Energy-Conserving Air Indexes for Nearest Neighbor Search

Baihua Zheng[1] Jianliang Xu[2] Wang-Chien Lee[3] Dik Lun Lee[4]

[1] Singapore Management University, Singapore bhzheng@smu.edu.sg
[2] Hong Kong Baptist University, Hong Kong xujl@comp.hkbu.edu.hk
[3] Penn State University, University Park, PA 16802, USA wlee@cse.psu.edu
[4] Hong Kong University of Science and Technology, Hong Kong dlee@cs.ust.hk

Abstract. A location-based service (LBS) provides information based on the location information specified in a query. *Nearest-neighbor* (NN) search is an important class of queries supported in LBSs. This paper studies energy-conserving air indexes for NN search in a wireless broadcast environment. Linear access requirement of wireless broadcast weakens the performance of existing search algorithms designed for traditional spatial database. In this paper, we propose a new energy-conserving index, called *grid-partition index*, which enables a single linear scan of the index for any NN queries. The idea is to partition the search space for NN queries into grid cells and index all the objects that are potential nearest neighbors of a query point in each grid cell. Three grid partition schemes are proposed for the grid-partition index. Performance of the proposed grid-partition indexes and two representative traditional indexes (enhanced for wireless broadcast) is evaluated using both synthetic and real data. The result shows that the grid-partition index substantially outperforms the traditional indexes.

Keywords: mobile computing, location-based services, energy-conserving index, nearest-neighbor search, wireless broadcast

1 Introduction

Due to the popularity of personal digital devices and advances in wireless communication technologies, location-based services (LBSs) have received a lot of attention from both of the industrial and academic communities [9]. In its report "IT Roadmap to a Geospatial Future" [13], the Computer Science and Telecommunications Board (CSTB) predicted that LBS will usher in the era of pervasive computing and reshape mass media, marketing, and various aspects of our society in the decade to come. With the maturation of necessary technologies and the anticipated worldwide deployment of 3G wireless communication infrastructure, LBSs are expected to be one of the killer applications for wireless data industry.

In the wireless environments, there are basically two approaches for provision of LBSs to mobile users:[5]

[5] In this paper, mobile users, mobile clients, and mobile devices are used interchangeably.

– **On-Demand Access**: A mobile user submits a request, which consists of a query and its current location, to the server. The server locates the requested data and returns it to the mobile user.
– **Broadcast**: Data are periodically broadcast on a wireless channel open to the public. Instead of submitting a request to the server, a mobile user tunes into the broadcast channel and filters out the data based on the query and its current location.

On-demand access employs a basic client-server model where the server is responsible for processing a query and returning the result directly to the user via a dedicate point-to-point channel. On-demand access is particularly suitable for light-loaded systems when contention for wireless channels and server processing is not severe. However, as the number of users increases, the system performance deteriorates rapidly. On the other hand, wireless broadcast, which has long been used in the radio and TV industry, is a natural solution to solve the scalability and bandwidth problems in pervasive computing environments since broadcast data can be shared by many clients simultaneously. For many years, companies such as Hughes Network System have been using satellite-based broadcast to provide broadband services. The *smart personal objects technology* (SPOT), announced by Microsoft at the 2003 International Consumer Electronics Show, has further ascertained the industrial interest on and feasibility of utilizing wireless broadcast for pervasive data services. With a continuous broadcast network (called *DirectBand Network*) using FM radio subcarrier frequencies, SPOT-based devices such as watches, alarms, etc., can continuously receive timely, location-specific, personalized information [5]. Thus, in this paper, we focus on supporting LBSs in the wireless broadcast systems.

A very important class of problems in LBSs is *nearest-neighbor (NN) search*. An example of a NN search is: "Show me the nearest restaurant." A lot of research has been carried out on how to solve the NN search problem for spatial databases [12]. Most of the existing studies on NN search are based on indexes that store the locations of the data objects (e.g., the well-known R-tree [12]). We call them *object-based indexes*. Recently, Berchtold et. al. proposed a method for NN search based on indexing the pre-computed solution space [1]. Based on the similar design principle, a new index, called *D-tree*, was proposed by the authors [14]. We refer this category of indexes as *solution-based indexes*. Both of object-based indexes and solution-based indexes have some advantages and disadvantages. For example, object-based indexes have a small index size, but they sometimes require backtracking to obtain the result. This only works for random data access media such as disks but, as shown later in this paper, does not perform well on broadcast data. Solution-based indexes overcome the backtracking problem and thus work well for both random and sequential data access media. However, they in general perform well in high-dimensional space but poorly in low-dimensional space, since the solution space generally consists of many irregular shapes to index.

The goal of this study is to design a new index tailored for supporting NN search on wireless broadcast channels. Thus, there are several basic requirements for such a design: 1) The index can facilitate energy saving at mobile clients; 2) The index is access and storage efficient (because the index will be broadcast along with the data); 3) The index is flexible (i.e., tunable based on a weight between energy saving and access latency; and 4) A query can be answered within one linear scan of the index. Based on the above design principles, we propose a new energy-conserving index called *Grid-Partition Index* which novelly combines the strengths of object-based indexes and solution-based indexes. Algorithms for constructing the grid-partition index and processing NN queries in wireless broadcast channel based on the proposed index are developed.

The rest of this paper is organized as follows. Section 2 introduces air indexing for a wireless broadcast environment and reviews existing index structures for NN search. Section 3 explains the proposed energy-conserving index, followed by description of three grid partition schemes in Section 4. Performance evaluation of the Grid-Partition index and two traditional indexes is presented in Section 5. Finally, we conclude the paper with a brief discussion on the future work in Section 6.

2 Background

This study focuses on supporting the NN search in wireless broadcast environments, in which the clients are responsible for retrieving data by listening to the wireless channel. In the following, we review the air indexing techniques for wireless data broadcast and the existing index structures for NN search. Throughout this paper, the Euclidean distance function is assumed.

2.1 Air Indexing techniques for Wireless Data Broadcast

One critical issue for mobile devices is the consumption of battery power [3, 8, **?**]. It is well known that transmitting a message consumes much more battery power than receiving a message. Thus, data access via broadcast channel is more energy efficient than on-demand access. However, by only broadcasting the data objects, a mobile device may have to receive a lot of redundant data objects on air before it finds the answer to its query. With increasing emphasis and rapid development on energy conserving functionality, mobile devices can switch between *doze mode* and *active mode* in order to conserve energy consumption. Thus, *air indexing* techniques, aiming at energy conservation, are developed by pre-computing and indexing certain auxiliary information (i.e., the arrival time of data objects) for broadcasting along with the data objects [8]. By first examining the index information on air, a mobile client is able to predict the arrival time of the desired data objects. Thus, it can stay in the doze mode to save energy most of the time and only wake up in time to tune into the broadcast channel when the requested data objects arrive.

A lot of existing research focuses on organizing the index information with data objects in order to improve the service efficiency. A well-known data and index organization for wireless data broadcast, called $(1, m)$ interleaving technique [8]. As shown in Figure 1, a complete index is broadcasted preceding every $\frac{1}{m}$ fraction of the broadcast cycle, the period of time when the complete set of data objects is broadcast. By replicating the index for m times, the waiting time for a mobile device to access the forthcoming index can be reduced. The readers should note that this interleaving technique can be applied to any index designed for wireless data broadcast. Thus, in this paper, we employ the $(1, m)$ interleaving scheme for our index.

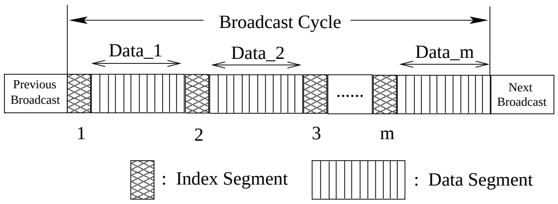


Fig. 1. Data and Index Organization Using the (1, m) Interleaving Technique

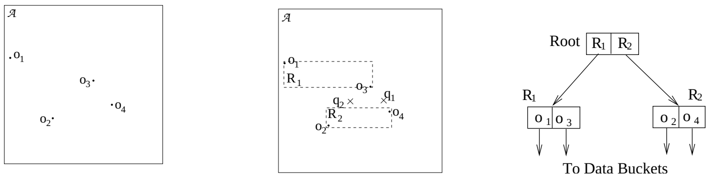
Two performance metrics are typically used for evaluation of air indexing techniques: **tuning time** and **access latency**. The former means the period of time a client staying in the active mode, including the time used for searching the index and the time used for downloading the requested data. Since the downloading time of the requested data is the same for any indexing scheme, we only consider the tuning time used for searching the index. This metric roughly measures the power consumption by a mobile device. To provide a more precise evaluation, we also use **power consumption** as a metric in our evaluation. The latter represents the period of time from the moment a query is issued until the moment the query result is received by a mobile device.

2.2 Indexes for NN Search

There is a lot of existing work on answering NN search in the traditional spatial databases. As mentioned earlier, existing indexing techniques for NN search can be categorized into object-based index and solution-based index. Figure 2(a) depicts an example with four objects, o_1, o_2, o_3, and o_4, in a search space $\mathcal{A}$. This running example illustrates different indexing techniques discussed throughout this paper.

Object-Based Index The indexes in this category are built upon the locations of data objects. R-tree is a representative [7]. Figure 2(c) shows R-tree for the running example. To perform NN search, a branch-and-bound approach is employed to traverse the index tree. At each step, heuristics are employed to choose the next branch for traversal. At the same time, information is collected to prune the future search space. Various search algorithms differ in the searching order and the metrics used to prune the branches $[4, 12]$.

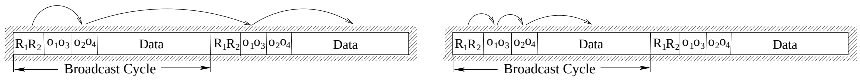
Backtracking is commonly used in search algorithms proposed for traditional disk-access environment. However, this practice causes a problem for the linear-access broadcast channels. In wireless broadcast environments, index information is available to the mobile devices only when it is on the air. Hence, when an algorithm retrieves the index packets in an order different from their broadcast sequence, it has to wait for the next time the packet is broadcast (see the next paragraph for details). In contrast, the index for traditional databases is stored in resident storages, such as memories and disks. Consequently, it is available anytime.



(a) Object Distribution (b) MBR Structures (c) R-tree Index

Fig. 2. A Running Example and R-tree Index

Since the linear access requirement is not a design concern of traditional index structures, the existing algorithms do not meet the requirement of energy efficiency. For example, the index tree in Figure 2(c) is broadcast in the sequence of root, R_1, and R_2. Given a query point p_2, the visit sequence (first root, then R_2, finally R_1) results in a large access latency, as shown in Figure 3(a). Therefore, the branch-and-bound search approach is inefficient in access latency. Alternatively, we may just access the MBRs sequentially (see Figure 3(b)). However, this method is not the best in terms of index search performance since unnecessary MBR traversals may be incurred. For example, accessing R_1 for q_1 is a waste of energy.



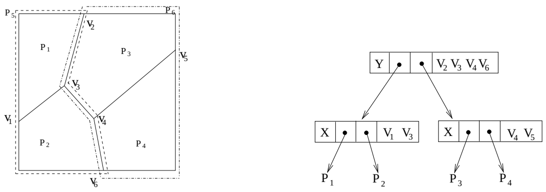
(a) Branch-and-Bound Search (b) Sequential Search

Fig. 3. Linear Access on a Wireless Broadcast Channel

Solution-Based Index The indexes in this category are built on the pre-computed solution space, rather than on the objects [1]. For NN search, the solution space can be represented by *Voronoi Diagrams* (VDs) [2]. Let $O = \{o_1, o_2, \cdots, o_n\}$ be a set of points. $\mathcal{V}(o_i)$, the *Voronoi Cell* (VC) for o_i, is defined as the set of points q in the space such that $dist(q, o_i) < dist(q, o_j)$, $\forall\ j \neq i$. The VD for the running example is depicted in Figure 4(a), where P_1, P_2, P_3, and P_4 denote the VCs for the four objects, o_1, o_2, o_3, and o_4, respectively.

With a solution-based index, the NN search problem can be reduced to the problem of determining the VC in which a query point is located. Our previously proposed index, D-tree, has demonstrated a better performance for indexing solution-space than traditional indexes, and hence is employed as a representative of indexes in this category [14]. D-tree indexes VCs based on the divisions that form the boundaries of the VCs. For a space containing a set of VCs, D-tree recursively partitions it into two sub-spaces having similar number of VCs until each space only contains one VC[6]. D-tree for the running example is shown in Figure 4(b).

In summary, existing index techniques for NN search are not suitable for wireless data broadcast. An object-based index incurs a small index size, but the tuning time is poor because random data access is not allowed in a broadcast channel. On the other hand, a solution-based index, typically used for NN search in a high dimensional space, does not perform well in a low dimensional space due to the fact that efficient structures for indexing VCs are not available. In the following, we propose a new energy-conserving index that combines the strengths of both the object-based and the solution-based indexes.



(a) Divisions

(b) D-tree

Fig. 4. D-tree Index for the Running Example

3 A Grid-Partition Index

In this section, we first introduce the basic idea of our proposal and then describe the algorithm for processing NN search based on the Grid-Partition air index.

3.1 The Basic Idea

In an object-based index such as R-tree, each NN search starts with the whole search space and gradually trims the space based on some knowledge collected during the search process. We observe that an essential problem leading to the poor search performance of object-based index is the large overall search space. Therefore, we attempt to reduce the search space for a query at the very beginning by partitioning the space into disjointed grid cells. To do so, we first construct the whole solution space for NN search using the VD method; then, divide the search space into disjointed grid cells using some grid partition algorithm (three partition schemes will be discussed in Section 4). For each grid cell,

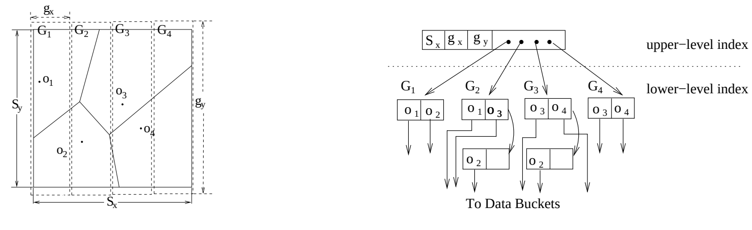
[6] D-tree was proposed to index any pre-computed solution space, not only for NN search.

we index all the objects that are potential nearest neighbors of a query point inside the grid cell. Each object is the nearest neighbor only to those query points located inside the VC of the object. Hence, for any query point inside a grid cell, only the objects whose VCs overlap with the grid cell need to be checked.

Definition 1 *An object is* associated *with a grid cell if the VC of the object overlaps with the grid cell.*

Since each grid cell covers a part of the search space only, the number of objects associated with each grid cell is expected to be much smaller than the total number of objects in the original space. Thus, the initial search space for a NN query is reduced greatly if we can quickly locate the grid cell in which a query point lies. Hence, the overall performance is improved. Figure 5(a) shows a possible grid partition for our running example. The whole space is divided into four grid cells, i.e., G_1, G_2, G_3, and G_4. Grid cell G_1 is associated with objects o_1 and o_2 since their VCs, P_1 and P_2, overlap with G_1; likewise, grid cell G_2 is associated with objects o_1, o_2, and o_3, and so on and so forth.

The index structure for the proposed grid-partition index consists of two levels. The upper-level index is built upon the grid cells, and the lower-level index is upon the objects associated with each grid cell. The upper-level index maps a query point to a corresponding grid cell, while the lower-level index facilitates the access to the objects within each grid cell. The nice thing is that once the query point is located in a grid cell, its nearest neighbor is definitely among the objects associated with that grid cell, thereby preventing any rollback operations and enabling a single linear access of the upper-level index for any query point. In addition, to avoid rollback operations in the lower-level index, we try to maintain each grid cell in a size such that its associated objects can fit into one data packet, which is the smallest transmission unit in wireless broadcast. Thus, for each grid cell a simple index structure (i.e., a list of object and pointer pairs) is employed. In case that the index for a grid cell cannot fit into one packet, a number of packets are sequentially allocated. In each grid cell, the list of object and pointer pairs are sorted according to the dimension with the largest scale (hereafter called *sorting dimension*), i.e., the dimension in which the grid cell has the largest span. For example, in Figure 5(a), the associated objects for grid cell G_2 are sorted according to the y-dimension, i.e., o_1, o_3, and o_2. This arrangement is to speed up the nearest neighbor detecting procedure as we will see in the next subsection.



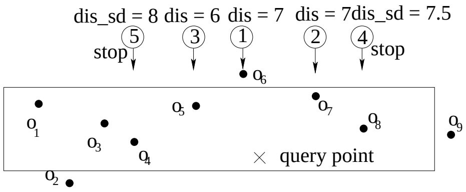
(a) FP (b) Index Structure

Fig. 5. Fixed Grid Partition for the Running Example

3.2 Nearest-Neighbor Search

With a grid-partition index, a NN query is answered by executing the following three steps: 1) *locating grid cell*, 2) *detecting nearest neighbor*, and 3) *retrieving data*. The first step locates the grid cell in which the query point lies. The second step obtains all the objects associated with that grid cell and detects the nearest neighbor by comparing their distances to the query point. The final step retrieves the data to answer the query. In the following, we describe an efficient algorithm for detecting the nearest neighbor in a grid cell. This algorithm works for all the proposed grid partition schemes. We leave the issue of locating grid cell to Section 4. This allows us to treat the problems of partitioning grid and locating grid cells more coherently.

In a grid cell, given a query point, the sorted objects are broken into two lists according to the query point in the sorting dimension: one list consists of the objects with coordinates smaller than the query point, and the rest form the other. To detect the nearest neighbor, the objects in those two lists are checked alternatively. Initially, the current shortest distance *min_dis* is set to infinite. At each checking step, *min_dis* is updated, if the distance between the object being checked and the query point, *cur_dis*, is shorter than *min_dis*. The checking process continues until the distance of the current object and the query point in the sorting dimension, *dis_sd*, is longer than *min_dis*. The correctness is justified as follows. For the current object, its *cur_dis* is longer than or equal to *dis_sd* and, hence, longer than *min_dis* if *dis_sd* is longer than *min_dis*. For the remaining objects in the list, their *dis_sd*'s are even longer and, thus, it is impossible for them to have a distance shorter than *min_dis*.



(a)

Step	Obj_{check}	*dis_sd*	*cur_dis*	*min_dis*
1	o_6	1	7	7
2	o_7	4	7	7
3	o_5	4	6	6
4	o_8	7.5	stop	
5	o_4	8	stop	

(b)

Fig. 6. An Example for Detecting Nearest Neighbor

Figure 6(a) illustrates an example, where nine objects associated with the grid cell are sorted according to the x-dimension since the grid cell is flat. Given a query point shown in the figure, nine objects are broken into two lists, with one containing o_6 to o_1 and the other containing o_7 to o_9. The algorithm proceeds to check these two lists alternatively, i.e., in the order of o_6, o_7, o_5, o_8, $\cdots$, and so on. Figure 6(b) shows the intermediate results for each step. In the first list, the checking stops at o_4 since its *dis_sd* (i.e., 7.5) is already longer than *min_dis* (i.e., 6). Similarly, the checking stops at o_8 in the second list. As a result, only five objects rather than all nine objects are evaluated. Such improvement is expected to be significant when the scales of a grid cell in different dimensions differ greatly.

4 Grid Partitions

Thus far, the problem of NN search has been reduced to the problem of *grid partition*. How to divide the search space into grid cells, construct the upper-level grid index, and map a query point into a grid cell are crucial to the performance of the proposed index. In this section, three grid partition schemes, namely, *fixed partition*, *semi-adaptive partition*, and *adaptive partition*, are introduced. These schemes are illustrated in a two-dimensional space, since we focus on the geospatial world (2-D or 3-D space) in the real mobile environments.

Before presenting grid partition algorithms, we first introduce an important performance metric, *indexing efficiency* η, which is employed in some of the proposed grid partition schemes. It is defined as the ratio of the reduced tuning time to the enlarged index storage cost against a *naive* scheme, where the locations of objects are stored as a plain index that is exhaustively searched to answer a NN query. The indexing efficiency of a scheme i is defined as $\eta(i) = \Big((T_{naive} - T_i)/T_{naive}\Big)^{\alpha} / \Big((S_i - S_{naive})/S_{naive}\Big)$, where T is the average tuning time, S is the index storage cost, and α is a control parameter to weigh the importance of the saved tuning time and the index storage overhead. The setting of α could be adjusted for different application scenarios. The larger the α value, the more important the tuning time compared with the index storage cost. This metric will be used as a performance guideline to balance the tradeoff between the tuning time and the index overhead in constructing the grid-partition index.

4.1 Fixed Partition (FP)

A simple way for grid partition is to divide the search space into fixed-size grid cells. Let S_x and S_y be the scales of the x- and y-dimensions in the original space, g_x and g_y be the fixed width and height of a grid cell. The original space is thus divided into $\frac{S_x}{g_x} \cdot \frac{S_y}{g_y}$ grid cells. With this approach, the upper-level index for the grid cells (shown in Figure 5(b)) maintains some *header* information (i.e., S_x, g_x, and g_y) to assist in locating grid cells, along with a one-dimensional array that stores the pointers to the grid cells. In the data structure, if the header information and the pointer array cannot fit into one packet, they are allocated in a number of sequential packets.

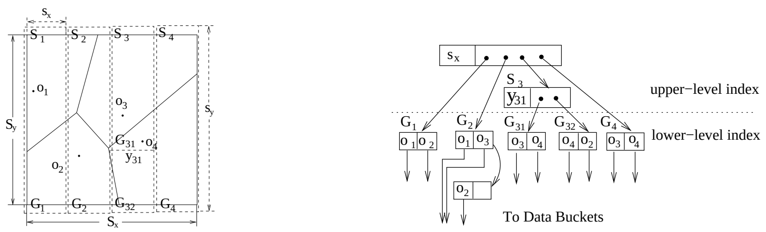
The grid cell locating procedure works as follows. We first access the header information and get the parameters of S_x, g_x, and g_y. Then, given a query point (q_x, q_y), we use a mapping function, $adr(q_x, q_y)=\lfloor \frac{q_y}{g_y} \rfloor \cdot \lceil \frac{S_x}{g_x} \rceil + \lfloor \frac{q_x}{g_x} \rfloor$, to calculate the address of the pointer for the grid cell in which the query point lies. Hence, at most 2-packet accesses (one for the header information and maybe additional one for the pointer if it is not allocated in the same packet) in locating grid cells are needed, regardless of the number of grid cells and the packet size.

Aiming to maximize the packet utilization in the index, we employ a greedy algorithm to choose the best grid size. Let *num* be the number of expected grid cells. We continue to increase *num* from 1 until the average number of objects associated with the grid cells is smaller than the fan-out of a node. Further increasing *num* will decrease the packet occupancy and thus degrade the performance. For any *num*, every possible combination of g_x and g_y such that $\frac{S_x}{g_x} \cdot \frac{S_y}{g_y}$ equals *num*, is considered. The indexing efficiency for the resultant grid partition with width g_x and height g_y is calculated. The grid partition achieving the highest indexing efficiency is selected as the final solution.

While the fixed grid partition is simple, it does not take into account the distribution of objects and their VCs. Thus, it is not easy to utilize the index packets efficiently, especially under a skewed object distribution. Consequently, under the fixed grid partition, it is not unusual to have some packets with a low utilization rate, whereas some others overflow. This could lead to a poor average performance.

4.2 Semi-Adaptive Partition (SAP)

To adapt to skewed object distributions, the semi-adaptive partition only fixes the size of the grid cells in either width or height. In other words, the whole space is equally divided into stripes along one dimension. In the other dimension, each stripe is partitioned into grid cells in accordance with the object distribution. The objective is to increase the utilization of a packet by having the number of objects associated with each grid cell close to the fan-out of the node. Thus, once the grid cell is identified for a query point, only one packet needs to be accessed in the step of detecting nearest neighbor.



(a) SAP (b) Index Structure

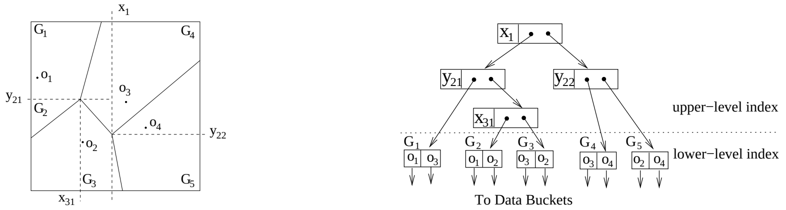
Fig. 7. Semi-Adaptive Partition for the Running Example

Figure 7 illustrates the semi-adaptive grid partition for our running example, where the height of each stripe is fixed. Similar to the FP approach, the root records the width of a stripe (i.e., s_x) for the mapping function and an array of pointers pointing to the stripes. In each stripe, if the associated objects can fit into one packet, the objects are allocated directly in the lower-level index (e.g., the 1st and 4th pointers in the root). Otherwise, an extra index node for the grid cells within the corresponding stripe is allocated (e.g., the 3rd pointer in the root). The extra index node consists of a set of sorted discriminators followed by the pointers pointing to the grid cells. However, if there is no way to further partition a grid cell such that the objects in each grid cell can fit the packet capacity, more than one packet is allocated (e.g., the 2nd pointer in the root).

To locate the grid cell for a query point (q_x, q_y), the algorithm first locates the desired stripe using a mapping function, $adr(q_x, q_y) = \lfloor \frac{q_x}{s_x} \rfloor$. If the stripe points to an object packet (i.e., only contains one grid cell), it is finished. Otherwise, we traverse to the extra index node and use the discriminators to locate the appropriate grid cell. Compared with FP, this partition approach has a better packet occupancy, but takes more space to index the grid cells.

4.3 Adaptive Partition (AP)

The third scheme adaptively partition the grid using a kd-tree like partition method [11]. It recursively partitions the search space into two complementary subspaces such that the number of objects associated with the two subspaces is nearly the same. The partition does not stop until the number of objects associated with each subspace is smaller than the fan-out of the index node.



(a) AP (b) Index Structure

Fig. 8. Adaptive Partition for the Running Example

The partition algorithm works as follows. We partition the the space horizontally or vertically. Suppose that the vertical partition is employed. We sort the objects in an increasing order of the left-most x-coordinates (LXs) of their VCs. Then, we examine the LXs one by one beginning from the median object. Given a LX, the space is partitioned by the vertical line going through the LX. Let num_l and num_r be the numbers of associated objects for the left subspace and the right subspace, respectively. If $num_l = num_r$, the examination stops immediately. Otherwise, all the LXs are tried and the LX resulting in the smallest value of $|num_l - num_r|$ is selected as the discriminator. Similarly, when the horizontal partition is employed, the objects are sorted according to the lowest y-coordinates (LYs) of their VCs and the discriminator is selected much the same way in the vertical partition. In selecting the partition style between the vertical and horizontal partitions, we favor the one with a smaller value of $|num_l + num_r|$. Figure 8 shows the adaptive grid partition for the running example, where each node in the upper-level index stores the discriminator followed by two pointers pointing to two subspaces of the current space.

In this approach, the index for the grid cells is a kd-tree. Thus, the point query algorithm for the kd-tree is used to locate the grid cells. Given a query point, we start at the root. If it is to the left of the discriminator of the node, the left pointer is followed; otherwise, the right pointer is followed. This procedure is not stopped until a leaf node is met. However, as the kd-tree is binary, we need some paging method to store it in a way to fit the packet size. A top-down paging mechanism is employed. The binary kd-tree is traversed in a breadth-first order. For each new node, the packet containing its parent is checked. If that packet has enough free space to contain this node, the node is inserted into that packet. Otherwise, a new packet is allocated.

4.4 Discussion

For NN search, the VD changes when objects are inserted, deleted, or relocated. Thus, the index needs to be updated accordingly. Since updates are expected to happen infrequently regarding NN search in mobile LBS applications (such as finding nearest restaurant and nearest hotel), we only briefly discuss the update issue here.

When an object o_i is inserted or deleted, the VCs around o_i will be affected. The number of affected VCs is approximately the number of edges of the VC for o_i, which is normally very small. For example, in Figure 2(a) adding a new object in P_1 changes the VCs for o_1, o_2, and o_3. Assuming the server maintains adequate information about the VCs, the affected VCs can be detected easily.

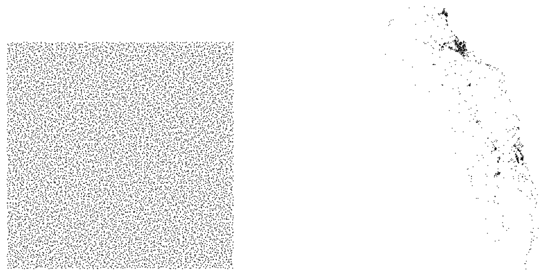
For all the proposed grid partitions, we can identify the grid cells that overlap with the affected VCs and update the index for the objects associated with each of them. When updates are rare, the partition of the grid cells is not modified. On the other hand, partial or complete grid re-partition can be performed periodically.

5 Performance Evaluation

To evaluate the proposed grid-partition index, we compare it with D-tree [14] and R-tree, which represent the solution-based index and the object-based index for NN search respectively, in terms of tuning time, power consumption, and access latency. Two datasets (denoted as UNIFORM and REAL) are used in the evaluation (see Figure 9). In the UNIFORM dataset, 10,000 points are uniformly generated in a square Euclidean space. The REAL dataset contains 1102 parks in the Southern California area, which is extracted from the point dataset available from [6].

Since the data objects are available a priori, the STR packing scheme is employed to build R-tree [10]. As we discussed in Section 2.2, the original branch-and-bound NN search algorithm results in a poor access latency in wireless broadcast systems. In order to cater for the linear-access requirement on air, we revise it as follows. R-tree is broadcast in a width-first order. For query processing, no matter where the query point is located, the MBRs are accessed sequentially, while impossible branches are pruned similarly in the original algorithm [12].

The system model in the simulation consists of a base station, a number of clients, and a broadcast channel. The available bandwidth is set to $100K$ bps. The packet size is varied from 64 bytes to 2048 bytes. In each packet, two bytes are allocated for the packet id. Two bytes are used for one pointer and four bytes are for one coordinate. The size of a data object is set to 1K bytes. The results presented in the following sections are the average performance of $10,000,000$ random queries.



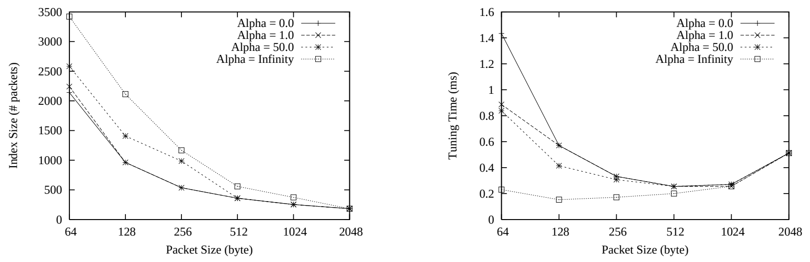
(a) UNIFORM (b) REAL

Fig. 9. Datasets for Performance Evaluation

5.1 Sensitivity to Indexing Efficiency

Indexing efficiency has been used in the FP and SAP grid partition schemes as guidance for determining the best cell partition. The control parameter α of indexing efficiency, set to a non-negative number, weighs the importance of the saved packet accesses and the index overhead. We conduct experiments to test the sensitivity of tuning time and index size to α.

Figure 10 shows the performance of grid-partition index for the UNIFORM dataset when the FP partition scheme is employed. Similar results are obtained for the REAL dataset and/or other grid partitions and, thus, are omitted due to the space limitation. From the figure, we can observe that the value of α has a significant impact on the performance, especially for small packet capacities. In general, the larger the value of α, the better the tuning time and the worse the index storage cost since a larger α value assigns more weight to reducing tuning time. As expected, the best index overhead is achieved when α is set to 0, and the best tuning time is achieved when α is set to infinity. The setting of α can be adjusted based on requirements of the applications. The index overhead for air indexes is also critical as it directly affects the access latency. Thus, for the rest of experiments, the value of α is set to 1, giving equal weight to the index size and the tuning time.

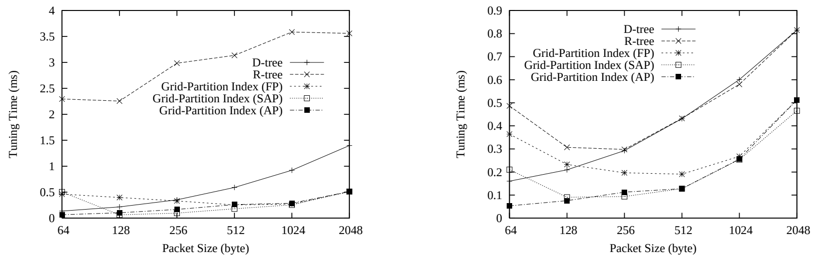


(a) Index Size (b) Tuning Time

Fig. 10. Performance under Different α Settings (UNIFORM, FP)

5.2 Tuning Time

This subsection compares the different indexes in terms of tuning time. In the wireless data broadcast environment, improving the tuning time generally saves power consumption. Figures 11(a) and (b) show the tuning time performance of compared indexes under UNIFORM and REAL datasets, respectively.



(a) UNIFORM (b) REAL

Fig. 11. Tuning Time vs. Packet Size

Two observations are obtained. First, the proposed grid-partition indexes outperform both D-tree and R-tree in most cases. As an example, let's look at the case when the packet size is 512 bytes. For D-tree, the tuning time is $0.59ms$ and $0.43ms$ for UNIFORM and REAL datasets, respectively. For R-tree, it needs $3.12ms$ and $0.43ms$, respectively. The grid-partition indexes have the best performance, i.e., no larger than $0.27ms$ for UNIFORM dataset and no larger than $0.19ms$ for REAL dataset.

Second, among the three proposed grid partition schemes, the SAP has the best overall performance and is the most stable one. The main reason is that the SAP scheme is more adaptive to the distribution of the objects and their VCs than the FP scheme, while its upper-level index (i.e., the index used for locating grid cells) is a simpler and more efficient data structure than that of the AP scheme. As a result, in most cases the SAP accesses only one or two packets to locate grid cells and another one to detect the nearest neighbor.

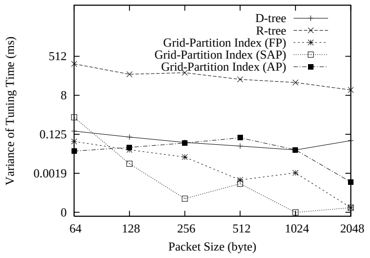


Fig. 12. Variance of Tuning Time (UNIFORM)

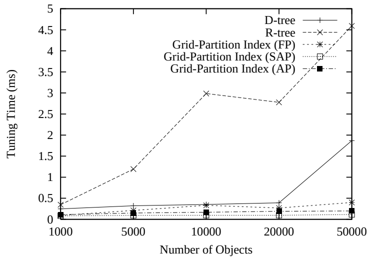


Fig. 13. Performance vs. Size of Datasets

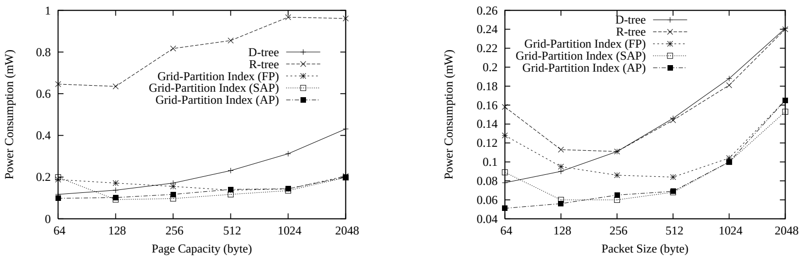
We notice that the grid-partition indexes with FP and SAP work worse than D-tree when the packet size is 64 bytes. This is caused by the small capacity of packet, which can fit in very limited objects information. Hence, the small size of the packet results in a large number of grid cells and causes duplications.

We also measure the performance stableness of the compared indexes. Figure 12 shows the variance of their tuning time for the UNIFORM dataset. It can be observed again that the grid-partition indexes outperform both D-tree and R-tree in nearly all the cases. This means the power consumption of query processing based on grid-partition index is more predictable than that based on other indexes. This property is important for power management of mobile devices.

In order to evaluate the scalability of the compared indexes to the number of data objects, we measure the tuning time of indexes by fixing the packet size to 256 bytes and varying the number of objects from 1,000 to 50,000 (all uniformly distributed). As shown in Figure 13, the larger the population of the objects, the worse the performance as expected. The performance ratings among different indexes under various numbers of data objects are consistent. However, it is interesting to note that the performance degradation of the grid-partition indexes is much more gracefully than that of D-tree and R-tree, as the number of data objects increases. This indicates that the proposed grid-partition indexes are more pronounced for large databases.

5.3 Power Consumption

According to [8], a device equipped with the Hobbit chip (AT&T) consumes around $250mW$ power in the active mode, and consumes $50\mu W$ power in the doze mode. Hence, the period of time a mobile device staying in doze mode during query processing also has an impact on the power consumption. To have a more precise comparison of the power consumption based on various indexes, we calculate the power consumption of a mobile device based on the periods of active and doze modes obtained from our experiments. For simplicity, we neglect other components that consume power during query processing and assume that 250 mW constitutes the total power consumption. Figure 14 shows the power consumption of a mobile device under different air indexes, calculated based on the formula: $P = 250 \times Time_{active} + 0.05 \times Time_{doze}$.



(a) UNIFORM (b) REAL

Fig. 14. Power Consumption vs. Packet Size

As shown in the figure, the grid-partition indexes significantly outperform other indexes. For UNIFORM dataset, the average power consumptions of D-tree are $0.23mW$, and that of R-tree is $0.69mW$. The power consumptions of grid-partition indexes are $0.17mW$, $0.14mW$, and $0.14mW$ for FP, SAP, and SP, respectively. For the REAL dataset, the improvement is also dramatic. The power consumptions of D-tree and R-tree are $0.13mW$ and $0.14mW$, while the grid-partition indexes consume $0.09mW$, $0.08mW$, and $0.06mW$. Although D-tree provides a better tuning time performance when the packet size is 64 bytes, that does not transform into less power consumption than the grid-partition index. This is caused by the large index overhead of D-tree, compared with that of grid-partition indexes (see Figure 15).

In summary, the grid-partition indexes can reduce the power consumption by the efficient search performance and small index overhead. Hence, it can achieve the design requirement of energy efficiency without any doubt and is extremely suitable for the wireless broadcast environments in which the population of users is supposed to be huge while the resources of mobile devices are very limited.

5.4 Access Latency

The access latency is affected by the storage cost of the index and the interleaving algorithm to organize data and index. Since index organization is beyond the scope of this paper, we count the access latency using the well-known (1, m) scheme to interleave the index with data [8], as explained in Section 2.1. Figure 15 shows the access latency for all the index methods. In the figures, the latency is normalized to the expected access latency without any index (i.e., half of the time needed to broadcast the database).

We can see that the D-tree has the worst performance because of its large index size. The performance of those proposed grid-partition indexes is similar to that of the R-tree. They only introduce little latency overhead (within 30% in most cases) due to their small index sizes.

When different grid partition schemes are compared, the FP performs the best for a small packet capacity (< 256 bytes), whereas the SAP and the AP perform better for a large packet capacity (> 256 bytes). This can be explained as follows. When the packet capacity is small, the number of grid cells is large since we try to store the objects with a grid cell in one packet in all the three schemes. Thus, the index size is dominated by the overhead for storing the grid partition information (i.e., the upper-level index). As this overhead in the FP is the least (four parameters plus a pointer array), it achieves the smallest overall index size. However, with increasing packet capacity, the overhead for storing the upper-level index becomes insignificant. Moreover, with a large packet capacity the FP has a poorer packet occupancy than the other two. This is particularly true for the REAL dataset, where the objects are highly clustered. As a result, the index overhead of the FP becomes worse.

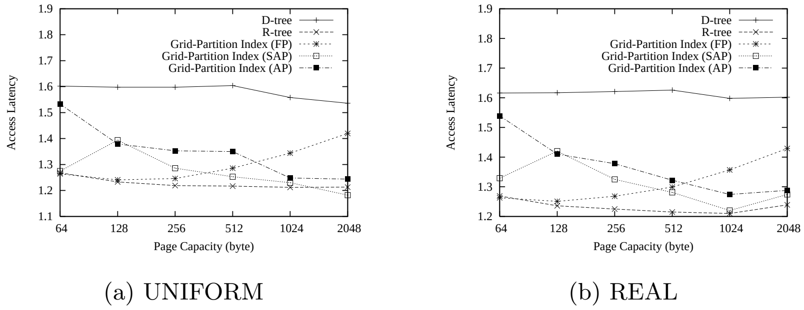


Fig. 15. Access Latency vs. Packet Capacity

6 Conclusion

Nearest-neighbor search is a very important and practical application in the emerging mobile computing era. In this paper, we analyze the problems associated with using object-based and solution-based indexes in wireless broadcast environments, where only linear access is allowed, and enhance the classical R-tree to make them suitable for the broadcast medium. We further propose the grid-partition index, a new energy-conserving air index for nearest neighbor search that combines the strengths of both the object-based and solution-based indexes. By studying the grid-partition index, we identify an interesting and fundamental research issue, i.e., grid partition, which affects the performance of the index. Three grid partition schemes, namely, fixed partition, semi-adaptive partition, and adaptive partition, are proposed in this study.

The performance of the grid-partition index (with three grid partition schemes) is compared with an enhanced object-based index (i.e., R-tree) and a solution-based index (i.e., D-tree) using both synthetic and real datasets. The results show that overall the grid-partition index substantially outperforms both the R-tree and D-tree. As the grid-partition index (SAP) achieves the best overall performance under workload settings, it is recommended for practical use.

Although the grid-partition index is proposed to efficiently solve NN search, it can also serve other queries, such as window queries and continuous nearest neighbor search. As for future work, we plan to extend the idea of the grid-partition index to answer multiple kinds of queries, including k-NN queries. As a first step, this paper only briefly addresses the update issue in a general discussion. We are investigating efficient algorithms to support updates. In addition, we are examining generalized NN search such as "show me the nearest hotel with room rate $< \$200$".

References

1. S. Berchtold, D. A. Keim, H. P. Kriegel, and T. Seidl. Indexing the solution space: A new technique for nearest neighbor search in high-dimensional space. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 12(1):45–57, January/February 2000.
2. M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*, chapter 7. Springer-Verlag, New York, NY, USA, 1996.
3. M-S. Chen, K-L. Wu, and S. Yu. Optimizing index allocation for sequential data broadcasting in wireless mobile computing. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(1), 2003.
4. K. L. Cheung and W.-C. Fu. Enhanced nearest neighbour search on the r-tree. *SIGMOD Record*, 27(3):16–21, 1998.
5. Microsoft Corporation. What is the directband network? URL at http://www.microsoft.com/resources/spot/direct.mspx, 2003.
6. Spatial Datasets. Website at http://dias.cti.gr/∼ytheod/research/datasets/spatial.html.
7. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'84)*, pages 47–54, 1984.
8. T. Imielinski, S. Viswanathan, and B. R. Badrinath. Data on air - organization and access. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 9(3), May-June 1997.
9. D. L. Lee, W.-C. Lee, J. Xu, and B. Zheng. Data management in location-dependent information services. *IEEE Pervasive Computing*, 1(3):65–72, July-September 2002.
10. S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. Str: A simple and efficient algorithm for r-tree packing. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 497–506, Birmingham, UK, April 1997.
11. 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.
12. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, pages 71–79, May 1995.
13. Computer Science and Telecommunications Board. *IT Roadmap to a Geospatial Future*. The National Academies Press, 2003.
14. J. Xu, B. Zheng, W.-C. Lee, and D. L. Lee. Energy efficient index for querying location-dependent data in mobile broadcast environments. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE'03)*, pages 239–250, Bangalore, India, March 2003.