Energy Efficient Index for Querying Location-Dependent Data in Mobile Broadcast Environments

Jianliang Xu[†] Baihua Zheng[†] Wang-Chien Lee[‡] Dik Lun Lee[†]

[†] Hong Kong University of Science and Technology,
Clear Water Bay, Hong Kong
{xujl,baihua,dlee}@cs.ust.hk

[‡] Penn State University, University Park, PA 16802
wlee@cse.psu.edu

Abstract

We are witnessing in recent years growing interest for location-dependent information services among mobile users. This paper examines the issue of processing location-dependent queries in a mobile broadcast environment. Different from a traditional environment, mobile users are concerned with not only access latencies but also power conservation. The planar point location algorithms and conventional spatial index structures are shown inefficient. In this paper, we propose a new index data structure, called *D-tree*, for querying location-dependent data in mobile broadcast environments. The basic idea is to index data regions based on the divisions between them. We describe how to construct the binary D-tree index, how to process location-dependent queries based on this index structure, and how to packet the D-tree to fit the packet capacity. The performance of the D-tree is evaluated using both synthetic and real datasets. Experimental results show that the proposed D-tree provides a much better overall performance than the well-known existing schemes such as the R*-tree.

Keywords: location-dependent data, index structure, power conservation, data broadcast, mobile computing

1 Introduction

We are witnessing in recent years growing interest for location-dependent information services (LDISs) among mobile users, thanks to the rapid technological development in high-speed wireless networks, personal portable devices, and location identification techniques [7, 20, 23]. Examples of mobile LDISs include location-dependent information access (e.g., traffic reports and attractions) and nearest neighbor queries (e.g., finding the nearest restaurant).

Wireless broadcasting is an attractive approach for data dissemination in a mobile environment. Disseminating data through a broadcast channel allows simultaneous access by an arbitrary number of mobile users and thus allows efficient usage of scarce bandwidth. Owing to this scalability feature, the wireless broadcast channel has been considered one alternative storage media (called "air storage") of the traditional hard disks [1, 15]. Many studies addressing various problems in mobile broadcast environments, such as scheduling and bandwidth allocation, have appeared in the literature [1, 12, 15, 22]. It is expected that in the near future a number of mobile LDISs (e.g., region-wide traffic reports and tourism information) will utilize broadcast for the dissemination of information to the rapidly increasing population of mobile users. This paper investigates the issue of querying location-dependent data in mobile broadcast environments.

One critical issue for mobile devices is the consumption of battery power [15]. Without any auxiliary information on the broadcast channel, a client may have to access all objects in a broadcast cycle in order to retrieve the desired data. This method requires the client to listen to the broadcast channel all the time, which is power inefficient. *Air indexing* techniques address this issue by pre-computing some index information and interleaving it with the data on the broadcast channel [15]. By first accessing the broadcast index, the mobile client is able to predict the arrival time of the desired data. Thus, it can stay in the *power saving* mode most of the time and tune into the broadcast channel only when the requested data arrives. The drawback of this solution is that broadcast cycles are lengthened due to additional index information. As such, there is a trade-off between access latency and tuning time. Three criteria have been used to evaluate the performance of air indexing techniques [13, 15]:

- *Access Latency*: the period of time elapsed from the moment a mobile client issues a query to the moment when the requested data is received by the client.

- *Tuning Time*: the period of time spent by a mobile client staying *active* in order to obtain the requested data.

- *Indexing Efficiency*: the ratio of the tuning time saved against the non-indexing scheme to the index overhead.

While access latency measures the overhead of an index structure and the efficiency of data and index organization on the broadcast channel, tuning time is frequently used to estimate the power consumption by a mobile client since sending/receiving data is power dominant in a mobile environment [17].[1] Indexing efficiency, which correlates access latency and tuning time, is used to evaluate the efficiency of indexing techniques in terms of minimizing the tuning time while maintaining an acceptable access latency overhead.

Several indexing techniques dedicated to the wireless broadcast channel, i.e., the hashing indexing [14], the signature approach [18], and the hybrid approach [13], have been introduced in the

[1]For systems in which mobile clients are charged on a per-bit basis, tuning time can also be used to measure the access cost.

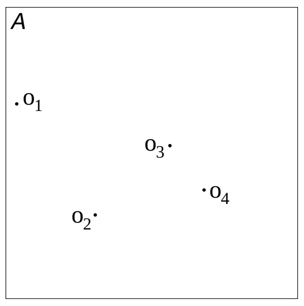
literature. Moreover, imbalanced indexing for skewed access [6] and indexing for multi-attribute queries [12] have also been investigated. However, these studies concentrated on one-dimensional index for equality-based queries, and thus are inapplicable to location-dependent query processing where point queries are involved. Imielinski et al. investigated the issue of interleaving the index and the data on the linear wireless channel such that the tuning time is nearly optimized while maintaining the access latency as short as possible [14, 15]. Recently, there is a study [11] that discussed query processing for spatial objects over the broadcast channel. However, its main focus was on how to utilize the limited client cache to reduce the tuning time when traversing spatial index trees. To the best of our knowledge, the issue of querying location-dependent data in a mobile broadcast environment has not been addressed before.

In this paper, we are interested in exploring efficient index structures for broadcasting location-dependent data on air. We first review some existing index structures. Through an illustration with some simple examples, we show their limitations for querying location-dependent data in mobile broadcast environments. Next, we present a new index data structure, called *D-tree*. The basic idea is to index data regions (i.e., valid scopes of data instances, formally defined in Section 2) based on the divisions between them. We describe how to construct the binary D-tree index, how to process location-dependent queries based on this index structure, and how to packet the D-tree to fit the packet capacity. The performance of the D-tree is evaluated using both synthetic and real datasets. Experimental results show that the proposed D-tree provides a much better overall performance than the well-known schemes such as the R*-tree.

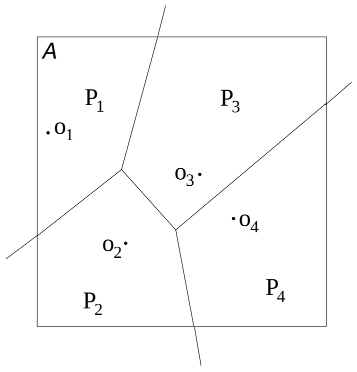
The rest of this paper is organized as follows. Section 2 gives the background on the information system model and the index broadcast model. In Section 3, we review some existing index structures that can be used for querying location-dependent data. Section 4 presents the proposed D-tree index structure along with the partition, query processing, and packeting algorithms. The performance evaluation is presented in Section 5. Finally, Section 6 concludes this paper.

2 Preliminaries

This section provides some background on location-dependent services and information broadcast systems. It is assumed that location-dependent services are provided to mobile clients by an information broadcast system. We refer to the geographical area covered by the system as the *service area*, denoted by $\mathcal{A}$. Mobile clients are assumed to be able to identify their locations using systems such as the global positioning system (GPS). A mobile client can move freely from one location to another within the service area and make queries ubiquitously with its current location or a future location to the broadcast system.



(a) Object Distribution



(b) Diagram of Valid Scopes

Figure 1: Running Example

A *data type* refers to the type of location-dependent information service (e.g., traffic report or nearest restaurant). In this paper, we treat each data type individually and consider the query processing issues for one data type. The dataset is collection of N data instances that return an answer to the query according to the location it is bound. A data instance has a certain *valid scope*, within which this instance is the only correct answer. For example, if the Euclidean distance function is used, given the four data objects, o_1, o_2, o_3, and o_4, in Figure 1(a), Figure 1(b) shows their valid scopes, P_1, P_2, P_3, and P_4, regarding nearest neighbor search.[2] Given any point, say in P_3, o_3 is always the nearest object to that point. 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, the problem of querying location-dependent data is how to efficiently determine which data instance to return regarding a query location. In this paper, we focus on this problem in a two-dimensional space since this is assumed in most LDISs. To study this problem, in the following we formally introduce the concept of *data region.* A data region is usually assumed to take the shape of a polygon.

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

As can be seen, one salient feature for data regions is that they are connected and adjacent to each other. This study aims at investigating efficient index structures to support location-dependent

[2]Note that for nearest neighbor search, as a special type of location-dependent queries, one alternative solution is to index the object positions and perform queries over the positions. This paper discusses general location-dependent queries where only data instances and their valid scopes (bound by streets, roads, etc.) are available, but uses nearest neighbor search as the example to illustrate ideas for it is easy to be understood.

query processing on a wireless broadcast channel: *given data regions $P_1, P_2, \cdots, P_N$, how to index all the regions such that point queries can be processed efficiently in terms of access latency and tuning time*? For interleaving the data and the index on the wireless channel, the $(1, m)$ technique [15] is employed (see Figure 2). That is, the index is broadcast preceding every $\frac{1}{m}$ fraction of the broadcast cycle. As in [15], to reduce the tuning time, each index segment (except for the root) and each data segment contain a pointer pointing to the root of the next index. The access protocol for querying location-dependent data involves the following steps:

- Initial probe: The client tunes into the broadcast channel and determines when the next index will be broadcast. It then turns into the power saving mode until the next index arrives.
- Index search: The client searches the index. It follows a sequence of pointers (i.e., selectively tunes into the broadcast index) to locate the approximate data region and find out when to tune into the broadcast channel to get the desired data. It waits for the arrival of the data in the power saving mode.
- Data retrieval: The client tunes into the channel when the desired data arrives and downloads the data.

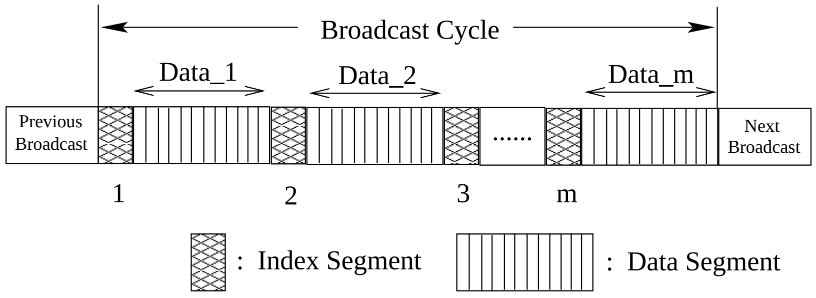


Figure 2: Data and Index Organization on the Broadcast Channel Using the (1, m) Interleaving Technique

In wireless communications, a bit stream is normally delivered in the unit of *packet* (or *frame*), for the purposes such as error-detecting, error-correction, and synchronization [15]. For example, in the GPRS network a packet can contain the data of up to 1600 bytes [5]. As a result, data are accessed by clients also in the unit of packet. Thus, the tuning time for an index structure is measured in terms of number of packet accesses [12, 15].

3 Review of Existing Approaches

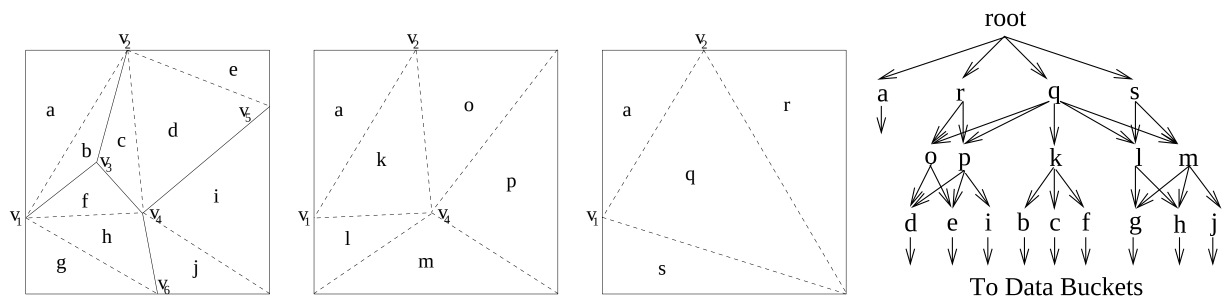
In general, indices based on simple shapes perform efficiently. Thus, the actual shape of a spatial object is often abstracted before being inserted to the index. To serve this purpose, there are two categories of techniques: *object decomposition* and *object approximation* [4, 9]. With object decomposition, the shape of each object is represented as the geometric union of simpler shapes such as triangles and trapezoids. With object approximation, regular shapes such as bounding rectangles or spheres are used to approximate spatial objects. In the following subsections, we briefly introduce several typical solutions and analyze their limitations when they are applied to location-dependent queries in a mobile broadcast environment.

3.1 Object Decomposition

The object decomposition technique has been used to address the planar point location problem [3]: given a polygonal subdivision of the plane (n vertices and m segments) and a query point p, how can we efficiently determine which face of the subdivision contains p? This is similar to our problem. However, we concern not only the search efficiency (which contributes to the tuning time) but also the access latency (which is attributed to the index storage size). As a result, the point location algorithms perform poorly overall as we will show in the performance evaluation later in this paper. There is a long stream of research on the point location problem. Kirkpatrick's algorithm [16] and trapezoidal map [3] are two typical solutions. The former achieves an $O(\log n)$ search time and an $O(n)$ space, while the latter is a randomized algorithm and has an expected search time of $O(\log m)$ and an expected space of $O(m)$. However, the constant factors in these worst performance measures are large, especially for the space measure [3].

In Kirkpatrick's algorithm, the original subdivision is first triangulated. Then we recursively remove some vertices, along with all the edges adjacent to them, and re-triangulate the new subdivision. This operation is continued until the number of triangles contained in the space is smaller than some predefined threshold (T_{min}). Figures 3(a), 3(b), and 3(c) show the triangulation processing for our running example, where T_{min} is set to five. From 3(a) to 3(b), vertices v_3, v_5, and v_6 are removed; and from 3(b) to 3(c), vertex v_4 is removed. A hierarchical index tree, as shown in Figure 3(d), is built upon the triangles generated in the course of recursive triangulation. Given a query point, the probe begins at the root. It checks each child sequentially until a triangle containing the query point is found, then continues the search from that node all the way down to a leaf node. In this paper, we call the index structure built using Kirkpatrick's algorithm the *trian-tree*.

In trapezoidal map, the planar subdivision is viewed as a collection of line segments. The index structure is built along with inserting the set of line segments in the subdivision. When inserting a line segment, from each new vertex we draw two vertical extension lines, one going upwards and the other going downwards. The extension does not stop until it meets an existing



(a) Triangulation of the Original Space

(b) Re-triangulation after v_3, v_5, and v_6 Removed

(c) Re-triangulation after v_4 also Removed

(d) The Index Structure

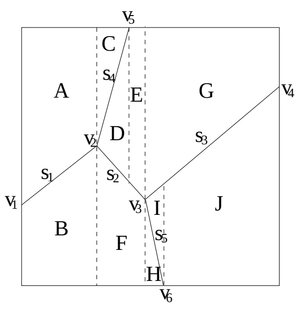
Figure 3: Index Construction Using Kirkpatrick's Algorithm (the Trian-Tree)

line segment. Eventually the original subdivision is decomposed into a set of trapezoids. Obviously the insertion order of line segments influences the index structure. A randomized incremental approach is employed in our experiments [3]. Figure 4(a) shows the final trapezoidal map of our example, where the segments are inserted in the order of s_1, s_2, s_3, s_4, and s_5. The index structure is thus obtained in Figure 4(b). There are two kinds of nodes in the index structure: one is *x-node* (in the shape of a circle) recording the x-coordinate of a vertex; the other is *y-node* (in the shape of a hexagon) recording a line segment. 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. We call the index structure built using trapezoidal map the *trap-tree* in this paper. For both the trian-tree and the trap-tree, an intuition is that their index sizes are probably large. Even for our example with only four regions and six vertices, the index trees have about 10 nodes. This will significantly increase the broadcast cycle and result in a long access latency in a wireless broadcast environment.

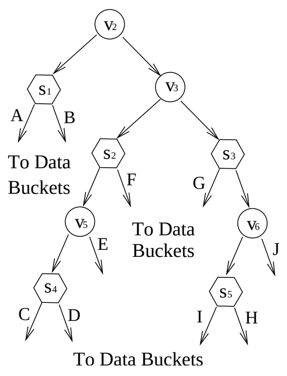
3.2 Object Approximation

The object approximation technique has been commonly employed for disk indexing in the spatial database field [9]. The problem is that given a very large amount of spatial objects, efficient disk indices[3] need to be created to support various spatial operations. Since the index data on a disk are accessed in the unit of *page* and the disk access time is pre-dominant in the performance, the objective of an index structure is often to minimize the number of page accesses. This seems similar to our case that we have to access the index on air in the unit of *packet* and we want to optimize the number of packet accesses. However, it is worth noting that the motives are different:

[3]Due to a large amount of objects, the main memory may not be able to accommodate the index structure so that it has to be stored on a disk.



(a) Final Trapezoidal Map of the Running Example



(b) The Index Structure

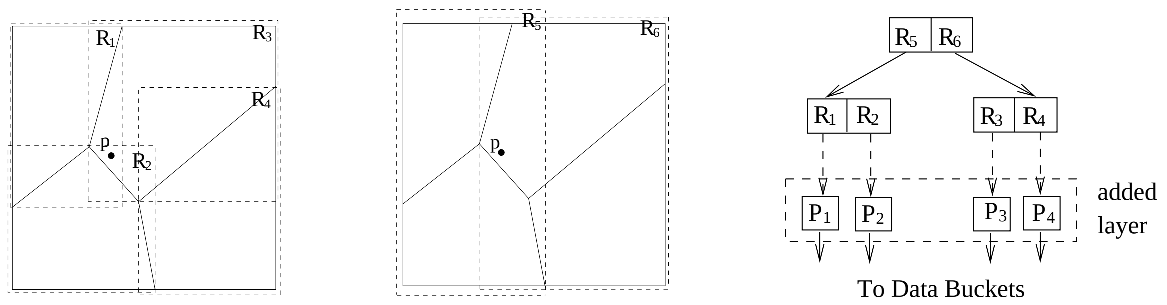
Figure 4: Index Construction Using Trapezoidal Map (the Trap-Tree)

disk indexing is required only for large databases, whereas packet-based data access is a "physical" requirement for wireless communications and thus in our case minimizing packet accesses is required for databases of any size.[4] In addition, as we will illustrate in a moment, the nature of our problem renders the approximation-based spatial index structures inefficient.

One of the most classical spatial index structures is the R-tree [10]. There are also some variants such as the R$^+$-tree [21] and the R*-tree [2]. The basic idea is to use a minimal bounding rectangle (MBR) to approximate a spatial object, and the index is established based on a sequence of MBRs. 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 its data objects. Different variants of the R-tree differ from each other in the criteria used to insert an object and to split an overflow node. Extensive experiments conducted in [2] showed that the R*-tree gained a superior performance for different types of queries and operations. Figure 5(c) shows the structure of the R*-tree (excluding the added layer) for our example, where the corresponding MBRs are shown in Figures 5(a) and 5(b). 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 contain p. When a leaf node is reached, bounding rectangles are tested and their objects are fetched to verify whether they

[4]This probably explains the fact that the indexing issue for location-dependent data has not got enough attention in the spatial indexing research: since the dataset size for one type of LDIS is relatively small, it is believed that the main-memory-based index structures such as the point location algorithms can perform well in traditional environments.



(a) MBRs of Data Regions for the Running Example

(b) MBRs in the Internal Nodes of the Index

(c) The Index Structure

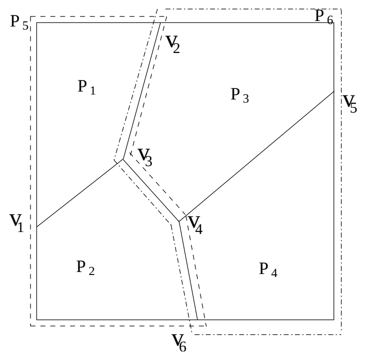
Figure 5: Index Construction Using the R*-tree

contain p. When applying the R*-tree to the wireless broadcast scenario, to save the tuning time, we modify the tree structure slightly as follows. One layer is added to the R*-tree at the bottom as shown in Figure 5(c). This layer consists of the actual shapes of data regions so that in the containment test expensive downloading of actual data is avoided.

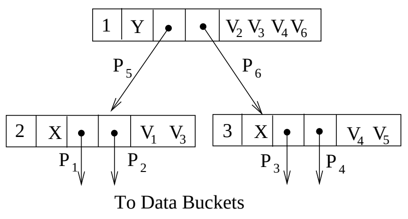
As can be seen, the problem with the R-type tree is that if a point is covered by two or more sibling MBRs it may need to explore several subtrees before the wanted object can be located. This will increase the search time and hence the tuning time. Unfortunately, since in our scenario all data regions are connected, this overlapping problem is inherent after approximating them to MBRs. As an example, suppose the query point is p in Figure 5. The search first reaches the leaf node R_1 through the root and R_5. Since it is outside P_1 pointed by R_1, it rolls the search back to R_2. Likewise, it is outside of P_2 and next rolls the search back to R_6. Finally, it obtains the correct answer in node R_3. Thus, 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 contained in P_3.

4 The D-tree Index Structure

This section describes the index data structure of the proposed D-tree. In the following, we first present the overall idea of the D-tree in Section 4.1. The partition algorithm is described in Section 4.2. Section 4.3 provides the algorithm for location-dependent query processing based on the D-tree index structure. Finally, Section 4.4 explains how to packet the binary D-tree to fit the packet capacity.



(a) Divisions for the Running Example



(b) The D-tree Index

Figure 6: Index Construction Using the D-tree

4.1 An Overall Picture

As discussed in the last section, the object decomposition and approximation approaches suffer from a long access latency and/or a long tuning time. In the meantime, we observe that the actual shapes of data regions are contained 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 better to add one layer for the region shapes to avoid unnecessary data downloading and possible access delays. Therefore, we propose a new data 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 height-balanced tree similar to the kd-tree [9]. However, the D-tree is built based on the divisions of data regions instead of the hyperplanes for the kd-tree. For a space that consists of a set of data regions, we recursively partition it into two *complementary* subspaces containing the same number of regions until each space has one region only. The partition of two subspaces is represented by a set of divisions between regions, i.e., one or more polylines. The overall orientation of the partition (hereafter denoted as *partition dimension*) can be either vertical (y-dimensional) or horizontal (x-dimensional). Figure 6(a) 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 overall and the rest two are x-dimensional. The algorithm of finding the partition for a space will be described in Section 4.2.

The data structure of a D-tree node (formally termed as *index bucket*) is illustrated in Figure 7. The meaning of each attribute is summarized in Table 1. In the D-tree, an internal node contains the

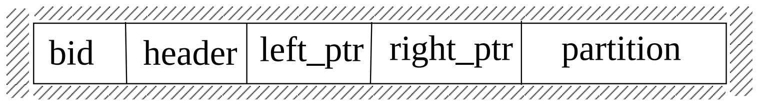


Figure 7: Data Structure of the D-tree Node (Index Bucket)

Attribute	**Description**
bid	the unique id for each node in the tree structure
header	include the flag indicating if the bucket occupied more than one packet, the style and the size of the partition
left_ptr	the type (data or node pointer) and the offset to the beginning of left child
right_ptr	the type (data or node pointer) and the offset to the beginning of right child
partition	a sequence of coordinates that represent the partition of left and right children

Table 1: Illustration of the Attributes in an Index Bucket

partition that divides the current space into two complementary subspaces, a left (right) pointer pointing to the node containing the data regions that lie in the lefthand (righthand) subspace, and some control parameters including *bid* and *header*. A leaf node contains the partition of two data regions, pointers pointing to the data buckets corresponding to the regions, and the control parameters as well. Thus, a spatial data region is inferred by the partitions when following the path from the root towards a leaf node. Note that in the data structure of Figure 7 we place the pointers before the partition on purpose. We will explain this in Section 4.4. For the moment, there is no difference for where the pointers are placed. The binary D-tree satisfies the following four properties (the correctness was proved in [24]).

1. Every node has exactly two children.

2. All objects in the left subtree of a node are in the lefthand subspace of the partition, and all objects in the right subtree are in the righthand subspace.

3. The tree is height-balanced, i.e., all leaves appear at about the same level, and one-level difference at most.

4. The search time for a point query is $\Theta(\log N)$ in terms of nodes visited, where N is the number of data regions in the original space.

The D-tree index structure for the running example is depicted in Figure 6(b), where the header attribute is simplified and only contains the partition dimension. When compared to the trian-tree in Figure 3 and the trap-tree in Figure 4, the size of the D-tree is much smaller. When compared to the R*-tree in Figure 5, the search time for the D-tree is expected shorter since it is a two-node search for any query point.

4.2 Partition Algorithm

Finding a good partition for a space is crucial to the efficiency of the D-tree. This subsection describes the proposed space partition algorithm. There are many ways of dividing one space into two complementary subspaces that contain 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 up-most y-coordinates, leftmost x-coordinates, or down-most y-coordinates and perform the subspace identification similarly. Moreover, if N is odd, we may identify the first $(N+1)/2$ regions as the first subspace. Consequently, four partition styles are evaluated when N is even and eight when N is odd.

For each partition style, the size of the partition is measured in terms of number of coordinates that represent the partition. In selecting among different partition styles, we choose the one with the smallest partition size. For the tie breaking purpose, we define *inter-prob* of two subspaces as the probability of a query being issued from their interacting part. For example, when the partition dimension is y, the inter-prob is the probability of a query being issued from the area between the rightmost x-coordinate of the lefthand subspace and the leftmost x-coordinate of the righthand subspace (e.g., D_2 in Figure 8(a)). Ties are broken by favoring the one with the lowest inter-prob. The reason will become clear when the reader proceeds to the next two subsections and we will mention it again later on in Section 4.4.

We now describe the algorithm, *PartitionSize* (Algorithm 1), that takes an array of data regions and the partition style as the input and evaluates the partition size with this style. For ease of illustration, the algorithm is presented for the style in which the partition dimension is y, the sorting is based on the 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 (line 1-3), we identify the lefthand subspace and construct the extent for this subspace. This is straightforward. However, it is worth noting that the extent of one subspace could be one or more closed polygons. In the second phase (line 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* (line 6-8). In addition, we truncate the remaining segments by *right_lmc* (line 9-15). Note that this operation does not change the partition size, but it identifies *right_lmc* in the partition. Figure 8(a) shows our example, where the extent of the lefthand subspace is the union of the dash-dot line and the solid

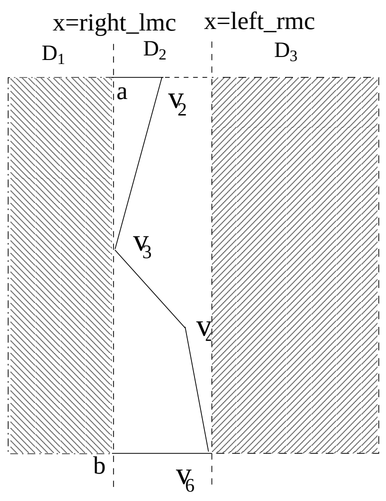
Algorithm 1 PartitionSize: Evaluate the Size of the Partition

```
Input: an array of data regions,
       the partition style   // assuming $N/2$ regions in the left, $y$-dimensional partition,
                             // and rightmost x-coordinate sorting in the following procedure
Output: the partition and its size
Procedure:
1: sort the regions in an 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 number
    of coordinates.
```

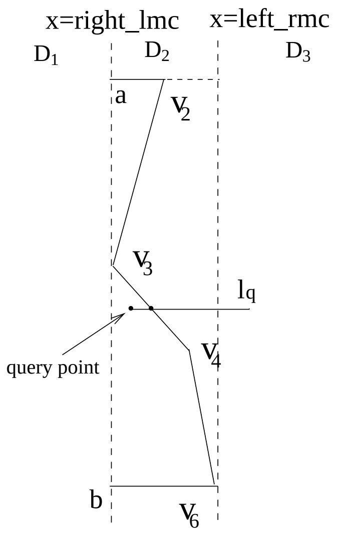
line. The output partition of Algorithm 1 for this example would be the solid line. The complexity of Algorithm 1 is O($N\log N+M$), where N is the number of regions and M is the number of line segments. Therefore, the complexity of the recursive partition procedure for the original space is O($N\log^2 N+M\log N$).

4.3 Query Processing Algorithm

This subsection presents the algorithm for processing location-dependent point queries based on the D-tree index structure. As we discussed in Section 2, the D-tree administers data regions (valid scopes) of a collection of data instances. Thus, given a query point, the query processing problem becomes equivalent to the problem of locating the appropriate data region pointing to the actual data. The search algorithm works as follows. It starts from the root, and recursively follows either the left pointer or the right pointer according to the partition until a data pointer is reached. Since the D-tree is a binary height-balanced tree and there is no spatial overlapping among sibling nodes, the search time is $\Theta(\log N)$ in terms of nodes visited. The procedure is described in Algorithm 2, where the partition style is assumed the y-dimensional partition and the rightmost x-coordinate sorting. It is trivial to extend the algorithm to other partition styles.



(a) Partition



(b) Query Processing

Figure 8: Examples with the Partition and Query Processing Algorithms

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 Figure 8(a), we can see that after a partition a space is divided into three parts: D_1, D_2, and D_3, where D_1 is bound by the leftmost x-coordinate of the righthand subspace (i.e., $right\_lmc$) and D_3 is by the rightmost x-coordinate of the lefthand subspace (i.e., $left\_rmc$). If p falls in D_1 (i.e., $p.x < right\_lmc$), it goes to the lefthand subspace (line 7-10). If p falls in D_3 (i.e., $p.x > left\_rmc$), it goes to the righthand subspace (line 11-14). Otherwise, p falls in D_2 and it has to be determined by examining the partition since D_2 is shared by both subspaces (line 15-26). This operation is illustrated by an example. Suppose that in Figure 8(b) the solid line is the partition and that the query point is p. Consider a horizontal ray l_q emanating from p and to the right. If the number of times that this ray l_q intersects the line segments making up the partition is odd then p is to left of the partition; otherwise if the number is even then p lies to the right of the partition. In Figure 8(b), it turns out that the number is one. Thus, we know the query point is in the lefthand subspace.

4.4 Packeting of the D-tree

As discussed in Section 2, wireless data are accessed in the unit of packet. Thus, it needs to allocate the nodes of the binary D-tree into packets of fixed size. In this paper, we employ a top-down approach to carry out packet allocation [19]. The algorithm works as follows. The D-tree is traversed in a breadth-first order. When a new node arrives, if inclusion of the new node in

Algorithm 2 Location-Dependent Query Processing

Input: the D-tree and the query point p

Output: the pointer to the correct data instance

Procedure: // assuming y-dimensional partition and
// rightmost x-coordinate sorting in the following 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 by ptr;
4:     determine if the query point p is to the left or to the right of the partition (line 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.
```

the packet where the parent node is allocated does not exceed the packet capacity, the new node is allocated space in this packet. Otherwise, a new packet is allocated to the tree and the new node is allocated in the beginning of this packet. We do not split a tree node unless it has to (i.e., for nodes with sizes larger than the packet capacity), because splitting a small node will result in two-packet access instead of one if not splitting. In addition, to save storage size, at last we merge some partial packets at the leaf level in a greedy way. The pseudo code of the procedure is described in Algorithm 3. This algorithm has a complexity of O(N).

For a large node that occupies more than one packet, the following special arrangement is made. An additional coordinate, RMC (i.e., the rightmost x-coordinate in the partition for y-

Algorithm 3 D-tree Packeting Algorithm

Input: a binary D-tree and the packet capacity *packet_size*
Output: a packeted 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 packet of the parent node then
6:       while $node.size > packet\_size$ do
7:          new a packet;
8:          $node.size := node.size - packet\_size$;
9:       end while
10:      new a packet;
11:      reduce the remaining space of the packet by $node.size$;
12:      allocate the node in the newed packet(s);
13:      for each newed packet fill in other info such as the packet id;
14:   else
15:      allocate the node in the packet of the parent node;
16:      reduce the remaining space of the packet by $node.size$;
17:   end if
18: end while
19: $remain := 0$;
20: for all packets at the leaf level do
21:    if $remain <$ occupied space of the current packet then
22:       $remain$ := remaining space of the current packet;
23:    else
24:       merge the current packet with the last packet;
25:    end if
26: end for
```

dimensional partition or the down-most y-coordinate for x-dimensional partition), is inserted before the partition, and the partition starts with the point of LMC (i.e., the leftmost coordinate in the partition for y-dimensional partition or the up-most coordinate for x-dimensional partition). The previous example in Figure 8(a) is used to demonstrate the advantage of this arrangement. For query points falling in D_1 and D_3, once knowing RMC and LMC we can detect which pointer to follow in the first packet and hence prevent further packet access (which reduces the tuning time). Consequently, as a support to this early termination of packet access, we place the pointers before the partition as discussed in Section 4.1. In addition, it is obvious that for two partition styles of the same size, the lower the inter-prob of two subspaces, the higher the probability of successful early detection of the next pointer. For nodes with sizes larger than the packet capacity, this will result in less index search time. Therefore, in the space partition algorithm (Section 4.2), we break ties by favoring the partition style with a lower inter-prob.

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

Notation	**Definition**
N	number of data instances
s_p	packet capacity
p_i	query probability for data instance i
$(n_{i,1}, \cdots, n_{i,m_i})$	the set of nodes on the path from the root to the leaf 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 *interacting part* of two subspaces of node $n_{i,j}$ (e.g., D_2 in Figure 8(a))

Table 2: Definition of Notations

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

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

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

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

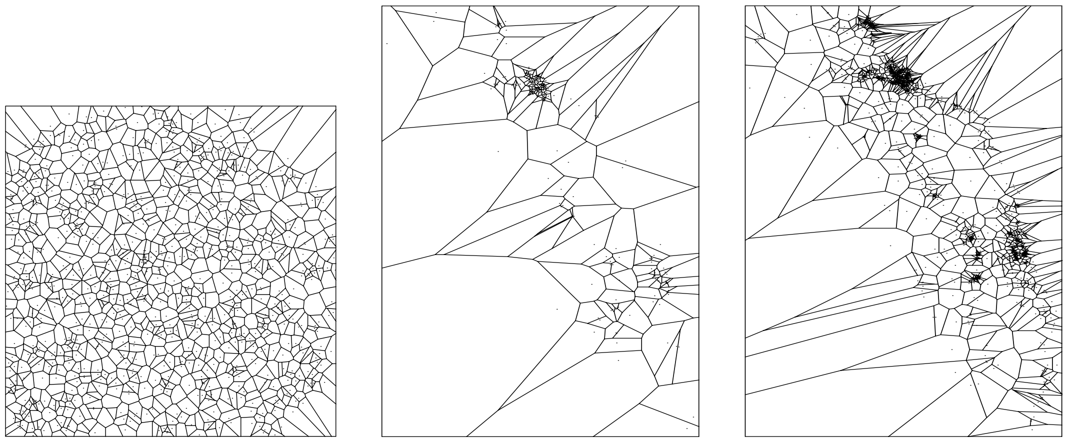
Therefore, the expected search time of the packeted D-tree is obtained as follows:

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

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

5 Performance Evaluation

This section presents the performance evaluation results for the proposed D-tree index structure. Three datasets are used in the experiments (see Figure 9). In the first dataset (UNIFORM), we randomly generate 1000 points in a square Euclidean space. The second set (HOSPITAL) and the third set (PARK) contain the positions of the hospitals and parks in the Southern California area, which are extracted from the point dataset available from [8]. The valid scopes (or data regions) of the points regarding nearest neighbor search, as shown in the figures, are constructed using the voronoi diagram approach [3]. The distributions of the data regions in the latter two datasets are highly clustered. A uniform access distribution over data points is assumed in the evaluation.



(a) UNIFORM: Uniform Distribution (N=1000)

(b) HOSPITAL: Hospitals in Southern CA (N=185)

(c) PARK: Parks in Southern CA (N=1102)

Figure 9: Datasets Used for Performance Evaluation

Parameter	**D-Tree**	**Trian-Tree**	**Trap-Tree**	**R*-Tree**
bid size	2 bytes	2 bytes	2 bytes	2 bytes
header size	2 bytes	0	0	0
pointer size (each)	4 bytes	4 bytes	4 bytes	2 bytes
coordinate size	4 bytes			
data instance size	1K bytes			
packet capacity	64 bytes - 2K bytes			

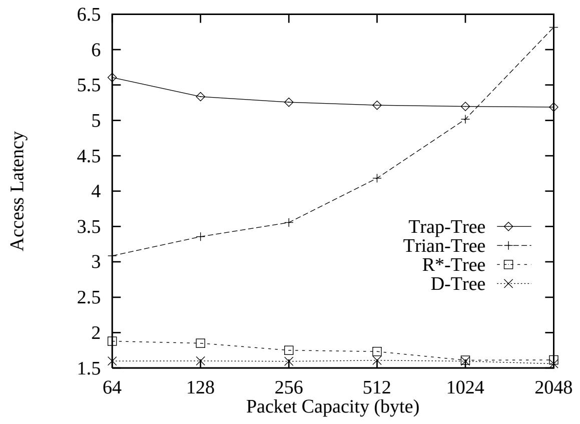
Table 3: System Parameters Setting

The D-tree is compared to the trian-tree, the trap-tree, and the R*-tree index methods. As in the D-tree, the nodes in the trian-tree and the trap-tree do not fit the packet capacity either. Similar to the D-tree in Section 4.4, we packet the trap-tree using the top-down packeting approach. For the trian-tree, the nodes are packeted greedily as they are traversed in a breadth-first order. This is because in the trian-tree a node may be pointed by more than one parent node (refer to Figure 3), making the top-down packeting approach impractical. For the R*-tree, it is obvious that a better search approach is to exploit candidate packets in a depth-first order such that once a containment test in leaf nodes evaluates to true the search can be terminated without accessing useless branches. We employ this search method for the R*-tree in the experiments. The trap-tree and the D-tree is broadcast on the wireless channel in a breadth-first order to facilitate merging of the leaf nodes. The trian-tree is broadcast in a breadth-first order due to the aforementioned reason. The R*-tree is broadcast in a depth-first order to enable rollback operations.

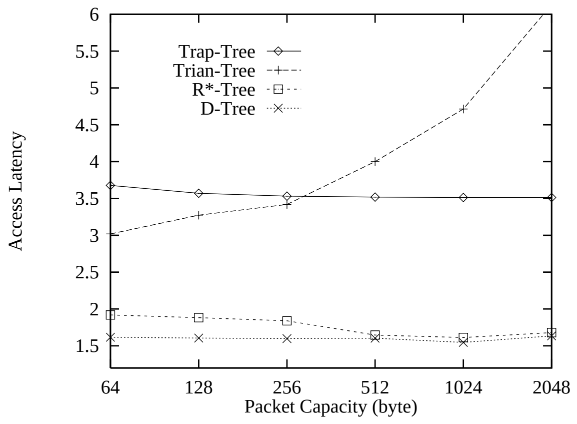
The system parameters for the evaluation are set as in Table 3. For the trian-tree and the trap-tree, the header size is set to 0 since the size of a triangle or a segment is fixed. Hence, there is no need to specify the size of each partition. For the R*-tree, the pointer size is set to 2 bytes since its nodes fit well the packet capacity and a pointer is just the offset to the beginning of the packet containing its child. The header info is also unnecessary in the R*-tree.

We assume that *flat* broadcast is employed for broadcasting data on the wireless channel. The $(1, m)$ interleaving technique [15] is used to interleave the index and the data on the broadcast 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 [15]. The results are obtained based on 1,000,000 randomly generated queries. In the following several subsections, we present the results in terms of access latency, tuning time, and indexing efficiency respectively.

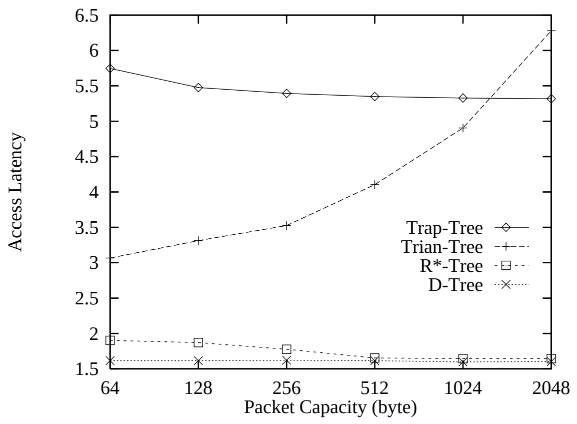
5.1 Access Latency



(a) UNIFORM



(b) HOSPITAL



(c) PARK

Figure 10: Expected Access Latencies for the Three Datasets

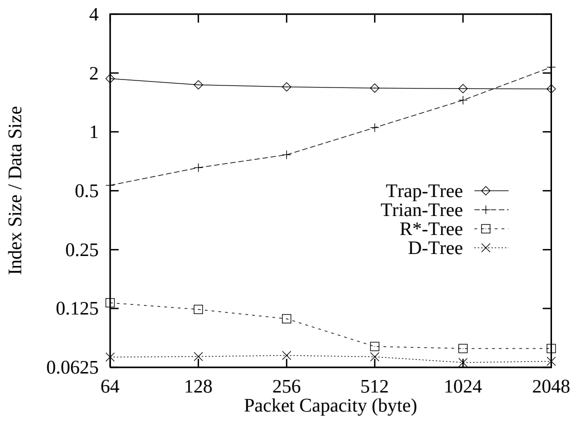


Figure 11: Normalized Index Sizes for the PARK Dataset

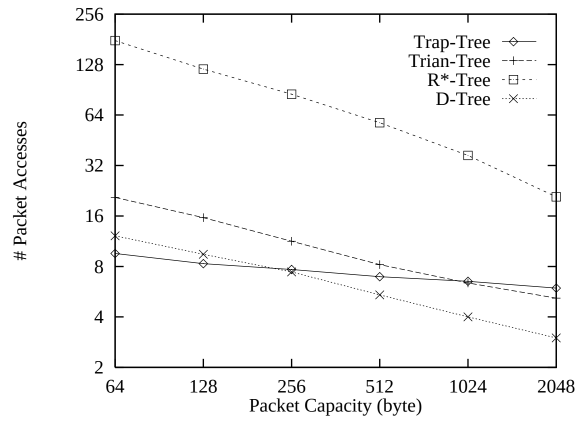
This subsection evaluates the access latencies of the various index methods. Figures 10(a), 10(b), and 10(c) show the results as a function of packet capacity for the three datasets. The latencies in the results are normalized to the expected access latency without any index (or called optimal access latency, i.e., half of the time needed to broadcast the database). The access latency is affected by the index size. The larger the index size, the longer the access latency. Figure 11 shows the normalized index sizes of the index methods for the PARK dataset. Comparing Figure 11 and Figure 10(c), we can see that the relative performance in index size and access latency is consistent.

Let's compare the performance of different index structures. From Figure 10, the trian-tree and the trap-tree have the worst performance, with an latency of several times of the optimal latency. This indicates that these two index structures are almost impractical unless they can provide an extremely good tuning time and we concern the tuning time only. As expected, the D-tree gets the access latency as short as the R*-tree, and is much better than the R*-tree for a small packet capacity. The access latency overhead due to the D-tree indexing is maintained at a similar level for all settings of the packet capacity. It is about 50% worse than the optimal latency in all three datasets. We expect that the index overhead at this level is acceptable provided that a good tuning time is achieved.

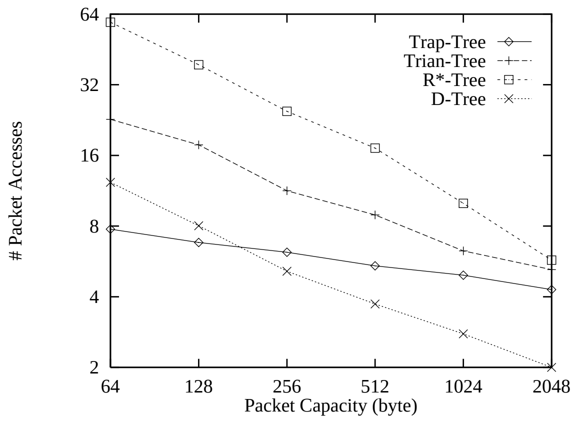
5.2 Tuning Time

This subsection investigates the tuning time for the index structures. To have a close comparison, we measure the tuning time only for the index search step since the tuning time for the steps of initial probe and data retrieval is the same for all the methods. Figure 12 shows the results.

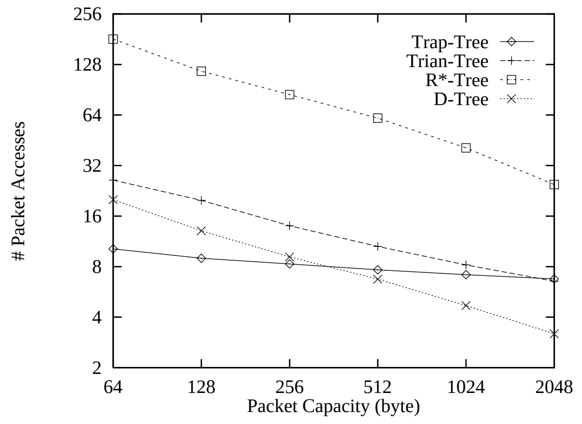
As can be seen, the R*-tree does not have a good performance because of the overlapping problem mentioned before, which makes clients often search more than one leaf node before the wanted



(a) UNIFORM



(b) HOSPITAL

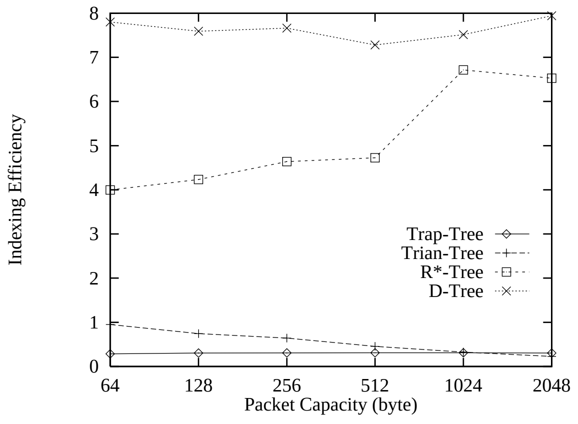


(c) PARK

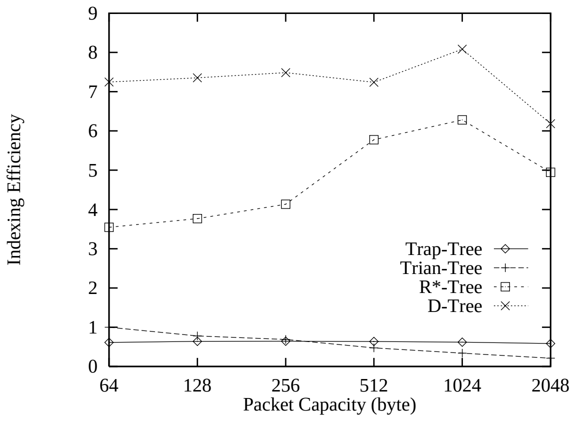
Figure 12: Tuning Time for the Three Datasets

pointer can be reached. For all the three datasets, the D-tree gains a much better performance than the trian-tree and the trap-tree when the packet has a capacity larger than 256 bytes. When the packet capacity is smaller than 256 bytes, the D-tree performs slightly worse than the trap-tree. This can be explained as follows. Although both the D-tree and the trap-tree have a similar average search time (i.e., log$N$ nodes), in the D-tree the sizes of the partitions for the nodes at the highest levels might be a little bit larger. Thus, for a small packet capacity it needs a few packets to accommodate a single node, which leads to its worse performance. When the packet capacity increases, the binary D-tree can take advantage of the top-down packaging approach and package a large number of branches into a single packet. This decreases the tree height greatly and reduces the tuning time. However, the trap-tree, because of its large index size (see Figure 11), cannot compress the tree a lot. As a result, the tuning time of the D-tree is only about half of that of the trap-tree when the packet has a capacity of 2K bytes.

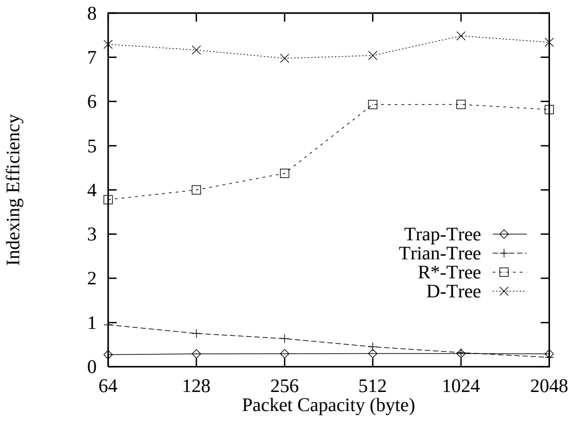
5.3 Indexing Efficiency



(a) UNIFORM



(b) HOSPITAL



(c) PARK

Figure 13: Indexing Efficiency for the Three Datasets

We now evaluate the index structures in terms of indexing efficiency. The larger the indexing efficiency, the more worthy the index. The results for the three datasets are shown in Figure 13(a), 13(b), and 13(c).

Due to an extremely large index size, the trap-tree has the worst performance, although it can provide a short tuning time for a small packet capacity in Figure 12. With a medium index size, the trian-tree performs better than the trap-tree. However, the performance of these two index structures is far from that of the R*-tree and the D-tree. The proposed D-tree is superior in all cases. This means that the best tradeoff between tuning time and index size is achieved by the D-tree. This is expected since, as shown in the last two subsections, the D-tree can provide a very good tuning time while maintaining the index overhead at a reasonable level.

6 Conclusion

While LDISs are becoming increasingly popular among mobile users, data broadcast provides an elegant scalability to an unlimited client population. It is natural to employ data broadcast to disseminate location-dependent data (such as region-wide information) to mobile clients. In this paper, we have studied the issue of querying location-dependent data in a mobile broadcast environment.

Through careful analysis of some existing index structures, we found these indices not efficient for LDISs implemented in a mobile broadcast environment. A new index structure, called D-tree, has been proposed. Different from the existing approaches, the D-tree neither decomposes nor approximates data regions, rather indexes them directly based on the divisions between the regions. The partition algorithm, the query processing algorithm, and the packeting algorithm for the D-tree have been described.

We have evaluated the performance of the proposed D-tree thoroughly using both synthetic and real datasets. The following results are obtained. In terms of access latency, the D-tree substantially outperforms the planar point algorithms (i.e., the trian-tree and the trap-tree) and maintains a similar level of index overhead to the R*-tree. In terms of tuning time, the D-tree index shows a much better performance than the trap-tree for a large packet capacity, and the R*-tree and the trian-tree in all cases. It performs slightly worse than the trap-tree only when the packet capacity is very small. As a result, the D-tree provides a much better overall performance than the other index structures in terms of indexing efficiency. As such, the D-tree index is recommended for practical use in querying location-dependent data in mobile broadcast environments.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 199–210, San Jose, CA, USA, May 1995.

[2] N. Beckmann and H.-P. Kriegel. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 322–331, 1990.

[3] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry.* New York: Springer-Verlag, 1997.

[4] 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.

[5] J. Cai and D. J. Goodman. General packet radio service in GSM. *IEEE Communications Magazine*, 35(10):122–131, October 1997.

[6] M.-S. Chen, P. S. Yu, and K.-L. Wu. Indexed sequential data broadcasting in wireless mobile computing. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97)*, pages 124–131, Baltimore, MD, USA, May 1997.

[7] K. Cheverst, N. Davies, K. Mitchell, and A. Friday. Experiences of developing and deploying a context-aware tourist guide: The GUIDE project. In *Proceedings of the 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'2000)*, pages 20–31, Boston, MA, USA, August 2000.

[8] Spatial Datasets. Website at http://dias.cti.gr/~ytheod/research/datasets/spatial.html.

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

[10] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 47–54, 1984.

[11] S. E. Hambrusch, C.-M. Liu, W. G. Aref, and S. Prabhakar. Query processing in broadcasted spatial index trees. In *Proceedings of the 7th International Symposium on Spatial and Temporal Databases (SSTD'01)*, pages 502–521, Los Angeles, CA, July 2001.

[12] Q. L. Hu, W.-C. Lee, and D. L. Lee. Power conservative multi-attribute queries on data broadcast. In *Proceedings of the 16th International Conference on Data Engineering (ICDE'2000)*, pages 157–166, San Diego, CA, USA, February 2000.

[13] Q. L. Hu, W.-C. Lee, and D. L. Lee. A hybrid index technique for power efficient data broadcast. *Distributed and Parallel Databases (DPDB)*, 9(2):151–177, March 2001.

[14] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Power efficiency filtering of data on air. In *Proceedings of the 4th International Conference on Extending Database Technology (EDBT'94)*, pages 245–258, Cambridge, UK, March 1994.

[15] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Data on air - organization and access. *IEEE Transactions of Knowledge and Data Engineering (TKDE)*, 9(3), May-June 1997.

[16] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 15(2):28–35, 1983.

[17] R. Kravets and P. Krishnan. Power management techniques for mobile communication. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'98)*, pages 157–168, Dallas, TX, USA, October 1998.

[18] W.-C. Lee and D. L. Lee. Using signature techniques for information filtering in wireless and mobile environments. *Journal of Distributed and Parallel Databases (DPDB), Special Issue on Database and Mobile Computing*, 4(3):205–227, July 1996.

[19] B. C. Ooi, R. Sacks-Davis, and K. J. Mcdonell. Spatial indexing in binary decomposition and spatial bounding. *Information Systems*, 16(2):211–237, 1991.

[20] Q. Ren and M. H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of the 6th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'2000)*, pages 210–221, Boston, MA, USA, August 2000.

[21] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB'87)*, pages 507–518, 1987.

[22] J. Xu, D. L. Lee, and B. Li. On bandwidth allocation for data dissemination in cellular mobile networks. *ACM/Kluwer Journal of Wireless Networks (WINET), to appear*, 2002.

[23] J. Xu, X. Tang, and D. L. Lee. Performance analysis of location-dependent cache invalidation schemes for mobile environments. *IEEE Transactions on Knowledge and Data Engineering (TKDE), to appear*, 2002.

[24] J. Xu, B. Zheng, W.-C. Lee, and D. L. Lee. The D-tree: A new index structure for querying location-dependent data. Technical Report, Department of Computer Science, Hong Kong Univ. of Science and Technology, March 2002.