Spatio-Temporal Access Methods: Part 2 (2003 - 2010)*

Long-Van Nguyen-Dinh[1] Walid G. Aref[1] Mohamed F. Mokbel[2]

[1]Department of Computer Science, Purdue University, West Lafayette, IN 47907-1398

[2]Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455

{lnguyend,aref}@cs.purdue.edu, mokbel@cs.umn.edu

Abstract

In spatio-temporal applications, moving objects detect their locations via location-aware devices and update their locations continuously to the server. With the ubiquity and massive numbers of moving objects, many spatio-temporal access methods are developed to process user queries efficiently. Spatio-temporal access methods are classified into four categories: (1) Indexing the past data, (2) Indexing the current data, (3) Indexing the future data, and (4) Indexing data at all points of time. This short survey is Part 2 of our previous work [28]. In Part 2, we give an overview and classification of spatio-temporal access methods that are published between the years 2003 and 2010.

1 Introduction

Spatio-temporal databases deal with moving objects that change their locations over time. Generally, moving objects report their locations obtained via location-aware devices to a spatio-temporal database server. The server can store all updates from the moving objects so that it is capable of answering queries about the past. Some applications need to know current locations of moving objects only. In this case, the server may only store the current status of the moving objects. To predict future positions of moving objects, the spatio-temporal database server may need to store additional information, e.g., the objects' velocities.

Many query types are supported by a spatio-temporal database server, e.g., range queries "*Find all objects that intersect a certain spatial range during a given time interval*", k-nearest neighbor queries "*Find k restaurants that are closest to a given moving point*", or trajectory queries "*Find the trajectory of a given object for the past hour*". These queries may execute on past, current, or future time data. A large number of spatio-temporal index structures has been proposed to support spatio-temporal queries efficiently.

In [28], we give a brief survey of spatio-temporal access methods that are published on or before 2003. This survey is Part 2 of [28], where we cover and classify spatio-temporal access methods published in the years 2003 to 2010. Since 2003, new spatial-temporal access methods have been developed that use entirely new approaches or that address weaknesses in existing approaches. Besides having separate indexing structures for past data, present data, or future data, some access methods have been proposed to deal with data at all points in time. Several spatio-temporal indexing methods are proposed for special environments, e.g., road networks or indoor networks. Figure 1 gives an overall overview of spatio-temporal access methods until 2010. Lines in the Figure indicate the evolutionary relationships among the spatio-temporal access methods.

The rest of this paper proceeds as follows. Sections 2, 3, and 4 survey spatio-temporal indexing methods for historical, current time, and future prediction data, respectively. Section 5 surveys spatio-temporal indexing methods for combined past, present, and future data. Section 6 gives concluding remarks.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*The authors acknowledge the support from the NSF under Grant No. IIS-0811954.

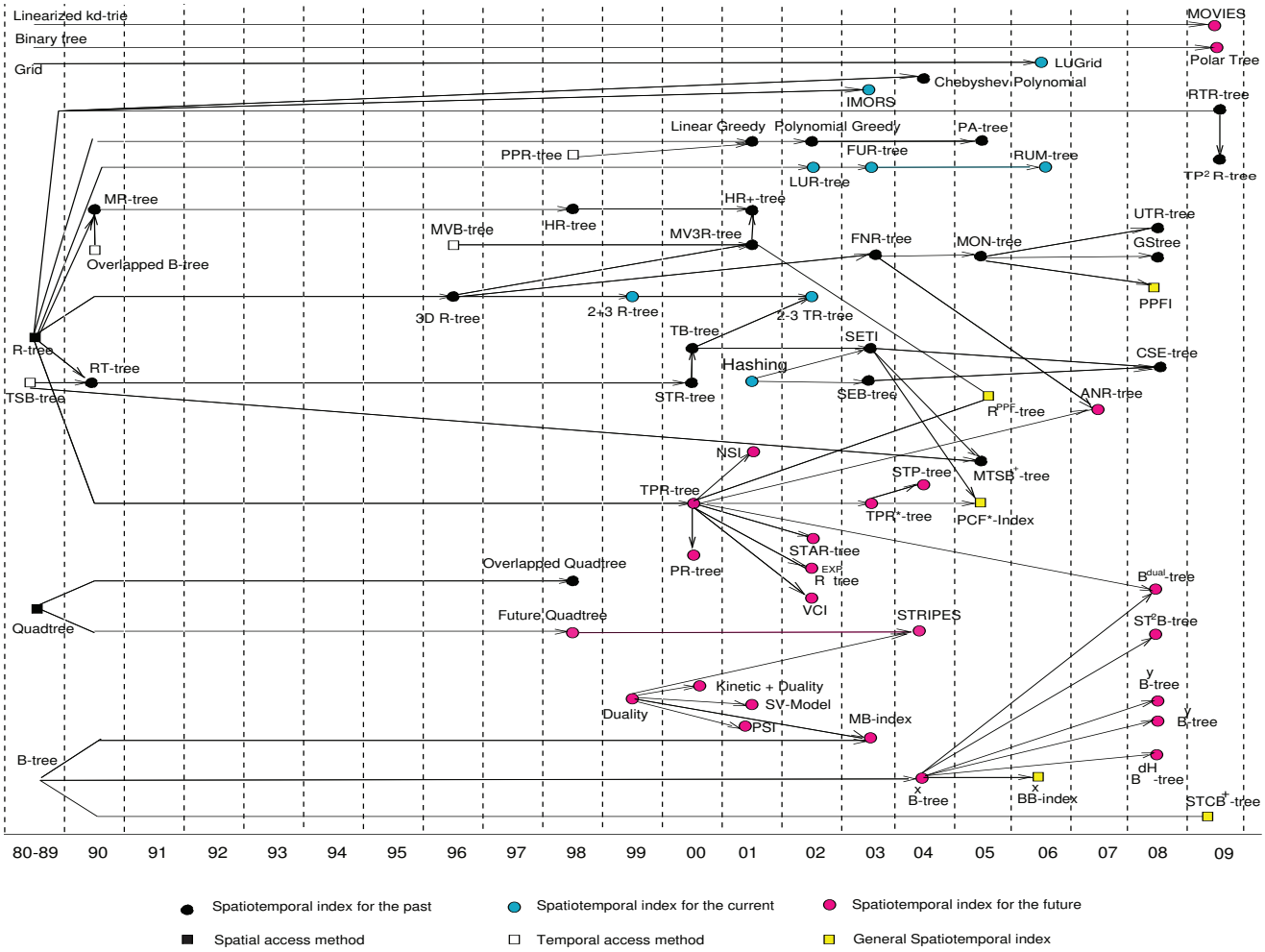

Figure 1: Survey of Spatiotemporal Access Methods.

2 Indexing the Past

In this section, we survey methods for indexing historical spatio-temporal data. We follow the same categorization as the one in Part 1 [28]. Since moving objects update their locations frequently, storing all historical data may not be feasible. Sampling helps reduce the size of stored historical data. Linear or non-linear interpolation techniques may be used between consecutive sampled data points to form trajectory paths. In contrast to sampling, objects may elect to update their locations only when they experience significant changes in location.

2.1 Three-dimensional Structures

In this category, time is modeled as the third dimension in addition to the two location dimensions (X,Y).

MTSB-tree [49]: The Multiple TSB-tree supports the historical and spatial range close-pair queries for moving objects. A range close-pair query finds pairs of objects that are closer than a given threshold distance during a given time interval and that both lay inside a given spatial range. Similar to SETI [4], in the MTSB-tree, space is divided into disjoint cells, each maintaining a temporal index for its objects' movements. Unlike SETI's R-tree-based [17] temporal index, the MTSB-tree uses a Time-Split B-tree (TSB-tree) [26] in each cell. Thus, all trajectory segments within the spatial and temporal range query are produced in increasing time order by merging the query results from the TSB-trees of all cells. Then, a stream of such trajectory segments is fed into a plane-sweep algorithm that sweeps both time and space to produce close-pair objects on the fly.

FNR-tree [16]: The Fixed Network R-Tree indexes objects moving in a two-dimensional (2D) fixed network. A fixed network is a set of connected line segments. The FNR-tree uses a 2D R-tree to index the static line segments of the network. Each leaf entry contains a line segment and a pointer to the root of a 1D R-tree that indexes the time intervals of objects' movements. Each time a moving object leaves a given line segment of the network, the FNR-tree searches the 2D R-tree to find the line segment and then inserts an interval (TEntrance, TExit) into the corresponding 1D R-tree. Every new entry is simply inserted in the rightmost leaf of the 1D R-tree. Thus, in the FNR-tree, the objects' movements are assumed not to begin or end in the middle of a line segment and their speeds or directions cannot be changed in the middle of a line segment.

MON-tree [10]: The MON-tree is an extension to the FNR-tree. In the MON-tree, the constrained network is modeled as a set of junctions (nodes) and routes (edges) in which routes are non-intersecting polylines. Unlike the FNR-tree where an object's position is updated only when the object leaves a line segment, in the MON-tree the object's position is associated with a real number between 0 and 1 that is the relative position of the object within a route. Similar to the FNR-tree, the MON-tree uses a top 2D R-tree to index the polylines' bounding boxes. For each polyline, a bottom 2D R-tree indexes the movements of the objects within the polyline. An object's movement is represented by a spatial range $(p1, p2)$ and a temporal interval $(t1, t2)$. The MON-tree uses a hash structure with the polyline identifier as the hash key to locate the polyline's corresponding bottom R-tree into which an object's movement is to be inserted.

2.2 Overlapping and Multi-version Structures

In this category, the temporal dimension is discriminated from the spatial dimensions in a variety of ways.

PA-tree [29]: The Parametric tree indexes historical trajectory data of moving objects by dividing the temporal domain into m disjoint time intervals. Each trajectory is split into a series of m line segments that correspond to the m disjoint time intervals and each segment is approximated with a single continuous Chebyshev polynomial. The PA-tree restricts the maximum deviation between the approximation and the original movement to make sure the approximation is conservative and tight. A two-level index indexes trajectory segments within each time interval. The first level is an R*-tree-like structure that indexes the two leading coefficients of the Chebyshev polynomial describing the object movement along each dimension and the maximum deviation errors. In the second level, higher-degree coefficients as well as the corresponding maximum deviations are stored for each trajectory segment. If the first level index is unable to determine whether the trajectory satisfies the query predicates, the additional coefficients in the second level index are used in the filtering step.

GStree [21]: The Graph Strip Tree indexes the current or past positions of moving objects in a constrained graph defined by edges and vertexes as in the MON-tree [10]. A strip tree [1] stores and models each edge that can be a list of line segments or curves by interpolating the edge linearly. The GStree is a balanced binary tree in which a leaf node is a minimum bounding area of an edge. These leaf nodes are merged two at a time to form internal nodes, such that the two merged nodes result in a minimum bounding area for their parent node. These internal nodes are merged in the same manner until the GStree reaches the root. The GStree assumes that all objects' positions are updated continuously in a constant update interval. It also assumes that there are $(m + 1)$ positions for each moving object defined on m equal intervals in the time domain $[0, T]$. Each leaf node of the GStree points to two data structures: the strip tree representing the edge stored in the leaf node, and an array of size m to store m interval trees of m time intervals. Each interval tree represents the moving objects' positions during the corresponding time interval on the corresponding edge. The strip trees and the GStree are stored in memory while the arrays are stored in disk. When new objects on an edge appear at a new time interval, a new interval tree containing these objects is created and is concatenated to the array.

2.3 Trajectory-oriented Access Methods

This category of access methods for historical spatio-temporal data deals with trajectory-oriented queries [35] that are either topological or navigational. Topological queries involve the topology of trajectories, e.g., check if an object enters or bypasses a given area. Navigational queries involve derived information, e.g., the maximum

speed of an object during the last hour, or the orientation of an object (East, North, etc.).

CSE-tree [45]: The Compressive Start-End Tree has an idea similar to those of SETI [4] and the SEB-tree [40] to index the trajectories of moving objects. As in SETI, the space is partitioned into disjoint cells, and then a CSE-tree as a temporal index is maintained for each spatial cell. As in the SEB-tree, the CSE-tree stores a temporal interval as a 2D point. All segments indexed by a CSE-tree are divided into several groups according to the end time of the segment. A B+-tree-based End-Time index is built to index all groups. For each group, another B+-tree-based Start-Time index is organized to index all segments in that group. The CSE-tree is used in GPS data sharing applications, where people are interested in exchanging their GPS tracks. Thus, a track that consists of a sequence of track segments is inserted into the GPS data sharing system at the time a user uploads it, rather than at the time the GPS device updates a new location. Therefore, each Start-Time index has a different frequency update rate. When the update frequency of a Start-Time index goes below a threshold, the Start-Time index is transformed to a sorted dynamic array to utilize the storage.

Polar Tree [33]: The Polar Tree is an in-memory unbalanced binary tree to index object headings (orientations) of interest to a given focal point (site). Each site has a focal scope that represents its maximum range for detecting the objects moving in its vicinity. A focal scope is a circle of radius R centered at the fixed location of the site. An object's heading is represented as a counterclockwise angle w.r.t. the positive x-axis. By collecting objects' headings, the mode of progression can be observed. The Polar Tree divides the scope of a site into non-overlapping polar sectors corresponding to the nodes in the Polar Trees. The root node represents the entire scope of the site and has no entries. An internal node with a radius, say r, has exactly two children with radius $r/\sqrt{(2)}$ and the central angle w of the internal node is bisected into equal parts for its children. When a leaf node overflows, it becomes an internal node with two new children. Each object's heading is inserted at suitable internal nodes or leaves of the Polar Tree built for a specific focal point. Therefore, the Polar Tree is used to monitor objects' orientations and detect if many objects get closer to or move away from a site.

Chebyshev Polynomial [3]: A d-dimensional spatio-temporal trajectory is projected to its d dimensions to have d 1D series that are each approximated by a Chebyshev polynomial [27] and is represented by a vector of coefficients. All coefficient vectors are combined to form one vector or one single multi-dimensional point representing the trajectory. An arbitrary multidimensional access method indexes the points of all trajectories. It assumes that all trajectories are sampled at the same set of timestamps. Thus, the polynomial approximations for all trajectories have the same degree and the time dimension can be ignored. The Lower Bounding Lemma states that the Euclidean distance between two trajectories is lower bounded by the Euclidean distance between the two vectors of Chebyshev coefficients weighted by a factor. Based on the lemma, a query to find all trajectories within a range from a given trajectory or a kNN query of a specified trajectory can be answered using the index.

RTR-tree and TP^{2}R-tree [19]: The indoor movement is more constrained than outdoor Euclidean movement and is characterized by entities (doors, rooms, hallways, etc.) and topology predicates (enter, leave, cross, etc.). Location data is collected by RFID readers that are deployed at fixed locations in the indoor space to locate moving objects. The RTR-tree and TP^{2}R-tree [19] are two R-tree-based indexes for trajectories of objects moving in symbolic indoor space. The RTR-tree is a 2D R-tree on the Reader-Time space that organizes a trajectory as a list of line segments. To answer a range query, the Euclidean range space is transformed into sets of RFID readers and then the RTR-tree is searched for the corresponding readers. The TP^{2}R-tree uses the same Reader-Time space, but transforms a trajectory to a set of points, augmented with a temporal parameter. This way, the TP^{2}R-tree achieves better node organization.

3 Indexing the Current Positions (NOW)

The notion of the "current" or NOW in database systems is challenging (e.g., see [9]). In this section, we give an overview of spatio-temporal access methods that answer queries on the NOW status of moving objects.

LUGrid [47]: The Lazy-Update Grid-based index adapts the grid file [30] to exploit lazy insertion and deletion to handle the frequent updates to the locations of continuously moving objects. Incoming updates are grouped based on the to-be-updated disk-page, are stored in a memory grid, and then are lazily flushed into disk

in batch. Thus, multiple updates are reduced to only a single disk update. In lazy deletion, obsolete entries remain temporarily in disk rather than being immediately deleted. The LUGrid uses an in-memory memo structure to keep track of current object ids. This way, the obsolete entries can be referred from the current entries in the memo and are lazily removed when their disk pages are accessed.

RUM-tree [46, 39]: The R-tree with update memo deals with the frequent updates of moving objects. Instead of deleting the object from the old location and reinserting it into the new location, upon a location update, we just insert the object in the new location. This results in multiple obsolete (old) locations of an object. As in the LUGrid [47], the RUM-tree uses an in-memory memo to track the obsolete entries to avoid purging old entries immediately during an update process. Upon touching a disk page, say p, e.g., during query processing, and before reporting an object, say o, as output, o's version in p is checked against the memo to make sure that o's version is current. A garbage cleaner (GC) inside the RUM-tree removes the obsolete entries lazily and makes sure that the size of the in-memory memo is bounded. Various mechanisms are used by GC to remove obsolete entries including a vacuum cleaner approach that regularly sweeps the space with some frequency and a clean-upon-touch approach that cleans a leaf node whenever it is fetched during an insert or update operation.

IMORS [20]: Indexing Moving Objects on Road Sectors indexes the current positions of objects with high update frequencies moving on a fixed road network. IMORS uses an R*-tree to index road sectors (non-intersecting segments of a road, each bounded by two intersection points). Each road sector entry on leaf nodes of the R*-tree points to a road sector block (BRS) that contains the identifiers of the moving objects on the road sector. In addition, data blocks (Bdata) store data records of the velocities and coordinates of moving objects. There are bi-directional pointers between each record in Bdata and a corresponding moving object in a BRS. To insert a new moving object into IMORS, one data record with the object's coordinates and other attributes is inserted into a Bdata and the object identification (oid) is inserted into a BRS corresponding to a road sector where the new object is contained. To update a new position of a moving object, first, the record of the object from the Bdata is retrieved by its oid and then is updated. If the object is still on the same BRS, no more operations are needed. Otherwise, the oid entry in the old BRS found directly by the pointer from the data record is deleted and is inserted into the new BRS. The road sector can be found by searching the R*-tree.

4 Indexing the Future Positions

To index the future positions of moving objects, access methods predict the future locations of the objects in a variety of ways, e.g., using the objects' reference locations and their velocities.

4.1 The Original Space-time Space

Here, the predicted positions of moving objects are calculated and plotted in the original spatio-temporal space.

MOVIES [12]: The main-memory MOVing objects Indexing using frEquent Snapshots supports predictive queries on moving objects. Instead of creating only one index and modifying it according to incoming updates, MOVIES constructs an index w.r.t. the most updated data of moving objects and uses that index for a short period of time and then throws it away when a new index is constructed. Thus, MOVIES provides a read-optimized index. Query results may not consider the most recent updates if the updates arrive before a new index is scheduled for construction. MOVIES uses linearized kd-tries [31, 44] to index moving objects' locations. It uses Predictive (PI) or Non-Predictive Indexing (NPI). In PI, an object's predicted position is translated immediately to be indexed in MOVIES when an update arrives using the moving object's linear function. In contrast, NPI does not compute predicted positions at indexing time but rather at the query processing time.

4.2 Transformation Methods

Here, the time-space domain is transformed to a space so that it is easier to represent and query data in the future.

4.2.1 Duality transformation

STRIPES [32]: The Scalable Trajectory Index for Predicted Positions in Moving Object Databases models the movements of an object as a linear function in a d-dimensional space. It transforms the predicted positions of

a moving object in the d-space into points in a 2d dual space (storing the coordinates and the velocities of the moving objects). Two distinct indexes are maintained. The first and second indexes cover the time intervals from 0 to L and from L to $2L$, respectively. During every time period L, objects update their locations and velocities by being inserted into the second index and being deleted from the first index. As time elapses, the first index becomes empty and the second index is populated. Each STRIPES index is a disk-based bucket PR quadtree [37], where each dual plane (Vi, Pi) is partitioned equally into four quadrants, with Vi and Pi being the velocity and coordinate values at Dimension i in the original d-space. Because STRIPES may lead to low page utilization, it stores leaf nodes with occupancy less than 50% in half a page and leaf nodes with over 50% occupancy in a full page. Insertion, deletion, and update are as in the PR quadtree. Update tuples contain both the old and new locations, hence an update is a delete of the old location followed by an insert of the new one.

MB-index [14]: The MB-index transforms the objects in d-space to a 2d-space that corresponds to the objects' locations and velocities. One dimension is kept fixed and for each other dimension, (d-1) dimensional hyperplanes that are perpendicular to that dimension partition the space so that the points distribute uniformly among partitions. For each partition, a B-tree indexes the points ordered by the value of the fixed dimension. A range query is transformed into a 2d-polyhedron. Then, each hyperplane of the polyhedron is intersected with each partition to find the point candidates that are then checked for inclusion inside the query polyhedron.

4.2.2 Space-filling-curve (SFC) Transformation

B^x-tree [18]: A B^x-tree extends the B$^+$-tree to index moving objects. Object locations are transformed to 1d values using an SFC that preserves location proximity. The B^x-tree partitions the time axis into equal intervals corresponding to the maximum duration between two updates of any object location. Each interval is then sub-partitioned into equal phases. When one moving object updates its location, the B^x-tree computes the predicted location of the object at the end timestamp of the next phase using the moving object's linear time function. Then, the B^x-tree uses an SFC to convert this predicted location to a 1d-value. This value and the partition information are concatenated and indexed by the B^x-tree. A range query is answered by an iterative query enlargement algorithm to explore appropriate partitions. Deletion is the same as in the B$^+$-tree. The positional information for the object at its insertion and the last insertion time are assumed to be known. Unlike update in the B$^+$-tree, an object in the B^x-tree is only updated when its linear moving function changes. Upon update, the B^x-tree deletes the object out of its tree partition and then reinserts the updated value into the most recent partition. Similar to STRIPES [32], the B^x-tree keeps two partitions at a time.

B^y-tree and α B^y-tree [6]: These two indexes extend the B^x-tree [18] by using separate update frequencies for each moving object. Thus, the B^y-tree and the α B^y-tree work well in the environments with hightly variable update periods. The α B^y-tree uses the parameter α to balance the performance of updates and queries. Instead of updating values every phase (as in the B^x-tree), the α B^y-tree updates values after every α phases.

ST^{2}B-tree [8]: The ST^{2}B-tree indexes the future positions of moving objects in exactly the same way as that of the B^x-tree. However, it improves on the B^x-tree by adapting to data skewness in space and time. The ST^{2}B-tree partitions the space into disjoint Voronoi cells using a set of reference points identified dynamically. Based on the data density within a Voronoi region, each region has an individual grid. A moving object is assigned to one grid cell within a Voronoi region. A tree partition in the ST^{2}B-tree grows and shrinks one after the other over time. The Voronoi regions and their corresponding grids are adjusted given the new data distribution.

B^{dual}-tree [48]: The B^{dual}-tree builds on the B^x-tree. Unlike the B^x-tree that indexes only objects' locations, the B^{dual}-tree captures both d-dimensional locations and velocities in a dual 2d-space. The dual space is partitioned along all dimensions into cells. Then, an SFC transforms the 2d-values in the dual space into 1d-values that are indexed in a B$^+$-tree. Any cell in the partitioning space can be regarded as a d-dimensional moving rectangle (MOR) that captures the locations and velocities of all objects inside it similar to the time parameterized bounding rectangle (TPBR) in the TPR-tree [36]. Each internal entry in the B$^+$-tree maintains a set of MORs, indicating the spatial region and range of velocity covered by the sub-tree of the entry. Unlike the B^x-tree where spatiotemporal search is reduced to several 1D range queries on the B$^+$-tree, the B^{dual}-tree uses

R-tree-like query algorithms as in the TPR-tree. A range query searches the sub-tree of an internal entry only if any of its MORs intersect the query region. Insertion, update and deletion of a moving object in the B^{dual}-tree are similar to those of the B^x-tree.

B^{dH}-tree [38]: The B^{dH}-tree builds on the B^x-tree. Instead of using a B^+-tree, the B^{dH}-Tree uses a B^{link}-tree [41] that is a B^+-tree whose internal nodes at the same level are linked. Unlike the B^x-tree, the B^{dH}-tree transforms data using a dynamic Hilbert SFC whose order can be automatically adjusted according to data distribution. Thus, sub-regions with different data distributions in space have different Hilbert SFC resolutions. Merging subregions lowers their SFC order while splitting a subregion causes the divided subregions to have a higher order. Insertion, deletion and update are the same as those of the B^x-tree.

To sum up, dual transformation techniques suffer from three main drawbacks: (1) The *dual* space does not capture all the information in the *primal* space. (2) There is no guarantee that objects near each other in the *primal* space will still be near each other in the *dual* space. (3) Rectangular range queries in the *primal* space are always transformed into polygonal range queries in the *dual* space, which calls for more complex algorithms.

4.3 Time-parameterized Access Methods

STP-tree [42]: The Spatio-temporal Prediction Tree generalizes the TPR [36] and TPR*-trees [43] by arbitrary polynomial moving functions to index future positions of moving objects. It assumes that each moving object, say o, maintains the most-recent locations and is able to continuously revise its individual motion function, say $o(t)$. While different objects can follow different motion functions, the server indexes the same motion type for all objects. The motion type is a polynomial function of any degree D that approximates the moving function $o(t)$ in the d-dimensional space. The precise motion functions $o(t)$ are used only when the server cannot determine if a candidate qualifies predictive range queries. In contrast to the TPR*-tree, the parameterized MBR in the STP-tree is represented by two polynomials over time starting at two opposite corners of a d-dimensional rectangle enclosing the locations of the children through the maximum timestamp of all children. Insertion, update, and deletion in the STP-tree are similar to those in the TPR*-tree. The nodes' bounding polynomial matrices guide the search for objects, given the objects' polynomials.

ANR-tree [5]: The Adaptive Network R-tree indexes the future trajectories of objects in a constrained network. As in the FNR-tree [16], the ANR-tree is a two-level index. At the top, a 2D R-tree indexes the network. At the bottom level, another 1D R-tree is used to index adaptive units. An adaptive unit extends a 1D MBR of the TPR-tree by grouping adjacent objects with the same direction and similar speed. The predicted movement of an object is not a linear function, but is bounded by the fastest and slowest movements of the object.

5 Access Methods for Past, Present, and Future Spatio-temporal Data

R^{PPF}-tree [34]: The R^{PPF}-tree (Past, Present, and Future) indexes positions of moving objects at all points in time. The past positions of an object between two consecutive samples are linearly interpolated and the future positions are computed via a linear function from the last sample. The R^{PPF}-tree applies partial persistence to the TPR-tree to captures the past positions. Leaf and non-leaf entries of the R^{PPF}-tree include a time interval of validity - [insertion time, deletion time]. When a node, say x, is split at time t, entries in x alive at t are copied to a new node, say y and their timestamps are set to [t, ?) (i.e., their deletion times are unidentified). While a time-parameterized bounding rectangle (TPBR) of the TPR-tree is valid from the current time, the structure of a TPBR in the R^{PPF}-tree is valid from its insertion time. The *straightforward*, *optimized*, and *double* TPBRs are studied. In the straightforward approach, the bounding rectangle is the integral of the TPBR from its insertion time to infinity. In the optimized TPBR, the bounding rectangle is the integral of the TPBR from its insertion time to (current time + H time units in the future that can be efficiently queried). The straightforward and optimized TPBRs cannot be tightened since these rectangles start from their insertion times. The double TPBR allow tightening by having two components: a tail MBR and a head MBR. The tail MBR starts at the time of the last update and extends to infinity and thus is a regular TPBR of the TPR-tree. The head MBR bounds the

finite historical trajectories from the insertion time to the last update. Querying is similar to the regular TPR-tree search with the exception of redefining the intersection function to accommodate for the double TPBR.

PCFI$^+$-index [25]: The Past-Current-Future+-Index builds on SETI [4] and TPR*-tree [43]. As in SETI, space is divided into non-overlapping cells. For each cell, an in-memory TPR*-tree indexes the current positions and velocities of objects. Current data records are organized as a main-memory hash index, hence allowing efficient access to current positions. To index the objects' past trajectories, the PCFI$^+$-index uses a sparse on-disk R*-tree to index the lifetimes of historical data that only contains the segments from one cell. Insertion, update, and deletion are similar to those of SETI and TPR*-tree. Upon update, if the new location resides inside the same partition, a new segment is inserted into the historical data file; the TPR*-tree updates the new location for the object. Otherwise, a split occurs and two segments are inserted into the historical data file at different pages; the corresponding entry in the old TPR*-tree is removed and is inserted into another TPR*-tree. If the insertion of a segment overflows a page, the corresponding R*-tree entry is updated to set its end time.

BB^x-index [22]: The BB^x-index uses the B^x-tree techniques to support the present and future. To index the past, the BB^x-index keeps multiple tree versions. Each tree entry has the form $< x\_rep, tstart, tend, pointer >$, where x_rep is the transformed 1D object location value using an SFC, $tstart$ and $tend$ are the insert and update times of the object, respectively. Each tree corresponds to a timestamp signature being the end timestamp of a phase when the tree is built and a lifespan being the minimum start time and the maximum end time of all the entries in that tree. Unlike the B^x-tree that concatenates the timestamp signature and the 1D transformed value, the BB^x-index maintains a separate tree for each timestamp signature, and models the moving objects from the past to the future. Insertion is the same as in the B^x-tree. Instead of deleting an object, update sets the end time of the object to the current time, followed by an insertion of the updated object into the latest tree.

UTR-tree [11]: The Uncertain Trajectory R-tree extends the MON-tree [10] with uncertainty within a constrained network. A top R-tree indexes directed atomic route sections (ARSs) that connect two junctions of a route and do not contain any other junctions from which moving objects can exit the traffic flow. Leaf entries that corresponds to the ARSs of the same route have a pointer to a bottom R-tree of the route. The bottom R-trees of the UTR-tree keep the latest locations and trajectories of moving objects, and thus support historical, present, and near-future queries. Between two exact locations of the moving object at two consecutive update times, the UTR-tree derives the uncertain trajectory of the object and indexes it in the corresponding bottom R-tree.

STCB$^+$-tree [23]: The Spatio-temporal Compressed B$^+$-tree uses multiple Compressed B$^+$-trees [24] (a CB$^+$-tree is a B$^+$-tree with at least 75%-full leaf nodes) to index trajectories of moving objects. One CB$^+$-tree, termed the TCB$^+$-tree, indexes the temporal dimension and for each spatial dimension, one CB$^+$-tree, termed the SCB$^+$-tree, indexes the spatial attributes of moving objects in that dimension. The temporal and spatial coordinates of endpoints of all trajectory segments are indexed by the CB$^+$-trees. Insertion and deletion in the CB$^+$-tree are the same as in the B$^+$-tree. In the TCB$^+$-tree, each interval identified by two consecutive key values stores a bucket including the identifiers of the objects moving in that interval. The generation of new temporal data is totally ordered. Thus, insertions into the TCB$^+$-tree append new entries to the rightmost leaf. The SCB$^+$-tree distinguishes the onward, backward, and still motions of a trajectory according to the departure and arrival positions of the trajectory. Similar to the TCB$^+$-tree, for each interval identified by two consecutive key values, the SCB$^+$-tree keeps a bucket including the identifiers and locations of objects moving within the interval. To answer a spatio-temporal query, the TCB$^+$- and the SCB$^+$-trees are searched for the temporal and spatial output, respectively; the final output is the object identifiers common in all the outputs.

PPFI [15]: The PPFI uses a 2D R*-tree to index a road network. Each road sector has a 1D R*-tree (Rs) that index the time interval for the past trajectory of objects moving along the sector. Pointers link the leaf entries created by the consecutive updates of the same object. These leaf entries can be in the same Rs or in different Rs's. A hash structure describes the recent state of moving objects and predicts their near-future positions using the objects' linear moving functions. To insert a new moving object into the PPFI, two entries for the object are inserted into (1) the hash and (2) the leaf node of the corresponding Rs. These two entries point to each other. Using this hybrid data structure, the PPFI can support trajectory-based and predictive queries.

6 Conclusion

This short survey is a continuation of [28]. It classifies and overviews existing spatio-temporal access methods for the years 2003-2010. Although not the scope of this survey, it is important to highlight the works in [2, 7, 13] that provide spatio-temporal data generators and benchmarks to generate datasets for both general or road network environments and facilitate the evaluation of spatio-temporal access methods under various conditions.

References

[1] D. H. Ballard. Strip trees: a hierarchical representation for curves. *Commun. ACM*, 24(5):310–321, 1981.

[2] T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2):153–180, 2002.

[3] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. *SIGMOD*, pp. 599–610, 2004.

[4] V. Chakka, A. Everspaugh, and J. Patel. Indexing large trajectory data sets with SETI. In *CIDR*, 2003.

[5] J.-D. Chen and X.-F. Meng. Indexing future trajectories of moving objects in a constrained network. *J. Comput. Sci. Technol.*, 22(2):245–251, 2007.

[6] N. Chen, L.-D. Shou, G. Chen, and J.-X. Dong. Adaptive indexing of moving objects with highly variable update frequencies. *J. Comput. Sci. Technol.*, 23(6):998–1014, 2008.

[7] S. Chen, C. Jensen, and D. Lin. A benchmark for evaluating moving object indexes. *PVLDB*, 1(2):1574–1585, 2008.

[8] S. Chen, B. Ooi, K. Tan, and M. Nascimento. ST^{2}B-tree: A self-tunable spatio-temporal B$^+$-tree index for moving objects. In *SIGMOD*, pages 29–42, 2008.

[9] J. Clifford, C. Dyreson, T. Isakowitz, C. Jensen, and R. Snodgrass. On the semantics of "Now" in databases. *ACM TODS*, 22(2), 1997.

[10] V. T. De Almeida and R. H. Güting. Indexing the trajectories of moving objects in networks. *Geoinformatica*, 9(1):33–60, 2005.

[11] Z. Ding. UTR-tree: An index structure for the full uncertain trajectories of network-constrained moving objects. In *MDM*, pages 33–40, 2008.

[12] J. Dittrich, L. Blunschi, and M. A. Vaz Salles. Indexing moving objects using short-lived throwaway indexes. In *SSTD*, pages 189–207, 2009.

[13] C. Düntgen, T. Behr, and R. H. Güting. BerlinMOD: A benchmark for moving object databases. *The VLDB Journal*, 18(6):1335–1368, 2009.

[14] K. M. Elbassioni, A. Elmasry, and I. Kamel. An efficient indexing scheme for multi-dimensional moving objects. In *ICDT*, pages 422–436, 2003.

[15] Y. Fang, J. Cao, Y. Peng, and L. Wang. Indexing the past, present and future positions of moving objects on fixed networks. In *CSSE (4)*, pages 524–527, 2008.

[16] E. Frentzos. Indexing objects moving on fixed networks. In *SSTD*, pages 289–305, 2003.

[17] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, June 1984.

[18] C. Jensen, D. Lin, and B. Ooi. Query and update efficient B$^+$-tree based indexing of moving objects. In *VLDB*, 2004.

[19] C. Jensen, H. Lu, and B. Yang. Indexing the trajectories of moving objects in symbolic indoor space. In *SSTD*, 2009.

[20] K.-S. Kim, S.-W. Kim, T.-W. Kim, and K.-J. Li. Fast indexing and updating method for moving objects on road networks. *Web Information Systems Engineering Workshops*, 0:34–42, 2003.

[21] T. T. T. Le and B. G. Nickerson. Efficient search of moving objects on a planar graph. In *GIS*, pages 1–4, 2008.

[22] D. Lin, C. Jensen, B. Ooi, and S. Šaltenis. Efficient indexing of the historical, present, and future positions of moving objects. In *MDM*, pages 59–66, 2005.

[23] H. Lin. Indexing the trajectories of moving objects. In *Intl. MultiConf. of Engrs. and Computer Scientists*, pages 732–737, 2009.

[24] H. Lin, R. Chen, and S. Chen. Enhancement of data aggregation using a novel point access method. *WSEAS Transactions on Computers*, 7(12):2001–2010, 2008.

[25] Z.-H. Liu, X.-L. Liu, J.-W. Ge, and H.-Y. Bae. Indexing large moving objects from past to future with PCFI+-index. In *COMAD*, pages 131–137, 2005.

[26] D. Lomet and B. Salzberg. Access methods for multiversion data. In *SIGMOD*, pages 315–324, May 1989.

[27] J. Mason and D. C. Handscomb. *Chebyshev polynomials*. Chapman and Hall, 2003.

[28] M. Mokbel, T. Ghanem, and W. G. Aref. Spatio-temporal access methods. *IEEE Data Eng. Bull.*, 26(2):40–49, 2003.

[29] J. Ni and C. V. Ravishankar. PA-tree: A parametric indexing scheme for spatio-temporal trajectories. In *SSTD*, 2005.

[30] J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *TODS*, 9:38–71, 1984.

[31] J. Orenstein and T. Merrett. A class of data structures for associative searching. In *PODS*, pages 181–190, 1984.

[32] J. Patel, Y. Chen, and V. Chakka. STRIPES: An efficient index for predicted trajectories. *SIGMOD*, 2004.

[33] K. Patroumpas and T. Sellis. Monitoring orientation of moving objects around focal points. *SSTD*, pp 228-246, 2009.

[34] M. Pelanis, S. Šaltenis, and C. Jensen. Indexing the past, present, and anticipated future positions of moving objects. *TODS*, 31(1):255–298, 2006.

[35] D. Pfoser, C. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, pages 395–406, 2000.

[36] S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.

[37] H. Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.

[38] D. Seo, S. Song, Y. Park, J. Yoo, and M. Kim. B^{dH}-tree: A B^{+}-tree based indexing method for very frequent updates of moving objects. *Intl. Symp. on Computer Sc. and its Applications*, 0:314–319, 2008.

[39] Y. N. Silva, X. Xiong, and W. G. Aref. The RUM-tree: Supporting frequent updates in R-trees using memos. *VLDB J.*, 18(3):719–738, 2009.

[40] Z. Song and N. Roussopoulos. SEB-tree: An approach to index continuously moving objects. In *MDM*, 2003.

[41] V. Srinivasan and M. Carey. Performance of B-tree concurrency control algorithms. *SIGMOD Rec.*, 20(2):416–425, 1991.

[42] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In *SIGMOD*, pages 611–622, 2004.

[43] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, 2003.

[44] H. Tropf and H. Herzog. Multidimensional range search in dynamically balanced trees. *Angewandte Informatik*, 23(2):71–77, 1981.

[45] L. Wang, Y. Zheng, X. Xie, and W.-Y. Ma. A flexible spatio-temporal indexing scheme for large-scale GPS track retrieval. In *MDM*, pages 1–8, 2008.

[46] X. Xiong and W. G. Aref. R-trees with update memos. In *ICDE*, page 22, 2006.

[47] X. Xiong, M. Mokbel, and W. G. Aref. LUGrid: Update-tolerant grid-based indexing for moving objects. In *MDM*, page 13, 2006.

[48] M. Yiu, Y. Tao, and N. Mamoulis. The B^{dual}-tree: Indexing moving objects by space filling curves in the dual space. *VLDB J.*, 17(3):379–400, 2008.

[49] P. Zhou, D. Zhang, B. Salzberg, G. Cooperman, and G. Kollios. Close pair queries in moving object databases. In *GIS*, pages 2–11, 2005.