Bulletin of the Technical Committee on

Data Engineering

June 2002 Vol. 25 No. 2

IEEE Computer Society

Letters

Special Issue on Indexing of Moving Objects

Conference and Journal Notices

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

The Data Engineering Bulletin web page is http://www.research.microsoft.com/research/db/debull.

Letter from the Editor-in-Chief

Updating Bulletin Publishing Infrastructure

Over nine years ago, I became editor-in-chief of the Data Engineering Bulletin. My first issue as editor was in December, 1992. At that time the decision was made to not only publish the Bulletin in hardcopy as had been done previously, but to also make it available electronically. This was originally done via ftp. A few years later, web access was added. And a few years after that, the Bulletin became entirely electronically distributed, mostly via the use of web access. In 1998, the web access was enhanced. The table of contents for each issue was presented in a web page where clicking on an article downloaded exactly that article to the reader.

Despite the changes cited above, the original electronic publication infrastructure used to create each issue remained virtually unchanged since 1992. In 1992, I prevailed upon Mark Tuttle, currently at Compaq Cambridge Research Lab, to produce style files that would enable me to assemble the bulletin from separately submitted articles in latex, automatically generating front and back covers, and putting the issue's table of contents on the front cover. This infrastructure has served the Bulletin well, creating a very readable and very attractive publication.

But time moves on. And so did latex. The version upon which the original style files were based, Latex 2.09, is no longer current. Authors using the current version, Latex 2e, could not check out their articles in the form that would actually appear in the published issue. Neither could issue editors reliably assemble the issue, and check for problems. So the style files needed to be updated to work with Latex 2e.

So, at the risk of "going to the well" once too often, I again asked Mark Tuttle to help me out. Happily, Mark rose to the challenge. His new and enhanced style files were tested using the March issue, though not used for the version that was actually published since the enhancements were made after March. The new style files succeeded on that issue. Thus, the current issue (June) has been assembled entirely using the new style files that Mark has created for us. This letter is the most manifest recognition that Mark will receive for his hard work. So let me say a very special **"thank you"** to Mark Tuttle for a job very well done!

The Current Issue

Mobile computing is one of the great new application areas that have been made possible by the relentless advance of hardware technology. And one facet of this is what I have heard referred to as "location aware" computing. By whatever name one refers to this, a substantial part of what is involved is keeping track of where people, cars, and various moving objects are at any point in time. And here, fast access to information about object location is critical since an object may not stay for long in one area, and what might be enormously important is providing timely access to information that is relevant to an object's current location.

Thus mobile computing relies heavily on the indexing of moving objects. And that is the subject of the current issue. Christian Jensen, the issue editor is a great choice for handling this issue exactly because he is himself actively engaged in research on indexing moving objects. He knows the area well, and further, he knows well the work of others in this exciting new area. Christian has brought together work from both universities and industry. As I have stressed many times before, I think one special role of the Bulletin is to provide a window on what is going on in industry as there are few conventional outlets for learning about industry work. So I want to thank Christian for a fine job in bringing us the June issue on one of the important new directions in database technology.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

We don't sort twice as fast every 18 months on the same computer. However, processors have on average improved at about this rate for some four decades—a phenomenon known as Moore's Law, although it is not a law, but is better characterized as a self-fulfilling prophecy. Among other kinds of hardware close to the heart of a database researcher, disks and networks improve at even faster rates than do processors. These and many other advances in hardware are important drivers for research in software, including data management research.

Highly portable and unobtrusive, wirelessly on-line electronics with sensing capabilities are appearing on the horizon. This has brought increasing interest to areas such as ubiquitous and pervasive computing, mobile computing, assistive computing, spatio-temporal databases, sensor-data management, stream data processing, augmented reality, and ambient intelligence.

In step with applications and services being delivered increasingly to their users via mobile computing devices, instead of via desk-top computers, new kinds of services with new characteristics become of interest. Context awareness is particularly important to mobile users who find themselves in a range of different situations, characterized by diverse, specific needs. Location, the capture of which is made possible through increasingly sophisticated positioning technologies, is an essential aspect of context awareness.

The specific focus of the present issue of the Data Engineering Bulletin is on a range of aspects of the indexing of the positions of continuously moving objects.

The positions of moving objects are obtained via some form of sampling, and the past positions of a point object moving in d-dimensional space are frequently represented as a polyline, a sequence of line segments in $d + 1$-dimensional space connecting the time-referenced, sampled positions. In the first paper, Dieter Pfoser considers the problem of indexing such trajectories. Focusing also on information relating to the past, Papadias et al. in the next paper consider indexing in the context of aggregate computation for moving objects.

A somewhat different problem is that of indexing the positions of moving objects from the times their positions were last sampled and into the future. Here, it is typical to represent the positions of an object by a linear function. Papadopoulos et al. study the use of duality transforms for the indexing of such positions of objects moving in one-, so-called 1.5-, and two-dimensional spaces. Agarwal and Procopiuc cover both problems, surveying moving-object indexing from a computational geometry perspective and thus broadening this issue to also cover highly relevant results not published in typical database outlets. Šaltenis and Jensen point the attention to the need for faster update processing in indices for the current positions of moving objects.

Chon et al. proceed to consider the management of moving-object trajectories. They adopt a partitioning-based technique in place of indexing. Next, when attempting to develop a practical index with high query and update performance, empirical performance studies are of essence. Myllymaki and Kaufman describe a testbed for dynamic spatial indexing, an important piece of infrastructure in this regard. Finally, Kothuri and Ravada explore the support in the Oracle DBMS for spatio-temporal indexing.

This issue samples research results and also points to challenges in an area with many open problems. It is my hope that the issue offers a good feel for the breadth of fundamental problems and challenges inherent to moving-object indexing, and for what it entails to conduct research in this area. I also hope that the issue will inspire new research on moving-object indexing.

Christian S. Jensen
Department of Computer Science
Aalborg University, Denmark

Indexing the Trajectories of Moving Objects

Dieter Pfoser
Computer Technology Institute
11851 Athens, Hellas
pfoser@cti.gr

Abstract

Spatiotemporal applications offer a treasure trove of new types of data and queries. In this work, the focus is on a spatiotemporal sub-domain, namely the trajectories of moving point objects. We examine the issues posed by this type of data with respect to indexing and point out existing approaches and research directions. An important aspect of movement is the scenario in which it occurs. Three different scenarios, namely unconstrained movement, constrained movement, and movement in networks are used to categorize various indexing approaches. Each of these scenarios gives us different means to either simplify indexing or to improve the overall query processing performance.

1 Introduction

Several application areas contribute to the growing number of different types of spatiotemporal data. For example, we are currently experiencing rapid technological developments that promise widespread use of on-line mobile personal information appliances [12]. Mobility is a concern to many applications and services. One aspect of mobility is movement and thus the change of location. Applications in this context include emerging ones such as location-based services, but also "classical" ones such as fleet management and the optimal spatiotemporal distribution of vehicles [1].

Applications such as these warrant the study of indexing mobile objects. In particular, our interest is in *recording the movements of mobile objects, i.e., their trajectories, and indexing those for post-processing (e.g., data mining) purposes*. Thus, we will not concern ourselves with the indexing of the current positions and the predicted movements of objects such as treated in, e.g., [14, 15]. The size and shape of an object is often fixed and of little importance-only its position matters. Thus, the problem becomes one of recording the position of a moving object across time. The movement of an object may then be represented by a trajectory in the three dimensional space composed of two spatial dimensions and one time dimension [9].

Typically, access methods are developed having only the data and the queries in mind. However, for trajectory data we can also consider the constraints that the objects in their movement are subjected to. Specifically, we may distinguish among three movement scenarios, namely unconstrained movement (e.g., vessels at sea), constrained movement (e.g., pedestrians), and movement in transportation networks (e.g., trains and, typically, cars). As we will see in this work, the latter two scenarios allow us to optimize query processing techniques or to use simpler access methods to index the data.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

The outline of this paper is as follows. Section 2 defines the basic concepts and indentifies the challenges in indexing trajectories. Sections 3, 4, and 5 point out indexing approaches grouped according to the three movement scenarios. Finally, Section 6 concludes and points to research directions.

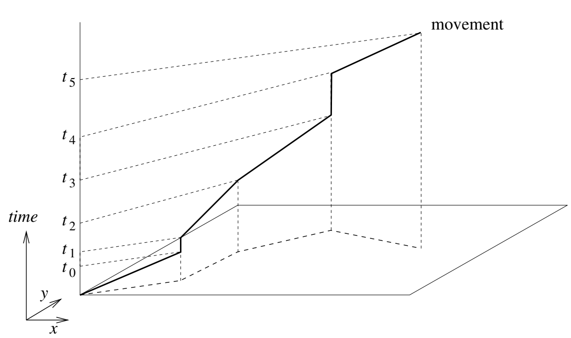
2 On Trajectories and Queries

The central question in this work is how we can ease the task of indexing point-object movements. This section gives a brief description of the data, the related queries, and the respective indexing challenges.

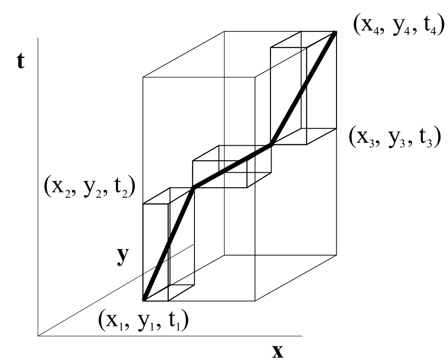
2.1 Trajectories

A trajectory is the data we obtain by recording the position of a moving point object across time. Consider the following application context. Optimizing transportation, especially in highly populated and thus congested areas, is a very challenging task that may be supported by an information system. A core application in this context is fleet management [1]. Vehicles equipped with GPS receivers transmit their positions to a central computer using either radio communication links or mobile phones. At the central site, the data is processed and utilized.

To record the movement of an object, we would have to know its position at all times, i.e., on a continuous basis. However, GPS and telecommunications technologies only allow us to sample an object's position, i.e., to obtain the position at discrete instances of time, such as every few seconds. A first approach to represent the movements of objects would be to simply store the position samples. This would mean that we could not answer queries about the objects' movements at times in-between those of the sampled positions. Rather, to obtain the entire movement, we have to interpolate. The simplest approach is to use linear interpolation, as opposed to other methods such as polynomial splines. The sampled positions then become the endpoints of line segments of polylines, and the movement of an object is represented by an entire polyline in 3D space. The solid line in Figure 1(a) represents the movement of an object. Space and time coordinates are combined to form a single coordinate system. The dashed line shows the projection of the movement into the 2D (spatial) plane [8, 9].



(a)



(b)

Figure 1: Trajectories: (a) 2+1 Dimensional Space, (b) Approximating Trajectories Using MBBs

In classical spatial databases only position information is available. In our case, however, we have also derived information, e.g., speed, acceleration, traveled distance, etc. Information is derived from the combination of spatial and temporal data. Further, we do not just index collections of line segments—these are parts of larger, semantically meaningful objects, namely trajectories. These semantic properties of the data are reflected in the types of queries of interest for the data.

Query Type		Operation	Signature
Coordinate-Based Queries		overlap, inside, etc.	range
Trajectory -Based Queries	Topological Queries	enter, leave, cross, bypass	range $\times$ {segments} $\rightarrow$ {segments}
	Navigational Queries	traveled distance, covered area (top or average), speed, heading, parked	{segments} $\rightarrow$ int {segments} $\rightarrow$ real {segments} $\rightarrow$ bool

Table 1: Types of Spatiotemporal Queries

2.2 Queries

The queries of interest for trajectories comprise adapted spatial queries, e.g., range queries of the form “find all objects within a given area at some time during a given time interval” and queries with no spatial counterparts. Here, the so-called trajectory-based queries are classified in *topological* queries, which involve the entire movement of an object (enter, leave, cross, and bypass), and *navigational* queries, which involve derived information, such as speed and heading. Table 1 summarizes the query types. The “Operation” column lists the operations used for several query types, and the “Signature” column presents the types involved. For example, a coordinate-based query uses the inside operation to determine the segments within the specified range. The notation {segments} simply refers to a set, it does not capture whether this set constitutes one or more trajectories. It is important to be able to extract information related to (partial) trajectories, e.g., “What were the trajectories of objects after they left Tucson between 7 a.m. and 8 a.m. today, in the next hour?” This type of query is referred to as *combined* query, since we first have to select the respective trajectories (“...left Tucson between 7 a.m. and 8 a.m. today...”) and subsequently the respective portions (“...in the next hour?”). More details on trajectory-related queries can be found in [8, 11].

2.3 Indexing Fundamentals

Trajectories are three-dimensional and can be indexed using spatial access methods. However, there are difficulties. Trajectories are decomposed into their constituent line segments, which are then indexed. The use of the R-tree [2] is illustrated in Figure 1(b). The R-tree approximates the data objects by Minimum Bounding Boxes (MBBs), here three-dimensional intervals. Approximating segments using MBBs proves to be inefficient. Figure 1(b) shows that we introduce large amounts of “dead space,” meaning that the MBBs cover large parts of space, whereas the actual space occupied by the trajectory is small. This leads to high overlap and consequently to a small discrimination capability of the index structure.

Other trajectory indexing problems include trajectory preservation and skewed data growth. As for the first problem, spatial indices tend to group segments into nodes according to spatial proximity. However, in the case of trajectories, it is beneficial to some queries if the index preserves trajectories, i.e., to group segments according to their trajectory and only then according to proximity (cf. [11]). The second problem refers to the fact that trajectory data grows mostly in the temporal dimension. The spatial dimensions are fixed, e.g., the city limits. Exploiting this property of the data can further increase query performance.

2.4 Movement Scenarios

Depending on the particular objects and applications under consideration, the movements of objects may be subject to constraints. Specifically, we may distinguish among three movement scenarios, namely unconstrained movement (e.g., vessels at sea), constrained movement (e.g., pedestrians), and movement in transportation networks (e.g., trains and, typically, cars). Unconstrained movement is the scenario mostly asserted in work on spatiotemporal access methods. However, it hardly represents reality. Constrained movement and movement in networks represent similar movement scenarios. The former assumes that there exist spatial objects that constrain the movement, e.g., when considering the movement of cars, houses, lakes, parks, etc. Movement in networks is then merely an abstraction of constrained movement. Here, one is only interested in positions of objects with respect to the network and not with respect to a two-dimensional reference system. For example, we may expect that many applications will be interested only in the positions of cars with respect to the road network, rather than in their absolute coordinates. The movement effectively occurs in a different space than for the first two scenarios.

Section 2.3 illustrated the indexing challenges related to trajectories. In exploiting auxiliary information such as infrastructure and networks, we can improve query performance and simplify indexing. The following three sections explore indexing and query processing approaches in relation to the three movement scenarios.

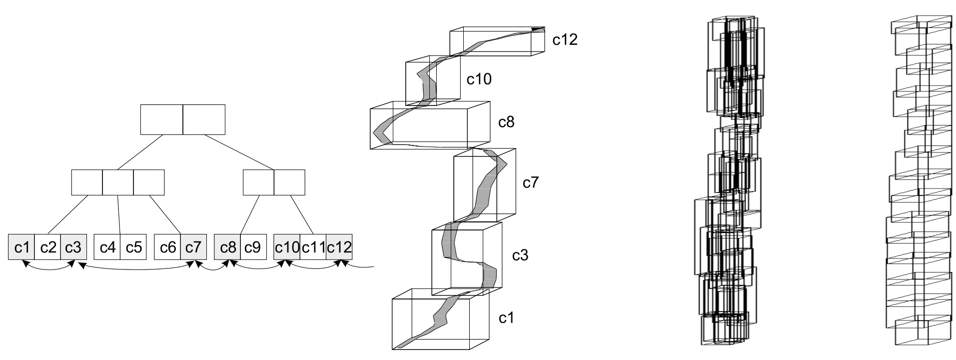
3 On Indexing Unconstrained Movement

Unconstrained movement is conceptually the simplest case for trajectory indexing. In the following, we describe several access methods that are tailored to the requirements of the data and the queries.

3.1 The TB-tree

The TB-tree [11] is an access method that considers the particularities of the data, namely trajectory preservation and temporal growth, and aims at efficiently processing related types of queries.

An underlying assumption for the R-tree is that all inserted geometries are independent. In our context this translates to all line segments being independent. However, line segments are part of trajectories and the R-tree only implicitly maintains this knowledge. With the TB-tree, we aim for an access method that strictly preserves trajectories. As a consequence, each leaf node in the index should only contain segments belonging to one trajectory. Thus, the index becomes actually a *trajectory bundle*. This approach is only feasible by making some concessions to the most important R-tree property, namely node overlap, or spatial discrimination. As a drawback, spatially close line segments from different trajectories will be stored in different nodes. This, in turn, increases the node overlap, decreases the space discrimination, and, thus, increases the classical range query cost. Thus, the TB-tree trades space discrimination for trajectory preservation.



(a) (b) (c)

Figure 2: TB-Tree: (a) Structure, Non-Leaf Level MBB Structure (B) R-Tree and (c) TB-Tree

Employing the above constraints, the TB-tree structure consists of a set of leaf nodes, each containing a partial trajectory, organized in a tree hierarchy. For query processing, it is necessary to be able to retrieve segments based on their trajectory identifier. Thus, leaf nodes containing segments of the same trajectory are linked together by a doubly-linked list data structure. This preserves trajectory evolution and overall improves especially the performance of trajectory-based and combined queries. Figure 2(a) shows a part of a TB-tree structure and a trajectory illustrating the overall data structure. For clarity, the trajectory is drawn as a band rather then a line. The trajectory symbolized by the grey band is fragmented across six nodes, c1, c3, etc. In the TB-tree these leaf nodes are connected through the doubly-linked list.

Figures 2(b) and (c) give an impression of the quality of the non-leaf level arrangement of bounding boxes in the R- and the TB-tree, respectively. The R-tree grouping is dominated by purely spatial characteristics such as proximity, ignoring the temporal growth of the data. The TB-tree structure is dominated by trajectory preservation. Since the data is inserted with a growing time horizon, the MBBs exhibit a temporal clustering.

The TB-tree structure is not the only approach to the indexing of trajectories. The following section gives a brief overview of what other techniques exist.

3.2 Other Approaches

Nasciemento et al. [7] adopt the trajectory model as described in Section 2.1 and investigate the suitability of multidimensional access methods for trajectory data. They compare the performance of various R-tree versions for different range-query loads.

Hadjileftheriou et al. [6] define an approach to reduce the dead space introduced by MBB approximations of trajectories (cf. Section 2.3) by introducing "artificial object updates." They effectively manipulate the partitioning of a trajectory into segments. A partially persistent tree structure [4, 16] is used to index the data. This approach generalizes previous work [3] in which it was assumed that the objects move with a linear function of time, whereas in [6] more complex functions are permitted.

Porkaew et al. [13, 5] examine the indexing of trajectories in native space (cf. Section 2.1) vs. parametric space. In parametric space, segments of trajectories are represented in terms of a location and a motion vector. In their experimental evaluation, the authors use an R-tree structure as an index for both representations.

Other works further propose access methods that go beyond trajectories towards the indexing of moving spatial shapes in general. These approaches however do not always consider continuous but only discrete changes, e.g., Tzouramanis et al. [17] employ overlapping linear quad-trees to index discretely changing spatial objects.

Tao and Papadias [16] propose a method called MV3R-tree to index the past locations of moving shapes. Their method combines multi-version B-trees and 3D R-trees to process timestamp and interval queries.

4 On Constrained Movement

We now assert that the movement of objects is constrained by elements termed *infrastructure*.

In terms of data, infrastructure represents "black-out" areas for movement and, thus, there are no objects and trajectories where there are infrastructure elements. However, in the index, approximations of the data are used, which introduce dead space. Consequently, the areas covered by infrastructure are not empty. This will incur unnecessary search in the index as well as produce a certain number of falsely reported answers, which must subsequently be eliminated. Both lead to extra I/O operations.

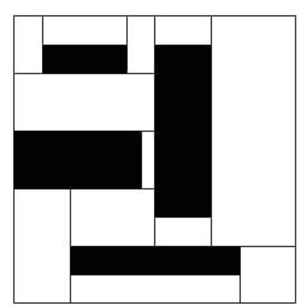


Figure 3: A Query Window Segmentation Example

To eliminate this extra I/O, we can use infrastructure in a pre-processing step, the idea being to not look for objects where there cannot be any? The strategy we choose is to query the infrastructure to save on querying the trajectory data. Overall, this will turn out to be favorable, since the number of infrastructure elements can be assumed to be very small compared to the trajectory data. Further, the trajectory data is growing with time, whereas the infrastructure data remains more or less constant.

The general principle is to decompose a given query window based on the infrastructure contained in it. The intuition is to segment the parts of the query window not occupied by infrastructure into well-shaped rectangles, i.e., as square as possible. In Figure 3, few but large infrastructure elements are shown as black rectangles, the possible outcome of such a segmentation process is shown as white rectangles.

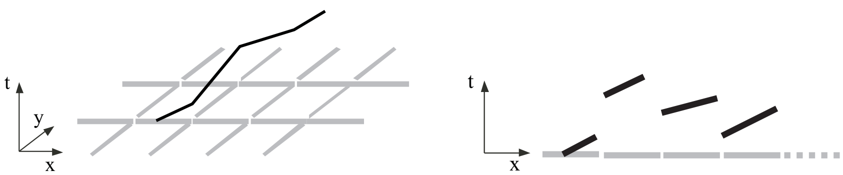
The query windows resulting from this segmentation (instead of the large query window ranging over infrastructure) are subsequently used to query the trajectory data index. In [10] the conditions under which this approach is favorable are established.

5 On Movement in Networks

In many applications, movement occurs in a network. When dealing with network-constrained movement, one is not interested in spatial extents, e.g., the thickness of the road, or the absolute position of the object in terms of its (x, y)-coordinates, but rather in relative positions with respect to the network, e.g., kilometer 21 of Highway 101.

The space defined by a network is quite different from the Euclidean space that the network is embedded into. Intuitively, the dimensionality of the networked-constrained space is lower than that of the space it is embedded in. In the literature, the term 1.5 dimensional has been used.

Modeling movement with respect to a network simplifies the trajectory data obtained. The two spatial dimensions are essentially reduced to one. Figure 4 illustrates this principle by showing the same trajectory in a three-dimensional and a two-dimensional space. A two-dimensional network is reduced to a set of one-dimensional segments, and the trajectory data is mapped accordingly.



(a) (b)

Figure 4: Network Movement: Trajectories in (a) 3D and (b) 2D Space

Lowering the dimensionality of the data reduces the indexing challenge considerably. Off-the-shelf database management systems typically do not offer three-dimensional indexes and thus do not contend with trajectories. Although it is desirable to design new access methods for new types of data, it may not be attractive in the short or medium term. For example, it took a dozen years before the R-tree found its way into some commercial database products. Depending on the type of data, it can be beneficial to use existing access methods by transforming the data. Considering movement in a network is such a transformation. We can use a simple access method such as a two-dimensional R-tree and this, in turn, allows for an easy integration of the new type of data into commercial database management systems.

6 Conclusions and Future Work

Spatiotemporal data is emerging from a broad range of applications. In this work, we present selected methods for the indexing of trajectories, a type of data that stems from recording the movement of point objects. The existing approaches are grouped according to *three movement scenarios*, constrained movement, unconstrained movement, and movement in networks. Each of these scenarios can aid the processing of spatiotemporal queries in different ways. Unconstrained movement is the typical showcase for the definition of new access methods. Constrained movement allows us to reduce the extent of query windows. Movement in networks reduces the dimensionality of the data and thus of the index.

Directions for future work can either be to define more efficient and/or more specialized access methods, or to satisfy existing needs arising from real applications [1]. This can be achieved by trying to handle spatiotemporal data with our current knowledge in connection with the means available from off-the-shelf database management systems [18].

Acknowledgements

The author would like to thank his co-authors on this subject, Christian S. Jensen and Yannis Theodoridis. Part of this research was supported by the CHOROCHRONOS project, funded by the European Commission DG XII, contract no. ERBFMRX-CT96-0056, and the Nykredit Corporation.

References

[1] V. Delis, D. Pfoser, Y. Theodoridis, and N. Tryfona. Movement mining in fleet management applications. Project Proposal, General Secretariat of Science and Technology, Athens, Hellas, 2001.

[2] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pp. 47–57, 1984.

[3] G. Kollios, D. Gunopulos, V. Tsotras, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatio-temporal access methods. *IEEE TKDE*, 13(5):758–777, 2001.

[4] A. Kumar, V. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE TKDE*, 10(1):1–20, 1998.

[5] I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic queries over mobile objects. In *Proc. EDBT*, pp. 269–286, 2002.

[6] Hadjieleftheriou M, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proc. EDBT*, pp. 251–268, 2002.

[7] M. Nascimento, J. R. O. Silva, and Y. Theodoridis. Evaluation of access structures for discretely moving points. In *Proc. of the International Workshop on Spatio-Temporal Database Management*, pp. 171–188, 1999.

[8] D. Pfoser. *Issues in the Management of Moving Point Objects*. Ph.D. thesis, Department of Computer Science, Aalborg University, Denmark, 2000.

[9] D. Pfoser and C. S. Jensen. Capturing the uncertainty of moving-object representations. In *Proc. of the 6th International Symposium on Spatial Databases*, pp. 111–132, 1999.

[10] D. Pfoser and C. S. Jensen. Querying the trajectories of on-line mobile objects. In *Proc. of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, pp. 66–73, 2001.

[11] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *Proc. VLDB*, pp. 395–406, 2000.

[12] E. Pitoura. DB-Globe: an IST-FET research project on global computing. European Commission DG XII, contract no. IST-2001-32645, http://softsys.cs.uoi.gr/dbglobe/, 2002.

[13] K. Porkaew, I. Lazaridis, and S. Mehrotra. Querying mobile objects in spatio-temporal databases. In *Proc. of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pp. 55–78, 2001.

[14] S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *Proc. ICDE*, pp. 463–472, 2002.

[15] S. Saltenis, C. S. Jensen, S. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. ACM SIGMOD*, pp. 331–342, 2000.

[16] Y. Tao and D. Papadias. Mv3R-tree: a spatiotemporal access method for timestamp and interval queries. In *Proc. VLDB*, pp. 431–440, 2001.

[17] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees and spatio-temporal query processing. *The Computer Journal*, 43(4):325–343, 2000.

[18] M. Vazirgiannis and O. Wolfson. A spatiotemporal model and language for moving objects on road networks. In *Proc. of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pp. 20–35, 2001.

Indexing and Retrieval of Historical Aggregate Information about Moving Objects*

Dimitris Papadias[†], Yufei Tao[†], Jun Zhang[†], Nikos Mamoulis[§], Qiongmao Shen[†], and Jimeng Sun[†]

[†]Department of Computer Science
Hong Kong University of Science and Technology
{dimitris, taoyf, zhangjun, qmshen,jimeng}@cs.ust.hk

[§]Department of Comp. Science and Inf. Systems
University of Hong Kong
nikos@csis.hku.hk

Abstract

Spatio-temporal databases store information about the positions of individual objects over time. In many applications however, such as traffic supervision or mobile communication systems, only summarized data, like the average number of cars in an area for a specific period, or phones serviced by a cell each day, is required. Although this information can be obtained from operational databases, its computation is expensive, rendering online processing inapplicable. A vital solution is the construction of a spatiotemporal data warehouse. In this paper, we describe a framework for supporting OLAP operations over spatiotemporal data. We argue that the spatial and temporal dimensions should be modeled as a combined dimension on the data cube and present data structures, which integrate spatiotemporal indexing with pre-aggregation. While the well-known materialization techniques require a-priori knowledge of the grouping hierarchy, we develop methods that utilize the proposed structures for efficient execution of ad-hoc group-bys. Our techniques can be used for both static and dynamic dimensions.

1 Introduction

The motivation of this work is that many (if not most) current applications require summarized spatio-temporal data, rather than information about the locations of individual points in time. As an example, traffic supervision systems need the number of cars in an area of interest, rather than their ids. Similarly mobile phone companies use the number of users serviced by individual cells in order to identify trends and prevent potential network congestion. Other spatio-temporal applications are by default based on arithmetic data rather than object locations. As an example consider a pollution monitoring system. The readings from several sensors are fed into a database which arranges them in regions of similar or identical values. These regions should then be indexed for the efficient processing of queries such as "find the areas near the center with the highest pollution levels yesterday".

The potentially huge amount of data involved in the above applications calls for pre-aggregation of results. In direct analogy with relational databases, efficient OLAP operations require materialization of summarized data. The motivation is even more urgent for spatio-temporal databases due to several reasons. First, in some

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

* This work was supported by grants HKUST 6081/01E and HKUST 6070/00E from Hong Kong RGC.

cases, data about individual objects should not be stored due to legal issues. For instance, keeping the locations of mobile phone users through history may violate their privacy. Second, the actual data may not be important as in the traffic supervision system discussed. Third, although the actual data may be highly volatile and involve extreme space requirements, the summarized data are less voluminous and may remain rather constant for long intervals, thus requiring considerably less space for storage. In other words, although the number of moving cars (or mobile users) in some city area during the peak hours is high, the aggregated data may not change significantly since the number of cars (users) entering is similar to that exiting the area. This is especially true if only approximate information is kept, i.e., instead of the precise number we store values to denote ranges such as high, medium and low traffic.

Throughout the paper we assume that the spatial dimension at the finest granularity consists of a set of regions (e.g., road segments in traffic supervision systems, areas covered by cells in mobile communication systems etc.). The raw data provide the set of objects that fall in each region every timestamp (e.g., cars in a road segment, users serviced by a cell). Queries ask for aggregate data over regions that satisfy some spatio-temporal condition. A fact that differentiates spatio-temporal, from traditional OLAP is the lack of predefined hierarchies (e.g., product types). These hierarchies are taken into account during the design of the system so that queries of the form "find the average sales for all products grouped-by product type" can be efficiently processed. An analogy in the spatio-temporal domain would be "find the average traffic in all areas in a 1km range around each hospital".

The problem is that the positions and the ranges of spatio-temporal query windows usually do not conform to pre-defined hierarchies, and are not known in advance. Another query, for instance, could involve fire emergencies, in which case the areas of interest would be around fire departments (police stations and so on). In the above example, although the hierarchies are ad-hoc, the spatial dimension is fixed, i.e., there is a static set of road segments. In other applications, the spatial dimensions may be volatile, i.e., the regions at the finest granularity may evolve in time. For instance, the area covered by a cell may change according to weather conditions, extra capacity allocated etc. This dynamic behavior complicates the development of spatio-temporal data warehouses.

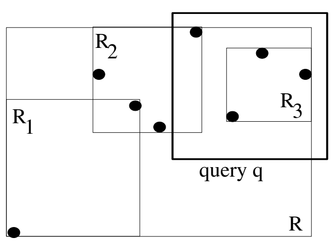
This paper addresses these problems by proposing several indexing solutions. First, we describe spatial trees suitable for the retrieval of aggregate information at a single timestamp. Then, we deal with static spatial dimensions focusing on queries that ask for historical aggregated data in a query window over a continuous time interval. An example would be "give me the number of cars in the city center during the last hour". For such queries we develop multi-tree indexes that combine the spatial and temporal dimensions. In contrast with traditional OLAP solutions, we use the index structure to define hierarchies and we store pre-aggregated data in internal nodes. Finally, we extend our techniques to volatile regions that change over time.

Depending on the type of queries posed, a spatio-temporal OLAP system should capture different types of summarized data. Since our focus is on indexing, we assume some simple aggregate functions like count, or average. In more complex situations we could also store additional measures including the source and the destination of data, direction of movement and so on. Such information will enable analysts to identify certain motion and traffic patterns which cannot be easily found by using the raw data. The proposed methods can be modified for this case. The rest of the paper is organized as follows. Section 2 describes aggregate spatial access methods, while Section 3 proposes indexing techniques for spatio-temporal data, applicable in the presence of static regions. Section 4 discusses structures for volatile regions and Section 5 concludes the paper with a discussion on future work.

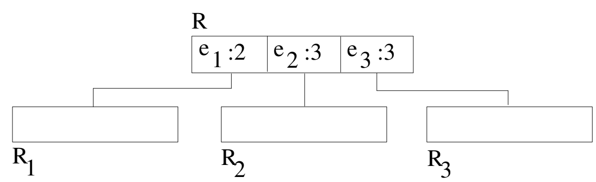
2 Spatial Aggregate Structures

A *window aggregate query* (WA for short) returns summarized information about objects that fall inside the query window, for example the number of cars in a road segment, the average number of mobile phone users per city block etc. An obvious approach to answer such queries is to first retrieve the actual objects by performing traditional window queries, and then compute the aggregate function. This, however, entails a lot of unnecessary effort, compromising performance. A solution for the problem is to store aggregate information in the nodes of specialized index structures.

The aggregate R-tree [8] improves the original R-tree [4, 3] towards aggregate processing by storing, in each intermediate entry, summarized data about objects residing in the subtree. In case of the *count* function, for example, each entry stores the number of objects in its subtree (the extension to any non-holistic functions is straightforward). Figure 1a shows a simple example where 8 points are clustered into 3 leaf nodes R_1, R_2, R_3, which are further grouped into a root node R. The solid rectangles refer to the MBR of the nodes. The corresponding R-tree with intermediate aggregate numbers is shown in Figure 1b. Entry e_1 : 2, for instance, means that 2 points are in the subtree of e_1 (i.e., node R_1). Notice that each point is counted only once, e.g., the point which lies inside the MBRs of both R_1 and R_2 is added to the aggregate result of the node where it belongs (e_1). The WA query represented by the bold rectangle in Figure 1a is processed in the following manner. First the root R is retrieved and each entry inside is compared with the query rectangle q. One of the 3 following conditions holds: (i) the (MBR of the) entry does not intersect q (e.g., entry e_1) and its sub-tree is not explored further; (ii) the entry partially intersects q (e.g., entry e_2) and we retrieve its child node to continue the search; (iii) the entry is contained in q (e.g., entry e_3), in which case, it suffices to add the aggregate number of the entry (e.g., 3 stored with e_3) without accessing its subtree. As a result, only two node visits (R and R_2) are necessary. Notice that conventional R-trees would require 3 node visits.



(a) Clustering of points



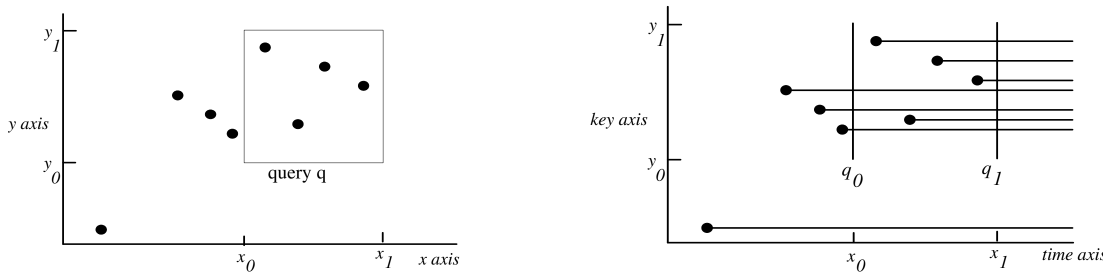
(b) The corresponding aR-tree

Figure 1: An aR-tree example

In summary, the improvement of the aR-tree over the conventional R-tree is that we do not need to visit the nodes (whose MBRs are) inside the query window, but only those nodes that intersect the edges of the window. The cost savings obviously increase with the size of the query window, an important fact because OLAP queries often involve large ranges. Notice, however, that despite the improvement of the aR-tree, query performance is still sensitive to the window size since, the larger the window, the higher the number of node MBRs that are expected to intersect its sides. Another structure, the aP-tree [10], overcomes this problem (i.e., the cost is independent of the query extent) by transforming points to intervals in the key-time plane as follows: the y-coordinate of the point can be thought of as a key value, while the x-coordinate represents the starting time of the interval. The ending time of all intervals is the current time (lying on the right boundary of the time axis). Figure 2a shows the points used in the example of Figure 1a, and Figure 2b illustrates the resulting intervals.

The original query is also transformed since the goal now is to retrieve the number of intervals that intersect the vertical line segment q_1 but not q_0. The intervals are stored using a variation multi-version B-trees [1] enhanced with aggregate information in intermediate entries. Query processing can be reduced to the vertical line segment intersection problem optimally solved by the multi-version B-tree, except that here we are interested in the aggregate number, instead of the concrete ids, of the qualifying objects. This fact differentiates query processing since we can avoid the retrieval of the actual objects intersecting q_1 and q_0 and the expensive computation of their set difference. The evaluation of [10] suggests that the aP- is faster than aR-tree at the expense of space consumption, which is $O(nlogn)$ (n is the number of records) as opposed to $O(n)$ for the aR-tree.

The *window-interval aggregate* query (WIA for short) is the natural extension of WA queries in the spatio-temporal domain. In particular, a WIA query (q_s,q_t) retrieves historical summarized information about objects



(a) Original points and query

(b) Transformed intervals and queries

Figure 2: Transformation of the problem

that fall inside the query window q_s during interval q_t. The next section discusses structures that can efficiently process such queries.

3 Indexing Static Spatial Dimensions

The most common conceptual model for data warehouses is the multidimensional data view. In this model, there is a set of numerical *measures* which are the items of analysis, for example *number* of *objects* (cars or mobile phone users). A measure depends on a set of dimensions, *Region* and *Time*, for instance. Thus, a measure is a value in the multidimensional space which is defined by the dimensions. Each dimension is described by a domain of attributes (e.g. days). The set of attributes may be related via a hierarchy of relationships, a common example of which is the temporal hierarchy (day, month, year). Figure 3 illustrates a simple case; observe that although the regions are 2-dimensional, they are mapped as one dimension in the warehouse. Region R_1 contains 150 objects during the first two timestamps and this number gradually decreases. The star schema [6] is a common way to map multi-dimensional data onto a relational database. A main table (called *fact table*) F, stores the multidimensional array of measures, while auxiliary tables $D_1, D_2, \ldots, D_n$ store the details of the dimensions. A tuple in F has the form $\langle D_i[].key, M[]\rangle$ where $D_i[].key$ is the set of foreign keys to the dimension tables and $M[]$ is the set of measures.

		aggregate results over timestamps					total sum
		369	369	367	364	359	1828
regions	R_4	12	12	12	12	12	60
	R_3	132	127	125	127	127	638
	R_2	75	80	85	90	90	420
	R_1	150	150	145	135	130	710
		T_1	T_2	T_3	T_4	T_5 now	aggregate results over regions

FACT TABLE

Figure 3: A data cube example

OLAP operations ask for a set of tuples in F, or for aggregations on groupings of tuples. Assuming that there is no hierarchy in the dimensions of the previous example, we identify four possible groupings: i) Group-by Region and Time, which is identical to F, ii-iii) group-by Region (Time), which corresponds to the projection of F on the *region* (*time*) -axis, and iv) the aggregation over all values of F which is the projection on the origin. Figure 3 depicts these groupings assuming that the aggregation function is $count$. The fact table, together with all possible combinations of group-bys, compose the *data cube* [5]. Although all groupings can be derived from F, in order to accelerate query processing some results may be pre-computed and stored as *materialized views*.

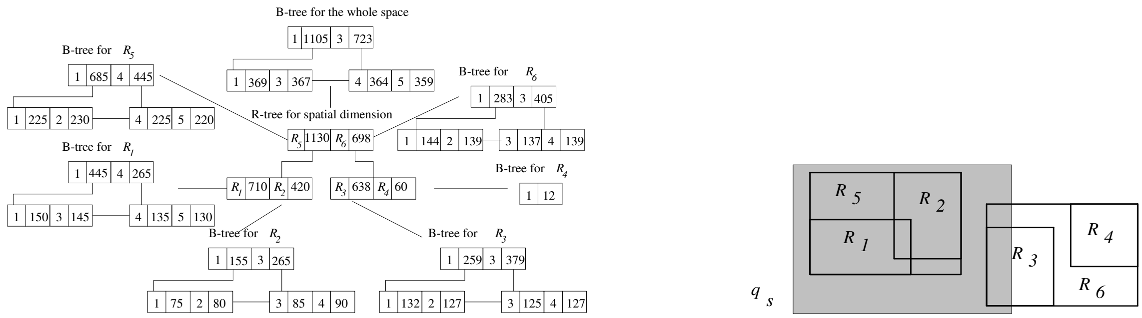
Since, the spatial dimension has no one-dimensional order we store the table in the secondary memory ordered by time and build a B-tree index to locate the blocks containing information about each timestamp. The processing of a typical WIA query employs the B-tree index to retrieve the blocks (i.e., table columns) containing information about q_t and then all regions are scanned sequentially. The aggregate data of those qualifying q_s is accumulated in the result. In the sequel, we refer to this approach as *column scanning*. An alternative approach, which achieves simultaneous indexing on both spatial and temporal dimensions, can be obtained by the generalization of the aR-tree to 3-dimensional space.[1] In particular, each entry r of the *aggregate 3DR-tree* (a3DR-tree) has the form $\langle r.MBR, r.pointer, r.lifespan, r.aggr[] \rangle$, i.e., for each region it keeps the aggregate value and the interval during which this value is valid. Whenever the aggregate information about a region changes, a new entry is created. Using the example of Figure 3, four entries are required for R_1: one for timestamps 1 and 2 where the aggregate value remains 150, and three more entries for the other timestamps where the aggregate value changes. Although the a3DR-tree integrates spatial and temporal dimensions in the same structure (and is, therefore, expected to be more efficient than column scanning for WIA queries that involve both conditions), it has the following drawbacks: (i) it wastes space by storing the MBR each time there is an aggregate change (e.g., the MBR of R_1 is stored four times), and (ii) the large size of the structure and the small fanout of the nodes compromises query efficiency.

In order to overcome these problems, we present a novel multi-tree structure, the *aggregate R- B-tree* (aRB-tree), which is based on the following concept: the regions that constitute the spatial hierarchy are stored only once and indexed by an R-tree. For each entry of the R-tree (including intermediate level entries), there is a pointer to a B-tree which stores historical aggregated data about the entry. In particular, each R-tree entry r has the form $\langle r.MBR, r.pointer, r.btree, r.aggr[] \rangle$ where $r.MBR$ and $r.pointer$ have their usual meaning; $r.aggr[]$ keeps summarized data about r accumulated over all timestamps (e.g., the total number of objects in r throughout history), and $r.btree$ is a pointer to the B-tree which keeps historical data about r. Each B-tree entry b, has the form $\langle b.time, b.pointer, b.aggr[] \rangle$ where $b.aggr[]$ is the aggregated data for $b.time$. If the value of $b.aggr[]$ does not change in consecutive timestamps, it is not replicated.

Figure 4a illustrates an aRB-tree using the data of the cube in Figure 3. For instance, the number 710 stored with the R-tree entry R_1, denotes that the total number of objects in R_1 is 710. The first leaf entry of the B-tree for R_1 (1, 150) denotes that the number of objects in R_1 at timestamp 1 is 150. Similarly the first entry of the top node (1, 445) denotes that the number of objects during the interval [1,3] is 445. The same information is also kept for the intermediate entries of the R-tree (i.e., R_5 and R_6). The topmost B-tree corresponds to the root of the R-tree and stores information about the whole space. Its role is similar to that of the extra row in Figure 3, i.e., answer queries involving only temporal conditions.

The aRB-tree facilitates the processing of WIA queries, by eliminating the need to visit nodes which are totally enclosed by the query. As an example, consider that a user is looking for all objects in some region overlapping the (shaded) query window q_s of Figure 4b during the time interval [1,3]. Search starts from the root of the R tree. Entry R_5 is totally contained inside the query window and the corresponding B-tree is retrieved. The top node of this B-tree has the entries (1, 685), (4, 445) meaning that the aggregated data correspond to the intervals [1,3], [4,5]. Therefore, the next level of the B-tree does not need to be accessed and the contribution of R_5 to the query result is 685. The second root entry of the R-tree, R_6, partially overlaps the query window so the corresponding node is visited. Inside this node only entry R_3, intersects q_s, and its B-tree is retrieved. The first entry of the top node suggests that the contribution of R_3, for the interval [1,2] is 259. In order to complete the result we will have to descend the second entry and retrieve the aggregate value of R_3 for timestamp 3 (i.e., 125). The final result (i.e., total number of objects in these regions in the interval [1,3]) is the sum 685+259+125.

[1] For the following discussion we assume aR-trees as the spatial aggregate structure because the aP-tree cannot be easily generalized to more than two dimensions.



(a) aRB-tree

(b) example query

Figure 4: Example of aRB-tree

This corresponds to the sum of aggregate data in the gray cells of Figure 3.

If the aggregate data is not very dynamic, the size of structure is expected to be smaller than the data cube because it does not replicate information that remains constant for adjacent intervals. Even in the worst case that the aggregate data of all regions change each timestamp, the size of aRB-trees is about double that of the cube since the leafs (needed also for the cube) consume at least half of the space. Furthermore, aRB-trees are beneficial regardless of the selectivity, since: (i) if the query window (q_s, q_t) is large, many nodes in the intermediate levels of the aRB-tree will be contained in (q_s, q_t) so the pre-calculated results are used and visits to the lower tree levels are avoided; (ii) If (q_s, q_t) is small, the aRB-tree behaves as a spatio-temporal index. This is also the case for queries that ask for aggregated results at the finest granularity. Next, we extend these concepts for volatile regions.

4 Indexing Dynamic Spatial Dimensions

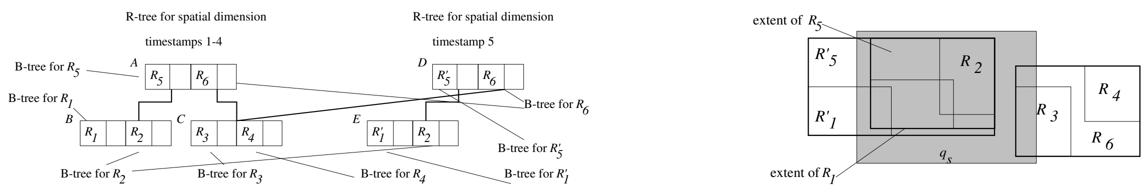
In this section we consider that the finest granularity regions in the spatial dimension, can change their extents over time and/or new regions may appear/disappear. Obviously, when the leaf-level regions change, the spatial tree structure is altered as well. We propose two solutions to this problem by employing alternative multi-tree indexes.

4.1 The aggregate Historical RB-tree

A simple approach to deal with volatile regions is to create a new R-tree every time there is a change. Assume that at timestamp 5, region R_1 is modified to R'_1 and this update alters the father entry R_5 to R'_5. Then, a new R-tree is created at timestamp 5, while the first one dies. In order to avoid replicating the objects that were not affected by the update, we propose the *aggregate Historical R-B-tree* (aHRB-tree), which combines the concepts of aRB-trees and HR-trees [7]. For example in Figure 5a, the two R-trees share node C, because the extents of regions R_3 and R_4 did not change. Each node[2] in the HR-tree, stores a lifespan, which indicates its valid period in history. The lifespans of nodes A and B are [1,4], while that of C is [1,*), where * means that the node is valid until the current time. The form of the entries is the same as in aRB-trees except that $r.aggr[]$, keeps aggregated information about the entry during the lifespan of the node that contains it, instead of the whole history.

Assume that the current time is after timestamp 5, and a query asks for objects in some region overlapping the query window q_s of Figure 5b during the time interval [1,5]. The figure illustrates the old and the new versions after the update at timestamp 5. Both R-trees of Figure 5a are visited. In the first tree, since R_5 is inside

[2] *Historical R-trees* (HR-trees) [7] decrease the level of redundancy by allowing consecutive R-trees to share common branches. Although traditional HR-trees do not store lifespans, we need this information in order to record the validity period of aggregate data in the R-tree nodes and avoid visiting the B-trees.



(a) aHRB-tree (b) example query

Figure 5: Example of aHRB-tree

q_s its child node B is not accessed. Furthermore, as the lifespan of R_5 (i.e., [1,4]) is entirely within the query interval, we retrieve the aggregate data in R_5 without visiting its associated B-tree. On the other hand, node C is accessed (R_6 partially overlaps q_s) and we retrieve the aggregate value of R_3 (for interval [1,5]) from its R-tree entry. Searching the subsequent R-trees is similar, except that shared nodes are not accessed. Continuing the above example, node E is reached and the B-trees of R'_1 and R_2 are searched, while we do not follow the pointer of R_6 (to node C) as C is already visited.[3]

Notice that independently of the query length (q_t), in the worst case the algorithm will visit the B-trees of two R-trees. These are the R-trees at the two ends of q_t. The lifespans of nodes in the trees for intermediate timestamps of q_t are entirely contained in q_t, so the relevant aggregate data stored with the R-tree entries are used directly. Furthermore, although in Figure 5a we show a separate B-tree for each HR-tree entry, the B-trees of various entries may be stored together in a space efficient storage scheme, described [9].

4.2 The aggregate 3DRB-tree

In HR-trees, a node (e.g., B) will be duplicated even if only one of its entries (e.g., R_1) changes. This introduces data redundancy and increases the size of aHRB-trees. The a3DRB-tree (*aggregate 3-dimensional R-B-tree*) avoids this problem, by combining B-trees with 3DR-trees. Every version of a region is modeled as a 3D box, so that the projection on the temporal axis corresponds to a time interval when the spatial extents of the region are fixed; different versions/regions are stored as distinct entries in the 3DR-tree. In particular, a 3DR-tree entry has the form $\langle r.MBR, r.lifespan, r.pointer, r.btree, r.aggr[] \rangle$, where $r.MBR$, $r.pointer$, $r.btree$ are defined as in aRB-trees; $r.aggr[]$ stores data over $r.lifespan$.[4] A typical query involving both spatial and temporal aspects ("find the total number of objects in the regions intersecting some window q_s during a time interval q_t") is also modeled as a 3D box.

Although both aHRB- and a3DRB- trees are aimed at volatile regions they have two important differences: (i) a3DRB-trees maintain a large 3DR-tree for the whole history, while aHRB-trees maintain several small trees, each responsible for a relatively short interval. This fact has implications on their query performance. (ii) The aHRB-tree is an *on-line* structure, while the a3DRB-tree is *off-line*, meaning that the lifespans of its entries should be known before the structure is created; otherwise, we have to store unbounded boxes inside the 3DR-tree, which affects query performance severely.

The experimental evaluation of [9] for static spatial dimensions suggests that the cube implementation is unsuitable in practice due to extreme query cost. aRB-trees consume a fraction of the space required by a3DR-trees, while they outperform them in all cases except for very short query intervals. Furthermore, unlike a3DR-trees where all the data must be known a priori, aRB-trees are on-line structures. For dynamic dimensions, the a3DRB-tree has the best overall performance in terms of size and query cost. Since however, it is an off-line structure, aHRB-trees are the best alternative for applications requiring on-line indexing.

[3] To be specific, the B-trees should be visited only if node E remains alive after timestamp 5. Otherwise, the aggregate values of R'_1 and R_2 for timestamp 5 are stored in E.

[4] The 3DR-tree structure of a3DRB-trees is similar to the a3DR-tree, but now each version is generated by an extent (rather than aggregate) change. Thus, there is no redundancy since the storage of MBRs is required to capture the new extent.

5 Conclusions

Numerous real-life applications require fast access to summarized spatio-temporal information. Although data warehouses have been successfully employed in similar problems for relational data, traditional techniques have three basic impediments when applied directly in spatio-temporal applications: (i) no support for ad-hoc hierarchies, unknown at the design time (ii) lack of spatio-temporal indexing methods, and (iii) limited provision for dimension versioning and volatile regions.

Here, we provide a unified solution to these problems by developing spatio-temporal structures that integrate indexing with the pre-aggregation technique. The intuition is that, by keeping summarized information inside the index, aggregation queries with arbitrary groupings can be answered by the intermediate nodes, thus saving accesses to detailed data. We first consider static dimensions and describe the basic structure (aRB-tree). Subsequently, we present a generalization of aRB-trees, which supports dynamic dimensions (aHRB-tree). For the same case, we also develop a solution based on a 3-dimensional modeling of the problem (a3DRB-tree). Our approach does not aim at simply indexing, but rather replacing the data cube for spatio-temporal data warehouses.

We believe that spatio-temporal OLAP is a new and very promising area, both from the theoretical and practical point of view. Since this is an initial approach, we limited this work to simple numerical aggregations. In the future, we will focus on supporting spatio-temporal "measures" like the direction of movement. This will enable analysts to ask sophisticated queries in order to identify interesting numerical and spatial/temporal trends. The processing of such queries against the raw data is currently impractical considering the huge amounts of information involved in most spatio-temporal applications.

Another interesting area concerns the extension of the proposed techniques to different access methods. For instance, we could apply the R-tree insertion algorithms of [2] in order to obtain on-line structures based on 3DR-trees. Furthermore, the integration of multi-version data structures may provide on-line methods more efficient than aHRB-trees. The problem with such methods (and all methods maintaining multiple R-trees) is the avoidance of multiple visits to the same node via different ancestors. Although various techniques have been proposed in the context of spatio-temporal data structures, it is not clear how they can be applied within our framework.

References

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

[2] Bliujute, R., Jensen, C., Saltenis, S., Slivinskas, G. R-Tree Based Indexing of Now-Relative Bitemporal Data. *VLDB*, 1998.

[3] Beckmann, N., Kriegel, H., Schneider, R., Seeger, B. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. *SIGMOD Conference*, 1990.

[4] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Conference*, 1984.

[5] Gray, J., Bosworth, A., Layman, A., Pirahesh, H. Data Cube: a Relational Aggregation Operator Generalizing Group-by, Cross-tabs and Subtotals. *ICDE*, 1996.

[6] Kimball, R. The Data Warehouse Toolkit. John Wiley, 1996.

[7] Nascimento, M., Silva, J. Towards Historical R-trees. *ACM SAC*, 1998.

[8] Papadias, D., Kalnis, P., Zhang, J., Tao, Y. Efficient OLAP Operations in Spatial Data Warehouses. *SSTD*, 2001.

[9] Papadias, D., Tao, Y., Kalnis, P., Zhang, J. Indexing Spatio-Temporal Data Warehouses. *ICDE*, 2002.

[10] Tao, Y., Papadias, D., Zhang, J. Aggregate Processing of Planar Points. *EDBT*, 2002.

Indexing Mobile Objects Using Duality Transforms

Dimitris Papadopoulos
UC Riverside
dimitris@cs.ucr.edu

George Kollios
Boston University
gkollios@cs.bu.edu

Dimitrios Gunopulos
UC Riverside
dg@cs.ucr.edu

Vassilis J. Tsotras
UC Riverside
tsotras@cs.ucr.edu

Abstract

We present techniques to index mobile objects in order to efficiently answer range queries about their future positions. This problem appears in real-life applications, such as predicting future congestion areas in a highway system, or allocating more bandwidth for cells where high concentration of mobile phones is impending. We address the problem in external memory and present dynamic solutions, both for the one-dimensional, as well as the two-dimensional cases. Our approach transforms the problem into a dual space that is easier to index. Finally we discuss advantages and disadvantages among the various schemes proposed in literature for indexing mobile objects.

1 Introduction

Applications providing location-based services, such as traffic monitoring, intelligent navigation, and mobile communications management, fail to be adequately supported by traditional database management systems. The assumption that data stored in the database remain constant, unless explicitly updated, is the foundation of a model where updates are issued in discrete steps. On the other hand, the above applications deal with continuously changing attributes [21, 26], as for example the object locations. If a DBMS were to update such dynamic attributes every unit of time, it would entail a prohibitively high update overhead.

An elegant solution to address the problem is to use a function of time $f(t)$ to abstract the location of a mobile object. The current location of a moving object at any time instant can then be easily calculated, since an update has to be issued only when the parameters of f change (e.g. speed, direction) [21, 25, 14, 17, 1, 3]. Clearly, this approach reduces the update overhead. Nevertheless, it presents novel challenges, such as the need for appropriate data models, query languages, and query processing and optimization techniques.

This paper considers the problem of indexing mobile objects. We are interested in answering range queries over the objects' future locations. In particular, we present efficient indexing techniques based on the duality transformation. Both the one-dimensional (moving on a line) and two-dimensional (moving on the plane) cases are discussed.

Section 2 provides a formal problem description, while Section 3 describes the duality transformation. The 1-dimensional case is addressed in Section 4. Section 5 introduces the 1.5-dimensional problem, which is a restricted, yet very interesting, version of the 2-dimensional case. The technique for indexing objects that move freely in two dimensions is illustrated in Section 6. Related work and discussion pointing out advantages and disadvantages of the methods that employ indexing techniques in the primal space and the dual space follows in Section 7. Finally, Section 8 concludes the paper.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Problem description

Consider a database that records the positions of moving objects in two dimensions on a finite terrain. For simplicity, we assume that objects move with a constant velocity vector starting from a specific location at a specific time instant. This is enough for calculating the future position of the object, provided that the characteristics of its motion remain the same. Objects update their motion information when their speed and direction change. Moreover, the system is dynamic, i.e. objects may be deleted or new objects may be inserted.

Let $P(t_0) = [x_0, y_0]$ be the initial position at time t_0. Then, its position at time $t > t_0$ is $P(t) = [x(t), y(t)] = [x_0 + v_x(t - t_0), y_0 + v_y(t - t_0)]$, where $V = [v_x, v_y]$ is its velocity vector. Figure 1 depicts the projection of the object trajectories on the (t, y) plane.

We would like to answer queries of the form: "Report the objects located inside the rectangle $[x_{1q}, x_{2q}] \times [y_{1q}, y_{2q}]$ at the time instants between t_{1q} and t_{2q} (where $t_{now} \leq t_{1q} \leq t_{2q}$), given the current motion information of all objects" (This is called the *two dimensional MOR query* in [14]).

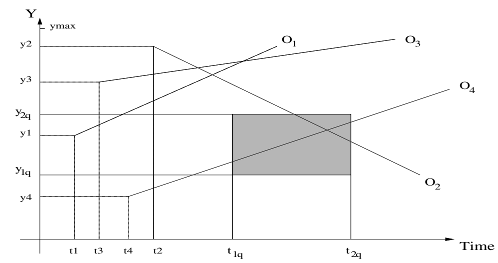


Figure 1: Trajectories and query in (t, y) plane.

We make the assumption that moving objects have velocities in $[v_{min}, v_{max}]$. This is a realistic assumption; for example objects moving slower than v_{min} can be considered (for all practical purposes) "static" and examined separately. Similarly, in a typical scenario objects have some maximum speed.

3 The dual space-time representation

Figure 1 corresponds to the straightforward approach of representing a moving object, i.e. by plotting its trajectory as a line in the time-location (t, y) plane (same for (t, x) plane). The equation describing each line is $y(t) = vt + a$ where v is the slope (velocity in this case) and a is the intercept, which is computed using the motion information. In this setting, also termed as the "primal" space, the query is expressed as the 2-dimensional interval $[(y_{1q}, y_{2q}), (t_{1q}, t_{2q})]$, and it reports the objects that correspond to the lines intersecting the query rectangle.

The general duality transform maps a hyper-plane h from R^d to a point in R^d and vice-versa. For example, a line with equation $y(t) = vt + a$ from the primal plane (t, y) is mapped to a point (v, a) in the dual plane, where one axis represents the velocity v and the other the intercept a (this is called Hough-X transform in [12]). Accordingly, the 1-d query $[(y_{1q}, y_{2q}), (t_{1q}, t_{2q})]$ becomes a polygon in the dual space. By using a linear

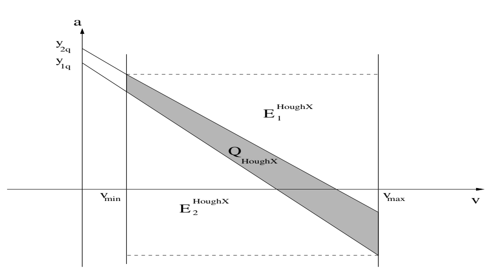


Figure 2: Query on the dual Hough-X plane.

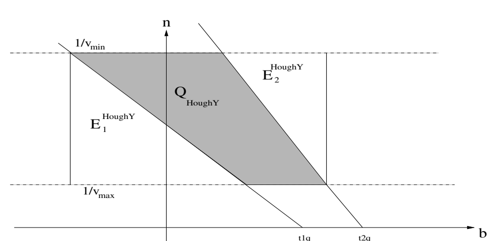


Figure 3: Query on the dual Hough-Y plane.

constraint query [8], the query Q in the dual Hough-X plane (Figure 2) is expressed in the following way [14]:

If $v > 0$, then $Q = C_1 \wedge C_2 \wedge C_3 \wedge C_4$, where:
$C_1 = v \geq v_{min}$, $C_2 = v \leq v_{max}$,
$C_3 = a + t_{2q}v \geq y_{1q}$ and $C_4 = a + t_{1q}v \leq y_{2q}$

If $v < 0$, then $Q = D_1 \wedge D_2 \wedge D_3 \wedge D_4$, where:
$D_1 = v \leq -v_{min}$, $D_2 = v \geq -v_{max}$,
$D_3 = a + t_{1q}v \geq y_{1q}$ and $D_4 = a + t_{2q}v \leq y_{2q}$

A class of transforms with similar properties may be used for the mapping. The problem setting parameters determine which one is more useful. For example, by rewriting the equation $y = vt + a$ as $t = \frac{1}{v}y - \frac{a}{v}$, we can arrive to a different dual representation. The corresponding point in this dual plane has coordinates (n, b), where $n = \frac{1}{v}$ and $b = -\frac{a}{v}$ (called the Hough-Y transform in [12]). Coordinate b represents the point where the line intersects the line $y = 0$ in the primal space. Note that the Hough-Y transform cannot represent horizontal lines. Similarly, the Hough-X transform cannot represent vertical lines. Nevertheless, since in our setting lines have a minimum and maximum slope (velocity is bounded by $[v_{min}, v_{max}]$), both transforms can be used.

4 Indexing in one dimension

By using the dual space-time representation, the problem of indexing mobile objects on a line is transformed into the problem of *simplex* range searching in two dimensions. In simplex range searching we are given a set S of 2-dimensional points, and we want to answer efficiently queries of the following form: given a set of linear constraints $ax \leq b$, find all points in S that satisfy all the constraints. Geometrically, the constraints form a polygon on the plane, and we want to find the points in the interior of the polygon. In [14] it was shown that simplex reporting in d-dimensions with a query time of $O(n^\delta + \kappa)$ I/O's, where N is the number of points, $n = N/B$, K is the number of reported points, $k = K/B$, and $0 < \delta \leq 1$, requires $\Omega(n^{d(1-\delta)-\epsilon})$ disk blocks, for any fixed ϵ.

A corollary of this lower bound is that in the worst case a data structure that uses linear space to answer the 2-dimensional simplex range query and thus the 1-dimensional MOR query, requires $O(\sqrt{n} + k)$ I/O's. In [14] an almost optimal solution based on partition trees [15] was presented. That solution achieves $O(n^{\frac{1}{2}+\epsilon} + k)$ query time using linear space, but the hidden constant factor becomes large even for small ϵ.

As a more practical approach (with good average query performance), [14] proposed to index the dual points using a point access method (PAM). Even though PAMs were designed to answer *orthogonal* range queries, Goldstein et al. [8] have proposed an algorithm to answer simplex range search using R-trees. This is accomplished by changing the search procedure of the tree. Apart from the R-tree family, this method can be applied to other access methods. In particular, [14] suggests using a k-d-tree like structure to index the 1-dimensional problem in the Hough-X (or Hough-Y) space, for an expected logarithmic query time.

A different approach is based on a *query approximation* idea using the Hough-Y dual plane. In general, the b coordinate can be computed at different horizontal $(y = y_r)$ lines. The query region is described by the intersection of two half-plane queries (Figure 3). The first line intersects the line $n = \frac{1}{v_{max}}$ at the point $(t_{1q} - \frac{y_{2q}-y_r}{v_{max}}, \frac{1}{v_{max}})$ and the line $n = \frac{1}{v_{min}}$ at the point $(t_{1q} - \frac{y_{2q}-y_r}{v_{min}}, \frac{1}{v_{min}})$. Similarly the other line that defines the query intersects the horizontal lines at $(t_{2q} - \frac{y_{1q}-y_r}{v_{max}}, \frac{1}{v_{max}})$ and $(t_{2q} - \frac{y_{1q}-y_r}{v_{min}}, \frac{1}{v_{min}})$, respectively.

Since access methods are more efficient for rectangular queries, suppose that we approximate the simplex query with a rectangular one. In Figure 3 the query rectangle will be $[(t_{1q} - \frac{y_{2q}-y_r}{v_{min}}, t_{2q} - \frac{y_{1q}-y_r}{v_{max}}), (\frac{1}{v_{max}}, \frac{1}{v_{min}})]$. Note that the query area is enlarged by the area $E^{HoughY} = E_1^{HoughY} + E_2^{HoughY}$ which is computed as:

$$E^{HoughY} = \frac{1}{2}(\frac{v_{max} - v_{min}}{v_{min} \times v_{max}})^2(\mid y_{2q} - y_r \mid + \mid y_{1q} - y_r \mid) \tag{1}$$

The objective is to minimize E^{HoughY} since it represents a measure of the extra I/O's that an access method will have to perform for solving an 1-dimensional MOR query. E^{HoughY} is based on both y_r (i.e. where the b coordinate is computed) and the query interval (y_{1a}, y_{2q}) which is unknown. Hence, c indices are kept (where c is a small constant) at equidistant y_r's. All c indices contain the same information about the objects, but use different y_r's. The i-th index stores the b coordinates of the data points using $y = \frac{y_{max}}{c} \times i, i = 0, \ldots, c-1$. Conceptually, y_i serves as an "observation" element, and its corresponding index stores the data as observed from position y_i. A given 1-dimensional MOR query will be forwarded to the index(es) that minimize E^{HoughY}. Since all 2-dimensional approximate queries have the same rectangle side $(\frac{1}{v_{max}}, \frac{1}{v_{min}})$, the rectangle range search is equivalent to a simple range search on the b coordinate axis. Thus each of the c "observation" indices can simply be a B+-tree [7]. [14] shows that the query can be answered with bounded error in logarithmic time.

5 The 1.5-dimensional problem

When the mobile objects are restricted to move on a given collection of routes (roads) on the finite terrain, an interesting case of the general 2-dimensional problem is formed, namely the 1.5-dimensional problem. There is a strong motivation for such an environment: for the applications we consider, objects (cars, airplanes) move through a network of predefined routes (freeways, airways).

The 1.5-dimensional problem can naturally be reduced to a collection of 1-dimensional queries. Specifically, each predefined route can be represented as a sequence of connected line segments. The positions of these line segments are indexed by a standard SAM. The extra cost for maintaining this index is negligible, since: (i) there are far less routes than moving objects, (ii) each route can be approximated by a small number of line segments, and, (iii) new routes are not introduced frequently. We apply the techniques for the 1-dimensional case in order to index the objects moving on a segment of a given route.

In order to answer the two dimensional MOR query, the SAM described above identifies the intersection of the routes with the query's spatial predicate, i.e. the rectangle $[x_{1q}, x_{2q}] \times [y_{1q}, y_{2q}]$. Since each route is modeled as a sequence of line segments, the intersection is also a set of line segments, possibly disconnected. Each such intersection corresponds to the spatial predicate of an 1-dimensional query for this route. In this setting we assume that when routes intersect, objects remain in the route previously traveled (otherwise an update is issued).

6 Indexing in two dimensions

The general 2-dimensional problem (Figure 4) is addressed by decomposing the motion of the object into two independent motions, one in the (t, x) plane and one in the (t, y) plane. Each motion is indexed separately. Next we present the procedure used in order to build the index, as well as the algorithm for answering the 2-d query.

6.1 Building the index

We begin by decomposing the motion in (x, y, t) space into two motions on the (t, x) and (t, y) plane.

Furthermore, on each projection, we partition the objects according to their velocity. Objects with small velocity magnitude are stored using the Hough-X dual transform, while the rest of them are stored using the Hough-Y transform, i.e into distinct index structures.

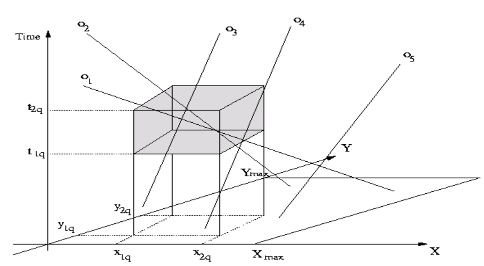


Figure 4: Trajectories and query in (x, y, t) space.

The reason for using different transforms is that motions with small velocities in the Hough-Y approach are mapped into dual points (n, b) having large n coordinates ($n = \frac{1}{v}$). Thus, since few objects have small velocities, by storing the Hough-Y dual points in an index structure such an R*-tree, MBR's with large extents are introduced, and the index performance is severely affected. On the other hand, by using a Hough-X index for the small velocities' partition, we eliminate this effect, since the Hough-X dual transform maps an object's motion to the (v, a) dual point.

When a dual point is stored in the index responsible for the object's motion in one of the planes, i.e. (t, x) or (t, y), information about the motion in the other plane is also included. Thus, the leaves of both indices for the Hough-Y partition store the record (n_x, b_x, n_y, b_y). Similarly, for the Hough-X partition, in both projections, we keep the record (v_x, a_x, v_y, a_y). In this way, the query can be answered using one of the indices; either the one responsible for the (t, x) or the (t, y) projection.

On a given projection, the dual points (i.e. (n, b) and (v, a)) are indexed using R*-trees [4]. The R*-tree has been modified in order to store points at the leaf level, and not degenerated rectangles. Therefore, we can afford storing extra information about the other projection. An outline of the procedure for building the index follows:

1. Decompose the 2-d motion into two 1-d motions on the (t, x) and (t, y) planes.
2. For each projection, build the corresponding index structure. To do so, partition the objects according to their velocity: Objects with small velocity are stored using the Hough-X dual transform, while the rest are stored using the Hough-Y dual transform. Motion information about the other projection is also included.

In order to pick one of the two projections and answer the simplex query, we use the techniques described next.

6.2 Answering the query

The 2-dimensional MOR query is mapped to a simplex query in the dual space. The query is the intersection of four 3-d hyperplanes, while the projection of the query to (t, x) and (t, y) planes are wedges, as in the 1-dimensional case.

A given a 2-dimensional MOR query, is first decomposed into two 1-dimensional queries, one for each projection. Furthermore, on a given projection, the simplex query is asked in both partitions, i.e. Hough-Y (for fast objects) and Hough-X (for slow objects).

On the Hough-Y plane the query region is given by the intersection of two half-plane queries, as shown in Figure 3. Consider the parallel lines $n = \frac{1}{v_{min}}$ and $n = \frac{1}{v_{max}}$. As illustrated in section 4, if the simplex query was answered approximately, the query area would be enlarged by $E^{HoughY} = E_1^{HoughY} + E_2^{HoughY}$ (the triangular areas in Figure 3). Also, let the actual area of the simplex query be Q^{HoughY}. Similarly, on the dual Hough-X plane (Figure 2), let Q^{HoughX} be the actual area of the query, and E^{HoughX} be the enlargement. The algorithm chooses the projection which minimizes the following criterion κ:

$$\kappa = E^{HoughY}/Q^{HoughY} + E^{HoughX}/Q^{HoughX} \tag{2}$$

Since the whole motion information is kept in the indices, it is used in order to produce the exact result set of objects. An outline of the algorithm for answering the exact 2-dimensional MOR query follows:

1. Decompose the query into two 1-d queries, for the (t, x) and (t, y) projection.
2. Get the dual query for each projection (i.e. the simplex query).
3. Calculate the criterion κ for each projection, and choose the one (say π) that minimizes it.
4. Answer the query by searching the Hough-X and Hough-Y partition, using projection π.
5. Put an object in the result set, only if it satisfies the query.
6. Use the full motion information to do the filtering "on the fly".

7 Related work and discussion

While intuitive, the space-time representation (on the "primal" space) is problematic, since trajectories correspond to long lines. Long lines are difficult to index efficiently with traditional indexing techniques. Consider for example using a Spatial Access Method, such an R-tree [11] or an R*-tree [4]. Here each line is approximated by a minimum bounding rectangle (MBR). Obviously, the MBR approximation has much larger area than the line itself. Furthermore, since the trajectory of an object is valid until an update is issued, it has a starting point but no end. Thus all trajectories expand till "infinity", i.e. they share an ending point on the time dimension.

Another approach is to partition the space into disjoint cells and store in each cell those lines that intersect it [24, 6]. This could be accomplished by using an index such an R+-tree [20], a cell-tree [10], and the PMR-quadtree [19]. The shortcoming of this solution is that it introduces data replication, since each trajectory is copied into all cells that intersect it. Moreover, using space partitioning would also result in high update overhead, since when an object changes its motion information, it has to be removed from all cells that stores its trajectory.

An interesting alternative for efficient indexing of mobile objects in the primal space was proposed by Saltenis et al. [17]. They introduced the time-parameterized R-tree (TPR-tree), which extends the R*-tree. The coordinates of the bounding rectangles in the TPR-tree are functions of time and, intuitively, are capable of following the objects as they move. The position of a mobile object is represented by its location at a particular time instant (reference position) and its velocity vector.

In [3] a main memory framework (kinetic data structure) was proposed and addresses the issue of mobility and maintenance of configuration functions among continuously moving objects. Application of this framework to external range trees [2] appears in [1].

Other related work considers nearest neighbor queries in a mobile environment [13, 9, 22], time-parameterized queries [23] and selectivity estimation for moving object range queries [5].

An important characteristic of the TPR-tree and the duality transforms is that they have linear space requirements. We now illustrate the advantages and disadvantages among these two approaches. Experimental results, for the two dimensional case, between them are presented in [16] and are not included here for brevity.

The performance of the TPR-tree highly depends on the knowledge of the characteristics of the workload. The average time interval between updates (UI) and the average length of the temporal part of the queries (W) have to be given as parameters to the index beforehand. These two parameters define the horizon $H = UI + W$. The TPR-tree is optimized for answering queries whose temporal part lies within this horizon (it actually provides the best query performance for queries within the horizon [16]). It also recalculates and updates its time-parameterized MBRs whenever an update is issued. This is necessary for the TPR-tree, since the size of the bounding regions increases over time. If these parameters are not known in advance (for example in applications where UI or W cannot be easily predicted), [18] has proposed to use the automatic horizon estimation feature. Nevertheless, the query performance of the TPR-tree worsens.

On the other hand, the duality transform method has larger space requirements, but better update performance. In our experiments [16], the duality transform uses about double the space, but has about 45% faster updates. The query performance was comparable (on average 20% more) for queries inside the horizon, but was much faster than the TPR-tree performance for queries outside the horizon. However, the duality transforms cannot easily accommodate 3-d datasets, as the TPR-tree does. This is because, in the three-dimensional setting, the movement of an object would have to be decomposed into three independent movements, one for each projection on the planes (t, x), (t, y) and (t, z), in order for the dual transformation method to be applied.

8 Conclusions

We examined the problem of indexing mobile objects using dual transformations. We considered both the 1-dimensional and 2-dimensional cases, and presented external memory indexing techniques, in order to efficiently answer range queries about the objects' future locations. An interesting future direction of research is joins among relations of mobile objects. Furthermore, it would be interesting to study how the mobile objects' agility (i.e. how often updates are issued) affects the performance and robustness of the indexing methods.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. *In Proc. PODS*, pp. 175–186, 2000.

[2] L. Arge, V. Samoladas, and J.S. Vitter. On Two-Dimensinal Indexability and Optimal Range Search Indexing. In *Proc. PODS*, pp. 346–357, 1999.

[3] J. Basch, L. Guibas, and J. Hershberger. Data Structures for Mobile Data. In *Proc. of the 8th ACM-SIAM Symposium on Discrete Algorithms*, pp. 747–756, 1997.

[4] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. ACM SIGMOD*, pp. 322–331, 1998.

[5] Y.-J. Choi and C.-W. Chung. Selectivity Estimation for Spatio-Temporal Queries to Moving Objects. In *Proc. ACM SIGMOD*, 2002 (to appear).

[6] H. D. Chon, D. Agrawal, and A. El Abbadi. Query Processing for Moving Objects with Space-Time Grid Storage Model. In *Proc. MDM*, pp. 121–130, 2002.

[7] D. Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, June 1979.

[8] J. Goldstein, R. Ramakrishnan, U. Shaft, and J.B. Yu. Processing Queries By Linear Constraints. In *Proc. PODS*, pp. 257–267, 1997.

[9] D. Gunopulos, G. Kollios, and V. J. Tsotras. All-Pairs Nearest Neighbors in a Mobile Environment. In *7th Hellenic Conference on Informatics*, Ioannina, Greece, August 1999.

[10] O. Gunther. The Design of the Cell Tree: An Object-Oriented Index Structure for Geometric Databases. In *Proc. ICDE*, 1989.

[11] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. ACM SIGMOD*, pp. 47–57, 1984.

[12] H. V. Jagadish. On Indexing Line Segments. In *Proc. VLDB*, pp. 614–625, 1990.

[13] G. Kollios, D. Gunopulos, and V. Tsotras. Nearest Neighbor Queries in a Mobile Environment. In *Proc. of the Spatio-Temporal Database Management Workshop, Edinburgh, Scotland*, pp. 119–134, 1999.

[14] G. Kollios, D. Gunopulos, and V. Tsotras. On Indexing Mobile Objects. In *Proc. PODS*, pp. 261–272, 1999.

[15] J. Matousek. Efficient Partition Trees. *Discrete and Computational Geometry*, 8:432–448, 1992.

[16] D. Papadopoulos, G. Kollios, D. Gunopulos, and V.J. Tsotras. Indexing Mobile Objects on the Plane. In *Proc. 5th International Workshop on Mobility in Databases and Distributed Systems*, 2002, (to appear).

[17] S. Saltenis, C. S. Jensen, S. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. ACM SIGMOD*, pp. 331–342, May 2000.

[18] S. Saltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *Proc. ICDE*, pp. 463–472, 2002.

[19] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, June 1990.

[20] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proc. VLDB*, pp. 507–518, 1987.

[21] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proc. ICDE*, pp. 422–432, 1997.

[22] Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proc. SSTD*, pp. 79–96, 2001.

[23] Y. Tao and D. Papadias. Time-Parameterized Queries in Spatio-Temporal Databases. In *Proc. ACM SIGMOD*, 2002.

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

[25] O. Wolfson, S. Chamberlain, S.Dao, L. Jiang, and G. Mendez. Cost and Imprecision in Modeling the Position of Moving Objects. In *Proc. ICDE*, pp. 588–596, 1998.

[26] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In *Proc. SSDBM*, pp. 111–122, 1998.

Advances in Indexing for Mobile Objects

Pankaj K. Agarwal
Department of Computer Science
Duke University
Durham, NC 27708
pankaj@cs.duke.edu *

Cecilia M. Procopiuc
AT&T Research Labs
180 Park Ave., Box 971
Florham Park, NJ 07932
magda@research.att.com

1 Introduction

With the rapid advances in positioning systems, such as GPS, ad-hoc networks, and wireless communication, it is becoming increasingly feasible to track and record the changing position of continuously moving objects. Several areas such as digital battlefields, air-traffic control, mobile communication, navigation system, and geographic information systems need the capability to index moving objects, so that various queries on them can be answered in real time. The queries might relate either to the current configuration of objects, to a past configuration, or to a future configuration — in the last case, we are asked to predict the behavior based on the current information. However, most existing database systems assume that the data is constant unless it is explicitly modified. Such systems are not suitable for representing, storing, and querying continuously moving objects, since, unless the database is continuously updated (at considerable time and resources expense), a query output will be obsolete. The need for new database technology that can support kinetic applications has given rise to a large number of questions, from defining suitable models and query languages [18, 28] to developing new engines capable of indexing and querying moving objects efficiently.

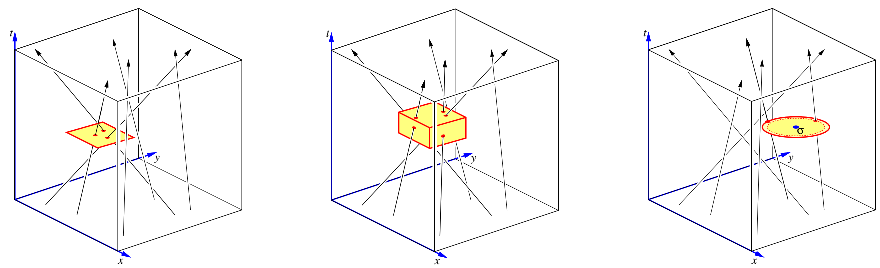
In this paper we discuss the latter line of research, and we review recent advances and remaining challenges. Since moving objects can be viewed as points moving along algebraic curves, the problem has a natural representation in the context of computational geometry. Hence, many results we present here draw heavily on the rich literature of geometric indexing structures, kinetic data structures, and geometric approximation techniques.

Data representation. Let $S = \{p_1, \ldots, p_n\}$ be a set of n points in $\mathbb{R}^2$, each moving continuously. Let $p_i(t) = (x_i(t), y_i(t))$ denote the position of p_i at time t, and let $S(t) = \{p_1(t), \ldots, p_n(t)\}$. Let $\bar{p}_i = \bigcup_t (p_i(t), t)$ denote the trajectory of p_i over time. We say that the motion of S is *linear* if each x_i, y_i is a *linear* function, i.e., each $\bar{p}_i$ is a line in $\mathbb{R}^3$, and the motion of S is *algebraic* if each x_i, y_i is a polynomial of bounded degree. For most of the results discussed in this paper, we assume the motion of S to be *piecewise linear*, i.e., each $\bar{p}_i$ is a polygonal chain in $\mathbb{R}^3$. The trajectories of points can change any time. We assume that the database system is modified whenever these values change. We will use the term *now* to mean the current time.

Queries. Any indexing structure should be able to answer queries based on the current, past, or future positions of the objects. In the case of future positions, the answer is a *prediction* of the query output based on the current

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Research by P.A. is partially supported by NSF under grants CCR-00-86013 EIA-98-70724, EIA-9972879, EIA-01-31905, and CCR-97-32787, and by a grant from the U.S.–Israel Binational Science Foundation.



(a) (b) (c)

Figure 1: Instances of Q1, Q2, and Q3 queries, respectively, with time as the z-axis.

trajectories of the points. In this paper we focus our attention on range-searching and nearest-neighbor queries, which are defined below.

Q1. Given an axis-aligned rectangle R in the xy-plane and a time value t_q, report all points of S that lie inside R at time t_q, i.e., report $S(t_q) \cap R$; see Figure 1 (a).

Q2. Given a rectangle R and two time values $t_1 \leq t_2$, report all points of S that lie inside R at any time between t_1 and t_2, i.e., report $\bigcup_{t=t_1}^{t_2} (S(t) \cap R)$; see Figure 1 (b).

Q3. Given a query point $\sigma \in \mathbb{R}^2$ and a time value t_q, a *nearest-neighbor* query requires reporting the nearest neighbor of σ in $S(t_q)$; Figure 1 (c).

Model of computation. In order to be useful in practice, any proposed method must scale well over large data sets. Since the data resides on disk, the focus of these approaches is to minimize the data transfer between disk and main memory, which is likely to be the bottleneck for any system. Most of the results in this paper will be discussed in the *parallel disk model* introduced by Vitter and Shriver [31], which closely models the way computers handle data. This model assumes that each disk access transmits a contiguous block of B units of data in a single *input/output operation* or *I/O*. The efficiency of a data structure is measured in terms of the amount of disk space it uses (measured in units of disk blocks) and the number of I/Os required to answer a query. Since we are also interested in solutions that are *output sensitive*, our query I/O bounds are expressed not only in terms of N, the number of points in S, but also in terms of K, the number of points reported by the query. Note that we need at least $\lceil N/B \rceil$ blocks to store all N points, and at least $\lceil K/B \rceil$ blocks to store the output from a range query. We refer to these bounds as "linear" and introduce the notation $n = \lceil N/B \rceil$ and $k = \lceil K/B \rceil$.

General approaches. Two general approaches have been proposed to index moving points. The first approach, which we refer to as a *time-oblivious* approach, regards time as a new dimension and indexes the trajectories $\bar{p}_i$ of input points p_i. One can either index the trajectories themselves using various techniques [24, 29], or one can work in a parametric space [1, 21], map each trajectory to a point in this space, and build an index on these points. An advantage of a time-oblivious scheme is that the index is updated only if the trajectory of a point changes or when a point is inserted into or deleted from the index. However, updating a trajectory can be quite expensive. Since this approach indexes either curves in $\mathbb{R}^3$ or points in higher dimensions, the query time tends to be large.

The second approach, which we refer to as *kinetic data structures*, builds a dynamic index on the moving points. Roughly speaking, at any time it maintains an index on the current configuration of the points. As the points move, the index is updated. The main observation is that although the points are moving continuously, the index is updated only at discrete time instances. This approach leads to fast query time, but the main disadvantage is that it can answer a query only at the current configuration. It can be extended to handle queries arriving in chronological order, i.e., the time stamps of queries are in nondecreasing order. Using the so-called persistence technique [15, 27], the index can answer queries on the near future or the near past configurations of points. Another weakness of this approach is the cost associated with updating the index as the points move.

One can combine the two approaches to take advantage of each one. In many applications, such as air-traffic control, it is more crucial to answer the queries related to the near future configurations (i.e., $|t_q - now|$ is small) efficiently and accurately. The idea is to still use time as an additional dimension, but optimize the index for the time close to now and store only an approximate representation of the point configurations far away from now. The index is updated periodically to maintain the above invariant. This approach, which we call *time-responsive* indexing, has been used in a number of recent papers [1, 4, 25, 32].

The paper is organized as follows. In Section 2 we review a few geometric techniques that will be useful for the indexes discussed here. For simplicity, we first discuss in Section 3 the above approaches for points moving on the x-axis. Section 4 reviews the known techniques for range searching amid points moving in the plane. Section 5 discusses nearest-neighbor queries on moving points, and we conclude in Section 6 by mentioning a few open problems.

2 Geometric Techniques

In this section we discuss a few geometric techniques, which lie at the heart of many of the indexing schemes discussed in subsequent sections.

Duality. Duality is a popular and powerful technique used in geometric algorithms [14] and has been used in indexing moving points [1, 21, 28]; it maps each point in $\mathbb{R}^2$ to a line in $\mathbb{R}^2$ and vice-versa. We use the following duality transform (a few other variants also appear in the literature): The dual of a point $(a, b) \in \mathbb{R}^2$ is the line $x_2 = ax_1 - b$, and the dual of a line $x_2 = \alpha x_1 + \beta$ is the point $(\alpha, -\beta)$. Let σ^* denote the dual of an object (point or line) σ, and for any set of objects Σ, let Σ^* denote the set of dual objects $\{\sigma^* \mid \sigma \in \Sigma\}$. An essential property of this transformation is that a point p is above (resp., below, on) a line h if and only if the dual point h^* is above (resp., below, on) the dual line p^*. The dual of a strip σ is a vertical line segment σ^*, in the sense that a point p lies inside σ if and only if the dual line p^* intersects σ^*. See Figure 2.

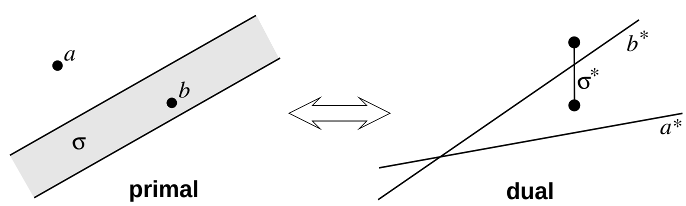


Figure 2: The duals of two points and a strip are two lines and a vertical line segment.

Partition trees. Partition trees, originally proposed by Willard [33], are one of the most commonly used internal memory data structures for geometric-searching problems [5, 23]. Recently, partition trees have been extended to the external memory model [2]. Let S be a set of N points in $\mathbb{R}^2$. A *simplicial partition* of S is a set of pairs $\Pi = \{(S_1, \triangle_1), (S_2, \triangle_2), \ldots, (S_r, \triangle_r)\}$, where the S_i's are disjoint subsets of S, and each $\triangle_i$ is a triangle containing the points in the corresponding subset S_i. A point of S may lie in many triangles, but it belongs to only one subset S_i. The size of Π, here denoted r, is the number of subset-triangle pairs. A simplicial partition is *balanced* if each subset S_i contains between N/r and $2N/r$ points. The *crossing number* of a simplicial partition is the maximum number of triangles crossed by a single line. By extending Matoušek's algorithm to the external memory model, Agarwal *et al.* [2] showed that for $r = O(B)$, a balanced simplicial partition Π for S of size r and crossing number $O(\sqrt{r})$ can be constructed in $O(nr)$ expected I/Os.

Using simplicial partitions, a partition tree T on S can be constructed as follows: Each node v in T is associated with a subset $S_v \subseteq S$ of points and a triangle $\triangle_v$. For the root of T, we have $S_{\text{root}} = S$ and $\triangle_{\text{root}} = \mathbb{R}^2$. Let $N_v = |S_v|$ and $n_v = \lceil N_v/B \rceil$. We construct the subtree rooted at node v as follows. If $N_v \le B$, then v is a leaf and we store all points of S_v in a single block. Otherwise, v is an internal node of degree $r_v = O(B)$. We compute a balanced simplicial partition $\Pi_v = \{(S_1, \triangle_1), \ldots, (S_{r_v}, \triangle_{r_v})\}$ for S_v with crossing number $O(\sqrt{r_v})$ and then recursively construct a partition tree T_i for each subset S_i. T uses $O(n)$ disk blocks and can be constructed in $O(N \log_B n)$ expected I/Os. For a query triangle σ, all points in $S \cap \sigma$ can be reported as follows: we visit T in a top down fashion. Suppose we are at a node v. If v is a leaf, we report all points of S_v that lie inside σ. Otherwise, we test each triangle $\triangle_i$ of Π_v. If $\triangle_i$ lies completely outside σ, we ignore it; if $\triangle_i$ lies completely inside σ, we report all points in S_i by traversing the ith subtree of v; finally, if σ crosses $\triangle_i$, we recursively visit the ith child of v. As shown in [1, 2], the query takes $O(n^{1/2+\varepsilon} + k)$ I/Os. Moreover, a point can be inserted into or deleted from the tree in $O(\log_B^2 n)$ expected I/Os.

Kinetic data structures. The *kinetic data structure* (KDS) framework, originally proposed by Basch *et al.* [12], stores only a "combinatorial snapshot" of the moving points at any time. Although the points are moving continuously, the data structure itself only depends on certain combinatorial properties (such as two points in 1D coincide) and changes only at discrete instants, called *events*. When an event occurs, we perform a *kinetic update* on the data structure. Since we know how the points move, we can predict when any event will occur. The evolution of the data structure is driven by a global *event queue*, which is a priority queue containing all future events. A good KDS is one that strikes a proper balance in maintaining a certificate set. On the one hand, the certificate set should be as stable as possible and not undergo drastic or unnecessarily frequent changes as the objects move; furthermore, it should be repairable by local operations when certificates fail. At the same time, the certificate set has to enable a fast computation of the attribute of interest in order to be useful. See [16] for an overview of KDS.

Persistence. Many database applications ask for updating the current index while querying both the current and earlier versions of the index. The *persistence* technique, proposed by Sarnak and Tarjan [27] and generalized by Driscoll *et al.* [15], provides a way to adapt KDS so that earlier versions of the index can be queried. The persistence technique basically stores the "difference" between two consecutive version of the index, by keeping track of the time interval during which an element is really present in the index. A B-tree can be made persistent as follows [13, 30]. Roughly speaking, each data element is augmented with a *life span* consisting of the time at which the element was inserted and (possibly) the time at which it was deleted. Similarly, each node in the B-tree is also augmented with a life span. We say that an element or a node is *alive* during its life span. Apart from the normal B-tree constraint on the number of elements in a node, we also maintain that a node contains $\Theta(B)$ alive elements (or children) in its life span. This means that for a given time t, the nodes with life span containing t make up a B-tree on the elements alive at that time. An insertion or deletion in a persistent B-tree is performed almost like a normal insertion; we omit the details from here. Becker *et al.* [13] show that each update operation takes $O(\log_B n)$ I/Os, and that ν update operations require $O(\nu/B)$ additional disk blocks.

3 One-Dimensional Indexing

In this section we describe the three general techniques discussed in the introduction for indexing a set S of points moving in $\mathbb{R}$. For simplicity, we assume that the motion of S is linear, i.e., $p_i(t) = a_i t + b_i$, where $a_i, b_i \in \mathbb{R}$, and $\bar{p}_i$ is a line in the xt-plane. Let $L = \{\bar{p}_i \mid p_i \in S\}$.

Time-oblivious indexing. Tayeb *et al.* [29] proposed an index based on the so-called PMR-quadtrees [26], in which a square (associated with a node of the quadtree) is split into four subsquares if at least B line segments intersect it, and a line segment is associated with every subsquare that it intersects. This approach introduces substantial data replication and is thus not space efficient. To reduce the data replication problem, they use a time parameter U and rebuild the entire index at time U. Although this approach does not guarantee any nontrivial bound on the query performance, it extends to algebraic or piecewise-linear motion of S without any modification.

An index with provable query time can be developed using partition trees, described in Section 2. A 1-dimensional Q1 query on S asks which of the lines in L intersect a line segment σ parallel to the x-axis. Let P be the set of points dual to the lines in L; see Section 2. Then the above query is equivalent to reporting the points in P that lie inside the strip σ^* dual to σ. Since σ^* can be regarded as an unbounded triangle, a Q1 query can be answered in $O(n^{1/2+\varepsilon} + k)$ I/Os, and a point can be inserted into or deleted from the index in $O(\log_B^2 n)$ expected I/Os [1, 21]. Agarwal *et al.* [1] also showed that partition trees can also be used to answer Q2 queries with the same performance bounds. The partition-tree approach can be generalized to handle algebraic motion, at the cost of increasing the query time, by using the so-called linearization technique [10, 34]. Suppose the trajectory of each point $p \in S$ is given by a polynomial of degree D in the time parameter t. Using linearization, S is mapped to a set P of points in $\mathbb{R}^{D+1}$ and a Q1 query in S is reduced to reporting the points of P that lie inside a $(D+1)$-dimensional slab. By preprocessing P into a higher dimensional partition tree, a query can be answered using $O(n^{1-/(D+1)+\varepsilon} + k)$ I/Os; see [1] for details.

Chronological queries. The query time can be significantly improved if the index is allowed to change over time and if queries arrive in *chronological order*, i.e., if the time stamp of the current query is t_q, then the time stamp of the future queries is at least t_q. It is well known that a one-dimensional range query among a set of static points in $\mathbb{R}^1$ can be answered in $O(\log n + k)$ I/Os using B-trees. By maintaining B-trees for moving points using the KDS framework outlined in Section 2, a range query can be answered in $O(\log_B n + k)$ I/Os. If the motion of S is algebraic, the index processes $O(N^2)$ events and the index can be updated in $O(\log_B n)$ expected I/Os at each event. Agarwal *et al.* [1] show that the kinetic B-trees can be combined with partition trees to obtain a tradeoff between query time and the total number of kinetic events. Specifically, for a parameter Δ, where $N \leq \Delta \leq N^2$, range queries arriving in chronological order can be answered using $O(N^{1+\varepsilon}/\sqrt{\Delta} + k)$ I/Os, and an event can be processed in $O(\log_B(\Delta/N))$ I/Os, provided the motion of S is linear. If the trajectories of points do not change, the index processes at most Δ events.

Time-responsive indexing. As observed in the previous section, the combinatorial structure of a kinetic data structure is fixed until there is an *event*. For kinetic B-trees, an event occurs when two points have the same value. Using the persistence technique described in Section 2, one can maintain all versions of kinetic B-trees in a total of $O(n + \sigma/B)$ blocks, where $\sigma = O(N^2)$ is the number of events processed by the KDS. Kollios *et al.* [21] used this approach to answer Q1 queries at any time in $O(\log_B n + k)$ I/Os. Agarwal *et al.* [1] showed that instead of maintaining all versions, one can maintain only a few past and future versions of the kinetic B-tree so that a query in the near past or future can be answered efficiently. Specifically, they showed that S can be stored in an index of size $O(n)$ so that a Q1 query can be answered in $O(\log_B n + k)$ I/Os, provided there are at most N events between t_q and now. The amortized cost of an event is $O(\log_B n)$ I/Os. Kollios *et al.* [22] have also used persistence to store multiple versions of other indexing schemes, such as R-trees, on moving objects.

Agarwal *et al.* [1] also described an index that combines partition trees with multiversion kinetic data structures to obtain an index whose query cost is a monotone function of $|t_q - now|$. However, they were not able to prove any bounds on the performance of their index in the worst-case. Subsequently, Agarwal *et al.* [4] proposed another index that provides a bound on the query performance, which is a monotone function of $|t - now|$. Since the index is of combinatorial nature, time is measured in terms of kinetic events. Let $\varphi(t)$ be the number of kinetic events that occur (or have occurred) between now and t. The query bounds then depend on $\varphi(t_q)$, the number of kinetic events between now and t_q. Their index answers a Q1 query at time t_q, with $NB^{i-1} \leq \varphi(t_q) \leq NB^i$, using $O(B^{i-1} + \log_B n + k)$ I/Os, and spends $O(\log_B^3 n)$ I/Os at each kinetic event. Recently Agarwal *et al.* [3] improved the query time to $O(\sqrt{B^{i-1}} + \log_B n + k)$ without affecting the update time.

4 Two-Dimensional Indexing

We now discuss how the techniques described in the previous section extend to points moving in $\mathbb{R}^2$. Again, we first discuss the time-oblivious approach, then the KDS framework, and finally the approaches that combine them.

Time-oblivious indexing. Pfoser *et al.* [24] propose two R-tree based schemes for indexing the trajectories of S, assuming that the motion of S is piecewise linear. For a point $p_i \in S$, let Γ_i denote the set of line segments in the trajectory $\bar{p}_i$, and set $\Gamma = \bigcup_i \Gamma_i$. Their first index, called the STR-tree, regards each segment of Γ independently and builds an R-tree on them. They propose new heuristics to split a node, which take the trajectories of S into account while inserting a new segment into the tree. Since the segments of a trajectory are stored at different parts of the tree, updating a trajectory is expensive. In the second index, called the TB-tree, they circumvent this problem by storing all line segments of the same trajectory at the same leaf of the tree.

If the motion of S is linear, the trajectories of points in S are a set of lines in $\mathbb{R}^3$. Using the fact that a line in $\mathbb{R}^3$ can be represented by four real parameters, Kollios *et al.* [21] proposed mapping each line to a point in $\mathbb{R}^4$ and using four-dimensional partition trees to answer Q1 queries. Agarwal *et al.* [1] developed a considerably faster index by reducing the problem to 1D, using the following observation: a line ℓ in $\mathbb{R}^3$ (xyt-space) intersects a horizontal rectangle R, parallel to the xy-plane, if and only if their projections onto the xt- and yt-planes both intersect. We apply a duality transformation to the xt- and yt-planes, as described in Section 2. Thus, each moving point p in the xy-plane induces two static points p^x and p^y in the dual xt-plane and the dual yt-plane, respectively. For any subset $P \subseteq S$, let P^x and P^y respectively denote the corresponding points in the dual xt-plane and the dual yt-plane. Any query rectangle (query segments in the xt- and yt-planes) induces two query strips σ^x and σ^y, and the result of a query is the set of points $p \in S$ for which $p^x \in \sigma^x$ and $p^y \in \sigma^y$. See Figure 3.

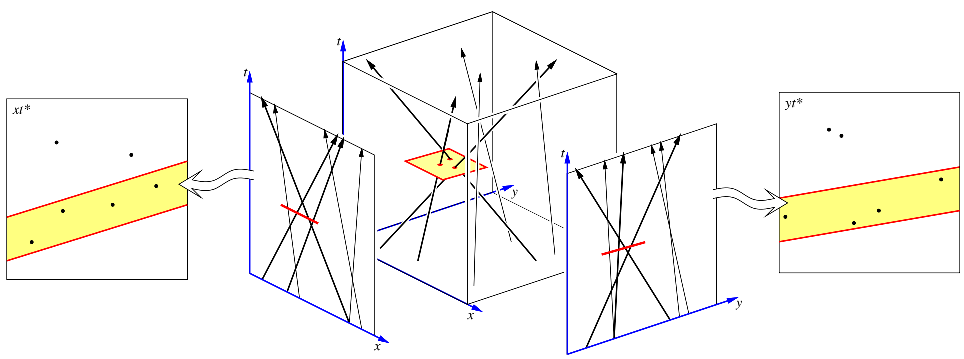


Figure 3: Decomposing a rectangle query among moving two-dimensional points into two strip queries among static two-dimensional points, by dualizing the xt- and yt-projections. A line intersects the rectangle if and only if both corresponding points lie inside the strips.

Agarwal *et al.* [1] propose a multi-level partition tree, a general technique that allows one to answer complex queries by decomposing them into several simpler components and by designing a separate data structure for each component, to answer Q1 and Q2 queries. Roughly speaking, they construct a *primary* partition tree T^x for the points P^x. Then at every node v of T^x, they attach a *secondary* partition tree T_v^y for the points S_v^y, where S_v is the set of points stored in the primary subtree rooted at v. The total space used by the index is $O(n \log_B n)$. A Q1 query is answered in almost the same way as in the basic partition tree. Given two query strips σ^x and σ^y, it first searches through the primary partition tree T^x for the points in $P^x \cap \sigma^x$. If it finds a triangle $\triangle_i$ associated with a node v of the partition tree T^x that lies completely inside σ^x, it searches in the secondary tree T_v^y to report all points of $P_v^y \cap \sigma^y$. Agarwal *et al.* [1] showed that the query takes $O(n^{1/2+\varepsilon} + k)$ I/Os and that the size can be reduced to $O(n)$ without affecting the asymptotic query time. As in the one-dimensional case, a

Q2 query can also be answered using within the same bound using multilevel partition trees, and the index can be generalized to handle algebraic motion of S using the linearization technique.

Chronological queries. Agarwal *et al.* [1] use the KDS framework on external range trees developed by Arge *et al.* [11] to answer Q1 queries on points moving in $\mathbb{R}^2$. The external range tree presented in [11] is a three-level structure. Omitting the details, which can be found in the original paper, we mention that the kinetic range tree uses $O(n \log_B n/(\log_B \log_B n))$ blocks, and a Q1 query with time stamp $t_q = now$ can be answered in $O(\log_B n + k)$ I/Os. The amortized cost of a kinetic event or trajectory change is $O(\log_B^2 n/ \log_B \log_B n)$ I/Os. If the trajectories of the points do not change, the total number of events is $O(N^2)$. As in the one-dimensional case, by combining partition trees with kinetic range trees, a tradeoff between query time and the number of events can be obtained.

Since external range trees are too complicated, a more practical approach is to use the KDS framework on kd-trees, as proposed in [7]. However, the invariants maintained by a kd-tree — at each node points are partitioned into two halves by a line and the orientation of the line alternates between being horizontal and vertical along a path — are too expensive to maintain for moving points. Agarwal *et al.* [7] propose two variants of kinetic kd-trees in internal memory model: the *pseudo kd-tree*, which allows the number of points stored in the two children of a node to differ by a constant fraction, and the *overlapping kd-tree*, which allows the bounding boxes of two children of a node to overlap. Both variants answer Q1 queries that arrive in chronological order in $O(N^{1/2+\varepsilon})$ time, for any constant $\varepsilon > 0$, process $O(N^2)$ kinetic events, and spend only polylogarithmic time at each event. In contrast to other traditional approaches, which require subtrees to be rebuilt once in a while and thus generate some expensive updates, the update time in their structure is worst-case as long as the trajectories of the points do not change. Efficient KDS for maintaining binary space partition trees (also known as cell trees [17]), which are generalizations of kd-trees have also been developed [6, 8].

Time-responsive indexing. Similar to the one-dimensional case, one can maintain multiversion external kinetic range trees in order to answer queries both in the past and the future. Agarwal *et al.* [1] describe a method for maintaining such a structure and show that its size is $O(n \log_B n/(\log_B \log_B n))$ blocks, and the amortized cost per event is $O(\log_B^2 n)$ I/Os. A Q1 query can be answered in $O(\log_B n + k)$ I/Os, provided there are at most $n/ \log_B n$ events between t and the current time. They also combine partition trees with multiversion kinetic data structures to obtain an index whose query cost is a monotone function of $|t_q - now|$. A different structure is proposed in [4]. It answers a Q1 query at time t_q in $O(\sqrt{N/B^i}(B^{i-1} + \log_B N) + k)$ I/Os, provided that the number of kinetic events that occur up to time t_q is in the interval $[NB^{i-1}, NB^i]$. The bounds were later improved in [3].

Since the above time-responsive indexing schemes are not practical, it is desirable to kinetize one of the simpler structures such as an R-tree. Recall that each node v of an R-tree is associated with a subset S_v of points and the smallest rectangle R_v containing S_v. Although the rectangles associated with the children of a node can overlap, the areas of overlap and the areas of the bounding boxes must be small to expect good query performance. Maintaining these properties over time is likely to be significantly less expensive than maintaining the stronger invariants that other indexes (e.g. kd-trees) require. Moreover, an R-tree works correctly even if the overlap areas are too large, though the query performance deteriorates.

There are two main challenges in extending an R-tree to moving points. For stationary points, a rectangle R_v can be represented by four real parameters, specifying the x- and y-coordinates of its bottom-left and top-right corners. But the smallest enclosing rectangle of moving points changes with time, and its shape can be quite complex. Figure 4 shows how the shape of the smallest interval containing a set of moving points changes with time. As proposed in [25, 32], this problem is circumvented by maintaining a rectangle $\tilde{R}_v$ at each node v such that $R_v(t) \subseteq \tilde{R}_v(t)$ for all values of t; see Figure 4. Another challenge in extending R-trees is partitioning the points of S_v among the children of v. Unlike the static case, the points have to be partitioned so that the overlap among the bounding rectangles of the children of v is small at all times. A trade-off between the query performance and the time spent in updating the tree can be obtained.

The first kinetic R-tree, called the TPR-tree, was proposed by Saltenis *et al.* [32]. It uses the notion of *time horizon* H in order to decide how to build the index at the current time. For example, in order to determine how the points of S_v should be partitioned among the children of v, it only considers the overlap among the bounding boxes for the time interval $[now, now + H]$. Similarly, it uses H to compute $\tilde{R}_v$. No kinetic events are defined, but the structure is updated due to external events such as changes in the trajectory, which is handled as a point deletion, followed by a point insertion. If the index is updated frequently, $\tilde{R}_v$ is a good approximation of R_v, but otherwise $\tilde{R}_v$ could be quite large, as evident in Figure 4 (i).

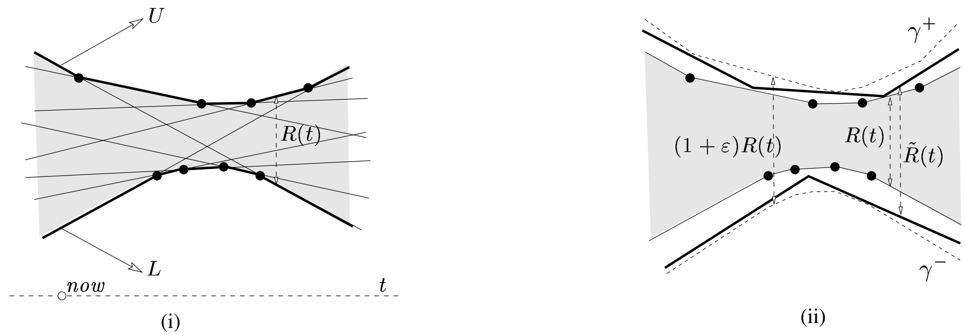


Figure 4: (i) $R(t)$ is the smallest interval containing $S(t)$. The rays U, L denote the approximation of $R(t)$ as computed in [32]. (ii) An ε-approximation of R by polygonal chains γ^-, γ^+ as computed in [25].

The second kinetic R-tree, called STAR-tree, was subsequently proposed by Procopiuc *et al.* [25]. Unlike the previous structure, they do not use the notion of time horizon. Instead they use kinetic events to update the index when the bounding boxes start overlapping a lot. Roughly speaking, if the bounding boxes of the children of a node v overlap considerably, it re-organizes the grand-children of v among the children of v. Using geometric approximation techniques developed in [9], it maintains a rectangle $\tilde{R}_v$ at each node, which is a close approximation of R_v. It provides a trade-off between the quality of $\tilde{R}_v$ and the complexity of the shape of $\tilde{R}_v$. For linear motion of S, the trajectories of the vertices of $\tilde{R}_v$ can be represented as polygonal chains. In order to guarantee that $\tilde{R}_v \subseteq (1+\varepsilon)R_v$, trajectories of the corners of $\tilde{R}_v$ need $O(1/\sqrt{\varepsilon})$ vertices. (Saltenis [32] represented the trajectory of each corner of $\tilde{R}_v$ by a ray.)

Hadjieleftheriou *et al.* [19] have proposed another kinetic R-tree, called PPR-tree, which also relies on storing an approximation of R_v at each node of the tree.

5 Nearest-Neighbor Queries

Given a set S of points in $\mathbb{R}^2$, the *Voronoi diagram* of S is the (maximal) planar subdivision in which the same point of S is nearest to all the points within a region. By preprocessing the Voronoi diagram of S for point-location queries [14], the point in S nearest to a query point can be computed efficiently. Although the Voronoi diagram of a set of moving points can be maintained efficiently, no efficient KDS for point-location structure is known.

Kollios et al. [20] describe a time-oblivious approach to answer nearest-neighbor queries (Q3 queries) for points moving along the x-axis. Although their index does not provide any bound on the worst-case query time, for linear motion of S, partition trees can be used to answer a nearest-neighbor query in $O(n^{1/2+\varepsilon})$ I/Os. Using the STAR-tree structure, Procopiuc *et al.* [25] describe branch-and-bound procedures that compute both exact and approximate k-nearest neighbors, for any $k \geq 1$. Although there are no theoretical bounds on the efficiency of the index, their approach has important practical value. They show through extensive experiments that the index can be expected to perform well for various data and velocity distributions.

Observing that the Euclidean metric can be approximated by a polygonal metric whose unit ball is a regular polygon with few edges, Agarwal *et al.* [1] developed an index for answering a δ-approximate nearest-neighbor query in $\mathbb{R}^2$, which returns a point of S whose distance from the query point at time t_q is at most $(1+\delta)$ times that to the nearest neighbor. Using $O(n/\sqrt{\delta})$ disk blocks, it answers a query in $O(n^{1/2+\varepsilon}/\sqrt{\delta})$ I/Os.

6 Discussion and Open Problems

Uncertainty. One important direction for further research is to incorporate uncertainty in the position and velocity of the input points. Even though the accuracy of high-end positioning systems is sufficient for most kinetic applications, commercially viable applications will have to contend with much lower accuracy systems. In addition, it is currently impractical to sample with high enough frequency so that all trajectory changes are noticed and transmitted to the indexing system. Instead, one has to sample the motion and contend with the accumulation of errors in the data.

Complex queries. Besides range and nearest-neighbor queries, many other types of queries are of practical interest. For example, report the areas of the highest density at some given time stamp, or detect when the number of data points in the neighborhood of a query point exceeds a certain threshold. The first type of query is important for traffic observation and control, while the second type has applications in location-based services.

Engineering issues. The techniques presented in this paper assume a unique database server that maintains the index on the kinetic objects. Clearly, the ideas should be extended to distributed databases and also address issues such as concurrency and data recovery. Although the research on both the front-end and back-end aspects of kinetic databases is still in an early stage, the pace of its evolution suggests that integrated solutions may not be too far in the future.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. Annu. ACM Sympos. Principles Database Syst.*, 2000. 175–186.

[2] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. *Journal of Computer and System Sciences*, 61:194–216, 2000.

[3] P. K. Agarwal, L. Arge, J. Erickson, and J. Vahrenhold. A time responsive indexing scheme for moving points. in preparation, 2002.

[4] P. K. Agarwal, L. Arge, and J. Vahrenhold. A time responsive indexing scheme for moving points. In *Proc. Workshop on Algorithms and Data Structures*, 2001.

[5] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.

[6] P. K. Agarwal, J. Erickson, and L. J. Guibas. Kinetic BSPs for intersecting segments and disjoint triangles. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 107–116, 1998.

[7] P. K. Agarwal, J. Gao, and L. Guibas. Kinetic medians and kd-trees. manuscript, 2002.

[8] P. K. Agarwal, L. J. Guibas, T. M. Murali, and J. S. Vitter. Cylindrical static and kinetic binary space partitions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 39–48, 1997.

[9] P. K. Agarwal and S. Har-Peled. Maintaining approximate extent measures of moving points. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, 2001.

[10] P. K. Agarwal and J. Matoušek. On range searching with semialgebraic sets. *Discrete Comput. Geom.*, 11:393–418, 1994.

[11] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symp. Principles of Database Systems*, pages 346–357, 1999.

[12] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 747–756, 1997.

[13] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.

[14] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[15] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38:86–124, 1989.

[16] L. J. Guibas. Kinetic data structures — a state of the art report. In P. K. Agarwal, L. E. Kavraki, and M. Mason, editors, *Proc. Workshop Algorithmic Found. Robot.*, pages 191–209. A. K. Peters, Wellesley, MA, 1998.

[17] O. Günther. The design of the cell tree: An object oriented index structure for geometric data bases. In *Proc. 5th IEEE Internat. Conf. Data Eng.*, pages 598–605, 1989.

[18] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Trans. Database Systems*, 25(1):1–42, 2000.

[19] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *Proc. VIII Conf. Extending Database Technology*, pages 251–268, 2002.

[20] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *Spatiotemporal Database Management*, pages 119–134, 1999.

[21] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. Annu. ACM Sympos. Principles Database Syst.*, pages 261–272, 1999.

[22] G. Kollios, V. J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Trans. Knowledge and Data Engineering*, 13:758–776, 2001.

[23] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.

[24] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving objects. In *Proc. International Conf. on Very Large Databases*, 2000.

[25] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. Star-tree: An efficent self-adjusting index for moving points. In *Proc. 4th Workshop on Algorithm Engineering and Experiments*, 2002.

[26] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[27] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.

[28] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proc. Intl Conf. Data Engineering*, pages 422–432, 1997.

[29] J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, pages 185–200, 1998.

[30] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[31] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[32] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 331–342, 2000.

[33] D. E. Willard. Polygon retrieval. *SIAM J. Comput.*, 11:149–165, 1982.

[34] A. C. Yao and F. F. Yao. A general approach to D-dimensional geometric queries. In *Proc. 17th Annu. ACM Sympos. Theory Comput.*, pages 163–168, 1985.

Towards Increasingly Update Efficient Moving-Object Indexing

Christian S. Jensen   Simonas Šaltenis
Department of Computer Science
Aalborg University
{csj,simas}@cs.auc.edu *

Abstract

Current moving-object indexing concentrates on point-objects capable of continuous movement in one-, two-, and three-dimensional Euclidean spaces, and most approaches are based on well-known, conventional spatial indices. Approaches that aim at indexing the current and anticipated future positions of moving objects generally must contend with very large update loads because of the agility of the objects indexed. At the same time, conventional spatial indices were often originally proposed in settings characterized by few updates and focus on query performance. In this paper, we characterize the challenge of moving-object indexing and discuss a range of techniques, the use of which may lead to better update performance.

1 Introduction

Several trends in hardware technologies combine to provide the enabling foundation for a class of mobile e-services where the locations of the moving objects play a central role. These trends encompass continued advances in the *miniaturization* of electronics, in *display devices*, and in *wireless communications*. Other trends include the improved *performance* of general computing technologies and the general improvement in the *performance/price ratio* of electronics. Perhaps most importantly, *positioning technologies* such as GPS (global positioning system) are becoming increasingly accurate and usable.

The coming years are expected to witness very large quantities of wirelessly on-line, i.e., Internet-worked, objects that are location-enabled and capable of movement to varying degrees. Example objects include consumers using WAP-enabled mobile-phone terminals and personal digital assistants and vehicles with computing and navigation equipment. Some predict that each of us will soon have approximately 100 on-line objects, most of which will be special-purpose, embedded computers.

These developments pave the way to a range of qualitatively new types of Internet-based services, which either make little sense or are of limited interest in the traditional context of fixed-location, PC- or workstation-based computing. Such services encompass traffic coordination, management, and way-finding; location-aware

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

* The authors' research is supported in part by the Danish National Centre for IT Research, the Nordic Academy of Advanced Study, and the Nykredit Corporation.

advertising; integrated information services, e.g., tourist services; safety-related services; and location-based games that merge virtual and physical spaces.

In location-enabled m-services, moving objects disclose their positional data (position, speed, velocity, etc.) to the services, which in turn use this and other information to provide specific functionality. Our focus is on location-enabled services that rely on access to the up-to-date locations of large volumes of moving objects. Due to the volumes of data, the data must be assumed to be disk resident; and to obtain adequate query performance, some form of indexing must be employed. The aim of indexing is to make it possible for multiple users to concurrently and efficiently retrieve desired data from very large databases. Indexing techniques are becoming increasingly important because the improvement in the rate of transfer of data between disk and main memory cannot keep pace with the growth in storage capacities and processor speeds. Disk I/O is becoming an increasingly pronounced bottleneck.

The techniques that have been proposed for indexing the current and near-future positions of moving objects are based on spatial indexing techniques, most prominently the R-tree, that were conceived in settings where updates were relatively few and where focus was on queries. Therefore, efficient update processing represents a substantial, and as yet unmet, challenge. Specifically, without more efficient update processing the applicability of moving-object indexing techniques will remain restricted to scenarios with relatively few objects or scenarios where few updates are needed per object per time unit.

This paper briefly describes a range of techniques that seem to offer additional opportunities for further improving the update processing capabilities of moving-object indices. We initially describe a setting for moving-object indexing. Then follows a section that considers six classes of techniques that may be applied to make a variety of moving-object indices more update efficient. A brief summary ends the paper.

2 Problem Setting

The application setting for the problem of moving-object indexing descibed here aims to concisely capture the complexities of the problem. We have at times opted for a concrete setting as opposed to a very general and abstract setting.

At the core of the problem setting is a set of so-called *moving objects*, which are capable of continuous movement. We assume that the movements of these objects are embedded in two-dimensional space, meaning that we ignore altitude. As examples, the moving objects can be pedestrians or people traveling in cars.

We also assume that one or more *services*, each with a database, are available to the moving objects. The moving objects are capable of communicating wirelessly with the services. Further, the moving objects are capable of reporting their movement information, including and most prominently their current position, to the services. This capability is achieved by means of one of a range of geo-location technologies. Indeed, we are particularly interested in the class of services where a moving object reports its movement information to the service. Such a location-enabled service records the movement of each object. It may record the past, current, and projected, future movement.

A service's record of an object's movement is inherently imprecise, for several reasons. First, the movement of an object is captured via some kind of sampling, so that movement information pertaining only to discrete instances of time is obtained. Movement information pertaining to all other times must be derived via some kind of interpolation or extrapolation. Second, the movement information in each sample is imprecise [7].

Different geo-location technologies yield different precisions, and the precisions obtained when using a single technology also varies, depending on the circumstances under which the technology is used. For example, the cellular infrastructure itself, the positioning technologies offered by companies such as SnapTrack and Cambridge Positioning Technologies, and GPS and server-assisted GPS offer quite different precisions. And, for example, GPS technology is dependent on lines of sight to several satellites, which affects the robustness of the technology. In other words, the accuracy of the positioning is highly dependent on a number of aspects, including the user's location.

Different services require movement information with different minimum precisions for them to work. For example, a weather information service requires user positions with very low precision, while an advanced location-based game, where the participants interact with geo-located, virtual objects, requires high precision. Stated in general terms, the highest precision that may be obtained is that offered by the geo-location technology. One may get close to the highest precision by sampling very frequently. Services that require higher precision cannot be accommodated. We will assume that a required precision is given by the service under consideration and that this precision can indeed be achieved. Further, each object is assumed to be aware of the movement information kept for it by the service. The object is then able to issue an update to the service when its actual movement information deviates by more than the required precision from the service's record [11].

A set of geo-referenced objects not capable of continuous movement is also part of the problem setting. Examples include schools, recreational facilities, and gas stations. These objects are part of the "content" queried by the services.

It should also be noted that the objects are not simply moving in a perfect, two-dimensional Euclidean space. Rather, objects are subjected to movement constraints, one category of which consists of objects that block the movements of objects. For example, a private property and a lake block the movement of a hiker. Another category consists of infrastructure that intuitively takes the movement of an object to a lower-dimensional space. For example, cars may be confined to a road network.

The workload experienced by the database at a service then consists of a sequence of updates intermixed with queries. The amount of updates is dependent on factors such as the number of objects, the required precision, the agility of the objects, and the service's representation of the objects' movements. The queries may be range queries, ranked or unranked k-nearest neighbor queries, and reverse nearest neighbor queries, to name but a few. In addition, these queries may be attached to moving objects, making them moving queries; and they may be active, meaning that they are evaluated continuously for a time period. It is assumed that the data at a service is stored on disk and that some kind of indexing is necessary to support the queries issued.

3 Update Techniques

We proceed to explore a range of techniques, the application of which may reduce the processing needed to accommodate the index updates implied by a workload. These techniques aim to process the individual updates more efficiently or to reduce the number of updates.

Representing Positions as Functions One approach to reduce the number of updates is to model the position of an object as a function of time. This reduces the need for updates because a function better estimates the real locations of an object for a longer period than does a constant position. An update is needed when the difference between the real location of the object and the position believed by the database (the "database position") exceeds the threshold dictated by the services being supported. This is illustrated in Figure 1, where the movement of a one-dimensional moving point is shown together with linear functions representing the object's movement between the updates. When linear functions are used, for a time point t, the database position of an object $\bar{x}(t) = \bar{x}(t_{upd}) + \bar{v}(t_{upd})(t - t_{upd})$ is described by two parameters—the position of the object, $\bar{x}(t_{upd})$, as recorded at the time of the last update t_{upd} ($t_{upd} \leq t$), and the velocity vector, $\bar{v}(t_{upd})$, as recorded at t_{upd}.

The velocity vector is the first derivative of the position function. Similarly, the acceleration vector is the second derivative of the position. In general, one could employ $n + 1$ parameters and n-th degree polynomials to approximate an object's movement, using the Taylor series:

$$\bar{x}(t) = \bar{x}(t_{upd}) + \bar{x}'(t_{upd})(t - t_{upd}) + \bar{x}''(t_{upd})\frac{(t - t_{upd})^2}{2!} + \cdots + \bar{x}^{(n)}(t_{upd})\frac{(t - t_{upd})^n}{n!}$$

Above, it is assumed that at any point in time, there is one function that models the position of a moving object in the index. Another direction involves the use of several functions at a time for capturing an object's position. Specifically, the time interval during which an object's position is modeled may be partitioned into sub-intervals, and one function may be given for each sub-interval. It is natural to require that the position given by a function at the end of its interval is equal to the position obtained from the function assigned to the next interval when applied to the start of this interval. If all functions are linear, a polyline in $n+1$-dimensional space results for an object moving in n-dimensional space. This direction may be useful when the path of an object in a transportation network is known.

Several researchers have explored the use of linear functions for representing and indexing the current and future positions of moving objects in one-dimensional to three-dimensional spaces [4, 8, 9, 10]. The more general approaches outlined here have yet to be explored, as does the combined indexing of the past, current, and future positions of moving objects.

Capturing and Using Expiration Times Independently of how an object's position is represented in an index, the accuracy of the database position and, thus, its utility for any application will tend to decrease as time passes. When an object has not reported its position to the database for a certain period of time, the database position is likely to be of little use to the services being supported, and the object is also unlikely to be interested in the services. The object's position should then simply be discarded from the database and the index.

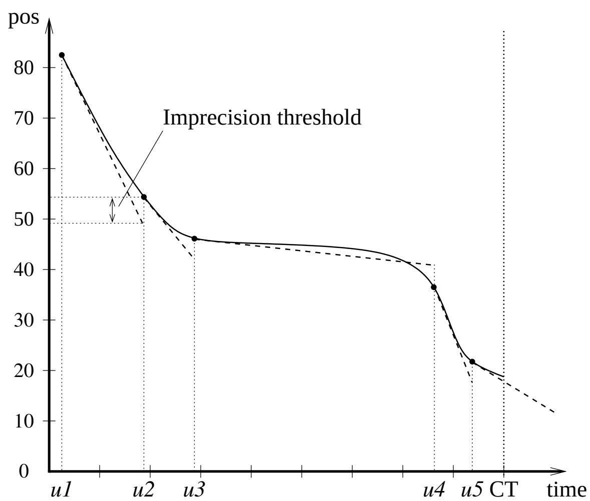


Figure 1: Approximating a One-Dimensional Moving Object by Linear Functions

In this scenario, it is natural to associate an expiration time with each position, thus enabling automatical removal of "expired" positions as well as filtering of yet-to-be-removed, expired entries in queries. As a result, the scheduling of explicit deletion operations and the reporting of expired objects in query results are avoided.

When expiration times are associated with object positions at the times they are inserted or updated, an opportunity exists for the index used to exploit the expiration times to obtain more efficient update processing. Indeed, experiments with the R$^{\mathrm{EXP}}$-tree [10] demonstrate that in order to avoid scheduling of explicit deletion operations to remove expired objects, expiration times can be introduced in the index structure, upon which the expired index entries can be removed in an automatic and lazy fashion. This reduces the average update costs. While the R$^{\mathrm{EXP}}$-tree introduces expiration times in the TPR-tree, the proposed technique is more general. It can be used with any other index structure.

Utilization of Buffering A promising technique for increasing the efficiency of index update operations is to use the ideas underlying Buffer trees [2], the aim being to remedy the inefficiencies of transferring blocks with little useful data between main memory and disk. In a Buffer tree, each node is associated with a main-memory-sized buffer that buffers the update operations concerning the subtree rooted at the node. The operations buffered at some tree node are performed in bulk whenever the node's buffer becomes overfull. With B being the number of (data value, pointer)-pairs that fit in a disk block, such bulk operations are almost B times faster, in amortized cost, than when performing the operations in the usual fashion. With B usually being on the order of hundreds, the performance of update operations may be boosted dramatically.

While some work has been done on buffering in R-trees in the context of workloads that intermix insertions and deletions [3], the existing Buffer-tree approaches do not allow queries to be performed if there are nonempty buffers. however, workloads that intermix queries and updates is a key characteristic of our application scenario, where queries should also be answered in a timely, on-line fashion, i.e., it is not acceptable to buffer queries the same way insertions and deletions are buffered.

It should be possible to extend the Buffer-tree techniques to support queries that are performed simultaneously with insertions and deletions. One approach to achieving this is to organize the buffers as index-like structures, so that queries can be efficiently performed on the buffers. This may reduce the performance of the queries, as they will have to perform additional I/O operations to search the relevant buffers. The goal is to provide a flexible mechanism, whereby one can tune how much the query performance should be sacrificed in order to gain the necessary update performance.

Exploitation of All Available Main Memory Main memory storage is much faster than disk storage and becomes increasingly voluminous and inexpensive. More efficient processing of updates against moving-object indices may be obtained through aggressive use of all available main memory. While buffering tends to result in the use of more main memory, simply using buffering does not imply that all available main memory is being utilized in any optimal fashion.

When aggressively using main memory, one may expect much of an inherently disk-based index to reside in main memory, in a form that is optimized for the particular main memory and processor environment. The challenge then becomes one of maintaining some parts of an index in its disk-based format and some parts in its main-memory format. Such main-memory variants of disk-based moving-object indices deserve study.

The areas of main-memory and real-time database management aggressively exploit main memory and may have ideas to offer. For example, while main-memory page buffers are traditionally organized simply as collections of pages, main-memory databases employ more elaborate index structures to optimize CPU performance. Aggressive use of main memory is also seen in an application area such as telecommunications, where there is a need to record large amounts of real-time discrete events.

Taking Movement Constraints Into Account Existing work on moving-object indexing typically assumes that the underlying objects live in 1-, 2-, or 3-dimensional Euclidean spaces (e.g., [1, 5, 8, 9]).

However, as pointed out earlier, in many situations, some objects constrain the locations of other objects. For example, the movements of hikers in a forest are constrained by fenced, private properties. As another example, the movement of a ship is constrained by shallow water and land. These blocking objects do not reduce the dimensionality of the space in which the objects are embedded. In other application contexts, the locations of objects are constrained to a transportation network embedded in, typically, 2-dimensional Euclidean space [4]. This in some sense reduces the dimensionality of the space available for movement—the term 1.5-dimensional space has been used. Examples abound. Cars typically move in transportation networks, and the destinations such as private residences, hotels, and shops may be given locations in transportation networks. Next, folklore has it that 80-90% of all automobile drivers move towards a destination. This suggests that drivers typically follow network paths that are known ahead of time.

Moving-object indices should be able to exploit these constraints to obtain better update performance. This may be achieved by exploiting the constraints to better represent the moving objects' locations in the indices and to better estimate the positions of the objects, both of which then lead to less frequent updates.

It should be noted that Euclidean distances are either not the only interesting notions of distance or are not of interest at all in these settings. Rather, indexing techniques that apply to settings with obstacles or network-constrained objects must contend with other notions of distance. For some such notions, the distance between two stationary objects varies over time.

Using Application Semantics It may be observed that modifications on moving-object indices have special properties that may be exploited: most modifications occur in the form of updates that combine deletions with insertions, and the newly reported, inserted location for an object is relatively close to its previously reported, deleted location. In the indexing literature, the insertion and deletion operations are usually considered as separate operations. However, in our application scenario, most update operations are deletion-insertion pairs. Processing these pairs as two separate operations results in two descents and two (partial) ascents of the index tree. For a portion of updates, especially those that change the updated data only slightly, the combined deletion-insertion operation can be performed in a single descent and a (partial) ascent of the tree, which may save I/O operations [6].

4 Summary

With the continued proliferation of wireless networks, visionaries predict that the Internet will soon extend to many billions of devices, or objects. A substantial fraction of these will offer their changing positions to the location-enabled services, they either use or support. As a result, software technologies that enable the management of the positions of objects capable of continuous movement are in increasingly high demand.

This paper argues that although indexing of the current and anticipated, future locations of moving objects is needed, there exists an as of yet unmet need for more efficient update processing in moving-object indexing. The paper then proceeds to describe a number of possible techniques, the use of which may render moving-object indices increasingly update efficient.

Many opportunities exist for testing out more specific incarnations of the described techniques in the contexts of concrete indexes. In addition, techniques not described in this paper exist that may also improve update efficiency. Distributed update processing is one such type of technique. Another promising direction is to exploit approximation techniques to offer monotonically improving, as-good-as-possible answers to queries within specified soft or hard deadlines. This may enable the querying of almost up-to-date data, which in turn may reduce the need for prompt updates. Yet another direction involves the use of relaxed index properties.

References

[1] P. K. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. *Proc. of the PODS Conf.*, pp. 175–186 (2000).

[2] L. Arge. External-Memory Algorithms with Applications in GIS. In *Algorithmic Foundations of Geographic Information Systems*, LNCS 1340, pp. 213–254 (1997).

[3] L. Arge, K. Hinrichs, J. Vahrenhold, J. S. Vitter. Efficient Bulk Operations on Dynamic R-trees. In *Proc. of Algorithm Engineering and Experimentation*, pp. 328–348 (1999).

[4] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. *Proc. of the PODS Conf.*, pp. 261–272 (1999).

[5] G. Kollios et al. Indexing Animated Objects Using Spatiotemporal Access Methods. TimeCenter TR-54 (2001).

[6] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. *Proc. of the 3rd Intl. Conf. on Mobile Data Management*, pp. 113–120 (2002).

[7] D. Pfoser and C. S. Jensen. Capturing the Uncertainty of Moving-Object Representations. In *the Proc. of the SSDBM Conf.*, pp. 111–132 (1999).

[8] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects. Manuscript (2001).

[9] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. *Proc. of the ACM SIGMOD Conf.*, pp. 331–342 (2000).

[10] S. Šaltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *Proc. of the IEEE International Conf. on Data Engineering*, pp. 463–472 (2002).

[11] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Distributed and Parallel Databases* 7(3): 257–387 (1999).

Data Management for Moving Objects

Hae Don Chon, Divyakant Agrawal, and Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
{hdchon,agrawal,amr}@cs.ucsb.edu

Abstract

We model a moving object as a sizable physical entity equipped with GPS, wireless communication capability, and a computer such as a PDA. Furthermore, we have observed that a real trajectory of a moving object is the result of interactions among moving objects in the system yielding a polyline instead of a line segment. In this paper, we first choose a partitioning approach to efficiently manage such trajectory information and develop a system called RouteManager based on a space-time grid that manages moving objects in a one-dimensional space. We then extend this to two-dimensional space. The time-dependent shortest path problem is an interesting application where such a system can be used.

1 Introduction

Global Positioning System (GPS) has been widely accepted as the technology for locating objects such as vehicles on a highway or soldiers in a battlefield. One of the features GPS provides, other than locating, is speed tracking so that we can get speed information without integrating GPS to a speedometer in the vehicle. Wireless communications technology such as Cellular Digital Packet Data (CDPD) [6] and Mobile IP [12] has also gained popularity. With these technologies in place, finding and managing the current locations of moving objects have become possible. Since keeping track of the location of many, continuously moving objects is hard to achieve, there are significant data management problems that need to be addressed. The focus of much of the research on this issue has been on how to manage location information efficiently without updating the location information every time it moves. One basic and fundamental observation on moving objects is that if a moving object maintains a constant speed for a certain period, future locations of a moving object can be modeled as a linear function of time during that period. Then the location information can be managed by maintaining the parameters of the linear functions instead of constant updating [18, 8, 2]. There is no known alternative to this approach so far. We also base our research on this observation and make further observations: (1) Moving objects with a few exceptions follow certain paths such as a route on a highway system. (2) Such paths on which moving objects move have some physical limitations such as the number of lanes and/or route conditions. (3) Due to the limitations, a real trajectory of a moving object is a polyline instead of a line segment. In Section 3, we explain how to manage such polyline information.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Among many possible applications that could benefit from this research is *Time-Dependent Shortest Path Problems*. In transportation applications, *Time-Dependent Shortest Path Problems*, where the cost (travel time) of edges between nodes varies with *time*, are of significant interest [11]. Traffic congestion is a huge problem, especially in metropolitan areas. According to a recent report from the Texas Transportation Institute [17], the total congestion cost for the 68 major U.S. urban areas in 1999 was $78 billion, and Los Angeles was ranked first ($12.5 billion) in the congestion cost as well as other congestion measures. In recent years, in order to avoid traffic congestion a growing number of vehicles are equipped with a navigation system which provides a shortest path to their destination as well as other features like roadside assistance [1]. The navigation system uses a digital map, which includes static information such as the distance of a path and the speed limits along the way. However, such information does not reflect the dynamically changing traffic information, hence could lead vehicles to incorrect shortest paths. In contrast to this static approach, if there is a way to obtain the cost in real time and then apply a time-dependent shortest path algorithm, it would result in a better solution for the shortest path queries. The system we develop in this paper could be used to gather this dynamically changing information in order to run a time-dependent shortest path algorithm.

In the following section, we discuss why we choose the partitioning approach over the indexing approach to manage moving object information. Also, more detailed assumptions and observations on moving objects are given in Section 2. In Section 3, we explain the system we have developed using the partitioning approach. The system is initially designed in the context of moving objects in a one-dimensional space. Later, in Section 4, we argue the importance of the time-dependent shortest path problems and describe how we can handle moving objects in a two-dimensional space using multiple one-dimensional components. We conclude the paper with a discussion of future direction of research in this area in Section 5.

2 Indexing vs. Partitioning

In general, there are two main paradigms for managing large data sets: the indexing approach and the partitioning approach. The indexing approach is a bottom up approach where data objects are clustered together and index structure is built to efficiently locate the individual data items. This approach is usually more appropriate for sparse unstructured data sets. Alternatively, the partitioning approach a priori partitions the data space and uses simple (often constant time) methods for locating data. This approach is usually more appropriate for uniformly distributed data with well defined structure [7]. In this section, we discuss why the partitioning approach is more appropriate for managing moving object information than the indexing approach.

We start by stating in detail some assumptions regarding our model of moving objects. The best example of moving objects is vehicles on a road although it is not the only example. The moving objects considered are not particles, meaning that they do not move irregularly such as in a Brownian motion. Instead, they follow a certain path such as a route on a highway system. This is true with few exceptions such as soldiers in a battlefield or persons in a large open field. In addition, they are sizable physical entities with the following characteristics: capable of locating their position, capable of wireless communication, and equipped with a computer such as a PDA. Although there are several other technologies to locate an object, GPS is by far the dominant technology and it is getting better and cheaper [4]. As a result, we expect that in the future many (if not all) moving objects, especially vehicles, will be equipped with a GPS. Wireless communication technologies such as CDPD or IEEE 802.11b can be used together with a PDA or laptop to connect to a control center, which manages data and answers queries to either moving objects or some other interested party such as advertisers.

Let us now consider how to model moving objects' current and future locations. Future locations of a moving object can be modeled as a linear function of time if the moving object maintains a constant speed for a certain period. Therefore, the trajectory of a moving object can be expressed in terms of parameters representing a linear function. In addition to this linearity, we make further observations on the paths (space) on which moving objects move: there are physical limitations of paths on which moving objects move such as the number of lanes, route conditions, etc. Based on our everyday experience, we can expect that as the number of vehicles in a section of a route increases, the average speed of the vehicles decreases. This is supported by a report from the Texas Transportation Institute [17]. The report observed that as the daily traffic volume (the average number of vehicles observed in a day at a location) increases, the average speed of vehicles decreases. Due to this physical limitation, the trajectory of a moving object is a result of interactions among all moving objects present in the system. Consequently, the resulting trajectory would be a polyline, a sequence of linear lines. For this reason, if we are to manage future location information about moving objects on a network of routes, the routes need to be broken into smaller sections, which then are associated with information such as the number of objects and the maximum velocity allowed in the section. Regardless of what approaches (indexing or partitioning) are used, we need at least to maintain such information in each smaller section. Moreover, as we will see in Section 3, there are times when an insertion of a moving object causes a series of updates of other moving objects in its way. Therefore, each section needs to be associated with the list of moving objects in the section as well. For these reasons, we take the partitioning approach in that we partition the domain space into a grid. Each cell in the grid corresponds to a certain period of time and a section. In addition, each cell is associated with a list of moving objects and the maximum velocity allowed in the cell. To our knowledge, this approach has not been previously explored for the management of dynamic objects. In the next section, we explain the space-time partitioning model and describe how insertion works.

3 Space-Time Grid

In the previous section, we argued that the partitioning approach is more appropriate for managing moving objects' trajectory information. In this section, we summarize the model we have developed [3] for maintaining dynamically changing information about moving objects assuming that a moving object moves on a one-dimensional path.

3.1 Data Structures, and Definitions

We assume that each section of a route is associated with a list of moving objects in the section and a predefined function of the maximum velocity at which moving objects can move. This maximum velocity function associated with a section will return the maximum velocity a moving object can move in the section depending on the number of moving objects currently in the section. We can derive the function from statistical data [5]. We partition the domain space $T \times Y$, where T corresponds to the time domain and Y corresponds to the space domain, into an array of cells such that $Y = [y_0, y_1) \cup ... \cup [y_{p-2}, y_{p-1}) \cup [y_{p-1}, y_{max}]$ and $T = [t_0, t_1) \cup [t_1, t_2) \cup ... \cup [t_{q-1}, t_q)$, where $t_q = t_0 + \Delta T$. A cell $g[i, j]$ is defined as a rectangle $[t_i, t_{i+1}) \times [y_j, y_{j+1})$, for some $t_i \in T, y_j \in Y$. Note that although time is increasing indefinitely, we only maintain the information about some predefined ΔT period of time. As time evolves, we shift the time domain forward. We also assume that on entry to the system, a moving object m is responsible to provide initial information such as the starting location (s), the starting time (t_s), the destination (e), and the intended velocity (v), which is the maximum velocity at which it intends to move. When a moving object m appears in the system providing its initial information, it is inserted into the system as a polyline. Shortly, we will explain how those information becomes a polyline.

Choosing appropriate data structures is critical to achieve efficient storage and retrieval of information. We use the following data structures in the implementation: A linked list with an extra pointer to the last element is used to represent the polyline corresponding to a moving object's trajectory. The entire grid is implemented as an array of cells, which efficiently supports random accesses to individual cells. An individual cell stores only the identifiers of intersected moving objects along with the velocity of the moving objects in the cell using a skip list [15]. Since we do not store the entire information about a moving object in each cell, we need to store it

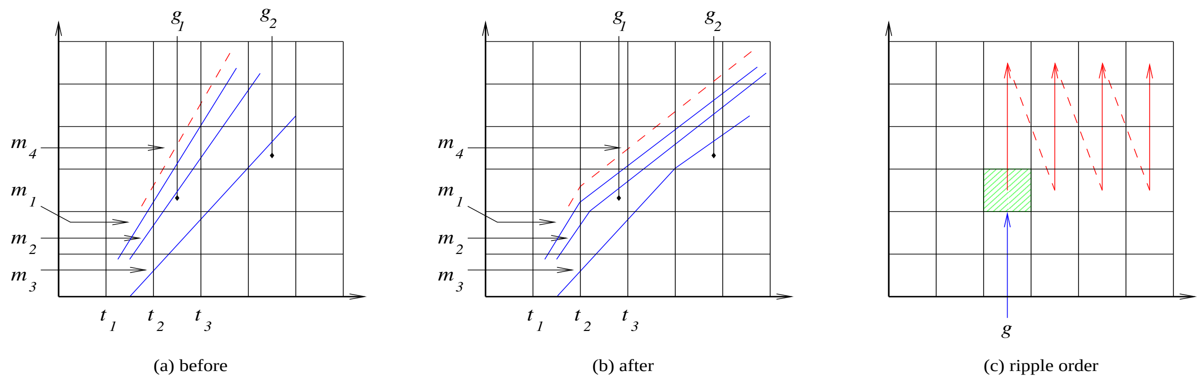


Figure 1: Ripple Effect

somewhere else, which we call an *Object Store*. A hashtable is used for the Object Store, which needs to support frequent random accesses.

Before we explain the insertion algorithm, we introduce a notation $\triangleright$ and a term *entry point* as follows: If the corresponding polyline of m intersects with a cell g, m is said to intersect with g and denoted by $m \triangleright g$. An entry point of m to a cell g is a point where m's corresponding polyline and g intersect for the first time.

3.2 Object Insertion

We now explain how to insert an object m into the system. With the initial information $m = (s, t_s, e, v)$, we find the first cell g (skip list) such that $m \triangleright g$ and repeat the following steps: we compute the maximum velocity allowed in g which depends on the number of moving objects in the cell, and adjust m's velocity accordingly. When there is a velocity change, we add the corresponding entry point to the linked list associated with m which represents a polyline. We then find the next cell to intersect by computing the entry point to the cell, and insert its identifier along with the adjusted velocity into g. We repeat these steps until we process the cell to which the destination e of m belongs. In that case, we complete the insertion process by inserting m into the Object Store, and return the polyline information to m.

There are, however, cases where one insertion may cause a series of updates. Consider the situation shown in Figure 1(a) where three moving objects, $\{m_1, m_2, m_3\}$, are in the system and a new one, m_4, is about to be inserted. Assuming that three moving objects is a threshold for moving objects to reduce their velocities to a certain degree, cell g_1 is about to have three objects so that existing objects as well as the new one will need to reduce their velocities based on the maximum velocity function associated with g_1. Furthermore, the updates of m_1 and m_2 cause the cell g_2 to have three moving objects. As a result, another update on m_3 has to be executed. Figure 1(b) shows the result of the insertion of m_4. A moving object m_4 in this case is said to cause a ripple effect. For this reason, we need to check if the number of objects in g reaches a threshold, before inserting a moving object m into g, by checking $f_g(|g|) > f_g(|g|+1)$, where $f_g(n)$ is the maximum velocity function associated with g. If it is true, an algorithm called *RippleInsert* is invoked with a (row, column) pair of g. Otherwise continue the insertion of m until it is finished.

The RippleInsert algorithm works as follows: suppose g is the first cell whose maximum velocity needs to be changed because of the insertion of a moving object. The information about all moving objects in g is then adjusted according to the new velocity allowed in g and continue this process with the cells whose time and location intervals are greater than or equal to those of g. Other cells need not to be processed since an insertion would not cause any update to the past and to previous locations. Figure 1 (c) shows the order in which cells are updated. The details of this algorithm can be found in [3].

With the data structures and this insertion algorithm in place, we can process range and k-nearest neighbor queries [5]. In the query processing algorithms we proposed, the target objects are moving as well as the query points. We do not consider the case where the target objects are stationary. If so, after getting the future or current location information of a moving object, the query becomes a traditional query (range or k-nearest neighbors), for which many algorithms have been proposed.

4 Time Dependent Shortest Paths

In the previous section, we developed a system that manages moving object information assuming that the moving objects are restricted to move in a one-dimensional space. In the real world, however, this would not be a realistic assumption. Rather, objects move in a two-dimensional space. More precisely, a moving object follows a certain path, which can be viewed as a path on a network of one-dimensional lines in a two-dimensional space [5]. Therefore, we need to extend the system to handle mobility in two-dimensional space. In this section, we explain the extension of the system and describe how it can be used for time-dependent shortest path problems.

Shortest path problems are among the most studied network flow problems and one of the most important application areas is transportation. In transportation applications, *Time-Dependent Shortest Path Problems*, where the cost (travel time) of edges between nodes varies with *time*, are of significant interest [11]. As in transportation applications, where the cost changes dynamically, the correctness of the shortest path very much depends upon the correctness of the cost model. For example, typical drivers choose a route based on static information that, in general, is heavily weighted on the length of the route. Unfortunately, as many drivers in metropolitan areas are aware, superiority of a route cannot entirely be based on the length of the route. In particular, current and future condition of routes such as the severity of congestion should play an important role in determining which route would incur minimum delay. In this case, using only the static information to find a shortest path is not enough. We need to take other variables into consideration.

In general, the time-dependent cost (normally, travel time) is given arbitrarily [10, 9, 11]. We noticed that this time-dependent shortest path problem would be an interesting application for the data management of moving objects. Once we extend the system to two-dimensional space, we will be able to provide the dynamically changing cost to a time-dependent shortest path algorithm [11] leading many vehicles to real shortest paths. This will result in better utilization of the road network and efficient traffic management.

We now extend the system to a two-dimensional space. Note first that any path in a network of one-dimensional lines in a two-dimensional space can be divided into subpaths, each of which belongs to a one-dimensional line. Therefore we can build the two-dimensional system as a network of one-dimensional subsystems. The one-dimensional system we developed in the previous section is associated with a route and we call it the RouteManager. Hence, each RouteManager maintains a local view of the corresponding route. A local view consists of the dynamically changing cost information for sections on the route. Based on these local views, a frontend interface to moving objects, PathServer, maintains a global view of the network as a directed graph. In this directed graph, each section on a route corresponds to an edge. When a moving object appears in the system, the PathServer finds a time-dependent shortest path on the graph, pushes it into a queue along with the path information, and returns it to the moving object. The queue is constantly checked by a thread, DistributionManager. If it is not empty, an element (a pair of an id of a moving object and the corresponding 2D path) is dequeued and broken into 1D subpaths. The DistributionManager then distributes the 1D subpaths to corresponding RouteManagers. For PathServer to keep up-to-date information about the routes, each RouteManager constantly sends an updated local view to the PathServer. Then a thread in the PathServer, PathManager, maintains up-to-date information in the graph structure. Hence, we are able to reuse the system (RouteManager) we developed in the previous section and take advantage of parallelism. Note that a ripple effect can occur in any RouteManager after DistributionManager distributes 1D subpaths to the corresponding RouteManagers. Once a ripple effect occurred, the moving objects affected will arrive at subsequent routes later than the routes antici-

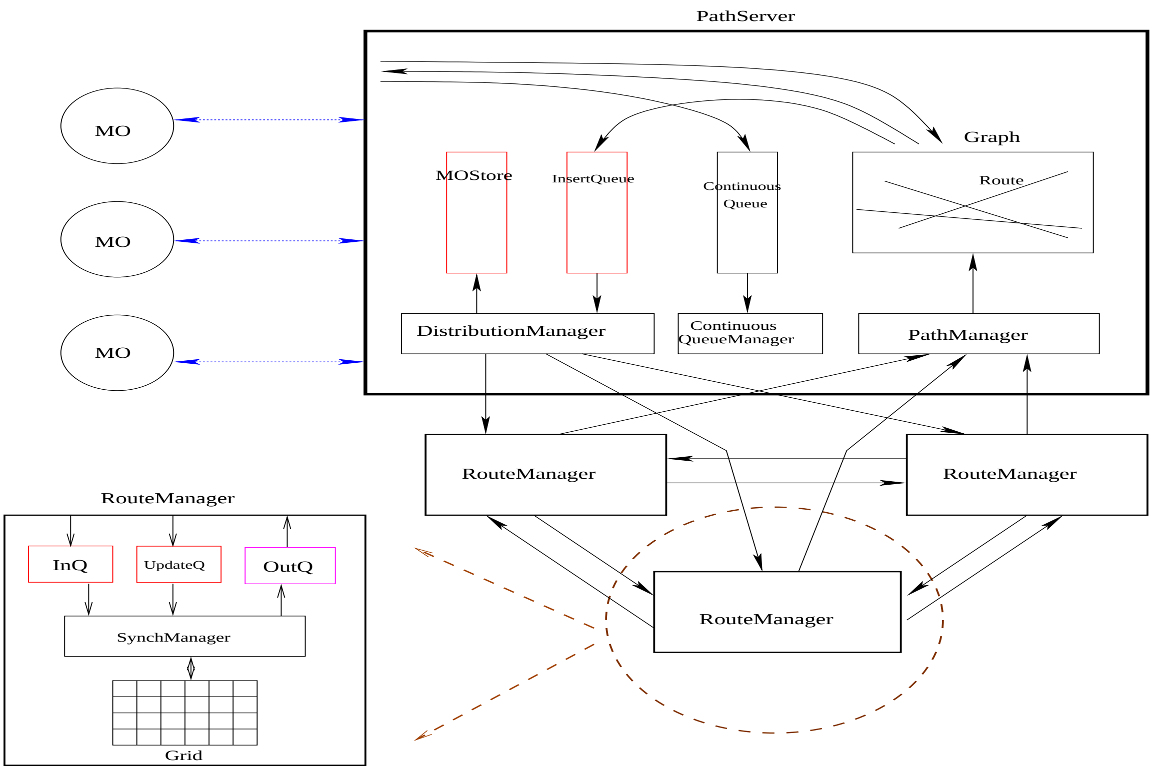


Figure 2: System Architecture

pate. Due to this, each RouteManager needs to communicate each other exchanging up-to-date information. We accomplish this by developing a SynchManager with an outgoing queue (OutQ) and an incoming queue (UpdateQ). The system is being developed using Java. PathServer and RouteManager are RMI Servers, possibly on different machines and any other Managers are threads. Figure 2 shows an overview of the system architecture.

5 Conclusion and Future Direction

The moving objects we have modeled in this paper are sizable physical entities following certain paths. Such paths have certain physical limitations and due to those limitations, the partitioning approach is better than the indexing approach to manage moving object information. In addition, we have observed that a real trajectory of a moving object is the result of interactions among moving objects in the system yielding a polyline instead of a line segment. We then built a system (RouteManager) based on a space-time grid that manages moving objects in a one-dimensional 1D space. Since in the real world moving objects would move on a two-dimensional network of one-dimensional lines, we extend the system to two-dimensional space as a network of one-dimensional components. The time-dependent shortest problem is an interesting application where such a system can be used. Also, it is of significant interest in transportation applications.

Since Sistla et al. [18] proposed a query model for moving objects, there has been much work in this field. Most of the early work [19, 21, 13, 8, 16, 2] did not take the following fact into account: the trajectory of a moving object is the result of the interactions among other moving objects yielding a polyline as a trajectory. Recently, however, Pfoser et al. [14], Chon et al. [3, 5], and Vazirgiannis et al. [20] worked on the assumption that the trajectory of a moving object is a polyline. One difference between [14] and [5, 20] is that in [14] the polyline is the result of recording of a moving object's past locations and only past information can be queried instead of the future. We believe that this assumption will be the basis for future research that will address issues of managing and querying future locations of moving objects. We also believe that the time-dependent shortest problem is a very interesting problem. Since several promising approaches have been proposed in the context of mobile data management, the research community needs to develop ways to evaluate different techniques. Also, research prototypes need to be build to determine the effectiveness of proposed applications.

Acknowledgements

This research was partially supported by the NSF under grant numbers EIA-0080134, EIA98-18320, IIS98-17432, and IIS99-70700 and the Caltrans-Testbed Center For Interoperability (TCFI).

References

[1] OnStar. http://www.onstar.com.

[2] H. Chon, D. Agrawal, and A. El Abbadi. Storage and Retrieval of Moving Objects. In *Proceedings of the Int. Conf. on Mobile Data Management*, pages 173–184, 2001.

[3] H. Chon, D. Agrawal, and A. El Abbadi. Using Space-Time Grid for Efficient Management of Moving Objects. In *MobiDE*, pages 59–65, May 2001.

[4] H. Chon, D. Agrawal, and A. El Abbadi. NAPA: Nearest Available Parking lot Application. In *Proceedings of the Int. Conf. on Data Engineering*, pages 496–497, 2002.

[5] H. Chon, D. Agrawal, and A. El Abbadi. Query Processing for Moving Objects with Space-Time Grid Storage Model. In *Proceedings of the Int. Conf. on Mobile Data Management*, 2002.

[6] C. Forum. Cellular Digital Packet Data System Specification, Release 1.1. Technical report, Jan. 1995.

[7] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 3(2):170–231, 1998.

[8] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Moving Objects. In *Proceedings of ACM Symp. on Principles of Database Systems*, pages 261–272, 1999.

[9] K. Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83:154–166, 1995.

[10] A. Orda and R. Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.

[11] S. Pallottino and M. Scutella. Shortest path algorithms in transportation models: classical and innovative aspects. In *In Equilibrium and Advanced Transportation Modelling, Kluwer*, pages 245–281, 1998.

[12] C. E. Perkins. Mobile IP. *IEEE Communications Magazine*, pages 84–99, May 1997.

[13] D. Pfoser and C. Jensen. Capturing the Uncertainty of Moving-Object Representations. In *Proc. of the SSDBM Conf.*, pages 111–132, 1999.

[14] D. Pfoser, C. Jensen, and Y. Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. In *Proceedings of the Int. Conf. on Very Large Data Bases*, pages 395–406, 2000.

[15] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of ACM*, 33(6):668–676, 1990.

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

[17] D. Schrank and T. Lomax. The 2001 Urban Mobility Report. Technical report, Texas Transportation Institute, 2001.

[18] A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proceedings of the Int. Conf. on Data Engineering*, pages 422–432, 1997.

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

[20] M. Vazirgiannis and O. Wolfson. A Spatiotemporal Model and Language for Moving Objects on Road Networks. In *7th International Symposium on Spatial and Temporal Databases*, pages 20–35, July 2001.

[21] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pages 111–122, 1998.

LOCUS: A Testbed for Dynamic Spatial Indexing

Jussi Myllymaki James Kaufman
IBM Almaden Research Center
{jussi,kaufman}@almaden.ibm.com

Abstract

We describe an extensible performance evaluation testbed for dynamic spatial indexing that is directly geared towards Location-Based Services (LBS). The testbed exercises a spatial index with several query types relevant for LBS: proximity queries (range queries), k-nearest neighbor queries, and sorted-distance queries. Performance metrics are defined to quantify the cost (elapsed time) of location updates, spatial queries, spatial index creation and maintenance.

The testbed is extensible with new spatial indexing methods, new query generators, and new index visualization methods. Synthetic but realistic, three-dimensional "moving object" data is generated with CitySimulator, a toolkit we have developed specifically for LBS research and made available on the IBM alphaWorks developer Web site.

1 Introduction

The DynaMark benchmark for dynamic spatial indexing is designed to evaluate the performance of middleware and database servers supporting Location-Based Services (LBS) [6]. Commonly referred to as "moving object databases," these systems manage continuously changing data representing the current, past, or future locations of mobile users. The high update load required of databases supporting Location-Based Services defines a new set of database requirements driving the need for new performance evaluation criteria. Queries typical of these systems include "find current position of user X," "find objects within distance Y from user X" and "find 10 nearest objects to user X."

In this paper, we describe the LOCUS Dynamic Spatial Indexing Testbed that provides a convenient way to conduct performance experiments on dynamic spatial indexing methods. The testbed was directly motivated by the need to evaluate the performance of systems supporting LBS applications. The testbed is designed to be extensible so as to easily include new spatial indexing methods, new location data sources (both simulated and real), new spatial query types, and new index visualization methods. It has been adapted to several commercial spatial data management systems as well as prototype systems, and includes a proximity query generator, a k-nearest neighbor query generator, and a sorted distance query generator for measuring spatial query performance.

The testbed is complemented by CitySimulator, a scalable data source that simulates a three-dimensional model city. CitySimulator generates realistic location data for large populations of mobile users. Combining the testbed with simulated location data, one can evaluate the performance of LBS middleware serving up to millions of moving objects. The system provides a powerful graphical user interface that allows the user to

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

control several aspects of the simulation at runtime, including traffic flow, traffic congestion, and blocked streets and routes.

The paper is organized as follows. In Section 2, we review existing literature on location management, dynamic spatial indexing, dataset generation, and performance benchmarking. In Section 3, we briefly describe salient features of the DynaMark benchmark, and then introduce the LOCUS testbed in Section 4. Methods for analyzing the performance and scalability of spatial data management systems are discussed in Section 5.

2 Related Work

Location-based services build upon two well-established areas of information technology: wireless communication and spatial data management. While the primary role of wireless communication is to provide a connection between a mobile subscriber and others, its proper function – routing of data packets and voice calls – requires knowledge about the position of each mobile subscriber to some degree of accuracy.

New applications for real-time location information are now emerging. These applications are largely driven by the expanding use of mobile phones, but also the United States Federal Communications Commission's (FCC) mandate that the originating location of all emergency 911 calls from mobile phones in the United States be determined with some minimum accuracy [3]. Today, mobile phone operators have a choice of half a dozen technologies for location determination, each with distinct power consumption, handset compatibility, time-to-first-fix, and in-building coverage characteristics.

A spatial index for location-based services contains a very large number of simple spatial objects (e.g. points) that are frequently updated. These "moving object databases" pose new challenges to spatial data management [15]. The workload is characterized by high index update loads and relatively simple but frequent queries (e.g. range queries). A location update may merely contain the position of a user but it may also include the user's trajectory (direction and speed over time) [10, 7]. Supporting trajectories adds additional requirements to the index and query scheme [1]. A location update may also expire at a certain point in time [9] and is inherently imprecise due to inaccuracies in location determination technologies.

Several benchmarks exist for transaction processing in relational databases (TPC benchmarks), object-relational databases (BUCKY), and object-oriented databases (OO7), but to our knowledge no standard benchmarks exist for LBS applications and dynamic spatial indexing. The SEQUOIA 2000 benchmark [11] was motivated by Earth Sciences and focused on retrieval performance over large datasets of complex spatial objects. Several spatial access methods were compared in [5], but apart from an initial insertion cost, the benchmark focused on query performance. A qualitative comparison of spatio-temporal indexing methods appears in [13].

Use of realistic datasets is critical to gaining meaningful data from performance evaluation experiments [2]. Most LBS experiments to date have used spatial data generated by random walks or algorithms using simple distribution functions. For instance, [5] looked at six random distributions for spatial data: diagonal, sinus, bit, x-parallel, clusters, and uniform distributions. A few generators have been proposed for creating spatio-temporal data. The GSTD generator [14] creates data for randomly moving points and rectangles. In [12], GSTD was extended to account for buildings and other "infrastructural" obstructions, but it still allowed objects to move free of influence from other moving objects.

The Oporto toolkit [8] describes an elaborate scheme with moving point objects (fishing boats), moving extended objects (schools of fish), stationary extended objects that change size (areas with plenty of plankton that attract fish, and storms that boats try to avoid), and static extended objects (harbors). Another approach for creating semantic moving object datasets is described in [9]. The dataset was created by distributing a set of destinations uniformly in a coordinate space and connecting every pair of destinations with two one-way roads. Moving objects (cars) are placed on roads randomly and move with a randomly assigned speed until a destination is reached, at which point a new destination is picked.

3 DynaMark Benchmark for Dynamic Spatial Indexing

The DynaMark benchmark measures the performance and scalability of a spatial data management system [6]. The benchmark executes a set of standard spatial queries against a set of standard location trace files. Performance metrics consist of the cost of updating a user's location in the spatial index and the cost of spatial queries against the index. The spatial queries are motivated by LBS applications but are similar to those found in traditional Geographic Information Systems (GIS). The difference is that the data being queried is highly dynamic and the queries are issued extremely frequently and have a dynamic reference point (a user's location).

The size of an individual benchmark run is determined by the number of mobile users contained in the location trace file and ranges from 10,000 to one million or above. Our trace files are generated by CitySimulator, a toolkit we have developed specifically for LBS research (see Section 4.4 for details). Each record in the location trace file represents a location update that contains the following information: object ID (identifies mobile user), timestamp (indicates the time when the location was determined or reported by the user), and X, Y, and Z coordinates of the location. The X and Y coordinates represent the longitude and latitude values of the location, and Z indicates elevation in meters.

The benchmark defines three types of queries: proximity queries, k-nearest neighbor (kNN) queries, and sorted-distance queries. These queries are typically centered around the location of a user issuing the request. A *proximity query* finds all objects that are within a certain range. The range condition may vary along different axes hence the query forms a 2D ellipse or 3D ellipsoid depending upon the topological dimension of the space. Alternatively, the range can be expressed by forming a 2D rectangle or 3D ortho-rhombus.

A *kNN query* finds the k nearest objects. The query may search other mobile users or it may search stationary objects such as coffee shops, gas stations, hospitals, etc.

A *sorted-distance query* lists each object in increasing distance order relative to the reference point. This is like a k-nearest neighbor query where k is the total number of objects, but the difference is that during query evaluation the number of objects inspected by the application is unknown (otherwise the system could execute it efficiently as a kNN query). Also, the result must include distance information.

4 LOCUS Dynamic Spatial Indexing Testbed

The LOCUS testbed provides a convenient way to run performance experiments on spatial data management systems and yield performance data that conform to the DynaMark benchmark specification. The testbed is written in portable C and has been tested on several platforms, including Windows, AIX, and Solaris. We are currently in the process of making the testbed code available on the IBM alphaWorks developer Web site.

4.1 Extensibility

The testbed is extensible with new spatial indexing methods, new query generators, and new index visualization methods (Figure 1). The testbed defines a C API for each of the extension types, using function pointer arrays to achieve something analogous to the Java "interface" concept.

The task of an *indexing method extension* is to provide a spatial indexing capability and support API calls such as "create index," "delete index," "insert index entry," and "search index." In Section 4.3 we list several indexing method extensions that are built into the default testbed. We distinguish between two types of indexing method extensions: native extensions and adapter extensions. A native indexing method extension implements the required spatial indexing capability by itself, without relying on a database system. An adapter extension converts the C API calls to calls in another system, for example SQL statements that run against a database.

A *query generator* creates one or more spatial queries, typically using the location of an existing user as a reference. The testbed has three built-in query generators, corresponding to the query types defined by the benchmark: proximity queries, k-nearest neighbor queries, and sorted-distance queries. The parameters needed

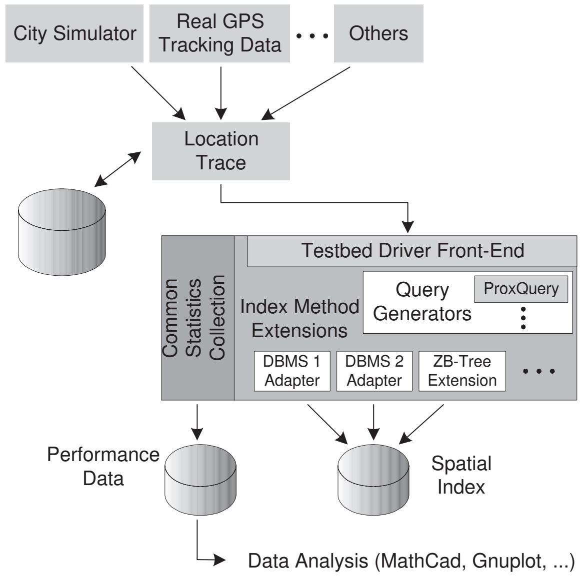


Figure 1: Architecture of LOCUS testbed.

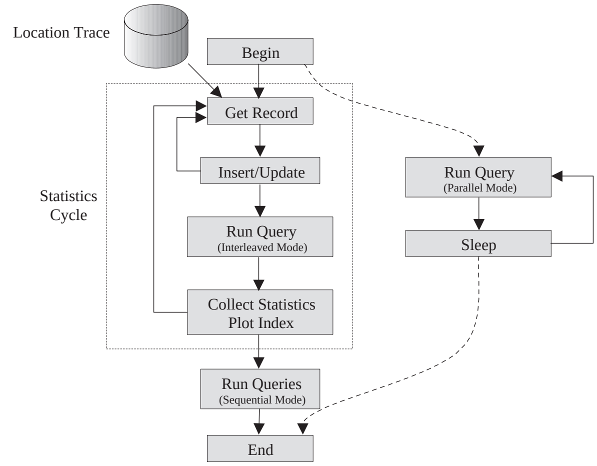


Figure 2: Update and query flow in the testbed.

by the query generators (e.g. the range value in proximity queries or the value of k in kNN queries) are defined in a configuration file. The same query generator code works in all query modes defined earlier.

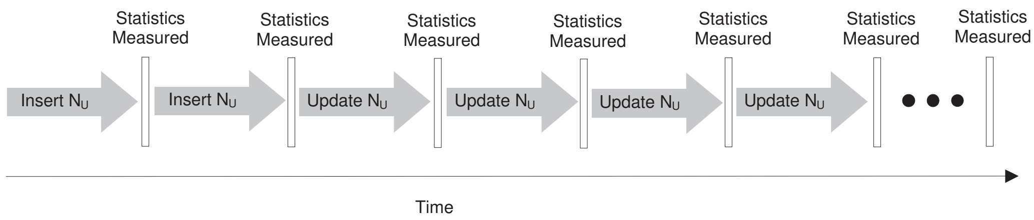
An *index visualization method* provides a simple way for the testbed to plot an index (e.g. minimum bounding rectangles of an R-tree index) using lines and rectangles, multiple colors or line types, and labels. Several index visualization methods are built into the testbed by default. The "gnuplot" visualization method produces files in GIF, Postscript, DXF, and other graphics formats. Using external tools, static GIF images can be converted to animations that show the evolution of the index as a function of time. An "OGS" visualization method outputs ST_Polygon geometries that can be loaded into any OpenGIS-compatible spatial database and analyzed with its visual data exploration tools. The testbed also provides a "generic" visualization method that outputs the index geometries as comma-separated values (CSV).

4.2 Update and Query Flow

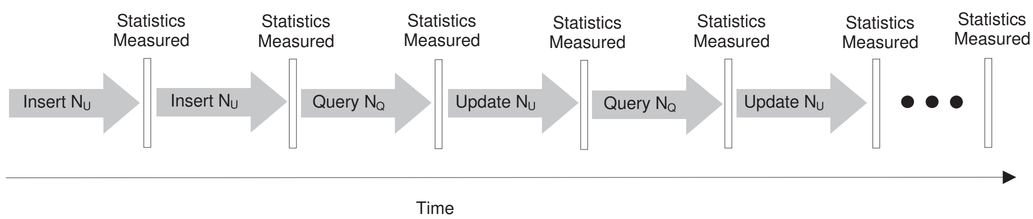
Figure 2 illustrates the general process flow of the testbed. The basic principle is that after every n updates to the index, the testbed runs m queries. The process repeats until the entire location trace file has been processed, or until a maximum experiment time has been reached.

The number of updates n and queries m executed in an iteration depend on the *query mode*. In the *immediate mode*, one spatial query ($m = 1$) is issued after every update ($n = 1$), with the position of the update being used as the reference point. In the *interleaved mode*, the values n and m are configured to some values N_U and N_Q, respectively. After processing N_U updates, the testbed runs N_Q queries where for each query a different, random user is picked as the reference point. In the *sequential mode*, the index is "loaded" by processing the first iteration (initial locations) of the location trace file and then queried by N_Q queries as in the interleaved mode. The *parallel mode* executes queries simultaneously with updates. A time parameter specifies the frequency of queries, and for each query a random user is picked as the reference point.

We generally prefer to run experiments in the interleaved mode because it provides a way to gather consistent, repeatable update and query cost measurements as the index grows and updates are performed. The execution plan in the interleaved mode is further illustrated in Figure 3. To minimize the impact of queries on updates, we conduct two experiments for each population size and indexing method. In an UPDATE experiment,



A) Flow of UPDATE Experiment



B) Flow of QUERY Experiment

Figure 3: Execution plan for updates, queries, and statistics collection in the testbed.

only updates are performed and no queries are executed. This gives us the minimum response time of individual index updates. In a QUERY experiment, we run updates and queries in the interleaved mode, but only inspect query performance data. Again, this is to minimize the interference between updates and queries. In Section 4.5 we describe how the independently-measured update and query cost metrics are combined into a total cost estimate and extrapolated to different population sizes to estimate maximum population sizes supported by the system under test.

4.3 Built-In Indexing Method Extensions

Most commercial database systems can be extended with a spatial data component, either provided by the vendor itself or by a third party. Some of the more popular spatial indexing methods these extensions employ include grid indices, R-trees, and Quadtrees (see [4] for a comprehensive survey). Some commercial solutions also make use of space-filling curves such as Z-order curves and Hilbert curves. It is fairly straightforward to use one of these existing spatial indexing methods for LBS applications. However, most of them were not designed to support the high update frequencies required by LBS applications. In fact, some of them perform poorly when faced with dynamically changing spatial data. Clearly, a controlled comparison between various spatial indexing schemes is required to assure satisfactory performance in LBS applications.

The testbed has been adapted to use most commercial database systems and their spatial extensions, including IBM DB2 with Spatial Extender, Informix Dynamic Server with Spatial DataBlade, Oracle 9i with Spatial Cartridge, and ESRI ArcSDE. We have also developed a generic Z-order indexing method that runs on top of any SQL database system.

The testbed has also been adapted to work with main-memory indexing methods. Two main-memory extensions are included by default: a naive array extension and a ZB-tree extension. The array extension implements a combination of a simple array and hash table which work together to record the position of moving objects. Updates into this data structure are extremely fast but queries obviously perform poorly. In contrast, the ZB-tree calculates Z-order values in the extension code and stores them in a binary tree. We used a readily available implementation of binary trees, namely the `libavl` library that implements several varieties of them. The

ZB-tree extension stores Z-order values in a right-threaded AVL-tree which performs very well for proximity queries (requiring only a left-to-right traversal of the leaf nodes of the tree).

4.4 Location Simulation

Location trace files for the testbed are produced by CitySimulator, a complementary tool available for download at the IBM alphaWorks developer Web site at http://alphaworks.ibm.com. CitySimulator is a scalable, three-dimensional model city that simulates an arbitrary number of mobile users moving about in a city, driving on streets or walking on sidewalks and entering buildings, where they move up and down the floors and stay on a floor some amount of time before returning to the streets and finding a new place to go to (or even leaving the city). We typically simulate population sizes ranging from 10,000 users up to 1 million users and use an average location report periodicity of 30 seconds.

Simulation control parameters allow easy creation of realistic events such as daily commutes. A graphical user interface allows observation of the simulation. Advanced settings allow exploration of efficiency of indexing algorithms over daily commute cycles. The output of the simulation can be viewed in real time; output is also produced as a comma-separated variable (CSV) text file, which contains a unique person ID, a time stamp, and spatial coordinates.

Custom city plans can be created as XML documents and imported into the simulator. The simulator also provides Java interfaces and abstract classes to facilitate extensions capable of generating city plans from, for example, real GIS data. A Java properties file allows easy configuration of a wide variety of program parameters. The GUI allows change of parameters affecting motion rules in real time. The simulator also includes an optional traffic flow model which causes the traffic flow to take on the characteristics of a compressible fluid. When this feature is enabled, obstructions to traffic flow can cause shock waves or traffic jams.

Simulated cities are constructed with collections of various Java objects. The high level objects in the simulation are “Place” and “Person.” Objects of type Building, Road, Intersection, and Floor all extend Place. Places have several attributes: coordinates, extents, altitude or floor number, and pointers to neighboring places. Places also contain enter and exit probabilities, up down probabilities, drift probabilities (on roads), scatter probabilities (on intersections), etc.

People move according to rules based on the place they are in. A person on a building floor does a random walk. At specific points (stairways) they may move up or down (if not at the top floor or ground floor respectively) or leave a building (if near a door). A person on a road moves with a linear combination of *Velocity = Random Walk Component + Drift Velocity*. Hence the magnitude of the drift velocity increases as a person moves closer to the center of a road. For two-way streets, the direction of the drift velocity changes sign at the center (so on one side of a road people move North or East, on the other side they move South or West). Road objects also have orientations that determine if the drift velocity is North/South or East West.

4.5 Methods for Performance Analysis

To estimate the maximum population size supported by a spatial data management system, we fitted a function $f(P) = a \cdot P + b$ to the measured update and query costs of the system. Parameter P is population size and $f(P)$ is the estimated cost of the database update or query for that population size. We then produced a closed-form equation that relates parameters a and b to P and i, where i is the average request interval in a real-time location tracking application. We assume that the workload consists of a sequence of database requests where a given request is an update with probability p and query with probability $1-p$. In a system where moving objects report their location every 5 minutes and every update is followed by a query, the values are $i = 300$ and $p = 0.5$.

In order for a location tracking application to function in real time, it holds that the total number of requests handled in time period i must have a lower overall cost than i itself. The closed-form equation representing this requirement is $P \leq \sqrt{b^2/4a^2 + ip/a} - b/2a$.

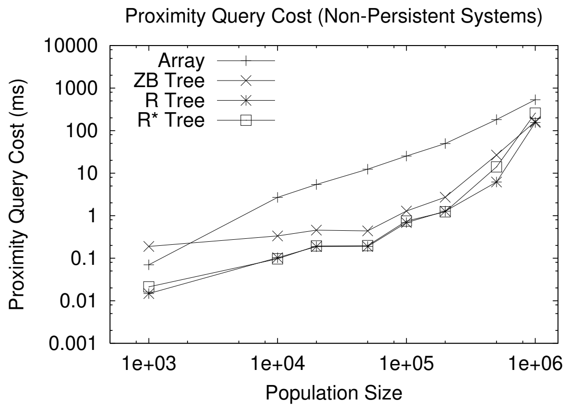


Figure 4: Actual cost of proximity queries for main-memory indexing methods.

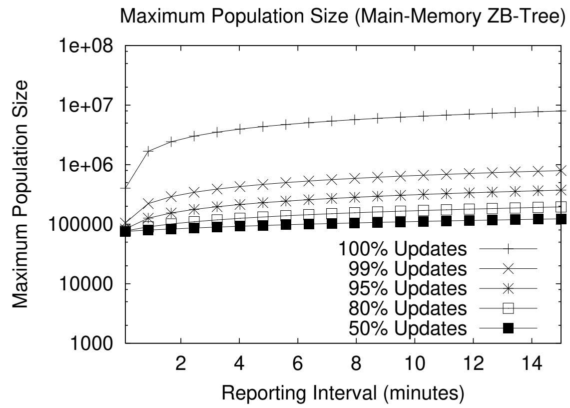


Figure 5: Maximum population size supported by a main-memory ZB-tree.

In Figure 4, we show performance results gathered by the testbed on several main-memory indexing methods running on a 1 GHz, 1 GB RAM server. The two variations of R-trees we tested performed best on all population sizes, although with the highest population size (1 million moving objects) the advantage appeared to all but evaporate. The ZB-tree performed almost equally well, except for the smallest population sizes. As expected, the naive array/hash table method was 1-2 orders of magnitude more expensive than the other methods.

Figure 5 shows the maximum population size supported by the main-memory ZB-tree method under various update/query workloads. When the workload consisted of updates only and each moving object reported its location every 5 minutes, the indexing method could handle roughly 5 million moving objects. The high capacity was made possible by the low update cost of the ZB-tree shown in our experiments, less than 50 microseconds per update. The maximum population size dropped quickly as the fraction of queries increased. With a 99% update ratio, maximum population size was 0.5 million, and with 95% updates it dropped to 0.3 million. With higher query loads the maximum population size settled at the 100,000 to 250,000 range.

5 Conclusion

We have described the LOCUS testbed for dynamic spatial indexing that is a generic platform for running controlled performance experiments on any spatial indexing method and query generator. The testbed is easy to extend with new spatial indexing algorithms and query types. The testbed currently supports three spatial query types common to LBS applications, and provides adapters for several commercial database systems and their spatial extensions. Native indexing methods that do not use a database system, e.g. memory-resident tree indices, can also be plugged into the testbed. The testbed also provides an index plotting capability for visualizing the internal structure of a spatial index (e.g. minimum bounding boxes of an R-tree) and producing portable GIF animations that are useful for analysis and development.

The testbed is complemented by CitySimulator, a scalable, three-dimensional model city that generates realistic location data for large populations of mobile users. It provides a powerful graphical user interface that allows the user to control several aspects of the simulation at runtime, including traffic flow, traffic congestion, and blocked streets and routes. A visualization display shows the progress of the simulation in real time. New city models can be imported as XML documents which describe the layout of the city, including roads, buildings, and vacant space.

References

[1] Pankaj K. Agarwal, Lars Arge, and Jeff Erickson. Indexing moving points. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 175–186, 2000.

[2] Thomas Brinkhoff. Generating network-based moving objects. In *Proceedings of the International Conference on Statistical and Scientific Database Management (SSDBM)*, pages 253–255, Berlin, Germany, July 2000.

[3] FCC Docket 94-102. FCC adopts rules to implement enhanced 911 for wireless services, June 1996. http://www.fcc.gov/e911.

[4] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[5] Hans-Peter Kriegel, Michael Schiwietz, Ralf Schneider, and Bernhard Seeger. Performance comparison of point and spatial access methods. In *Proceedings of the International Symposium on the Design and Implementation of Large Spatial Databases (SSD)*, pages 89–114, Santa Barbara, CA, 1989.

[6] Jussi Myllymaki and James Kaufman. DynaMark: A benchmark for dynamic spatial indexing, March 2002. Submitted for publication.

[7] Dieter Pfoser and Christian S. Jensen. Querying the trajectories of on-line mobile objects. In *Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access*, Santa Barbara, CA, May 2001.

[8] Jean-Marc Saglio and Jose Moreira. Oporto: A realistic scenario generator for moving objects. In *Proceedings of the International Workshop on Database and Expert Systems Applications (DEXA)*, pages 426–432, 1999.

[9] Simonas Saltenis and Christian S. Jensen. Indexing of moving objects for location-based services. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, San Jose, CA, February 2002.

[10] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and querying moving objects. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 422–432, Birmingham, UK, April 1997.

[11] Michael Stonebraker, James Frew, Kenn Gardels, and Jeff Meredith. The SEQUOIA 2000 storage benchmark. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, pages 2–11, Washington, DC, June 1993.

[12] Yannis Theodoridis. Generating semantics-based trajectories of moving objects. In *Proceedings of the International Workshop on Emerging Technologies for Geo-Based Applications*, Ascona, Switzerland, May 2000.

[13] Yannis Theodoridis, Timos K. Sellis, Apostolos Papadopoulos, and Yannis Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *Proceedings of the International Conference on Statistical and Scientific Database Management (SSDBM)*, pages 123–132, 1998.

[14] Yannis Theodoridis, Jefferson R. O. Silva, and Mario A. Nascimento. On the generation of spatiotemporal datasets. In *Proceedings of the International Symposium on Spatial Databases (SSD)*, pages 147–164, Hong Kong, China, July 1999.

[15] Ouri Wolfson, Bo Xu, Sam Chamberlain, and Liqin Jiang. Moving objects databases: Issues and solutions. In *Proceedings of the International Conference on Statistical and Scientific Database Management (SSDBM)*, pages 111–122, Capri, Italy, July 1998.

Spatio-Temporal Indexing in Oracle: Issues and Challenges

Ravi Kanth V. Kothuri and Siva Ravada
Spatial Technologies, NEDC
Oracle Corporation, Nashua, NH 03062, USA
{Ravi.Kothuri, Siva.Ravada}@oracle.com

Abstract

Spatio-temporal indexing has become an increasingly important area of database research. In this paper, we describe how such data can be indexed using current features of Oracle Spatial such as Spatial indexing, Linear Referencing System, and Oracle's function-based indexes. We examine the issues and limitations of this approach and possible future enhancements. This approach could be used as a first guide to implement most spatio-temporal applications such as mobile-object tracking.

1 Introduction

With the rapid growth in the wireless industry and the advances in cellular technology, mobility is increasingly becoming an important aspect of data objects and queries. Consequently, the spatial database research community has recently proposed new data models, defined meaningful query semantics, and devised efficient search and access methods for non-static moving spatial data.

The motion is modeled along the temporal dimension. The notion of the ever-increasing current time poses a variety of challenges when time and space are combined. We may distinguish between two paradigms for research contributions: those that concern "historical" or predictable-route data, where the motion is more or less pre-determined, and those that concern "on-going-motion" data, where only the current velocity is known.

Since the location of an object needs to be updated after every time instant, on-going-motion data are usually modeled in parametric space in terms of the time and velocity of an object, which is assumed to be unchanging (and triggers an update whenever there is a change). Several proposals have been made in the research literature for indexing such data [4, 9, 11].

However, in most mobile applications, the start and destination of an object's motion can be pre-determined using the route computed in the in-car navigation system, and the location of the object at any time can be estimated roughly by using the speed limits on the route. In other applications, data capturing the past motion of an object is considered. Several research solutions [6, 10] have been proposed for such "historical" motion data.

In this paper, we describe how historical motion data can be indexed using existing technologies of Oracle. This solution should be reasonably efficient for most applications, and it extends readily the salient features of commercial databases, such as backup and recovery, concurrency, and replication, to spatio-temporal data.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 Modeling Spatio-Temporal Data

We proceed to briefly describe the Oracle Spatial data model and how it supports linear referencing. Spatio-temporal data can be modeled as a direct application of linear-referenced spatial data.

Oracle Spatial models 2–4 dimensional spatial data using an *sdo_geometry* data type. For the 2-dimensional case, this data type models all the spatial data types defined by the Open GIS Consortium (OGC) and caters to most data occurring in GIS and CAD/CAM applications. The supported spatial data types includes simple primitive elements such as points, lines, curves, polygons (with and without holes), and complex elements that are made up of a combination of primitive elements. The sdo_geometry data type is implemented as an Oracle object datatype. This approach extends all the benefits of Oracle's object-relational database technology to spatial data, including replication.

In the above geometry model, Oracle Spatial allows numerical attributes to be associated with vertices of a linear feature (a line geometry). That is, in addition to storing a spatial (latitude, longitude) coordinate, each vertex can also contain two other numerical attributes referred to as the *measure* attributes. Some transportation customers use these measure attributes to represent the mile-post markers, traffic conditions, or speed-limits when the line string data represents a road network. Whereas indexing is supported on the 2-dimensional spatial coordinate data, linear referencing system (LRS) functions operate on the measure attributes of the geometry data. Some LRS functions are listed below. This linear-referencing system combines powerful spatial indexing capability with functional operations on linear measures and has been very useful in the transportation and other related industries.

- start_measure(g) and end_measure(g): return the starting and ending measure of an LRS geometry segment, respectively.
- clip_geometry(g, t) (or dynamic segmentation): clips a geometry segment at the specified interval of measures.
- split_geometry(g, measure): splits a geometry into two segments at specified measure value.

The linear referencing system can be used to support spatio-temporal data in Oracle.

For moving data points, the line string geometry can model the trajectory of the object and one of the measure attributes could represent the temporal dimension. Figure 1 shows an example. The first two dimensions denote the (x, y)-dimensions, and the third dimension denotes time. Identifying the trajectory of the object between times 10 and 20 can be performed by calling the "clipping" function of the spatial linear referencing system. This model can easily be modified to include the enhancements from partially-persistent R-trees [5] where a moving object's trajectory is modeled as different geometries.

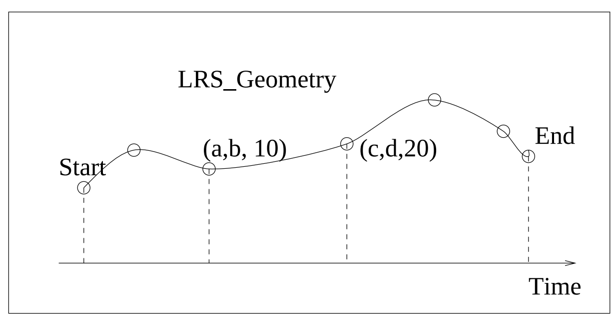


Figure 1: Example of a Spatio-Temporal Geometry Modeled Using an LRS_geometry

3 Indexing Spatio-Temporal Data

Although spatio-temporal data can be stored with both the spatial and temporal dimensions together as a single object, there is no explicit indexing support for spatio-temporal data in Oracle Spatial. However, both spatial and temporal dimensions can be extracted from the data without replicating the data and indexed using separate indexes. This approach is described in Figure 2.

During spatial indexing, the spatial dimensions are implicitly extracted from the spatio-temporal geometry and used in indexing.

The temporal index, on the other hand, is constructed by explicitly extracting the temporal aspect of the spatio-temporal geometry. This temporal aspect can be either a separate geometry, which is indexed by a 1-dimensional spatial R-tree index, or it can be the start and end measures of the geometries, which are indexed by "function-based" B-tree indexes. In either case, the function-based indexes in Oracle allow indexes to be constructed on functions of column values (in this case, functions returning the temporal aspect) without explicitly storing the function values. This means that the spatio-temporal geometry need not be redundantly stored/replicated, which helps ensure data integrity. Updates need only be performed on the spatio-temporal geometry column of the database table, and both (spatial and temporal) indexes are updated automatically by the database engine.

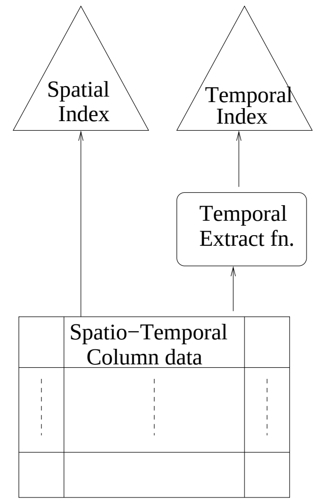


Figure 2: Indexing Spatio-Temporal Data in Oracle

We procced to describe spatial indexes in more detail. Temporal data can be indexed similarly, using either spatial indexes or by B-tree indexes.

Oracle Spatial indexes 2-dimensional spatial data using either a quadtree or an R-tree. These indexes are implemented using the extensible indexing framework of Oracle [1, 7]. This framework allows for the creation of new domain-specific indexes and associated query operators and provides for the integration of user-specified query, update and index creation routines inside the Oracle server. To query the constructed spatial indexes, new SQL-level predicates, referred to as *operators*, are defined. These operators can be included in the "where" clause of a SQL statement to select data that satisfy a specified predicate with respect to a specified query window. Such operators are executed using index-associated procedures for query processing and allow for incremental processing of queries (see [1, 7, 8] for more details). Oracle Spatial provides the following query operators on 2-dimensional spatial data:

- sdo_filter query: returns geometries whose index approximations intersect those of the 2-d spatial query window.
- sdo_relate query: returns geometries that interact with a query window. Different types of interaction (most common being "intersects," and "inside"), based on the 9-intersection model [2, 3] can be specified. This operator is equivalent to the *ST_relate* operator of SQL/MM. (Most SQL/MM suggested methods are available in some form in Oracle Spatial.)
- sdo_within_distance (or within-distance) queries: identify geometries that are within a specified distance from the query geometry.
- sdo_nn (nearest-neighbor) queries: identify the k nearest neighbors for a specified query geometry.

Combined with the functions defined in the LRS to operate on the temporal dimension aspect of the geometry, these operators provide effective query capabilities on spatio-temporal data.

4 Spatio-Temporal Queries

A spatio-temporal query consists of a spatial query window (an arbitrary 2-d geometry) and a time interval. Alternately, a more complicated query in the form of an LRS geometry where different times are associated with different vertices, can be specified. However, this is not a common case, and we will discuss these limitations at the end.

Spatio-temporal queries can be categorized broadly into three categories [6]. First, in *spatial-range, temporal-range* queries, data geometries that satisfy both the spatial and temporal ranges are returned. Examples of such a query include identifying all cars that pass through a toll-booth during a specified time interval. Second, in *spatial-range, temporal-knn* queries, the k data geometries that satisfy the spatial range and are closest to the specified time interval are returned. Examples of such a query include identifying the most recent cars that passed through an accident site. Third, in *spatial-knn, temporal-range* queries, the k data geometries that satisfy the temporal range and are closest to the specified query window in spatial dimension are returned. Examples of such a query include identifying the closest restaurants for a moving object at the current time.

Next we describe how each of the above three query types could be answered using Oracle Spatial. The syntax of Oracle Spatial functions used next is simplified for clarity.

Spatio-Temporal Range Query This query is quite easy to implement using the spatial index operators and the measure functions of LRS. If q is the 2-d spatial window, and s, e delimit the time interval, the following query returns the ids of the result geometries. In the query, the sdo_relate operator retrieves the geometries that satisfy the query window q and compares their start and end measures with the specified time interval.

```
select gid from geom_tab a
where sdo_relate(a.geom, q, 'intersects')= 'TRUE'
  and sdo_lrs.start_measure(a.geom) <= e and sdo_lrs.end_measure(a.geom) >= s;
```

Spatial-Range Temporal-knn Query This query is also easy to implement using the spatial index operators and the measure functions of LRS as follows: First, geometries intersecting the spatial query window are retrieved from the database using the sdo_relate operator. Next, the intersecting portions of the resulting LRS data geometries are obtained using the sdo_intersection function. This function uses the query window q to return the intersecting portions of the resulting data geometries. Then the start and end measures are obtained from the intersecting LRS geometries. In the next step, using the start and end measures, the temporal-distance to the temporal query range is computed. This is possible by using a simple function such as, say, temporal_dist to compute the minimum distance between query and data geometry temporal end points. Finally, the resulting geometries are ordered based on the temporal distance, and the k nearest geometries are returned. The SQL query looks as follows:

```
select gid
from ( select gid,
          temporal_dist(s, e, start_measure(res), end_measure(res)) t_dist
       from ( select a.gid gid, sdo_intersection(a.geom, q) res
              from geom_tab a
              where sdo_relate(a.geom, q, 'intersects') = 'TRUE' )
       order by t_dist )
where rownum <= k;
```

Spatial-knn Temporal-Range Query This query is evaluated as follows: First, the temporal range is used to select a set of geometries. The resulting geometries are then clipped using the temporal range endpoints. Next, the spatial distance between the clipped geometries and the query geometry is computed using the sdo_distance function provided by Oracle Spatial. Then the geometries are ordered based on the computed distances of the clipped geometries to the query geometry. Finally, the k nearest data geometries in the above set are returned as the result. The SQL query looks as follows:

```
select gid
from ( select gid, sdo_distance(q, res) s_dist
       from ( select gid, clip_geometry(geom, s, e) res
              from geom_tab
              where sdo_relate(temporal_fn(geom),
                      geometry_constructor(s,e),'inside')='TRUE' )
       order by s_dist )
where rownum <= k;
```

5 Issues and Limitations

Although the presented approach allows for reasonably-efficient indexing of spatio-temporal data, several issues arise. Since only either a spatial or a temporal index is used in answering the queries, the performance may not be as effective as having a single combined, spatio-temporal index. Several performance-enhancing proposals exist that can be implemented in future releases. Next, several proposals also exist to segment each LRS geometry into multiple parts, thus reducing the MBR-area covered in spatial indexes. These are also likely to be implemented in future releases. Finally, the temporal dimension can only be modeled as a numerical attribute in the above model. Although not ideal, translation from date to numerical values can be performed in the application layer.

6 Conclusions and Future Work

In this paper, we described a model for storing and indexing spatio-temporal data in Oracle. This approach combines the spatial indexing, linear referencing, and function-based indexes of Oracle to support most types of queries on spatio-temporal data. As a consequence, advantages—such as concurrency, recovery, scalability, and replication—of storing spatio-temporal data in a commercial database are easily extended to spatio-temporal data. This approach marks the first step to indexing spatio-temporal data in Oracle using existing technology. Some of the limitations of this approach can be easily removed if both spatial and temporal attributes are treated together in a combined index and query processing mechanism. Also, indexing spatio-temporal data in parametric space may be necessary to cater to some mobile applications. These efficient indexing strategies will be explored in future releases.

References

[1] S. Defazio, A. Daoud, L. A. Smith, and J. Srinivasan. Integrating IR and RDBMS using cooperative indexing. In *ACM SIGIR Conf. on Information Retrieval*, pp. 84–92, 1995.

[2] M. J. Egenhofer. Reasoning about binary topological relations. In *Symp. on Spatial Databases*, pp. 271–289, 1991.

[3] M. J. Egenhofer, A. U. Frank, and J. P. Jackson. A topological data model for spatial databases. In *Symp. on Spatial Databases*, pp. 271–289, 1989.

[4] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *ACM Symp. on Principles of Database Systems*, 1999.

[5] G. Kollios, V. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing animated objects using spatiotemporal access methods. *IEEE Trans. on Knowledge and Data Engineering*, 13(5):758–777, 2001.

[6] K. Prokaew, I. Lazaridis, and S. Mehrotra. Querying mobile objects in spatio-temporal databases. In *Symp. on in Spatial and Temporal Databases*, pp. 59–78, 2001.

[7] K. V. R. Kanth, S. Ravada, J. Sharma, and J. Banerjee. Indexing medium-dimensionality data in Oracle. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 521–522, 1999.

[8] K. V. R. Kanth and S. Ravada. Efficient processing of large spatial queries using interior approximations. In *Symp. on Spatial and Temporal Databases*, pp. 404–421, 2001.

[9] S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexing the positions of continuously moving objects. In *ACM SIGMOD Int. Conf. on Management of Data*, pp. 331–342, 2000.

[10] Y. Theodoridis, T. Sellis, A. N. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *Conf. on Scientific and Statistical Database Management*, pp. 123–132, 1998.

[11] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Int. Conf. on Scientific and Statistical Database Management*, pp. 111–122, 1998.

CALL FOR PARTICIPATION

28th International Conference on

Very Large Data Bases 2002

Kowloon Shangri-La Hotel

Hong Kong SAR, China

August 20-23, 2002

http://www.cs.ust.hk/vldb2002/

VLDB 2002 continues the 27-year tradition of VLDB conferences as the premier international forum for database researchers, vendors, practitioners, application developers, and users to present, discuss and exchange ideas on the latest developments in database technology. Program details can be found on the conference web site.

Register online today!
Early Registration Deadline:
July 2, 2002

Keynotes

Data Routing Rather than Databases: The Meaning of the Next Wave of the Web Revolution to Data Management – Adam Bosworth (BEA Systems, U.S.A.)

Foundation Matters – Chris Date (Independent Consultant, U.S.A.)

Tutorials

eBusiness Standards and Architectures

Application Servers and Associated Technologies

Querying and Mining Data Streams: You Only Get One Look

Automation in Information Extraction and Integration

Sensor Data Mining: Similarity Search and Pattern Analysis

Database Tuning: Principles, Experiments, and Troubleshooting Techniques

Searching and Mining Fine-Grained Semi-structured Data

Panels

Data Management Challenges in Very Large Enterprises

The Future Home of Data

Biodiversity and Ecosystem Informatics - Database Research, Technology Transfer, or Application Development?

Plus ...

10-year Best Paper Award

69 Research Papers

15 Industrial, Applications and Experience Papers

17 Demonstrations

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398