Spatio-Temporal Access Methods*

Mohamed F. Mokbel   Thanaa M. Ghanem   Walid G. Aref

Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398
{mokbel,ghanemtm,aref}@cs.purdue.edu

Abstract

The rapid increase in spatio-temporal applications calls for new auxiliary indexing structures. A typical spatio-temporal application is one that tracks the behavior of moving objects through location-aware devices (e.g., GPS). Through the last decade, many spatio-temporal access methods are developed. Spatio-temporal access methods focus on two orthogonal directions: (1) Indexing the past, (2) Indexing the current and predicted future positions. In this short survey, we classify spatio-temporal access methods for each direction based on their underlying structure with a brief discussion of future research directions.

1 Introduction

Spatio-temporal databases deal with objects that change their location and/or shape over time. A typical example of spatio-temporal databases is moving objects in the D-dimensional space. Moving objects learn about their own location via location detection devices, e.g., GPS devices. Then, the objects report their locations to the server using the underlying communication network, e.g., via wireless networks. The server stores the updates from the moving objects and keeps a history of the spatio-temporal coordinates of each moving object. In addition, the server stores additional information to help predict the future positions of moving objects. Typical queries that are supported by such a server include time slice queries e.g., "*Find all objects that cross a certain area at time t*" and window queries "*Find all objects that cross a certain area in the time interval* $[t_1, t_2]$". Time slice queries and window queries may ask about the past, current, or future times. Some queries are concerned only with the past, e.g., trajectory queries "*What is the maximum speed of a certain object in the last hour?*" Other queries are concerned only with the future, e.g., moving window queries "*Find the objects that intersect a moving area in a certain time interval*".

Numerous research have been done in developing spatio-temporal access methods as an auxiliary structure to support spatio-temporal queries. Figure 1 gives the evolution of spatio-temporal access methods with the underlying spatial and temporal structures. Lines in the Figure indicate the relation between a new proposed spatio-temporal index structure and the original structure that is based upon.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work was supported in part by the National Science Foundation under Grants IIS-0093116, EIA-9972883, IIS-0209120, and by Purdue Research Foundation.

The rest of this paper is organized as follow: Section 2 surveys spatio-temporal indexing methods that index the past (i.e., index historical spatio-temporal data). In Section 3, we survey spatio-temporal indexing methods that keep track of the current status of spatio-temporal data. Section 4 surveys the spatio-temporal indexing methods that help answer queries related to the future. In Section 5, we give an overview of available indexing toolkits that can help in implementing spatio-temporal access methods. Finally, Section 6 concludes the paper.

2 Indexing the Past

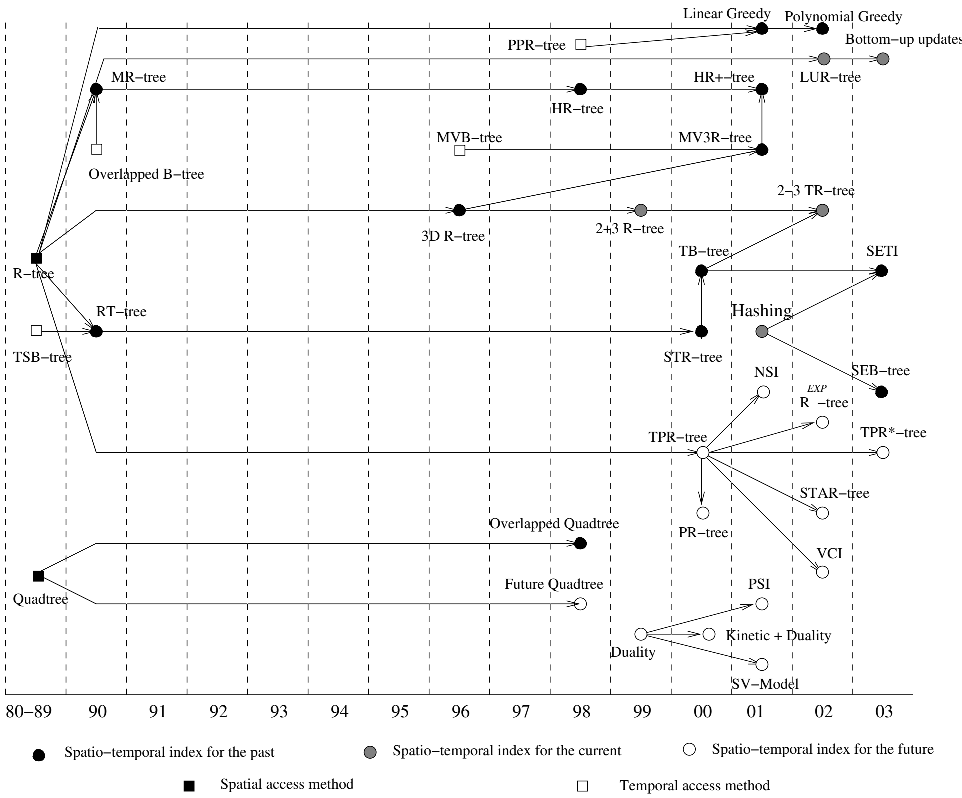

Figure 1: Survey of Spatio-temporal Access Methods

In this section, we are interested in indexing methods for historical spatio-temporal data. The size of the history is continuously increasing over time. Consider moving objects that continuously send their positions. Keeping track of all updates is almost infeasible. Two approaches are used to minimize the history size: (1) Sampling. The stream of data is sampled at certain time positions. Linear interpolation may be used between sample points to form trajectory lines. (2) Update on change only. Moving objects send information only when their data is changed (e.g., change in speed or direction). We categorize the spatio-temporal indexing schemes for historical data into three categories. The first category augments the temporal aspect into already existing spatial access methods. The second category manages both the spatial and temporal aspects into one structure. The third category takes a more radical step by indexing mainly the temporal dimension, while treating the spatial dimension as second priority. The following sections overview these three categories and their corresponding access methods.

2.1 Dealing with the Temporal Dimension

In this category of spatio-temporal indexing methods, the main concern is to handle the spatial domain. Dealing with temporal queries is considered as a secondary issue.

RT-tree [45]: The RT-tree combines the foundation of the R-tree [14] as a spatial access method and the TSB-tree [24] as a temporal access method. In the RT-tree, a new entry is added to the regular R-tree that indicates the start and end times of the current object. An RT-tree entry is of the form (id, MBR, t_s, t_e), where

id is the object identifier, MBR is the objects minimum bounding rectangle, and t_s and t_e give the time interval in which this object is valid. The RT-tree supports spatial queries as efficient as the regular R-tree. However time slice queries and interval queries may span the whole tree.

3D R-tree [41]: The 3D R-tree treats time as yet another dimension in addition to the spatial dimensions. The main idea is to avoid discrimination between spatial and temporal queries. The 3D R-Tree supports both the temporal and spatial queries, although with performance drawbacks. A main drawback is that timeslice queries are no longer dependent on the live entries at the query time, but on the total number of entries in the history.

STR-tree [28]: The STR-Tree is an extension of the R-Tree, with a different insert/split algorithm. Leaf nodes are in the form (id, t_{id}, MBR, o) where t_{id} is the trajectory identifier and o is the orientation of this trajectory in the MBR. The main idea of the STR-Tree is to keep spatial closeness and partial trajectory preservation by trying to keep line segments belonging to the same trajectory together while keeping spatial closeness as the R-Tree. A parameter p is introduced to balance between spatial properties and trajectory preservation. p indicates the number of levels reserved for trajectory preservation. When inserting a new line segment the goal is to insert it as close as possible to its predecessor in the trajectory within p levels. A smaller p decreases the trajectory preservation, while increasing the spatial closeness.

2.2 Overlapping and Multi-version Structures

In the second category of spatio-temporal indexes, the temporal dimension is discriminated from the spatial dimensions. The goal is to keep all spatial data that are alive at one time instance together in one index structure (e.g., the R-tree). The ultimate goal is to build a separate R-tree for each time instance. This approach requires excessive storage.

MR-tree [45]: The MR-tree employs the idea of overlapping B-trees [6] in the context of the R-tree. The main idea is to avoid the storage overhead of having separate R-trees for each timestamp. The saving in storage is achieved by not storing the common objects among consecutive R-trees. Instead, links from different roots point to the same nodes where all the node entries keep their values over the different timestamps. This idea is perfect in the case of a time slice query. The search is directed to the appropriate root, and then a spatial search is performed using the R-tree. However, the performance of time window queries is not efficient. Also, one major drawback is that many entries can be replicated. Consider the case that only one node entry is changed over two consecutive timestamps, then all other node entries need to be replicated in two consecutive R-trees.

HR-tree [25]: The Historical R-tree (HR-tree) is very similar to the MR-tree. The HR-tree has a concrete algorithm and implementation details of using the overlapping B-tree [6] in the context of the R-tree. The same idea of overlapping trees is applied in the context of quadtrees, where it results in *overlapping quadtrees* [43].

HR+-tree [37]: The HR+-tree is designed mainly to avoid the replication of some entries in the HR-tree. The main reason for having duplicate entries in the HR-tree is that the HR-tree has a condition that any node can contain only entries that belong to the same root, i.e., ones that have the same timestamp. The HR+-tree relaxes this condition by allowing entries from different timestamps to reside in the same node. However, the parent of this node in each R-tree has only access to the entries that belong to the parent's timestamp. In other words, a node may have multiple parents, where each parent has access only to a different part of the node.

MV3R-tree [38]: The MV3R-tree is based mainly on the multi-version B-tree (MVB-tree) [5]. The main idea is to build two trees, an MVR-tree to process timestamp queries, and a 3D R-tree to process long interval queries. Short interval queries are optimized to check which tree is to be used based on a threshold value.

Greedy algorithms in the PPR-tree [20]: The partially-persistent R-tree (PPR-tree) [21], designed mainly for bi-temporal databases, is extended to support spatio-temporal applications [20]. However, this would result in highly dead space. To overcome the dead space, artificial object updates are introduced. An optimal greedy algorithm [20] is used to find the optimal locations for the artificial updates in linearly moving objects. This work is extended in [15] to support objects moving using a combination of polynomial functions.

2.3 Trajectory-Oriented Access Methods

The third category of spatio-temporal access methods focus on trajectory-oriented queries. Dealing with spatial queries and gathering spatially closed objects together is of second concern.

TB-tree [28]: The Trajectory-bundle tree (TB-tree) is an R-tree-like structure that strictly preserves trajectories. A leaf node can only contain segments belonging to the same trajectory. As a drawback, line segments of different trajectories that lie spatially close will be stored in different nodes. The TB-tree grows from left to right. The left-most leaf node is the first inserted node and the right-most leaf node is the last inserted one. The TB-tree is an extension of the STR-tree to handle only trajectories.

SETI [8]: The Scalable and Efficient Trajectory Index (SETI) partitions the spatial dimension into static, non-overlapping partitions. The main observation is that the change of the spatial dimension is limited while the temporal dimension is continuously evolving. Thus, the spatial dimensions are partitioned statically. Within each partition the trajectory segments are indexed using an R-tree. Using a good partitioning function results in having line segments of the same trajectory stored in the same partition. Thus, trajectory preservation is achieved by minimizing the effect of the spatial dimensions in the R-tree. A segment that crosses the boundary of two spatial partitions is clipped and is stored twice in both partitions. This may lead to duplicates in the query result.

The SEB-tree [35]: The Start/End timestamp B-tree (SEB-tree) has an idea similar to SETI, where the space is partitioned into zones that may be overlapped. Each zone is indexed using the SEB-tree that considers only the start and end timestamps of the moving objects. Each moving object is hashed to its zone. A key difference over SETI is that there are no trajectories. Instead only two-dimensional points are indexed. By having the spatial zoning partitioning, two-dimensional points that belong to similar trajectories are kept together.

3 Indexing the Current Positions (NOW)

All previous spatio-temporal indexing techniques assume that all movements are known a priori. Thus, only closed trajectories are stored. Current positions of moving objects are neither stored nor queried. The issue of the current positions, or the NOW positions is challenging [10]. In the following, we give an overview of spatio-temporal index structures that help answer queries about NOW.

The 2+3 R-tree [26]: The 2+3 R-tree aims to index both the current and past information of moving objects. The main idea is to have two separate R-trees; one for the current two-dimensional points, and the second for the historical three-dimensional trajectories (two spatial dimensions and one temporal dimension). This idea is similar to the one proposed in the context of the bi-temporal indexes [21]. Once the current object is updated, the object trajectory is constructed with its three-dimensional MBR, and is inserted into the three-dimensional R-tree while being deleted from the two-dimensional R-tree. Depending on the query time, both trees may need to be searched.

The 2-3 TR-tree [1]: The 2-3 TR-tree has the same idea as the 2+3 R-tree where the 2-3 TR-trees also keeps two separate R-trees; a two-dimensional R-tree for the current objects, and a three-dimensional R-tree for the historical data. However, two differences can be distinguished: (1) In the 2-3 TR-tree, the three-dimensional R-tree keeps track of only the multi-dimensional points but not the trajectories; thus avoids the problem of high dead space. (2) The 2-3 TR-tree uses the underlying structure of the TB-tree to allow answers for trajectory-oriented queries.

The LUR-tree [22]: The Lazy Update R-tree (LUR-tree) is concerned only with the current positions of spatio-temporal objects. No historical data is stored into the LUR-tree. Once an object updates its location, the object's old entry is deleted and the new entry is inserted. The LUR-tree aims to handle the frequent updates of moving objects without degrading the performance of the R-tree index structure. The main idea is that as long as the new position of the moving object lies inside its MBR, there is no action taken other than updating the position. Once an object moves out from its MBR, two approaches are proposed: (1) The object is deleted and is reinserted causing the necessary merge and split operations. (2) If the object does not move very far from the MBR, the MBR can be extended to enclose the new location.

Bottom-up Updates [23]: The bottom-up approach for updating R-trees extends the idea of the LUR-tree. Several bottom-up approaches are investigated to accommodate the frequent updates of the moving objects. Examples of these approaches are extending the MBR to enclose the new value and moving the current object to one of the siblings. To avoid excessive I/O's while investigating the siblings and parents for updates, a compact main memory summary structure is introduced.

Hashing [34]: Another approach for keeping only the current information is proposed via hashing. The space is partitioned into zones that may be overlapped. An object does not update its entry to the database until it changes its zone. Thus the database always contains an approximated view of the moving object. To resolve this uncertainty, a filter layer is introduced between the database and the moving objects. The filter layer contains the exact positions of the moving objects. A range query is transformed into a set of zones. If one zone is completely enclosed in the range query, then all the zones entries are returned in the query result. However, if a zone intersects with the range queries, then the zone entries need to be sent to the filter layer to check whether they satisfy the range query or not.

4 Indexing the Current and Future Positions (NOW and the Future)

In this section, we are concerned about the current and future positions of moving objects. To predict the future positions of moving objects, we need to store extra information (e.g., the velocity and the destination). The motion of a moving object in the D-dimensional space is modeled by a reference location $\vec{x_{ref}} = (x_1, x_2, \cdots, x_d)$ at a reference time t_{ref} and a velocity vector $\vec{v} = (v_1, v_2, \cdots, v_d)$. The predicted location $\vec{x_t}$ of the moving object at any instance time $t > t_{ref}$ can be computed by $\vec{x_t} = \vec{x_{ref}} + \vec{v}(t - t_{ref})$. For simplicity, we will assume that objects move in the one-dimensional space. The object movement is modeled by the linear equation $x_t = at + b$, where a and b are constants. Notice that a indicates the constant velocity of the moving object and b is the starting location of the moving object. Also, we assume that the reference location is computed at time $t_{ref} = 0$.

4.1 The Original Space-Time Space

By plotting the equation $x_t = at + b$ in the two-dimensional space, where the horizontal space represents the time and the vertical space represents the location, we obtain a set of line segments in the time-space domain. Then, the problem of indexing future positions is transformed to indexing a set of two-dimensional lines where spatial access methods can be used [11].

PMR-quadtree for moving objects [40]: Tayeb et al. [40] use the PMR-quadtree [27] as their underlying spatial access methods for indexing the future trajectories. A key point is that when an update of moving objects occur, the whole index structure is destroyed and is rebuilt given the new information. To avoid excessive update operations, the index is rebuilt every ΔT time units. In the abstract level, the infinite time dimension is partitioned into equal size time slices, each with size ΔT. For each slice, a new PMR-quadtree is built based on the motion equation. However, due to the storage limit, only the current PMR-Quadtree is stored.

Generally, using spatial access methods to index the future trajectories has two main drawbacks: (1) Large amount of dead space due to representing the trajectory by its minimum bounding rectangle. (2) Data is skewed since all the trajectories have the same end time value.

4.2 Transformation Methods

To overcome the drawbacks of spatio-temporal indexing in the time-space domain, the time-space domain is transformed into another space. The main idea is that it is easier to represent and query the data in this new space representation.

Duality transformation [19]: Kollios et al. [19] use the duality transformation to transform a line segment (e.g., trajectory) from the time-space domain into a point in the two-dimensional space. The main idea is to represent the equation $x_t = at + b$ by the two-dimensional point (a, b) in a dual two-dimensional space where the velocity a is the horizontal dimension and the reference location b is the vertical dimension. Due to the highly skewed distribution in the $dual$ space, a kd-tree based spatial index (e.g., the LSD-tree [17]) is used instead of an R-tree. Since R-trees try to cluster data points into square regions, they will split using only the velocity dimension. However, a kd-tree-based index will use both dimensions in splitting. A range query that is a rectangle in the $primal$ space-time space is transformed into a polygon query in the $dual$ velocity-location space. Thus, the algorithm proposed by Goldstein et al. [13] is used to answer range queries.

Duality Transformation with the Kinetic Data Structure [2]: Agarwal et al. [2] use another form of the *duality* transformation. A moving object in the two-dimensional space (x, y) is plotted as a three-dimensional trajectory (x, y, t). The trajectory is projected into the (x, t) and (y, t) plans. The *duality* transformation is applied to both plans. The answer of the range query is the union of two range queries in the two plans. Instead of having a *kd*-tree-like structure (as in [19]), the so-called *kinetic data structure* [4] is used to index the dual space.

SV-Model [9]: Chon et al. [9] takes a more radical step, where they do not represent a moving object by its trajectory. Instead, a moving object is modeled by four parameters (s, e, t_s, v_0) for the starting location, the destination, the starting time, and the initial velocity, respectively. With the restriction that only two out of the four parameters can change their values, there are six different combinations to consider. Among these combinations, the best choice is to consider the starting location s and the velocity v_0 as constants. Thus the dual space will have the starting time t_s as the horizontal dimension and the destination e as the vertical dimension. This model is termed the SV-Model to indicate the constant starting location and velocity. Rectangular range queries are transformed to polygon queries in the *dual* space. The *dual* space is indexed by the SS-tree [44]. The assumption that all moving objects have the same starting location is handled by normalizing the motion of all moving objects to start from 0. For the constant velocity constraint, the assumption can be realistic in cases of highway traffic. In the case where there are velocity variations, the velocity is quantized to discrete values. For each value, a separate index structure is used.

PSI [29]: Porkaew et al. [29] propose the Parametric Space Indexing Technique (PSI). In the PSI approach, an R-tree is used to index a $(2d+1)$-dimensional space, where d dimensions correspond to the reference location $\vec{x_{ref}}$, and d dimensions correspond to the velocity $\vec{v}$, while one dimension corresponds to the time. Object movement is modeled by a $(2d+1)$-dimensional trajectory that is enclosed by its minimum bounding rectangle. The main idea is that the temporal range $[t_s, t_e]$ in which the motion is valid is stored in the index. Also, there is no notion of global time reference that objects refer to.

In summary, the transformation techniques suffer from three main drawbacks: (1) The *dual* space cannot capture all the information that is originally in the *primal* space. (2) There is no guarantee that objects that are near to each other in the *primal* space will still be near to each other in the *dual* space. (3) Rectangular range queries in the *primal* space are always transformed into polygon range queries in the *dual* space, which calls for complicated algorithms for evaluation.

4.3 Parametric Spatial Access Methods

A new trend of spatio-temporal access methods is to index the original time-space with parametric rectangles. The main idea is to make the bounding rectangles functions of time so that the enclosed moving objects will be in the same rectangles. In this case, for any time instance t, a snapshot of the index structure can be computed and evaluated for any query.

TPR-tree [33]: The Time Parameterized R-tree (TPR-tree) employs the idea of parametric bounding rectangles in the R-tree. At the construction time, the TPR-tree builds the so-called *conservative bounding rectangles* that enclose a set of moving objects. The lower bound of the conservative bounding rectangle is set to move with the minimum speed of the enclosed points, while the upper bound is set to move with the maximum speed of the enclosed points. In this case, the conservative bounding rectangle never shrinks, and is guaranteed to always contain the enclosed moving objects. To avoid the case where the bounding rectangles grow to be very large, whenever the position of an object o is updated, all the bounding rectangles on the nodes along the path to the leaf at which o is stored are recomputed.

PR-tree [7]: The PR-tree is similar to the TPR-tree. However, the PR-tree considers the problem of moving objects with spatial extents that are represented by parametric rectangles. Each parametric rectangle has a time interval that represents the start time and the end time of its movement. In contrast to the TPR-tree, where objects are considered moving forever, the PR-tree has the knowledge of the end time of moving objects. Thus a moving object is represented as a polygon on the space rather than a trajectory. Given the movement end times, the bounding rectangles of a set of moving objects (represented as polygons) can be computed as the convex hull of the moving objects.

NSI [29]: The NSI tree is similar to the TPR-tree in the sense that both define parametric bounding rectangles for moving objects. However, the difference is in the way the bounding rectangle is defined.

VCI R-tree [30]: The main idea of the Velocity Constrained Indexing (VCI) is to add an additional field v_{max} to each R-tree node. v_{max} stores the maximum allowed speed over all objects covered by this node. For any query at time t, all bounding rectangles are expanded using v_{max}. This approach is similar to that of the TPR-tree in the sense that both trees consider expanding the bounding rectangles over time to contain all the enclosed objects. However, the underlying model is different. In VCI indexing, there is no need to know the exact speed of each moving object. Instead, there is a restriction that all moving objects cannot exceed a certain maximum speed. In the TPR-tree, the exact velocity of moving objects is needed. On the other side, many false positives appear in the VCI indexing. The VCI indexing is designed specially to handle the issue of the shared execution of continuous queries.

STAR-tree [31]: The Spatio-temporal Self Adjusting R-tree (STAR-tree) is similar to the TPR-tree, with the introduction of the notion of *self-adjustment*. Whenever the query performance degrades, the STAR-tree adjusts itself without any input from the user.

R^{EXP}**-tree [32]:** The R^{EXP}-tree is an extension of the TPR-tree to handle moving objects with expiration times. The main idea is to avoid the drawback that may result for moving objects that do not update their movement for a long time. To benefit from the expiration time information, the R^{EXP}-tree employs a new type of bounding regions. In addition, the R^{EXP}-tree implements a lazy technique for removing expired entries from the index until the bounding rectangles are recomputed.

TPR*-tree [39]: The TPR*-tree uses exactly the same structure and assumptions as the TPR-tree [33]. Unlike the TPR-tree where it uses the same insert and delete functions of the R*-tree, the TPR*-tree provides a new set of insertion and deletion algorithms that aim at minimizing a certain cost function.

5 Available Indexing Toolkits

In this section, we give a brief overview of publicly available indexing toolkits that can be used to implement spatio-temporal access methods.

5.1 GiST: Generalized Search Trees for Database Systems

GiST [16] defines a framework of basic interfaces required to construct a hierarchical access method for database systems. GiST supports the class of balanced trees (e.g., the B-tree, the R-tree, and the SR-tree [18]). The main architecture of GiST contains two parts: *internal* methods and *type-specific* methods. Internal methods are the common methods to all balanced trees (e.g., search, insert, and delete). Such methods are hard-coded inside

GiST, and cannot be altered by the user. Type-specific methods are provided by the user based on the underlying index structure. Examples of type-specific methods are $Consistent, Union, Penalty$, and $PickSplit$. The reader is referred to [16] for more details.

GiST can be used to support spatio-temporal indexing methods that are based on R-tree-like structures. For example, the TPR-tree [33] is already implemented [42] using the publically available GiST code [12].

5.2 SP-GiST: A Framework for Supporting the Class of Space-Partitioning Trees

SP-GiST (Space-partitioning Generalized Search Tree) [3] is an extensible database index structure for the class of space-partitioning trees (e.g., the trie, the k-d tree, the quadtree, and their variants). SP-GiST allows fast realization of instances of space-partitioning index trees inside a commercial database system.

SP-GiST has *interface parameters* and *external methods* that allow SP-GiST to implement instance indexes of the class of space-partitioning trees and reflect the structural and behavioral differences among these trees. In addition, SP-GiST has *internal methods* that reflect the similarity among space-partitioning trees for insertion, deletion, and search that are already implemented inside the SP-GiST index engine. Table 1 gives the interface parameters and external methods of SP-GiST.

The realization of any space-partitioning tree T inside SP-GiST is achieved by providing the interface parameters and external methods for T. For example, Table 2 gives the realization of the k-d tree inside SP-GiST. Examples for the realization of other space-partitioning trees inside SP-GiST can be found in [3]. SP-GiST can be used to implement spatio-temporal indexing that rely on space-partitioning trees (e.g., [40, 43]). The SP-GiST code is publically available [36].

$NodePredicate$	The predicate to be used in the index nodes of a space-partitioning tree.
$KeyType$	Type of data in the leaf-level of the tree.
$NoOfSpacePartitions$	The number of disjoint partitions produced at each decomposition.
$Resolution$ $ShrinkPolicy$	Limit the number of space decompositions.
$BucketSize$	The maximum number of data items a data node can hold.
$Consistent()$	Boolean function used by the search method as a navigation guide through the space-partitioning tree.
$PickSplit()$	Boolean function that define a way of splitting the entries into a number of partitions and returns whether further partitioning should take place or not.
$Cluster()$	This method defines how tree nodes are clustered into disk pages.

Table 1: SP-GiST Interface Parameters and External Methods

Parameters	$NodePredicate$ = “left”, “right”, or blank; $KeyType$ = Point; $NoOfSpacePartitions$ = 2; $ShrinkPolicy$ = Leaf Shrink; $BucketSize$ = 1;
$Consistent(E, q, level)$	If ($level$ is odd AND $q.x$ satisfies $E.p.x$) OR ($level$ is even AND $q.y$ satisfies $E.p.y$) Return True, else Return False
$PickSplit(P, level)$	Put the old point in a child node with predicate “blank”; put the new point in a child node with predicate “left” or “right”; Return False

Table 2: Realization of k-d Tree Inside SP-GiST

6 Conclusion

In this short survey, we presented an overview of existing spatio-temporal index structures. Spatio-temporal indexing methods are classified based on the type and time of the queries they can support (e.g., the past, current, and future queries). With the variety of spatio-temporal access methods, it becomes essential to have a general and extensible framework to support the class of spatio-temporal indexing. One approach is to use the already existing extensible index structures (e.g., GiST and SP-GiST). Another approach is to develop a special framework for spatio-temporal indexing to capture the special needs of such class (e.g., the type of queries, the continuously evolving objects, and the frequent updates). There is still a lot of research work that needs to be investigated in spatio-temporal indexing. Most of the work so far supports selection operators and range queries. More research is needed to support other kinds of operators (e.g., spatio-temporal join) and queries (e.g., nearest-neighbor queries).

References

[1] M. Abdelguerfi, J. Givaudan, K. Shaw, and R. Ladner. The 2-3 TR-tree, A Trajectory-Oriented Index Structure for Fully Evolving Valid-time Spatio-temporal Datasets. In *Proc. of the ACM workshop on Adv. in Geographic Info. Sys., ACM GIS*, pages 29–34, Nov. 2002.

[2] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. In *Proc. of the ACM Symp. on Principles of Database Systems, PODS*, pages 175–186, May 2000.

[3] W. G. Aref and I. F. Ilyas. SP-GiST: An Extensible Database Index for Supporting Space Partitioning Trees. *Journal of Intelligent Information Systems , JIIS*, 17(2-3):215–240, 2001.

[4] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. of the ACM-SIAM symposium on Discrete algorithms, SODA*, pages 747–756, 1997.

[5] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal*, 5(4):264–275, 1996.

[6] F. W. Burton, J. G. Kollias, D. G. Matsakis, and V. G. Kollias. Implementation of Overlapping B-trees for Time and Space Efficient Representation of Collections of Similar Files. *The Computer Journal*, 33(3):279–280, 1990.

[7] M. Cai and P. Revesz. Parametric R-Tree: An Index Structure for Moving Objects. In *Proc. of the Intl. Conf. on Management of Data, COMAD*, Dec. 2000.

[8] V. P. Chakka, A. Everspaugh, and J. M. Patel. Indexing Large Trajectory Data Sets with SETI. In *Proc. of the Conf. on Innovative Data Systems Research, CIDR*, Asilomar, CA, Jan. 2003.

[9] H. D. Chon, D. Agrawal, and A. E. Abbadi. Storage and Retrieval of Moving Objects. In *Mobile Data Management*, pages 173–184, Jan. 2001.

[10] J. Clifford, C. E. Dyreson, T. Isakowitz, C. S. Jensen, and R. T. Snodgrass. On the Semantics of "Now" in Databases. *ACM Trans. on Database Systems , TODS*, 22(2), 1997.

[11] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[12] GiST: http://gist.cs.berkeley.edu/.

[13] J. Goldstein, R. Ramakrishnan, U. Shaft, and J.-B. Yu. Processing Queries By Linear Constraints. In *Proc. of the ACM Symp. on Principles of Database Systems, PODS*, pages 257–267, May 1997.

[14] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of the ACM Intl. Conf. on Management of Data, SIGMOD*, pages 47–57, June 1984.

[15] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient Indexing of Spatiotemporal Objects. In *Proc. of the Intl. Conf. on Extending Database Technology, EDBT*, pages 251–268, Czech Republic, Mar. 2002.

[16] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proc. of the Intl. Conf. on Very Large Data Bases, VLDB*, pages 562–573, Sept. 1995.

[17] A. Henrich, H.-W. Six, and P. Widmayer. The lsd tree: Spatial access to multidimensional point and nonpoint objects. In *Proc. of the Intl. Conf. on Very Large Data Bases, VLDB*, pages 45–53, Aug. 1989.

[18] N. Katayama and S. Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proc. of the ACM Intl. Conf. on Management of Data, SIGMOD*, pages 369–380, May 1997.

[19] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. In *Proc. of the ACM Symp. on Principles of Database Systems, PODS*, pages 261–272, June 1999.

[20] G. Kollios, V. J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing Animated Objects Using Spatiotemporal Access Methods. *IEEE Trans. on Knowledge and Data Engineering, TKDE*, 13(5):758–777, 2001.

[21] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE Trans. on Knowledge and Data Engineering, TKDE*, 10(1):1–20, 1998.

[22] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *Mobile Data Management, MDM*, pages 113–120, Jan. 2002.

[23] M. Lee, W. Hsu, C. Jensen, B. Cui, and K. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *Proc. of the Intl. Conf. on Very Large Data Bases, VLDB*, Sept. 2003.

[24] D. B. Lomet and B. Salzberg. Access Methods for Multiversion Data. In *Proc. of the ACM Intl. Conf. on Management of Data, SIGMOD*, pages 315–324, May 1989.

[25] M. A. Nascimento and J. R. O. Silva. Towards historical R-trees. In *Proc. of the ACM Symp. on Applied Computing, SAC*, pages 235–240, Feb. 1998.

[26] M. A. Nascimento, J. R. O. Silva, and Y. Theodoridis. Evaluation of Access Structures for Discretely Moving Points. In *Proc. of the Intl. Workshop on Spatio-Temporal Database Management, STDBM*, pages 171–188, Sept. 1999.

[27] R. C. Nelson and H. Samet. A Consistent Hierarchical Representation for Vector Data. In *Proc. of the ACM SIGGRAPH*, pages 197–206, Aug. 1986.

[28] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *Proc. of the Intl. Conf. on Very Large Data Bases, VLDB*, pages 395–406, Sept. 2000.

[29] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying Mobile Objects in Spatio-Temporal Databases. In *Proc. of the Intl. Symp. on Advances in Spatial and Temporal Databases, SSTD*, pages 59–78, Redondo Beach, CA, July 2001.

[30] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.

[31] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects. In *Proc. of the Workshop on Alg. Eng. and Experimentation, ALENEX*, pages 178–193, Jan. 2002.

[32] S. Saltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *Proc. of the Intl. Conf. on Data Engineering, ICDE*, Feb. 2002.

[33] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. of the ACM Intl. Conf. on Management of Data, SIGMOD*, pages 331–342, May 2000.

[34] Z. Song and N. Roussopoulos. Hashing Moving Objects. In *Mobile Data Management*, pages 161–172, Jan. 2001.

[35] Z. Song and N. Roussopoulos. SEB-tree: An Approach to Index Continuously Moving Objects. In *Mobile Data Management, MDM*, pages 340–344, Jan. 2003.

[36] SP-GiST: http://www.cs.purdue.edu/homes/aref/dbsystems_files/SP-GiST/index.html.

[37] Y. Tao and D. Papadias. Efficient Historical R-trees. In *Proc. of the Intl. Conf. on Scientific and Statistical Database Management, SSDBM*, pages 223–232, July 2001.

[38] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *Proc. of the Intl. Conf. on Very Large Data Bases, VLDB*, pages 431–440, Sept. 2001.

[39] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In *Proc. of the Intl. Conf. on Very Large Data Bases, VLDB*, Sept. 2003.

[40] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A Quadtree-Based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3):185–200, 1998.

[41] Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-Temporal Indexing for Large Multimedia Applications. In *Proc. of the IEEE Conference on Multimedia Computing and Systems, ICMCS*, June 1996.

[42] TPR-tree: http://www.cs.auc.dk/TimeCenter/software.htm.

[43] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping Linear Quadtrees: A Spatio-Temporal Access Method. In *Proc. of the ACM workshop on Adv. in Geographic Info. Sys., ACM GIS*, pages 1–7, Nov. 1998.

[44] D. A. White and R. Jain. Similarity Indexing with the SS-tree. In *Proc. of the Intl. Conf. on Data Engineering, ICDE*, pages 516–523, Feb. 1996.

[45] X. Xu, J. Han, and W. Lu. RT-Tree: An Improved R-Tree Indexing Structure for Temporal Spatial Databases. In *Proc. of the Intl. Symp. on Spatial Data Handling, SDH*, pages 1040–1049, July 1990.