Indexing Techniques for Spatiotemporal Databases

George Lagogiannis
University of Patras, Greece

Christos Makris
University of Patras, Greece

Yannis Panagis
University of Patras, Greece

Spyros Sioutas
University of Patras, Greece

Evangelos Theodoridis
University of Patras, Greece

Athanasios Tsakalidis
University of Patras, Greece

INTRODUCTION

We can define as spatiotemporal any database that maintains objects with geometric properties that change over time, where usual geometric properties are the spatial position and spatial extent of an object in a specific d-dimensional space. The need to use spatiotemporal databases appears in a variety of applications such as intelligent transportation systems, cellular communications, and meteorology monitoring. This field of database research collaborates tightly with other research areas such as mobile telecommunications, and is harmonically integrated with other disciplines such as CAD/CAM, GIS, environmental science, and bioinformatics.

Spatiotemporal databases stand at the crossroad of two other database research areas: spatial databases (Güting, 1994; Gaede & Gunther, 1998) and temporal databases (Salzberg & Tsotras, 1999). The efficient implementation of spatiotemporal databases needs new data models and query languages and novel access structures for storing and accessing information. In Güting, Bohlen, Erwig, Jensen, Lorentzos, Schneider, and Vazirgiannis (2000) a data model and a query language capable of handling such time-dependent geometries, including those changing continuously that describe moving objects, were proposed; the basic idea was to represent time-dependent geometries as attribute data types and to provide an abstract data type extension to the traditional database data models and query languages. In that paper, it was also discussed how various temporal and spatial models could possibly be extended to be spatiotemporal models.

BASIC ALGORITHMIC TOOLS

The problem of spatiotemporal indexing is considered in the standard external memory model. In this model each disk access transfers a contiguous block of B data items in a single input/output operation (I/O). The space complexity of a data structure is measured in terms of the amount of disk blocks it uses, while the time complexity of its various operations is expressed with the number of needed I/Os. In the sequel, we let N denote the number of stored objects and let $n=N/B$ and $k=K/B$, where K denotes the size of the desired output.

R-Tree and Variants

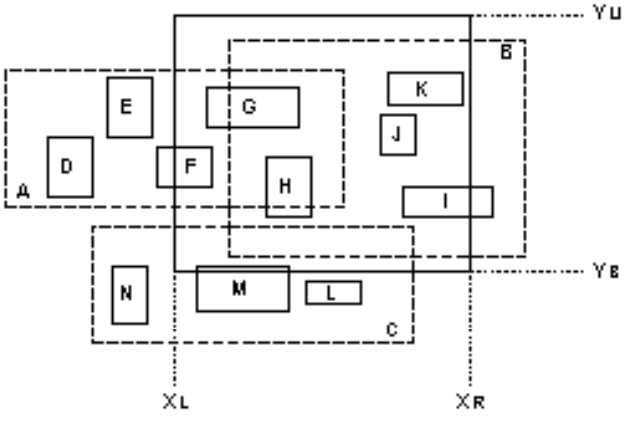
The R-tree (Guttman, 1984) is a spatial access method; it stores objects in R^d with a spatial position and extent and is able to answer various geometric queries. In a nutshell, it is an hierarchical structure, resembling a classical B-tree, where every node has size equal to the disk block size, all objects are stored at the leaves of the structure, and all leaves are at the same distance from the root. Each node v of the R-tree corresponds to a d-dimensional rectangle Rect(v); the rectangle corresponding to a leaf is the minimum rectangle that encloses the objects stored at the leaf, while the rectangle

corresponding to an internal node encloses all the rectangles corresponding to its children. Moreover, every node contains between m and M children where m, M are parameters whose value depends on the block size; this means that the height of an R-tree is $O(\log_m n)$. Searching in the R-tree is done in a similar way as in the B-tree, where for both point and region queries, the paths where rectangles intersect with the query object are followed. In contrast to the B-tree, the R-tree does not guarantee that traversing a path of the tree is enough when searching for an object, as the bounding rectangles of entries in the same nodes may overlap one another. In the worst case, the search algorithm may have to visit all nodes, in order to answer a query. See, for example, the 2-dimensional R-tree of Figure 1 that stores 11 rectangles (n=11) with block size B=4. For the given axis-parallel range query (x_L, x_R, y_B, y_U) of the figure below, the search algorithm may have to visit all nodes in order to determine the K=7 rectangles that are enclosed or intersected by the query rectangle.

The search efficiency of the R-tree could be improved if the spatial overlap between sibling nodes could be minimized. There exists a set of heuristics to achieve that, leading to two known variants of the R-tree: the R+-trees (Sellis, Roussopoulos, & Faloutsos, 1987) and the R*-trees (Beckmann, Kriegel, Schneider, & Seeger, 1990). R+-trees do not allow the subspaces of each internal node to overlap with the subspace of its sibling nodes. However, this clipping of the subspaces has, as an immediate consequence, the creation of much more subspaces and thus much more leaves. Also an object is assigned to more than one leaf causing a much larger tree structure. On the other hand, R*-trees embed maintenance algorithms that aim at minimizing the following penalty metrics: (i) the area and the perimeter of each bounding rectangles, (ii) the overlap between two sibling bounding rectangles, and (iii) the distance between the centroid of a bounding rectangle and that of the node containing it. As discussed in the original publication, the minimization of these metrics decreases the probability that a node is accessed by a range query. Experimental studies show that R*-trees can achieve 50% better query times from the common R-trees.

The interested reader should consult Gaede and Gunther (1998) and Manolopoulos, Theodoridis, and Tsotras (2000) for more information concerning the aforementioned structures.

Figure 1. An example of an R-tree



Partition Trees

Partition trees are data structures that can handle simplex range searching queries and are based on the idea of simplicial partitions. A simplicial partition for a set S of N points in R^d is a collection of pairs $P(S) = \{(S_1, s_1\}, (S_2, s_2), \ldots, (S_m, s_m)\}$ where the S_i's are disjoint subsets of S whose union is S and s_i is a simplex containing S_i. The crossing number of $P(S)$ is the maximum number of simplices in $\{s_1, \ldots, s_m\}$ that can be crossed by an arbitrary hyperplane. Matousek (1992) has proved that for any set S and given parameter r, $r \leq N$, it is possible to construct a simplicial partition of size r for S, whose crossing number is $O(r^{1-1/d})$, in $O(N^{1+\varepsilon})$ time, for any $\varepsilon>0$. Using this theorem recursively, it is possible to come up with a partition tree T for the set S. The root node has $O(r)$ children, which correspond to the simplices of the initial partition and are the roots of recursively defined partition trees for the simplicial partitions. A query with a given query simplex can be answered by starting at the root of T and descending towards the leaves, based on the relation between the query simplex and the simplices in the partitions of the various nodes. In Agarwal, Arge, Erickson, Franciosa, and Vitter (2000), they described an external memory version of static partition trees that needed $O(n)$ disk blocks, so that d-dimensional simplex queries could be answered in $O(n^{1-1/d+\varepsilon}+k)$ I/O s.

INDEXING TECHNIQUES

Research on spatiotemporal access methods has mainly focused on two aspects: (i) storage and retrieval of historical information concerning the positions of the moving points, and (ii) prediction of future positions. There are two kinds of spatiotemporal databases: those that deal with *discrete* and those that deal with *continuous* movements. In the sequel, we will briefly refer to discrete spatiotemporal databases, and we will mainly focus our presentation on continuous spatiotemporal databases.

A sequence of spatiotemporal movements in a discrete environment can be considered to be an ordered sequence of database snapshots of the object positions/extents taken at time instants $t_1<t_2<\ldots$, with each time instant denoting the moment where a change took place. By taking this point of view then it could be possible to handle the index-