. The quadblocks discovered (if any) intersect the border of the window for the specific time point.
3. For each leaf discovered in step 2, following the F-pointer at first step and the chain of FC-pointers at second step, discover the leaves that *evolve* from this leaf at the next time point. Discard from further consideration the leaves the range of which does not intersect with the respective range specified in step 2. If there are no codes that fall within this range, search for an ancestor of the sub-quadrant corresponding to this range. The codes (quadblocks) discovered (if any) intersect the border of the window for the specific time point. Proceed to the tree for the next time point by repeating step 3 for each remaining leaf. Stop when the last time point of the time interval is reached.

Note, that a search for an ancestor of a quadblock, is a search for the maximum FD code that is smaller than the FD code of the quadblock. If (i) such a code exists, (ii) has a depth D smaller than the quadblock and (iii) the two FD codes coincide in their first D 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 under consideration is not the first in its node, we examine the presence of an ancestor in this node. Otherwise, if the previous node is among the nodes that reside in main memory, we examine the presence of an ancestor in this node. If it is not in main memory, but the respective previous node of the preceding tree is in main memory, we use F and possibly FC pointers to reach the previous node for the current tree with very few disk accesses. If, however, none of the above holds, we have to perform a search in the current tree for the previous 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 quadblock may be common to a number of subsequent quadblocks (due to the order under which sub-quadrants are treated in step 3). Thus, keeping in a variable the ancestor discovered (if any) for one sub-quadblock may help us to avoid the repetition of the same disk accesses later in the processing of the same time point.

3.3 The Cover Query

– Find out whether or not the window is totally covered by black regions at each time point within the time interval.

The Cover window query returns YES/NO answers. For the time point corresponding to Figure 4, it would return as answer NO. The algorithm that processes this kind of queries is as follows:

1. Break the window into maximal sub-quadrants.
2. For the first of these sub-quadrants (in order according to the directional code of their North-West corner), perform a search in the B$^+$tree of the first time point and access (discover) the leaf that should contain the related FD code. If the code of the sub-quadrant is present in the leaf, continue. If not, examine the FD codes in the same leaf that are before and after the code of the sub-quadrant (one of them at least exists). If these codes correspond to a sibling or successor of the sub-quadrant mark that the answer for the specific time point will be NO (the window cannot be totally covered). However, continue processing for this time point in order to discover leaves needed for the remaining time points. If the adjacent FD codes do not correspond to siblings or successors, search for an ancestor of this node. If such an ancestor does not exist, mark NO.
3. For the leaf discovered in step 2, following the F-pointer at first step and the chain of FC-pointers at second step, discover the leaves that *evolve* from this leaf at the next time point. Examine these leaves for the presence of the code of step 2, or any of its ancestors or successors (in rare cases, a search from the root of the related tree may be needed in order to examine the presence of an ancestor). According to the nodes discovered, NO may be marked for this time point. Repeat step 3, until you have reached the last image. For those images that have been marked with NO, such that all the images after them have been also marked with NO, the answer is definitely NO. The algorithm does not 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 its right end have all been marked with NO. Next, repeat step 2 and handle the next unprocessed subquadrant. The algorithm stops when, for every image that has not been excluded, all the sub-quadrants have been processed. For those images that a NO answer has not been marked, the answer for the corresponding time points is YES.

Note the difference of policy from the previous algorithms. In the Cover query we keep in main memory only the nodes related to one quadrant of the current tree, as well as the respective nodes of the preceding tree. It is evident that, we must reserve space for holding the YES/NO answers for all the images in the time interval. This approach is likely to produce NO answers for groups of images and not single images, while it saves us from unnecessary disk accesses.

The Cover window query algorithm can be extended so as to work for partially black windows, where the black percentage exceeds a specified threshold. That is, we could answer a Fuzzy Cover window query of one of the following two forms:

– 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.
– Find out the percentage of the window area that is covered by black regions at each time point within the time interval.

3.4 General Comment for Window Query Algorithms

Alternative naive algorithms for answering the above spatio-temporal queries are easy to devise. These algorithms would perform a suitable range search for all the trees that correspond to the given time interval (starting from the respective roots). This alternative approach would not take into account the "horizontal" pointers that link leaves of different trees and is expected to have significantly worse I/O performance.

All the presented algorithms can be easily transformed to work backwards: by starting from the end of the time interval and by using the B-pointer and BC-pointers.

4 Experiments

We implemented the structure of Overlapping Linear Quadtrees in C++. Note that, in order to maximize overlapping, in our implementation a disk page may host a number of consecutive B^+tree leaves (more details on this B^+tree variation appear in [15]). We performed experiments for page sizes equal to 1K and 2K bytes. For 1K pages, the capacity of internal nodes was 124 keys and the size of each leaf was 1/12 of a page. For 2K pages, the capacity of internal nodes was 252 keys and the size of each leaf was 1/24 of a page. The size of our images was 512×512 pixels and we used the algorithm OPTIMAL_BUILD described in [11] for converting the images from raster to linear FD representation.

At the start, the first image was created and its FD codes 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 B^+tree with average storage utilization equal to $\ln 2$. This image represents the last image in a large sequence of overlapped images and its quadcodes were also kept in a main memory compacted binary array ($512 \times 512/8$ bytes). Next, the second image was created as a modification of the first image and its FD codes were inserted in the second B$^+$tree, so that the identical subtrees between the two trees overlapped. There was no I/O cost for black quadrants that were identical between the two consecutive images, since, by making use of the main memory compacted array, we were able to sort out the respective identical FD codes.

The first image of each pair was created according to the model of increased image aggregation coefficient [6], $agg(I)$, of an image I. This quantity has been defined and studied in [6] and expresses the coherence of unicolor regions of the image. Starting from a random image with given black/white analogy (an image where each pixel has been colored independently with probabilities that obey this black/white analogy) and using the algorithm presented in [6], an image with the same black/white analogy and higher aggregation (more realistic) can be created.

The second image of each pair was formed by randomly changing the color of a given percentage of the image pixels. 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 better I/O performance is expected for the algorithms presented.

Every experiment was repeated 10 times using a pair of similar images. After the two images were created, windows of sizes equal to 64×64 or to 128×128 pixels were set and each of the three algorithms was executed 10 times for a random window position. In other words, each of the algorithms was run 10×10 = 100 times. Besides, for each window position, the analogous naive algorithms were executed (algorithms that do not make use of horizontal pointers, but perform independent searches through roots). In each run, we kept track of the number of disk reads needed to perform the query.

In the following, various experimental results are depicted, by assuming that the change probability is 2%. In the left part of Figure 5, for the Strict Containment query (processed by making use of horizontal pointers) one can see the number of node accesses as a function of aggregation for various black/white analogies of the first image. The window size is 64×64 pixels and the page size 1K. In the right part of the same figure, for the Strict Containment query one can see the number of node accesses as a function of aggregation for the naive algorithmic approach and the one that uses horizontal pointers. The black percentage is 70% and the page size is again 1K. Results (different plots) for window sizes equal to 64×64 and 128×128 pixels are depicted.

In the left part of Figure 6, another diagram for the Strict Containment query is depicted. Except for the page size which is equal to 2K, the rest of experimental setting is the same as in the previous diagram. In the right part of the same figure an analogous diagram for the Border Intersect query, for page size equal to 1K, is depicted.

In the left part of Figure 7, another diagram for the Border Intersect query is depicted. The experimental setting is identical to the one of the previous diagram, with the exception of the page size which is equal to 2K. The right part of Figure 7 refers to the Cover query: one can see the number of node

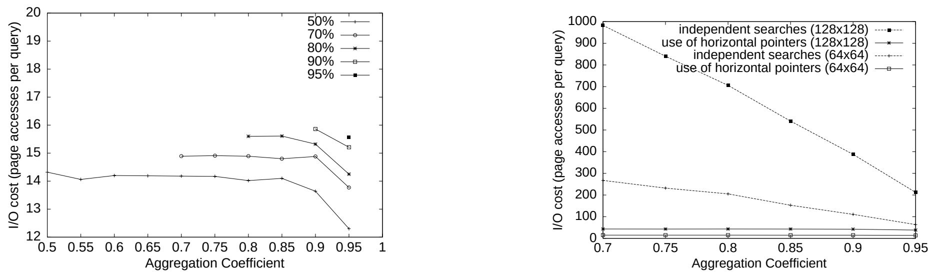


Fig. 5. The I/O efficiency (when page size is 1K) of the Strict Containment query, as a function of aggregation, for various black percentages (left) and for two algorithmic approaches (right).

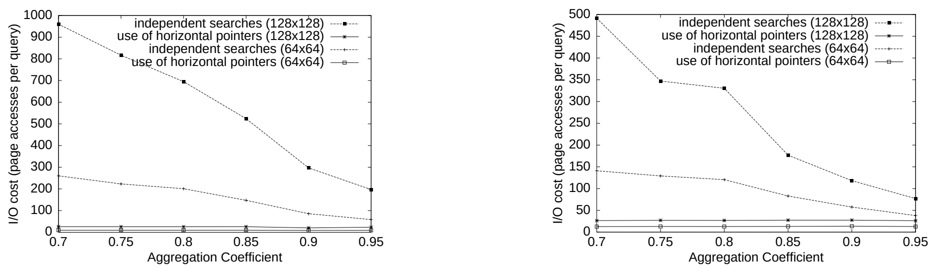
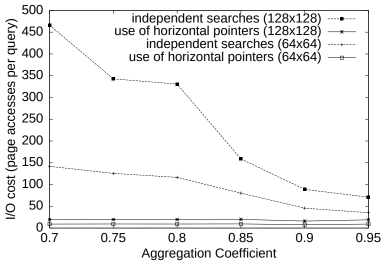


Fig. 6. The I/O efficiency of the Strict Containment auery, for page size of 2K (left) and of the Border Intersect query, for page size of 1K (right), as a function of aggregation.

accesses as a function of aggregation for the naive algorithmic approach and the one that uses horizontal pointers. The window size is 64×64 pixels and the black percentage is 70%. Results (different plots) for page sizes equal to 1K and 2K are depicted.

Note that, since the Cover query is of the YES/NO type and the intelligent exclusion of group of images from further consideration (see subsection 3.3) has been used, the number of node accesses is extremely small. A general remark that can be made for the diagrams in which the naive approach and the approach of Section 3 are compared, is that the use of horizontal pointers leads to significantly higher I/O efficiency for all the three algorithms.

In the future, we plan to perform further experiments for a number of cases. The parameters that may vary in these experiments are: the image size, the disk page size (or the number of leaves fitting in a page), the method of creating the first image, the window size (in relation to the whole image), the black/white analogy for the model based on aggregation, and the percentage of difference in creating the second image of each pair.



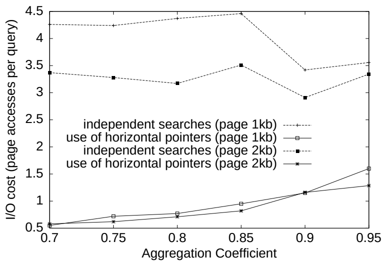


Fig. 7. The I/O efficiency of the Border Intersect query for page size of 2K, (left) and of the Cover query for page sizes of 1K and 2K (right), as a function of aggregation.

5 Conclusions

In this paper, we presented three algorithms for processing spatio-temporal queries in an image database that is organized with Overlapping Linear Quadtrees. More specifically, we presented three temporal window query processing algorithms: strict containment, border intersect and cover. Besides, we presented experiments that we performed for studying the I/O efficiency of these algorithms. The experiments were based on synthetic pairs of evolving images. The first image of each pair was formed according to the model of random images with specified aggregation of black regions [6]. The second image of each pair was formed by random change of pixels, a rather pessimistic method of image changing. In real situations it is expected that the above algorithms will perform even better. Even in this case, our experiments showed that, thanks to the presence of "horizontal" pointers in the leaves of the Overlapping Linear Quadtrees, our algorithms run with very few disk accesses.

In the future, we plan to develop algorithms for other/new spatio-temporal queries that take advantage of Overlapping Linear Quadtrees and study their behavior. Moreover, we plan to investigate the possibility of analyzing the performance of such algorithms.

Acknowledgments

The second author, who is a Post-doctoral Scholar of the State Scholarship Foundation of Greece, would like to thank this foundation for its financial assistance.

References

1. F.W. Burton, M.W. Huntbach and J. Kollias: "Multiple Generation Text Files Using Overlapping Tree Structures", *The Computer Journal*, Vol.28, No.4, pp.414-416, 1985. 87

2. F.W. Burton, J.G. Kollias, V.G. Kollias and D.G. Matsakis: "Implementation of Overlapping B-trees for Time and Space Efficient Representation of Collection of Similar Files", *The Computer Journal*, Vol.33, No.3, pp.279-280, 1990. 87
3. V. Gaede and O. Guenther: "Multidimensional Access Methods", *ACM Computer Surveys*, Vol.30, No.2, pp.170-231, 1998. 85
4. C.S. Jensen, J. Clifford, R. Elmasri, S.K. Gadia, P. Hayes and S. Jajodia (ed.): "A Consensus Glossary of Temporal Database Concepts", *ACM SIGMOD Record*, Vol.23, No.1, pp.52-64, 1994. 85
5. Y. Manolopoulos and G. Kapetanakis: "Overlapping B+trees for Temporal Data", *Proceedings of the 5th Jerusalem Conference on Information Technology (JCIT)*, pp.491-498, Jerusalem, Israel, 1990. Address for downloading: http://delab.csd.auth.gr/publications.html 87
6. Y. Manolopoulos, E. Nardelli, G. Proietti and M. Vassilakopoulos: "On the Generation of Aggregated Random Spatial Regions", *Proceedings of the 4th International Conference on Information and Knowledge Management (CIKM)*, pp.318-325, Washington DC, 1995. 86, 94, 94, 94, 96
7. M. Nascimento and M. Eich: "An Introductory Survey to Indexing Techniques for Temporal Databases", Southern Methodist University, Technical Report, 1995. 86
8. M.A. Nascimento and J.R.O. Silva: "Towards Historical R-trees", *Proceedings of ACM Symposium on Applied Computing (ACM-SAC)*, 1998. 86
9. B. Saltzberg and V. Tsotras: "A Comparison of Access Methods for Time Evolving Data", *ACM Computing Surveys*, to appear. Address for downloading: ftp://ftp.ccs.neu.edu/pub/people/salzberg/tempsurvey.ps.gz. 86
10. H. Samet: "The Design and Analysis of Spatial Data Structures", *Addison-Wesley*, Reading MA, 1990. 85, 87
11. H. Samet: "Applications of Spatial Data Structures", *Addison-Wesley*, Reading MA, 1990. 88, 93
12. Y. Theodoridis, M. Vazirgiannis and T. Sellis: "Spatio-Temporal Indexing for Large Multimedia Applications", *Proceedings of the 3rd IEEE Conference on Multimedia Computing and Systems (ICMCS)*, 1996. 86
13. 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)*, pp.123-132, Capri, Italy, 1998. 86
14. T. Tzouramanis, Y. Manolopoulos and N. Lorentzos: "Overlapping B+trees - an Implementation of a Transaction Time Access Method", *Data and Knowledge Engineering*, Vol.29, No.3, pp.381-404, 1999. 87
15. T. Tzouramanis, M.Vassilakopoulos and Y. Manolopoulos, "Overlapping Linear Quadtrees: a Spatio-temporal Access Method", *Proceedings of the 6th ACM Symposium on Advances in Geographic Information Systems (ACM-GIS)*, pp.1-7, Bethesda MD, November 1998. 86, 86, 86, 88, 93
16. M. Vassilakopoulos, Y. Manolopoulos and K. Economou: "Overlapping for the Representation of Similar Images", *Image and Vision Computing*, Vol.11, No.5, pp.257-262, 1993. 87
17. M. Vassilakopoulos, Y. Manolopoulos and B. Kroell: "Efficiency Analysis of Overlapped Quadtrees", *Nordic Journal of Computing*, Vol.2, pp.70-84, 1995. 87
18. X. Xu, J. Han, and W. Lu: "RT-tree - an Improved R-tree Index Structure for Spatiotemporal Databases", *Proceedings of the 4th International Symposium on Spatial Data Handling (SDH)*, 1990. 86