The D-Tree: An Index Structure for Planar Point Queries in Location-Based Wireless Services

Jianliang Xu, *Member*, *IEEE*, Baihua Zheng, *Member*, *IEEE*, Wang-Chien Lee, *Member*, *IEEE*, and Dik Lun Lee

Abstract—Location-based services (LBSs), considered as a killer application in the wireless data market, provide information based on locations specified in the queries. In this paper, we examine the indexing issue for querying location-dependent data in wireless LBSs; in particular, we focus on an important class of queries, *planar point queries*. To address the issues of responsiveness, energy consumption, and bandwidth contention in wireless communications, an index has to minimize the search time and maintain a small storage overhead. It is shown that the traditional point-location algorithms and spatial index structures fail to achieve either objective or both. This paper proposes a new index structure, called *D-tree*, which indexes spatial regions based on the divisions that form the boundaries of the regions. We describe how to construct a binary D-tree index, how to process queries based on the D-tree, and how to page the binary D-tree. Moreover, two parameterized methods for partitioning the original space, called *fixed grid assignment* (FGA) and *adaptive grid assignment* (AGA), are proposed to enhance the D-tree. The performance of the D-tree is evaluated using both synthetic and real data sets. Experimental results show that the proposed D-tree outperforms the well-known indexes such as the R^*-tree, and that both the FGA and AGA approaches can achieve different performance trade-offs between the index search time and storage overhead by fine-tuning their algorithmic parameters.

Index Terms—Location-dependent data, mobile computing, index structure, energy conservation, spatial database, data broadcast.

1 Introduction

Thanks to the rapid technological development in wireless networks, mobile devices, and location identification techniques, location-based services (LBSs) are emerging as one of the killer applications for mobile computing and wireless data services [11], [19], [23], [32]. Although LBSs exist in traditional computing environments (Guides@Yahoo, for example), their greatest potential is in a mobile computing environment, where users enjoy unrestricted mobility and ubiquitous information access. An LBS returns location-dependent data (LDD) based on location information specified in a query. Due to the mobility of a mobile user, the specified location is typically the current location of the user. Several studies in the literature have investigated cache management and query evaluation techniques to enhance the performance of LBSs [18], [23], [26], [28], [33], [34]. Different from these previous studies, this paper aims to develop an efficient index structure to support *planar point queries*, which is reduced from *LDD queries*, for LBSs in a mobile and wireless environment.

- *J. Xu is with the Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, KLN, Hong Kong. E-mail: xujl@comp.hkbu.edu.hk.*
- *B. Zheng is with the School of Information Systems, Singapore Management University, 469 Bukit Timah Road, Singapore 259756. E-mail: bhzheng@smu.edu.sg.*
- *W.-C. Lee is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802. E-mail: wlee@cse.psu.edu.*
- *D.L. Lee is with the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, KLN, Hong Kong. E-mail: dlee@cs.ust.hk.*

Manuscript received 24 Mar. 2003; revised 6 Nov. 2003; accepted 8 Apr. 2004.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0017-0303.

1.1 Location-Dependent Data Queries

Consider a geographical area $\mathcal{A}$ (referred to as *service area*) covered by an LBS, which provides a type of location-dependent data (e.g., pollution level). A *data instance* is an answer to a query for the offered data type with respect to a user specified location. Each data instance has a certain *valid scope* within which this instance is the only correct answer. For example, Fig. 1 shows four sensor nodes, o_1, o_2, o_3, and o_4, monitoring air pollution and their respective monitoring areas, P_1, P_2, P_3, and P_4. Thus, P_1, P_2, P_3, and P_4 are the valid scopes of their corresponding data instances (i.e., the readings of sensor nodes at o_1, o_2, o_3, and o_4). Given any query location q in, say, P_3, o_3 is the node by which the location q is covered. The sensor readings of node o_3 are returned as the pollution level for a query issued at q. This example will be used throughout this paper as the running example for different index structures.

Given a set of data instances and their valid scopes, an *LDD query* is defined as a query in which a data instance is returned if and only if the query location falls in the corresponding valid scope of the data instance.

Definition 1. *Given a set of data instances $O = \{o_1, o_2, \cdots, o_N\}$ and the corresponding set of valid scopes $S = \{S_1, S_2, \cdots, S_N\}$, an* **LDD query**, *issued at location q, retrieves the data instance o_i from O if and only if q is located in S_i, where $1 \leq i \leq N$ and N is the total number of data instances in the data set.*

In this paper, we study LDD queries in a two-dimensional space, which is assumed for most LBSs [11], [19], [23]. In the following, we list several possible applications of such LDD queries:

- **Location-Sensitive Information Access:** Information includes local traffic reports, pollution levels, public facilities, attractions, entertainments, etc.

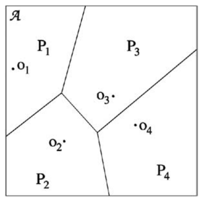


Fig. 1. Running example.

Each instance of such a type of information is valid within a certain geometric region, which may be bounded by streets, rivers, etc., or based on some predefined maps.

- **Object Tracking:** While many positioning techniques return geometric locations (e.g., the latitude/longitude pair in the GPS), some applications may be more interested in finding out, for example, in which monitoring zone or geographical area an object is situated. In such applications, a data instance is associated with the symbolic representation of a zone, and its valid scope is the limit of this zone.
- **Location Model Conversion:** Many advanced context-aware applications are in support of symbolic locations such as parks, shopping malls, or room numbers, but implemented with geometric location-based positioning techniques [19]. The conversion from a geometric location to a symbolic location can be reduced to the problem of LDD queries.
- **Nearest Neighbor Search:** The nearest neighbor problem (e.g., finding the nearest restaurant) can be seen as an LDD query problem when the solution space is precomputed. For example, in a Euclidean space, the solution space for nearest neighbors can be precomputed using the Voronoi Diagram approach [4], where the Voronoi Cell of each spatial object is regarded as its valid scope. As a result, a nearest neighbor search can be solved as an LDD query.

1.2 Indexing Problem for LDD Queries

As mentioned, this study is to explore the use of index to facilitate efficient evaluation of LDD queries. To formulate the indexing problem, we start by defining a notion of *data region*.

Definition 2. *A* **data region**, P_i, *is a spatial representation of the valid scope for a data instance. One data region corresponds to one data instance, such that* $\cup_{i=1}^{N} P_i = \mathcal{A}$, *and* $P_i \cap P_j = \phi\ \forall 1 \leq i, j \leq N$ *and* $i \neq j$, *where* $\mathcal{A}$ *is the service area and* N *is the number of data instances in the data set.*

Without loss of generality, a data region is assumed to take the shape of a polygon. To efficiently evaluate an LDD query, an index can be constructed based on data regions of the service area such that, given a query point (i.e., location), the query can identify the data region covering the query point and return the associated data instance (e.g., the pollution index value). Suppose each data region object maintains a pointer pointing to its associated data instance. Thus, an LDD query is reduced to a *planar point query* defined as follows:

Definition 3. *Given a set of data regions* $P = \{P_1, P_2, \cdots, P_N\}$, *such that* $\cup_{i=1}^{N} P_i = \mathcal{A}$, *and* $P_i \cap P_j = \phi\ \forall 1 \leq i, j \leq N$ *and* $i \neq j$, *a* **planar point query** *returns the polygon* P_i *from* P *that covers the query point* $p \in \mathcal{A}$.

Therefore, for the rest of this paper, we focus on the planar point queries and investigate efficient indexes to support their query evaluation in mobile and wireless environments, where the amount of data access must be a multiple of pages/packets and the performance objectives are different from traditional environments.

The readers should note that, although the above problem formulation assumes nonoverlapping data regions, it can be extended to handle data sets with overlapped regions without loss of generality. Basically, an overlapped area can be treated as a separate data region pointing to multiple data instances.

1.3 Performance Objectives in Mobile and Wireless Environments

We now analyze and discuss some requirements for design and implementation of indexing techniques in location-based wireless services. There are two basic modes for disseminating information to mobile clients in a wireless environment:

- **On-Demand Access**: The mobile client submits a request, including the type of information wanted and the query location, to the server through an uplink channel. The server locates the appropriate data and returns it to the mobile client through a downlink channel. On-demand access is essentially the same as traditional client-server systems, except that communication between clients and the server takes place on wireless channels.
- **Periodic Broadcast**: Data is broadcast on a wireless channel open to the public. When a query is issued, the mobile client tunes into the broadcast channel and filters out the data according to the query and the specified location. A major advantage of broadcast is that it allows simultaneous access by an arbitrary number of mobile users. It is envisaged that in the near future many LBSs (e.g., region-wide traffic reports and tourist information) will utilize broadcast for the dissemination of information to the rapidly increasing population of mobile users.

Indexing is an important technique used to enhance the performance of query evaluation [1], [5]. Indexes for traditional database applications are usually very large and typically stored on disks. Thus, indexing techniques for traditional client-server systems and on-demand access are referred to as *disk indexing*. In contrast, indexing techniques employed in wireless broadcast to address scalability issue and to facilitate power saving on mobile devices are referred to as *air indexing* [14]. To retrieve a data instance in wireless data broadcast without indexing, a mobile client has to continuously monitor the broadcast until the data arrives. This will consume a lot of energy since the client has to remain *active* during its waiting time. The basic idea of air indexing is to include some index information about the arrival times of data instances on the broadcast channel. By accessing the index, mobile clients are able to predict the arrivals of their desired data. Thus, they can stay in *power saving* mode during waiting time and switch to *active* mode only when the data of interest arrives.

The metrics used to evaluate disk indexing and air indexing techniques are different. The performance of disk indexes is often measured by the *number of disk page accesses* required to evaluate a query since the query response time is generally dominated by disk I/O time. In addition, even

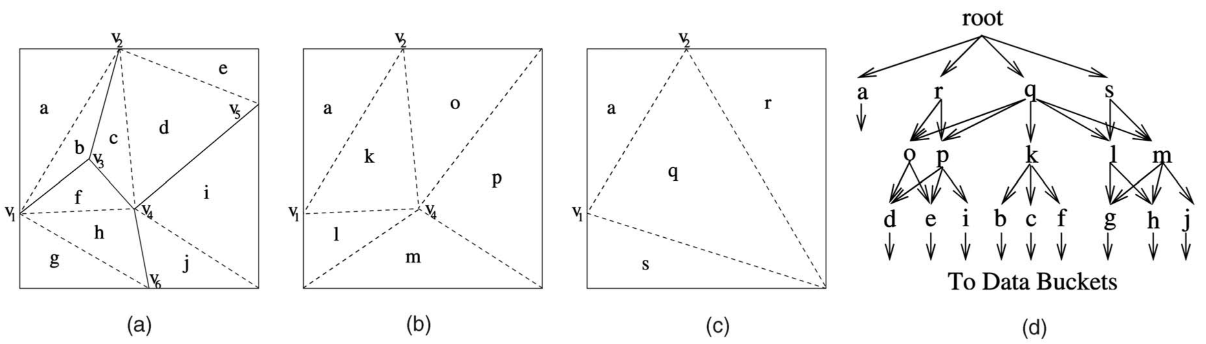


Fig. 2. Trian-tree: Index construction using Kirkpatrick's Algorithm. (a) Triangulation of the original subdivision, (b) retriangulation after v_3, v_5, and v_6 are removed, (c) retriangulation after v_4 is also removed, and (d) index structure.

though hard disk storage is very cheap today, the *index storage overhead* should be kept as small as possible since a large index typically incurs a high update cost. On the other hand, *access latency* and *tuning time* have been used to evaluate the performance of air indexes. Access latency is the period of time elapsed from the moment the mobile client issues a query to the moment when the requested data is received by the client. Tuning time is the amount of time the mobile client stays *active* in order to obtain the requested data. While access latency measures the overhead of broadcasting index information, tuning time reflects the energy consumption by the mobile client since sending/receiving data is power dominant [17].[1] In wireless communications, a bit stream is normally delivered in the unit of the *packet* (or *frame*) for such purposes as error-detecting, error-correction, and synchronization. Thus, tuning time is measured in terms of the *number of packet accesses* [12], [14].

A fundamental issue for both disk indexing and air indexing techniques is the design of the underlying index structures. Although disk indexing and air indexing are employed for different (ultimate) performance objectives, an index structure serving them shares two common design goals: 1) the index should offer a good index search performance in terms of the number of page/packet accesses required to evaluate a query;[2] 2) the index should have a small size.

1.4 Contribution and Paper Organization

In this paper, we develop efficient index structures for LDD queries (reduced to planar point queries) that are good for both disk indexing and air indexing. The main contributions of this paper are summarized as follows:

- We propose a new index structure, called *D-tree*. The basic idea is to index data regions based on the divisions between them. We describe how to construct a binary D-tree index, how to process queries based on this index structure, and how to page the binary D-tree to fit the page capacity.
- Two parameterized methods for partitioning the original space, namely, *fixed grid assignment* (FGA) and *adaptive grid assignment* (AGA), are proposed to enhance the D-tree.
- The performance of the D-tree is evaluated using both synthetic and real data sets. Experimental results show that 1) the proposed D-tree outperforms the well-known indexes such as the R^*-tree for both disk indexing and air indexing, and 2) both the FGA and AGA approaches can achieve different performance trade-offs between the index search time and storage overhead by fine-tuning their algorithmic parameters.

The rest of this paper is organized as follows: In Section 2, we review some existing index structures that can be used for answering planar point queries. Section 3 presents the proposed D-tree index structure along with the space partitioning, query processing, and paging algorithms. A performance comparison of the D-tree and other index structures is presented in Section 4. Section 5 describes the techniques to enhance the D-tree, which is followed by a performance evaluation in Section 6. The related work is described in Section 7. Finally, Section 8 concludes the paper.

2 Existing Indexes for Planar Point Queries

LDD queries have been reduced to planar point queries on nonoverlapped spatial regions. The actual shapes of spatial regions need to be captured in the index. To do so, there are two typical types of techniques that can be employed, *object decomposition* and *object approximation* [5], [8]. The former represents the shape of each data region as the geometric union of simple shapes such as triangles or trapezoids, whereas the latter uses simple shapes such as bounding rectangles to approximate data regions. In the following sections, we briefly review several representative solutions and analyze their limitations for planar point queries in mobile and wireless environments.

2.1 Object Decomposition

The Kirkpatrick's algorithm [16] and the trapezoidal map [4] are two well-known methods that fall into the category of object decomposition. They are improved variants of the Quad-tree [24]. Both of them are based on the principle of recursive decomposition of space. The Kirkpatrick's algorithm first triangulates the original subdivision. It then recursively removes some vertices, along with all the edges connected to them, and retriangulates the new subdivision. This operation continues until the number of triangles contained in the space is smaller than some predefined threshold (T_{min}). Figs. 2a, 2b, and 2c show the triangulation process of our running example, where T_{min} is set to five. From Fig. 2a to Fig. 2b, vertices v_3, v_5, and v_6 are removed;

1. For systems in which the mobile client is charged on a per-bit basis, tuning time is also used to measure the access cost.

2. For the purposes of illustration, in the rest of the paper, a disk page and a wireless packet/frame are uniformly called a page.

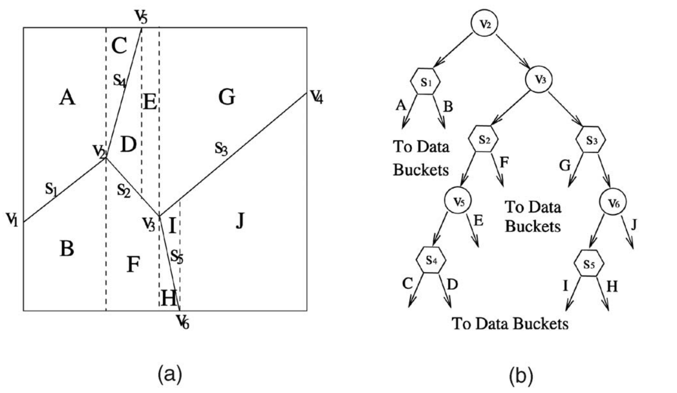


Fig. 3. Trap-tree: Index construction using trapezoidal map. (a) Final trapezoidal map and (b) index structure.

from Fig. 2b to Fig. 2c, vertex v_4 is removed. A hierarchical index tree, shown in Fig. 2d, is built on the triangles generated in the course of recursive triangulation.

Given a query point, the search begins from the root. It checks each child sequentially until a triangle covering the query point is found. Then, the search continues from that node all the way down to a leaf node. In this paper, we call the index structure built by the Kirkpatrick's algorithm the *trian-tree*.

In the trapezoidal map approach, the planar subdivision is viewed as a collection of line segments. The index structure is built when the line segments are inserted one by one into the subdivision. From each new vertex created by the insertion of a line segment, we draw two vertical extension lines, one going upward and the other going downward. The extension does not stop until it meets a line segment that has been previously inserted. Eventually, the original subdivision is decomposed into a set of trapezoids.

Fig. 3a shows the final trapezoidal map of our example, where line segments are inserted in the order of s_1, s_2, s_3, s_4, and s_5. The corresponding index structure is shown in Fig. 3b. The index consists of two kinds of nodes: the *x-nodes*, represented by circles, each of which records the x-coordinate of a vertex; the *y-nodes*, represented by hexagons, each of which records a line segment. Essentially, the left (right) subtree of an x-node recursively represents the subdivision to the left (right) of the extension lines originating from the x-node. Likewise, the left (right) subtree of a y-node recursively represents the subdivision above (below) the line segment represented by the y-node.

Given a query point p, the search process begins at the root and terminates when a leaf node is met. At an x-node, we evaluate whether p lies to the left or to the right of the vertical line that goes through the stored x-coordinate. At a y-node, we evaluate whether p lies above or below the stored line segment. In this paper, we call the index structure built using the trapezoidal map approach the *trap-tree*.

We can see from the example that both the trian-tree and the trap-tree have a fairly large index size. For our example, which consists of only four regions and six vertices, the index trees have about 10 nodes. As we will see in a later performance evaluation, the large index size results not only in a high access latency overhead, but also in inferior search efficiency. In general, the trap-tree achieves a better performance than the trian-tree in mobile and wireless environments [29]. Therefore, the trap-tree is used in the performance evaluation as the representative of the object decomposition algorithms.

2.2 Object Approximation

The object approximation technique has been commonly employed for disk indexing in the spatial database field [8]. The R-tree is a classical spatial index structure [9]. The basic idea is to approximate a spatial object with a minimal bounding rectangle (MBR) and to index the MBRs recursively. Each node in the index tree contains a number of entries according to the page capacity. An entry in an internal node contains a child-pointer pointing to a lower level node in the tree and a bounding rectangle covering all the rectangles in the lower nodes in the subtree. In a leaf node, an entry consists of a pointer pointing to the data and a bounding rectangle which bounds the data's spatial region. The variants of the R-tree differ from each other in terms of the criteria used to insert an object and to split an overflowing node. Extensive experiments conducted in [2] showed that the R^*-tree gives a superior performance for different types of queries and operations. Fig. 4c shows the structure of the R^*-tree for our running example, where the corresponding MBRs are shown in Figs. 4a and 4b. In the performance evaluation, the R^*-tree is used as the representative of the object approximation technique.

Given a query point p, the search algorithm descends the tree from the root. The algorithm recursively traverses down the subtrees of bounding rectangles that cover p. When a leaf node is reached, the bounding rectangles are tested and their objects are fetched to verify if they cover p. When applying the R^*-tree to planar point queries, to

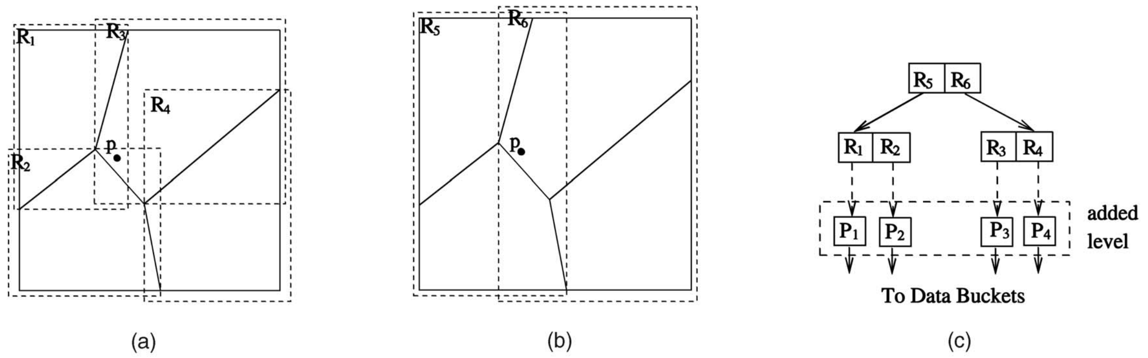
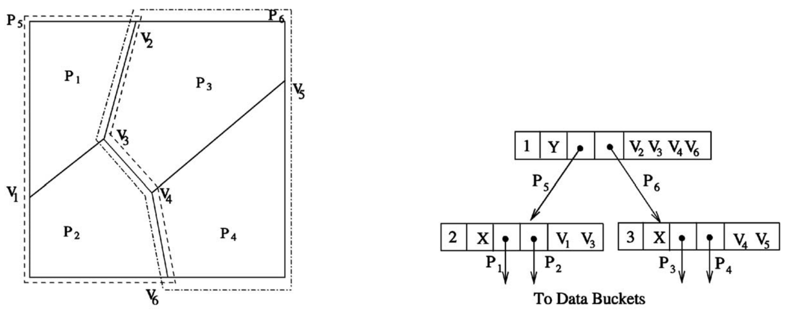


Fig. 4. Index construction using the R^*-tree. (a) MBRs of data regions, (b) MBRs in the root node, and (c) index structure.



(a) (b)

Fig. 5. Index construction using the D-tree. (a) Divisions in the example. (b) D-tree structure.

reduce the search time, we modify the tree structure slightly as follows: One level is added to the R*-tree at the bottom, as shown in Fig. 4c (the level indicated by the dashed line). This level consists of the actual shapes of data regions so that, in the containment test, a costly access of actual data is avoided.

A problem with R-tree-based methods is that if a point is covered by two or more sibling MBRs, it may need to explore several subtrees before the target object can be located. This will increase the search time. Unfortunately, in planar point queries, all data regions are adjacent to each other and, as such, their MBRs will overlap. The nature of this problem renders the approximation-based spatial index structures inefficient. As an example, suppose the query point is p in Figs. 4a and 4b. We need to access a total of six nodes (i.e., the root, R_5, R_1, R_2, R_6, and R_3) before we know it is covered by P_3.

3 The D-Tree Index Structure

This section describes the proposed D-tree index structure. We first present the overall idea of the D-tree in Section 3.1. The space partitioning algorithm is described in Section 3.2. Section 3.3 provides the algorithm for query processing based on the D-tree. Finally, Section 3.4 explains how to page the D-tree.

3.1 An Overall Picture

As discussed in the last section, the object decomposition and approximation approaches suffer from a large index size and/or a long search time. We also observed that the actual shapes of data regions are contained, either explicitly or implicitly, in the index structures of both approaches. With object decomposition, the shape of each region is embedded in the index structure, while with object approximation, it is approximated by an MBR with the exact shape encoded in an additional level of nodes, as shown in Fig. 4c. Based on this observation, we propose a new index structure, called *D-tree*, to index data regions directly based on the divisions between them. This new index structure neither decomposes nor approximates data regions. In the following, we illustrate the overall idea.

The D-tree is a binary tree which recursively partitions a space consisting of a set of data regions into two *complementary* subspaces containing about the same number of regions until each subspace has one region only. The partition between two subspaces is represented by one or more polylines. The overall orientation of the partition, hereinafter referred to as *partition dimension*, can be either x-*dimensional* or y-*dimensional*, which is obtained, respectively, by sorting the data regions based on their y-coordinates or x-coordinates (see Section 3.2 for details). Fig. 5a shows the partitions for our running example. The polyline $pl(v_2, v_3, v_4, v_6)$ partitions the original space into P_5 and P_6, and $pl(v_1, v_3)$ and $pl(v_4, v_5)$ further partition P_5 into P_1 and P_2, and P_6 into P_3 and P_4, respectively. The first polyline is y-dimensional and the remaining two are x-dimensional. The algorithm of finding the partition for a space will be described in Section 3.2.

The data structure of a D-tree node is illustrated in Fig. 6. The meaning of each attribute is summarized in Table 1. In the D-tree, an internal node contains the partition that divides the current space into two complementary subspaces, a left (right) pointer storing the address of the node containing the data regions that lie in the lefthand or upper (righthand or lower) subspace, and some control parameters including *bid* and *header*. A leaf node contains the partition of two data regions, the pointers storing the addresses of the data instances corresponding to the regions, and the control parameters as well. A spatial region can be inferred from the partitions when following the path from the root towards the corresponding leaf node. Note that in the data structure of Fig. 6, we place the pointers before the partition on purpose. We will explain this in Section 3.4. For the moment, there is no difference for where the pointers are placed. The binary D-tree satisfies the following four properties:

1. Every node has exactly two children.
2. All objects in the left subtree of a node are to the left of or above the partition, and all objects in the right subtree are to the right of or below the partition.
3. The tree is height-balanced, i.e., the levels of the leaf nodes differ by at most one (see Section 3.2 for proof of correctness).
4. The search time for a point query is $\Theta(\log N)$ in terms of the number of nodes visited, where N is the number of data regions in the original space (see Section 3.3 for illustration of correctness).

The D-tree index structure for the running example is depicted in Fig. 5b, where the header attribute is simplified and contains the partition dimension only. Compared to the trian-tree in Fig. 2 and the trap-tree in Fig. 3, the size of the

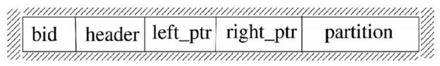


Fig. 6. Data structure of a D-tree node.

TABLE 1
Illustration of the Attributes in a D-Tree Node

Attribute	Description
bid	the unique id for each node in the tree structure
header	including the flag indicating whether the node occupies more than one page, the style and the size of the partition
left_ptr	the type (data or node pointer) and the address of the left child
right_ptr	the type (data or node pointer) and the address of the right child
partition	a sequence of coordinates that represents the partition

D-tree is much smaller. Compared to the R*-tree in Fig. 4, the search time for the D-tree is expected to be shorter since it searches only two nodes for any query point.

3.2 Partitioning Algorithm

Finding a good partition for a space is crucial to the efficiency of the D-tree. This section describes the proposed space partitioning algorithm. There are many ways of dividing one space into two complementary subspaces that contain almost the same number of data regions. For example, we can sort the regions according to their rightmost x-coordinates, and identify the space that encloses the first $N/2$ regions as one subspace and the rest as the other. Alternatively, we can sort the regions according to their lowest y-coordinates, leftmost x-coordinates, or uppermost y-coordinates, and perform the subspace identification in a similar fashion. Moreover, if N is odd, we may consider the first $(N+1)/2$ or $(N-1)/2$ regions as one subspace. Consequently, we evaluate four partition styles when N is even and eight when N is odd.

For each partition style, the size of the partition is measured in terms of the number of coordinates that represent the partition. In selecting among different partition styles, we choose the one with the smallest partition size. If they are equal, for tie breaking, we define the *interprob* of two subspaces as the probability of a query being issued from their *interlocking part*, i.e., for a y-dimensional (x-dimensional) partition, the area between the rightmost x-coordinate (lowest y-coordinate) of the lefthand (uppermost) subspace and the leftmost x-coordinate (uppermost y-coordinate) of the righthand (lower) subspace (e.g., D_2 in Fig. 7a). Ties are broken by favoring the one with the lowest interprob. The reason for this will become clear when the reader proceeds to the next two sections, and we will mention it again later on in Section 3.4.

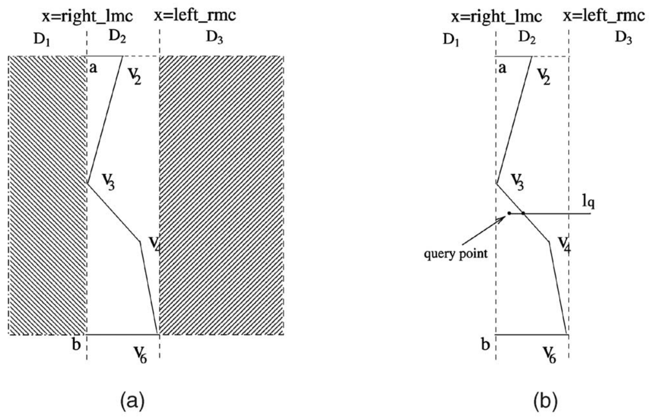


Fig. 7. Examples with the Space Partitioning and Query Processing Algorithms. (a) Partition. (b) Query processing.

If several partition styles have the same lowest interprob, we randomly select one style. Algorithm 1 gives an overview of the space partitioning algorithm.

```
Algorithm 1 SpacePartitioning: Partition the Original Space
            Recursively
Input: an array of data regions, each represented by the
       coordinates of its vertices
Output: the binary D-tree
Procedure:
1. for each possible partition style do
2.   evaluate the size of the partition (Algorithm 2);
3. end for
4. construct the index node using the partition style with the
   smallest size;
5. recursively invoke Algorithm 1 to construct the left
   subtree;
6. recursively invoke Algorithm 1 to construct the right
   subtree.
```

We now describe the algorithm *PartitionSize* (Algorithm 2) that takes an array of data regions and the partition style as the input, and evaluates the partition size with this style. To simplify the illustration, the algorithm is presented for the style in which the sorting is based on the regions' rightmost x-coordinates, and $N/2$ regions are given to the lefthand subspace. It is obvious to extend it to other partition styles. The algorithm consists of two phases. In the first phase (lines 1-3), we identify the lefthand subspace and construct the extent for this subspace. Specifically, we scan all polygons in the lefthand subspace and build a hash table in which each element corresponds to a line segment and contains the id(s) of its associated polygon(s). Those line segments with only one associated polygon constitute the extent. Note that the extent of a subspace could consist of one or more closed polygons.[3] In the second phase (lines 4-16), we prune some unnecessary segments in the extent such that the remaining segments are sufficient to guide a query point to the appropriate subspace. Thus, we prune the segments that are to the left of the vertical line that goes through the leftmost x-coordinate of the righthand subspace, *right_lmc* (lines 6-8). In addition, we truncate the remaining segments by *right_lmc* (lines 9-15). Note that this operation does not change the partition size, but it identifies *right_lmc* in the partition, which is useful in paging (see Section 3.4). Fig. 7a gives an example, where the extent of the lefthand subspace is the union of the dash-dot line and the solid line. The output partition of Algorithm 2 for this example would be the solid line. The complexity of Algorithm 2 is $O(N \log N + M)$, where N is the number of

3. For example, in Fig. 5a, when the partition $pl(v_2, v_3, v_4, v_6)$ has one or more vertices (say, v_3 and/or v_4) touching the left edge of the space, the left subspace will consist of more than one closed polygon.

regions and M is the number of line segments. Therefore, the overall complexity of the recursive partitioning for the original space (i.e., Algorithm 1) is $O(N^2 \log N + MN)$.

```
Algorithm 2 PartitionSize: Evaluate the Size of the
                              Partition
Input: an array of data regions, each represented by the
          coordinates of its vertices,
          the partition style // assuming $N/2$ regions in the
                              // left and rightmost
                              // x-coordinate sorting in the
                              // following procedure
Output: the partition and its size
Procedure:
1. sort the regions in the increasing order of their rightmost
   x-coordinates;
2. identify the first $N/2$ regions as the lefthand subspace;
3. construct the extent for the lefthand subspace;
4. $right\_lmc$ := the leftmost x-coordinate of the righthand
   subspace;
5. for each segment $s((x_1, y_1), (x_2, y_2))$ in the extent do
6.   if $s((x_1, y_1), (x_2, y_2))$ is to the left of the vertical line
     $x = right\_lmc$, then
7.     remove $s((x_1, y_1), (x_2, y_2))$ from the extent;
8.   end if
9.   if $s((x_1, y_1), (x_2, y_2))$ intersects with the line
     $x = right\_lmc$ at $(right\_lmc, y_s)$ then
10.    if $x_1 > x_2$ then
11.      reduce $s((x_1, y_1), (x_2, y_2))$ to
         $s((x_1, y_1), (right\_lmc, y_s))$;
12.    else
13.      reduce $s((x_1, y_1), (x_2, y_2))$ to
         $s((x_2, y_2), (right\_lmc, y_s))$;
14.    end if
15.  end if
16. end for
17. return the set of polylines consisting of the remaining
    segments and its size in terms of the number of
    coordinates.
```

Theorem 1. *All leaf nodes in the D-tree have levels that differ by no more than one, i.e., the length of the path from the root to a leaf node is either $\lfloor logN \rfloor$ or $\lceil logN \rceil$ for a D-tree of size N, where N is the number of regions indexed.*

Proof. We prove the theorem by induction: 1) for $N = 2$ and $N = 3$, the proof is trivial; 2) for $N > 3$, suppose all leaf nodes differ by at most one level for any D-tree of size $N' < N$. We are going to show that it is also true for $N' = N$.

Let's first consider the case where N is an odd number. After the first partition, the original space is divided into two subspaces S_1 and S_2. Without loss of generality, assume S_1 contains $\lfloor N/2 \rfloor$ data regions and S_2 contains $\lfloor N/2 \rfloor + 1$ data regions. Thus, in the D-tree index, the root contains two subtrees T_1 and T_2 of sizes $\lfloor N/2 \rfloor$ and $\lfloor N/2 \rfloor + 1$, respectively. Since $\lfloor N/2 \rfloor < N$, based on our assumption, for any leaf node in T_1, the length of the path from the root to the leaf node is either $\lfloor log(\lfloor N/2 \rfloor) \rfloor + 1$ (i.e., $\lfloor logN \rfloor$) or $\lceil log(\lfloor N/2 \rfloor) \rceil + 1$ (i.e., $\lfloor logN \rfloor$ if $N = 2^m + 1, m = 1, 2, 3, \cdots$; or $\lceil logN \rceil$ otherwise). Similarly, the length of the path from the root to a leaf node in T_2 is either $\lfloor log(\lfloor N/2 \rfloor + 1) \rfloor + 1$ (i.e., $\lceil logN \rceil$ if $N = 2^m - 1, m = 2, 3, 4, \cdots$; or $\lfloor logN \rfloor$ otherwise) or $\lceil log(\lfloor N/2 \rfloor + 1) \rceil + 1$ (i.e., $\lceil logN \rceil$).

Similarly, we can prove that the length of the path from the root to a leaf node is either $\lfloor logN \rfloor$ or $\lceil logN \rceil$ when N is an even number. □

3.3 Query Processing Algorithm

This section presents the algorithm for processing point queries based on the D-tree index structure. The algorithm works as follows: It starts from the root and recursively follows either the left subtree or the right subtree according to the query point and the partition until a leaf node is reached. The whole procedure is described in Algorithm 3, where the partition style is assumed to be y-dimensional. It is trivial to extend the algorithm to the x-dimensional partition style.

```
Algorithm 3 Query Processing on the D-tree
Input: the query point $p$ // assuming a $y$-dimensional
                           // partition in the following
                           // procedure
Output: the pointer to the correct data instance
Procedure:
1. $ptr$ := the pointer to the root;
2. while $ptr$ is not a data pointer do
3.   get the partition of the current node pointed to by $ptr$;
4.   determine if the query point $p$ is to the left or to the
     right of the partition (lines 5-26):
5.   $right\_lmc$ := the leftmost x-coordinate of the partition;
6.   $left\_rmc$ := the rightmost x-coordinate of the partition;
7.   if $p.x < right\_lmc$ then
8.     $ptr$ := the left pointer of the current node;
9.     continue;
10.  end if
11.  if $p.x > left\_rmc$ then
12.    $ptr$ := the right pointer of the current node;
13.    continue;
14.  end if
15.  draw a horizontal ray $l_q$ emanating from $p$ and to the
     right;
16.  $num$ := 0;
17.  for each segment in the partition do
18.    if the ray $l_q$ intersects the segment then
19.      $num$ := $num$+1;
20.    end if
21.  end for
22.  if $num$ mod 2 == 1 then
23.    $ptr$ := the left pointer of the current node;
24.  else
25.    $ptr$ := the right pointer of the current node;
26.  end if
27. end while
28. return $ptr$.
```

In the algorithm, the key issue is to determine whether a given query point p is located to the left or to the right of the partition. As shown in Fig. 7a, we can see that after partitioning, a space is divided into three parts: D_1, D_2, and D_3, where D_1 is bounded by the leftmost x-coordinate of the righthand subspace and D_3 is bounded by the rightmost x-coordinate of the lefthand subspace. If p falls in D_1 (i.e., $p.x < right\_lmc$), it goes to the lefthand subspace (lines 7-10).

If p falls in D_3 (i.e., $p.x > left\_rmc$), it goes to the righthand subspace (lines 11-14). If p falls in D_2, it has to be determined by examining the partition since D_2 is shared by both subspaces (lines 15-26). This operation can be illustrated by an example. In Fig. 7b, suppose that the solid line is the partition and that the query point is p. Consider a horizontal ray l_q emanating from p to the right. If the number of times that this ray l_q intersects the line segments making up the partition is odd, p lies to the left of the partition. If the number is even, p lies to the right of the partition. In Fig. 7b, since the number turns out to be one, we know that the query point is in the lefthand subspace.

Since there is no spatial overlapping among sibling nodes in the D-tree, the search time for each query, in terms of the number of nodes visited, is the length of the unique path from the root to the corresponding leaf node. From Theorem 1, the length of the path from the root to a leaf node is either $\lfloor logN \rfloor$ or $\lceil logN \rceil$. Thus, the search time for the D-tree is $\Theta(\log N)$.

3.4 Paging of the D-Tree

Both wireless data and disk storage are accessed in the unit of the page. Thus, it needs to allocate the nodes of a binary D-tree into pages of fixed size. In this paper, we employ a top-down approach to carry out the page allocation [22]. The algorithm works as follows: The D-tree is traversed in a breadth-first order. When inserting a new node, if the inclusion of the new node in the page where the parent node is allocated does not exceed the page capacity, the new node is allocated space in this page. Otherwise, a new page is allocated to the tree and the new node is allocated to the beginning of this page. We do not split a tree node unless it is larger than the page capacity since splitting a small node will result in two page accesses, instead of one if the node is not split. Finally, to save storage space, we merge some partial pages at the leaf level in a greedy way. The pseudocode of the procedure is described in Algorithm 4. This algorithm has a complexity of $O(N^2)$.

```
Algorithm 4 D-tree Paging Algorithm
Input: a binary D-tree and the page capacity $page\_size$
Output: a paged D-tree
Procedure:
1. arrange the tree nodes in the queue in a breadth-first
   traverse;
2. while there are nodes left in the queue do
3.   pick up the first node in the queue;
4.   $node.size$ := the size of this node;
5.   if no parent or $node.size >$ remaining space of the page
     of the parent node then
6.     while $node.size > page\_size$ do
7.       create a new page;
8.       $node.size := node.size - page\_size$
9.     end while
10.    create a new page;
11.    reduce the remaining space of the page by $node.size$;
12.    allocate the node in the new pages;
13.    for each new page, fill in other info such as the page
       id;
14.  else
15.    allocate the node in the page of the parent node;
16.    reduce the remaining space of the page by $node.size$;
17.  end if
18. end while
19. $remain$ := 0;
20. for all leaf pages do
21.   if $remain >=$ occupied space of the current page
      then
22.     merge the last page with the current page;
23.   end if
24.   $remain$ := remaining space of the current page;
25. end for
```

TABLE 2
Definition of Notations

Notation	Definition
N	number of data instances
s_p	page capacity
p_i	query probability for data instance i
$(n_{i,1}, \cdots, n_{i,m_i})$	the path from the root to the leaf node pointing to instance i
$s_{i,j}$	size of node $n_{i,j}$
$l_{i,j}$	average query time from the root to node $n_{i,j}$
$r_{i,j}$	probability of a query being issued from the *interlocking part* of two subspaces of node $n_{i,j}$ (e.g., D_2 in Figure 7(a))

For a large node that occupies more than one page, the following special arrangement is made. An additional coordinate, RMC (i.e., the rightmost x-coordinate in the partition for a y-dimensional partition or the lowest y-coordinate for an x-dimensional partition), is inserted before the partition. The partition starts with the point of LMC (i.e., the leftmost coordinate in the partition for a y-dimensional partition or the uppermost coordinate for an x-dimensional partition). The previous example in Fig. 7a is used to demonstrate the advantage of this arrangement. For query points falling in D_1 and D_3, once we know RMC and LMC in the first page, we can decide which subtree to follow, thereby eliminating further page access and improving the search performance. Consequently, we place the index pointers before the partition to support this early termination of page access, as discussed in Section 3.1. In addition, it is obvious that for two partition styles of the same size, the lower the interprob of two subspaces, the higher the probability of successful early detection of the next pointer. For nodes with sizes larger than the page capacity, this will result in a shorter index search time. Therefore, in the space partitioning algorithm (Section 3.2), we break ties by favoring the partition style with a lower interprob.

The query processing over the paged D-tree is similar to that over the binary D-tree. We are now going to derive the search time of the paged D-tree in terms of the number of page accesses. To do so, we first define some notations in Table 2.

It is easy to obtain the search time of the root:

$$l_{i,1} = (1 - r_{i,1}) + r_{i,1} \cdot \lceil s_{i,1}/s_p \rceil. \tag{1}$$

Then, the search time from the root to node $n_{i,j}$ $(1 < j \le m_i)$ is derived recursively:

$$l_{i,j} = \begin{cases} l_{i,j-1} & \text{if } n_{i,j} \text{ and } n_{i,j-1} \text{ are in the same page;} \\ l_{i,j-1} + (1 - r_{i,j}) + r_{i,j} \cdot \lceil s_{i,j}/s_p \rceil & \text{otherwise.} \end{cases} \tag{2}$$

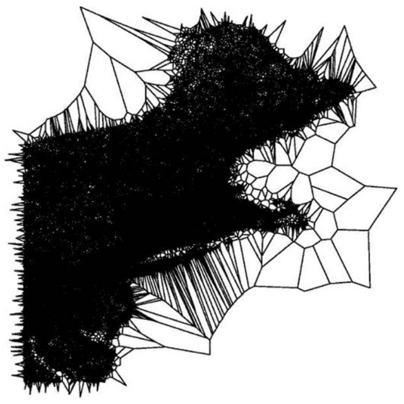


Fig. 8. The POST data set used for performance evaluation.

Therefore, the average search time of the paged D-tree is obtained as follows:

$$E(T) = \sum_{i=1}^{N} p_i \cdot l_{i,m_i}, \tag{3}$$

where l_{i,m_i} is defined in (1) for $m_i = 1$ and (2) for $m_i > 1$.

4 Performance Evaluation

This section compares the performance of the proposed D-tree with the existing index structures. We present the results mainly for two data sets, while we vary the data set size in some experiments. In the first data set (UNIF), we randomly generate 100,000 points in a square Euclidean space. The second set (POST) contains the positions of 123,593 post offices in the northeast of the United States [7]. The valid scopes (or data regions) of the points for nearest neighbor search (see Fig. 8 for the POST data set) are constructed using the Voronoi Diagram approach [4]. The data regions in the POST data set are more clustered than those in the UNIF data set. We have evaluated two access patterns: 1) *data-uniform*: queries are uniformly distributed over the data instances; 2) *location-uniform*: query locations are randomly distributed within the whole region. Due to space limitations, we report the results for the location-uniform pattern only; similar performance trends are observed for the data-uniform pattern.

The D-tree is compared to the trap-tree and the R^*-tree, the two typical index structures in the categories of object decomposition and object approximation, respectively. As in the D-tree, the nodes in the trap-tree do not fit the page capacity either. Thus, we page the trap-tree using the same top-down approach as in the D-tree (see Section 3.4). For the R^*-tree, it is obvious that a better search approach is to examine candidate pages in depth-first order, such that once a containment test in a leaf node evaluates to true, the search can be terminated without accessing useless branches. We employ this search method for the R^*-tree in the experiments. For the same reason, in wireless broadcast, the R^*-tree is broadcast in depth-first order on the wireless channel. The trap-tree and the D-tree are broadcast in breadth-first order to facilitate grouping of partial leaf nodes. To reduce the index size, the added level in the R^*-tree is also paged in a greedy manner.

The system parameters for the experiments are set as shown in Table 3. For the trap-tree, the header size is set at 0

TABLE 3
System Parameters Setting

Parameter	D-Tree	Trap-Tree	R*-Tree
bid size	2 bytes	2 bytes	2 bytes
header size	2 bytes	0	0
size of an index pointer	4 bytes	4 bytes	2 bytes
size of a coordinate	4 bytes		
size of a data instance	1K bytes		
page capacity	64 bytes - 4K bytes		

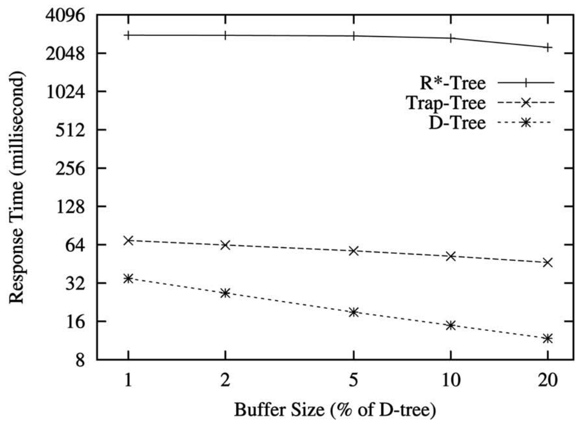
since its partition size is fixed. For the R^*-tree, the size of an index pointer is set at two bytes since its nodes fit the page capacity perfectly and an index pointer stores just the address of the page containing its child. The header info is also unnecessary in the R^*-tree. In the following two sections, we present the experimental results for these index structures in on-demand access and broadcast environments, respectively. The results are obtained for 1,000,000 randomly generated queries on a PC with Pentium 4 1.8G CPU and 1G main memory.

4.1 Query Performance with On-Demand Access

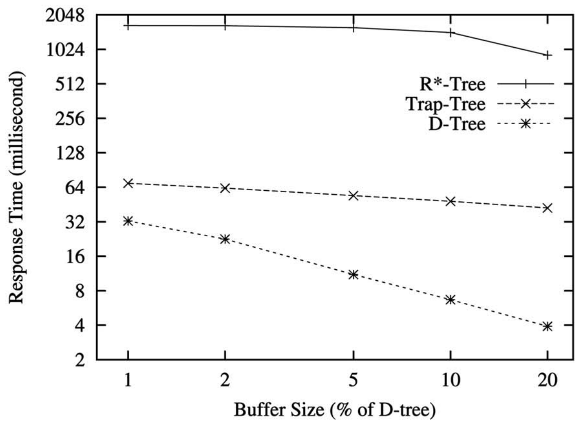
This section evaluates the index structures used for disk indexing in on-demand access environments. We measure the response time of evaluating point queries on the server. We assume the server has a buffer available to hold recently accessed index pages in the main memory. The buffer employs the LRU replacement policy and its size is set at a percentage (varied from 1-20 percent) of the D-tree size. The disk page capacity is set at 4K bytes. Fig. 9 shows the response time as a function of buffer size. It is clear that the proposed D-tree has the best performance. Its response time is two orders of magnitude shorter than that of the R^*-tree and is also shorter than that of the trap-tree at least by a factor of two. This is mainly due to its better index search performance in terms of the number of disk pages accessed (see Fig. 10), which dominates the CPU time and counts for more than 98.5 percent in the total response time as observed in the experiments. The R^*-tree performs poorly, for which we shall explain later in this section.

With increasing buffer size, the improvement of the D-tree over the other two indexes becomes more significant. This can be explained as follows. With the D-tree and the trap-tree, evaluating a query requires accessing all nodes on the path from the root to a leaf node. Thus, with a larger LRU-based buffer, they can cache more high-level nodes to reduce disk I/O. However, the trap-tree can cache less portion of the entire tree than the D-tree as it has a larger index size (see Fig. 13). As for the R^*-tree, because of the branch-and-bound nature of its search algorithm, adding buffer has less effect in reducing disk I/O.

Fig. 11 shows the query response time as a function of database size. The D-tree consistently outperforms the trap-tree by a factor of about two. The improvement of the D-tree over the R^*-tree, however, increases with the database size. This is because with the R^*-tree, the larger the database size, the higher the degree of overlapping among index subspaces and, hence, more backtracking operations are needed for evaluating a query.

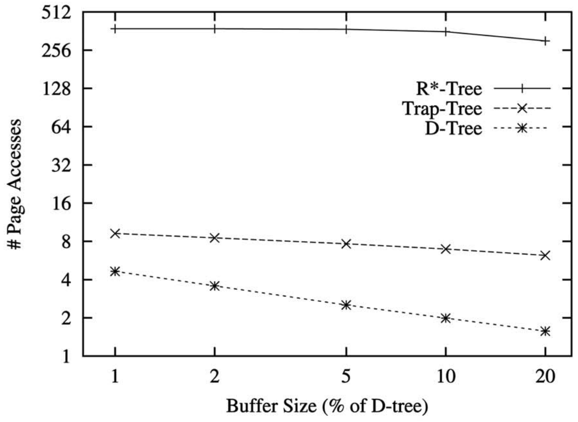


(a)

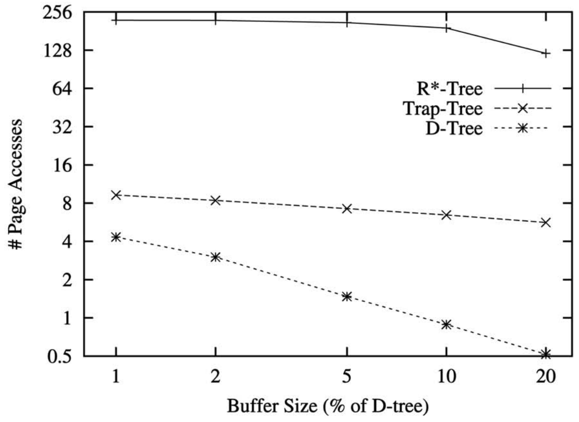


(b)

Fig. 9. Query response time as a function of buffer size. (a) UNIF. (b) POST.

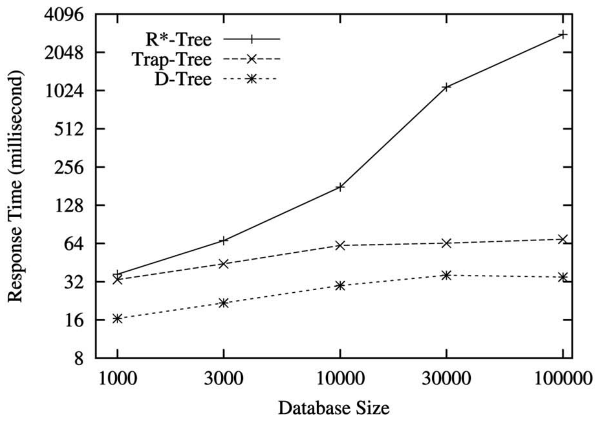


(a)

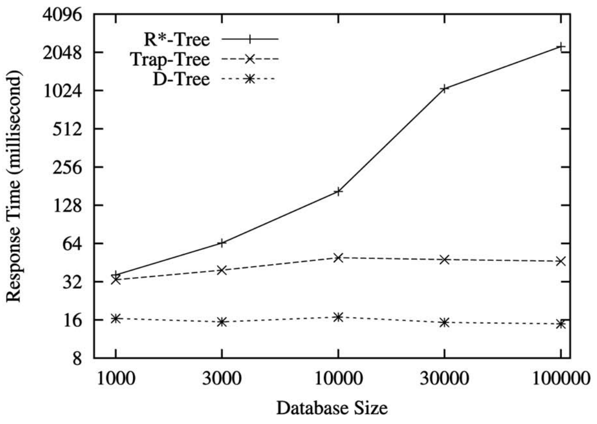


(b)

Fig. 10. Number of disk page accesses as a function of buffer size. (a) UNIF. (b) POST.



(a)

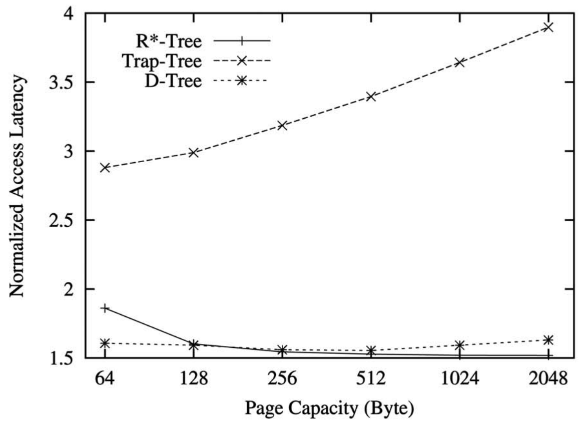


(b)

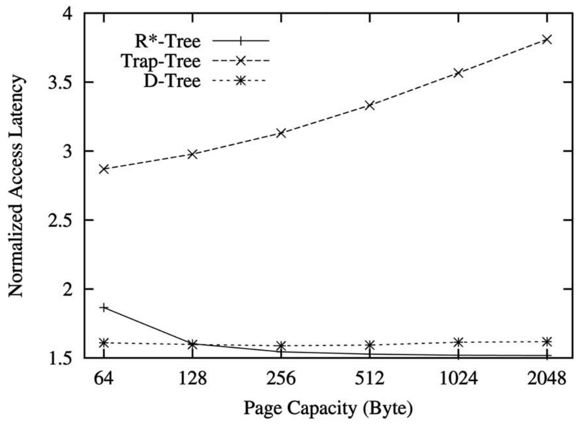
Fig. 11. Query response time as a function of database size. (a) Buffer size = 1 percent. (b) Buffer size = 10 percent.

4.2 Query Performance with Wireless Data Broadcast

This section evaluates the index structures in wireless data broadcast environments. As mentioned in the introduction, we are concerned with access latency and tuning time in such an environment. We assume *flat* broadcast is employed for broadcasting data on the wireless channel. The $(1, m)$ interleaving technique [14] is used to multiplex the index and data on the channel. The optimal value of m depends on the index size. It is calculated for each index structure separately based on the technique presented in [14]. We do not cache the index pages on mobile clients



(a)



(b)

Fig. 12. Normalized access latency as a function of page capacity. (a) UNIF. (b) POST.

since otherwise we need mechanisms to synchronize the cache and the broadcast, which remains a challenge and is beyond the scope of this paper.

Fig. 12 shows the access latency under different page capacities. The results presented are normalized by the average access latency without any index (or called *optimal access latency*, i.e., half of the time needed to broadcast the whole data set). The access latency is affected by the index size, i.e., the larger the index size, the longer the access latency. Fig. 13 shows the index sizes normalized by the data size for the POST data set. Comparing Fig. 13 and Fig. 12b, we can see that the relative performance in index size and access latency is consistent.

We now compare the performance of the different index structures. The trap-tree has the worst performance, with a latency of several times of the optimal latency. This suggests that the trap-tree is almost impractical unless it can provide an extremely good tuning time, which is our main concern. The D-tree achieves an access latency as short as that of the R^*-tree. It is much better than the R^*-tree when the page capacity is small. The access latency overhead due to the D-tree indexing remains at more or less the same level for all settings of the page capacity. It is about 55 percent worse than the optimal latency for both data sets. We expect that the indexing overhead at this level is acceptable provided that a good tuning time is achieved.

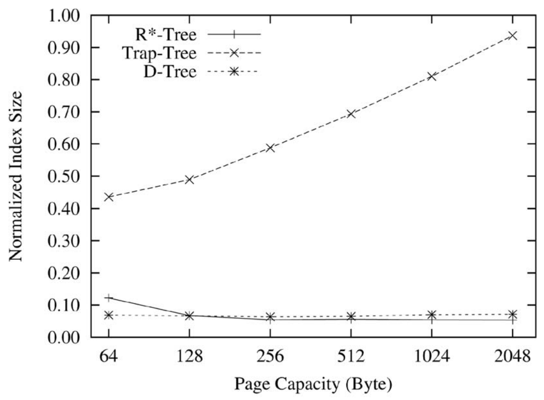


Fig. 13. Normalized index size as a function of page capacity.

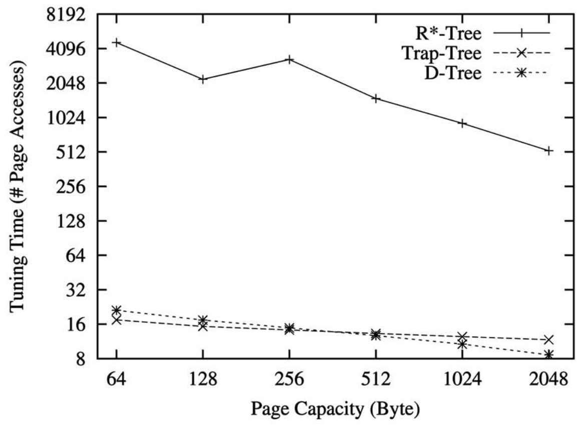
Fig. 14 shows the tuning time for index search. The R^*-tree does not have a good tuning time because the overlapping problem mentioned before often makes the search algorithm access more than one leaf node before the wanted index page can be located. For both data sets, the D-tree has a better performance than the trap-tree when the page capacity is larger than 1K bytes. When the page capacity is smaller than 1K bytes, the D-tree performs worse than the trap-tree. This can be explained as follows. With a large data set, the sizes of the partitions for the D-tree nodes at the highest levels are significant. For example, the root node of the D-tree for either data set has a size of about 5K bytes. Thus, for a small page capacity (e.g., 64 bytes), the root node occupies many pages, thus deteriorating the search performance, although efficient data structures have been applied to avoid accessing the whole node for a large number of queries (see the second paragraph in Section 3.4). When the page capacity increases, the top-down paging approach can pack a large branch of the binary D-tree into a single page. This decreases the height of the tree greatly and reduces the tuning time. However, due to its large index size (see Fig. 13), the trap-tree still has a large height after packing. As a result, the tuning time of the D-tree is only about half of that of the trap-tree when the page has a capacity of 2K bytes.

5 Extensions to the D-Tree

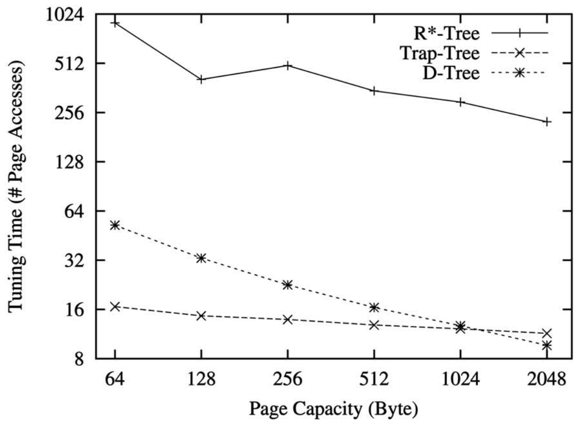
In the previous section, we have shown that, overall, the D-tree substantially outperforms the spatial index structures in the performance comparison. However, the index search performance of the D-tree may deteriorate with a small page capacity because each of the first few partitions would occupy a large number of pages. This section presents extensions for the D-tree to overcome this problem. The basic idea is to first divide the original space into smaller grid cells and then build indexes for those grid cells separately. In the following, we present two parameterized algorithms for performing grid assignment, namely, *fixed grid assignment* (FGA) and *adaptive grid assignment* (AGA).

5.1 Fixed Grid Assignment (FGA)

The fixed grid assignment (FGA) algorithm assigns the grid cells in a fixed manner. The algorithm takes two input parameters: α_x and α_y, which are the numbers of expected partitions along the x and y dimensions, respectively. Let S_x



(a)



(b)

Fig. 14. Tuning time as a function of page capacity. (a) UNIF. (b) POST.

and S_y, respectively, be the horizontal and vertical lengths of the original space. The FGA algorithm simply divides the original space into $\alpha_x \cdot \alpha_y$ grid cells of the same size (i.e., $\frac{S_x}{\alpha_x}$ wide and $\frac{S_y}{\alpha_y}$ high). Note that a region may be split into a few smaller regions if it overlaps with the division lines. A D-tree is then built for each grid cell.

With this approach, the index consists of two levels (see Fig. 15). The higher-level index is basically a one-dimensional array that stores the index pointers to the D-trees of the grid cells. When paged, the higher-level index is sequentially allocated to a number of pages. Thus, the number of page accesses for the higher-level index is either one when the desired index pointer is stored in the first page, or two otherwise. The index search algorithm works as follows: First, we access the header information and get the parameters of S_x, S_y, α_x, and α_y. Then, given the query point (x, y), we use a mapping function, $adr(x, y) = \lfloor \frac{x \cdot \alpha_x}{S_x} \rfloor + \lfloor \frac{y \cdot \alpha_y}{S_y} \rfloor \cdot \alpha_x$, to calculate the address of the pointer pointing to the D-tree covering this query point. The search over the D-tree is the same as before.

5.2 Adaptive Grid Assignment (AGA)

The adaptive grid assignment (AGA) algorithm takes one input parameter, β, which is the threshold set for the maximum number of regions in a grid cell. Similar to the kd-tree [21], the AGA algorithm recursively divides the current space into two subspaces such that the numbers of the regions contained in each subspace are about the same. The dividing dimension alternates between x and y. This procedure stops until all subspaces contain less than β regions. In this paper, we employ the following heuristic to determine the discriminator of each dimension. If the current dividing dimension is x (y), we sort the rightmost x-coordinates (uppermost y-coordinates) of all data regions in the current space and choose the median as the discriminator. Similar to FGA, a D-tree is then built for each grid cell.

The index of this algorithm also consists of two levels (see Fig. 16). The higher-level index is a kd-tree, in which the leaf nodes store the index pointers to the D-trees of the grid cells. Compared to FGA, this approach has a slightly larger higher-level index, but maintains approximately the same number of data regions in each grid cell. The kd-tree index can be paged using the same top-down method described in Section 3.4. The index search procedure is simple. First, we follow the kd-tree to locate the grid cell covering the query point. Then, we proceed to search the corresponding D-tree.

5.3 Packing the D-Trees

In both FGA and AGA, a D-tree corresponds to a grid cell and all the D-trees are paged separately. However, the sizes of some D-trees might be less than the page capacity, especially when the page capacity is large. Thus, to save storage overhead, it is possible to pack some small D-trees into a single page. The packing algorithm (Algorithm 5) is

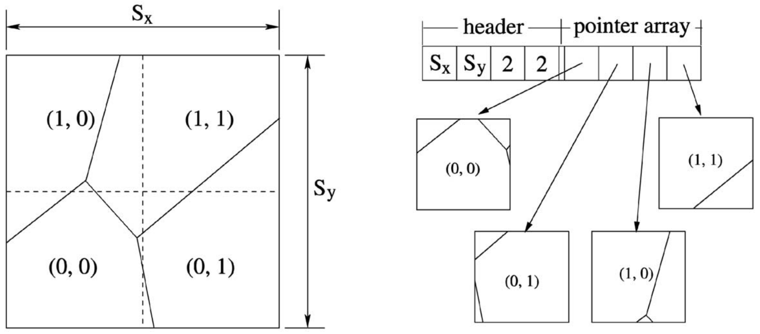


Fig. 15. An example of the fixed grid assignment ($\alpha_x = 2, \alpha_y = 2$).

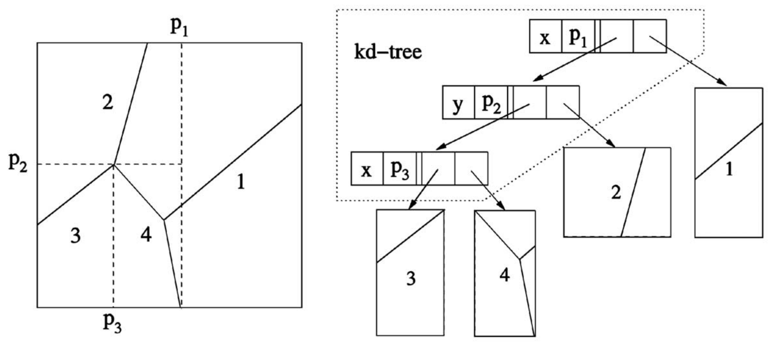


Fig. 16. An example of the Adaptive Grid Assignment ($\beta = 3$).

developed to serve this purpose. It works as follows: We first distinguish small and large D-trees, which are, respectively, smaller and larger than the page capacity. In the first-round packing (lines 3-10), we attempt to utilize the unused space in the pages occupied by large D-trees. In the second-round packing (lines 11-17), we group small D-trees in a greedy manner.

Algorithm 5 Packing D-trees
Input: a set of D-trees for packing and the page capacity $page\_size$
Output: a set of packed D-trees
Procedure:

```
1. $small\_tree\_set$ := the set of D-trees of a size less than the $page\_size$
2. $large\_tree\_set$ := the set of D-trees of a size larger than the $page\_size$
3. for each D-tree $dt_s$ in $small\_tree\_set$ do
4.   for each D-tree $dt_l$ in $large\_tree\_set$ do
5.     if $dt_s$ can be accommodated in the last page of $dt_l$ then
6.       reallocate $dt_s$ to the last page of $dt_l$;
7.       remove $dt_s$ from the set of $small\_tree\_set$;
8.     end if
9.   end for
10. end for
11. $last\_tree$ := NULL;
12. for each D-tree $dt$ in $small\_tree\_set$ do
13.   if $dt$ can be accommodated in the page where $last\_tree$ is allocated then
14.     reallocate $dt$ to the page of $last\_tree$;
15.   end if
16.   $last\_tree$ := $dt$;
17. end for
```

5.4 Handling Updates

In LBSs, when data instances are added or deleted, the valid scope diagram changes in part. In addition, the valid scopes of data instances might be modified over time. Although these updates on valid scopes are expected to happen infrequently in LBSs, this section discusses how to update the index accordingly with the FGA and AGA approaches.

For the FGA approach, since the grid is fixed, we can just identify those grid cells where updates occur and rebuild the D-trees (partially or completely) corresponding to those affected grid cells. No further action needs to be taken.

For the AGA approach, the update cost would be a little bit higher. As with the FGA approach, we first identify those affected grid cells. For each affected grid cell, if it is a region insertion and if the number of regions in the grid cell now exceeds the threshold β, we split the grid cell into two and rebuild the D-trees for both of them; if it is a region deletion and if the number of the regions now drops below the minimum number threshold, we merge it with the subtree rooted at its sibling node, reassign the grid cells if necessary, and rebuild the D-tree(s); for all other cases, we only revise the corresponding D-tree.

The D-tree packing procedure discussed in the last subsection aims to save storage cost. However, when updates happen, it will incur additional cost since the updates may increase the size of a D-tree and, thus, need to reallocate other D-trees in the same page. As such, for disk indexing, one should balance the trade-off between the storage cost and update cost when considering whether to employ the packing procedure. However, for air indexing, it is recommended that the packing procedure is performed since the index is generally built on-the-fly and the index size (transferred to access delay) is a big concern in periodical broadcast.

6 Performance of FGA and AGA

This section evaluates the performance of the FGA and AGA grid assignment approaches. For FGA, the size of the header for the array is set at eight bytes. In the higher-level index of FGA and AGA, the size of an index pointer is set at two bytes. Other parameters are set the same as before (see Table 3). In on-demand access and broadcast environments, an index is employed for different performance objectives. Nevertheless, as demonstrated in Section 4, the performance of an index is essentially determined by the index size and search time in terms of the number of page accesses required to evaluate a query. Hence, in the following, we present the results in terms of these two metrics. The trap-tree is also included as a yardstick. Due to space limitations, we report the results for the POST data set only; similar results are obtained for the UNIF data set.

6.1 Performance of FGA

The FGA approach has two parameters, α_x and α_y, to tune. For simplicity, we attach α_x to α_y such that $\alpha_x = \alpha_y \cdot \frac{S_x}{S_y}$, i.e., a grid cell has the same shape as the original space. We are interested in finding out how the parameter, α_y, affects the performance. Fig. 17 shows the results, where 64 and 1K denote the page capacities of 64 bytes and 1K bytes.

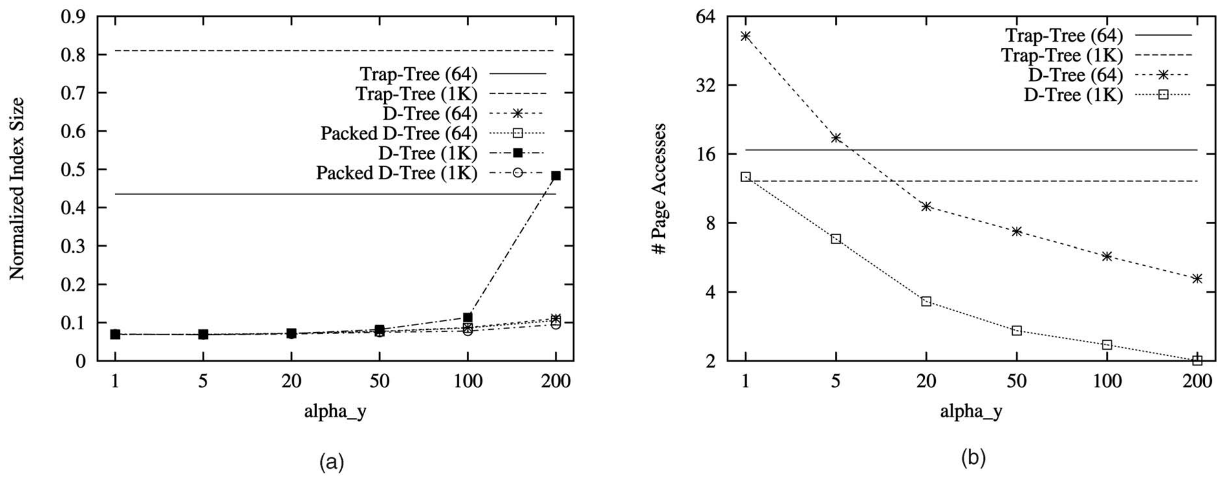


Fig. 17. Performance of FGA (POST). (a) Normalized index size. (b) Average index search time.

Several observations can be made. First, tuning the α_y parameter strikes a balance between the index size and search time. The larger the α_y value, the shorter the index search time but the larger the index size due to a higher degree of duplication of data regions in different grid cells (e.g., the number of regions to be indexed is increased by 114 percent for $\alpha_y = 200$). Therefore, if we want to optimize the overall performance, a medium value of α_y is the best choice as the index search time can be improved significantly while only a small increase in the index size is observed. Fifty to 100 is recommended as the setting of α_y since the index overhead with this range of settings is within 10 percent, but the index search time is improved by a factor of eight. Second, with an appropriate value of α_y (i.e., larger than 5), the D-tree outperforms the trap-tree in terms of both the index size and search time. Third, the packing procedure is important for reducing the index size for large α_ys. For example, for $\alpha_y = 200$ and page capacity 1K, the packing reduces the size from 48.3 percent to 9.5 percent.

6.2 Performance of AGA

There is one parameter, the maximum number of regions in a grid cell, β, associated with the AGA approach. We are interested in examining the impact of β over the performance. Fig. 18 shows the results, where β varies from infinity to 20.

From the figures, we can make the following observations. First, similar to FGA, tuning the β parameter affects both the index size and search time. However, decreasing β does not necessarily improve the index search time. For example, for page capacity 1K, when β drops from $1,000$ to 500, the index search time increases slightly. This can be explained as follows: When β is decreased, more grid cells are created, which enlarges the higher-level kd-tree index and, hence, worsens the index search time in the kd-tree index. If in this case, the search performance in the lower-level D-trees is not improved very much, the overall search performance becomes worse. When the overall performance is considered, 50-100 is the best choice for β since further decreasing β slightly improves the search performance but drastically increases the index size. Second, the D-tree performs much better than the trap-tree when the value of β is less than 1,000. Last, the packing procedure works only for page capacity 1K and $\beta < 50$. This suggests that in AGA, the variance of the number of regions contained in a grid cell is little, as expected.

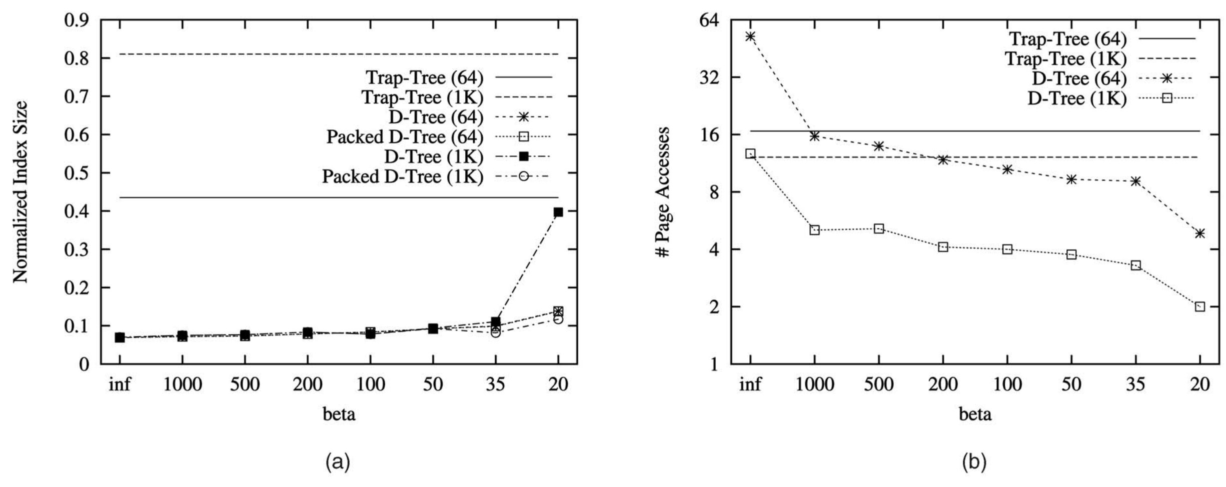


Fig. 18. Performance of AGA (POST). (a) Normalized index size. (b) Average index search time.

TABLE 4
Performance Comparison of FGA and AGA
(Page Capacity = 64 Bbytes, Unpacked and Packed)

	UNIF		POST	
	FGA	AGA	FGA	AGA
Normalized Index Size	0.073	0.077	0.077	0.079
Average Index Search Time	10.4	11.4	7.34	11.8
Variance of Index Search Time	1.92	1.33	11.1	3.43

6.3 Comparison of FGA and AGA

This section compares the performance of FGA and AGA. From the results presented in the last two sections, we select the settings that achieve a good overall performance for the two approaches and compare their relative performance. The results are summarized in Tables 4 and 5, where the index size is set at approximately 8 percent of the data size. When the page capacity is 64 bytes, the packing procedure has no effect on the performance. As can be seen, in general, FGA achieves a better average search performance than AGA. This is mainly because its fixed assignment of grid cells makes the upper-level index very small and efficient. The main problem with FGA is that when the regions are clustered, the numbers of regions contained in different grid cells are not balanced. Therefore, it has a worse variance performance than AGA.

7 Related Work

This section reviews several topics that are highly relevant to indexing and querying LDD. There is a long stream of research on spatial indexing (see [1], [5], [8] for recent surveys), which addresses the problem of designing efficient disk-based indexes to support various spatial operations on a large number of spatial objects. The R-tree was introduced as one of the earliest structures for indexing spatial objects of nonzero sizes [9]. The R-tree serves as the basis of many later spatial indexing structures such as the R^+-tree [25], the R^*-tree [2], and the skd-tree [22]. All of these R-tree-based indexes share the basic assumption that spatial objects are approximated by their bounding rectangles before being inserted into the indexes. We have discussed and compared the representative R^*-tree, which showed an inferior performance than the proposed D-tree for planar point queries.

Recent research on spatial indexing has concentrated on high-dimensional data such as multimedia objects and OLAP [1]. At high-dimensionalities, the R-tree suffers from a low fanout and a high degree of overlapping among index subspaces. Solutions to address these two problems have been extensively explored. To improve fanout, the TV-tree indexes important dimensions only, not all dimensions [20]. To reduce overlapping, the X-tree modifies the splitting algorithm of the R-tree and allows super-nodes that span over multiple pages [3]. To facilitate nearest neighbor search on high-dimensional data, the SS-tree [27], the SR-tree [15], the M-tree [6], and distance-based indexing [31] have been recently proposed. However, in low-dimensional databases, the performances of these index structures were shown to be similar to that of the R^*-tree. In this paper, we address the problem of spatial query processing in regard to real-world physical objects. As such, we consider objects in a two-dimensional space only.

Many algorithms have also been proposed for dealing with the planar point-location problem in computational geometry [4]. However, the studies of planar point-location have mainly focused on computational complexity (assuming a small-size data set that fits in main memory). In contrast, we are concerned with the index size and search time in mobile and wireless environments. Therefore, as shown in Sections 4 and 6, the typical point-location algorithm, the trapezoidal map [4], does not perform well overall.

Several indexing techniques dedicated to wireless broadcast (e.g., the hashing index [13], the signature index [12], and the exponential index [30]) have been introduced in the literature. However, these studies concentrated on a one-dimensional index for equality-based queries and, thus, are inapplicable to indexing LDD where point queries are involved. Imielinski et al. explored the issue of interleaving the index tree and the data on the linear wireless channel such that the tuning time is nearly optimized while making the access latency as short as possible [14]. This work complements our study in the sense that the interleaving technique can be employed to organize the proposed D-tree and the data on the wireless channel as we did in the experiments. More recently, a study [10] discussed query processing for spatial objects over the broadcast channel. However, its main focus was different from ours as it studied how to utilize the limited client cache to reduce the tuning time when traversing spatial index trees.

8 Conclusion

While LBSs are becoming increasingly important to mobile users, query efficiency is a major challenge to service implementation. In this paper, we have studied the issue of using indexing techniques for planar point queries to enhance performance of LDD queries.

Through careful analysis, we found that existing index structures are not efficient when implemented as a page-based index structure for planar point queries. Thus, a new index structure, called D-tree, has been proposed in this paper. Different from the existing approaches, the D-tree

TABLE 5
Performance Comparison of FGA and AGA (Page Capacity = 1K Bytes)

	Unpacked				Packed			
	UNIF		POST		UNIF		POST	
	FGA	AGA	FGA	AGA	FGA	AGA	FGA	AGA
Normalized Index Size	0.078	0.076	0.071	0.078	0.078	0.076	0.075	0.078
Average Index Search Time	2.99	4.00	3.63	4.00	2.99	4.00	2.71	4.00
Variance of Index Search Time	0.02	0.01	1.72	0.01	0.02	0.01	0.44	0.01

neither decomposes nor approximates data regions; rather, it indexes them directly based on the divisions between the regions. The space partitioning algorithm, the query processing algorithm, and the paging algorithm for the D-tree have been described. Two parameterized methods for partitioning the original space, FGA and AGA, have also been proposed to enhance the D-tree.

We have evaluated the performance of the proposed D-tree thoroughly using both synthetic and real data sets. The following results were obtained. In terms of index storage overhead, the D-tree substantially outperforms the trap-tree and performs similarly to the R^*-tree. In terms of index search performance, the D-tree is much better than the R^*-tree. Although the D-tree alone performs worse than the trap-tree for a small page capacity, the enhancement of the FGA and AGA approaches makes the D-tree outperform the trap-tree for a full range of page capacities. Overall, the D-tree achieves a significantly better performance than the existing index structures.

This paper represents the first step into the area of LDD indexing. The D-tree was proposed for efficiently processing planar point queries. Nevertheless, it can be easily extended to answer window queries. We are going to evaluate its performance with window queries in a future study. The basic idea of the D-tree is to index data regions based on their boundaries. We plan to extend this idea to the cases where region boundaries are in arbitrary shapes (e.g., circles) and/or some regions may be completely encircled within other regions. Moreover, this paper considered LDD only under a geometric location model. There are location-dependent applications that employ symbolic location models (e.g., cell-id-based location resolution). Efficient index structures for a symbolic location model deserve future work. Finally, the update issue of LDD indexing is an interesting research topic being pursued by the authors.

ACKNOWLEDGMENTS

A preliminary version of this paper was presented at the Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE '03), Bangalore, India, March 2003. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions that improved the quality of this paper. This work was supported by grants from the Research Grant Council, Hong Kong SAR, China (Grants HKUST6225/02E and HKUST6179/03E) and a grant from Hong Kong Baptist University (Grant FRG/02-03/II-34). W.-C. Lee was supported in part by US National Science Foundation grant IIS-0328881.

REFERENCES

[1] C. Böhm, S. Berchtold, and D.A. Keim, "Searching in High-Dimensional Spaces—Index Structures for Improving the Performance of Multimedia Databases," *ACM Computing Surveys,* vol. 33, no. 3, pp. 322-373, Sept. 2001.

[2] N. Beckmann and H.-P. Kriegel, "The R^*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD Conf. Management of Data,* pp. 322-331, 1990.

[3] S. Berchtold, D.A. Keim, and H.-P. Kriegel, "The X-Tree: An Index Structure for High-Dimensional Data," *Proc. 22nd Int'l Conf. Very Large Data Bases (VLDB '96),* Sept. 1996.

[4] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry—Algorithms and Applications.* New York: Springer, 2000.

[5] E. Bertino, B.C. Ooi, R. Sacks-Davis, K.L. Tan, J. Zobel, B. Shilovsky, and B. Catania, *Indexing Techniques for Advanced Database Systems.* Boston: Kluwer Academic, 1997.

[6] P. Ciaccia, M. Patella, and P. Zezula, "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces," *Proc. 23rd Very Large Databases Conf.,* pp. 426-435, Aug. 1997.

[7] The R-tree Portal, http://www.rtreeportal.org, 2003.

[8] V. Gaede and O. Günther, "Multidimensional Access Methods," *ACM Computing Surveys,* vol. 30, no. 2, pp. 170-231, June 1998.

[9] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Conf. Management of Data,* pp. 47-54, 1984.

[10] S.E. Hambrusch, C.-M. Liu, W.G. Aref, and S. Prabhakar, "Query Processing in Broadcasted Spatial Index Trees," *Proc. Seventh Int'l Symp. Spatial and Temporal Databases (SSTD '01),* pp. 502-521, July 2001.

[11] J. Hightower and G. Borriello, "Location System for Ubiquitous Computing," *Computer,* vol. 34, no. 8, pp. 57-66, Aug. 2001.

[12] Q.L. Hu, W.-C. Lee, and D.L. Lee, "Power Conservative Multi-Attribute Queries on Data Broadcast," *Proc. 16th Int'l Conf. Data Eng. (ICDE 2000),* pp. 157-166, Feb. 2000.

[13] T. Imielinski, S. Viswanathan, and B.R. Badrinath, "Power Efficiency Filtering of Data on Air," *Proc. Fourth Int'l Conf. Extending Database Technology (EDBT '94),* pp. 245-258, Mar. 1994.

[14] T. Imielinski, S. Viswanathan, and B.R. Badrinath, "Data on Air—Organization and Access," *IEEE Trans. Knowledge and Data Eng.,* vol. 9, no. 3, May-June 1997.

[15] N. Katayama and S. Satoh, "The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* May 1997.

[16] D.G. Kirkpatrick, "Optimal Search in Planar Subdivisions," *SIAM J. Computing,* vol. 15, no. 2, pp. 28-35, 1983.

[17] R. Kravets and P. Krishnan, "Power Management Techniques for Mobile Communication," *Proc. Fourth Ann. ACM/IEEE Int'l Conf. Mobile Computing and Networking (MobiCom '98),* pp. 157-168, Oct. 1998.

[18] U. Kubach and K. Rothermel, "Exploiting Location Information for Infostation-Based Hoarding," *Proc. Seventh Ann. ACM/IEEE Int'l Conf. Mobile Computing and Networking (MobiCom '01),* pp. 15-27, July 2001.

[19] D.L. Lee, W.-C. Lee, J. Xu, and B. Zheng, "Data Management in Location-Dependent Information Services: Challenges and Issues," *IEEE Pervasive Computing,* vol. 1, no. 3, pp. 65-72, July-Sept. 2002.

[20] K. Lin, H.V. Jagadish, and C. Faloutsos, "The TV-Tree: An Index Structure for High-Dimensional Data," *Very Large Databases J.,* vol. 3, no. 4, pp. 517-542, 1994.

[21] B.C. Ooi, *Efficient Query Processing in Geographic Information Systems.* Springer Verlag, 1990.

[22] B.C. Ooi, R. Sacks-Davis, and K.J. Mcdonell, "Spatial Indexing in Binary Decomposition and Spatial Bounding," *Information Systems,* vol. 16, no. 2, pp. 211-237, 1991.

[23] Q. Ren and M.H. Dunham, "Using Semantic Caching to Manage Location Dependent Data in Mobile Computing," *Proc. Sixth Ann. ACM/IEEE Int'l Conf. Mobile Computing and Networking (MobiCom 2000),* pp. 210-221, Aug. 2000.

[24] H. Samet, "The Quadtree and Related Hierarchical Data Structures," *ACM Computing Survey,* vol. 16, no. 2, pp. 187-260, June 1984.

[25] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects," *Proc. 13th Int'l Conf. Very Large Data Bases (VLDB '87),* pp. 507-518, 1987.

[26] A.Y. Seydim, M.H. Dunham, and V. Kumar, "Location Dependent Query Processing," *Proc. Second ACM Int'l Workshop Data Eng. for Wireless and Mobile Access (MobiDE '01),* pp. 47-53, May 2001.

[27] D. White and R. Jain, "Similarity Indexing with the SS-Tree," *Proc. 11th IEEE Int'l Conf. Data Eng. (ICDE '95),* 1995.

[28] J. Xu, X. Tang, and D.L. Lee, "Performance Analysis of Location-Dependent Cache Invalidation Schemes for Mobile Environments," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 2, pp. 474-488, Mar./Apr. 2003.

[29] J. Xu, B. Zheng, W.-L. Lee, and D.L. Lee, "Energy-Efficient Index for Querying Location-Dependent Data in Mobile Broadcast Environments," *Proc. 19th IEEE Int'l Conf. Data Eng. (ICDE '03),* pp. 239-250, Mar. 2003.

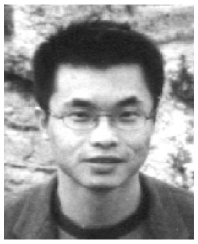
[30] J. Xu, W.-L. Lee, and X. Tang, "Exponential Index: A Parameterized Distributed Indexing Scheme for Data on Air," *Proc. Second ACM/USENIX Int'l Conf. Mobile Systems, Applications, and Services (MobiSys '04),* June 2004.

[31] C. Yu, B.C. Ooi, K.-L. Tan, and H.V. Jagadish, "Indexing the Distance: An Efficient Method to KNN Processing," *Proc. 27th Int'l Conf. Very Large Data Bases (VLDB '01)*, pp. 421-430, Sept. 2001.

[32] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D.L. Lee, "Location-Based Spatial Queries," *Proc. 18th ACM SIGMOD Conf.*, pp. 443-454, June 2003.

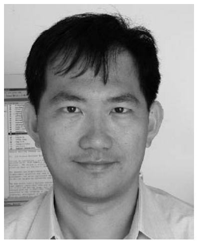
[33] B. Zheng, J. Xu, and D.L. Lee, "Cache Invalidation and Replacement Strategies for Location-Dependent Data in Mobile Environments," *IEEE Trans. Computers*, special issue on database management and mobile computing, vol. 51, no. 10, pp. 1141-1153, Oct. 2002.

[34] B. Zheng, W.C. Lee, and D.L. Lee, "Semantic Cache for Mobile Proximity Search," *ACM Wireless Network*, 2004.



Jianliang Xu received the BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998, and the PhD degree in computer science from Hong Kong University of Science and Technology in 2002. He is an assistant professor in the Department of Computer Science at Hong Kong Baptist University. His research interests include mobile and pervasive computing, Web content delivery, and wireless networks. He has served as a session chair and program committee member for several international conferences including IEEE INFOCOM '04, IEEE MDM '04, and ACM SAC '04. He is a coeditor of a book entitled *Web Content Delivery*, to be published by Kluwer. He also serves as an executive committee member of the ACM Hong Kong Chapter. He is a member of the IEEE.

Baihua Zheng received the PhD degree in computer science from Hong Kong University of Science and Technology. Currently, she is an assistant professor in the School of Information Systems at Singapore Management University. Her research interests include mobile/pervasive computing and spatial databases. She is a member of the IEEE, the IEEE Computer Society, and the ACM.



Wang-Chien Lee received the BS degree from the Information Science Department, National Chiao Tung University, Taiwan, the MS degree from the Computer Science Department, Indiana University, and the PhD degree from the Computer and Information Science Department, the Ohio State University. He is an associate professor in the Computer Science and Engineering Department at Pennsylvania State University. Prior to joining the faculty at Penn State, he was a principal member of the technical staff at Verizon/GTE Laboratories, Inc. His primary research interests lie in the areas of mobile and pervasive computing, data management, wireless networks, and Internet technologies. He has guest-edited special issues on mobile database related topics for several journals, including *IEEE Transactions on Computers*, *IEEE Personal Communications Magazine*, *ACM WINET*, and *ACM MONET*. He was the program committee cochair for the First International Conference on Mobile Data Access (MDA '99) and the International Workshop on Pervasive Computing (PC 2000). He has also been a panelist, session chair, industry chair, and program committee member to various symposia, workshops, and conferences. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Dik Lun Lee received the MS and PhD degrees in computer science from the University of Toronto in 1981 and 1985, respectively. He is a professor in the Department of Computer Science at the Hong Kong University of Science and Technology, and was an associate professor in the Department of Computer and Information Science at the Ohio State University, Columbus, Ohio. He has served as a guest editor for several special issues on database-related topics, and as a program committee member and chair for numerous international conferences. He was the founding conference chair for the International Conference on Mobile Data Management. His research interests include document retrieval and management, discovery, management and integration of information resources on Internet, and mobile and pervasive computing. He was the chairman of the ACM Hong Kong Chapter.