6 Access Methods and Query Processing Techniques

Adriano Di Pasquale[3], Luca Forlizzi[3], Christian S. Jensen[1], Yannis Manolopoulos[2], Enrico Nardelli[3], Dieter Pfoser[1], Guido Proietti[3,4], Simonas Šaltenis[1], Yannis Theodoridis[5], Theodoros Tzouramanis[2] and Michael Vassilakopoulos[2]

[1] Aalborg University, Denmark
[2] Aristotle University, Thessaloniki, Greece
[3] Universita Degli Studi di L'Aquila, Italy
[4] National Research Council, Roma, Italy
[5] University of Piraeus, Greece

6.1 Introduction

The performance of a database management system (DBMS) is fundamentally dependent on the access methods and query processing techniques available to the system. Traditionally, relational DBMSs have relied on well-known access methods, such as the ubiquitous B^+-tree, hashing with chaining, and, in some cases, linear hashing [52]. Object-oriented and object-relational systems have also adopted these structures to a great extend.

During the past decade, new applications of database technology—with requirements for non-standard data types and novel update and querying capabilities—have emerged that motivate a re-examination of a host of issues related to access methods and query processing techniques.

As an example, a range of applications, like cadastral, utilities, shortest path finding, etc., involve geographic, or spatial, data, which are not supported well by existing technology, making it not only desirable, but plain necessary to examine access methods and query processing techniques afresh.

Specifically, Oracle's Spatial Data Engine uses Linear Quadtrees [2] at a conceptual level, but uses B^+-trees as the storage mechanism for the quadtrees at the implementation level. As another example, R-trees [34] have been implemented by Oracle. However, R-trees are mapped to B^+-trees [75], in order to not change other system components, such as the transaction manager, the recovery manager, the buffer manager, etc. Thus, R-trees do not support deletions physically, but only logically, because this is the practice of B^{link}-trees [77], which is the brand of B-trees implemented in commercial systems.

When, as we have seen, DBMSs supporting only the traditional access methods fall short in supporting spatial data, it is not surprising that new access methods and query processing techniques are needed if DBMSs are to support the many and diverse emerging applications that call for the management of spatio-temporal data. While access methods and techniques exist that support time and, as discussed above, space, existing proposals are unable to simultaneously support time and space, either efficiently or at all.

This chapter introduces a number of spatio-temporal access methods (STAMs) and query processing techniques related to spatio-temporal applications, such as bitemporal spatial applications, trajectory monitoring, and the processing of evolving raster images, such as thematic layers or satellite images. These structures are divided into two categories, according to the spatial methods they extend: R-tree-based methods versus quadtree-based methods.

Early R-tree-based approaches for indexing spatio-temporal data either treat time as a spatial dimension or, conceptually, accommodate time by maintaining a

time-indexed collection of R-trees. The chapter presents more R-tree-based methods that are customized more comprehensively to accommodate the special properties of the different kinds of spatio-temporal data considered, and that as a result generally demonstrate better performance.

Briefly, the quadtree-based methods described in this chapter enhance previous methods based on Linear Quadtrees by appropriately embedding additional structuring for linking evolving raster images.

The chapter also examines other issues related to the physical database level, namely benchmarking, data generation, distributed indexing techniques, and query optimization. Finally, relevant work by other researchers is covered, and an epilog concludes the chapter.

6.2 R-tree-Based Methods

6.2.1 Preliminary Approaches

The most straightforward way to index spatio-temporal data is to consider time simply as an additional spatial dimension, along with the other spatial dimensions. As a result, a two-dimensional rectangle (x_1, y_1, x_2, y_2) with an associated time interval $[t_1, t_2)$ is viewed as a three-dimensional box $(x_1, y_1, x_2, y_2, t_1, t_2)$. Viewing time as another dimension is attractive because several tools for handling the resulting multi-dimensional data are already available [30]. The approach of treating time as just another dimension may have the drawback of excessive dead space [96].

The technique of overlapping offers an alternative solution. In general, overlapping has been used at a number of occasions, where successive data snapshots are similar. For example, it has been used as a technique to compress similar text files [8], B-trees and B^+-trees [11,50,85], as well as main-memory quadtrees [98,99]. In the context of STAMs, the technique of overlapping has been adopted in the cases presented in the sequel.

Thus, most of the proposed STAMs can be characterized as belonging to one of the following two categories:

- the "time is an extra dimension" approach, and
- the "overlapping trees" approach.

We proceed to present several access methods that follow these approaches and were developed in the CHOROCHRONOS framework; in Section 6.7, we introduce methods proposed by other researchers.

3D R-tree

The 3D R-tree proposed in [96] exactly considers time as an extra dimension and represents two-dimensional rectangles with time intervals as three-dimensional boxes. Figure 6.1 illustrates an example 3D R-tree storing five boxes (A, B, C, D, and E) organized in two nodes (R1 and R2). This tree can be the original R-tree [34] or any of its variants.

The 3D R-tree approach assumes that both ends of the interval $[t_1, t_2)$ of each rectangle are known and fixed. If the end time t_2 is not known, this approach does not work well. For instance, in Figure 6.1, assume that an object extends from some fixed time until the current time, *now* (refer to [16] for a thorough discussion on the notion of *now*). One approach is to represent *now* by a time instant sufficiently far in the future. But this leads to excessive boxes and consequent poor performance. Standard spatial access methods, such as the R-tree and its variants, are not well suited to handle such "open" and expanding objects. One special case where this

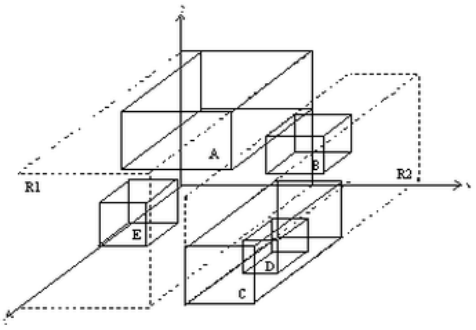


Fig. 6.1. The 3D R-tree.

problem can be overcome is when all movements are known a priori. This would cause only "closed" objects to be entries of the R-tree.

The 3D R-tree was implemented and evaluated analytically and experimentally [96,100], and it was compared with the alternative solution of maintaining two separate indices: a spatial (e.g., a 2D R-tree) and a temporal one (e.g., a 1D R-tree or a segment tree). Synthetic (uniform-like) datasets were used, and the retrieval costs for pure temporal (during, before), pure spatial (overlap, above), and spatio-temporal operators (the four combinations) were measured. The results suggest that the unified scheme of a single 3D R-tree is obviously superior when spatio-temporal queries are posed, whereas for mixed workloads, the decision depends on the selectivity of the operators.

2+3 R-tree

One possible solution to the problem of "open" geometries is to maintain a pair of two R-trees [63]:

- a 2D R-tree that stores two-dimensional entries that represent current (spatial) information about data, and
- a 3D R-tree that stores three-dimensional entries that represent past (spatio-temporal) information; hence the name 2+3 R-tree.

The 2+3 R-tree approach is a variation of an original idea proposed in [41,42] in the context of bitemporal databases, and which was later generalized to accommodate more general bitemporal data [10].

As long as the end time (t_2) of an object interval is unknown, it is indexed by the (2D) *front* R-tree, keeping the start time (t_1) of its position along with its object identifier. When t_2 becomes known, then:

- the associated entry is migrated from the front R-tree to the (3D) *back* R-tree, and
- a new entry storing the updated current location is inserted into the front R-tree.

Should one know all object movements a priori, the front R-tree would not be used at all, and the 2+3 R-tree reduces to the 3D R-tree presented earlier. It is also important to note that both trees may need to be searched, depending on the time instant with respect to which the queries are posed.

HR-tree

Historical R-trees (HR-trees, for short) have been proposed in [61] and implemented

and evaluated in [63]. This STAM is based on the overlapping technique. In the HR-tree, conceptually a new R-tree is created each time an update occurs. Obviously, it is not practical to physically keep an entire R-tree for each update. Because an update is localized, most of the indexed data and thus the index remain unchanged across an update. Consequently, an R-tree and its successor are likely to have many identical nodes. The HR-tree exploits this and represents all R-trees only logically. As such, the HR-tree can be viewed as an acyclic graph, rather than as a collection of independent tree structures.

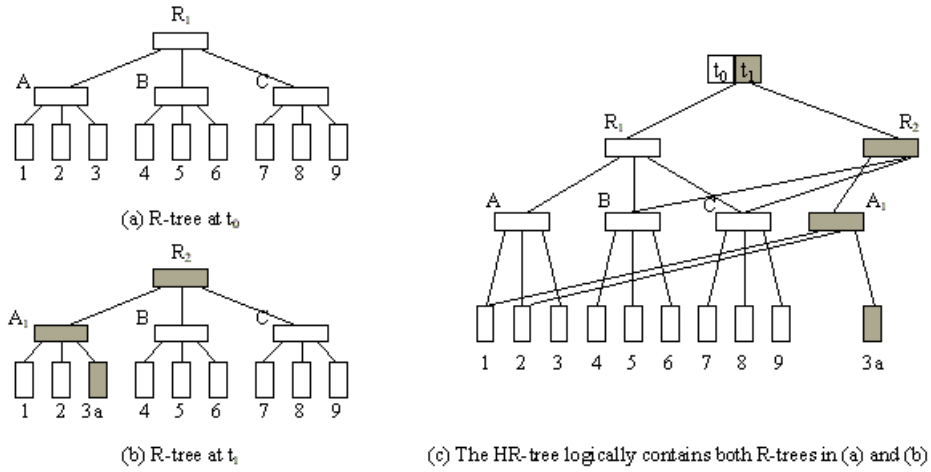


Fig. 6.2. HR-tree Example.

Figure 6.2 illustrates overlapping trees for successive time instants t_0 and t_1, where two subtrees from t_0 remain unchanged at t_1. With the aid of an array pointing to the root of the underlying R-trees, one can easily access the desired R-tree when performing a timeslice query. In fact, once the root node of the desired R-tree for the time instant specified in the query is obtained, the query processing cost is the same as if all R-trees where kept physically.

The concept of overlapping trees is simple to understand and implement. Moreover, when the number of objects that change location in space is relatively small, this approach is space efficient. However, if the number of moving objects from one time instant to another is large, this approach degenerates to independent tree structures, since no common paths are likely to be found.

Recently, Nascimento et al. [63] implemented the HR-tree and the 2+3 R-tree and presented a performance comparison, also including a 3D R-tree implementation, using synthetic datasets generated by GSTD (the scenarios illustrated in Section 6.5). They assumed spatio-temporal data specified as follows.

- the data set consisted of two-dimensional points, which were moving in a discrete manner within the unit square;
- updates were allowed only in the current state of the (hence, chronological) database;
- the timestamp of each point version grew monotonically following a transaction time pattern, and
- the cardinality of the data set remained fixed as time evolved.

The HR-tree was found to be more efficient than the other two methods for timeslice queries, whereas the reverse was true for time interval queries. Also, the HR-tree usually led to a rather large structure.

6.2.2 The Spatio-Bitemporal R-tree

The R^{ST}-tree proposed by Šaltenis and Jensen in [80] is capable of indexing spatio-bitemporal data with discretely changing spatial extents. In contrast to the indexing structures described previously, the R^{ST}-tree supports data that has two temporal dimensions and two spatial dimensions. The *valid time* of data is the time(s)—past, present, or future—when the data is true in the modeled reality, while the *transaction time* of data is the time(s) when the data was or is current in the database [36,76]. Data for which both valid and transaction time is captured is termed *bitemporal*.

As mentioned earlier, most of the previously proposed spatio-temporal indices [96,62] assume only one time dimension and use either the technique of overlapping index structures or add time as another dimension to an existing spatial index.

The former approaches do not generalize well to two time dimensions, and treating time as a spatial dimension has certain limitations. In particular, time intervals associated with data objects can be *now-relative*, meaning that their end points track the progressing current time. Consider the recording of addresses. The time a person resides at a given address may often extend from a known start time (the valid-time interval begin) to some unknown future time, which is captured by letting the valid-time interval extend to the progressing current time. The same applies to transaction time; the time a data object is inserted into the database is known, but it is unknown when the tuple will be deleted. This notion of *now* is peculiar to time and has no counterpart in space.

In order to support these aspects of time together with spatial dimensions, the R^{ST}-tree is based on the R^*-tree [12] and attempts to reuse ideas presented in the GR-tree [9]. This latter index also extends the R^*-tree and is arguably the best index for general bitemporal data, which encompasses now-relative data.

Since the index is based on the R^*-tree, the spatial value of an object may be a point or may have extent. Examples of discretely changing spatio-temporal point data include demographic data that captures the changing locations of peoples' residences. Also, cadastral systems exemplify data with spatial extents. Here, the shapes and locations of land parcels, approximated by rectangles for indexing purposes, are recorded together with the histories of their change. Such histories may contain now-relative time intervals. We proceed to characterize now-relative data.

The Data and Queries Supported

We adopt the standard four-timestamp format for capturing valid and transaction time [83], where each tuple is timestamped with four time attributes: $VT^{\vdash}$ and $VT^{\dashv}$ for valid time; $TT^{\vdash}$ and $TT^{\dashv}$ for transaction time. To represent now-relative time intervals, $VT^{\dashv}$ can be set to *now* and $TT^{\dashv}$ can be set to UC (*until changed*).

Consider the example relation in Table 6.1. Tuple 1 records that the information "John lived at Pos1" was true from 3/97 to 5/97 and that this was recorded during 4/97 and is still current. Tuple 3 records that "Jane lives at Pos3" from 5/97 until the current time, that we recorded this belief on 5/97, and that this remains part of the current state. In the case of Tuple 3, the valid-time end being equal to *now* means that we currently do not believe that Jane will live at Pos3 next month (on 10/97). This assumption can be too pessimistic. For example, there can exist a restriction that a person can only move with a month notice. We would then believe Jane to live at Pos3 next month as well. To record this type of knowledge, Clifford et al. [16] proposed to use $now + \Delta$ in the valid-time end attribute. The offset Δ can be any integer, positive or negative. The latter is useful when information about changes in positions is delayed. Tuples 6 and 7 exemplify the usage of positive and negative offsets.

	Person	Position	$TT^{\vdash}$	$TT^{\dashv}$	$VT^{\vdash}$	$VT^{\dashv}$
(1)	John	Pos1	4/97	UC	3/97	5/97
(2)	Tom	Pos2	3/97	7/97	6/97	8/97
(3)	Jane	Pos3	5/97	UC	5/97	*now*
(4)	Julie	Pos4	3/97	7/97	3/97	*now*
(5)	Julie	Pos4	8/97	UC	3/97	7/97
(6)	Ann	Pos5	5/97	UC	3/97	$now + 1$
(7)	Scott	Pos6	4/97	UC	5/97	$now - 2$

Table 6.1. The Demographic Relation.

In a bitemporal database, tuples are never physically deleted. Instead, they are removed from the current state, by changing the $TT^{\dashv}$ value UC to the fixed value CT–1[1] (e.g., Tuple 2). A modification is modeled as a deletion followed by an insertion (e.g., an update led to Tuples 4 and 5).

The temporal aspects of a tuple can be represented by a two-dimensional *bitemporal region* in the space spanned by transaction time and valid time [36] (see Figure 6.3). A now-relative transaction-time interval yields a rectangle that "grows" in the transaction-time direction as time passes (Tuple 1). Having both transaction-

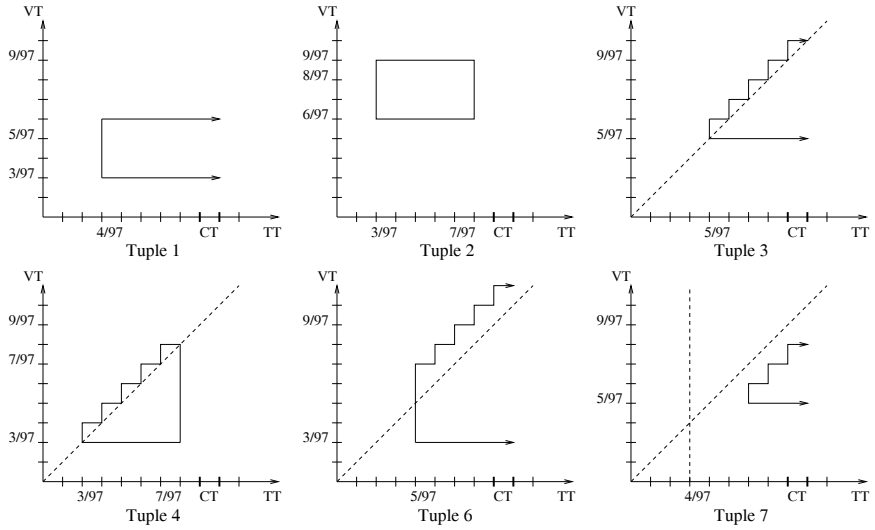


Fig. 6.3. Bitemporal Regions of Tuples from Table 6.1.

time and valid-time intervals being now-relative yields a stair-shaped region growing in both the transaction time and valid time as time passes (Tuple 3).

This representation of the bitemporal extents of tuples suggests the use of some spatial index as the basis for a bitemporal index, which then also facilitates the incorporation of spatial dimensions into the resulting spatio-bitemporal index.

The queries supported are the well-known intersection queries, where data with a spatio-bitemporal extent that overlaps with a specified query extent, which is also spatio-bitemporal.

Index Structure and Algorithms

[1] We use closed intervals and let [$TT^{\vdash}$, $TT^{\dashv}$] denote the interval that includes $TT^{\vdash}$ and $TT^{\dashv}$.

Index Structure
The new index has the same overall structure as the R-tree (and the R*-tree) [34]. As for the R-tree, each internal node is a record of index entries, each of which is a pair of a pointer to a node at the next level in the tree and a region that encloses all regions in the node pointed to. As something new, the leaves of the R^{ST}-tree record the exact bitemporal geometries of the spatio-bitemporal regions indexed and allow regions that grow. The same types of regions are also used as bounding regions in the non-leaf nodes (see Figure 6.4). The following format is used for index entries.

$$(TT^{\vdash}, TT^{\dashv}, VT^{\vdash}, VT^{\dashv}/\Delta, \textit{now-flag}, \langle \textit{spatial part} \rangle, \langle \textit{pointer} \rangle)$$

The first three components were introduced in the previous section and may obtain the same values as described there. Variable UC is represented as a special, reserved value from the domain of timestamps. The fourth and fifth components compactly encode the values of the $VT^{\dashv}$ attribute. A value of the form $now + \Delta$ is captured by setting the *now-flag* and storing Δ in $VT^{\dashv}/\Delta$; other values are stored in this attribute, without the *now-flag* set.

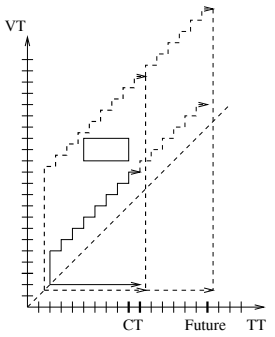


Fig. 6.4. A Sample Stair-Shaped MBR.

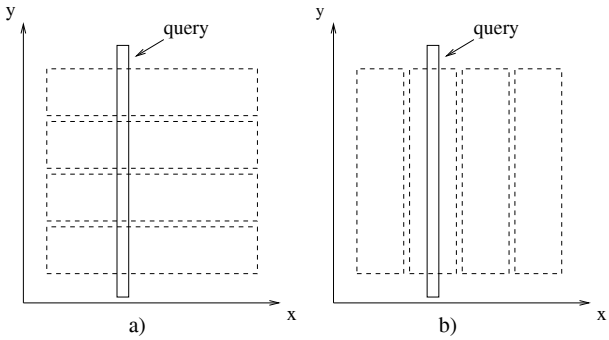


Fig. 6.5. Different Geometries of MBRs.

Index Algorithms
Since the R^{ST}-tree structure is the same as that of the R*-tree, the R*-tree search, deletion, and insertion algorithms can be re-used in the new index, provided that they employ a suite of new lower-level algorithms that manipulate the new kinds of regions described above. These new algorithms include an algorithm that determines whether a pair of regions overlap and algorithms that compute the volume and margin of a region, the intersection of a pair of regions, and the minimum bounding region of a node. It should also be noted that a logical deletion is implemented as a physical deletion of an old region followed by an insertion of a new one with a fixed $TT^{\dashv}$.

The insertion algorithm is crucial, because it is responsible for maintaining the tree in an efficient way. The R*-tree insertion algorithm is based on heuristics that minimize the volumes of bounding regions, the overlap among bounding regions (the volume of their intersection), and the margin of bounding regions.

In the R^{ST}-tree, the quantities of volume, overlap, and margin are functions of time, and the insertion algorithm should consider not only the current values of these, but also how they evolve. This is achieved in a relatively straightforward and flexible manner, by introducing a *time parameter* p in the tree insertion algorithm, which then computes the areas (and margins) of regions as of p time units into the future. Other than this, the insertion algorithm follows that of the R*-tree,

with only some differences in the splitting of overfull nodes [80]. When using a sufficiently large time parameter, a prioritization of the types of regions is obtained. Non-growing regions are naturally preferred over growing, rectangular regions, and these are preferred over growing, stair-shaped regions. Relaxed prioritizations are achieved by using smaller time parameter values. Experimental studies show that the choice of an appropriate time parameter value in an index is not very sensitive to differences in the data and query workloads.

Prioritizing Space versus Time

The heuristics used in the R*-tree are based on the assumption that intersection queries are square on average, i.e., all the dimensions are constrained by intervals of approximately the same length.

Due to the quite different semantics of the temporal and spatial dimensions, this may not always be a good assumption for the R^{ST}-tree. In some applications, most queries can be much more restrictive in the spatial dimensions than on the temporal dimensions. For example, queries in a cadastral system may retrieve the current knowledge of the full history of ownership of some piece of land. In other applications, queries can be most restrictive in the temporal dimensions. Specifically, timeslice queries, which specify time points in the temporal dimensions, have very natural semantics and are often important. Non-square queries may also be due to the use of different units of measurement in the spatial and temporal dimensions.

In order to obtain a versatile spatio-temporal index that supports well the full spectrum queries, it is desirable to introduce a mechanism that allows the R^{ST}-tree to be tuned to support better either spatially or temporally restrictive queries.

In any R-tree-based index, one dimension can be prioritized over the others by intentionally favoring minimum bounding rectangles that are narrow in this dimension and long in the other dimensions. In Figure 6.5, a two-dimensional space is considered. The two sets of minimum bounding rectangles cover the same areas and do not overlap. Scenario (b) favors queries restrictive in the x dimension and not in the y dimension.

[80] proposed a simple way to prioritize the dimensions in an R-tree-based index, which works with the existing tree algorithms. For each n-dimensional rectangle, weighted extents $((\Delta x_1)^{\alpha_1}, (\Delta x_2)^{\alpha_2}, ..., (\Delta x_n)^{\alpha_n})$ are used, instead of simply using the extents $(\Delta x_1, \Delta x_2, ..., \Delta x_n)$. If all α_i are equal to one, none of the dimensions are prioritized. The priority of dimension i is increased by setting α_i to a value greater than 1, and the priority of dimension i lowered by setting α_i to a value smaller than 1. Setting α_i to 0 makes the algorithms disregard the dimension.

Following this scheme the R^{ST}-tree uses a single parameter $\alpha \in [-1, 1]$. The volume of a four-dimensional region r is then computed as follows.

$$volume(r) = \begin{cases} bitemporal\_area(r)^{1+\alpha} \cdot spatial\_area(r) & \text{if } \alpha \leqslant 0 \\ bitemporal\_area(r) \cdot spatial\_area(r)^{1-\alpha} & \text{otherwise,} \end{cases}$$

where *bitemporal_area* is the area of the region time-parameterized bitemporal extent and *spatial_area* is the area of its spatial extent.

According to the criteria for classification of STAMs as proposed by Theodoridis et al. [90], the R^{ST}-tree supports two-dimensional points and regions; it is bitemporal; supports *now*-relative time intervals in both time dimensions; both the cardinality and the positions of the spatial objects may change over time; the index is dynamic; and spatial, temporal, and spatio-temporal containment queries are supported with the ability to adapt the index to spatially or temporally restrictive queries.

6.2.3 The Time-parameterized R-tree

So far, we have mainly considered the indexing of discretely moving spatial objects. In this section, we proceed to explore the indexing of continuously moving objects. The rapid and continued advances in positioning systems, e.g., GPS, wireless communication technologies, and electronics in general promise to render it increasingly feasible to track and record the changing positions of objects capable of continuous movement.

Continuous movement poses new challenges to database technology. In conventional databases, data is assumed to remain constant unless it is explicitly modified. Capturing continuous movement with this assumption would entail either performing very frequent updates or recording outdated, inaccurate data, neither of which are attractive alternatives.

A different tack must be adopted. The continuous movement should be captured directly, so that the mere advance of time does not necessitate explicit updates [101]. In other words, rather than storing simple positions, functions of time that express the objects' positions should be stored. Updates are then necessary only when the parameters of the functions change. We use a linear function for each object, with the parameters being the position and velocity vector of the object at the time the function is reported to the database.

The *Time-parameterized R-tree* (TPR-tree, for short) efficiently indexes the current and anticipated future positions of moving point objects (or "moving points", for short). The technique has been proposed by Šaltenis et al. in [81] and extends the R*-tree [12].

Different views of the indexed space distinguish different possible approaches to the indexing of the future linear trajectories of moving objects. Assuming the objects move in n-dimensional space (n=1,2,3), their future trajectories can be indexed as lines in (n+1)-dimensional space, where one dimension is time [91]. As an alternative, one may map the trajectories to points in a $2n$ dimensional space, which are then indexed [39]. Queries must subsequently also be transformed to counter the data transformation. Yet another alternative is to index data in its native, n-dimensional space, which is possible by parameterizing the index structure using velocity vectors and thus enabling the index to be "viewed" as of any future time. The TPR-tree adopts this latter alternative. This absence of transformations yields a quite intuitive indexing technique.

The Data and Queries Supported

We represent the linear trajectory of a moving point object by two parameters: a vector of the coordinates of a reference position at some specified time t_{ref}, $\bar{x}(t_{ref})$, and a velocity vector $\bar{v}$. Then, object positions at times not before the current time are given by $\bar{x}(t) = \bar{x}(t_{ref}) + \bar{v}(t - t_{ref})$. As will be explained shortly, the same two vectors of values, the reference position and the velocity, are used in the bounding rectangles in the TPR-tree nodes.

The TPR-tree supports queries on the future trajectories of points. A query retrieves all points with trajectories that cross the specified query region in ($\bar{x}$, t)-space. We distinguish between three kinds of queries, based on the regions they specify. Let R, R_1, and R_2 be three n-dimensional rectangles and t, $t^{\vdash} < t^{\dashv}$, three time values that are not less than the current time.

Type 1 timeslice query: $Q=(R, t)$ specifies a hyper-rectangle R located at time point t.

Type 2 window query: $Q=(R, t^{\vdash}, t^{\dashv})$ specifies a (n+1)-dimensional hyper-rectangle that has spatial coordinates specified by R and that spans in time from $t^{\vdash}$ to $t^{\dashv}$.

Type 3 moving query: $Q{=}(R_1, R_2, t^{\vdash}, t^{\dashv})$ specifies the $(n{+}1)$-dimensional trapezoid obtained by connecting R_1 at time $t^{\vdash}$ to R_2 at time $t^{\dashv}$.

The second type of query generalizes the first, and is itself a special case of the third type. To illustrate, consider the one-dimensional data set in Figure 6.6, which may represent temperatures measured at different locations. Here, queries $Q0$ and $Q1$ are timeslice queries, $Q2$ is a window query, and $Q3$ is a moving query.

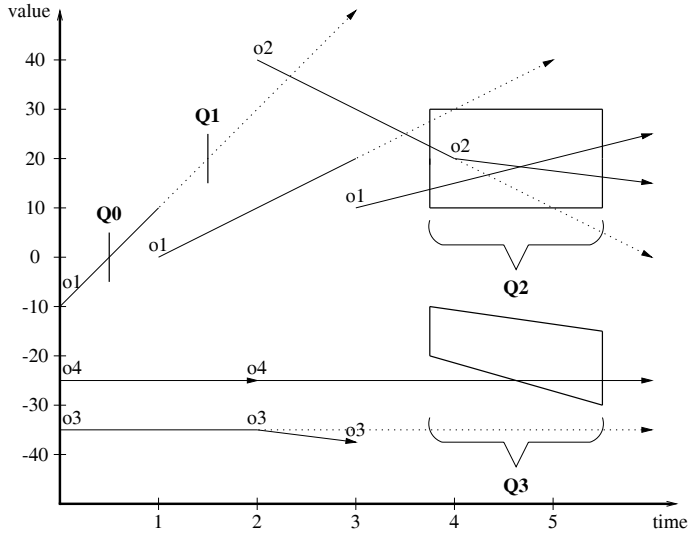


Fig. 6.6. Query Examples for One-dimensional Data.

Let $iss(Q)$ denote the time when a query Q is issued. The two parameters, reference position and velocity vector, of an object as seen by a query Q depend on $iss(Q)$, because objects update their parameters as time goes. Consider object $o1$: its movement is described by one trajectory for queries with $iss(Q) < 1$, another trajectory for queries with $1 \leqslant iss(Q) < 3$, and a third trajectory for queries with $3 \leqslant iss(Q)$. For example, the answer to query $Q1$ is $o1$ if $iss(Q1) < 1$, and no object qualifies for this query if $iss(Q1) \geqslant 1$.

This example illustrates that queries far in the future are likely to yield answers that are of little use, because the positions predicted at query time will be less and less accurate as queries move into the future, and because updates not known at query time may occur in the meantime. Therefore, real-world applications can be expected to issue queries that are concentrated in some limited time window, termed the *querying window* (W), that extends from the current time. We assume that $iss(Q) \leqslant t \leqslant iss(Q){+}W$ for Type 1 queries, and $iss(Q) \leqslant t^{\vdash} \leqslant t^{\dashv} \leqslant iss(Q){+}W$ for queries of Types 2 and 3.

Index Structure and Algorithms

Index Structure

The TPR-tree is a balanced, multi-way tree with the structure of an R-tree. Entries in leaves are pairs of the position of a moving-point object and a pointer to the moving-point object, and entries in internal nodes are pairs of a pointer to a subtree and a rectangle that bounds the positions of all moving objects or other bounding rectangles in that subtree.

In an entry of a leaf, the position of a moving point is represented by a reference position at time t_{ref} and a corresponding velocity vector. We choose t_{ref} to be equal to the index load time, t_l. Other possibilities include setting t_{ref} to some constant value, e.g., 0, or using different t_{ref} values in different nodes.

To bound a group of n-dimensional moving points, n-dimensional bounding hyper-rectangles ("rectangles", for short) are used that are also time-parameterized, i.e., their coordinates are functions of time. A time-parameterized bounding rectangle bounds all enclosed points or rectangles at all times not earlier than the current time.

A tradeoff exists between how tightly a bounding rectangle bounds the enclosed moving points or rectangles across time and the storage needed to capture the bounding rectangles. It would be ideal to employ time-parameterized bounding rectangles that are *always minimum*, but the storage cost appears to be excessive.

Instead of using such always minimum bounding rectangles, the TPR-tree employs "conservative" bounding rectangles, which are minimum at some time point, but possibly (and most likely!) not at later times. Following the representation of moving points, we let t_{ref}=t_l and capture a time-parameterized bounding rectangle as a tuple ($[x_1^{\vdash}, x_1^{\dashv}]$, $[x_2^{\vdash}, x_2^{\dashv}]$, ..., $[x_d^{\vdash}, x_d^{\dashv}]$, $[v_1^{\vdash}, v_1^{\dashv}]$, $[v_2^{\vdash}, v_2^{\dashv}]$, ..., $[v_d^{\vdash}, v_d^{\dashv}]$) that contains a minimum bounding rectangle of all the enclosed points or rectangles at time t_l and the minimums and maximums of the coordinates of velocity vectors of the enclosed objects. The bounding rectangle at time t is then given as follows: $[x_i^{\vdash}(t), x_i^{\dashv}(t)] = [x_i^{\vdash} + v_i^{\vdash}(t - t_l), x_i^{\dashv} + v_i^{\dashv}(t - t_l)]$, where $i = 1, ..., d$.

Figure 6.7 illustrates conservative bounding intervals. The begin of the conservative interval in the figure starts at the position of object A at time 0 and moves left at the speed of object B, and the end of the interval starts at object B at time 0 and moves right at the speed of object A. It is worth noting that conservative bounding intervals never shrink. In the best case, when all of the enclosed points have the same velocity vector, a conservative bounding interval has constant size, although it may move.

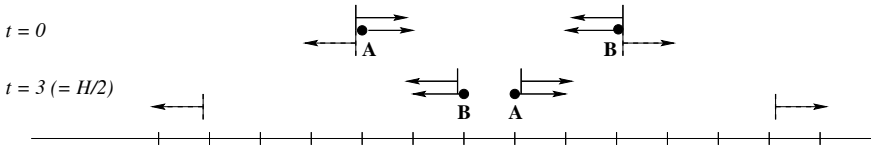


Fig. 6.7. Conservative (Dashed) vs. Always Minimum (Solid) Bounding Intervals.

Such bounding rectangles are termed load-time bounding rectangles because they are minimal at t_l and bounding for all times not before t_l. Because they never shrink, but are likely to grow too much, it is desirable to be able to adjust them occasionally. As the index is only queried for times greater or equal to the current time, it follows that it is attractive to adjust the bounding rectangles every time any of the moving points or rectangles that they bound are updated.

We call the resulting rectangles update-time bounding rectangles. Each update-time bounding rectangle is minimal at the time of the last update that "touched" it, but all bounding rectangles are stored according to the same reference time (t_l). Figure 6.8 illustrates load time and update time bounding intervals.

Algorithms for Querying

Answering a timeslice query using the TPR-tree proceeds as for the regular R-tree, the only difference being that all bounding rectangles are computed for the time t^q specified in the query before intersection is checked. Thus, a bounding

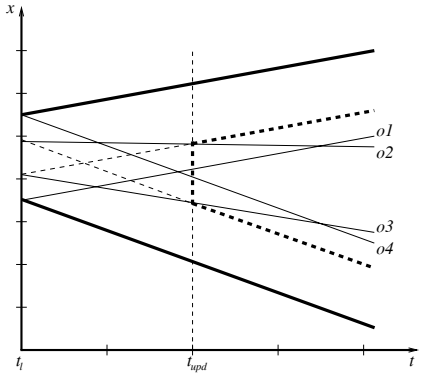


Fig. 6.8. Load-Time (Bold) and Update-Time (Dashed) Bounding Intervals for Four Moving Points.

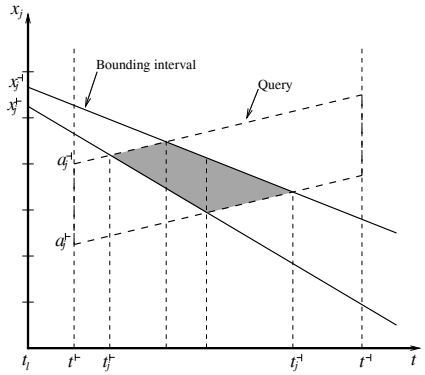


Fig. 6.9. Intersection of a Bounding Interval and a Query.

interval specified by $(x^{\vdash}, x^{\dashv}, v^{\vdash}, v^{\dashv})$ satisfies a query $(([a^{\vdash}, a^{\dashv}]), t^q)$, if and only if $a^{\vdash} \leqslant x^{\dashv} + v^{\dashv}(t^q - t_l) \wedge a^{\dashv} \geqslant x^{\vdash} + v^{\vdash}(t^q - t_l)$.

To answer window and moving queries, we need to be able to check if, in the $(\bar{x}, t)$-space, the trapezoid of a query (see Figure 6.9) intersects with the trapezoid formed by the part of the trajectory of a bounding rectangle that is between the start and end times of the query. With one spatial dimension, this is relatively simple. For more dimensions, generic polyhedron-polyhedron intersection tests can be used [35], but due to the restricted nature of this problem, a simpler and more efficient algorithm was devised for the TPR-tree [81].

Algorithms for Insertion and Bulk Loading

The insertion and bulk loading algorithms of the R*-tree aim to minimize objective functions such as the volumes of the bounding rectangles, their margins (perimeters), and the overlap among the bounding rectangles (the volume of their intersection). In our context, these functions are time-dependent, and we consider their evolution in the time interval $[t_l, t_l + H]$, where H is the average length of the time periods when queries "see" a newly formed bounding rectangle. In particular, given an objective function $A(t)$, the following integral is minimized:

$$\int_{t_l}^{t_l+H} A(t)\ dt$$

If $A(t)$ is volume, the integral computes the volume of the trapezoid that represents part of the trajectory of a bounding rectangle in $(\bar{x}, t)$-space (see Figure 6.9). Parameter H is larger than the querying window W, and $H - W$ is inversely proportional to the average index update rate. The more frequently that object trajectories are updated, the shorter a bounding rectangle lives before it is recomputed.

The TPR-tree insertion algorithm is the same as that of the R*-tree, except that, instead of measures of volume, intersection volume, margin, and distance, integrals of these functions are used as in above formula. In addition, the algorithm for splitting overfull nodes chooses possible distributions of entries into two new nodes is adjusted.

The TPR-tree bulk loading algorithm attempts to minimize the volume integrals of the tree time-parameterized bounding rectangles across $[t_l, t_l + H]$. This is achieved by choosing an appropriate trade off between packing the moving points according to their velocities and packing them according to their reference positions. The former favors relatively large values of H, while the latter is more suitable for

small values of H. The algorithm, described in detail in [81], is based on the STR algorithm [80].

6.2.4 Trajectory Bundle

Similarly to the TPR-tree, the Trajectory Bundle [69] is an R-tree-based access method, but while the TPR-tree indexes the current and anticipated future positions of moving objects, the Trajectory Bundle indexes the past trajectories of point objects capable of continuous movement. The trajectory of an object moving in two-dimensional space is similar to a "string" in three-dimensional space, where the third dimension is time. More specifically, the position of an object is sampled, which leads to a polyline representation of continuous movement (see Figure 6.10).

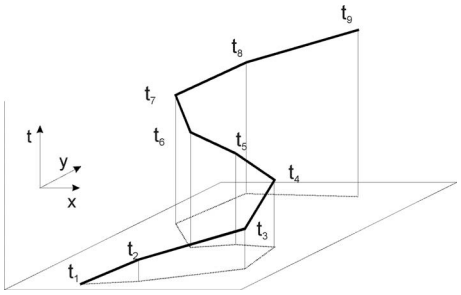


Fig. 6.10. Moving Object Trajectory.

Several approaches to the indexing of historical spatio-temporal data exist. However, most assume that the spatial data changes discretely over time and do not address continuous movement. Although the time dimension of this spatio-temporal space can be perceived as a spatial dimension, its semantics are different. In particular, the presence of a time dimension leads to derived information, e.g., speed, acceleration, traveled distance, etc., which the access method must contend with. Next, a successful access method must recognize that the individual line segments it indexes are parts of larger constructs, namely trajectories.

The Data and Queries Supported

The Trajectory Bundle indexes the past trajectories of point objects, which are assumed to be represented as polylines in the three-dimensional space spanned by valid time and two spatial dimensions—see Figure 6.10.

The aspects of spatio-temporal data mentioned above result in new types of queries that an access method must satisfy. We distinguish between two types.

- *coordinate-based* queries, such as point, range, and nearest-neighbor queries in the resulting three-dimensional space, and
- *semantics-based queries*, usually involving trajectory metadata, such as speed and heading of objects.

Coordinate-based queries are inherited from spatial and temporal databases. The semantics-based queries are classified in *trajectory* queries, which rely on parts of trajectories that go beyond individual segments, and *navigational* queries.

Trajectory queries stem from *spatio-temporal topology* and involve predicates such as "enters," "leaves," "crosses," or "bypasses" [24]. For example, whether an object enters a given area can be determined only after examining more than one

segment of its trajectory. An object entered an area with respect to a given time interval if the start point of its least recent segment is outside the area and the end point of its most recent segment is inside the given area. Similar definitions hold for the other predicates.

Navigational queries relate to information *derived* from the trajectory information and include "speed," "heading," "area covered," etc. Such quantities depend on the time interval considered, e.g., the heading of an object in the last ten minutes may have been strictly East, but considering the last hour, it may have been Northeast.

Further, *combined queries* are important. Such queries extract information related to partial trajectories by identifying the relevant trajectories and then the relevant parts of the trajectories. Trajectories can be selected based on their object identifier. Alternatively, they can be identified via a spatio-temporal range predicate, by a trajectory query, or by a query using derived information. The relevant parts of the identified trajectories are again delimited by a spatio-temporal range, a trajectory query, or derived properties. The example query "What were the trajectories of objects that left Tucson between 7 a.m. and 8 a.m. today during the first hour after they left?" uses the range "between 7 a.m. and 8 a.m." to identify the trajectories, while the temporal range "during the first hour after they left" delimits the relevant parts of the trajectories. Along these lines, a variety of combined queries can be constructed.

Index Structure and Algorithms

Unlike all previous access methods the Trajectory Bundle strictly preserves trajectories. Each leaf contains only segments belonging to one single trajectory. This clustering of line segments based on their trajectory membership comes at a cost. Specifically, the R-tree attempts to place segments that are spatially close in the same leaf. In the Trajectory Bundle, such segments can be in different nodes. This tends to increase overlap among sibling nodes, which affects the query performance for conventional range queries. However, as we shall see, the trajectory preservation is important for answering certain spatio-temporal query types.

Insertion Algorithms

The Trajectory Bundle uses the R-tree structure and algorithms as its basis. It effectively "cuts" trajectories into pieces consisting of (up to) M line segments, where M is the number of segments that fit in a leaf. Figure 6.11 illustrates insertion of a new line segment, which is divided into six steps. First, the leaf that contains its predecessor segment is found. This node is found by traversing the tree from the root, stepping into every child node that overlaps with the minimum bounding box of the new line segment (stage 1 in Figure 6.11). In case the leaf is full, a new leaf node must be introduced. To obtain trajectory preservation, the new leaf is placed at the "end" of the tree, meaning that the new leaf is inserted at the right-most parent node. In Figure 6.11, we step up the tree until we find a non-full parent node (stages 2 through 4). We choose the right-most path (stage 5) to insert the new node. If the parent node has space (stage 6), we insert the new leaf as shown in the figure. In case it is full, we split it by creating a new node at intermediate level 1 that has the new leaf as its only descendent. If necessary, the split is propagated upwards, i.e., a new "right most" branch is created.

It can be argued that this insertion strategy leads a high degree of overlap in the index. This would certainly be the case if we were indexing arbitrary three-dimensional data. However, when indexing spatio-temporal data, we only "neglect" the two spatial dimensions, with respect to space discrimination. The temporal dimension offers some discrimination because data is inserted with increasing time coordinates. The insertion strategy makes use of this property.

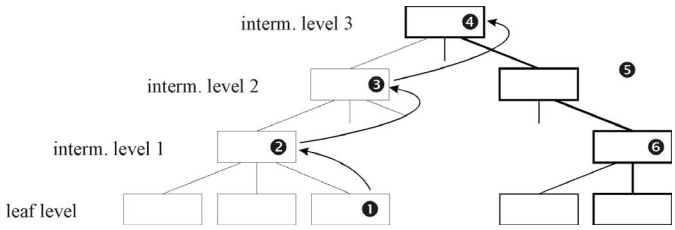


Fig. 6.11. Insertion.

Trajectory Preservation
The leaves in the Trajectory Bundle each contain a partial trajectory, and a trajectory is contained in a set of "disconnected" leaves. We shall see shortly that it is necessary to be able to retrieve segments based on their trajectory identity. To enable this, doubly linked lists are introduced that connect leaves belonging to the same trajectory. Figure 6.12 shows a partial index structure and a trajectory that illustrate this approach. For clarity, the trajectory is drawn as a band. The trajectory symbolized by the gray band is fragmented across six leaves, c1, c3, etc., which form a doubly linked list as symbolized by the two-way arrows. Upon arriving at an arbitrary leaf, these links allow retrieval of partial trajectories at minimal cost.

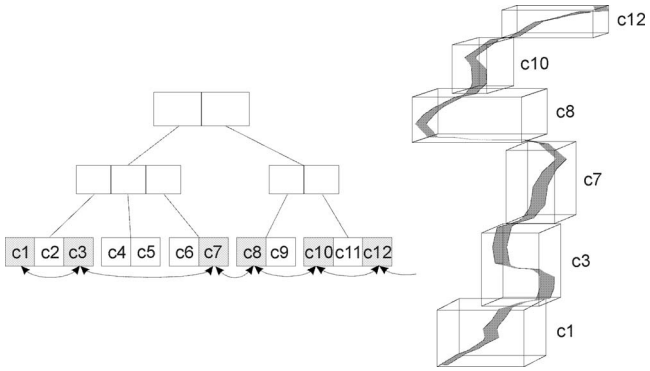


Fig. 6.12. Trajectory Bundle Tree Structure.

Algorithms for Querying
The Trajectory Bundle can be used for processing *coordinate-based*, *semantics-based*, and *combined* queries. The former are well-known and are not discussed further. Semantics-based queries comprise trajectory and navigational queries. We will show how to reduce trajectory queries to ordinary range queries. Navigational queries are special in that they compute results that are not explicitly stored. Since these computations are not based on indexing, we omit discussion of them. Algorithms for combined queries are different in that not only a spatial, but also a semantic search, is performed, i.e., they not only involve range queries, but also retrieve other segments belonging to trajectories identified in range queries.

Algorithms for Trajectory Queries
Trajectory queries involve relationships such as enters, leaves, crosses, and bypasses. These relationships can be computed using the algorithm for range queries.

Consider the query "Which taxis left Tucson between 7 a.m. and 8 a.m. today?" This query specifies a spatio-temporal range, namely "Tucson between 7 a.m. and 8 a.m. today." The cube in Figure 6.13 represents a spatio-temporal range. Trajectory

t_2 belongs to an object leaving the range, and trajectories t_1, t_3, and t_4 are entering, crossing, and bypassing the range. To detect a trajectory leaving a range, we have to examine the segments of the trajectories intersecting the four sides of the spatio-temporal range as shown in Figure 6.13. If a trajectory leaves or enters a range, we will only find one segment, and it will be directed inwards or outward depending on whether it enters or leaves the range. In case a trajectory crosses (t_3) we will find two or more segments. In the case it bypasses (t_4), we will not find any segment. Thus, we can use modified range queries to evaluate the topological predicates of trajectory queries.

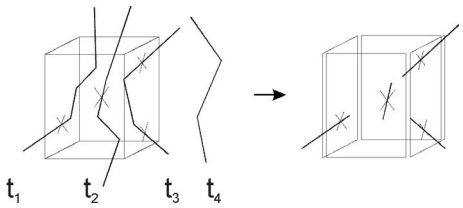


Fig. 6.13. Trajectory Queries.

Algorithms for Combined Queries
The first step in processing combined queries is to retrieve an initial set of segments based on some spatio-temporal range. We can apply the standard range-search algorithm used in the R-tree. In Figure 6.14, we search the tree using the cube c_1 and retrieve two segments of trajectory t_2 (labeled 1 and 2), and four segments of trajectory t_1 (labeled 3 through 6). The six segments are shown in dark gray and are contained in cube c_1. In the second step, we make use of the doubly linked lists in the Trajectory Bundle and retrieve for segments 1 and 2 of t_2 and segments 3 through 6 of t_1 the partial trajectories contained in range c_2. We have two possibilities: a connected segment can be in the same leaf or in another leaf. If it is in the same, finding it is trivial. If it is in another node, the doubly linked list is used.

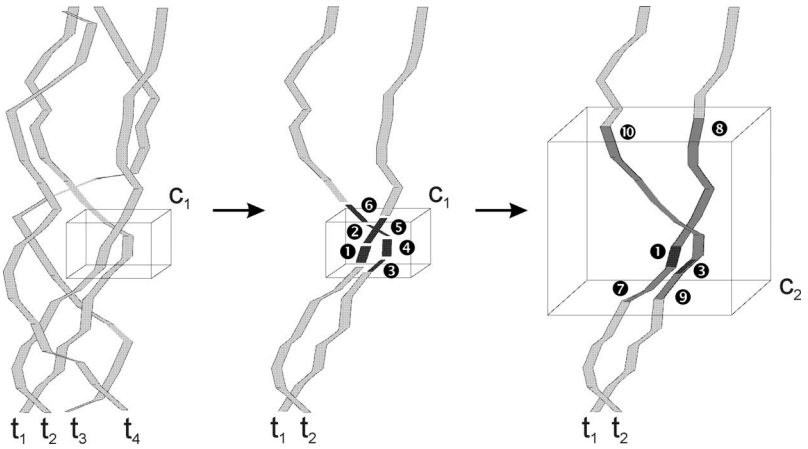


Fig. 6.14. Stages in Processing Combined Spatio-temporal Queries.

When retrieving partial trajectories, care must be taken not to retrieve the same trajectory more than once. Once a partial trajectory is retrieved, we store its object identifier, and, for each retrieved trajectory, we check whether it was retrieved already.

6.3 Quadtree-Based Methods

All methods of this section assume binary images of $S \times S$ unit squares termed *pixels*, where a pixel associated with the foreground (background) is assumed to be black (respectively, white). Without loss of generality, let $S{=}2^m$, where m is an integer used to decompose the image. More specifically, at level m, which is stage 0 of the decomposition, there is the whole image, of side length S. At the first stage of decomposition, the image consists of four quadrants of side length $S/2$. At the second stage, each quadrant is then subdivided into four quadrants of side length $S/2^2$. The decomposition stops when a quadrant is wholly black or wholly white. The decomposition can be represented as a tree of outdegree 4, where the root (at level m) corresponds to the whole image, and each node (at level ℓ, where $1 \leqslant \ell \leqslant m$) corresponds to a quadrant of side length $S/2^{m-\ell}$. The sons of a node are labeled from left to right NW (North-West), NE (North-East), SW (South-West) and SE (South-East). Leaves are black or white (also termed *homogeneous*), and non-leaf nodes are gray (termed *non-homogeneous*).

6.3.1 The MOF-tree

Description of the MOF-tree

The MOF-tree has been proposed in [57,51] for indexing *multiple overlapping features*, but it can also be used as a STAM. The MOF-tree is based on a recursive space decomposition into four quadrants of equal size, in the same way as in quadtrees. Assume that a set $\langle \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_N \rangle$ of images of size $S \times S$ is given. Image $\mathcal{I}_j$ corresponds to the j-th layer of the image $\mathcal{I} = \bigcup_{j=1,N} \mathcal{I}_j$. In the following, we will use the term *feature* to refer to a specific layer of $\mathcal{I}$. The space decomposition stops only when a quadrant is either fully covered by *all* the features in it, or is completely uncovered. In both cases, the quadrant is homogeneous.

The decomposition can be represented as a tree of outdegree 4, as described in the introductory paragraph above. Internal nodes are associated with non-homogeneous quadrants, while homogeneous quadrants correspond to leaves. In Figure 6.15, an example of a MOF-tree in a $2^2 \times 2^2$ image space representing two images is given, in which features partially covering a quadrant are depicted inside a circle, while features totally covering a quadrant are depicted inside a square. Note that internal nodes can be fully covered by some feature and partially covered by others (for example, see the SE son of the root, which is fully covered by the vertical feature and partially covered by the horizontal feature).

Implementing a MOF-tree in Secondary Memory

A linear MOF-tree version can be obtained by coding all its nodes (except the leaves associated with empty quadrants) by means of a base-5 *locational key* (l-key, for short) of length m, that can be recursively defined as follows: let the root of the MOF-tree have l-key 0; a node q' at level ℓ whose father q has l-key $L(q)$ will have l-key [28]:

$$L(q') \;=\; L(q) + s \cdot 5^{\ell}$$

where $s = 1, 2, 3, 4$ if q' is a NW, NE, SW, SE child of q, respectively. This allows the use of a conventional index such as the B^+-tree to efficiently support random access to every MOF-tree node [2].

For a large class of operations to be executed on the MOF-tree, we need to know, when accessing an internal node, which features are contained in and fully cover the associated quadrant. Hence, an internal node is represented by means of a record containing an N-bit vector *FEATURE*, whose j-th bit is set to 1 if and only if the j-th feature is contained in the associated quadrant, and an N-bit vector *COVER*,

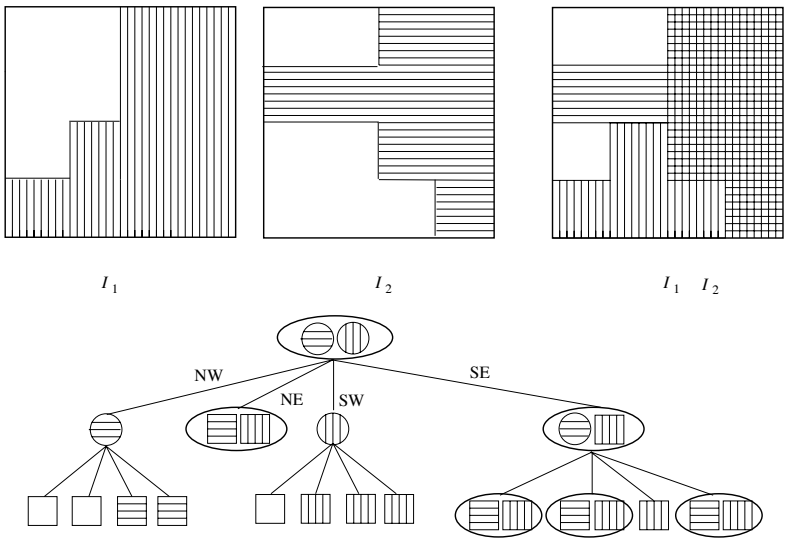


Fig. 6.15. Two overlapping features and the corresponding MOF-tree.

whose j-th bit is set to 1 if and only if the j-th feature fully covers the associated quadrant. Records associated with leaves lack the *COVER* vector. To distinguish between the two kinds of records, we also associate with each of them a *LEAF* bit, whose value is 1 if and only if the corresponding node is a leaf.

The obtained list of records can be sorted according to increasing values of the l-keys. For example, the MOF-tree in Figure 6.15 has the following sequence, where the structure of internal records is (*LEAF*, l-key, *FEATURE, COVER*), while for leaf records (shown in italics for readability), we have a structure (*LEAF*, l-key, *FEATURE*):

(0,00,11,00), (0,10,01,00), *(1,13,01), (1,14,01), (1,20,11)*,
(0,30,10,00), *(1,32,10), (1,33,10), (1,34,10)*, (0,40,11,10),
(1,41,11), (1,42,11), (1,43,10), (1,44,11).

Spatio-Temporal Queries

Traditional spatial queries for two-dimensional spatial data can be naturally extended to spatio-temporal queries over the set $\mathcal{I}$ of two-dimensional images. Let $\mathcal{I}_j(x, y)$ denote the pixel in $\mathcal{I}_j$ having coordinates (x, y), and let assume $\mathcal{I}_j(x, y) = 1$ if the pixel is black, $\mathcal{I}_j(x, y) = 0$ otherwise. Here, we consider only the *exist*$(P(x_0, y_0, t_0), \Delta_x, \Delta_y, \Delta_t)$ query, which must return `true` if and only if there exists at least an image $\mathcal{I}_j \in \mathcal{I}$, with $t_0 \leqslant j < t_0 + \Delta_t$ such that $\mathcal{I}_j(x, y) = 1$ and $x_0 \leqslant x < x_0 + \Delta_x, y_0 \leqslant y < y_0 + \Delta_y$.

Let $w = \{(x, y) \in \mathbb{N}^2 | x_0 \leqslant x < x_0 + \Delta_x, y_0 \leqslant y < y_0 + \Delta_y\}$ denote the so-called *window query*, that is, the image space spanned by the given query. As shown by Proietti in [72], the query is solved by initially decomposing in $O(\Delta_x + \Delta_y)$ time the window into its constituting *maximal blocks*, that is, the black blocks that would be generated by applying the quadtree decomposition process to the window. Dyer has shown that the number of maximal quadtree squares inside w is $O(\Delta_x + \Delta_y)$ [23]. Afterwards, we associate with each maximal block p its respective l-key $L(p)$ and we sort these l-keys in increasing order. For each maximal block p, we have to know whether or not p contains at least one black pixel associated with $\mathcal{I}_j, t_0 \leqslant j < t_0 + \Delta_t$. To do that, we search the B^+-tree with the l-key $L(p)$. One of the following can happen:

1. $L(p)$ appears in the B$^+$-tree. Then, we check whether the *FEATURE* vector has at least one bit in the interval $[t_0, t_0 + \Delta_t - 1]$ equal to 1: if so, we return `true`; otherwise, we examine the next maximal block, if any.
2. $L(p)$ does not appear in the B$^+$-tree. In this case, we check the record associated with the l-key immediately smaller than $L(p)$. Let x be such l-key. Two cases are possible:
   2.1. x is associated with a quadrant p' containing p. This happens if and only if the string obtained from x by discarding all the zeros after the rightmost non-zero digit is a prefix of $L(p)$. In this case we say that p' is an *ancestor* of p and that p is a *descendant* of p'. Then, we check whether the *FEATURE* vector of x has at least one bit in the interval $[t_0, t_0 + \Delta_t - 1]$ equal to 1: if so, we return `true`; otherwise, we examine the next maximal block, if any.
   2.2. Otherwise, x is associated with a block p' disjoint from p. In this case, p is fully uncovered, and we examine the next maximal block, if any.

The following theorem is from [51].

Theorem 1. *Let us consider a sequence of two-dimensional binary images* $\langle \mathcal{I}_1, \mathcal{I}_2, \ldots, \mathcal{I}_N \rangle$ *in a* $S \times S$ *image space, represented by using a MOF-tree stored in a B*$^+$*-tree of order* r*. Then, an exist*$(P(x_0, y_0, t_0), \Delta_x, \Delta_y, \Delta_t)$ *query can be solved with* $O((\Delta_x + \Delta_y) \cdot \log_r n)$ *accesses to secondary memory, where* n *is the number of elements in the MOF-tree.*

Proof. Knuth [40] showed that the height $h(r, n)$ of a B$^+$-tree of order r with n elements is:

$$h(r, n) \leqslant 1 + \log_r \left(\frac{n+1}{2} \right).$$

From the algorithm described above, it follows that for each maximal block $O(\log_r n)$ accesses to secondary memory occur in case (1). Concerning case (2), we note that, apart from the initial descent of the B$^+$-tree, which has a cost of $O(\log_r n)$ accesses, we can at most execute an additional access to secondary memory to check the elements preceding the one accessed. As the number of maximal blocks is $O(\Delta_x + \Delta_y)$, the theorem follows. ■

Note that by using a multiple Linear Quadtrees representation, the query can be performed by applying the algorithm proposed by Nardelli et al. in [58] Δ_t times, thus obtaining an $O\left((\Delta_x + \Delta_y) \cdot \sum_{j=t_0}^{t_0+\Delta_t-1} \log_r n_j\right) = O(\Delta_t \cdot (\Delta_x + \Delta_y) \cdot \log_r n)$ time complexity, where n_j is the number of elements in the MOF-tree associated with $\mathcal{I}_j$. Hence, the improvement obtained by using an MOF-tree is linear in the number of queried features.

6.3.2 The MOF$^+$-tree

Description of the MOF$^+$-tree

An interesting variant of the MOF-tree, named the *MOF$^+$-tree*, was proposed by Proietti in [71]. This variant can be obtained by means of a particular coding technique of the *FEATURE* vector, which eliminates the need for the *COVER* vector.

This coding technique depends on both the feature distribution and the refinement process. To illustrate the coding, we analyze how the pointer version of the MOF$^+$-tree is built. The underlying idea in the building process is that we can describe the distribution of a given feature by simply marking the changing from a non-homogeneous to a homogeneous state. More precisely, let us focus on a specific feature $\mathcal{I}_j$ of $\mathcal{I}$, associated with the j-th bit of the *FEATURE* vector. At the root level, if $\mathcal{I}_j$ covers (either partially or fully) the image space, we set *FEATURE*[j]=1;

otherwise, we set $FEATURE[j]$=0. In this latter case, the j-th bits of the *FEATURE* vectors of all the root descendants down to the leaf level are set to 0 as well. In the former case, two situations are possible:

1. The image space is fully covered by $\mathcal{I}_j$: in this case, the j-th bits of the *FEATURE* vectors of all the root descendants down to the leaf level are set to 1;
2. Otherwise, the j-th bits of the *FEATURE* vectors of all the root children are set to 0, thus introducing an *alternation* in the j-th bit values. The process goes on recursively with alternations in the j-th bit value until a quadrant q homogeneous (i.e., either fully covered or uncovered) with respect to $\mathcal{I}_j$ is reached: in this case, the j-th bit of the *FEATURE* vector of q is copied into the corresponding j-th bits of all its non-leaf children[2], thus introducing a *persistence* in the j-th bit values. Afterwards, the j-th bits of the *FEATURE* vectors of all the descendants of q down to the leaf level are set to 1 if $\mathcal{I}_j$ covers q, and otherwise to 0.

Since the sequence of values of the j-th bits of the *FEATURE* vectors along any path from the root to a leaf is tied to the refinement process, the three possible states (i.e., absence, partial presence, or total presence of a given feature) can be represented using only one bit. More precisely, suppose we want to know the distribution of $\mathcal{I}_j$ with respect to a non-leaf node q. For the sake of generality, suppose that q has a non-leaf child q', having in turn a non-leaf child q''. This is the most general case, since if q does not have any non-leaf nephew, then we can establish the distribution of $\mathcal{I}_j$ in q by simply looking to all its (at most 16) leaf descendants. Depending on the values of the j-th bits of the *FEATURE* vectors of q, q', and q'', the cases reported in Table 6.2 are possible.

q	q'	q''	Distribution of $\mathcal{I}_j$ with respect to q
0	0	0	fully uncovered
0	0	1	fully covered
0	1	0	partially covered
0	1	1	if the parent of q has the j-th bit of the *FEATURE* vector set to 0, then q is fully covered; otherwise, it is partially covered (if q is the root, this configuration is not admissible)
1	0	0	if the parent of q has the j-th bit of the *FEATURE* vector set to 0, then q is partially covered; otherwise, it is fully uncovered (if q is the root, then it is partially covered)
1	0	1	partially covered
1	1	0	fully uncovered
1	1	1	fully covered

Table 6.2. Distribution of $\mathcal{I}_j$ with respect to q.

Implementing a MOF$^+$-tree in Secondary Memory

Differently from the MOF-tree, a linear MOF$^+$-tree version can be obtained by making use of a unique record structure, that is (*LEAF*, l-key, *FEATURE*). This allows up to 30% space savings [71]. For example, for the image in Figure 6.15, we have the following sequence (leaf records are shown in italics):

(0,00,11), (0,10,00), *(1,13,01)*, *(1,14,01)*, *(1,20,11)*, (0,30,00), *(1,32,10)*,

[2] Notice that, by definition, a leaf q has the j-th bit of the *FEATURE* vector set to 1 if and only if $\mathcal{I}_j$ covers q.

(1,33,10), *(1,34,10)*, (0,40,00), *(1,41,11)*, *(1,42,11)*, *(1,43,10)*, *(1,44,11)*.

It is easy to verify that by using a MOF$^+$-tree, any spatio-temporal query can be solved with at most a constant number of additional accesses on secondary storage with respect to a MOF-tree. The exist query can be solved by making use of the algorithm described in the previous section, with case (1) rewritten as follows:

1.1. $L(p)$ appears in the B$^+$-tree and the *LEAF* bit is set to 1. Then, we check whether or not the *FEATURE* vector has at least one bit in the interval $[t_0, t_0 + \Delta_t - 1]$ equal to 1. If so, we return `true`; otherwise, we examine the next maximal block, if any.

1.2. $L(p)$ appears in the B$^+$-tree and the *LEAF* bit is set to 0. Then, to determine whether at least one feature in the interval $[t_0, t_0 + \Delta_t - 1]$ covers p, we check the children and the nephews of p, and we apply the rules from Table 6.2, which entails at most a constant number of additional descents of the index.

Therefore, it can be proved that the $exist(P(x_0, y_0, t_0), \Delta_x, \Delta_y, \Delta_t)$ query can be solved with $O((\Delta_x + \Delta_y) \cdot \log_r n)$ accesses to secondary memory, where n is the number of elements in the MOF$^+$-tree.

6.3.3 Overlapping Linear Quadtrees

Description and Implementation of OLQ

The *Overlapping Linear Quadtrees* (OLQ, for short), proposed by Tzouramanis et al. [92,93], is a structure suitable for storing consecutive binary raster images according to transaction time. This corresponds to a database of evolving images (e.g., satellite meteorological images). This structure saves considerable space without sacrificing query performance in accessing every single image. Moreover, it can be used for answering efficiently window queries for consecutive images (spatio-temporal queries).

If a sequence of N images has to be stored in a Linear Quadtree, each image having a unique timestamp t_i (for i=1, 2, ..., N), then updates will overwrite old versions, and only the most recently inserted images are retained. However, in applications where spatial queries refer to the past, all past versions also need to be accessible. OLQ converts the ephemeral Linear Quadtree to a *persistent* data structure, where past states are also maintained [22].

In this subsection, we present this structure and five temporal window queries: strict containment, border intersect, general border intersect, cover, and fuzzy cover. Experiments with the OLQ based on synthetic pairs of evolving images (random images with specified aggregation) have shown [85] considerable storage savings in comparison to a group of independent Linear Quadtrees. Moreover, the I/O performance of different queries has been studied based on the same synthetic data as well as on real images [93]. It has been shown that using algorithms that take advantage of the special properties of OLQ, in comparison to staightforward algorithms, leads to significantly better I/O performance.

In the sequel, we present how overlapping is applied to Linear Region Quadtrees (the most widely used variations of Region Quadtrees for secondary memory). A Linear Quadtree representation consists of a list of values where there is one value for each black node of the pointer-based quadtree. The value of a node is an address describing the position and size of the corresponding block in the image. These addresses can be stored in an efficient structure for secondary memory (such as a B-tree or any of its variations). The most popular linear implementations are the FL (Fixed Length), the FD (Fixed length – Depth) and the VL (Variable Length) linear implementations [78]. As justified in [92], the FD implementation was the

most appropriate choice for OLQs. In this implementation, the address of a black quadtree node has two fixed size parts: the first part denotes the path (directional code) to this node (starting from the root) and the second part the depth of this node. The right part of Figure 6.16 presents a quadtree which corresponds to the binary image shown on the left of the same figure. In the left part of the figure, also, the directional code of each black node of the depicted tree can be seen.

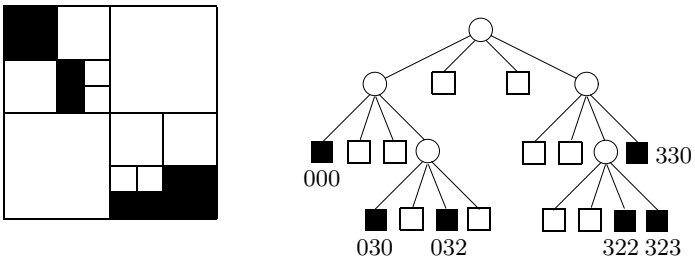


Fig. 6.16. An image, its quadtree and the linear codes of black nodes.

Each quadtree, in a sequence of quadtrees modeling time evolving images, can be represented in secondary memory by storing the linear FD codes of its leaves in a Linear Quadtree (in reality, in a B^+-tree). The OLQ structure is formed by overlapping consecutive Linear Quadtrees, that is by storing the common subtrees of the two trees only once [92]. Since in the same quadtree, a pair of black ancestor and descendant nodes cannot occur, two FD linear codes that coincide at all the directional digits cannot occur, either. This means that the directional part of the FD codes is sufficient for building Linear Quadtrees at all the levels. At the leaf-level, the depth of each black node should also be stored so that images are accurately represented and that overlapping can be correctly applied. The top part of Figure 6.17 depicts the Linear Quadtrees that correspond to two Region Quadtrees, and the bottom part depicts the resulting overlapping linear structure. Note that with the OLQ structure, there is no extra cost for accesses in a specific Linear Quadtree.

All nodes of the OLQ structure have an extra field, *StartTime*, that can be used to detect whether a node is being shared by other trees. We assign a value to *StartTime* during the creation of a node. There is no need for future modification of this field. In addition, leaf-nodes have one more field, *EndTime*, that is used to register the transaction time when a specific leaf changes and becomes historical.

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

Figure 6.18 shows the chaining of the leaves of three successive Linear Quadtrees. The leaf on the left-top corner of the figure corresponds to the first time instant, t=1, and contains 3 quadcodes. Suppose that during time instant t=2, 8 new quadcodes are inserted. In such a case, we have a node split. During time instant t=3, a set of 5 quadcodes is deleted. Thus, two nodes of the tree corresponding to time instant t=2 are merged to produce a new node as depicted in the figure.

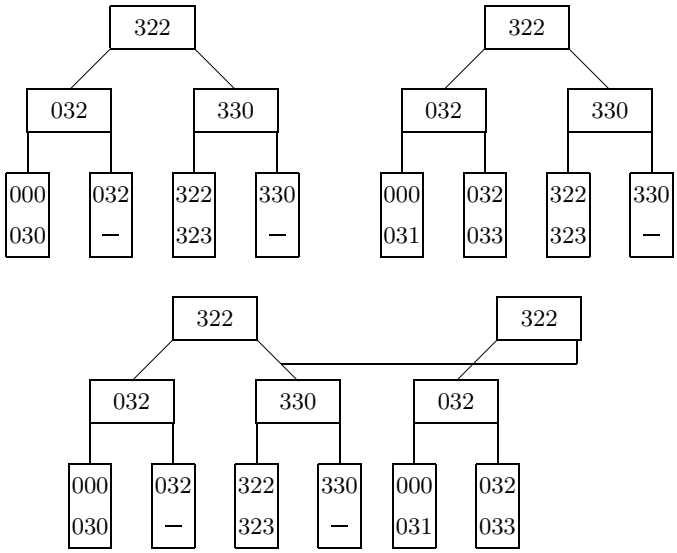


Fig. 6.17. Two B^+-trees storing Linear Quadtree codes and the corresponding OLQ structure.

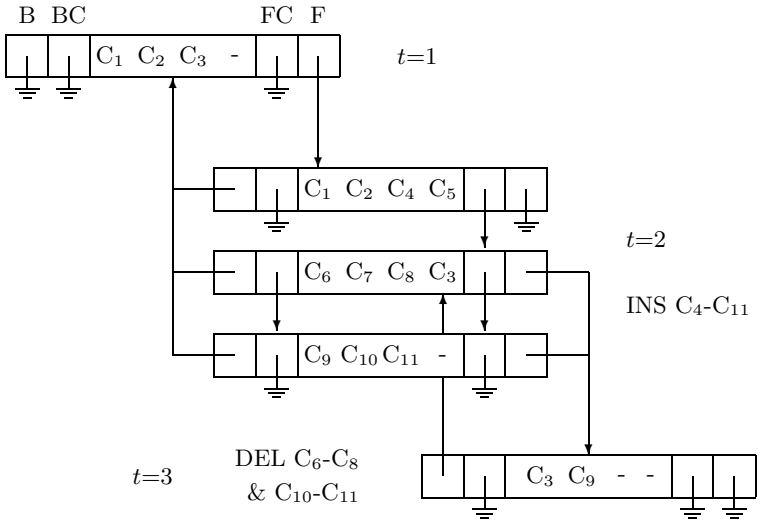


Fig. 6.18. Forward and backward chaining for the support of temporal queries.

Spatio-Temporal Query Processing

In this subsection, we present five different temporal window queries for evolving regional data that can be answered efficiently by using the OLQ structure. Given a window belonging in the area covered by our images and a time interval, the following spatio-temporal queries can be expressed:

The Strict Containment Window Query. Find the black regions that totally fall inside the window (including the ones that touch the window borders from inside), at each time point within the time interval.
Figure 6.19 depicts a raster image corresponding to a specific time point, partitioned in quadblocks, and a query window. The Strict Containment Window Query for this time point would return quadblocks 2 and 4.

The General Border Intersect Window Query. Find the black regions that completely fall inside the window or intersect a border of the window (including the

ones that touch a border of the window from inside or outside), at each time point within the time interval.
The General Border Intersect Window Query for the time point corresponding to Figure 6.19 would return quadblocks 1, 2, 3, 4, and 5.

The Border Intersect Window Query. Find the black regions that intersect a border of the window (including the ones that touch a border of the window from inside or outside), at each time point within the time interval.
The Border Intersect Window Query for the time point corresponding to Figure 6.19 would return quadblocks 1, 3, 4, and 5.

The Cover Window Query. Find out whether or not the window is totally covered by black regions at each time point within the time interval.
The Cover Window Query returns YES/NO answers. For the time point corresponding to Figure 6.19, it would return No as an answer.

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

- Find out whether or not the percentage of the window area that is covered by black regions is larger than a given threshold, at each time point within the time interval.
- Find out the percentage of the window area that is covered by black regions, at each time point within the time interval.

The second kind of Fuzzy Cover Window Query for the time point corresponding to Figure 6.19 would return 80% as its answer. The first kind would return YES or NO depending on the comparison of 80% with the threshold given.

For such queries, [95] presented algorithms that take into account the horizontal pointers. Extensive experiments showed remarkable improvements of the response times of these sophisticated algorithms in comparison to those of the corresponding straightforward algorithms. All the presented algorithms can easily be transformed to work forward or backward: by starting from the beginning or the end of the time interval and by using the F- and FC-pointers or the B- and BC-pointers, respectively.

6.3.4 Multiversion Linear Quadtree

Description and Implementation of MVLQ

As proposed by Tzouramanis et al. [85], the technique for transforming Overlapping B^+-trees to a STAM can also be applied to other classical temporal access methods [84], especially to those that are modifications of the B-tree family. In this section, we present the *Multiversion Linear Quadtree* (MVLQ, for short) [94], which is based on hierarchical decomposition of space and adapts ideas from the Linear Region Quadtrees [28,79] and the Multiversion B-tree (MVBT) [7].

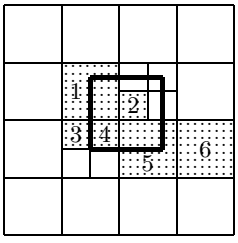


Fig. 6.19. The quadblocks of a binary raster image and a query window (thick lines).

MVLQ associates time intervals with spatial objects in each node. Data records residing in leaves contain records of the form $\langle (C, L), T \rangle$, where (C, L) is the FD code of a black node of the Region Quadtree and T represents the time interval when this black node appears in the image sequence. Non-leaf nodes contain entries of the form $\langle C', T', Ptr \rangle$, where Ptr is a pointer to a descendent node, C' is the smallest C recorded in that descendent node and T' is the time interval that expresses the lifespan of the latter node. For reasons explained in [94], the FD implementation was chosen for the linear representation of the black nodes of a quadtree (the same choice as for the OLQ).

In each MVLQ node, we added a new field, *StartTime*, to hold the time instant when it was created. This field is used by the manipulation algorithms, which will be examined in the sequel. In addition, in each leaf we add a field *EndTime* that registers the transaction time when a specific leaf changes and becomes historical. The structure of the MVLQ is accompanied by two additional main memory substructures:

- The *root* table*: it is built on top of the MVLQ structure. MVLQ hosts a number of version trees and has a number of roots in such a way that each root stands for a time/version interval T''=$[T_i, T_j)$, where $i, j \in \{1, 2, ..., N\}$ and $i < j$. Each record in the root* table represents the root of a MVLQ and has the form $\langle T'', Ptr' \rangle$, where T'' is the lifespan of that root and Ptr' is a pointer to its physical disk address.
- The *Depth First-expression* (in short DF-expression, [37]) of the most recently inserted image: its usage is to keep track of all the black quadblocks of the most recently inserted image, and to be able to know at no I/O cost the black quadrants that are identical between this image and the one that will appear next. Thus, given a new image, we know beforehand which exactly are the FD code insertions, deletions, and updates. The DF-expression is a compacted array that represents an image based on the preorder traversal of its quadtree.

Manipulation Algorithms

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

Insertion

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

- If $StartTime = t_i$, then *a key split* occurs and the leaf splits. Assuming that b is the node capacity, after the key split the first $\lceil b/2 \rceil$ entries of the original node are kept in this node and the rest are moved to a new leaf.
- Otherwise, if $StartTime < t_i$, a copy of the original leaf must first be allocated, since it is not acceptable to change past states of the spatio-temporal structure. In this case, we remove all non-present (past) versions of quadcodes from the copy node. This operation is called *version split* [7], and the number of present versions of quadcodes after the version split must be in the range $[(1+e)d, (k-e)d]$, where k is a constant integer, $d = b/k$ and $e > 0$. If a version split leads to less than $(1+e)d$ quadcodes, then a merge is attempted with a sibling or a copy of that sibling containing only its present versions of quadcodes (the choice depends on the *StartTime* of the sibling). If a version split leads to more than $(k-e)d$ quadcodes in a node, then a key split is performed.

Deletion
Given a "real world" deletion of a quadcode at time point t_j, its implementation depends on the *StartTime* of the corresponding leaf:

- If $StartTime = t_j$, then the appropriate entry of the form $\langle C, L, T\rangle$ is removed from the leaf. After this *physical* deletion, the leaf is checked to see whether it holds enough entries. If the number of entries is above d, the deletion is completed. If the number is below, the *node underflow* is handled as in the classical B^+-tree, with the one difference that if a sibling exists (preferably the right one), then we have to check its *StartTime* before proceeding to a merge or a key redistribution.
- Otherwise, if $StartTime < t_j$ then the quadcode deletion is handled as a *logical* deletion, by updating the temporal information T of the appropriate entry from $T = [t_i, *)$ to $T = [t_i, t_j)$, where t_i is the insertion time of that quadcode. If an entry is logically deleted in a leaf with exactly d present quadcode versions, then a *version underflow* [7] occurs that causes a version split of the node, copying the present versions of its quadcodes into a new node. Evidently, the number of present versions of quadcodes after the version split is below $(1 + e)d$, and a merge is attempted with a sibling or a copy of that sibling.

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

Example
Consider the two consecutive images (with respect to their timestamps t_1=1 and t_2=2) on the left of Figure 6.20. The MVLQ structure after the insertion of the first image is given in Figure 6.21(a). At the MVLQ leaves, the level L of each quadcode should also be stored, but for simplicity only the FD-locational codes appear. The structure consists of three nodes: a root R and two leaves A and B. The node capacity b equals 4 and the parameters k, d, and e equal 2, 2, and 0.5,

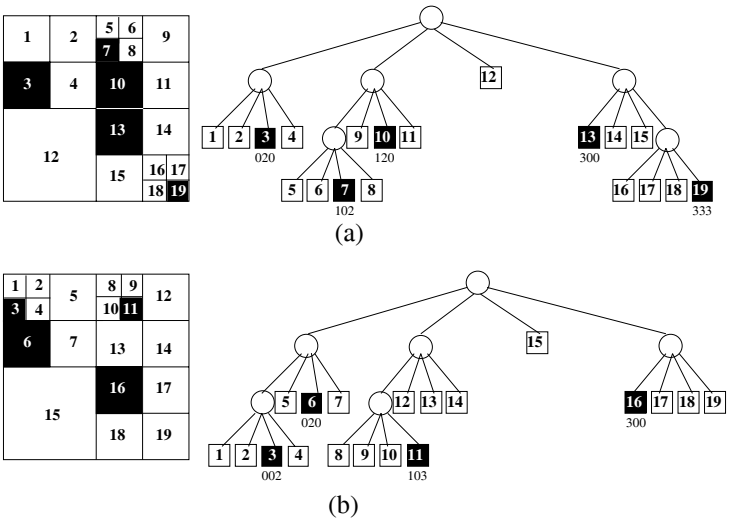


Fig. 6.20. Two similar binary raster images and their corresponding Region Quadtrees.

respectively. The second version of the structure is constructed based on the first one, by inserting the FD code $\langle 002, 0\rangle$ (in the form $\langle C, L\rangle$), the deletion of $\langle 102, 0\rangle$, the insertion of $\langle 103, 0\rangle$, and the deletion of FD codes $\langle 120, 1\rangle$ and $\langle 333, 0\rangle$.

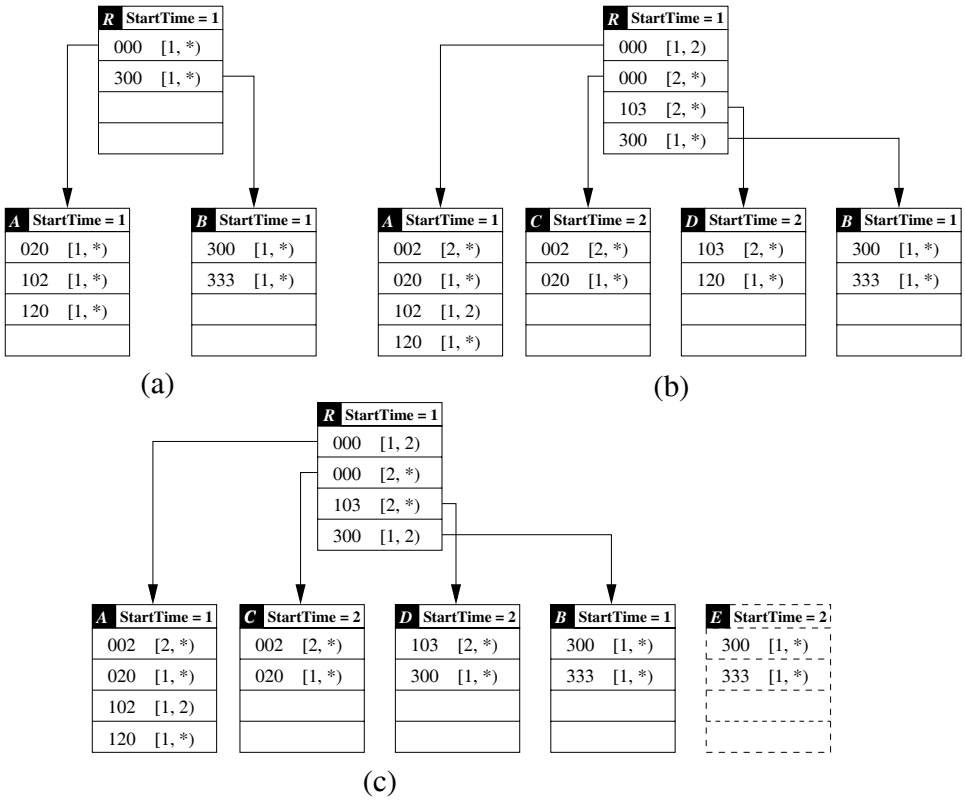


Fig. 6.21. (a) The MVLQ structure after the insertion of the first image (b) a preliminary result during the insertion of the second image, and (c) the final result after the insertion of the second image.

Figure 6.21(b) shows the intermediate result of the insertion of FD code $\langle 002, 0\rangle$, the deletion of $\langle 102, 0\rangle$, and the insertion of FD code $\langle 103, 0\rangle$. When we attempt to insert the quadcode 103 in the leaf A of Figure 6.21(b), the leaf overflows, and a new leaf C is created after a version split. All present versions of quadcodes of leaf A are copied into leaf C, and the parent R is updated for the structural change. Leaf C holds now more than $(k - e)d = 3$ entries, and a key split is performed producing a new leaf D. Again, the parent R is updated.

The final status of MVLQ after the insertion of the second image is illustrated in Figure 6.21(c). The quadcode 120 is deleted from leaf D of Figure 6.21(b) and a node underflow occurs (the number of entries is above d), which is resolved by merging this node with its right sibling B or a copy of it, containing only its present versions of quadcodes. After finding that the *StartTime* of leaf B is smaller than t_2, a version split on that leaf is performed, which is followed by a merge of the new (but temporary) leaf E and leaf D, in leaf D. The process terminates after the physical deletion of quadcode 333 from leaf D. The final number of entries in leaf D equals d. Both versions of MVLQ (Figure 6.21(a) and Figure 6.21(c)) have the same root R, although in general, more than one roots may exist.

Comments on Insertion, Deletion, and Update Algorithms
Generally, insertion of a new image occurs in two stages. The first stage is to sort the quadcodes of the new image and compare this sequence against the set of quadcodes of the last inserted image, using the binary table of its DF-expression. Thus, there is no I/O cost for black quadrants that are identical between the two successive images. During the next stage, we use the root* table to locate the root that corresponds to the most recently inserted image. Then, following ideas of the approach of [45], we build the new tree version by performing all the quadcode insertions, updates, and deletions in a batched manner, instead of performing them one at a time. (We did not follow this approach in the example of Figure 6.21 for simplicity reasons). It is obvious that after a batch operation with insertions, deletions, and updates at a specific time point, we may have conceptual node splittings and mergings. Thus, a specific leaf may split in more than two nodes, and, similarly, more than two sibling leaves may merge during FD code deletions.

Spatio-Temporal Query Processing
The new indexing structure of MVLQ is based on transaction time and it is an extension of MVBT and Linear Quadtree for spatio-temporal data. In order to improve spatio-temporal query processing over the stored raster images, we added four horizontal pointers in every MVLQ leaf. Their names, roles, and functions are the ones that were presented in Section 6.3.3, and they are described in full detail in [95]. The structure supports all the well-known spatial queries for quadtree-based spatial databases (spatial joins, nearest neighbor queries, similarity and spatial selection queries, etc.) without taking into account the notion of time. It can also support efficiently all the typical temporal queries for transaction-time databases (most of which have been examined in [7,14]) without considering issues of space. However, the major feature of the MVLQ is that it can efficiently handle all the special types of spatio-temporal window queries for quadtree-based spatio-temporal databases, described in Section 6.3.3 and analyzed further in [94,95].

6.4 Data Structures and Algorithms for the Discrete Model

The discrete model for spatio-temporal data types [26] presented in Section 4.4 offers a precise basis for the implementation of data structures for a spatio-temporal database management system; it is in fact a high-level specification of such data structures. In this section, we briefly explain how these definitions translate into data structures and present a couple of algorithms on the data structures that implement operations specified in Section 4.4.

6.4.1 Data Structures

Two general issues need to be considered in the translation from data type specifications to physical data structures. First, some requirements arise because the data structures are to be used within a DBMS and must serve to represent attribute data types within some given data model. This means that values are placed under control of the DBMS into memory, which in turn implies that: (i) one should not use pointers, and (ii) representations should consist of a small number of memory blocks that can be moved efficiently between secondary and main memory. One way to fulfill these requirements is to implement each data type by a fixed number of records and arrays; arrays are used to represent the varying size components of a data type value and are allocated to the required size. All pointers are expressed as array indices.

The Secondo extensible DBMS (see Section 7.4), in which we are implementing this model, offers a specific concept for the implementation of attribute data types. Such a type has to be represented by a record (called the *root record*), which may have one or more components that are (references to) the so-called *database arrays*. Database arrays are basically arrays with any desired field size and number of fields; additionally, they are automatically either represented "inline" in a tuple, or outside in a separate list of pages, depending on their size [19]. The root record is always represented within the tuple. In our subsequent design of data structures we will apply this concept. Each data type will be represented by a record and possibly some (database) arrays. In other DBMS environments, one can store the arrays using the facilities offered there for large-object management.

On the other hand, many of the data types presented in Section 4.4 are set-valued. Sets will be represented in arrays. We always define a unique order on the set domains and store elements in the array in that order. This way, we can enforce that two set values are equal if and only if their array representations are equal, which enables efficient comparisons.

Non-Temporal Data Types

For the discrete base types and the time type, the implementation is straightforward: they are represented as a record consisting of the given programming language value[3] plus a Boolean flag indicating whether the value is defined. Type *point* is represented similarly by a record with two reals and a flag.

A *points* value is represented as an array containing records with two `real` fields, representing points. Points are in lexicographic order. The root record contains the number of points and the (database) array.

The data structures for *line* and *region* values are designed similar to corresponding structures reported in [29]. A *line* value is a set of line segments. This is represented as a list of *halfsegments*. The idea of halfsegments is to store every segment twice: once for the left end point and once for the right end point. These are called the *left* and *right halfsegment*, respectively, and the relevant point in the halfsegment is called the *dominating point*. The purpose is to support plane-sweep algorithms, which traverse a set of segments from left to right and have to perform an action (e.g., insertion into a sweep status structure) on encountering the left and another action on meeting the right end point of a segment. A total order is defined on halfsegments, which is the lexicographic order extended to treat halfsegments with the same dominating point (see [29] for a definition).

Hence, we represent the *line* value as an array containing a sequence of records, each of which represents a halfsegment (four reals plus a flag to indicate the dominating point); these are ordered as just mentioned. The root record manages the array plus some auxiliary information such as the number of segments, total length of segments, bounding box, etc.

A *region* value can be viewed as a set of line segments with some additional structure. This set of line segments is represented by an array of *halfsegments* containing the ordered sequence of halfsegment records, as for *line*. In addition, all halfsegments belonging to a cycle and to a face are linked together (via extra fields such as *next-in-cycle* within halfsegment records). Two more arrays, *cycles* and *faces*, represent the structure. The array *cycles* contains records representing cycles by a pointer[4] to the first halfsegment of the cycle and a pointer to the next cycle of the face. The latter is used to link together all cycles belonging to one face. Array *faces* contains for each face a pointer into the *cycles* array to the first cycle of the face. A unique order is defined on cycles and faces, but is not described here.

[3] For *string* we assume an implementation as a fixed length array of characters.

[4] From now on, by "pointer" we mean an integer index of a field of some array.

The root record for *region* manages the three arrays and has additional information such as bounding box, number of faces, number of cycles, total area, perimeter, etc. Algorithms constructing region values generally compute the list of halfsegments and then call a *close* operation offered by the *region* data type, which determines the structure of faces and cycles and represents it by setting pointers. More details on the representation strategy can be found in [29], although some details are different here.

Intervals (s, e, lc, rc) are represented by corresponding records. A value of type *range*(α) is represented as an array of interval records ordered by value (all intervals are disjoint, hence there exists a total order). A value of type *intime*(α) is represented by a corresponding record.

Unit Types

We have to distinguish between units that can be represented in a fixed amount of space, called *fixed size units*, and those that cannot, *variable size units*. Fixed size units are *const*(*int*), *const*(*string*), *const*(*bool*), *ureal*, and *upoint*[5]. Variable size units are *upoints*, *uline*, and *uregion*.

Fixed size units can be represented simply in a record that has two component records to represent the time interval and the unit function, respectively. For example, for *ureal* the second record represents the quadruple (a, b, c, r).

For the representation of variable size units, we introduce *subarrays*. Conceptually, a subarray is just an array. Technically it consists of a reference to a (database) array together with two indices identifying a subrange within that array. The idea is that all units within a *mapping* (i.e., a sliced representation) share the same database arrays. Variable sized units are also all represented by a record whose first component is a time interval record. In the sequel we only describe the second component.

A *upoints* unit function is stored in a subarray containing a sequence of records representing *MPoint* quadruples, in lexicographic order on the quadruples. The *upoints* unit is represented in a record whose second component record contains a subarray reference and a three-dimensional bounding "box" (the number of points can be inferred from the subarray indices).

A *uline* unit function is stored similarly in a subarray containing a sequence of records representing *MSeg* pairs, which in turn are *MPoint* quadruples. Pairs are ordered lexicographically by their two component quadruples on which again lexicographic order applies. Again the *uline* unit is represented in a record whose second component consists of a subarray reference and a bounding cube.

A *uregion* unit function is basically a set of *MSeg* values (moving segments, trapeziums in 3D) with some additional constraints. We store these *MSeg* records in the same way and order in a subarray *msegments* as for *uline*. In addition, each record has two extra fields that allow for linking together all moving segments within a cycle and within a face. Furthermore, *uregion* has two additional subarrays *mcycles* and *mfaces* identifying cycles and faces, as in the *region* representation. The second component record of a *uregion* unit contains the three subarrays and a bounding cube for the unit.

For both *uline* and *uregion* one might add further summary information in the second component record, such as the (a, b, c, r) quadruples for the time-dependent length (for *uline*) or for perimeter and size (for *uregion*).

Sliced Representation

The data structure associated with the *mapping* type constructor organizes a collections of units (slices) as a whole. This data structure is parameterized by the unit

[5] We omit the other *const*(α) types, as they are not so relevant here.

data structures. We observe that all unit data structures are records whose first component represents a time interval, and whose second component may contain one or more subarrays.

The *mapping* data structure is illustrated in Figure 6.22. It is basically a (database) array *units* containing the unit records ordered by their time intervals. If the unit type uses k subarrays, then the *mapping* data structure has k additional database arrays. The database arrays mentioned in the unit subarray references will be the database arrays provided in the mapping data structure. The main array *units* as well as the k additional arrays are referenced from a single root record for the *mapping* data structure. Note that the structure has the general form required for attribute data types.

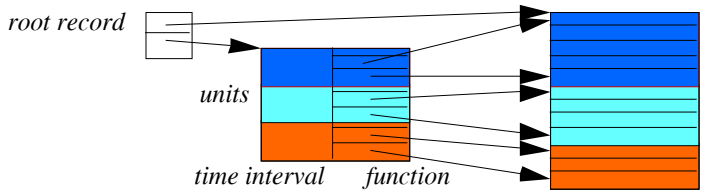


Fig. 6.22. A *mapping* data structure containing three units, for a unit type with one subarray, such as *upoints*.

6.4.2 Two Example Algorithms

We proceed to briefly describe two algorithms thereby illustrating the use of the data model described in Section 4.4 and of the data structures just defined. The first one implements the **atinstant** operation on a moving region, i.e., it determines the region value at a given time instant. The second one implements the **inside** operation on a moving point and a moving region, hence it returns a moving Boolean capturing when the point was inside the region.

Algorithm *atinstant*
The moving region is represented as a value of type *mapping* (*uregion*). The idea of the algorithm is to perform binary search on the array containing the region units to determine the unit u containing the argument time instant t. Then, a subalgorithm is called that evaluates each moving segment within the region unit at time t resulting in a line segment in two dimensions. These are composed to obtain the region value returned as a result.

algorithm *atinstant* (mr, t)
input: a moving region mr as a value of type *mapping*(*uregion*), and an *instant* t
output: a *region* r representing mr at instant t
method:
determine $u \in mr$ such that its time interval contains t;
if u exists **then return** *uregion_atinstant*(u, t) **else return** $\varnothing$ **endif**
end *atinstant*.

algorithm *uregion_atinstant*(u, t)
input: a moving region unit ur (of type *uregion*) and an *instant* t
output: a *region* r, the function value of ur at instant t

method:
  let $ur = (i, F)$; $r := \varnothing$;
  for each mface $(c, H) \in F$ **do**
    $c' := \{\iota(s,t) | s \in c\}$;
    $H' := \varnothing$;
    for each $h \in H$ **do**
      $h' := \{\iota(s,t) | s \in h\}$; $H' := H' \cup \{h'\}$
    endfor;
    $r := r \cup \{(c', H')\}$
  endfor;
  return r
end *uregion_atinstant*.

In the second algorithm the ι function defined in Section 4.4 is used to evaluate a moving segment at an instant of time to get a line segment.

The time complexity of this algorithm is basically $O(\log n + r)$, where n is the number of units in mr, and r is the size of the region returned (the number of segments). This is so because in the first step of *atinstant*, the unit can be found by binary search in $O(\log n)$ time, and because the traversal of the unit data structure takes linear time. However, to construct a proper region data structure as described in Section 6.4.1, one has to produce the list of halfsegments in lexicographic order, and hence needs to sort the r result segments. This results in a time complexity of $O(\log n + r \log r)$. Note that if the region value is just needed for output (e.g., for display on a graphics screen) then $O(\log n + r)$ is indeed sufficient.

The above algorithm works assumes instant t to be internal to the unit time interval. For simplicity, we have ignored the problem of possibly degenerated region values in the end points of the unit time interval. This necessitates a more complex cleanup after finding the line segments, as sketched at the end of Section 4.4. This problem can be avoided altogether if we spend a little more storage space, and represent a unit with a degenerated region at one end instead by two units, one with an open time interval, and the other with a correct region representation for the single instant at the end.

Analogous implementations of the *atinstant* operation can be obtained for all other moving data types. The first algorithm *atinstant* is in fact generic; one only needs to plug in other subalgorithms for the other data types.

Algorithm *inside*
Here the arguments are two lists (arrays) of units, one representing a moving point, the other a moving region. The idea is to traverse the two lists in parallel, computing the refinement partition of the time axis on the way (see Figure 6.23).

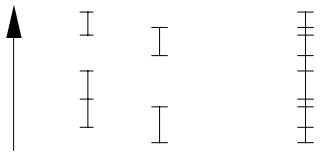


Fig. 6.23. Two sets of time intervals on the left, their refinement partition on the right.

For each time interval i in the refinement partition, an *inside* algorithm is invoked on the point and region units valid during that time interval. A set of Boolean units results, which capture when the point was inside the region. Note that even a linearly

moving point within a single *upoint* unit can enter and leave the region of the region unit several times.

algorithm *inside* (mp, mr)
input: a moving point mp (of type *mapping*(*upoint*)), and a moving region mr (of type *mapping*(*uregion*))
output: a moving Boolean mb, as a value of type *mapping*(*const*(*bool*)), representing when mp was inside mr
method:
let $mp = \{up_1, \ldots, up_n\}$ such that the list $\langle up_1, \ldots, up_n \rangle$ is ordered by time intervals;
let $mr = \{ur_1, \ldots, ur_m\}$ such that the list $\langle ur_1, \ldots, ur_m \rangle$ is ordered by time intervals;
$mb := \varnothing$;
scan the two lists $\langle up_1, \ldots, up_n \rangle$ and $\langle ur_1, \ldots, ur_m \rangle$ in parallel, determining in each step a new refinement time interval i and from each of the two lists either a unit up or ur, respectively, whose time interval contains i, or *undefined*, if there is no unit in the respective list overlapping i:
for each refinement interval i **do**
if both up and ur exist **then**
ub:=*upoint_uregion_inside*(up, ur);
mb:=*concat*(mb, ub)
endif
endfor;
return mb
end *inside*.

The operation *concat* on two sets of units is essentially the union, but merges adjacent intervals with the same unit value into a single unit. On the array or list representations, as given in the mapping data structure, this can be done in constant time (comparing the last unit of mb with the first unit of ub).

algorithm *upoint_uregion_inside*(up, ur)
input: a *upoint* unit up, and a *uregion* unit ur
output: a set of moving Boolean units, as a value of type *mapping*(*const*(*bool*)), representing when the point of up was inside the region of ur during their intersection time interval
method:
let $up = (i', mpo)$ and $ur = (i'', F)$ and let $i = (s, e, lc, rc)$ be the intersection time interval of i' and i''; [6]
if the 3d bounding boxes of mpo and F do not intersect **then return** $\varnothing$
else
determine all intersections between mpo and msegments occurring in (the cycles of faces of) F. Each intersection is represented as a pair $(t, action)$ where t is the time instant of the intersection, and $action \in \{enter, leave\}$;[7]
sort intersections by time, resulting in a list $\langle (t_1, a_1), \ldots, (t_k, a_k) \rangle$ if there are k intersections. Note that actions in the list must be alternating, i.e., $a_i \neq a_{i+1}$;
let t_0=s and t_{k+1}=e;

[6] For simplicity, the remainder of the algorithm assumes the intersection interval is closed. It is straightforward, but lengthy, to treat the other cases.

[7] The *action* can be determined if we store with each msegment (trapezium or triangle in 3D) a face normal vector indicating on which side is the interior of the region.

```
    if k=0 then
      if mpo at instant s is inside F at instant s then
        return {((s, e, true, true), true)}
      else return {((s, e, true, true), false)}
      endif
    else
      if a1=leave then
        return {((ti, ti+1, true, true), true)|i ∈ {0, ..., k}, i is even}
          ∪{((ti, ti+1, false, false), false)|i ∈ {0, ..., k}, i is odd}
      else
        return {((ti, ti+1, true, true), true)|i ∈ {0, ..., k}, i is odd}
          ∪{((ti, ti+1, false, false), false)|i ∈ {0, ..., k}, i is even}
      endif
    endif
  endif
end upoint_uregion_inside.
```

Here, the moving point *mpo* is a line segment in 3D that may stab some of the moving segments of F, which are trapeziums in 3D. In the order of time, with each intersection the moving point alternates between entering and leaving the moving region represented in the region unit. Hence a list of Boolean units is produced that alternates between *true* and *false*. In case no intersections are found ($k = 0$), one needs to check whether at the start time of the time interval considered the point was inside the region. This can be implemented by a well-known technique in computational geometry, the "plumbline" algorithm, which counts how many segments in 2D are above the point in 2D.

The first algorithm *inside* requires time $O(n+m)$, where n, m are the numbers of units in the two arguments, except for the calls to algorithm *upoint_uregion_inside*. This second algorithm requires $O(s)$ time for finding all intersections, with s the number of msegments in F. Furthermore, $O(k \log k)$ time is needed to sort the k intersections, and to return the $k + 1$ Boolean units. If no intersections are found, the check whether *mpo* is inside F at the start time s requires $O(s)$ time. The total time for all calls to *upoint_uregion_inside* is $O(S + K \log k')$, where S is the total number of msegments in all units, K is the total number of intersections between the moving point and faces of the moving region, and k' is the largest number of intersections occurring in a single pair of units. In practical cases, k' is likely to be a small constant, and $K \log k'$ will be dominated by S, hence the total running time will be $O(n + m + S)$. If the moving point and the moving region are sufficiently far apart, so that not even the bounding boxes intersect, then the running time is $O(n + m)$.

This algorithm illustrates nicely how algorithms for binary operations on moving objects can generally be reduced to simpler algorithms on pairs of units. Again, the first algorithm is generic; one only needs to plug in algorithms for specific operations on pairs of units.

6.5 Benchmarking and Data Generation

6.5.1 Benchmarking

As already presented, spatio-temporal data management concerns the design and implementation of access methods that aim at reducing query response time. The performance of an access method depends on the setting it is subjected to, which can be characterized by, e.g., the type of the dataset (points, rectangles, line segments),

the distribution of the dataset, the available buffering strategy, the disk page size, and the queries.

A benchmark is composed of a dataset, an access method, and a set of queries. The output of a benchmark is a set of values describing the performance of the access method for the given dataset and queries. Often, the values describe separately the I/O time and CPU time needed to compute the queries. In order to run benchmarks and thus observe the behavior of access method under varying settings, is is advantageous to have available a flexible benchmarking environment that enables the experimentation with differing (i) datasets, (ii) access methods and (iii) query types.

In order to compare different spatial join strategies, the authors of [32] propose the exploitation of a spatial data generator that is capable of producing datasets of rectangles in two-dimensional space with different characteristics such as rectangle size, data distribution, and dataset size. Datasets are specified by means of so-called models that consist of a set of parameter settings. Models can be reused and modified to generate similar datasets. Often, the objective is to simulate real-life datasets by synthetic ones. The comparison of spatial join techniques is achieved by executing each algorithm on the same datasets and by collecting the results (query response time).

The generator proposed in [32] is limited to spatial datasets. However, a spatio-temporal dataset generator has been proposed in [89]. This generator is capable of producing datasets consisting of moving points that simulate, e.g., the movement of airplanes or ships.

In addition to a data generator, a benchmarking environment needs implementations of access methods and execution of queries. Based on preliminary work [33], a benchmarking environment for spatial query processing is proposed by Gurret et al. in [31]. The system is called BASIS (A Benchmarking Approach for Spatial Index Structures). The application of the system for spatial join processing strategies is studied in [73].

The main parts of the system are depicted in Figure 6.24 and are explained below.

- *Buffer Manager*. It is used to manage the buffers defined for each argument file. The user can control the buffer size.
- *File Manager*. It is used to manage the files stored in the system. A file can be either a dataset file or an access method file. Each BASIS file has a specific internal representation. External files containing datasets must be transformed into this representation before use.
- *Access Methods*. Many access methods can be implemented and integrated into the system. Currently, the system supports R-trees, R*-trees, B-trees, and Grid-files. However, new access methods can be implemented using the API provided.
- *Query Processor*. This component provides the tools needed for executing spatial queries. The query processor is based on iterators. Each complex query is decomposed into a set of more primitive queries, and each primitive query is assigned to an iterator. The complex query is composed by combining the iterators together as a tree. Currently, the system supports range, point, and spatial join queries. However, one can build new iterators in order to compare the access methods.
- *Datasets*: The BASIS system uses either real-life datasets (e.g., from TIGER or Sequoia 2000) or synthetic ones. The dataset generators that have been described previously can be used for this purpose. The only requirement is that these datasets must be transformed to the BASIS internal representation.

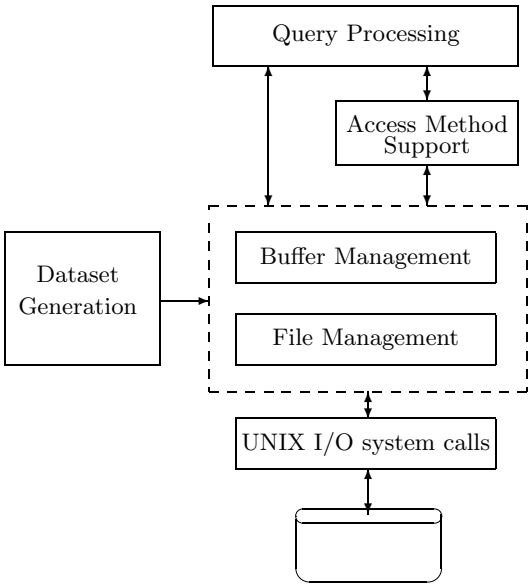


Fig. 6.24. The main components of the BASIS architecture.

Although a lot of work has been performed for spatial benchmarking, spatio-temporal benchmarking must also be investigated thoroughly. STAMs can be implemented using BASIS, thus enabling the execution of spatio-temporal benchmarks.

6.5.2 Data Generation

In order for the user of a benchmarking environment to conduct an extensive series of experiments under a variety of conditions, one should be able to generate a variety of datasets. A fundamental issue in the generation of synthetic spatio-temporal datasets is the availability of a rich set of parameters that control the data generation.

The GSTD Rationale

Theodoridis et al. proposed the GSTD ("Generate Spatio-Temporal Data") algorithm for building sets of moving point or rectangular objects [89]. For each object o, GSTD generates tuples of the format (id, t, pl, pu, f), where id is the object identifier, t is the object timestamp, pl and pu are the lower-left and upper-right corners, respectively, of the object spacestamp (an MBR, assuming a two-dimensional scenario), and f is a flag denoting whether the spacestamp is (spatially) valid or not.

In the GSTD algorithm, three parameters are available.

- *duration* of an object instance; involving change of timestamps between consecutive instances,
- *shift* of an object; involving change of spatial location (in terms of center point shift), and
- *resizing* of an object; involving change of an object size (only applicable to non-point objects).

The GSTD methodology is as follows. Initially, all objects are given starting locations, such that their center points are distributed in the workspace with respect to a chosen distribution, and their extents are either set to zero (in case of point data) or calculated according to the desired density of the data. After the initialization phase, each new instance of an object is generated as a function of the current

instance and the values of the three parameters, which are calculated according to a desired distribution.

In this scenario, it is possible that a coordinate may fall outside the workspace [90]; GSTD manipulates invalid instances according to one among three alternative approaches:

- the *"radar"* approach, where coordinates remain unchanged, although falling beyond the workspace,
- the *"adjustment"* approach, where coordinates are adjusted (according to linear interpolation) to fit the workspace, and
- the *"toroid"* approach, where the workspace is assumed to be toroidal, so that when an object leaves at one edge of the workspace, it enters back at the "opposite" edge.

In the first case, the output instance is appropriately flagged (f=0 in the generated tuple) to denote its invalidity, although the subsequent instance is still calculated with respect to it. In the other two cases, it is the modified instance that is stored in the resulting data file and used for the generation of the subsequent instance. Notice that in the "radar" approach, the number of objects present (i.e., valid) at each time may vary. The three alternative approaches are illustrated in Figure 6.25. For the sake of simplicity, only centers of spacestamps are illustrated; black (gray) locations represent valid (invalid) instances. In the example of Figure 6.25(a), the "radar" fails to detect s_3, hence s_3 is invalid, although the next location s_4 is based on that. Unlike for "radar", the other two approaches always calculate a valid instance s'_3 to be stored in the data file which, in turn, is used by GSTD for the generation of s_4.

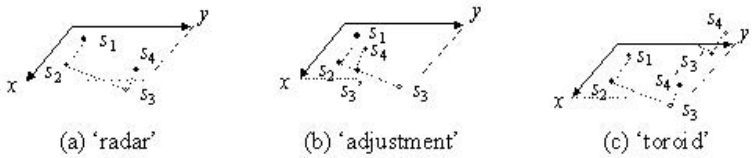


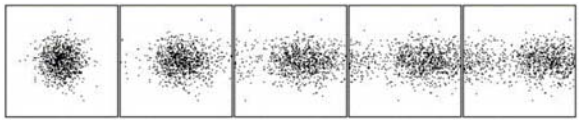
Fig. 6.25. GSTD manipulation of invalid instances.

Through the careful use of the different distributions for the above parameters, GSTD may simulate a variety of interesting scenarios. For instance, using a random distribution for *duration* and *shift*, all objects move equally fast (or slow) and uniformly in the workspace. In contrast, using a skewed distribution for *duration*, a relatively large number of slow objects moving randomly results.

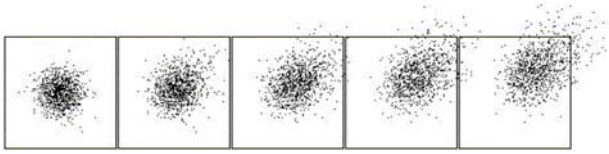
Examples of Generated Datasets

Figure 6.26 presents six different scenarios that illustrate the GSTD capability at simulating desired headings (scenarios 1 through 3) and speeds of objects (scenarios 4 through 6) [90]. Moreover, scenarios 1 and 2 follow the "toroid" and "radar" approach, respectively, while scenarios 3 through 6 follow the "adjustment" approach.

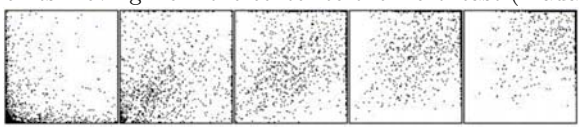
More specifically, scenarios 1 and 2 illustrate points with initial Gaussian spatial distribution moving towards the East and Northeast, respectively. In the former case, where the "toroid" approach was adopted, the points that leave at the right side re-enter on the left side of the workspace. Scenario 3 illustrates an initially skewed distribution of points and their movement towards the Northeast. Since the



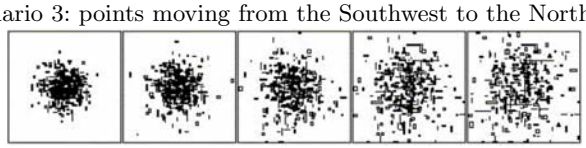
Scenario 1: points moving from the center to the East ("toroid approach")



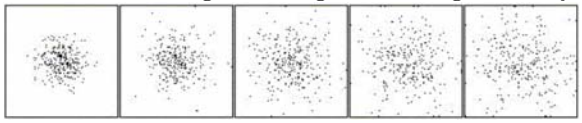
Scenario 2: points moving from the center to the Northeast ("radar approach")



Scenario 3: points moving from the Southwest to the Northeast



Scenario 4: rectangles moving and resizing randomly



Scenario 5: points moving randomly at low speed

Scenario 6: points moving randomly at high speed

Fig. 6.26. Example files generated by GSTD.

"adjustment" approach is used, the points concentrate around the upper-right corner. In Scenario 4, rectangles initially located around the middle of the workspace are moving and being resized randomly. The randomness of *shift* and *resizing* are obtained by applying a uniform distribution to these. Finally, scenarios 5 and 6 exploit the speed of objects as a function of the GSTD input parameters. By increasing (in absolute values) the minimum and maximum values of *shift*, users can generate "faster" objects while the same behavior could be achieved by decreasing *duration*. Similarly, the heading of objects can be controlled, as in scenarios 1 through 3.

6.6 Distribution and Optimization Issues

6.6.1 Distributed Indexing Techniques

A novel architectural choice for obtaining acceptable performance of spatio-temporal DBMSs subjected to huge volumes of data and high frequencies of updates is the so-called *network computing*: many powerful and inexpensive workstations connected through a fast communication network.

Several characteristics make this environment attractive. The most important one is that a set of sites has more computing power and resources than a single site, independently from the equipment of a site. Moreover, the network offers a transfer

speed that is not comparable with those of magnetic or optical disks. Hence, in this framework it is possible and realistic to efficiently implement main memory applications using the main memory of distributed machines. This solution has performances that are not comparable with the traditional centralized ones.

In this paradigm of *Scalable Distributed Data Structures* (SDDSs) data objects are distributed among a variable number of servers and accessed by a set of clients. Both servers and clients are distributed among the nodes of the network. Clients and servers communicate by sending and receiving messages using *point-to-point* or *multicast* protocols[8]. Servers store objects uniquely identified by a key. Every server stores a single block (called *bucket*) of at most b data items, for a fixed number b.

A critical aspect of this solution is to accommodate the dynamic growth of a data file with scalable performance. The key to scalability is to be able to dynamically distribute data across multiple servers of a distributed system. This redistribution of objects should take place continuously as the numbers of objects and requests in the system grow.

An SDDS has to satisfy the following properties:

1. A file expands to new servers only if the used servers are loaded enough. This ensures an efficient use of resources.
2. There is no distinguishable server acting as a centralized controller. This avoids that a server becomes a bottleneck with the increase of the size of the file.
3. No operation involves the execution of an action on more than one client. This is required since clients are autonomous but not continuously available in general.

Efficiency in SDDSs is evaluated with respect to the communication network. This means that performance is measured in terms of the overall number of messages on the network.

Since there is no centrally located address structure that binds keys of all objects into one or more server locations, each client as well as each server is required to have a *local index*. This index is the client's or server's version of the address structure and represents its viewpoint on the latest information about object locations. Consequently, clients can make *addressing errors*, and mechanisms to cope with and to recover them have to be introduced. The goal of a distributed access method is to minimize the number of client and server address errors, as well as the number of local index correction messages between servers and clients.

Litwin et al. were the first to define an SDDS, by proposing a distributed version of linear hashing, namely LH* [47], supporting insert and exact search of one-dimensional objects. Other proposals have been advanced for SDDSs supporting also range-queries on one-dimensional objects, namely RP* [48], DRT [43], RBST [13], BDST [20], and the distributed B^+-tree [15].

To be useful for the management of spatio-temporal data, though, an SDDS has to be able to deal with k-dimensional data. The first SDDS for managing k-dimensional points over a network where multicast is available was proposed by Nardelli in [53] and analyzed in [54,55]. The solution was based on a data structure, named *lazy k-d-tree*, for managing a collection of k-d-trees. The solution featured optimal algorithms for exact, partial, and range search. Distributed k-d-trees are also able to operate in a network where multicast is not available, like in other proposals [46].

In the SDDS model, the split of a server is the typical way to scale up when the number of objects grows. Whenever a server s is in *overflow*, meaning that (due to insertions) it manages a number of objects greater than its capacity b, half of its objects is transferred to a new server s'. The split of a server is a local operation, and

[8] Multicast is a restricted version of broadcast, where only a subset of all machines on the network are collectively addressed.

clients and other servers are not kept, in general, up-to-date with the evolution of the structure. This means that it is possible for client requests to be sent to a wrong server s because the clients local index does not contain the latest object location information. This *address error* is managed by s by forwarding the received request to the server s' that is the *pertinent* server in the viewpoint of s. But s' can be a wrong server as well. The process then continues until the actual pertinent server s^* is found. This server manages the request and sends also local index updates back to the client and to the involved server (see Figure 6.27).

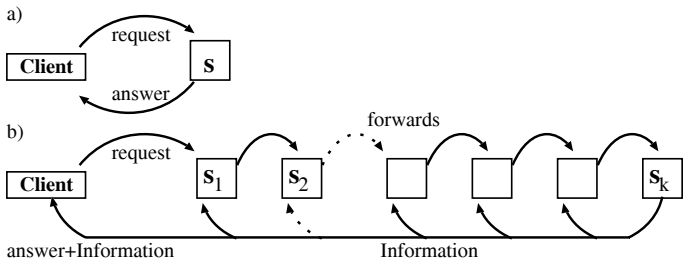


Fig. 6.27. Requests (a) without and (b) with address errors.

Due to the impossibility of carrying over to the distributed environment both a balancing technique and a monotonic search process, the worst case number of messages for search in distributed search trees has a lower bound of $\Omega(\sqrt{n})$ [44]. But in [13] a slight relaxation of these requirements allowed to introduce the first distributed data structure with a poly-logarithmic search time for point and range queries while supporting both the insertion and deletion of elements. This was further improved in [20].

In a more recent proposal [21], amortized analysis of the performance of distributed searching for both one-dimensional and multi-dimensional data has been considered. The result is the definition of an SDDS, namely DRT*, representing a variant of DRT [43] for the one-dimensional case and a variant of the distributed k-d-tree [53] for the multi-dimensional case. In [21] it was proved that both for the one-dimensional and the k-dimensional case, inserts and exact searches have in DRT* an amortized cost of $O\left(\log_{(1+m/n)} n\right)$ messages, where m is the number of requests and n is the total number of servers of the structure. This was obtained by showing that the way local indices change during the evolution of the DRT* structure is similar to the structural changes happening in the set union problem [49], and that request management and splits are strictly related to operations used in the set union problem. Moreover, since in an SDDS m and n are related, inserts and exact searches in DRT* have an amortized almost constant costs, namely $O\left(\log_{(1+A)} n\right)$ messages, while a range query has an amortized cost of $O\left(\log_{(1+A)} n + \lceil k/b \rceil\right)$ messages, where k is the number of items returned by the search, b is the capacity of each server, and $A = b/2$. Considering that in real application environments, A is a large value, of the order of thousands, we can assume to have a constant cost for exact searches in practical cases. Only worst case analysis was previously considered and the result of an almost constant cost for the amortized analysis of the general k-dimensional case appears to be very promising in the light of the well known difficulties in proving optimal worst case bounds for k-dimensions.

Since one of the approaches to obtain efficient indexing techniques for spatio-temporal data is to consider time as a 'spatial' dimension, DRT* offers a very efficient and promising technique for efficient spatio-temporal indexing.

6.6.2 Query Optimization

Query optimization, the task of selecting a suitable strategy for executing an operation, e.g., a query, is an essential task for any DBMS. This is even more true for a spatio-temporal DBMS, due to the huge volumes of data involved. Initial steps towards spatio-temporal query optimization have been taken, but much research remains to be done.

An important part of query optimization is to have available an accurate *statistical model* enabling concise descriptions of datasets with few parameters. This is useful in spatio-temporal DBMSs for analyzing STAM characteristics (e.g., how many nodes there are in a MOF-tree), which is an essential ingredient to estimate cost and selectivity of spatio-temporal queries. This task is performance critical: for instance, query optimizers use query result size estimates to select query execution strategies. In the following, we briefly review some of the results achieved in recent years with respect to query optimization for spatial data; many of these results need to be extended to spatio-temporal data.

Concerning selectivity estimation queries, an analytical formula to compute the *selectivity of a window query* as a function of the underlying data morphology and distribution has been given in [38,74]. When formula parameters are unknown, one typically makes *uniformity* and *independence* assumptions. Unfortunately, these assumptions do not hold for real datasets and generally lead to pessimistic results [17]. For one-dimensional data, some non-uniform distributions have been applied with success, but difficulties remain for multi-dimensional data. In fact, some of the proposed non-uniform models (e.g., the Gaussian distribution [60] or clustering ad-hoc methods [64,12]), show their limitations for data having a different nature from the data they were designed for.

The recent introduction of the concept of fractal dimension has allowed to better describe the statistical properties of data, thus enhancing selectivity estimation in several contexts. For point-data, using the fractal dimension, it is possible to accurately estimate the selectivity of (self) spatial joins [5] and nearest-neighbor queries [70].

Next, novel results for *region data* have been proposed in [65], where a realistic statistical model was proposed: more precisely, they showed that the *complementary cumulative distribution function*[9] (CCDF) of the region areas follow a power law, and this observation is used to compute the selectivity of window queries. When the enclosed spatial objects are *lines* (e.g., roads, rivers, and utilities), it has been observed that the CCDF of the lengths, obeys to an exponential law [66], and once again this result is useful in predicting query performances [59].

Finally, a recent paper by Acharya et al. [3] presents a novel technique based on the notion of *spatial skew* of rectangular data. Using this technique, the authors partition the input rectangles into subsets and approximate each partition, thus obtaining an accurate selectivity estimation over a broad range of spatial queries.

Concerning the analysis of SAMs, we briefly review results that relate to R-trees [34]. An early result was related to the optimal packing for R-trees construction [38], based on data distribution. More recently, the fractal dimension of a set of point has been used to estimate the performance of R-trees for range queries [27]. In [88], a model for the prediction of I/O cost of spatial queries is given, using the concept of *density* of data. In [67], the node distribution of an R-tree storing region data has been studied: the authors showed that the area distribution of the regions is recursively propagated up to the root. Based on this observation, the authors were

[9] Remember that the cumulative distribution function of $f(x) : \Re \rightarrow \Re$ is defined as $F(x) = \int_{-\infty}^{x} f(t)dt$, while the complementary cumulative distribution function is defined as $\overline{F}(x) = \int_{x}^{+\infty} f(t)dt$.

able to accurately estimate the search effort for range queries and to predict the selectivity of a self spatial join posed on the dataset [68].

Finally, we mention another application of fractal theory to spatial data: the estimation of the number of quadtree blocks needed to store a spatial dataset consisting of a single region, once that the fractal dimension of the periphery of the region is known [25].

6.7 Related Work

We divide the work related to access methods and query processing in spatio-temporal databases into two categories: the ones related to general STAMs and those related to indexing moving points.

One of the first results in STAMs is reported in [102]. Specifically, the authors propose MR-trees. These structures are very similar to the HR-trees of [61], which were presented earlier in the chapter. The authors also an R-tree-based structure, termed RT-trees, which is quite different from the presented structures up to now.

RT-trees index objects in two-dimensional space and view time as complementary information that is incorporated as time intervals inside a two-dimensional R-tree structure. More specifically, each RT-tree node contains entries of the form (S, T, P), where S is the spatial information (i.e., the covering MBR), T is the temporal information (i.e., the covering interval), and P is a pointer to either a subtree or the detailed description of an object. Let $T = [t_i, t_j)$, where $i \leqslant j, lett_j$ be the current timestamp, and let t_{j+1} be the successor of t_j. If an object does not change its spatial location from t_j to t_{j+1}, then the spatial information S remains the same, and the temporal information T is updated to T', by increasing the interval upper bound, i.e., T'=$[t_i, t_{j+1})$. When an object changes its spatial location, a new entry with temporal information T=$[t_{j+1}, t_{j+1})$ is created and inserted.

The insertion strategy makes RT-trees efficient data that is mostly static. If the number of updates is large, many entries are created and the RT-tree grows considerably. It should also be observed that the RT-tree node construction depends primarily on the spatial information S, while T only plays a secondary role. Hence the RT-tree is not able to support efficiently temporal queries (e.g., "find all objects that exist in the database within a given time interval").

Several structures similar to MOF- and MOF$^+$-trees have also appeared in the literature. Cheiney at al. [18] have proposed a Region Quadtree-based structure, called *Fully Inverted Quadtrees* (FI-quadtrees, for short) to index an image database. FI-quadtrees are suitable for answering queries on image content (exact and fuzzy search). Vassilakopoulos et al. [97] have proposed *Dynamic Inverted Quadtrees* (DI-quadtrees, for short) which improved FI-quadtrees, since it requires far less disk space, image pattern is performed more efficiently, an it is dynamic since there is no demand for obligatory reorganization. Along the same line, lately in [87] Tourir has proposed *Multi-layer Quadtress*, a new access method based on PM1-quadtrees [79] to represent thematic layers with line segments. All these structures are designed for use in spatial applications, however, they could be used for spatio-temporal applications equally well. In particular, the later one could be used to index trajectories.

The recent work by Zimbrao et al. [104,103] is similar to RT-trees, as they propose another R-tree variant, the structure of *Temporal R-tree* (TR-tree, for short), which uses features from the MVBT structure (e.g. version split and block copy mechanism) [7]. Although TR-trees, like the MVLQ structure, are based on the MVBT structure, they differ since they are designed for vector data rather than raster data. Manipulation algorithms are given in [104] and performance evaluation results against 2+3 R-trees, HR-trees, and RT-trees can be found in [103]. In the latter paper, it is reported that for mixed sets of queries (i.e., time instant and time interval queries), TR-trees outperform their opponents.

In [86], a preliminary effort is descibed that aims to establish a STAM based on k-d-trees. The proposed structure is called *Multi-dimensional Persistent tree* (MP-tree) and implements the idea of partial persistence as do all the previously examined structures. This work is limited in that the structure is not an external balanced multi-way tree, but rather an main-memory resident unbalanced ternary tree. However, their approach can be accommodated in other structures beyond main-memory k-d-trees.

In [82] the *Adaptive tree structure* (AT structure) is put forward. This hybrid method consists of a pair of a spatial (i.e., k-d-trees) and a temporal (i.e. ternary trees) structure. Then, according to the demand, the method selects the data structure that is expected to perform most efficiently. In [56], the AT structure is used to index moving points. These works assume main-memory environments.

Related work on indexing the current and future positions has concentrated mostly on points moving in one-dimensional space. The authors in [91] use PMR-quadtrees [79] for indexing the future linear trajectories of one-dimensional moving point objects as line segments in (x, t)-space. The segments span the time interval that starts at the current time and extends *horizon* time units into the future. A tree expires after U time units, and a new tree must be made available for querying. This approach introduces data replication in the index; a line segment is usually stored in several nodes.

Kollios et al. [39] employ the dual data transformation where a line $x = x(t_{ref}) + v(t - t_{ref})$ is transformed to the point $(x(t_{ref}), v)$, enabling the use of regular spatial indices. It is argued that indices based on k-d-trees are well suited for this problem because these best accommodate the shapes of the (transformed) queries on the data. Kollios et al. suggest, but do not investigate in detail, how this approach can be extended to two and higher dimensions. They also propose two other methods that achieve better query performance at the cost of data replication. These methods do not seem to apply to more than one dimension.

The authors in [6] propose to use the notion of *kinetic* main-memory data structures for mobile objects. The idea is to schedule future events that update a data structure so that necessary invariants hold. Agarwal et al. [1] apply these ideas to external range trees [4]. Their approach may possibly be applicable to R-trees or time-parameterized R-trees where events would fix MBRs, although it is unclear how to contend with future queries that arrive in non-chronological order. Agarwal et al. address non-chronological queries using partial persistence techniques and also show how to combine kinetic range trees with partition trees to achieve a trade-off between the number of kinetic events and query performance.

Within the direction of indexing the past positions of moving objects, most approaches deal with spatial data changing discretely over time and do not take continuous changes into account.

6.8 Conclusions

The present chapter has described a significant number of STAMs and related query processing techniques. The chapter also examines other issues related to the physical database level, namely benchmarking, data generation, distributed indexing techniques, and query optimization. These contributions have appeared in the literature mostly during the last five years. The proliferation and diversity of access methods seen here stem from the very different requirements of the many practical applications involving spatio-temporal data.

Perhaps the nature of the data supported is the most fundamental characteristic that may be used to categorize the proposed access methods. For instance, spatial data can be of vector or raster type. To support these different kinds of data, a

structure based on R-trees or on quadtrees, respectively, should be selected as most appropriate.

It appears that the access methods now available are able to contend with most static spatio-temporal data, which is spatial data that remains unchanged for a time interval. Much more work is necessary to support applications that involve the past and present positions of continuously moving objects.

It is evident that formal mathematical analysis of the many new access methods is not an easy task. More work, e.g., on parametric complexity analysis, is needed in the area of analytical studies. Because of the shortcoming of mathematical analyses, performance comparisons based on empirical experiments with real or synthetic data also play a significant role in the design and evaluation of access methods and query processing techniques. Better infrastructure for empirical studies and more comprehensive comparisons of ranges of access methods are desirable.

For all the access methods proposed for spatio-temporal data, several manipulation algorithms have been reported. Mostly, these algorithms concern standard operations, i.e., insertion, deletion, bulk loading, and fairly simple types of queries, such as window queries based on time intervals, space intervals, or both. There has only been presented little work on more sophisticated algorithms, e.g., for spatio-temporal nearest neighbor queries and spatio-temporal join. Finally, query optimization largely remains a *terra incognita* in relation to spatio-temporal data. Very few papers have appeared that address this topic, which should be investigated in the future, e.g., with the objective of obtaining cost models, heuristics, and algebraic transformation rules.

References

1. L. Arge, P. Argawal, and J. Erickson. Indexing moving points. In *Proceedings 19th ACM PODS Symposium (PODS'00)*, pages 175–186, 2000.
2. D.J. Abel. A B^+-tree structure for large quadtrees. *Computer Vision, Graphics and Image Processing*, 27(1):19–31, 1984.
3. S. Acharya, V. Poosala, and S. Ramaswamy. Selectivity estimation in spatial databases. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 13–24, 1999.
4. L. Arge, V. Samoladas, and J.S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings 19th ACM PODS Symposium (PODS'00)*, pages 346–357, 1999.
5. A. Belussi and C. Faloutsos. Estimating the selectivity of spatial queries using the correlation fractal dimension. In *Proceedings 21st Conference on Very Large Data Bases (VLDB'95)*, pages 299–310, 1995.
6. J. Basch, L. Guibas, and J. Hershberger. Data structures for mobile data. In *Proceedings 8th ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 747–756, 1997.
7. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, 1996.
8. F.W. Burton, M.W Huntbach, and J. Kollias. Multiple generation text files using overlapping tree structures. *The Computer Journal*, 28(4):414–416, 1985.
9. R. Bliujūtė, C.S. Jensen, S. Šaltenis, and G. Slivinskas. R-tree based indexing of now-relative bitemporal data. In *Proceedings 24th Conference on Very Large Data Bases (VLDB'98)*, pages 345–356, 1998.
10. R. Bliujūtė, C.S. Jensen, S. Šaltenis, and G. Slivinskas. Light-weight indexing of bitemporal data. In *Proceedings 9th Conference on Statistical and Scientific Database Management Systems (SSDBM'00)*, pages 125–138, 2000.
11. F.W Burton, J.G Kollias, and D.G. Matsakis. Implementation of overlapping B-trees for time and space efficient representation of collection of similar files. *The Computer Journal*, 33(3):279–280, 1990.
12. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust method for points and rectangles. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 322–331, 1990.

13. F. Barillari, E. Nardelli, and M. Pepe. Fully dynamic search trees can be balanced in $O(\log^2 n)$ time. Technical Report 146, Università di L'Aquila, 1997. Accepted in *Journal of Parallel and Distributed Computing*.
14. J. Bercken and B. Seeger. Query processing techniques for multiversion access methods. In *Proceedings 22nd Conference on Very Large Data Bases (VLDB'96)*, pages 168–179, 1996.
15. Y. Breitbart and R. Vingralek. Addressing and balancing issues in distributed B^+-trees. In *Proceedings 1st Workshop on Distributed Data and Structures (WDAS'98)*, 1998.
16. J. Clifford, C.E. Dyreson, T. Isakowitz, C.S. Jensen, and R.T. Snodgrass. On the semantics of "now". *ACM Transactions on Database Systems*, 22(2):171–214, 1997.
17. S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems*, 9(2):163–186, 1984.
18. J.P. Cheiney and A. Tourir. Fi-quadtree - a new data structure for content-oriented retrieval and fuzzy search. In *Proceedings 2nd Symposium on Spatial Databases (SSD'91)*, pages 23–32, 1991.
19. S. Dieker and R.H. Güting. Efficient handling of tuples with embedded large objects. Technical Report Informatik-236, FernUniversität Hagen, 1998. Also in *Data and Knowledge Engineering*, 32:247-268, 2000.
20. A. Di Pasquale and E. Nardelli. Balanced and distributed search trees. In *Proceedings 2nd Workshop on Distributed Data and Structures (WDAS'99)*, pages 73–90, 1999.
21. A. Di Pasquale and E. Nardelli. Distributed searching of k-dimensional data with almost constant cost. In *Proceedings 4th East European Conference on Advances in Databases and Information Systems (ADBIS'00)*, volume 1884 Lecture Notes in Computer Science, pages 239–250, 2000.
22. J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.
23. C.R. Dyer. The space efficiency of quadtrees. *Computer Graphics and Image Processing*, 19(4):335–348, 1982.
24. M. Erwig, R.H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-temporal data types: an approach to modelling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.
25. C. Faloutsos and V. Gaede. Analysis of n-dimensional quadtrees using the Hausdorff fractal dimension. In *Proceedings 22nd Conference on Very Large Data Bases (VLDB'96)*, pages 40–50, 1996.
26. L. Forlizzi, R.H. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 319–330, 2000.
27. C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of R-trees using the concept of fractal dimension. In *Proceedings 13th ACM PODS Symposium (PODS'94)*, pages 4–13, 1994.
28. I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.
29. R.H. Güting, T. de Ridder, and M. Schneider. Implementation of the ROSE algebra: Efficient algorithms for realm-based spatial data types. In *Proceedings 4th Symposium on Spatial Databases (SSD'95)*, pages 216–239, 1995.
30. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computer Surveys*, 30(2):170–231, 1998.
31. C. Gurret, Y. Manolopoulos, A. Papadopoulos, and P. Rigaux. BASIS: a benchmarking approach for spatial index structures. In *Proceedings Workshop on Spatiotemporal Database Management (STDBM'99)*, pages 152–170, 1999.
32. O. Günther, V. Oria, P. Picouet, J.-M. Saglio, and M. Scholl. Benchmarking spatial joins a la carte. In *Proceedings 7th Conference on Statistical and Scientific Database Management Systems (SSDBM'98)*, pages 32–41, 1998.
33. C. Gurret and P. Rigaux. An integrated platform for the evaluation of spatial query processing strategies. In *Proceedings 9th Conference on Database and Expert Systems Applications (DEXA'98)*, pages 757–766, 1998.
34. A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 47–57, 1984.

35. O. Günther and E. Wong. A dual approach to detect polyhedral intersections in arbitrary dimensions. *BIT*, 31(1):3–14, 1991.
36. C.S. Jensen and R. Snodgrass. Semantics of time-varying information. *Information Systems*, 21(4):311–352, 1996.
37. E. Kawaguchi and T. Endo. On a method of binary picture representation and its application to data compression. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):27–35, 1980.
38. I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings 2nd Conference on Information and Knowledge Management (CIKM'93)*, pages 490–499, 1993.
39. G. Kollios, D. Gunopoulos, and V.J. Tsotras. On indexing mobile objects. In *Proceedings 18th ACM PODS Symposium (PODS'99)*, pages 261–272, 1999.
40. D.E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.
41. A. Kumar, V.J. Tsotras, and C. Faloutsos. Access methods for bi-temporal databases. In *Proceedings Workshop on Temporal Databases*, pages 235–254, 1995.
42. A. Kumar, V.J. Tsotras, and C. Faloutsos. Designing access methods for bi-temporal databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):1–20, 1998.
43. B. Kröll and P. Widmayer. Distributing a search tree among a growing number of processor. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 265–276, 1994.
44. B. Kröll and P. Widmayer. Balanced distributed search trees do not exists. In *Proceedings 4th Int. Workshop on Algorithms and Data Structures (WADS'95)*, volume 995 Lecture Notes in Computer Science, pages 50–61, 1995.
45. S.D. Lang and J.R. Driscoll. Improving the differential file technique via batch operations for tree structured file organizations. In *Proceedings 2nd IEEE Conference on Data Engineering (ICDE'86)*, pages 524–532, 1986.
46. W. Litwin and M.A. Neimat. k-RP$^*_s$ - a high performance multi-attribute scalable data structure. In *Proceedings 4th Conference on Parallel and Distributed Information System (PDIS'96)*, pages 120–131, 1996.
47. W. Litwin, M.-A. Neimat, and D.A. Schneider. LH* - linear hashing for distributed files. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 327–336, 1993.
48. W. Litwin, M.-A. Neimat, and D.A. Schneider. RP*: a family of order preserving scalable distributed data structures. In *Proceedings 20th Conference on Very Large Data Bases (VLDB'94)*, pages 342–353, 1994.
49. J. Van Leeuwen and R.E. Tarjan. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31:245–281, 1984.
50. Y. Manolopoulos and G. Kapetanakis. Overlapping B$^+$-trees for temporal data. In *Proceedings 5th Jerusalem Conference on Information Technology (JCIT'90)*, pages 491–498, 1990.
51. Y. Manolopoulos, E. Nardelli, A. Papadopoulos, and G. Proietti. MOF-tree: a spatial access method to manipulate multiple overlapping features. *Information Systems*, 22(8):465–481, 1997.
52. Y. Manolopoulos, Y. Theodoridis, and V. Tsotras. *Advanced Database Indexing*. Kluwer Academic Publishers, 1999.
53. E. Nardelli. Distributed k-d trees. In *Proceedings 16th Conference of Chilean Computer Science Society (SCCC'96)*, pages 142–154, 1996.
54. E. Nardelli, F. Barillari, and M. Pepe. Design issues in distributed searching of multi-dimensional data. In *Proceedings 3rd International Symposium on Programming and Systems (ISPS'97)*, 1997.
55. E. Nardelli, F. Barillari, and M. Pepe. Distributed searching of multi-dimensional data: a performance evaluation study. *Journal of Parallel and Distributed Computing*, 49(1):111–134, 1998.
56. S. Nishida, H. Nozawa, and N. Saiwaki. Proposal of spatio-temporal indexing methods for moving objects. In *Proceedings Entity-Relationship Workshop (ER'98)*, pages 484–495, 1998.
57. E. Nardelli and G. Proietti. Managing overlapping features in spatial database applications. In *International Computer Symposium (ICS'94)*, pages 1297–1302, 1994.

58. E. Nardelli and G. Proietti. Efficient secondary memory processing of window queries on spatial data. *Information Sciences*, 84:67–83, 1995.
59. E. Nardelli and G. Proietti. Size estimation of the intersection join between two line segment datasets. In *Proceedings 4rd East-European Conference on Advances in Databases and Information Systems (ADBIS'00)*, pages 229–238, 2000.
60. R. Nelson and H. Samet. A population analysis of quadtrees with variable node size. Technical Report CAR-TR-241, University of Maryland, Computer Science Department, 1986.
61. M.A. Nascimento and J.R.O. Silva. Towards historical R-trees. In *Proceedings 13th ACM Symposium on Applied Computing (ACM-SAC'98)*, 1998.
62. M.A. Nascimento, J.R.O. Silva, and Y. Theodoridis. Access structures for moving points. Technical Report TR-33, TimeCenter, 1998.
63. M.A. Nascimento, J.R.O Silva, and Y. Theodoridis. Evaluation for access structures for discretely moving points. In *Proceedings Workshop on Spatio-Temporal Database Management (STDBM'99)*, pages 171–188, 1999.
64. J. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 326–336, 1986.
65. G. Proietti and C. Faloutsos. Accurate modeling of region data. Technical Report 98-137, Carnegie-Mellon University, 1998. Also in *IEEE Transactions on Knowledge and Data Engineering*, 13(6):874-883, November/December 2001.
66. G. Proietti and C. Faloutsos. Selectivity estimation of windows queries for line segment datasets. In *Proceedings 7th Conference on Information and Knowledge Management (CIKM'98)*, pages 340–347, 1998.
67. G. Proietti and C. Faloutsos. I/O complexity for range queries on region data stored using an R-tree. In *Proceedings 15th IEEE Conference on Data Engineering (ICDE'99)*, pages 628–635, 1999.
68. G. Proietti and C. Faloutsos. Analysis of range queries and self spatial join queries on real region datasets stored using an R-tree. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):751-762, September/October 2000.
69. D. Pfoser, C.S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving objects. In *Proceedings 26th Conference on Very Large Data Bases (VLDB'00)*, pages 395–406, 2000.
70. A. Papadopoulos and Y. Manolopoulos. Performance of nearest neighbor queries in R-trees. In *Proceedings 6th International Conference on Database Theory (ICDT'97)*, pages 394–408, 1997.
71. G. Proietti. The MOF$^+$-tree: A space efficient representation of images containing multiple overlapping features. *Journal of Computing and Information*, 2:42–56, 1996.
72. G. Proietti. An optimal algorithm for decomposing a window into its maximal blocks. *Acta Informatica*, 36(4):257–266, 1999.
73. A. Papadopoulos, P. Rigaux, and M. Scholl. A performance evaluation of spatial processing strategies. In *Proceedings 6th Symposium on Spatial Databases (SSD'99)*, pages 286–307, 1999.
74. B. Pagel, H. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance. In *Proceedings 12th ACM PODS Symposium (PODS'93)*, pages 214–221, 1993.
75. S. Ravada and J. Sharma. Oracle8i spatial: Experiences with extensible databases. In *Proceedings 6th Symposium on Spatial Databases (SSD'99)*, pages 355–359, 1999.
76. R.T. Snodgrass and T. Ahn. A taxonomy of time in databases. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 236–246, 1985.
77. Y. Sagiv. Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences*, 3(2):275–296, 1986.
78. H. Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1990.
79. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.
80. S. Šaltenis and C.S. Jensen. R-tree based indexing of general spatio-temporal data. Technical Report TR-45 and Chorochronos CH-99-18, TimeCenter, 1999.
81. S. Šaltenis, C.S. Jensen, S. Leutenegger, and M. Lopez. Indexing the positions of continuously moving objects. In *Proceedings ACM SIGMOD Conference on Management of Data*, pages 331–342, 2000.

82. N. Saiwaki, A. Naka, and S. Nishida. Spatio-temporal data management for highly interactive environment. In *Proceedings 6th IEEE Workshop on Robot and Human Communication (ROMAN'97)*, pages 571–576333, 1997.
83. R.T. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, 1987.
84. B. Salzberg and V. Tsotras. A comparison of access methods for time evolving data. *ACM Computing Surveys*, 31(2):158–212, 1999.
85. T. Tzouramanis, Y. Manolopoulos, and N. Lorentzos. Overlapping B^+-trees: an implementation of a temporal access method. *Data and Knowledge Engineering*, 29(3):381–404, 1999.
86. T. Teraoka, M. Maruyama, Y. Nakamura, and S. Nishida. The MP-tree: a data structure for spatio-temporal data. In *Proceedings 14th IEEE Annual Phoenix Conference on Computers and Communications*, pages 326–333, 1995.
87. A. Tourir. A multi-layer quadtree: a spatial data structure for multi-layer processing. *Geoinformatica*, 2001.
88. Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proceedings 15th ACM PODS Symposium (PODS'96)*, pages 161–171, 1996.
89. Y. Theodoridis, J.R.O. Silva, and M.A. Nascimento. On the generation of spatiotemporal datasets. In *Proceedings 6th Symposium on Spatial Databases (SSD'99)*, pages 147–164, 1999.
90. Y. Theodoridis, T. Sellis, A. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *Proceedings 7th Conference on Statistical and Scientific Database Management Systems (SSDBM'98)*, pages 123–132, 1998.
91. J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.
92. T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees: a spatio-temporal access method. In *Proceedings 6th ACM Symposium on Advances in Geographic Information Systems (ACM-GIS'98)*, pages 1–7, 1998.
93. T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Processing of spatio-temporal queries in image databases. In *Proceedings 3rd East-European Conference on Advances in Databases and Information Systems (ADBIS'99)*, pages 85–97, 1999.
94. T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Multiversion linear quadtrees for spatio-temporal data. In *Proceedings 4rd East-European Conference on Advances in Databases and Information Systems (ADBIS'00)*, pages 279–292, 2000.
95. T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees and window query processing in spatio-temporal databases. *The Computer Journal*, 43(4):325–344, 2000.
96. Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-temporal indexing for large multimedia applications. In *Proceedings 3rd IEEE Conference on Multimedia Computing and Systems (ICMCS'96)*, pages 441–448, 1996.
97. M. Vassilakopoulos and Y. Manolopoulos. Dynamic inverted quadtrees - a structure for pictorial databases. *Information Systems*, 20(6):483–500, 1995.
98. M. Vassilakopoulos, Y. Manolopoulos, and K. Economou. Overlapping for the representation of similar images. *Image and Vision Computing*, 11(5):257–262, 1993.
99. M. Vassilakopoulos, Y. Manolopoulos, and B. Kröll. Efficiency analysis of overlapped quadtrees. *Nordic Journal of Computing*, 2:70–84, 1995.
100. M. Vazirgiannis, Y. Theodoridis, and T. Sellis. Spatio-temporal composition and indexing large multimedia applications. *Multimedia Systems*, 6(4):284–298, 1998.
101. O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proceedings 10th Conference on Scientific and Statistical Database Management*, pages 111–122, 1998.
102. X. Xu, J. Han, and W. Lu. RT-tree - an improved R-tree index structure for spatiotemporal databases. In *Proceedings 4th Symposium on Spatial Data Handling (SDH'90)*, pages 1040–1049, 1990.
103. G. Zimbrao, J. Moreira de Souza, R. Chaomey Wo, and V. Teixeira de Almeida. Efficient processing of spatiotemporal queries in temporal geographical information systems. In *Proceedings 4th Multiconference on Systemics, Cybernetics and Informatics, 6th Conference on Information Systems, Analysis and Synthesis (SCI/ISAS'2000),* Vol.8, Part.II, pages 46–51, 2000.

104. G. Zimbrao, J. Moreira de Souza, and V. Teixeira de Almeida. The temporal R-tree. Technical Report ES-429/99, Federal University of Rio de Janeiro, Computer Science Department, 1999.