Available at
www.ElsevierComputerScience.com
POWERED BY SCIENCE DIRECT®
Data & Knowledge Engineering xxx (2003) xxx–xxx
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Benchmarking access methods for time-evolving regional data ☆

Theodoros Tzouramanis [a], Michael Vassilakopoulos [b], Yannis Manolopoulos [c,*]

[a] *Department of Information and Communication Systems Engineering, University of the Aegean, P.O. Box 19, 832 00 Karlovasi, Samos, Greece*
[b] *Department of Informatics, Technological Educational Institute of Thessaloniki, P.O. Box 14561, 541 01 Thessaloniki, Greece*
[c] *Department of Informatics, Aristotle University of Thessaloniki, 541 24 Thessaloniki, Greece*

Received 4 February 2003; received in revised form 7 May 2003; accepted 12 August 2003

Abstract

In this paper we present a performance comparison of access methods for time-evolving regional data. Initially, we briefly review four temporal extensions of the Linear Region Quadtree: the Time-Split Linear Quadtree, the Multiversion Linear Quadtree, the Multiversion Access Structure for Evolving Raster Images and Overlapping Linear Quadtrees. These methods comprise a family of specialized access methods that can efficiently store and manipulate consecutive raster images. A new simpler implementation solution that provides efficient support for spatio-temporal queries referring to the past through these methods, is suggested. An extensive experimental space and time performance comparison of all the above access methods follows. The comparison is made under a common and flexible benchmarking environment in order to choose the best technique depending on the application and on the image characteristics. These experimental results show that in most cases the Overlapping Linear Quadtrees method is the best choice.

Keywords: Spatio-temporal DBs; Optimization and performance; Access methods; Linear region Quadtree; Image representations

☆ Research performed under the European Union's TMR Chorochronos project, contract number ERBFMRX-CT96-0056 (DG12-BDCN).
* Corresponding author. Tel.: +30-31-996363; fax: +30-31-996360.
E-mail addresses: ttzouram@icsd.aegean.gr (T. Tzouramanis), vasilako@it.teithe.gr (M. Vassilakopoulos), manolopo@delab.csd.auth.gr (Y. Manolopoulos).

0169-023X/$ - see front matter
doi:10.1016/j.datak.2003.08.008

1. Introduction

In spatio-temporal databases (STDBs) special techniques should be developed for the efficient storage and access of spatial objects, their geometric representations, trajectories and other time-varying characteristics [20]. Existing spatial access methods can not handle continuously changing data. The reason is that the continuous change of the location and shape of spatial objects with time, implies that the spatial index has to be continuously updated. This is clearly an undesired solution since time-evolving spatial data may be updated very frequently resulting in serious performance overhead, whereas the support of historical queries will be impossible.

Up until now, a number of spatio-temporal access methods (STAMs) have been proposed for STDBs. These methods fall into two general classes: Quadtree-based and R-tree-based access methods. The Quadtree-based methods enhance previous methods based on Linear Quadtrees [4,13] by embedding additional structuring. In particular, the Quadtree-based approaches either use overlapping to represent successive database states (Overlapping Linear Quadtrees [22]), or conceptually couple time intervals with information about spatial subregions in each tree node by adopting ideas from specific transaction-time access methods (Multiversion Linear Quadtree [21]).

The R-tree-based approaches for indexing spatio-temporal data either treat time as just another spatial dimension (3D R-tree [26] and 2 + 3 R-tree [12]), or accommodate time by maintaining a time-indexed collection of R-trees (MR-tree and RT-tree [28], HR-trees [11], HR + trees [18] and MV3R-tree [19]). Generally speaking, STDBs cannot be viewed as spatial databases (SDBs) with time as an extra dimension, because time behaves differently from other dimensions in most applications. Therefore, it is not efficient to implement such an approach by simply adapting methods for high-dimensional SDBs. There is also a small number of R-tree-based proposals aiming at indexing moving points and trajectories [15]. This class of access structures usually stores current location and some additional information (such as speed and direction) to predict future object locations.

The present paper considers *access methods for time-evolving regional data*, that is regional data that are stored in raster format (raster images). These images can be represented with Quadtrees and, thus, in the sequel we focus in Quadtree-based STAMs. The R-tree-based STAMs are not suitable for representing raster images, whereas in cases where a lot of empty space exists in the MBRs, the index ability in decreased when pruning space and objects in top-bottom traversals. In this paper, we deal with four Quadtree-based STAMs: the Time-Split Linear Quadtree, the Multiversion Linear Quadtree, the Multiversion Access Structure for Evolving Raster Images and Overlapping Linear Quadtrees. These methods comprise a family of specialized access methods that can efficiently store and manipulate consecutive raster images. More specifically, the contribution of this research is twofold:

- the first contribution is a new simple method that delivers the most satisfactory trade-off between space and access time than other similar existing methods, in case spatial queries refer to past states. This is achieved through the use of four "horizontal" pointers in the leaves of the STAMs.
- the second contribution is the exhaustive performance comparison of the four access methods, under a common platform. This way, the comparison is fair for all methods, so that it can be safely concluded which is the best one.

In order to evaluate such STAMs, extensive experimentation using benchmark data is required. Sequences of evolving synthetic raster images with real-world semantics were generated by the G-TERD method [24], which is a benchmarking tool for access methods for time-evolving regional data. It is important to highlight that by using the G-TERD benchmark tool, the experiments are reproducible and the results hold not only for a specific environment but in more general settings as well. Thus future researchers are able to reproduce the conditions and confirm the results [29].

The rest of the paper is organized as follows. The next section provides background information on regional data and surveys various transaction-time proposals that are directly related to this study. Section 3, presents an overview of Time-Split Linear Quadtree, Multiversion Linear Quadtree, Multiversion Access Structure for Evolving Raster Images and Overlapping Linear Quadtrees, whereas Section 4 studies spatio-temporal query processing with regards to the investigated access methods. Section 5 contains an extensive experimental performance comparison, regarding space requirements, data redundancy, index building cost, buffering cost, query time response and several other important parameters that characterize the performance of the four STAMs. Finally, Section 6 discusses the experimental results, whereas Section 7 summarizes the contributions and makes suggestions for further research.

2. Related work

2.1. Indexing regional data

In the sequel we assume a two-dimensional space, although the presented methods can be easily extended into higher dimensions in most cases. We also assume that a raster image is represented as a $2^n \times 2^n$ array of pixels, where n is a positive integer. If the pixel colors are black and white only, the image is said to be *binary*, where 1 stands for black and 0 for white color.

2.1.1. Region Quadtree

The term *Quadtree* describes a class of hierarchical spatial access methods based on the principle of recursive space decomposition. The simpler example of a Quadtree representation of data corresponds to the representation of two-dimensional binary raster images. It is called *region Quadtree* and is based on the successive image decomposition into four disjoint quadrants of $2^{n-1} \times 2^{n-1}$ pixels. If a part is not entirely black or white, then it is recursively subdivided into four sub-quadrants, until each sub-quadrant is unicolor. Fig. 1b depicts an example of a region Quadtree, which represents the $2^3 \times 2^3$ binary image of Fig. 1a.

The root of the tree of Fig. 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 northwest (NW), northeast (NE), southwest (SW) and southeast (SE) quadrants. Leaves 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.

2.1.2. Linear Region Quadtree

The region Quadtree is a main memory structure. However, non-volatile storage should be used to store data in large database applications. In such cases, information about the black leaves can

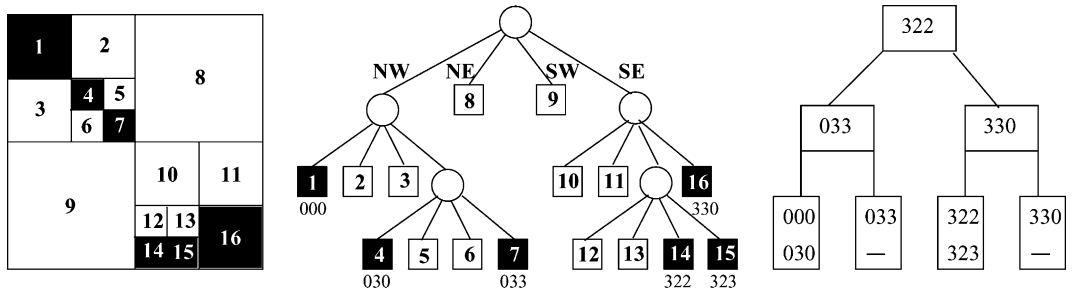

Fig. 1. (a) A $2^3 \times 2^3$ raster image array of black and white pixels and its corresponding (b) region Quadtree and (c) Linear region Quadtree.

be inserted into a B^+-tree, thus producing a pointer-less version of the Quadtree. The latter method is called *Linear Region Quadtree* (*Linear Quadtree* in the sequel). Each black Quadtree leaf is represented by a pair of numbers $\langle C, L \rangle$. The first number C, the *locational code*, is a base-4 number of n digits $(q_{n-1}, q_{n-2}, \ldots, q_0)$ with values 0, 1, 2 or 3, which correspond to the NW, NE, SW and SE quadrants, respectively. Each one of the n digits is a code that supports the Quadtree traversal along a path from its root to the appropriate leaf. The second number L of the pair $\langle C, L \rangle$ is the Quadtree level where the black node is located (the Quadtree root resides at level n, the pixels at level 0). If a black Quadtree node resides on level i, where $n \geqslant i \geqslant 0$, then the first $n - i$ digits determine the path from the root to this node and the last i digits are 0's ("don't care").

This linear representation of the Quadtree, is called FD (Fixed length—Depth) linear implementation. Two other linear implementations are met in the literature: Fixed Length (FL) and Variable Length (VL) (see [14] for details). As justified in [22], the FD linear implementation was the most appropriate choice for this study.

Fig. 1c presents a Linear Quadtree which corresponds to the Quadtree of Fig. 1b and the binary raster image array of Fig. 1a. For simplicity, only the FD locational codes (*quadcodes* in the sequel) of the black leaves appear in the Linear Quadtree, whereas their level is not shown. For example, node 7 (16) of the Quadtree is represented by the pair of numbers $\langle 033, 0 \rangle$ ($\langle 330, 1 \rangle$). The 033 (330) corresponds to the NW, SE and SE (SE, SE and "don't care" for the last digit) directions followed to reach this node from the root.

2.2. Indexing transaction-time data

Two concepts of time are usually considered in TDBs, *valid* and *transaction time*. According to Jensen et al. [5], valid time is the time during which a fact is true (valid) in the real world. Transaction time is the time during which a fact is stored in the database. In terms of data modeling, these two temporal aspects are orthogonal in that each could be recorded independently. Each one of them usually comprises of a *start time-point* and an *end time-point* or, equivalently, a semi-closed *interval* [*StartTime*, *EndTime*). Depending on which concept(s) of time is (are) considered, a TDB is characterized as *transaction time*, *valid-time* or *bi-temporal*. A survey on access methods used in TDBs can be found in [16]. The most cited temporal access methods that

support efficient indexing on transaction time are the following. The Time-Split B-tree (TSBT, [8]) indexes record versions in two dimensions; one dimension for the conventional record key and the other one the corresponding timestamp. The Multiversion B-tree (MVBT, [1]) indexes the evolution of one-dimensional data in transaction-time databases. The Multiversion Access Structure (MVAS, [27]) is similar to the MVBT. However, it demands smaller space requirements by using better policies for splits and merges. Overlapping B^+-trees (OB^+ trees, [10,17]) transform the "ephemeral" B^+-tree into a partially persistent [3] structure, by storing common sub-trees of successive versions of B^+-trees only once, in order to avoid the registration of identical (redundant) data. The Quadtree-based spatio-temporal extensions of these access methods are the main subject of this research.

3. Access methods for time-evolving regional data

3.1. Framework and assumptions

We assume that a sequence of evolving raster images is stored in the database. Each of them has a unique timestamp T_i, where $i = 1, 2, \ldots, N$, and N is the total number of images. This temporal attribute expresses transaction time. Table 1 summarizes the basic parameters that will be used in the remainder of this paper.

A STAM implicitly associates a transaction-time interval $[T_{start}, T_{end})$ to each spatial object to represent the object lifetime. Moreover, it is assumed that when a change occurs in the real world, the database is updated at the same time (i.e., transaction time and valid time intervals coincide). When a new object is inserted at time T_i, this time interval is set equal to $[T_i, *)$, where the symbol '$*$' stands for the special value *now* [2] and means that the respective object will be valid until some time-point in the future, that is not known beforehand. A real world deletion at time-point $T_j (i < j)$ is carried out as a *logical* deletion by changing the T_{end} timestamp from $*$ to T_j. A leaf record of a STAM is said to be *current* or *alive* if its T_{end} timestamp is '$*$' i.e. it has not been

Table 1
Parameters used and definitions

Parameter	Description
$2^n \times 2^n$	size of a binary raster image ($n > 0$)
N	total number of time-evolving raster images
T_i	unique timestamp associated with image i ($i = 1, 2, \ldots, N$)
$\langle C, L \rangle$	FD linear representation ($\langle quadcode, level \rangle$) of a black Quadtree node
$[T_{start}, T_{end})$	semi-closed time interval representing the lifetime of a quadcode
R	number of non-redundant quadcodes ever inserted
R_{red}	number of redundant quadcodes ever created
R_{T_i}	number of (valid) quadcodes in timestamp T_i
S	number of different quadcodes ever inserted ($S \leqslant 2^n \times 2^n$ and $S \leqslant R$)
L_A	number of current quadcodes in node A
M	number of disk pages containing leaves of the structure
M_{T_i}	number of disk pages containing valid leaves for timestamp T_i
b	tree node capacity

updated or deleted. A leaf (an internal) node of a STAM is *current* or *alive* if it contains at least one alive record (if it points to at least one current leaf), otherwise it is *historical* or *dead*. Finally, there are two types of node splits allowed when a node overflows. *Key split* (*K-split*): is a split on the key, as the one in a standard B^+-tree. *Time split* (*T-split*): is a split on a specific value of the transaction time. Records valid at or after this specific *splitting-time* value go to the new current node and records valid before the splitting-time value remain in the original node, which becomes historical.

3.2. Time-Split Linear Quadtree

In this section, we present a spatio-temporal extension of TSBT, the Time-Split Linear Quadtree (TSLQ [23,25]), which converts the "ephemeral" Linear Quadtree to a *persistent* structure. TSLQ couples timestamps with spatial objects in each node. *Data* records residing in leaves are of the form $\{\langle C, L\rangle, T_{start}\}$, where $\langle C, L\rangle$ is the FD code of a black Quadtree node and T_{start} is the insertion timestamp of this FD code in the TSLQ. Non-leaf nodes contain *index* records of the form $\{C', T'_{start}, Ptr\}$, where *Ptr* is a pointer to a descendent node, C' is the smallest C of that descendent node and T'_{start} is the time-point at which the latter node became current.

The TSBT structure does not support physical record deletions. Thus, even if the resulting node after a T-split is sparse of records, TSBT does not run the risk of further future record reduction. On the contrary, TSLQ supports logical and physical deletions of existing records. Therefore, there is always a possibility that, as a result of consecutive physical deletions, a current node might remain with zero records in the future. To avoid this, we set a node consolidation threshold P_{TSLQ_cons} to control the minimum number of alive records a current TSLQ node must contain after a record deletion or a T-split. However, the existence of pure K-splits does not guarantee that for a time-point earlier than the current one, the number of valid records in a node would be above this threshold. Nevertheless, the role of P_{TSLQ_cons} threshold is very important and analogous to the node consolidation threshold used in the B^+-tree, which leads to a merging in the case of a sparse node. The requirement that a current page must have at least P_{TSLQ_cons} records alive, enables clustering of the alive FD codes at the current time in a small number of pages, which in turn will minimize the space overhead and the I/O cost for the next image insertion.

3.2.1. Insertion

Two types of nodes may exist: current and historical ones. In each TSLQ node, we added a new field, called "*StartTime*", to register the time-point when this node starts being current. As all new FD codes are inserted into current nodes, only current nodes are subject to a split. They can be split either by key (i.e. by FD code), or by time. The split dimension, i.e. by quadcode or time, is determined by a TSBT split policy. Lomet and Salzberg have proposed three different splitting policies for the TSBT (for a discussion of them see [8,9]). For our TSLQ implementation, the IKS policy was adopted, [1] since it generally exhibits a good balance between space overhead (i.e. the degree of redundancy of stored data) and query performance.

[1] The IKS policy performs a K-split if two thirds or more of the splitting node consists of current (valid) data. Otherwise a T-split is performed and the *time of last update* among all records in the overflowing node, is used as the splitting-time. The time of last update is the time value after which there were only insertions of records with new keys.

The *K-split* creates a new node with *StartTime* value equal to that of the initially overflowing node A. Assuming that L_A is the number of current records in the original node, then the first $\lceil L_A/2 \rceil$ of them are kept in the original node and the remaining are moved into the new node. Thus, after the K-split, the two TSLQ nodes share the same number of current records and not necessarily the same total number of records. This ensures that both TSLQ nodes will contain more than P_{TSLQ_cons} current records after the K-split, apparently assuming that $P_{TSLQ\_cons} \leqslant 1/2 \times b$, where b is the node capacity. According to the IKS policy, in order to have a K-split, the following must hold: $2/3 \times (b+1) \leqslant L_A \leqslant b+1$. Therefore, for the consolidation threshold it holds that: $P_{TSLQ\_cons} \leqslant 1/3 \times (b+1) \leqslant L_A/2$.

The selected split-key for the correct partitioning of the overflowing original node must be the $\lceil L_A/2 \rceil$-th current record of the node. Records with values larger than the split-key are directed to the new node, whereas records with values smaller than the split-key (the split-key included) remain in the original node. For example, consider the leaf A in Fig. 2a. The node capacity b is 6, and the records are of the form $\{\langle C, L\rangle, T_{start}\}$. After the insertion of the FD code $\langle 311, 0\rangle$ in timestamp 2, leaf A overflows. A wrong handling of the K-split is shown in Fig. 2b. After the K-split, leaf A contains $\lceil L_A/2 \rceil$ current records as expected, but the search for the record $\{\langle 220, 0\rangle, 1\}$ will fail. Fig. 2c shows the correct handing of the overflow, where the $\lceil L_A/2 \rceil$-th current record of the overflowed node is the last record on that node after the K-split.

The TSLQ T-split is also more complicated than that in TSBT. The detailed algorithm appears in [25]. The TSBT T-split produces one historical node and one new node with split-time being the

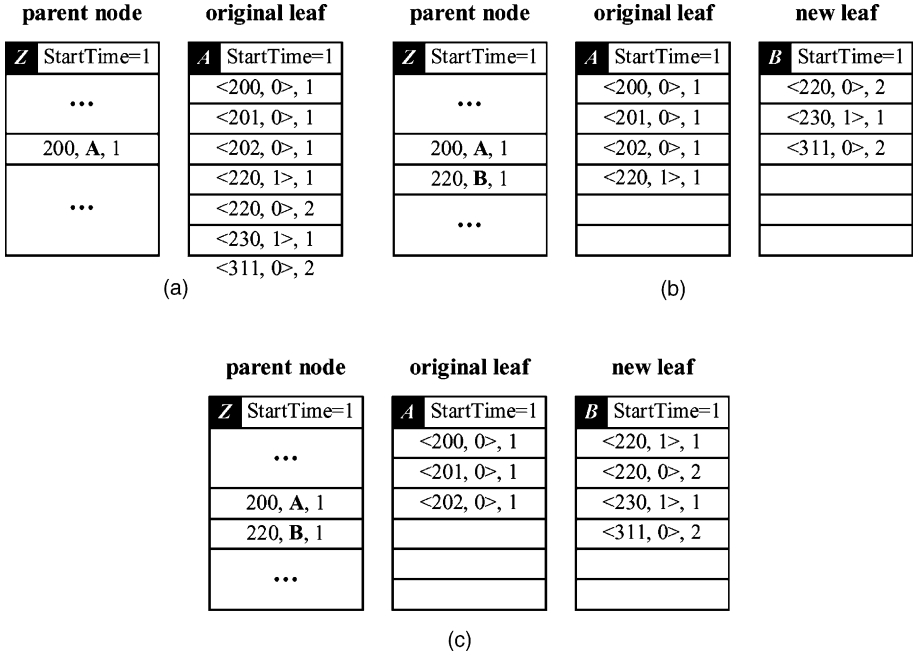

Fig. 2. (a) Leaf A overflows after inserting the FD code $\langle 311, 0\rangle$ in timestamp 2 (b) a wrong K-split and (c) the correct K-split of node A.

time of last update. On the contrary, the split-time in the TSLQ T-split may not be the time of last update of the overflowing node, but it may be a more recent timestamp as well. If $P_{TSLQ\_cons} > 0$ and the classical TSBT T-split procedure leads to a TSLQ node containing less than P_{TSLQ_cons} current records, then before proceeding to the T-split, an alive sibling (preferably the right one) of that TSLQ node must first be allocated. If such a sibling exists, [2] the TSLQ T-split differs than that of TSBT. In this case, the time of last update of the overflowing node is compared to the sibling *StartTime*. If they coincide, then all the alive records of the original node are copied into its sibling. Records with T_{start} greater than or equal to the split-time, need not have copies in the original node that turns to historical. Otherwise, if the time of the last update of the overflowing node and the *StartTime* of the sibling differ, then both the overflowing and the sibling node need a T-split. The more recent time between the times of two nodes last updates is chosen as the split-time for both T-splits. The *StartTime* field of the new current nodes gets the value of the split-time stamp. If the total number of records of the two new nodes is below b, then the two nodes are merged; otherwise a record redistribution between them occurs so that both nodes are at least half full. The total number of current records after the two T-splits is guaranteed to be at least $2 \times P_{TSLQ\_cons} - 1$.

3.2.2. Deletion

The deletion algorithm in the TSLQ is also significantly different from that of TSBT, which merely posts *deletion markers* for all deleted records and does not merge sparse nodes. On the contrary, the implementation of a real world deletion of an FD code $\langle C, L\rangle$ in the TSLQ at time-point T_j depends on the *StartTime* field of the corresponding leaf. For details, see [23,25].

3.2.3. Update

Updating (i.e. changing the value of the level L of) an existing FD code leaf entry at time-point T_j is handled depending on the *StartTime* field of the corresponding leaf:

- If $StartTime < T_j$ then the update is carried out by (i) the logical deletion of the existing 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'. The number of current records in the leaf does not change but if the procedure leads to a leaf overflow, the IKS policy must be followed.
- Otherwise, if $StartTime = T_j$ then the update is carried out by (i) the physical deletion of the existing entry and (ii) the insertion of a new version of that entry. In this case, both the number of current records and the total number of records in the leaf do not change.

3.2.4. Internal nodes

The insert, delete, update and K-split algorithms for TSLQ index nodes are similar to the ones for TSLQ leaves. However, the T-split algorithm differs from that of TSLQ leaves. In particular, whenever an index node becomes historical all its descendent nodes are also forced to split with the same split-time. This introduces additional redundancy but a historical internal node cannot

[2] If an alive sibling does not exist, then the current TSLQ tree version consists of only a single node, i.e. the overflowing node.

refer to current nodes since those nodes can split or move. After the T-split, index node entries with T_{start} greater than or equal to the split-time, need not have copies in the original and hereafter historical node.

3.3. Multiversion Linear Quadtree

The *Multiversion Linear Quadtree* (MVLQ [21]) is based on the hierarchical decomposition of space and maintains the evolution of a Linear Quadtree over time, by adapting ideas from the MVBT. MVLQ couples time intervals with spatial objects in each node. Data records residing in leaves contain entries of the form $\{\langle C, L\rangle, [T_{start}, T_{end})\}$, where $\langle C, L\rangle$ is the FD code of a black region Quadtree node and $[T_{start}, T_{end})$ represents the semi-closed time interval when this black node appears in the image sequence. Non-leaf nodes contain entries of the form $\langle C', [T'_{start}, T'_{end}), Ptr\rangle$, where *Ptr* is a pointer to a descendent node, C' is the smallest C value in that descendent node and $[T'_{start}, T'_{end})$ is the time interval that expresses the lifetime of the latter node. In each MVLQ node, the field *StartTime*, registers the node creation time.

MVLQ hosts a number of version trees and has a number of roots so that each root stands for a time/version interval $T' = [T_i, T_j)$, where $i, j \in \{1, 2, \ldots, N\}$ and $i < j$. On top of the MVLQ, an additional main memory structure, *the root* table*, is used to index the MVLQ roots. Each record in the root* table represents an MVLQ root and obeys the form $\{T', Ptr'\}$, where T' is the root lifetime and Ptr' is a pointer to its physical disk address. The MVLQ insert, delete and update algorithms are significantly different from the corresponding MVBT algorithms.

3.3.1. Insertion

A quadcode insertion leads to a structural change if at least one new MVLQ node is created. 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 K-split occurs and the leaf splits. The first $\lceil (b+1)/2 \rceil$ 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 TSBT T-split must first be performed in the original leaf since it is not acceptable to structurally change nodes that may contain historical data (as in MVBT). The split-time chosen is always the current time. The number of (current) quadcodes after the T-split must be in the range from P_{MVLQ_low} to P_{MVLQ_high}, which are two given thresholds (for details see [1,21]). If a T-split leads to less than P_{MVLQ_low} quadcodes, then a merge or key redistribution is attempted with an alive sibling (preferably the right one) or a copy of that sibling containing only its current versions of quadcodes (the choice depends on the *StartTime* field of the sibling). If a T-split leads to more than P_{MVLQ_high} quadcodes in a node, then a K-split is performed.

3.3.2. Deletion

The implementation of a deletion of an existing quadcode at time-point T_j depends on the *StartTime* field of the corresponding leaf.

- If $StartTime = T_j$, then the appropriate entry of the form $\{\langle C, L\rangle, [T_{start}, *)\}$ is removed from the leaf. After this physical deletion, the leaf is checked if it contains enough entries. If the number of entries is above P_{MVLQ_cons}, where P_{MVLQ_cons} ($\leqslant P_{MVLQ\_low}$) is a node consolidation threshold similar to that of TSLQ, then the deletion is completed. If the latter number is below that threshold, then the node underflows. This case is handled as in the classical B^+-tree, with the difference that if an alive sibling exists (preferably the right one) then we check its *StartTime* field before performing a merge or a key redistribution (if the *StartTime* value of the sibling is smaller than the current time, a T-split is performed first).
- Otherwise, if $StartTime < T_j$ then the quadcode deletion is handled as a logical deletion, by updating the temporal information T_{end} of the appropriate entry from $T_{end} = *$ to $T_{end} = T_j$. If an entry is logically deleted in a leaf with exactly P_{MVLQ_cons} current quadcodes, then a *version underflow* occurs that causes the node T-split, copying the current quadcodes into a new node. Evidently, considering that $P_{MVLQ\_cons} \leqslant P_{MVLQ\_low}$ holds, the number of quadcodes after the T-split is below P_{MVLQ_low} and a merge or key redistribution is attempted with an alive sibling or a copy of that sibling.

3.3.3. Update

The update procedure is similar to the TSLQ one and can be applied to MVLQ with slight modifications. For brevity, the description of these modifications is not reported here.

3.3.4. Internal nodes

When an index or leaf node becomes historical, its index entry into its parent is updated for this change or it is totally removed, depending on the *StartTime* field of the parent node. Note that adding or deleting index entries in an internal node may cause its overflow or underflow, respectively. This case is handled in a way similar to the case of the leaves.

An important property that the MVLQ inherited from MVBT is that in each node except for the roots, the number of valid entries for a given timestamp T_i is either zero or at least P_{MVLQ_cons}. As in the case of MVBT, a new node must contain at least P_{MVLQ_low} entries, and no more than P_{MVLQ_high} entries. Therefore the number of inserts, deletes and updates that are necessary until the overflow or underflow of this node, depends on the values of these three specific thresholds.

3.4. Multiversion Access Structure for Evolving Raster Images

The Multiversion Access Structure for Evolving Raster Images (MVASERI, [25]) combines concepts from three different indexing structures. The initial motivation for the MVASERI design comes from the Multiversion Access Structure (MVAS, [27]). However, the new approach could be also viewed as a modified MVLQ or as another partial persistent variant of the ''ephemeral'' Linear Quadtree, designed as a directed acyclic graph of pages.

Although MVLQ and MVASERI leaf and internal node entries have exactly the same form, MVASERI uses extra storage-conservation overflow and underflow policies that substantially differ from those of MVLQ. These policies try to reduce storage in two ways: (a) by partial moving or copying records from a node, and (b) by using available space in an existing node rather than creating a new one.

3.4.1. Insertion

To insert a new record at time-point T_i, an entry of the form $\{\langle C, L\rangle, [T_{start}, T_{end})\}$ is simply added in the appropriate leaf. If there is not enough space in the leaf and an overflow occurs, then two thresholds control this process: $P_{MVASERI\_cons}$ and $P_{MVASERI\_high}$, where $4 \times P_{MVASERI\_cons} \leqslant P_{MVASERI\_high} \leqslant b$ (for details see [27]). An overflow in leaf A is handled by using the *StartTime* field and the number of alive records L_A. This algorithmic handling appears in [25]. The split-time of a T-split is always the current time. When the number of entries in a node generated after a T-split is below $P_{MVASERI\_cons}$, then we examine for an alive sibling. If such a sibling exists, the MVLQ would copy all the sibling's alive records to the new and sparse T-split node, thus logically deleting the sibling. Instead, the MVASERI will copy or move only that many alive records from the sibling, as needed for the T-split node to avoid violating the threshold. This modification reduces duplication and extra space.

3.4.2. Deletion

A deletion at time-point T_j is handled depending on the *StartTime* field of the leaf A.

- If $StartTime = T_j$ then the appropriate entry is removed from the leaf. If after this physical deletion the total number of entries in A is below $P_{MVASERI\_cons}$, then *a node underflow* occurs.
- Otherwise, if $StartTime < T_j$ then the T_{end} field of that record changes to T_j. If after this logical deletion the number of alive entries in A becomes less than $P_{MVASERI\_cons}$, then *a version underflow* occurs.

In both cases, in order to avoid having alive nodes sparse of current records, the MVASERI detaches some or all the current quadcodes from an alive sibling, according to the rules presented in [25].

3.4.3. Update

The update is handled in a way similar to that of TSLQ and MVLQ. Thus, for brevity its description is not included.

3.4.4. Internal nodes

Inserts, deletes, updates, overflows and underflows in index nodes are similar to the corresponding cases of leaves as described above. If due to node consolidations, the current root is left with only one current child, then this child becomes the current root. A difference to the MVLQ method, is that during an insert, delete or update in MVASERI, the quadcode range [3] of a node may change. In such a case, the index entry of this node into its parent may not be valid any more and may have to be deleted (logically or physically, depending on the *StartTime* field of the parent). Afterwards, a new entry for the descendant node with its new lowest quadcode and a new

[3] *Quadcode range* of a node is the set of quadcode values that are either contained within the specific node, or would be contained, if they had been inserted.

T_{start} must be posted into the parent node. This ensures that top-down tree searches for copied or moved quadcodes will lead to the proper leaf.

The main difference between the MVASERI insert, delete and update algorithms and the corresponding ones of MVAS, is that the processing of the overflow and underflow conditions in MVASERI are strongly affected by the *StartTime* value of each node involved in the process. This is because MVASERI processes a huge number of record inserts, deletes and updates per time-slot. On the contrary, only a single insert, delete or update may occur in MVAS per time-slot. Therefore, the *StarTime* value of an MVAS node is always a timestamp earlier than the time under process and it is not taken into account. The general remark is that the MVASERI algorithms are more complicated than the MVAS ones because MVASERI is used in applications with different data characteristics. This is the cost for MVASERI to reduce the number of T-splits and both the historical part of the structure and the redundant data.

3.5. Overlapping Linear Quadtrees

Overlapping Linear Quadtrees method (OLQ, [22]) is a transaction-time spatio-temporal index implemented as a sequence of Linear Region Quadtrees. The basic idea behind OLQ is that, given two Linear Region Quadtrees where the second one is based on some changes upon the first one, the second Linear Quadtree can be represented by registering only the modified branches of the first one. The sub-trees that remain the same, are not replicated but simply re-used. Thus, each tree has a separate root and substantial space is saved.

The OLQ leaves contain entries of the form $\{\langle C, L\rangle\}$, where $\langle C, L\rangle$ is the FD code of a black Quadtree node. By contrast to TSLQ and MVLQ, the insertion time-point or the time interval that represents the quadcode life span, are not saved in the OLQ leaves. Internal nodes contain records of the form $\{C', Ptr\}$, where *Ptr* is a pointer to a descendent node and C' is the smallest C value in that descendent node.

The record capacity of OLQ leaves is kept small. This factor maximizes the probability that a leaf will not change (a few FD codes are very likely to remain unchanged) and will belong to consecutive Linear Quadtrees. Thus, we keep the number of newly created paths between consecutive tree instances as low as possible, since consecutive images have large identical parts. In our implementation, the leaf capacity was 10 FD codes, thus a data page may host a number of consecutive leaves. When, during an image insertion, a new page is allocated on the disk, it is filled with new OLQ leaves as they are generated by the OLQ modification algorithm. When a data page runs out of free space, or a new image is to be inserted, then a new data page is allocated. As a consequence, it is guaranteed that all the OLQ leaves contained in a data page, have been generated during the same timestamp thus sharing the same *StartTime* value.

On the other hand, this technique helps so that some leaves (due to their small size) may remain valid for a long time period. As a consequence, top-down tree searches through OLQ roots need to visit a large number of pages hosting only a few long-life leaves that are however alive for the time-point of interest. In the worst case, a page may be accessed just because it contains only a single long-lived leaf of 5 alive FD codes. This degrades the efficiency of an image insert because the current tree version is scattered over a large number of pages created in various transaction

times and each of them contributing little to the current database. An efficient solution to this is to set a page usefulness threshold P_{OLQ_usef} at the leaf level that causes the T-split of all current leaves contained in a page, every time the total number of current records in that page is above the threshold. This technique enables better clustering of alive records per any transaction time and it is similar to the node consolidation in TSLQ, MVLQ and MVASERI.

The OLQ structure is accompanied by an additional substructure, the *Header Table*, which is built on top of the OLQ and is used similarly to the root* table in MVLQ and MVASERI. Each record in this table is of the form $\langle T_i, Ptr_i \rangle$, where T_i is the timestamp when the respective image recorded in the relation and Ptr_i points to the root of the corresponding Linear Quadtree.

The advantage of OLQ is its implementation simplicity, which is owed to the following properties.

- Except for the Header Table records, no other record involves any timestamping because nodes do not accommodate records from various versions.
- The form of OLQ index and data records and the search traversal are identical to the B^+-tree ones.
- Finally, except for the P_{OLQ_usef} page usefulness threshold, no additional threshold is considered during the index construction.

Therefore, OLQ is simply implemented as a forest of typical B^+-trees that share their common subtrees. However, this comes at the expense of the space performance. Even if a single record changes in a node, this node cannot be shared by different trees; rather a new copy of the node is generated. Further details can be found in [22].

3.6. Image insertion in TSLQ, MVLQ, MVASERI or OLQ

The structures of TSLQ, MVLQ, MVASERI and OLQ are accompanied by one more additional main memory table, which represents the last inserted image in the preorder traversal of its Quadtree. This compacted array is called *Depth First-expression* (DF-expression, [7,13]) and it keeps track of all the black quadblocks of the last inserted image, in order to know, at no I/O cost, the common black quadrants between this image and the next one. Thus, given a new image and before proceeding to its insertion, we do know the specific FD code inserts, deletes and updates. The DF-expression is composed of the symbols 'B', 'W' and 'G', corresponding to black, white and gray Quadtree nodes, respectively. For example, the DF-expression of the image of Fig. 1 is GGBWWGBWWBWWGWWGWWBBB.

Generally, we face an image insertion in two stages. Firstly, we sort the quadcodes of the new image and compare this sequence against the quadcodes of the last inserted image, using the binary table of its DF-expression. After discovering all the FD code inserts, deletes and updates at no I/O cost, we build the new tree version. Simultaneously, the DF-expression of the new image is constructed, replacing step-by-step the DF-expression of the previous one.

Table 2 summarizes the basic characteristics of TSLQ, MVLQ, MVASERI and OLQ. Some of the listed issues in the table are closely related.

Table 2
The basic characteristics of TSLQ, MVLQ, MVASERI and OLQ

Access method	T-split	Pure K-split	T-split and K-split	Sparse node merge	Number of roots	Leaves per disk page	Building thresholds	Additional sub-structure(s)
TSLQ	Yes	Yes	No	Yes	One	One	P_{TSLQ_cons}, P_{TSLQ_high}[a]	DF-expression
MVLQ	Yes	No	Yes	Yes	Many	One	P_{MVLQ_cons}, P_{MVLQ_low}, P_{MVLQ_high}	root* table and DF-expression
MVASERI	Yes	No	Yes	Yes	Many	One	$P_{MVASERI\_cons}$, $P_{MVASERI\_high}$	root* table and DF-expression
OLQ	Yes	No	Yes	Yes	Many	Many	P_{OLQ_usef}	Header Table and DF-expression

[a] P_{TSLQ_high} $(=2/3\times b)$ is the threshold which is introduced by the TSBT IKS split policy.

4. Spatio-temporal query processing

4.1. Benchmark queries for time-evolving regional data

Very common queries in STDBs and GISs are the window queries that also involve time. In particular, a spatial query retrieves all the spatial information related to a certain user criterion and refers to the current database version. On the other hand, a spatio-temporal query may also refer to specific past states or time-intervals of the database lifetime. Here, we present a set of necessary queries for the examination of the efficiency of access methods for time-evolving regional data. These queries are representative of typical STDB applications and, therefore, it would be very important for the TSLQ, MVLQ, MVASERI, and OLQ techniques, to support efficiently as many of them as possible.

Given a sequence of N binary raster images, each one associated with an unique timestamp T_i, for $i = 1, 2, \ldots, N$, we distinguish between two types of spatio-temporal queries for time-evolving regional data:

- *Snapshot* (*or Pure-timeslice*) *Queries*: that retrieve all the quadblocks alive at timestamp T_i, which therefore can reconstruct the corresponding image. In Fig. 3, an example of a specific

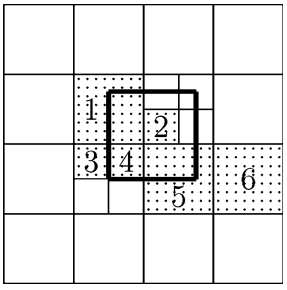

Fig. 3. The quadblocks of a image and a query window (thick lines).

image partitioned in quadblocks is depicted. The Snapshot Query for this timestamp would give back quadblocks 1–6.

- *Time-interval Window* (*or Pure-coordinate*) *Queries*: that retrieve all the quadblocks corresponding to black regions intersecting with the area of a given $k \times k$ window, at each time-point within the time interval $[T_i, T_j]$, where $i, j = 1, 2, \ldots, N$, and $i < j$.

The support for time-interval window queries is of primary importance since these queries are usually the building blocks for other more sophisticated operations. Therefore, their efficient processing is vital to the overall system performance. Five different time-interval window queries have been proposed in [22]. Given N raster images, a $k \times k$ window and a time interval $[T_i, T_j]$, where $i, j = 1, 2, \ldots, N$, and $i < j$, the following queries may be expressed:

The Strict Containment Window Query: Find the black regions totally inside the window (including those touching the window borders from inside), at each time-point within the time interval $[T_i, T_j]$. For the image of Fig. 3 and the depicted query window, the Strict Containment Window Query would give back quadblocks 2 and 4.

The Border Intersect Window Query: Find the black regions intersecting a border of the window (including those touching a window border from inside or outside), at each time-point within the time interval $[T_i, T_j]$. The Border Intersect Window Query for the time-point corresponding to the image of Fig. 3, would give back quadblocks 1, 3, 4 and 5.

The General Border Intersect Window Query: Find the black regions completely inside the window or intersecting a window border (including those touching a border from inside or outside), at each time-point within the time interval $[T_i, T_j]$. The General Border Intersect Window Query for the time-point corresponding to Fig. 3 would give back quadblocks 1–5.

The Cover Window Query: Find out whether or not the window is completely covered by black regions at each time-point within the time interval $[T_i, T_j]$. The Cover Window Query gives back YES/NO answers. For the time-point corresponding to Fig. 3, it would answer NO.

The Fuzzy Cover Window Query: This query can take two different forms:

1. Find out whether the percentage of the black part of the window area is over a given threshold, at each time-point within the time interval $[T_i, T_j]$,
2. Find out the percentage of the black part of the window area at each time-point within the time interval $[T_i, T_j]$.

The second kind of Fuzzy Cover Window Query for the time-point corresponding to Fig. 3 would give 80% as answer. The first kind would return YES/NO answers depending on the comparison of 80% with the given threshold.

4.2. Supplementary lists to speed up query processing

In order to keep track of the image evolution and efficiently satisfy spatio-temporal queries over the stored images, we embed some additional information in the leaves of TSLQ, MVLQ, MVASERI and OLQ. First, we add one more extra field in every leaf, called *EndTime*, to register when a leaf becomes historical. As long as a leaf remains current, its *EndTime* value equals to *now*. Moreover, in order to improve the efficiency of time-interval window queries over the stored images, we incorporated four horizontal pointers in every leaf of TSLQ, MVLQ, MVASERI and

OLQ. This way, there is no need to traverse consecutive tree instances to retrieve the history of an FD code. These pointers are called: F-pointer, ST-pointer, B-pointer and ET-pointer.

- F- (B-)pointer points to a leaf with a more recent (an earlier) creation timestamp than the one of the leaf accommodating the pointer. These pointers connect in two ways the leaves that are involved in a T-split procedure.
- ST- (ET-)pointer always points to a leaf (a historical leaf) with the same *StartTime* (*EndTime*) field value. It is used to link in increasing quadcode order, all the leaves that have been created (logically deleted) in the same transaction time.

The B-pointer is also used for the proper setting of the F-pointer during leaf mergings. If two current leaves are merged with *StartTime* values equal to *now*, then one of them has to be totally removed. As a consequence, the historical leaf (if any) that points through its F-pointer to the recently removed leaf has to be updated and point to the node produced after the merge. This historical leaf is accessed through the B-pointer of the recently removed leaf.

By using the F- and ST- (or B- and ET-)pointers, the satisfaction of a time-interval window query involves three steps.

1. Using the header table for OLQ or the root* table for MVLQ and MVASERI, the very first (the current, respectively) root is located. This part does not hold for TSLQ since it has only one root.
2. The answer for the first (for the current, respectively) timestamp is located by searching the corresponding tree version in an umbrella-like fashion, i.e. a Snapshot Query at the first (at the current, respectively) timestamp.
3. Finally, by using the forward (the backward, respectively) leaf chaining, historical nodes of later (earlier) timestamps are called recursively, until the current (the very first) timestamp of the time-interval is reached.

Example. Fig. 4 depicts a forward chaining example of TSLQ, MVLQ, MVASERI and OLQ leaves. The left part of the figure illustrates a set of leaves of four successive tree versions, whereas the middle part shows how these leaves can be forward-chained through the F- and ST-pointers. For simplicity, the B- and ET-pointers in the leaves are not depicted. The figure assumes a case where, among others, the leaves A, B, C and D were created during timestamp T_1. At time instant T_2, the leaves E and G were created after a T-split on leaves A and C, respectively, whereas the leaf F was created after a K-split on E. The leaves of timestamp T_1 that were not valid any more, were connected to the new nodes by using the F-pointer field, whereas the ST-pointer field was used to chain together the three new nodes. Similarly, a number of leaves was also generated during the timestamps T_3 and T_4. For instance, at timestamp T_3 the leaves B and G were merged, thus producing node H. In all cases, the F- and ST-pointer fields are maintained accordingly.

Assume now that a specific time-interval window query within the time interval $[T_1, T_4]$ is posed. Also, assume that the spatio-temporal rectangle of this query is the shadowed area of Fig. 4a. The answer to this query for the timestamp T_1 is provided through a range search in the first tree version, which accesses the leaves A, B, C and D. Then, from the first of the above leaves that is not valid in the time T_2 (leaf A), we follow the F-pointer at a first step and the chain of ST-pointers

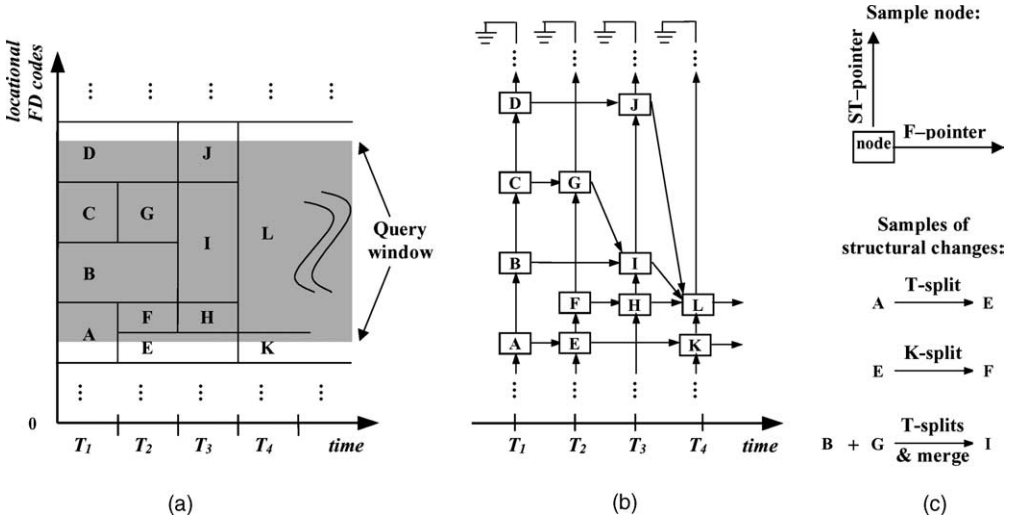

Fig. 4. (a) Layout of a set of data pages intersecting the shadowed area of the rectangle of a time-interval window query, (b) the corresponding forward linked lists, and (c) explaining the figure.

at a second step, and thus we locate all the leaves generated during time T_2 that contribute to the query (leaves E, F and G, respectively, in the sequential page access order). Therefore, the answer for the time T_2 is the union of the leaves that are shared between the two successive timestamps and the newly discovered leaves, i.e. in total the leaves E, F, B, G and D. While the last time-point of the time interval $[T_1, T_4]$ has not been reached, we proceed to the next tree version by repeating the above procedure for each leaf that contributes to the query answer in the remaining timestamps.

Note, that when we process the query for the tree of a certain time-point, we keep in main memory the collected leaves for this tree, as well as some of the accessed tree leaves of the preceding time-point (only those that are common between the two successive trees). Also, note that as we claimed in Section 3.6, all the FD code inserts, deletes and updates per timestamp are performed in an increasing quadcode order. Thus, the leaves that are generated in the same transaction time, are all created in an order that also enjoys the property of increasing quadcode range order. This means that for OLQ, there is no need to use the ST-pointer to connect two consecutively created leaves allocated within the same data page, and thus considerable disk space can be saved.

The horizontal leaf chaining proposed for OB^+ trees [17] uses four leaf pointers, the B-, BC-, F- and FC-pointers. This method was successfully adopted in OLQ [22], and in a similar manner can be adapted to TSLQ, MVLQ and MVASERI. However, it suffers from the *duplicate result problem* [30], since a quadcode may be reported more than once due to multiple visits of the same page. The approach proposed in the current paper avoids multiple accesses of the same page. It is also much simpler and boosts query processing and image insertions. More details can be found in [25].

Efficient algorithms for processing the five time-interval window queries discussed previously have been reported in [22]. These algorithms use the B-, BC-, F- and FC-horizontal pointers that link leaves of different trees and can be easily adapted to TSLQs, MVLQ, MVASERI and OLQ for the four new pointers B-, ET-, F-, and ST-, with slight modifications.

5. Performance comparison

5.1. Data generation

In order to compare the four Quadtree-based STAMs of Section 3, extensive experimentation using sequences of raster images is required. Due to the lack of large real data sets, the region data sets used in the experiments were sequences of artificial images following the real-world behavior of spatial data that change over time. For the generation of such data, we used the G-TERD [24], a generator especially designed for time-evolving regional data. The basic concepts that determine the function of G-TERD are: the two-dimensional workspace; the position and movement of a rectangular scene-observer over the workspace; the structure of complex regional objects, their color, maximum speed, zoom and rotation-angle per time-slot; the influence of other moving or static objects on the speed and on the moving direction of an object; the statistical distribution of each changing factor; and, finally, time. G-TERD supports also a variety of widely-used continuous and discrete data distributions. It is highly parameterized and very flexible in the simulation of a variety of real-world scenarios.

To demonstrate the use of G-TERD, in Figs. 5–7 we present a number of sample data sets. These are the data sets that were used for the experimentation purposes of this paper. The number of evolving images in each data set is $N = 200$ and the snapshots illustrated correspond to time-slots $T_i = 40$, 80, 120, 160 and 200. The image size in Fig. 5 is 2048×2048 pixels, whereas it is 1024×1024 pixels in the remaining figures.

5.1.1. Scenario 1

Scenario 1 in Fig. 5 illustrates complex static objects and a scene-observer of 2048×2048 pixels field of vision that shifts over the 6000×6000 pixels workspace. The speed of the scene-observer is constant and its orientation is diagonal, from southwest to northeast. Five data sets of $N = 200$ images each, were generated from this scenario. The only difference between them is the value of the speed of the scene-observer, i.e. 1, 3, 5, 7 or 9 pixels per time-slot in each spatial axis. The scenario involves 100 complex static objects comprised by 100 long-lived rectangular sub-objects each and uniform initial distribution of their centers, and a color palette of 16 colors.

5.1.2. Scenario 2

Fig. 6 presents Scenario 2, where complex moving, re-sizing and rotating objects appear from the bottom of the workspace. Each object comprises of 100 rectangular sub-objects. The orientation of the moving objects on the y-axis is south to north, whereas on the x-axis it is random and the speed is uniformly distributed in the domain $[-4, 4]$. Many new objects are created per time-slot and remain alive for a long time period. Therefore, the objects have sufficient time to cross the whole workspace. When they arrive at the upper edge, G-TERD prevents them from leaving the

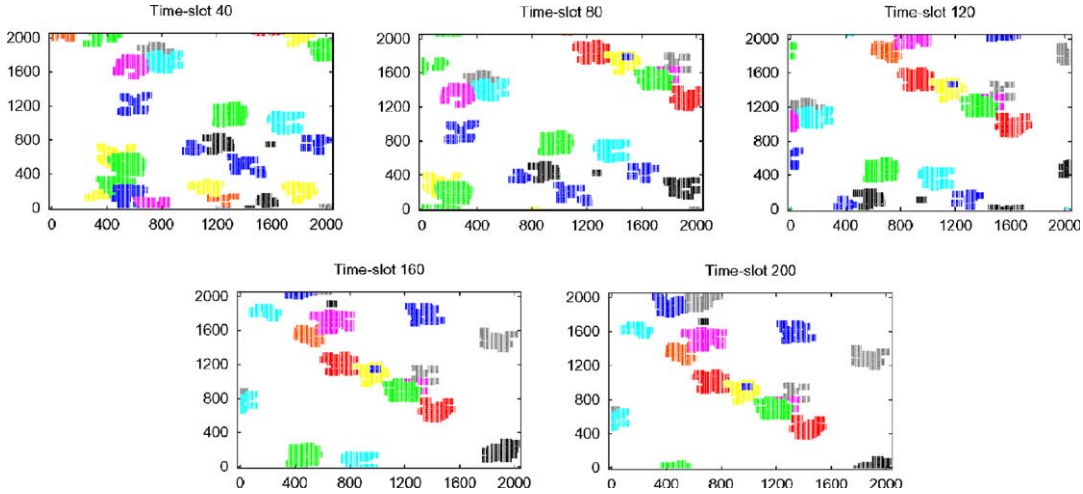

Fig. 5. Static objects and a scene-observer moving from southwest to northeast.

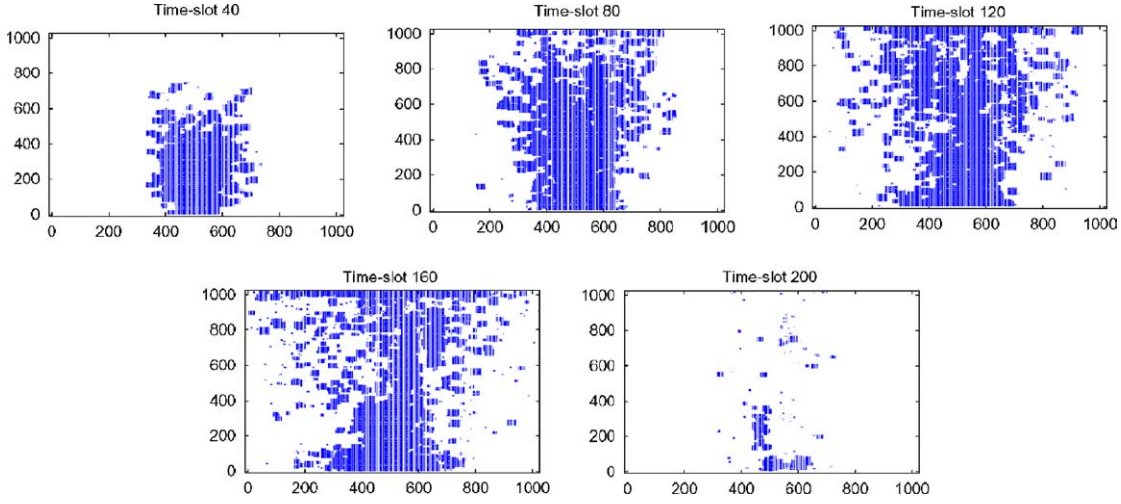

Fig. 6. Moving objects that appear at the bottom and cross the workspace.

workspace and they are thus scattered over the entire upper part of the workspace, until their deletion time comes. A static scene-observer which supports only black and white colors, monitors the time-evolving scene. Five different data sets of $N = 200$ images each, were generated and the parameter that varies was the speed of the objects on the y-axis (1, 3, 5, 7 or 9 pixels per time-slot).

5.1.3. Scenarios 3a and 3b

Scenario 3 is a complex scenario and Fig. 7 depicts some of its snapshots. This scenario is based on two sub-scenarios and merges their data sets. The first sub-scenario creates complex static objects that live for the whole scene lifetime. The second sub-scenario creates moving objects

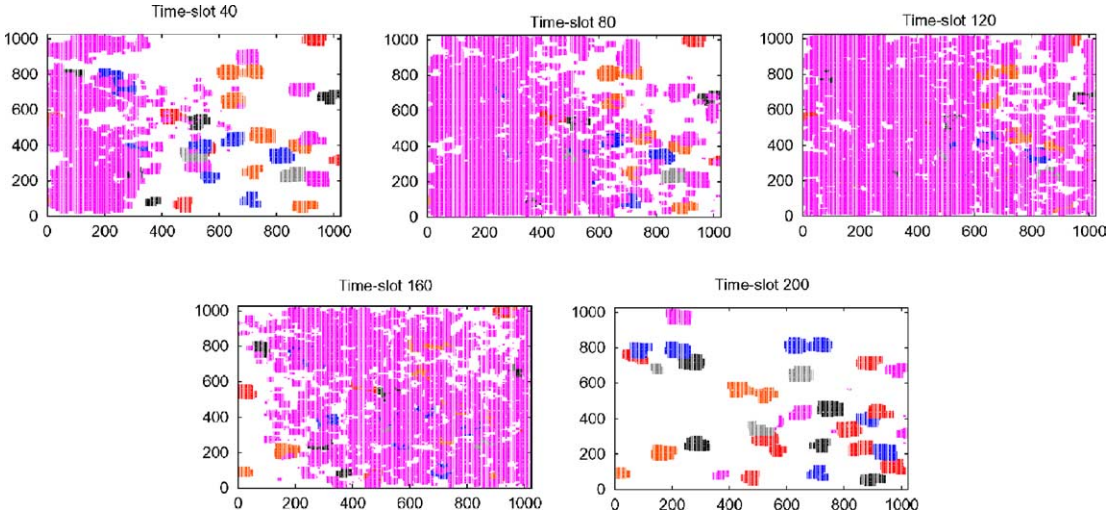

Fig. 7. A complex scenario that combines two sub-scenarios.

Table 3
Storage gain by using 2-bytes timestamps in node entries (page size 1024 bytes)

Access method	Actual timestamps		Relative timestamps		Storage gain	
	b_{leaf}	$b_{internal}$	b_{leaf}	$b_{internal}$	In b_{leaf} (%)	In $b_{internal}$ (%)
TSLQ	110	84	142	100	22.5	16.0
OLQ	10	126	10	126	0	0
MVLQ, MVASERI	76	62	110	84	30.9	26.2

generated along the y-axis and cross the workspace but each one at a different speed. The scene-observer is static, and 16 colors are supported.

In the experimentation that will be presented in the sequel, two different versions of scenario 3 were used, *scenarios 3a* and *3b*. In the first case, 50 static and 50 moving regional objects comprising 100 rectangular sub-objects each were generated. Five different data sets for five values of the maximal speed of the moving objects on x-axis were generated: 1, 3, 5, 7 and 9 pixels per time-slot, respectively. The minimal speed on the x-axis was set at zero, whereas the speed on the y-axis was uniformly distributed in the domain $[-2, 2]$. In the case of scenario 3b, five data sets of 200 images each were also generated. 50, 100, 150, 200 or 250 static and moving regional objects per data set (dedicated to each object class as two equal portions of half the number of objects) of 100 rectangular sub-objects each were created. The speed on the x- (y-)axis was randomly distributed between 0 (−2) and 5 (2) pixels per time-slot, and both the re-sizing and the rotation-angle were set between −1 and +1 pixels per time-slot. The snapshots illustrated in Fig. 7 correspond to one of the generated data sets.

Each 32-bit pixel of the above $4 \times 5 \times 200 = 4000$ images corresponds to a workspace pixel and it was originally representing a value in the scale of the number of colors of the corresponding color palette. We transformed each multicolored image to black and white, by considering as black all the pixels that had been colored (i.e. held a non-null color value) by G-TERD. Note that we are

not interested on the ratio between black and white color analogy because the images are not unobstructed synthetic images. [24] provides details for all the user-defined parameters and statistical models mentioned in this subsection.

5.2. Experimental setup

TSLQ, MVLQ, MVASERI and OLQ were implemented in C/C++, and all experiments were performed on a 450 MHz Pentium II-MMX PC, with 400 Mbytes RAM, running Microsoft Windows NT Server 4.0. The disk page size was 1024 bytes and 2048 bytes, the FD locational code size, the timestamp size and the pointer size were 4 bytes each (standard integer implementation in C/C++), whereas the FD code level size was 1 byte. As already mentioned in Section 3.5, in order to maximize overlapping in OLQ leaf level, the capacity of OLQ leaves was fixed to 10 quadcodes. Thus, a 1024-bytes page can host 14 consecutive OLQ leaves. On the other side, the temporal information stored in each node of TSLQ, MVLQ and MVASERI lowers the node capacity and decreases space utilization and query performance. Instead of storing the actual timestamps with 4 bytes each, we keep the *relative* timestamps, which is the actual time-point minus the *StartTime* value of the corresponding node. This way, we use only 2 bytes for each timestamp in node entries and the T_{start} and T_{end} of each node entry are in the range $[0, 65535]$. If this range is exceeded (e.g., entries have excessively long lifetimes), appropriate T-split and data duplication is introduced to ensure correctness. However, it is assumed that this should happen in practice very rarely. As Table 3 shows, this mechanism increases the node capacity in TSLQ, MVLQ and MVASERI substantially but it does not affect OLQ, since entries in OLQ nodes are not timestamped.

As mentioned previously, the number of evolving images is $N = 200$ per data set. Thus, the Header Table of OLQ and the root* table of MVLQ and MVASERI are very small and, thus, stored in main memory. The same holds also for the DF-expression, whose size in the worst case is 1.33 Mbytes for a 2048×2048 image and 0.33 Mbytes for a 1024×1024 image. For every image insertion (for converting it from raster to linear FD format), we used the algorithm OPTIMAL_BUILD of [14]. At the start of every experiment, the FD codes of the first image were inserted in an empty tree. The codes were inserted one at a time, as they were produced by OPTIMAL_BUILD.

In the sequel, we introduce an extensive experimental comparison of the four STAMs regarding storage requirements, index building cost, data redundancy, impact of buffering, query processing cost and several other important parameters that characterize their performance. We do not present graphs for some data sets or scenarios because they lead to the same observations as with the illustrated experiments.

We assume that the space efficiency of an access method for time-evolving regional data is measured by the number of occupied pages and by the values of the following parameters that describe intrinsic method characteristics.

Definition 1. Duplication ratio *DR* on an index is the average number of times that a single FD code appears in the index:

$$DR = \frac{R + R_{red}}{R} \tag{1}$$

Definition 2. Multiple version utilization *MVU* [9] is the fraction of the total data space occupied by all the non-redundant FD codes:

$$MVU = \frac{R}{M \times b} \tag{2}$$

where R does not include deletion markers [4] (if they exist) in order to have a fair comparison of the four STAMs. *MVU* measures how effectively a structure is, together with the particular T- and K-split policies and index construction thresholds, in supporting multiversion data. It expresses the cost of carrying multiple FD code versions, in terms of total space consumed by these retained versions. It, also, reflects the index maintenance cost for the entire image collection and the cost of storing redundant copies of the quadcode versions to support spatio-temporal queries. Therefore, *MVU* is correctly one of most representative parameters of the overall performance of an access method for multiversion data.

Definition 3. Single version current utilization *SVCU* [9] is the fraction of data pages containing FD codes alive at the current time occupied by these alive records:

$$SVCU = \frac{R_{now}}{M_{now} \times b} \tag{3}$$

where R_{now} and M_{now} are the number of alive quadcodes and the number of pages containing alive leaves, respectively, for the current timestamp. *SVCU* measures the occupancy of the current data and reflects how effective a method is in minimizing the current database space. In general, the smaller the current database, the smaller the I/O cost for an image insertion and the response of queries based on current time will be.

5.3. Index construction

5.3.1. Index construction thresholds

According to [21] and by taking into account the node capacity of MVLQ, the three threshold values are set as: $P_{MVLQ\_high} = 0.809 \times b$, $P_{MVLQ\_low} = 0.39 \times b$ and $P_{MVLQ\_cons} = 0.20 \times b$. According to [27], the two thresholds used in MVASERI index construction are set to: $P_{MVSERI\_high} = 0.80 \times b$ and $P_{MVSERI\_low} = 0.20 \times b$, respectively, for $b \geqslant 5$ and any analogy on quadcode inserts, deletes and updates. Also, according to [9] we set TSLQ $P_{TSLQ\_high} = 2/3 \times b$. All the above threshold values were selected in order to minimize the time split frequency, and hence the index size and data duplication without compromising the index construction time and the query time, in terms of I/O activity.

The only two thresholds which have not been decided upon by previous related studies, are the TSLQ page consolidation threshold P_{TSLQ_cons} and the OLQ page usefulness threshold P_{OLQ_usef}. Their values will be selected after an experimental comparison. In the sequel we present this experimental investigation for the P_{TSLQ_cons} consolidation threshold in the TSLQ nodes. The

[4] The use of deletion markers is the result of the drawback of the TSLQ method to handle efficiently quadcode deletions (see [25] for details).

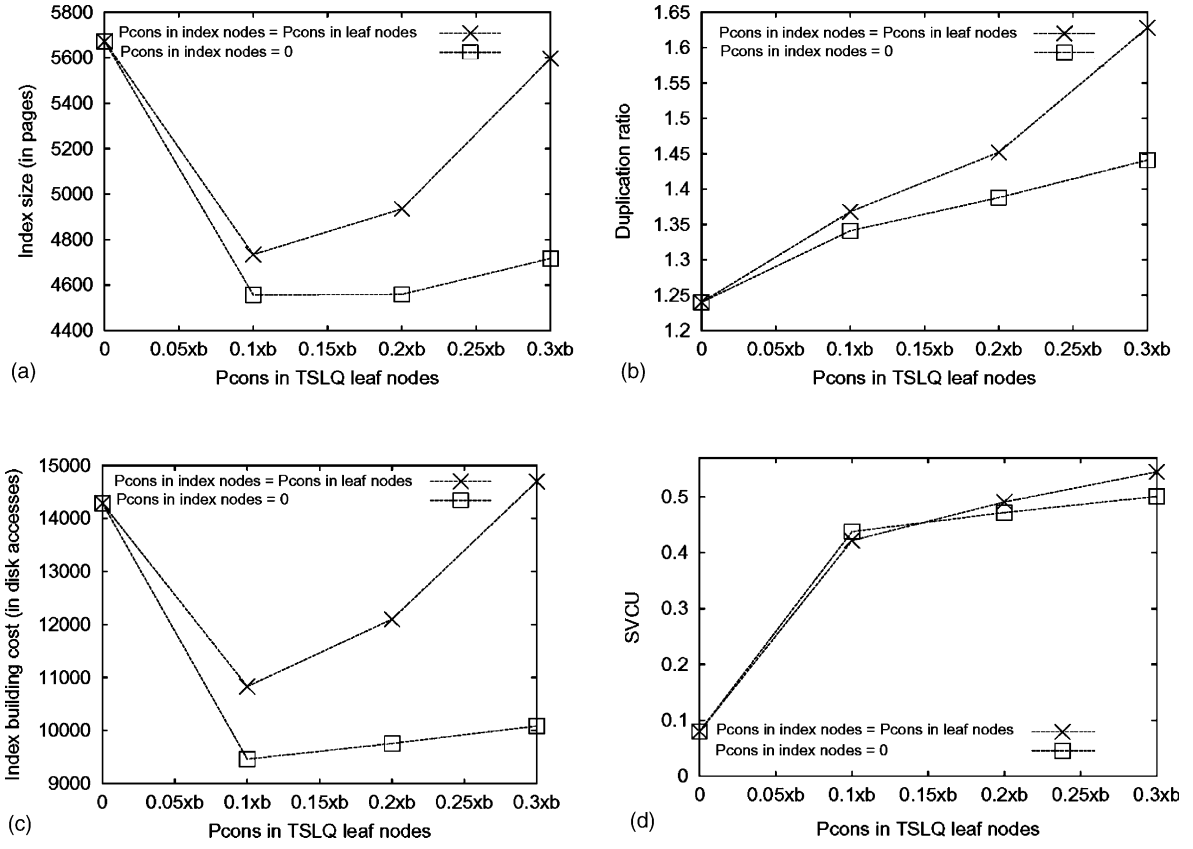

Fig. 8. (a) Disk space used, (b) *DR*, (c) I/O index building cost and (d) the TSLQ *SVCU* as a function of various P_{TSLQ_cons} values in the TSLQ nodes.

corresponding study for the P_{OLQ_usef} threshold in the OLQ leaf level leads to similar observations and thus it is omitted.

As mentioned earlier in the description of TSLQ method, when an internal node becomes historical after a T-split, then all its descendent nodes are also forced to T-split with the same split-time of the corresponding internal node. Thus, the appropriate P_{TSLQ_cons} value for internal levels is an interesting issue for experimentation.

Fig. 8 illustrates the behavior of TSLQ for various P_{TSLQ_cons} values in the leaf level and two different policies for its value in internal levels. In the first policy, the P_{TSLQ_cons} value in internal nodes equals to that in the leaf level. In the second policy this value is set to zero. The maximum P_{TSLQ_cons} value used in the experiments is $1/3 \times b$ since $P_{TSLQ\_cons} \leqslant 1/3 \times (b+1)$ must hold (see Section 3.2), whereas the data sets were generated by scenario 2. Fig. 8 shows that the P_{TSLQ_cons} increase in the internal levels is followed by an increase of the index size, the *DR*, the I/O building cost and the *SVCU*. This is because, on the one hand the node consolidation threshold helps in avoiding having nodes with zero records and in minimizing the number of pages of the current database; however, on the other hand, it increases the number of T-splits in index and data nodes.

In particular, the higher the P_{TSLQ_cons} in internal levels, the higher the number of T-splits required and the higher the corresponding additional data duplication.

Fig. 8a–c indicate that the total index size and the I/O building cost decreases when the $P_{TSLQ\_cons} = 0$ in internal levels. Especially when this is combined with a P_{TSLQ_cons} in leaves set at $0.10 \times b$ or $0.20 \times b$, the index size and the I/O building cost are 21% and 35%, respectively, smaller than the case of no P_{TSLQ_cons}. Fig. 8d shows that when the P_{TSLQ_cons} increases, the *SVCU* increases spectacularly and thus the space occupied by the current database is reduced. This figure illustrates that when $P_{TSLQ\_cons} = 0$ in internal nodes and $0.20 \times b$ in leaves, then the *SVCU* is very close to its maximal value (i.e. it is 90% of its maximal value). Therefore, for the rest of this study we fix these values in the TSLQ internal nodes and leaves, respectively, because this is the most satisfactory trade-off between total space, access time and dispersion into the disk of the quadcodes of the last inserted image. Fig. 8 shows that, although the P_{TSLQ_cons} threshold does not guarantee that for any past time-point the number of valid records in a node would be above this threshold, the space and the I/O building cost savings it offers are considerable. In an experimental study similar to the above, the P_{OLQ_usef} page usefulness threshold is also set at the 20% of the total page capacity in FD codes.

5.3.2. Storage requirements

Fig. 9a (Fig. 9b and c) compares the disk space occupied by each of the four STAMs as a function of the speed of the moving scene-observer (the speed of the objects, respectively) of the data sets generated from scenario 1 (scenarios 2 and 3a, respectively). An interesting observation is that increasing the speed of the scene-observer or the moving objects in the continuously changing scene, leads to an index size increase. This is because the faster the changes in the time-evolving scene are, the higher the *difference* between two consecutive raster images [5] of a data set will be. As a consequence, the common FD codes between two consecutive images are reduced and the quadcode inserts, deletes and updates per timestamp increase.

Another interesting observation stems from the slope of the curves of Fig. 9d, which depicts the index size as a function of the number of objects appearing in the time-evolving scene. One can see that the larger the number of objects appearing on the scene, the smaller the size for all the examined structures is. This holds because the increase of the number of static and moving objects of scenario 3b, leads to a workspace which is gradually more filled with moving objects and sub-objects (recall Fig. 7). Therefore, the corresponding black and white raster images generated by G-TERD form many large and solid black regions (''islands'') which remain common between consecutive images because of the large overlapping between the regional objects. This results in consecutive images having small differences and STAMs that do not need enormous storage. Especially, the case where 250 objects with 100 rectangular sub-objects each, are generated in scenario 3b, results to a packed workspace with $250 \times 100 = 25{,}000$ small static and moving rectangles on it. This time-evolving scene introduces many black and white raster images that have minimal differences as they are almost black.

[5] The *difference* between two consecutive raster images is defined as the percentage of different FD codes that their corresponding Quadtrees contain.

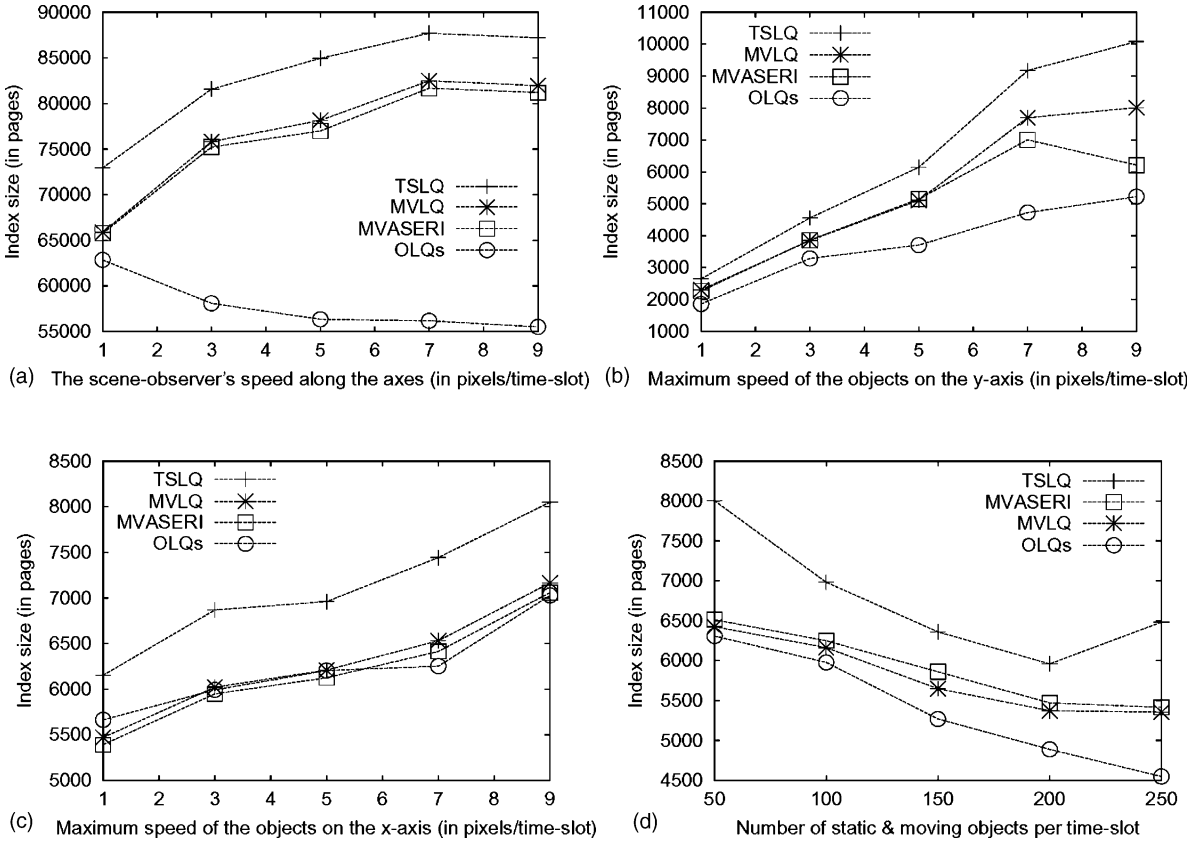

Fig. 9. Index size as a function of the data sets generated from (a) scenario 1, (b) scenario 2, (c) scenario 3a and (d) scenario 3b.

All the parts of Fig. 9 agree that (i) OLQ is superior than the other three access methods in terms of storage requirements, (ii) MVASERI and MVLQ demand about the same space, and (iii) TSLQ is clearly the most space consuming structure. The OLQ size is up to 30%, 40% and 50% smaller than that of MVASERI, MVLQ and TSLQ, respectively. This is due to the fact that there is no timestamping in OLQ node entries; thus, OLQ pages may contain up to 140 FD codes (14 leaves $\times$ 10 FD codes each). Therefore, from Table 3 we infer that the OLQ page may contain up to 21.5% more leaf records than in MVLQ and MVASERI, and the same maximum number of leaf records as TSLQ. On the other hand, TSLQ has larger node capacity than MVLQ and MVASERI but it can not take any advantage of it because of its inefficiency to handle logical deletions by means of deletion markers. TSLQ is based on the TSBT which is a method for temporal applications where the most frequent changes are object inserts and updates. Thus, in the case of evolving images that cause multiple quadcode deletions, the existence of deletion markers and pure K-splits that do not sweep historical data from the current database degrades TSLQ space performance.

It is also interesting that, even if the MVAS improves significantly the overall space used by MVBT in the temporal domain [27], the existence of batch inserts, deletes and updates per

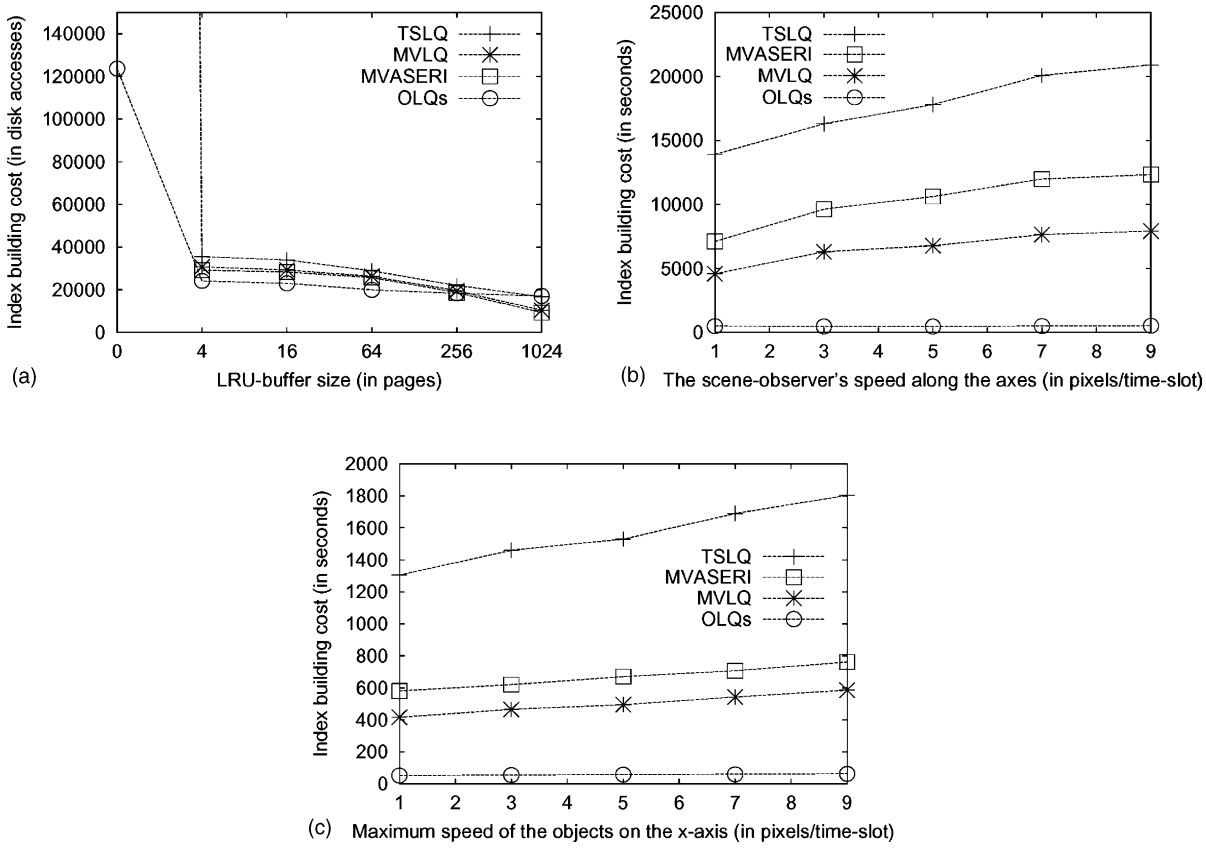

Fig. 10. (a) I/O building cost as a function of the LRU-buffer size for data sets of scenario 3a, (b) and (c) actual building time cost as a function of the data sets of scenarios 1 and 3a, respectively.

transaction time in the case of time-evolving regional data reduces this efficiency in MVASERI in comparison to MVLQ. However, it remains better than MVLQ in all our experiments.

5.3.3. LRU-buffer and index building cost

Another important component of our experiments is the buffer manager, for which we assume a least-recently-used (LRU) page replacement policy. Fig. 10a depicts the impact of the LRU-buffer size on the I/O building time cost of each access method. The data sets used correspond to scenario 3a. We experiment with buffer sizes of 0, 4, 16, 64, 256 and 1024 pages, or about 0%, 0.07%, 0.25%, 1.05%, 4.30% and 17%, respectively, of the average index size. From the figure it appears that after an LRU-buffer size of about 10 pages (i.e. 0.15% of the average index size), the number of accessed pages stabilizes regardless of the further increase of LRU-buffer size. This is because during an image insertion, its FD codes are firstly sorted in increasing order. Then, the building algorithm performs all the FD code modifications (i.e. inserts, deletes and updates) in a batch manner and increasing quadcode order. Therefore, every leaf is updated only once per transaction time and in most cases, it will not be accessed again in the same transaction time. This way, there

is no need to use large buffers. In the remaining of this study, it is assumed that the LRU-buffer size is 256 pages, unless otherwise stated.

One interesting aspect is the time cost for index construction. Fig. 10b and c show the index construction cost in terms of actual processing time for the four STAMs. The measured time includes CPU time, search time in the LRU-buffer, disk seek time and disk transfer rate. Both figures show a remarkably fast (slow) OLQ (TSLQ) construction time, which comes as a result of the simplicity (complexity) of its quadcode insert, delete and update algorithms. On the other hand, the index construction of MVLQ and MVASERI is from 10 to 20 times slower (from 2 to 3 time faster) than the corresponding one of OLQ (TSLQ). Both MVLQ and MVASERI structures spend a considerable amount of time when T-splits, K-splits or merges are performed.

Fig. 11 depicts the construction costs for all the examined methods in terms of disk accesses and indicates analogous conclusions as in the case of the storage requirements. Depending on the data set, the disk I/O for OLQ is on average 30% faster, 2% slower and 7% faster in scenarios 1, 3a and 3b, respectively, than the corresponding one for MVASERI, which presents the second-best performance. TSLQ pays large costs in index construction time although the existence of the P_{TSLQ_cons} in the leaf level is a beneficial parameter with respect to reducing the current database

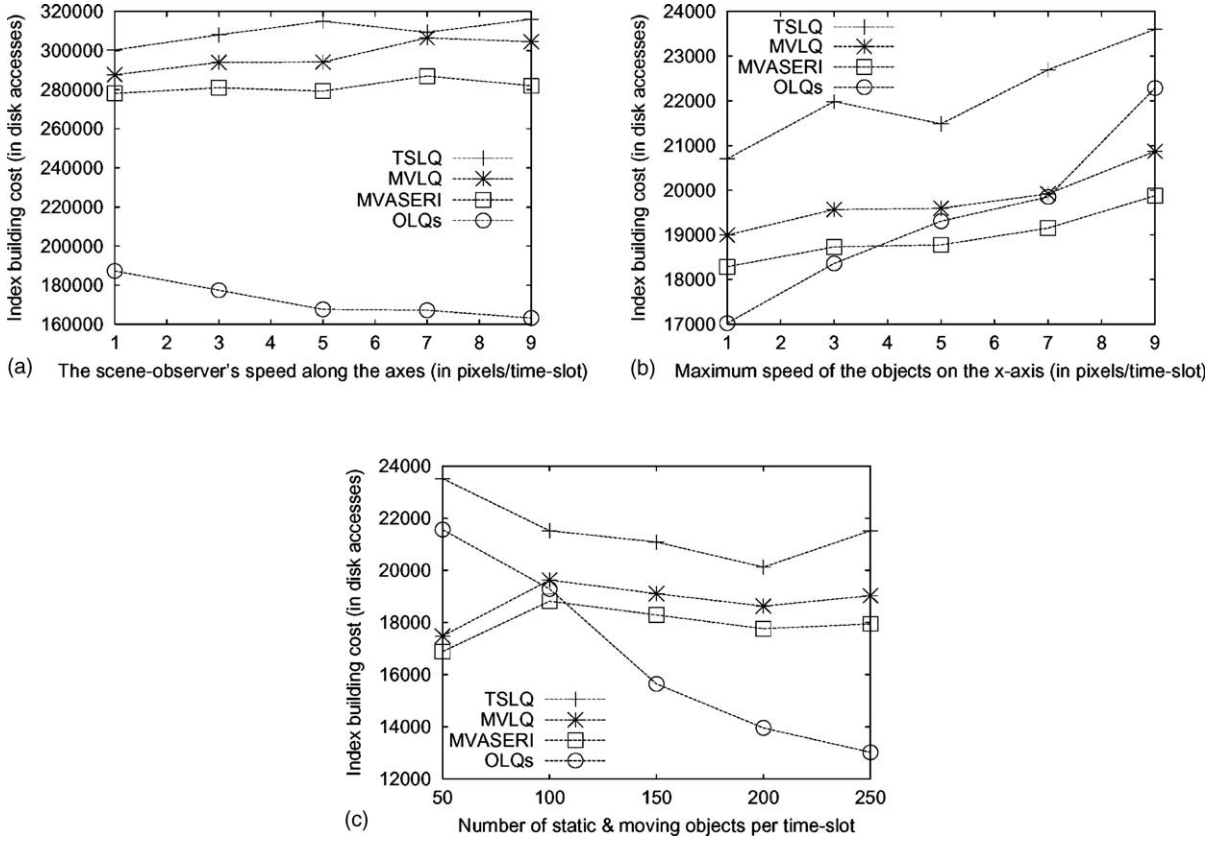

Fig. 11. I/O building cost for all the methods as a function of the data sets generated by G-TERD for (a) scenario 1, (b) scenario 3a and (c) scenario 3b.

size and the I/O building cost. Its performance is burdened mainly by its inefficiency in handling logical deletions.

Another interesting remark is that the results shown in Fig. 11a and b are not proportional to the actual time that was reported in Fig. 10b and c, respectively. This means that the construction of MVLQ, MVASERI and TSLQ is impeded by many CPU processes and buffer searches.

5.3.4. Duplication ratio

Fig. 12 refers to the duplication ratio *DR* and shows that the MVASERI and MVLQ spend less space for data redundancy by managing more carefully the number of T-splits performed. The left (middle) part of the figure compares the *DR* of the Quadtree-based STAMs as a function of the speed of the moving scene-observer (regional objects) of the data sets generated from scenario 1 (scenario 2, respectively). The right part of the figure compares the *DR* as a function of the number of regional objects in the time-evolving scene of scenario 3b. A general remark is that the higher the speed or the number of objects in the scenario is, the higher the percentage of difference between two consecutive images will be and the smaller the amount of duplication. This is because

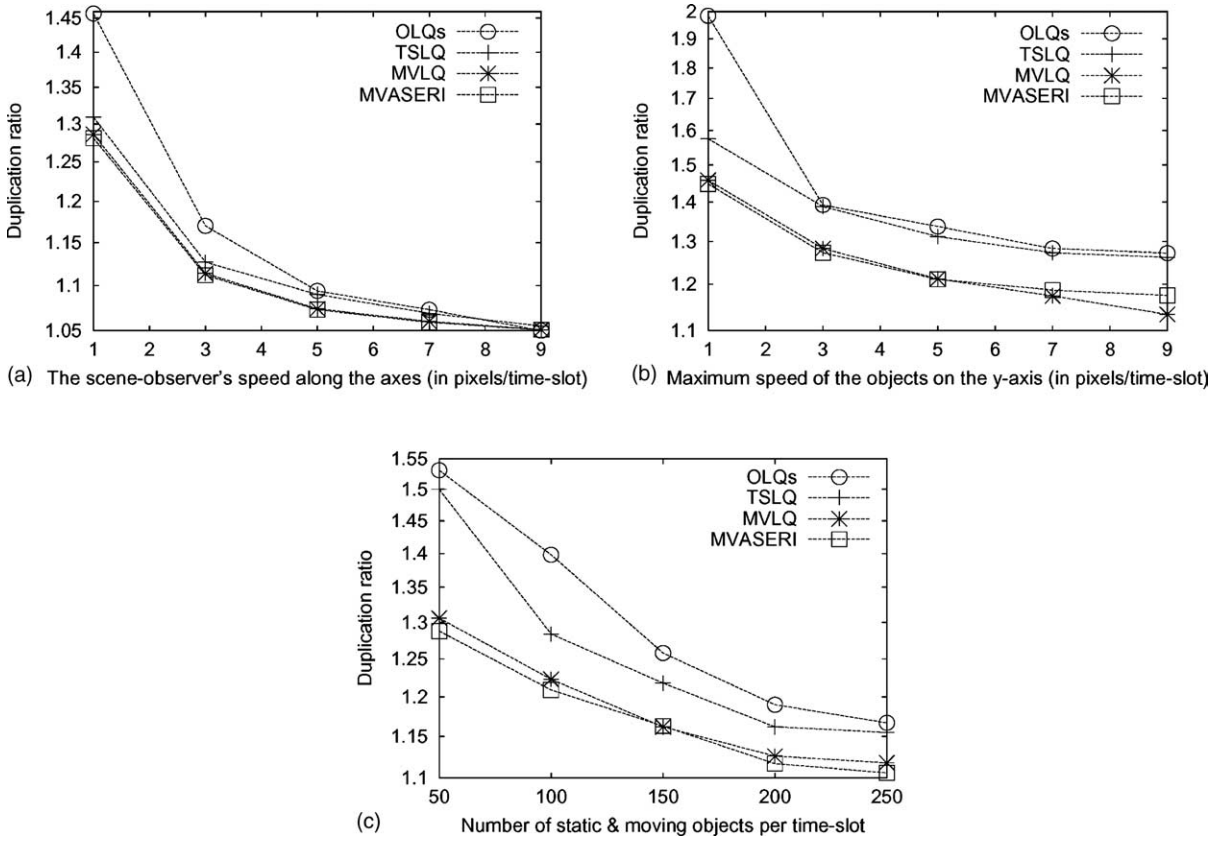

Fig. 12. Duplication ratio as a function of the data sets generated for the (a) scenario 1, (b) scenario 2, and (c) scenario 3b.

the common FD codes between two consecutive images are reduced and, consequently, so is the percentage of candidate records for duplication between successive tree versions.

5.3.5. MVU and SVCU

MVU and *SVCU* are parameters that measure the page utilization of a multiversion structure. Therefore, the capacity b which is used in the calculation of Eqs. (2) and (3), must be the total page capacity in records. In TSBT, MVLQ and MVASERI, the node capacity coincides with the page capacity. However, in the case of OLQ, a page may host 14 leaves of 10 quadcode records each, meaning that the b used in the equations equals $14 \times 10 = 140$.

The *MVU* parameter is studied in Fig. 13a (Fig. 13c) as a function of the data sets of scenario 1 (scenario 3a, respectively). The figure shows that OLQ outperforms all the other structures. Its *MVU* utilization is very close to 0.69, meaning that OLQ performance is comparable to having a forest of independent B^+-trees, without storing any records redundantly. That is, the redundant copies are being stored without compromising the OLQ storage utilization. The figure shows, also, that the *MVU* qualitative behavior of MVLQ, MVASERI and TSLQ structures does not seem to depend much upon the speed of the scene-observer or the number of the moving objects

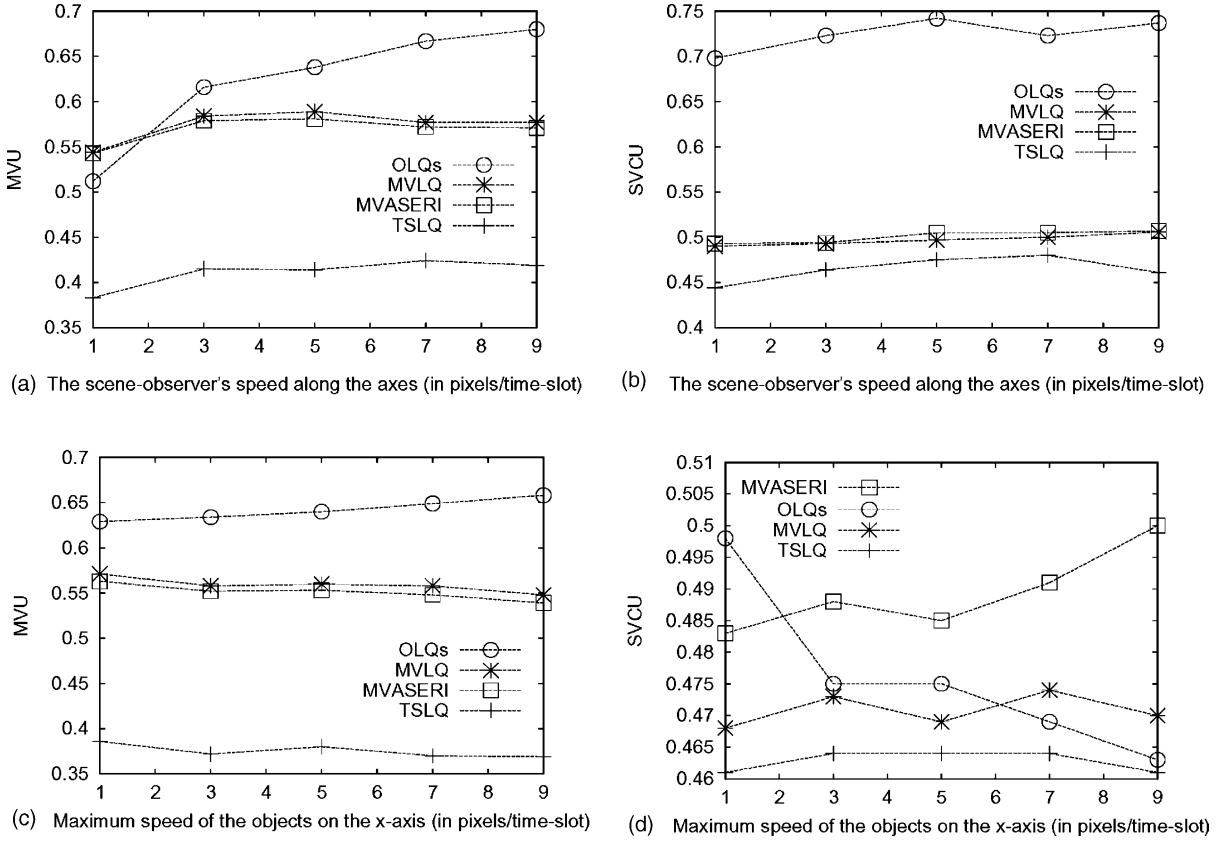

Fig. 13. (a) *MVU* and (b) *SVCU* as a function of the data sets of scenario 1, (c) *MVU* and (d) *SVCU* as a function of the data sets of scenario 3a.

appearing in the time-evolving scene. According to the *MVU* definition, a correlation between *MVU* and index sizes should exist. Indeed, the comparison of Figs. 9, 13a and c shows that high (low) *MVU* values correspond to access methods with low (high) storage requirements and confirm our expectations. As also expected, this comparison indicates that a stabilized (increased) *MVU* behavior leads to an increased (stabilized) index size.

In Fig. 13b (Fig. 13d) the *SVCU* is given for all the methods and for the data sets of scenario 1 (scenario 3a, respectively). As Eq. (3) demonstrates, *SVCU* measures the average page occupancy that a structure displays in the current database. Both figures indicate a stabilized *SVCU* performance for MVLQ, MVASERI and TSLQ at about 0.47, 0.49 and 0.46, respectively. On the other hand, the *SVCU* of OLQ depends highly on the scenario used. For the 2048×2048 images of scenario 1, the *SVCU* of OLQ is the highest and over 0.69, i.e. the average page utilization of B^+-trees. However, for the 1024×1024 images of scenario 3a, the OLQ *SVCU* provides an average value of about 0.47. To save space, we do not provide results from scenarios 2 and 3b, which, nevertheless, coincide with the corresponding results of scenario 3a.

The behavior of TSLQ, MVLQ and MVASERI in Fig. 13b and d as well as the behavior of OLQ in Fig. 13d indicate that their leaves contain some records that remain valid for long time periods. As a consequence, the current tree version is scattered over a large number of pages created in various timestamps. These 'long-life' pages may not violate the corresponding consolidation threshold but they may contribute a little to the current database. In the worst case, a long-life page may be accessed by the current database of TSLQ, MVLQ, MVASERI and OLQ because only $0.20 \times b$ of the contained records are alive ($P_{TSLQ\_cons} = P_{MVLQ\_cons} = P_{MVASERI\_cons} = P_{OLQ\_usef} = 0.20 \times b$, where b is the total page capacity in records).

In Fig. 13b with the experiment of the 2048×2048 images of scenario 1, OLQ outperforms the other three structures because the number of quadcode inserts, deletes and updates per new image is huge, the probability of a leaf update is very high and the number of 'long-life' leaves is thus very small. Fig. 14a illustrates the results with the same data sets, using a disk page of 2048 bytes. The figure provides results for two different OLQ leaf capacities: 10 and 20 FD codes per leaf. In the case of 10 records per leaf, the OLQ behavior is analogous to the one illustrated in Fig. 13b and confirms that the page size increase weakens the OLQ *SVCU* performance. In the case of 20 records per leaf, the probability of a leaf update increases and the *SVCU* performance of OLQ strengthens. Therefore, the proper setting of OLQ leaf capacity regarding the page size, the image size and the number of inserts, deletes and updates per timestamp may lead to high *SVCU* values. Fig. 14a indicates, also, that the page size increase does not affect the *SVCU* performance of TSLQ, MVLQ and MVASERI.

5.4. Query processing

The evolving binary images were snapshot views of the scenarios presented in Section 5.1. Query processing cost was measured for a static database state, i.e. no new image insertions were processed concurrently to the queries. Unless otherwise mentioned, the disk page is set at 1024 bytes. There are two classes of queries to study, the Snapshot Query and the five time-interval window queries mentioned in Section 4.1. In each query execution, we kept track of the average number of disk reads needed to satisfy the query. The results of some of the conducted experiments are reported below.

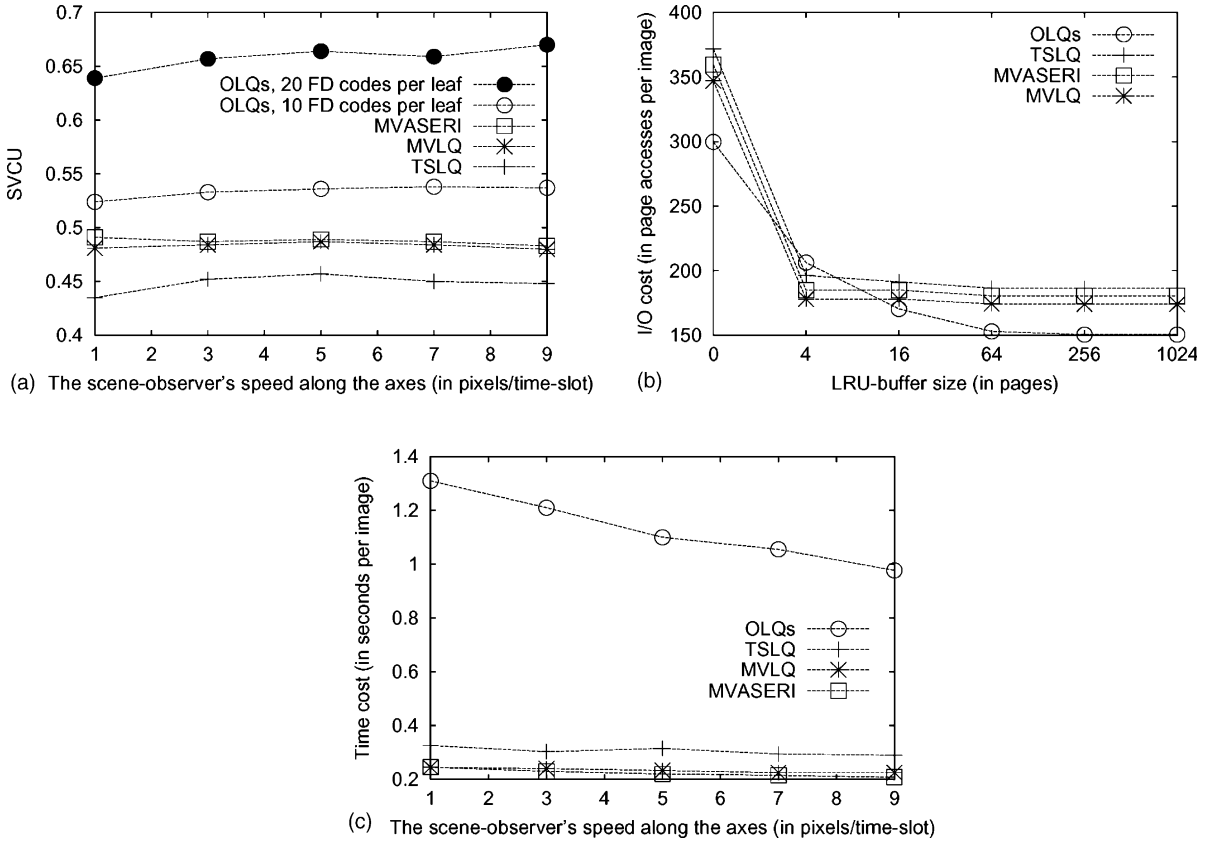

Fig. 14. (a) The *SVCU* for scenario 1 and page size 2048 bytes, (b) the Snapshot Query time response vs. LRU-buffer size for data sets from scenario 3a, (c) the Snapshot Query time response as a function of the data sets from scenario 1.

5.4.1. Snapshot queries

The Snapshot Query retrieves all the quadcodes that were valid at a certain timestamp T_i, where $i = 1, 2, \ldots, N$. Its algorithm performs independent searches through the TSLQ, MVLQ, MVASERI and OLQ root(s), as if all images were separately stored in Linear Quadtrees. This algorithm does not use the F- and ST- (or B- and ET-)pointers that link leaves of different trees. Fig. 14b compares the Snapshot Query time response of all the four STAMs for six LRU-buffer sizes. The data sets used in this experiment are G-TERD data sets generated by scenario 3a. The results are the average number of I/Os in the case of the Snapshot Query for each one of the $N = 200$ different versions.

The figure shows that for LRU-buffer sizes of more than the threshold of 4 pages (which is about the 0.07% of the index size in average), the query processing cost for the TSLQ, MVLQ and MVASERI is independent of the LRU-buffer size. This is because as we claimed earlier, for the Snapshot Query umbrella-like searches are performed in TSLQ, MVLQ, MVASERI and OLQ and records are located in increasing quadcode order. Thus, every TSLQ, MVLQ and MVASERI leaf is accessed only once per Snapshot Query and the same holds for the internal nodes when the LRU-buffer size is larger than the height of the respective tree. However, some OLQ pages

contributing to the answer may be accessed more than once if they host more than one leaf valid for the specific timestamp. As a result, the query I/O cost of OLQ that appears in Fig. 14b does not stabilize as fast as the corresponding I/O cost of the other three examined methods.

Figs. 14c and 15 depict the response for the Snapshot Query in terms of actual time (measured in seconds) and disk activity (measured in I/Os), respectively. The results are given for each one of the 200 timestamps per generated data set. The LRU-buffer size is fixed at 256 pages and it is cleared between two consecutive runs of the Snapshot Query. The comparison of Figs. 14c and 15a corresponding to the same scenario, shows that (as expected) the qualitative performance behavior of TSLQ, MVLQ and MVASERI structures in terms of actual time coincides the corresponding behavior of the I/O activity. However, it is interesting to note that OLQ response in actual time is slower than the corresponding response of the other three structures, whereas in disk activity it is faster. The reason is that by contrast to TSLQ, MVLQ and MVASERI, each OLQ leaf page is partitioned into many small leaves. Therefore, the Snapshot Query time response is burdened by additional CPU processes and buffer searches.

It is interesting, also, that TSLQ, MVLQ and MVASERI indicate almost the same behavior in terms of disk activity in all the scenarios. On the contrary, OLQ appear to perform up to 40%

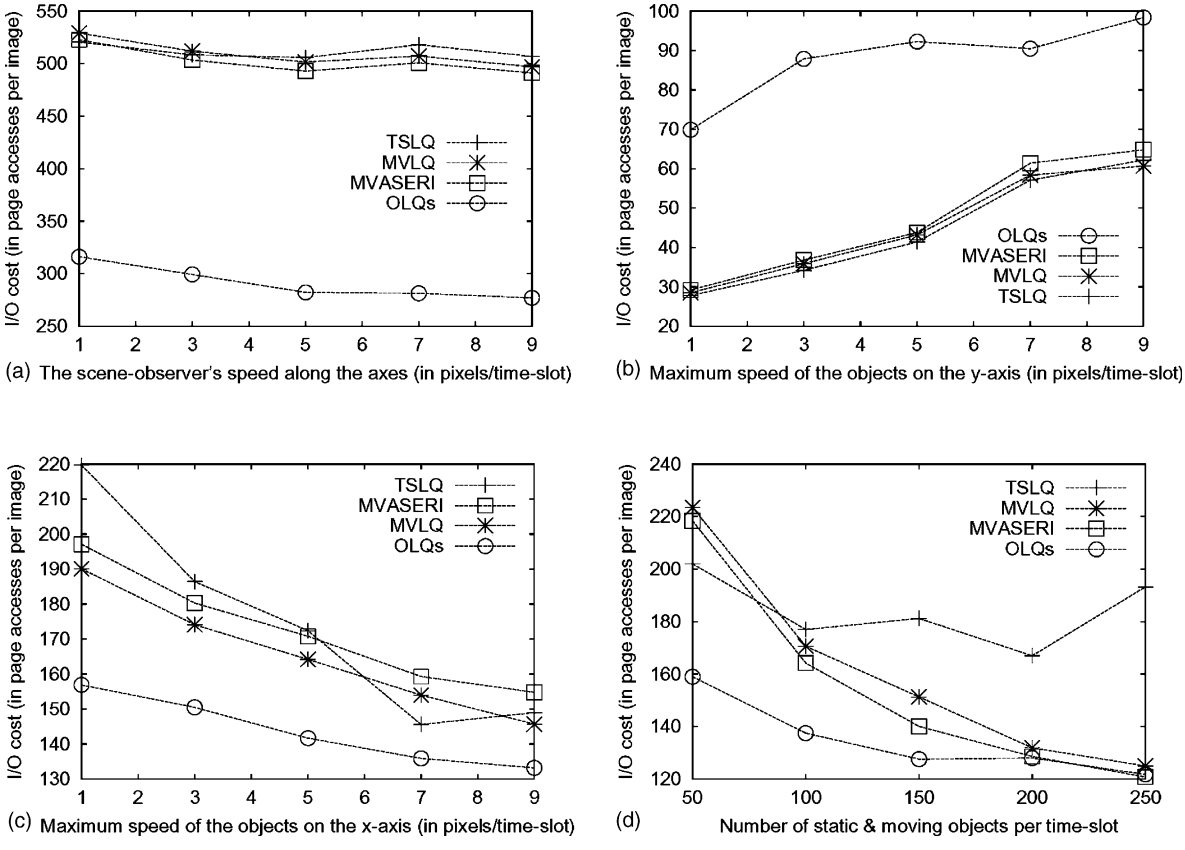

Fig. 15. Snapshot Query response time as a function of the data sets of (a) scenario 1, (b) scenario 2, (c) scenario 3a and (d) scenario 3b.

faster, 60% slower, 20% faster and 25% faster I/O processing in scenarios 1, 2, 3a and 3b, respectively. This OLQ query processing behavior is analogous to that of the *SVCU* parameter and the observations of Section 5.3.5 are valid here as well. The Snapshot Query may visit a large number of OLQ pages hosting only a few long-life leaves that are however alive for the time-point of interest. Our experiments with disk pages of 2048 bytes and the data sets of scenario 1, resulted in TSLQ, MVLQ, MVASERI and OLQ behavior similar to that of *SVCU* illustrated in Fig. 14a.

5.4.2. Time-interval window queries

In the sequel, we investigate the performance when querying spatial information during a time interval. The implementation of the algorithms of the five time-interval window queries is based on algorithms described in [22]. We argue that all the leaves that contain or should contain records contributing to the query for a certain timestamp must be kept in main memory. These leaves are needed in order to continue the process to the next timestamp. Thus, it is obvious that different LRU-buffer sizes would result into different response times. To avoid risking the use of an LRU-buffer size or a cashing technique that would bias for any structure, we decided to leave this issue for future investigation. For example, an average LRU-buffer size (in pages) for a fast Strict Containment Window Query response in MVASERI is

$$LRU\text{-}buffer\_size_{ave} = \left\lceil \left(\frac{k \times k}{2^n \times 2^n} \right) \times \left(\frac{\bar{R}_{T_i}}{b \times 1/4 \times \ln 5} \right) \right\rceil \tag{4}$$

where $k \times k$ and $2^n \times 2^n$ are the query window and image size, respectively, $\bar{R}_{T_i}$ is the average number of FD codes per timestamp and $1/4 \times \ln 5$ is the MVASERI average storage utilization per timestamp (its proof is analogous to the $\ln 2$ B-tree average storage utilization).

Below we report the average I/O activity for each of the five time-interval window queries presented in Section 4.1 for the time internal $[T_1, T_N]$, where $N = 200$ is the total number of inserted images. The qualitative performance behavior remains asymptotically the same for smaller intervals in the time domain. Queries with window sizes of 4×4, 16×16, 32×32, 64×64, 128×128 and 256×256 pixels were carried out for all structures. Every query was executed 100 times for every window size and every time for a randomly positioned window in the image space. The buffer was cleared between two consecutive runs of each query.

Fig. 16 depicts some diagrams for the Strict Containment Window Query for a 64×64 window size, whereas the query processing algorithm uses the horizontal F- and ST-pointers at the leave level. An interesting observation is that the shapes of all curves are almost the same as in Fig. 9, which gives the index size of each structure for the corresponding scenario. This clearly shows that the number of seeks (I/O cost) and that the data volume transferred to satisfy this query depend directly on the storage requirements of each method.

However, the results shown in Fig. 16a for the OLQ query response in scenario 1 are not consistent with our expectations with regards to the corresponding OLQ size. More specifically, the response of OLQ in the Strict Containment Window Query was expected to be faster than the corresponding MVASERI, MVLQ and TSLQ responses, regardless of the speed of the scene-observer. The explanation for this behavior is that in the case of a scene-observer with a low speed, the generated successive raster images exhibit minimal differences from each other and a relatively small number of quadcode inserts, deletes and updates is performed in a tree leaf. In

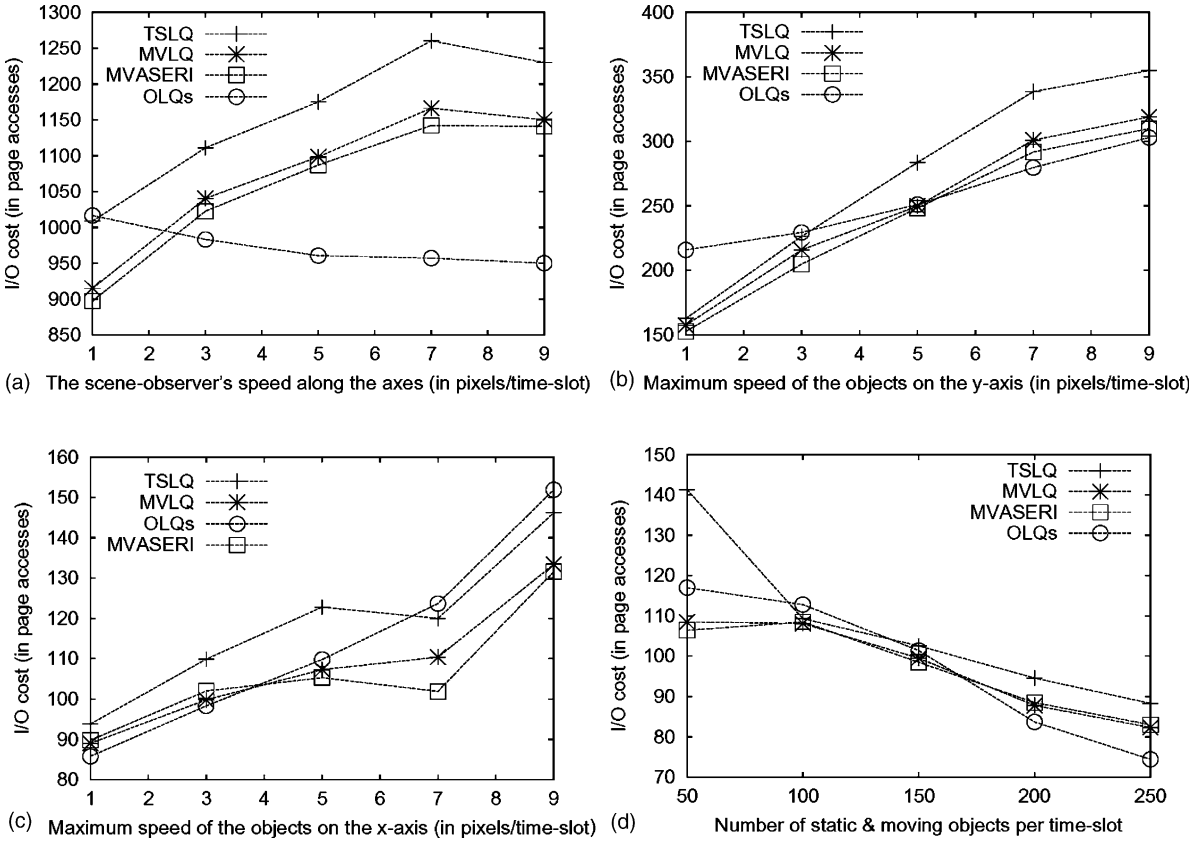

Fig. 16. I/O cost of the Strict Containment Window Query for 64×64 windows as a function of the data sets of (a) scenario 1, (b) scenario 2, (c) scenario 3a and (d) scenario 3b.

TSLQ, MVLQ and MVASERI, if a leaf has enough available free slots it may absorb this number of inserts, deletes and updates without performing a split. On the other hand, OLQ duplicates (T-splits) every leaf of the "ephemeral" Linear Quadtree before proceeding to any quadcode insert, delete or update. This behavior does not affect significantly the OLQ index size since every new leaf is only a fraction of a data page. However, it influences the time response of all the time-interval queries which use the horizontal F- and ST-pointers.

Fig. 16 indicates that when the difference between two consecutive images is small (e.g. when the scene-observer or the objects are moving slowly) MVASERI and MVLQ are up to 30% faster than OLQ and up to 22% faster than TSLQ. Reversely, when there is significant difference between two consecutive images, then OLQ are up to 15% faster than MVASERI and MVLQ and 25% faster than TSLQ.

The performance of TSLQ is always inferior to the performance of MVASERI and MVLQ because of its considerable index size and its low *MVU*, as a result of the existence of deletion marker records. This conclusion drawn for TSLQ holds for all our experiments in a consistent way.

In Fig. 17 the number of disk accesses is depicted for the Strict Containment Window Query as a function of the data sets of two different scenarios and for query windows with 128×128 and 256×256 sizes. The figure shows that the results for these window sizes are similar in shape to the corresponding results presented in Fig. 16 (with respect to 64×64 windows and the same scenarios) and are qualitatively proportional to the query size.

Fig. 18 shows some diagrams of the Border Intersect Window Query, as a function of the data sets of scenarios 2b and 3a, for various query window sizes. The plots are similar to the case of the Strict Containment Window Query and the corresponding data sets. The algorithmic handling of this query also uses the horizontal leaf lists in the examined access method. The results for the I/O time response on the General Border Intersect Window Query also have similarities in shape to those of the Strict Containment Window Query and, therefore we did not include them.

Fig. 19 refers to the Cover Window Query and shows the number of page accesses as a function of several various data sets and window sizes (i.e. 4×4, 16×16 and 32×32 pixels). This query can be answered through two different approaches [22]:

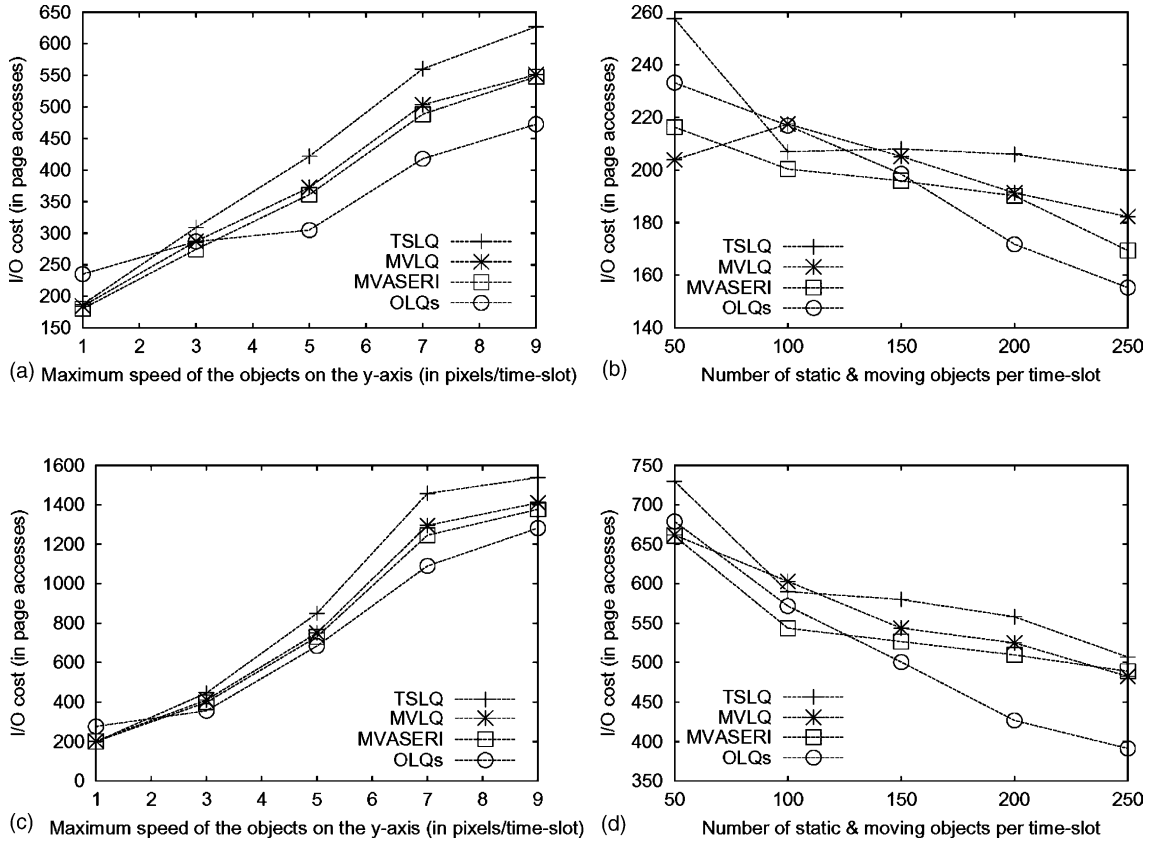

Fig. 17. I/O efficiency of the Strict Containment Window Query, as a function of the data sets of (a) scenario 2 for 128×128 windows, (b) scenario 3b for 128×128 windows, (c) scenario 2 for 256×256 windows, and (d) scenario 3b for 256×256 windows.

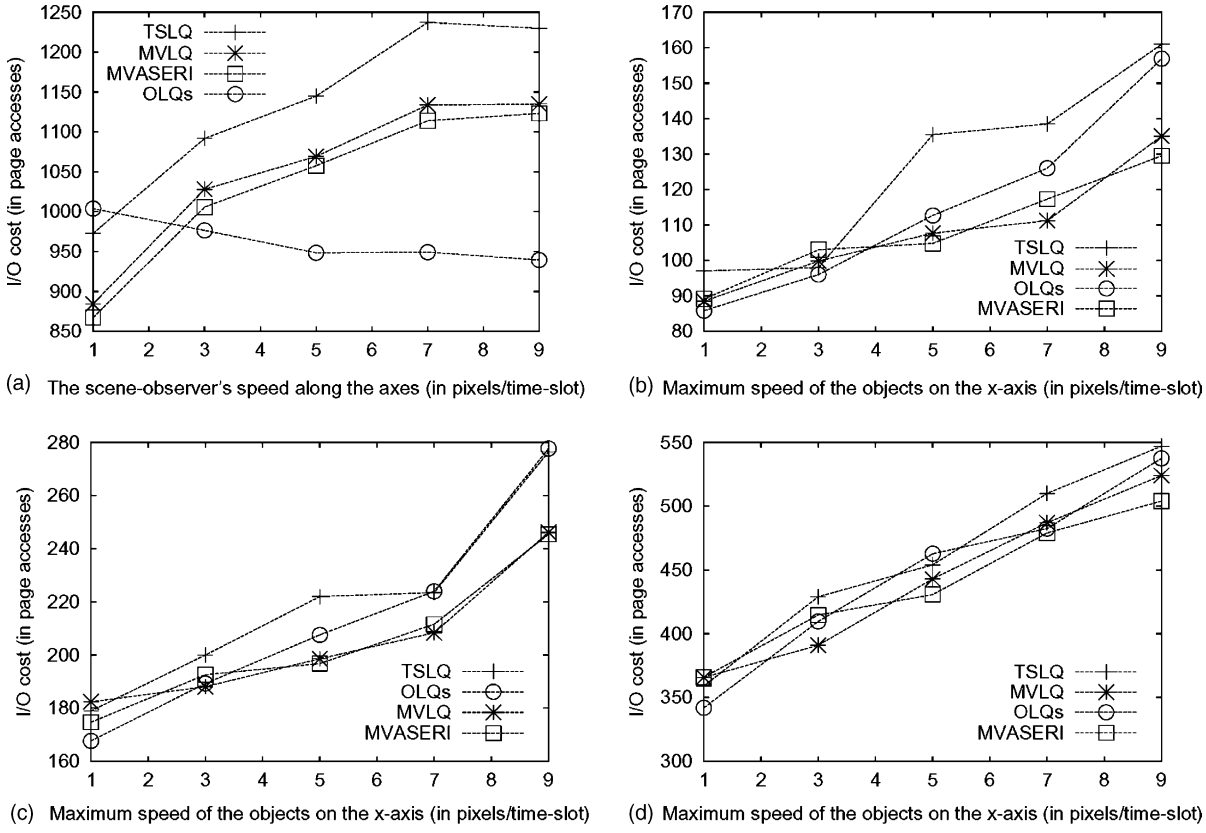

Fig. 18. I/O activity of the Border Intersect Window Query as a function of the data sets of (a) scenario 1 for 64×64 windows, (b) scenario 3a for 64×64 windows, (c) scenario 3a for 128×128 windows, and (d) scenario 3a for 256×256 windows.

1. by using the horizontal leaf linking approach (the F- and ST-pointers), or
2. by performing top-down searches per tree version and moving to the next timestamp when the YES/NO answer in the timestamp under process is concluded.

Tzouramanis et al. [22] claims that when two consecutive tree versions do not share much of their data, then the second algorithmic approach for the Cover Window Query outperforms the first one. This has also been confirmed by out own experiments. Thus, for the current experimentation, it was decided to perform top-down tree traversals and to not follow the horizontal leaf lists. The buffer is cleared between two consecutive query runs and not between two consecutive timestamps.

A general remark from Fig. 19 is that the OLQ approach leads to significantly higher I/O activity than the other three methods for the three window sizes. This is due to the fact that the algorithm of the Cover Window Query accesses not only leaves but also some internal nodes for each tree version in the TSLQ, MVLQ, MVASERI and OLQ structures. In the case of OLQ, most

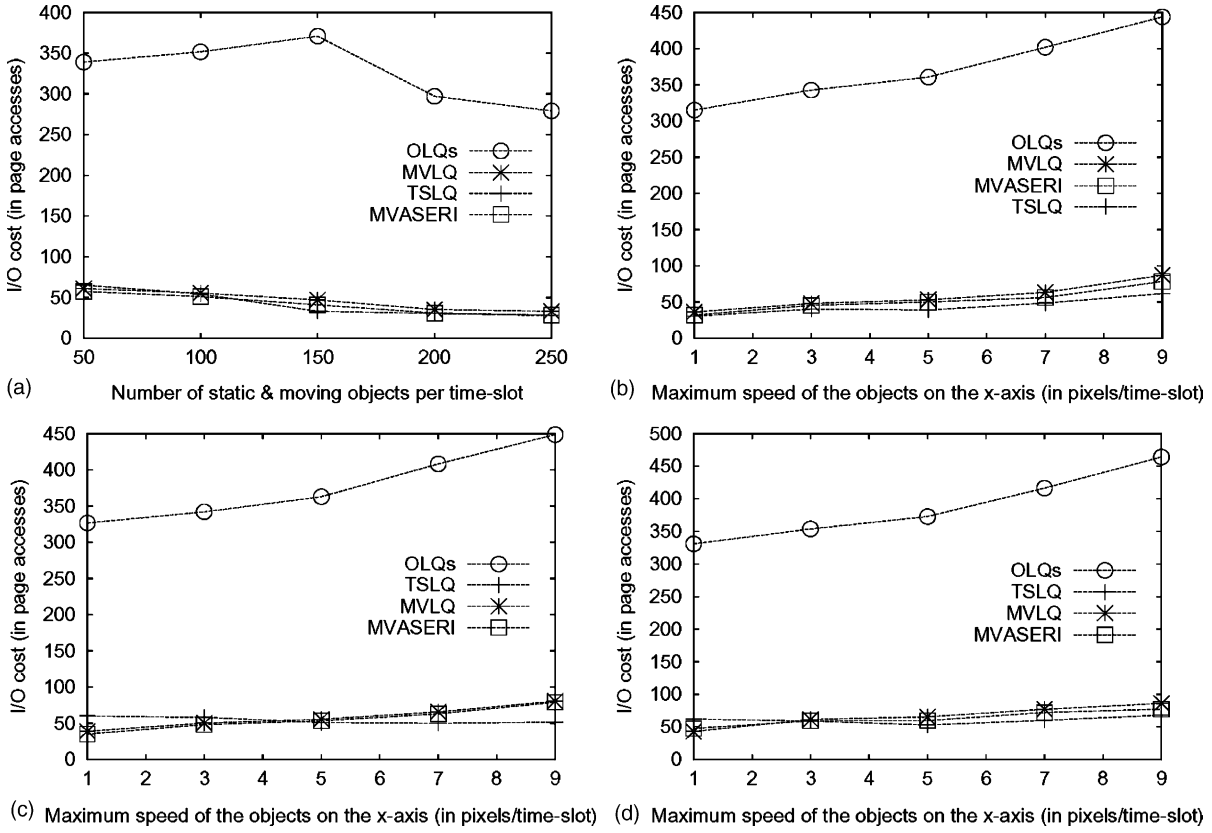

Fig. 19. Performance of the Cover Window Query as a function of the data sets of (a) scenario 3b for 4×4 windows, (b) scenario 3a for 4×4 windows, (c) scenario 3a for 16×16 windows, and (d) scenario 3a for 32×32 windows.

of the internal nodes are not shared between consecutive trees. The most representative example is the current OLQ root which is duplicated in every transaction time, even if a single FD code changes in the whole tree. Therefore, assuming that the number of similar but not identical raster images per scenario is $N = 200$, the cost for the Cover Window Query in OLQ will be at least 200 pages because of the corresponding tree roots.

This phenomenon does not hold in TSLQ, MVLQ and MVASERI since the corresponding modification algorithms duplicate an internal node only if it overflows. Otherwise, it is shared between consecutive tree instances and if an LRU-buffer with a proper size exists, the Cover Window Query accesses this page only once. In TSLQ, only one root exists for all the stored images. However, it can not take advantage of this against MVLQ and MVASERI, because the TSLQ size is larger than the size of the other two structures.

The last query examined is the Fuzzy Cover Window Query. Fig. 20 illustrates the time response of TSLQ, MVLQ, MVASERI and OLQ for two different scenarios and for query windows of various sizes. Recall that this query can take two different forms. However, both forms are processed with the same algorithm and, thus, they give the same query time response. This

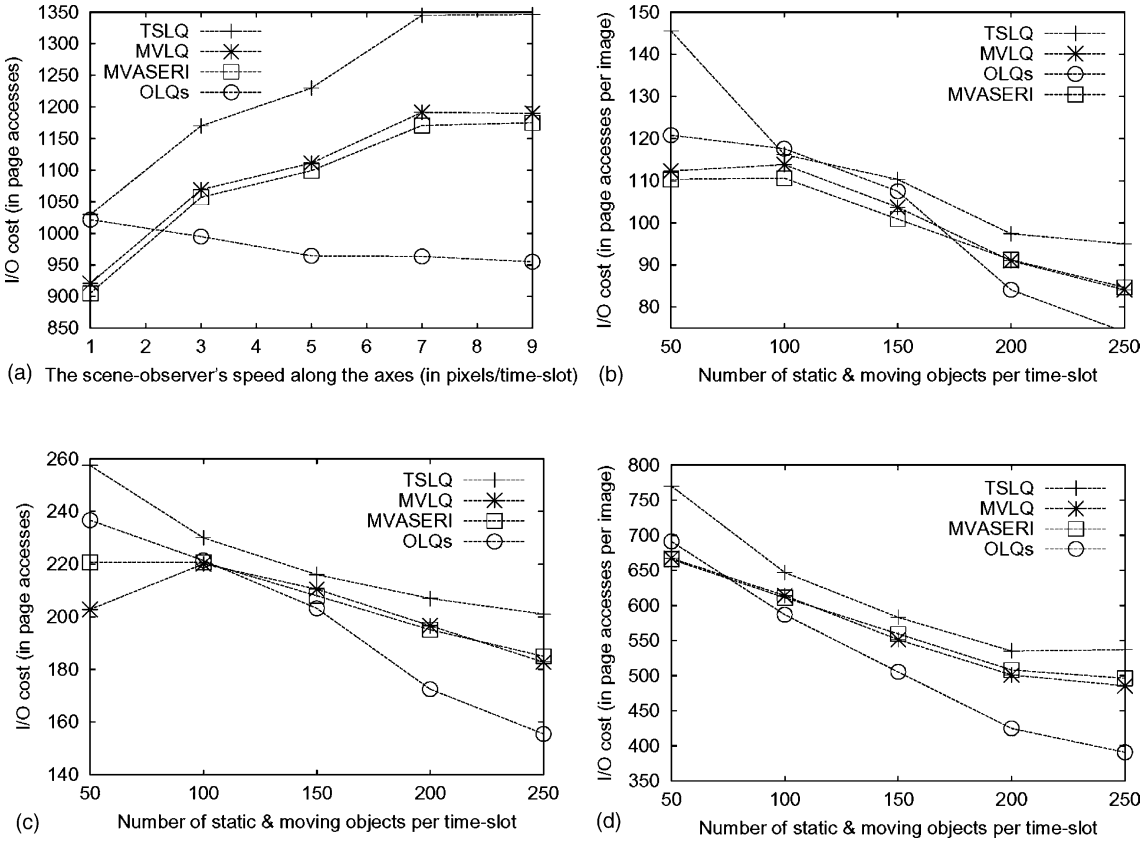

Fig. 20. I/O efficiency of the Fuzzy Cover Window Query, as a function of the data sets of (a) scenario 1 for 64×64 windows, (b) scenario 3b for 64×64 windows, (c) scenario 3b for 128×128 windows, and (d) scenario 3b for 256×256 windows.

algorithm uses the F- and ST-pointers to process the query. The observations are analogous to the case of the Strict Containment and the Border Intersect Window Queries for the corresponding data sets.

6. Discussion

Selecting the appropriate access method is crucial in achieving reasonable storage overhead and efficient information retrieval. Towards this goal, in the remaining of this section we discuss the major observations from a performance comparison of TSLQ, MVLQ, MVASERI and OLQ. These observations are also summarized in Table 4, according to the scenario used.

- OLQ are superior in comparison to all the other access methods in terms of storage requirements and index construction cost (actual time and I/O disk activity). It is notable that the building time of OLQ in seconds is at least 10 times faster than that for any other method. Also,

Table 4
Performance of TSLQ, MVLQ, MVASERI and OLQ according to the scenario used

	TSLQ			MVLQ			MVASERI			OLQ		
	Sce1	Sce2	Sce3	Sce1	Sce2	Sce3	Sce1	Sce2	Sce3	Sce1	Sce2	Sce3
Index size	F	G	G/F	F	G	VG/G	F	G	VG/G	VG	VG	VG
Building cost (s)	F	F	F	G	G	G	G	G	G	VG	VG	VG
Building cost (I/Os)	F	G	F/G	F	VG	VG/G	F	VG	VG/G	VG	VG	VG
DR	VG	G	G	VG	VG	VG	VG	VG	VG	VG	G	F
MVU	B	B	B	G	G	G	G	G	G	VG	VG	VG
SVCU	F	F	F	F	F	F	F	F	F	VG	F	F
Snapshot Query	F	VG	G/F	F	VG	G	F	VG	G	VG	F	VG
Strict Cont. W.Query	F	G	VG	G	VG	VG	G	VG	VG	VG	VG	VG
Border Int. W.Query	F	G	VG	G	VG	VG	G	VG	VG	VG	VG	VG
Gen.Bor.Int. W.Query	F	G	VG	G	VG	VG	G	VG	VG	VG	VG	VG
Cover W.Query	VG	VG	VG	VG	VG	VG	VG	VG	VG	VG	B	B
Fuzzy Cover W.Query	F	G	VG	G	VG	VG	G	VG	VG	VG	VG	VG

Explanations: Sce1 = scenario 1, Sce2 = scenario 2, Sce3 = scenarios 3a/3b, VG = Very Good, G = Good, F = Fair and B = Bad.

the *MVU* storage utilization parameter of OLQ approaches 69%, i.e. the B-tree average storage utilization, which means that OLQ store the redundant copies without any compromise. On the other hand, its I/O query response is at least as good as that of MVASERI for the four out the five time-interval window queries examined and in most of the cases for the Snapshot Query. However, the Cover Window Queries are inefficiently executed through OLQ. This is because the Cover Window Query performs top-down tree searches through all the OLQ roots and visits a large number of index nodes, most of which are not shared between consecutive tree versions.

- MVASERI provides the second-best performance. It is a rather robust access method that can address time-interval queries with good average performance, using only minimal extra space (about 10–30% more space than OLQ) and comparable or minimal extra building I/O cost with regard to OLQ. In relation to the other structures, MVASERI provides fast responses in the Snapshot Query and very fast responses in the Cover Window Query. It gives the smaller duplication ratio reducing thus the size of the historical database. It is superior to MVLQ and TSLQ in almost all our experiments. However, even if MVAS improves significantly the overall space used by MVBT in temporal databases, the existence of batch inserts, deletes and updates per transaction time for time-evolving regional data reduces this efficiency in MVASERI in favor of MVLQ.
- TSLQ is based on the TSBT which is a transaction-time access method designed for cases where the most frequent changes are object inserts and updates. In the face of evolving images that cause multiple quadcode inserts, deletes and updates per time-slot, the existence of deletion markers and pure K-splits that do not sweep historical data from the current database degrades TSLQ space and time performance. To deal with this drawback, we set the leaf consolidation threshold that is an important prerequisite for more compactly clustered alive records in current versions, thus improving the index size, I/O building cost and query response time.
- The page size increase does not affect much the performance of TSLQ, MVLQ and MVASERI. By contrast, OLQ are very sensitive to the page size. When the page size increases the search performance of OLQ deteriorates because the larger the page size, the higher the number of pages containing long-life OLQ leaves is. This affects greatly the *SVCU* storage utilization parameter, the Snapshot and the Cover Window Query response. However, the proper setting of the OLQ leaf capacity is capable of reducing the size of the current database, the Snapshot and the Cover Window Query response time.
- The storage requirements of all Quadtree-based STAMs depend mainly on the extent at which the time-evolving regional data change. The larger the difference from one image to the next, the larger the storage requirements. OLQ is the most sensitive structure in this respect. However, OLQ have the advantage that, depending on the image difference, its performance can become effective by the proper setting of its leaf capacity. Small or large image difference requires respectively very small or small OLQ leaves, in order to avoid the existence of many long-life leaves that degrade its *SVCU* value and its Snapshot and Cover Window Query responses.

7. Conclusions

In the present paper, we focused in benchmarking access methods for time-evolving regional data. We compared four Quadtree-based STAMs, the Time-Split Linear Quadtree (TSLQ), the Multiversion Linear Quadtree (MVLQ), the Multiversion Access Structure for Evolving Raster Images (MVASERI), and Overlapping Linear Quadtrees (OLQ), by tuning several parameters that affect their performance. None of the access methods studied can be viewed as the best in all cases, e.g. regardless of the application. However, in general:

- OLQ is the most space efficient method. Its index construction is impressively faster than those of the other examined structures. It performs in snapshots and time-interval window queries as good as or better than MVASERI. However, OLQ appears to be inefficient for the execution of the Cover Window Query. Its most important advantage is the simplicity of its implementation.
- MVASERI is always the fastest structure in Cover Window Query, whereas its response in snapshot and time-interval window queries is good in general. It is a good alternative if some extra space and time for index construction can be tolerated.
- MVLQ is superior compared to TSLQ and slightly inferior than MVASERI, in almost all our experiments.
- TSLQ is clearly the method with the greatest costs in space, building time and search response.

In summary, OLQ are the best choice for the manipulation of sequences of evolving raster images. Considering that with current technology storage is not an issue in comparison to the time performance, MVASERI could be chosen as an acceptable alternative method.

Another contribution of this paper is the design of a new technique to handle efficiently time-interval queries. This new technique links the leaves of different tree versions through "horizontal" pointers and provides very fast query response without reporting duplicated data. The only expense is the space overhead of holding four pointers per data node.

In the future we plan to develop a declustering scheme for the nodes of the TSBT, MVLQ, MVASERI and OLQ in order to exploit I/O parallelism for snapshot and time-interval window queries. Another interesting path for future research is the proposal of algorithms for other spatio-temporal queries (such as window queries with view point changing in time, spatio-temporal joins, as well as spatio-temporal nearest neighbor queries) in the context of Quadtree-based STAMs, and the investigation of their performance. For example, consider answering time-interval window queries while the scene-observer of G-TERD zooms in or out. Assuming that the window of the query remains constant in relation to the field of vision of the scene-observer, if the scene-observer zooms-in, the regional objects will appear larger and fewer objects will be in the query area. This query looks like a pyramid [6] in the three-dimensional space where one dimension stands for the time domain.

Suggestions for further research would also be to perform experiments based on similar artificial images created by shifts, rotations, scaling and other transformations. Besides, we believe it would be interesting to examine the presented methods with grayscale and/or colored images. Finally, it is important to note the generality of the methodology of transforming TSBT, MVBT,

MVAS and OB$^+$ trees into efficient STAMs by embedding ideas from the Linear Quadtree. Thus, this methodology can also be applied into other transaction-time access methods (especially to the B-tree-based ones).

Acknowledgements

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

References

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, An asymptotically optimal multiversion B-tree, The VLDB Journal 5 (4) (1996) 264–275.
[2] J. Clifford, C. Dyreson, T. Isakowitz, C.S. Jensen, R.T. Snodgrass, On the semantics of 'NOW' in temporal databases, ACM Transaction on Database Systems 22 (2) (1997) 171–214.
[3] J.R. Driscoll, N. Sarnak, D.D. Sleator, R.E. Tarjan, Making data structures persistent, Journal of Computer and System Sciences 38 (1989) 86–124.
[4] I. Gargantini, An effective way to represent Quadtrees, Communications of the ACM 25 (12) (1982) 905–910.
[5] C.S. Jensen et al. (Eds.), A Consensus Glossary of Temporal Database Concepts, ACM SIGMOD Record 23 (1) (1994) 52–64.
[6] G. Kollios, D. Gunopulos, V.J. Tsotras, A. Delis, M. Hadjieleftheriou, Indexing animated objects using spatiotemporal access methods, IEEE Transactions on Knowledge and Data Engineering 13 (5) (2001) 758–777.
[7] E. Kawaguchi, T. Endo, On a method of binary picture representation and its application to data compression, IEEE Transactions on Pattern Analysis and Machine Intelligence 2 (1) (1980) 27–35.
[8] D. Lomet, B. Salzberg, Access methods for multiversion data, in: Proceedings ACM SIGMOD Conference, 1989, pp. 315–324.
[9] D. Lomet, B. Salzberg, The performance of a multiversion access method, in: Proceedings of the ACM SIGMOD Conference, 1990, pp. 354–363.
[10] Y. Manolopoulos, G. Kapetanakis, Overlapping B$^+$ trees for temporal data, in: Proceedings Fifth Jerusalem Conference on Information Technology (JCIT'90) Jerusalem, Israel, 1990, pp. 491–498.
[11] M.A. Nascimento, J.R.O. Silva, Towards historical R-trees, in: Proceedings ACM Symposium on Applied Computing (ACM-SAC'98), 1998, pp. 235–240.
[12] M.A. Nascimento, J.R.O Silva, Y. Theodoridis, Evaluation for access structures for discretely moving points, in: Proceedings International Workshop on Spatio-Temporal Database Management (STDBM'99), 1999, pp. 171–188.
[13] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990.
[14] H. Samet, Applications of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990.
[15] T. Sellis, M. Koubarakis, et al. (Eds.), Spatiotemporal Databases—The Chorochronos Approach, LNCS Series, vol. 2520, Springer Verlag, 2003.
[16] B. Saltzberg, V.J. Tsotras, A comparison of access methods for time evolving data, ACM Computing Surveys 31 (2) (1999) 158–221.
[17] T. Tzouramanis, Y. Manolopoulos, N. Lorentzos, Overlapping B$^+$-trees: an implementation of a temporal access method, Data and Knowledge Engineering 29 (3) (1999) 381–404.
[18] Y. Tao, D. Papadias, Efficient historical R-trees, in: Proceedings IEEE Conference on Scientific and Statistical Database Management (SSDBM'01), 2001, pp. 223–232.

[19] Y. Tao, D. Papadias, MV3R-Tree: a spatio-temporal access method for timestamp and interval queries, in: Proceedings 27th Conference on Very Large Data Bases Conference (VLDB'01), Rome, 2001, pp. 431–440.
[20] Y. Theodoridis, T. Sellis, A. Papadopoulos, Y. Manolopoulos, Specifications for efficient indexing in spatiotemporal databases, in: Proceedings Seventh Conference on Statistical and Scientific Database Management Systems (SSDBM'98), Capri, Italy, 1998, pp. 123–132.
[21] T. Tzouramanis, M. Vassilakopoulos, Y. Manolopoulos, Multiversion linear Quadtree for spatio-temporal data, in: Proceedings Fourth East-European Conference on Advanced Databases and Information Systems (ADBIS-DASFAA'00), Prague, Czech Republic, 2000, pp. 279–292.
[22] T. Tzouramanis, M. Vassilakopoulos, Y. Manolopoulos, Overlapping linear Quadtrees and spatio-temporal query processing, The Computer Journal 43 (4) (2000) 325–343.
[23] T. Tzouramanis, M. Vassilakopoulos, Y. Manolopoulos, Time split linear Quadtree for indexing and querying image databases, in: Proceedings Eighth IEEE International Conference on Image Processing (ICIP'01), Thessaloniki, Greece, 2001, pp. 733–736.
[24] T. Tzouramanis, M. Vassilakopoulos, Y. Manolopoulos, On the generation of time-evolving regional data, GeoInformatica 6 (3) (2002) 207–231.
[25] T. Tzouramanis, M. Vassilakopoulos, Y. Manolopoulos, Designing and benchmarking access methods for time-evolving regional data, Technical Report, Data Engineering Lab, Dept. of Informatics, Aristotle University, Greece, 2002. Address for download: http://delab.csd.auth.gr/~theo/TechnicalReports.
[26] Y. Theodoridis, M. Vazirgiannis, T. Sellis, Spatio-temporal indexing for large multimedia applications, in: Proceedings Third IEEE Conference on Multimedia Computing and Systems (ICMCS'96), 1996, pp. 441–448.
[27] P.J. Varman, R.M. Verma, An efficient multi-version access structure, IEEE Transactions on Knowledge and Data Engineering 9 (3) (1997) 391–409.
[28] X. Xu, J. Han, W. Lu, RT-tree—an improved R-tree index structure for spatiotemporal databases, in: Proceedings Fourth International Symposium on Spatial Data Handling (SDH'90), 1990, pp. 1040–1049.
[29] J. Zobel, A. Moffat, K. Ramamohanarao, Guidelines for presentation and comparison of indexing techniques, ACM SIGMOD Record 25 (3) (1996) 10–15.
[30] D. Zhang, V.J. Tsotras, B. Seeger, Efficient temporal join processing using indices, in: Proceedings IEEE International Conference on Data Engineering (ICDE'02), 2002, pp. 103–113.

Theodoros Tzouramanis received a 5-year B.Eng. (1996) in Electrical and Computer Engineering and a Ph.D. (2002) in Informatics, both from the Aristotle University of Thessaloniki. He is currently a Lecturer at the Department of Information and Communication Systems Engineering of the University of the Aegean. He has also been with the Department of Sciences of the Technological Educational Institute of Thessaloniki. He has published over 10 papers in refereed scientific journals and conference proceedings. The citations to his research work exceed 50. His main research interests are access methods and query processing for temporal and spatio-temporal databases, multimedia databases and databases for Geographical Information Systems.

Michael Vassilakopoulos was born in Thessaloniki, Greece in 1967. He received a B.Eng. (1990) in Computer Engineering and Informatics from the University of Patras and a Ph.D. (1995) in Electrical and Computer Engineering from the Aristotle University of Thessaloniki. His B.Eng. thesis deals with Algorithms of Computational Geometry and his doctoral thesis deals with Spatial Databases. Apart from the above, his research interests include Access Methods, Spatio-temporal Databases, WWW Databases, Multimedia and GIS. He has published over 35 papers/chapters in refereed scientific journals, conference proceedings and books. He has been with the Department of Applied Informatics of the University of Macedonia, the Department of Informatics of the Aristotle University of Thessaloniki. For three years he also served the greek Public Administration as an Informatics Engineer where he participated in the evaluation and supervision of several information technology projects. Currently, he is an Associate Professor at the Department of Informatics of the Technological Educational Institute of Thessaloniki.

Yannis Manolopoulos was born in Thessaloniki, Greece in 1957. He received a B.Eng. (1981) in Electrical Eng. and a Ph.D. (1986) in Computer Engineering, both from the Aristotle University of Thessaloniki. Currently, he is Professor at the Department of Informatics of the latter university. He has been with the Department of Computer Science of the University of Toronto, the Department of Computer Science of the University of Maryland at College Park and the University of Cyprus. He has published over 120 papers in refereed scientific journals and conference proceedings. He is co-author of a book on "Advanced Database Indexing" and "Advanced Signature Indexing for Multimedia and Web Applications" by Kluwer. He is also author of two textbooks on Data Structures and File Structures, which are recommended in the vast majority of the computer science/engineering departments in Greece. He served/serves as PC Co-chair of the 8th National Computer Conference (2001), the 6th ADBIS Conference (2002) the 5th WDAS Workshop (2003), the 8th SSTD Symposium (2003) and the 1st Balkan Conference in Informatics (2003). Also, currently he is Vice-chairman of the Greek Computer Society. His research interests include access methods and query processing for databases, data mining, and performance evaluation of storage subsystems. Further information can be found at http://delab.csd.auth.gr.