Noname manuscript No.
(will be inserted by the editor)

Grid-Partition Index: A Hybrid Method for Nearest-Neighbor Queries in Wireless Location-Based Services

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

[1] Singapore Management University, 469 Bukit Timah Road, Singapore 259756
[2] Hong Kong Baptist University, Kowloon Tong, Hong Kong
[3] The Penn State University, University Park, PA 16802
[4] Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong

Abstract Traditional nearest-neighbor (NN) search is based on two basic indexing approaches: object-based indexing and solution-based indexing. The former is constructed based on the locations of data objects: using some distance heuristics on object locations. The latter is built on a pre-computed solution space. Thus, NN queries can be reduced to and processed as simple point queries in this solution space. Both approaches exhibit some disadvantages, especially when employed for wireless data broadcast in mobile computing environments.

In this paper, we introduce a new index method, called the *grid-partition index*, to support NN search in both on-demand access and periodic broadcast modes of mobile computing. The grid-partition index is constructed based on the Voronoi Diagram; i.e., the solution space of NN queries. However, it has two distinctive characteristics. First, it divides the solution space into grid cells such that a query point can be efficiently mapped into a grid cell around which the nearest object is located. This significantly reduces the search space. Second, the grid-partition index stores the *objects* that are potential NNs of any query falling within the cell. The storage of objects, instead of the Voronoi cells, makes the grid-partition index a hybrid of the solution-based and object-based approaches. As a result, it achieves a much more compact representation than the pure solution-based approach and avoids backtracked traversals required in the typical object-based approach, thus realizing the advantages of both approaches.

We develop an incremental construction algorithm to address the issue of object update. In addition, we present a cost model to approximate the search cost of different grid partitioning schemes. The performances of the grid-partition index and existing indexes are evaluated using both synthetic and real data. The results show that overall, the grid-partition index signif-

icantly outperforms object-based indexes and solution-based indexes. Furthermore, we extend the grid-partition index to support continuous-nearest-neighbor search. Both algorithms and experimental results are presented.

Key words nearest-neighbor search – continuous-nearest-neighbor search – index structure – location-dependent data – wireless broadcast

1 Introduction

Location-dependent information services (LDISs) refer to services that answer queries based on where the queries are issued. Due to the popularity of personal digital devices and advances in wireless communication technologies, LDISs have received a lot of attention from both the industrial and academic communities [13,18]. In its report "IT Roadmap to a Geospatial Future" [18], the Computer Science and Telecommunications Board (CSTB) predicted that LDISs will usher in the era of mobile/pervasive computing and reshape the mass media, marketing, and various other aspects of our society in the decade to come.

A very important class of problems in LDISs is *nearest-neighbor (NN) search*. An example of NN search is: "Show me the nearest restaurant." A lot of research has been carried out on how to solve the NN search problem in the traditional domain of spatial databases [17,21,26] and research areas involving high-dimensional spaces such as multimedia databases and OLAP [1,5,6,8,16,28]. Different from these efforts, in this paper, we study the NN search problem in wireless environments, which are most attractive to LDIS users [13]. Data access in wireless environments has unique characteristics which are different from those of traditional disk-based environments. In addition to the traditional *on-demand access* supported by point-to-point communication, *periodic broadcast* is an alternative to disseminate data to mobile users (see Section 2.1 for a detailed discussion of these two data access modes). Traditional disk-based indexing techniques can be employed to speed up query processing for on-demand access, while periodically broadcasting the index along with the data can guide the clients to intelligently listen to useful data only. Moreover, disks support random access, whereas the access in periodic broadcast is sequential. In addition, mobile and disk-based environments have different performance requirements; i.e., mobile clients are concerned not only with access latency but also with energy consumption [10,11,27]. This calls for the development of new index and search techniques. On the other hand, we shall focus ourselves on a low dimensional space, because most, if not all, LDISs are based on a two-dimensional space.

Although people are more familiar with point-to-point wireless connections, satellite-based broadcast has been used for many years by companies, such as Hughes Network System, to provide broadband data services. Broadcast dissemination has also been adopted by Microsoft Smart Personal Objects Technology (SPOT) to send timely, location-aware information to

customers via the DirectBand network. This demonstrates the industrial interest as well as commercial feasibility of broadcast methods for large-scale data delivery. Data broadcast allows simultaneous access of data by an arbitrary number of mobile clients at a constant cost, thus increasing the flexibility and scalability of the system. It is efficient in terms of power and resource consumption because broadcast is an inherent capability of wireless communication and does not require sophisticated protocols for channel setup and data exchange, making it very suitable for resource-constrained mobile environments. An additional benefit of broadcast is that clients can fetch any desirable information from the broadcast channel without revealing to the server their specific queries. For example, in the context of this paper, the user can retrieve the nearest objects from a broadcast channel without needing to send the server details of his physical location. Thus, the privacy of the user is protected. These benefits motivate us to investigate in this paper the nearest-neighbor problem in both wireless broadcast and point-to-point environments.

Most of the existing studies on NN search are based on indexes that store the locations of the indexed objects, e.g., the well-known R-tree [17]. 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]. We refer to this as a *solution-based index*. Both object-based indexes and solution-based indexes have advantages and disadvantages. For example, object-based indexes incur a low storage overhead, but they rely on backtracking to obtain query results. Backtracking is not a major constraint for random-access media (e.g., disks) but is a serious problem for sequential-access media (e.g., wireless channels with data broadcast; see Section 2.2 for details). Solution-based indexes overcome the backtracking problem by answering an NN query in a single linear scan of the index. As such, they work well for sequential-access media. Nevertheless, since the solution space typically consists of complex shapes (e.g., polygons), the solution-based index generally has a larger index size than that of the object-based index. Finally, solution-based indexes are tailored for a particular type of queries. They are expected to be incorporated in a spatial DBMS to provide efficient support for popular types of queries such as NN search. This is analogous to relational DBMSs that support both the hash index for efficient equality match and B^+-tree for more general queries.

In a previous paper, we proposed a novel grid-partition index that combines the strengths of object-based indexes and solution-based indexes [29]. The basic idea is as follows. First, the solution space of NN queries is partitioned into disjoint grid cells. For each grid cell, all of the possible NNs for an arbitrary query point within the cell are indexed. This approach effectively shrinks the initial search space to a grid cell by quickly locating the grid cell containing the query point. Moreover, since the object locations (rather than the complex shapes of solution space) are indexed, the index size is small. Three partitioning algorithms, namely *fixed partition*, *semi-adaptive*

partition, and *adaptive partition*, were developed. In this paper, we extend the previous work in the following aspects:

- A new index construction algorithm based on Delaunay Triangulation is proposed to support incremental update of the index. The method described in [29] was directly based on the Voronoi Diagram and update issues were not addressed.
- A cost model is derived to approximate the grid-partition index's performance in terms of index search cost.
- An extensive simulation is conducted to compare the performance of the grid-partition index with representative object-based and solution-based indexes in both air indexing and traditional disk indexing environments, whereas only an air indexing environment was considered in [29]. In addition, this paper includes more datasets and performance metrics to provide a thorough comparison between the grid-partition index and other indexes.
- A continuous-nearest-neighbor search algorithm is developed based on the grid-partition index. The experimental performance results are presented.

The rest of this paper is organized as follows. Section 2 provides the background for supporting NN search on air and analyzes the constraints of existing index structures. Section 3 introduces the basic idea of the grid-partition index, together with an incremental index construction approach, the NN search algorithm, and the cost model. A performance evaluation of the grid-partition index is presented in Section 4. Section 5 extends the usage of the grid-partition index to solve the continuous-nearest-neighbor problem in wireless environments. Finally, we conclude the paper with a roadmap of future work in Section 6.

2 Background

The goal of our study is to address the NN search issue in mobile computing environments, in which the data is delivered via wireless networks. In the following, first, we describe two typical mobile data dissemination approaches in wireless networks and their performance concerns; then, we review existing index structures for NN queries.

2.1 Mobile Data Dissemination and Performance Metrics

LDISs are very attractive in a mobile and wireless environment, where mobile clients enjoy unrestricted mobility and ubiquitous information access [13]. There are basically two approaches to disseminating location-dependent data to mobile clients:

- **On-Demand Access**: A mobile client submits a request, which consists of a query and the query's issuing location, to the server. The server returns the result to the mobile client via a dedicated point-to-point channel.
- **Periodic Broadcast**: Data are periodically broadcast on a wireless channel open to the public. After a mobile client receives a query from its user, it tunes into the broadcast channel to receive the data of interest based on the query and its current location.

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. Compared with on-demand access, broadcast is a more scalable approach since it allows simultaneous access by an arbitrary number of mobile clients.

Access efficiency and *energy conservation* are two critical issues for mobile clients. Access efficiency concerns how fast a request is satisfied, while energy conservation concerns how to reduce a mobile client's energy consumption when it accesses the data of interest. In the literature, two performance metrics, namely access latency and tuning time, are used to measure access efficiency and energy conservation, respectively [10, 11]:

- **Access latency**: The time elapsed between the moment when a query is issued and the moment when it is satisfied.
- **Tuning time**: The time a mobile client stays active to receive the requested data.

While access efficiency is a constantly tackled issue in most system and database research, energy conservation is very critical due to the limited battery capacity on mobile clients, which ranges from only a few hours to about half a day under continuous use. Moreover, only a modest improvement in battery capacity of 20-30% is expected over the next few years [11]. To facilitate energy conservation, a mobile device typically supports two operation modes: *active mode* and *doze mode*. The device normally operates in active mode; it can switch to doze mode to save energy when the system becomes idle.

With on-demand access, the query processing is the same as in traditional client-server mode, except that the query and result are transferred via a wireless network. The client tuning time (for sending the query and receiving the result) is independent of query processing strategies. Hence, the focus of this paper for on-demand access is to employ *disk indexing* on the server to expedite query processing. It is understood that disk I/O rather than CPU is the performance bottleneck for most disk-based database applications. Therefore, the design objective is to minimize the index search cost in terms of the number of index pages accessed during query processing (since indexes are accessed in the unit of *page*). However, the improvement of query latency due to an index structure comes at the cost

of storing and maintaining the index on disk. Fortunately, as disk storage is getting cheaper and bigger, the index storage overhead would not be a major concern.

With data broadcast, clients listen to a broadcast channel to retrieve data based on their queries and hence are responsible for query processing. Without any index information, a client has to download all data objects to process NN search, which will consume a lot of energy since the client needs to remain active during a whole broadcast cycle. A solution to this problem is *air indexing* [11]. The basic idea is to broadcast an index before data objects (see Figure 1 for an example). Thus, query processing can be performed over the index instead of actual data objects. As the index is much smaller than the data objects and is selectively accessed to perform a query, the client is expected to download less data (hence incurring less tuning time and energy consumption) to find the NN. In fact, the tuning time is proportional to the index search cost in terms of the number of index pages accessed during the search. The disadvantage of air indexing however, is that the broadcast cycle is lengthened (to broadcast additional index information). As a result, the access latency would be worsen. It is obvious that the larger the index size, the higher the overhead in access latency.

An important issue in air indexing is how to multiplex data and index on the sequential-access broadcast channel. Figure 1 shows the well-known $(1, m)$ scheme [11], where the index is broadcast in front of every $\frac{1}{m}$ fraction of the dataset. To facilitate the access of index, each data page includes an offset to the beginning of the next index. The general access protocol for processing NN search involves the following steps:

- Initial probe: The client tunes into the broadcast channel and determines when the next index is broadcast.
- Index search: The client tunes into the broadcast channel again when the index is broadcast. It selectively accesses a number of index pages to find out the NN object and when to download it.
- Data retrieval: When the page containing the NN object arrives, the client downloads it and retrieves the NN.

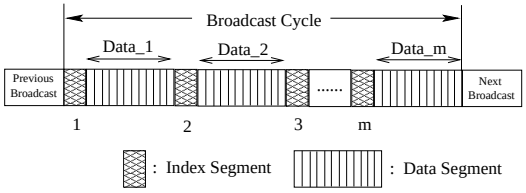


Fig. 1 Air Indexing in Wireless Broadcast Environments

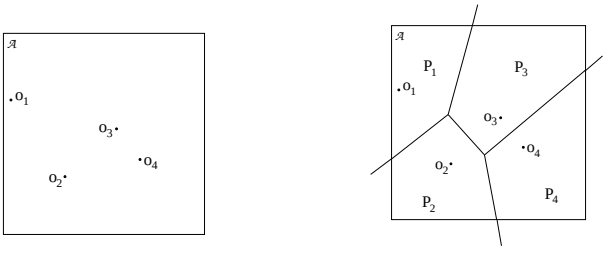
In general, low index search cost and small index size cannot be achieved simultaneously. In order to correlate the index search cost and index size,

we present a flexible performance measure, *indexing efficiency* (denoted by η). It is defined as the ratio of the reduced index search cost to the enlarged index size against a *naive* scheme, where each index segment stores the locations of all the objects and a client needs to conduct an exhaustive search to find its NN object. Formally, the indexing efficiency metric is defined as:

$$\eta(i) = \frac{\Big((T_{naive} - T_i)/T_{naive}\Big)^{\alpha}}{\Big((S_i - S_{naive})/S_{naive}\Big)} \tag{1}$$

where T_i is the average index search cost of an index i, S_i is the size of index i, and α is a constant parameter to weigh the importance of the saved search cost and the index overhead. The setting of α could be adjusted for different application scenarios. The larger the value of α, the more important the index search cost compared with the index size. This metric will be used as a performance guideline to tune the tradeoff between the index search cost and index size in constructing the grid-partition index.

To summarize this section, for on-demand access, the objective of disk indexing is to reduce the latency of query processing; for data broadcast, the objective of air indexing is to trade access latency for tuning time. Although the ultimate performance objectives are different in these two scenarios, they are essentially determined by the index size (for access latency in air indexing) and the index search cost in terms of index page accesses during query processing (for query latency in disk indexing and tuning time in air indexing). Therefore, the goal of this paper is to design new index structures that minimize the index search cost and index size for NN search. In addition, we require the indexes to support efficient NN search on sequential-access broadcast channels.



(a) Object Distribution (b) Voronoi Diagram

Fig. 2 A Running Example

2.2 Indexes for NN Queries

According to the information indexed, the existing index structures for NN search can be classified into two categories: object-based indexes and

solution-based indexes. A running example consisting of four objects in search space $\mathcal{A}$ is introduced to illustrate the basic ideas of these indexes (see Figure 2(a)).

Object-Based Indexes The indexes in this category are built upon the locations of data objects. The representative is R-tree [7], where objects are indexed using minimal bounding rectangles (MBRs). Most of the other indexes in the category were derived from R-tree. The MBRs for the objects in the running example and the corresponding R-tree index are shown in Figure 3, given that the node fan-out is two. If the objects to be indexed are all available, a packing algorithm, such as the Hilbert sort [12] or STR [14], can be employed to build the index so that both the index size and index search cost are reduced by improving the page occupancy.

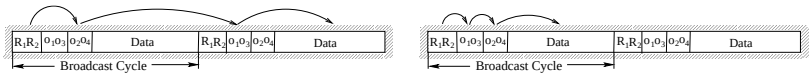
(a) MBR Structures (b) R-tree Index

Fig. 3 R-tree Index for the Running Example

To perform NN search, a branch-and-bound approach is employed to traverse the index tree. At each step, a heuristic is applied to choose the next branch to traverse. At the same time, information is collected to prune the future search space. Various search algorithms differ in terms of the searching order and the metrics used to prune the branches [3,9,17]. For example, suppose that there are two query points, q_1 and q_2, as shown in Figure 3(a). For query point q_1, after accessing the root, it visits R_2 first since it is closer to R_2 than R_1. In R_2, the NN of q_1 is o_4. Hence, it records the current minimal distance, $dist(q_1, o_4)$. Because $dist(q_1, o_4)$ is shorter than the minimum distance from q_1 to R_1, the search is stopped here. Similarly, for query point q_2, first it examines the root and then R_2. Next, it examines R_1 since the current minimal distance, $dist(q_2, o_2)$, is longer than the minimum distance from q_2 to R_1.

As illustrated, NN search for R-tree dynamically traverses the MBRs according to the given query point. This introduces two major weaknesses. First, since it relies on the heuristic to gradually prune the search space, the performance really depends on whether the heuristic has pruning power. A foreseeable effect is the great variance of the search performance. Second, the branch-and-bound search approach involves a lot of backtracking which works well for random-access disks only but not for sequential-access broadcast channels. Let's look at an example. Suppose that the index tree in our

running example is broadcast once in a pre-order traversal in a broadcast cycle (see Figure 4(a)). If we preserve the search order for q_2 (i.e., first the root, followed by R_2, followed by R_1) based on the original R-tree search algorithm, a significant access delay is incurred. This is because after accessing R_2, we have to wait for the next broadcast to access R_1 since R_1 has already been broadcast in the current cycle (as illustrated by the second arc in Figure 4(a)). Such a delay is incurred for every inconsistency between the searching order and the broadcasting order of MBRs. Therefore, the branch-and-bound search approach is inefficient in access latency. Alternatively, we may just access R_1 and R_2 sequentially (see Figure 4(b)). However, this method is not the best in terms of the tuning time since unnecessary accesses may be incurred. For example, accessing R_1 for q_1 is a waste of tuning time.



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

Fig. 4 Sequential Access on a Wireless Broadcast Channel

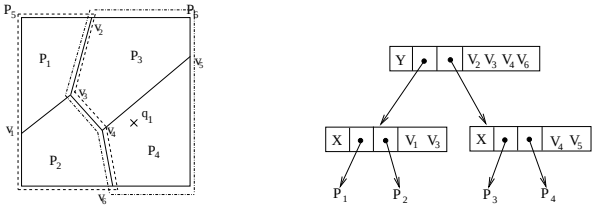
Vector-Approximation (VA) file is another representative index for NN search [25], which was proposed for efficient similarity search in high dimensional spaces. The basic idea is to assign b_j bits to represent the jth dimension; i.e., by dividing the space along the jth dimension into 2^{b_j} partitions, each containing about the same number of objects. The server records a set of marks $(m_j[1], \cdots, m_j[2^{b_j}])$ to capture the partitioning coordinates in the jth dimension. Therefore, the original d-dimensional space is divided into 2^b cells, with b equal to $\sum_{j=1}^{d} b_j$, and each cell is represented by a b-bits vector. Since an object falls into one and only one cell, it can be associated with the cell's vector. With such a flat structure, VA-file reduces the index size and expedites the sequential scan, which is unavoidable when the number of dimensions exceeds 10 [24, 25]. Nevertheless, as a space partitioning method, it performs much worse than R-tree in low-dimensional spaces, where the sequential scan is often not needed. Therefore, R-tree is employed in this paper as a representative of location-based indexes.

Another object-based index structure related to ours is the *grid file*, which was originally designed for multi-key file access [15]. The basic ideas behind the grid file and our proposed grid-partition index are different. In the grid file, the space is partitioned into grid cells according to the objects' distribution and each grid cell indexes only the objects inside its cell. On the contrary, in our proposed index, the space is partitioned into grid cells based on the solution space, and each object might be associated with several grid cells. A grid cell actually indexes all the objects that are potential NNs of

query points inside the cell. The objective of our grid-partition index is to efficiently solve NN search in mobile computing environments.

Solution-Based Indexes An object-based index is a generic index structure that supports various spatial operations such as joins besides NN search, and solution-based indexes are constructed for serving a specific type of queries. Motivated by the observation that the performance of object-based index for NN search decreases as the dimension of space increases, solution-based indexes have been proposed to build an index based on the pre-computed solution space [1]. For an NN search, the solution space can be represented by the *Voronoi Diagram* (VD) [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 $dist(q, o_i) < dist(q, o_j)$, $\forall\ j \neq i$. That is, $\mathcal{V}(o_i)$ consists of the set of points for which o_i is the NN. The VD for the running example is depicted in Figure 2(b), where P_1, P_2, P_3, and P_4 denote the VCs for four objects, o_1, o_2, o_3, and o_4, respectively.

Given the VD solution-space, the index is constructed over the VCs. The NN search problem is thus reduced to the problem of searching the VC in which a query point is located. The original idea of indexing the solution space introduced in [1] was for similarity search in multimedia databases; i.e., for the NN search under high-dimensional spaces. Besides suggesting the idea of indexing solution space for the first time, the major contribution of [1] was to propose an approximation scheme for the VCs which could be very complex in the high-dimensional space and design a decomposition technique to improve the search performance under high-dimensional spaces. However, as we mentioned in Section 1, this paper will focus on a low dimensional space, under which the advantages of previous idea of indexing the solution space might not be appreciated. On the other hand, our recently proposed D-tree can be used to index VCs in low dimensional spaces and showed a better performance than other existing indexes [27].[1] As such, D-tree is used as the representative for solution-based indexes to compare against the proposed grid-partition index.



(a) Divisions in the example (b) D-tree structure

Fig. 5 D-tree Index for the Running Example

[1] Note that D-tree was proposed for supporting general point-location queries, not limited to VC searching.

D-tree is to index any given solution space, like VD for NN search and the map of delivery areas for ZIP code queries. The basic idea of D-tree is to index regions in the solution space based on the divisions that form the boundaries of the regions (VCs are the corresponding regions for NN search). To construct the index, D-tree recursively partitions a space into two sub-spaces until each space contains only one region. Consider our running example: first, polyline $pl(v_2, v_3, v_4, v_6)$ partitions the original space into left subspace and right subspace (denoted by P_5 and P_6, respectively); next, $pl(v_1, v_3)$ 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. Figure 5 depicts D-tree structure for the running example. We use an example to illustrate the point query algorithm with D-tree. Suppose the query point is q_1 as shown in Figure 5(a). The search starts at the root and goes to the right child node since q_1 is to the right of the partition line $pl(v_2, v_3, v_4, v_6)$. Next, it follows the right pointer to access P_4 since q_1 is again to the right of the partition line $pl(v_4, v_5)$. By then, we know the NN to q_1 is o_4 as P_4 is the VC of o_4.

Compared with object-based index, search algorithms based on solution-based index do not bring any backtracking and hence are suitable for the sequential access media such as wireless channel. However, instead of indexing the positions of objects, it has to index all the VCs of the objects which has definitely enlarged the index size. Since index information also occupies wireless channels and hence has been also regarded as an important performance metric, existing solution-based indexes might not be the best choice for NN search under mobile devices.

3 A Grid-Partition Index

As mentioned in the previous sections, both object-based indexes and solution-based indexes for NN search have certain advantages and disadvantages. An object-based index has a small size, since it only indexes the necessary information; i.e. the position information of objects. However, it requires lots of backtracking in the whole search process and hence the index search cost under sequential-access broadcast mode could be high. On the other hand, a solution-based index in general avoids the backtracking problem by mapping the NN search problem into a point location query. Consequently, it achieves a good search performance for sequential-access broadcast. However, it indexes the VCs (in the shape of polygons) rather than data objects that can be represented by points directly, thus resulting in a large index size.

In the following subsections, we introduce the structure of the grid-partition index, which is a hybrid of object-based and solution-based indexes. The goal is to combine the advantages of both indexes. The grid-partition index starts with the solution space, but instead of indexing the VCs, it stores the object locations in the index. The following several sections describe the grid-partition index in detail.

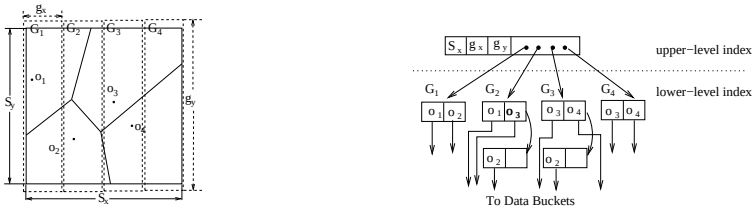
3.1 Basic Idea

In object-based indexes, each NN search starts with the whole search space and gradually limits the space based on some knowledge collected during the search. We have observed that an essential problem affecting the search performance 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 disjoint grid cells. For each grid cell, we index all the objects that could be NNs of at least one query point inside the grid cell.

Definition 1 *An object is **associated** with a grid cell if and only if it is the NN of some query point inside the grid cell.*

As explained in previous work we have conducted [29], the Voronoi Diagram (VD) mentioned in the last section can be used to conceptually illustrate the idea of associating objects to grid cells. Given a VD, an object is the NN only to the query points located inside its VC. For instance, in Figure 2(b), object o_1 is the NN only to the query points inside P_1. Therefore, for any query point inside a grid cell, only the objects whose VCs overlap with this grid cell form the candidate set of its NNs.

Given an NN query, first we locate the grid cell in which the query point lies, then search the answer based on objects associated with that grid cell only. Since each grid cell only covers a limited part of the search space, the number of objects associated with each grid cell is expected to be much smaller than the number of all available objects in the original space. Thus, if we can quickly locate the grid cell, the search space for an NN query will be greatly reduced. Therefore, the overall search performance can be improved. Figure 6(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, o_3, and so on. If a given query point is in grid cell G_1, the NN can be found among the objects associated with G_1 (i.e., o_1 and o_2), instead of among the whole set of objects.



(a) Grid Partition (b) Index Structure

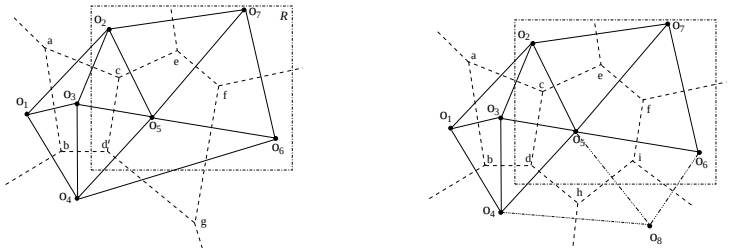
Fig. 6 Fixed Grid Partition for the Running Example

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 built upon the objects associated with each grid cell. The upper-level index maps a query point to the corresponding grid cell, while the lower-level index facilitates the access to the objects associated with each grid cell. The advantage is that once the query point is located in a grid cell, its NN is definitely among the objects associated with that grid cell, thereby preventing any backtracking operations and enabling a single linear access of the upper-level index for any query point. In addition, to reduce the search space as much as possible and to avoid backtracking operations in the lower-level index, we try to control the size of each grid cell such that its associated objects can fit into one page. Thus, for each grid cell, a simple index structure (i.e., a list of object-pointer pairs) is employed. In case the index for a grid cell cannot fit into one page, it is sequentially allocated to a number of pages. In each grid cell, the list of object-pointer pairs is sorted along the dimension (hereafter called the *sorting dimension*) in which the grid cell has the largest span. For example, in Figure 6(a), the associated objects for grid G_2, o_1, o_3, and o_2, are sorted according to the y-dimension. The purpose of this arrangement is to speed up the NN detection procedure, as we will see in Section 3.3.

3.2 Associating Objects to Grid Cells

In the last subsection, we explained how to associate objects to grid cells using the VD. However, if there are updates on object locations, we have to re-construct the VD from scratch and re-establish the association relationships between objects and grid cells. To address the update issue, we propose an incremental technique based on the *Delaunay Triangulation* (DT) for associating objects to grid cells.



(a) DT Example (b) DT after Insertion of o_8

Fig. 7 Object Association based on DT

The DT is the straight-line dual of the VD [2]. In the DT, the vertices are objects themselves, and every segment connecting two objects represents some relationship between those two objects, which satisfies the following property:

Property 1 *Two objects are directly connected in the DT if and only if their VCs share a common edge.*

Figure 7(a) shows seven objects (o_1 through o_7), their VCs (represented by dashed polygons), and the DT (represented by solid triangles). One important property of the DT is that the circum-center of any triangle is one of the endpoints in the VD. As in Figure 7(a), point a in the VD is the circum-center of triangle $\triangle o_1 o_2 o_3$. Based on this property, we develop an algorithm (see Algorithm 1) to select the associated objects for a grid cell. The algorithm assumes that the DT is pre-processed and maintained in some data structure.

Algorithm 1 Selecting Associated Objects for a Grid Cell

Input: Grid cell, locations of objects, and their DT;
Output: Associated objects of this grid cell;
Procedure:

```
1: for each object o_i do
2:    initialize count(o_i) = 0; mark(o_i) = 0;
3:    if o_i is inside the input grid cell then insert it into a queue Q
4: end for
5: if Q is empty then
6:    find nearest objects of the endpoints of the grid cell, and insert them into
      Q
7: end if
8: while Q is not empty do
9:    pop the first object o_i in the queue
10:   for each object o_j adjacent to o_i do
11:      obtain the triangles that have segment o_j o_i as one edge
12:      if there are two such triangles △1 and △2 then
13:         let c_1 be circum-center of △1 and c_2 be circum-center of △2
14:      else
15:         there is only one triangle △
16:         let c_1 be circum-center of △ and c_3 be the outermost point on the
            perpendicular bisector of segment o_j o_i
17:      end if
18:      if segment c_1 c_2 or ray c_1 c_3 overlaps with the grid cell then
19:         count(o_i)++
20:         if mark(o_j) != 1 then insert it into Q
21:      end if
22:   end for
23:   mark(o_i) = 1
24: end while
25: return all the objects with count > 0
```

The algorithm works as follows. The objects inside the grid cell are definitely associated with this grid cell (lines 1-4). If there are no objects inside the given grid cell, the NNs to the four endpoints of the grid cell

are selected as seed objects (lines 5-7). After obtaining these seed objects, we check their adjacent objects in a greedy manner (lines 8-24). If one edge of the adjacent object's VC overlaps with the grid cell, it is selected. For each newly added object, its adjacent objects are also checked. The counter associated with each object is used for update operations, as will be explained later in this subsection.

Let's examine an example. Suppose that the grid cell is R in Figure 7(a). The four objects, o_2, o_5, o_6, and o_7, are inside R and hence are selected as the associated objects with the grid cell. Next, we check the adjacent objects of o_2, namely, objects o_1 and o_3. For o_3, the two triangles that have segment $\overline{o_2o_3}$ as the common edge are $\triangle o_1o_2o_3$ and $\triangle o_2o_3o_5$, which have points a and c as their circum-centers, respectively. Segment $\overline{ac}$ overlaps with R. Thus, object o_3 is also an associated object of R. For o_1, the corresponding ray emanating from a does not overlap with R. Thus, o_1 is not selected. Similar operations are performed on unmarked adjacent objects of o_5, o_6, o_7, and o_3. Finally, objects o_2 through o_7 are identified as the associated objects of grid cell R.

We now show the correctness of Algorithm 1.

Theorem 1 *An object is selected by Algorithm 1 to associate with a grid cell if and only if it is an NN of the grid cell.*

Proof: If a VC overlaps with a grid cell, at least one of the VC's edges should be inside the cell. Now suppose that the algorithm does not select all of the associated objects of the cell. Without loss of generality, let's assume that the set of objects selected by the algorithm is $\mathcal{A}$, and those that are neighbors but not selected are denoted by $\mathcal{B}$. According to the definitions of object-grid association, the VCs of $\mathcal{A}$ and $\mathcal{B}$ together should cover the grid cell.

We next prove that the VCs of $\mathcal{A}$ and $\mathcal{B}$ are not connected in the VD as follows. Suppose they are connected, from Property 1, there must be two objects, o_1 from $\mathcal{A}$, and o_2 from $\mathcal{B}$, connected in the DT. In this case, according to the algorithm, when checking the adjacent neighbors of o_1, o_2 should have been selected, which is a contradiction.

Since the VCs of $\mathcal{A}$ cannot cover the whole cell (otherwise $\mathcal{A}$ is the complete solution), and the VCs of $\mathcal{A}$ and $\mathcal{B}$ are not connected in the VD, they cannot cover the whole cell. This contradicts to our previous claim that the VCs of $\mathcal{A}$ and $\mathcal{B}$ together cover the grid cell. Thus, the original assumption does not hold, and we prove that all the objects that are NNs of the grid cell are selected.

Next, we are going to prove that only the objects that are NNs of the grid cell will be selected. For any object selected by Algorithm 1, at least one edge of its VC overlaps with the grid cell. By definition of VC, this object must be an NN of some point lying in the grid cell. Therefore, any object selected by Algorithm 1 must be a nearest neighbor of the grid cell.

□

Object updates will result in insertion and deletion of some edges in the DT. An incremental algorithm with a time complexity of $O(\log(n))$ can be employed to update the DT when objects are deleted, inserted, or modified [2]. In the following, we describe the operations of changing the associated objects of a grid cell in accordance with edge insertion or deletion in the DT.

- **Insertion** When edge e is inserted in the DT, first we calculate the circum-center(s) of the triangle(s) having e as an edge. If there are two such triangles, a segment is constructed by connecting their circum-centers. Otherwise there is only one triangle; we construct a ray emanating from its circum-center. Then, we could obtain the grid cells that overlap the segment or the ray. The object(s) of the segment or the ray should be added to those grid cells if they are not associated with them, and their counters for those grid cells are increased by 1.
- **Deletion** When an edge is deleted, the circum-center(s) of its related triangle(s) can be obtained as in insertion. If the corresponding segment or ray overlaps with a grid cell, the counter(s) of the object(s) for that cell will be decreased by 1. If the counter reaches to 0, the object is removed from the corresponding grid cell.

Figure 7(b) provides an example for the insertion operation, where a new object o_8 is added. Using the incremental update algorithm for the DT, the affected edges can be obtained. Specifically, dash-dot edges $\overline{o_4o_8}$, $\overline{o_5o_8}$, and $\overline{o_6o_8}$ are added and edge $\overline{o_4o_6}$ is deleted. For edge $\overline{o_4o_6}$, the corresponding ray extending from circum-center g does not overlap with R. Hence, no action is needed. For newly added edge $\overline{o_5o_8}$, the corresponding segment $\overline{hi}$ overlaps with R. Hence, object o_8 is added. The same test can be carried out for the other two edges.

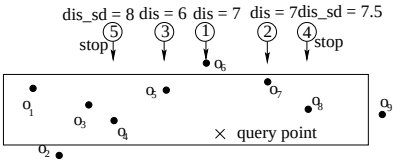
3.3 Nearest-Neighbor Search

With the grid-partition index, an NN query is answered by executing the following three steps: 1) *locating the grid cell*, 2) *detecting the NN*, and 3) *retrieving the 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 NN 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 NN object for a query point inside a grid cell. This procedure works for all the proposed grid partition approaches.

In a grid cell, given a query point, the sorted associated objects are broken down into two lists: one list consists of the objects with coordinates smaller than the query point in the sorting-dimension, and the other list consists of the remaining objects. To detect the NN, the objects in the two lists are checked alternately. Initially, the current shortest distance *min_dis* is set

to be infinite. At each checking step, let *cur_dis* be the distance between the object c being checked and the query point q; i.e., $\sqrt{(x_c - x_q)^2 + (y_c - y_q)^2}$; *min_dis* is updated if *cur_dis* < *min_dis*. Let *dis_sd* denote the distance between the current object and the query point along the sorting dimension; i.e., $|x_c - x_q|$ or $|y_c - y_q|$. The checking process continues until *dis_sd* > *min_dis*. The correctness can be justified as follows. For the current object, *cur_dis* $\geq$ *dis_sd* and hence we have *cur_dis* > *min_dis* if *dis_sd* > *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*.

Figure 8(a) gives an example in which nine objects that are associated with the grid cell (denoted by the solid line rectangle) are sorted according to the x-dimension over which the cell has a larger span. Given a query point as shown in the figure, the nine objects are broken into two lists, with one containing o_1 to o_6 and the other containing o_7 to o_9. The algorithm proceeds to check these two lists alternately; i.e., in the order of o_6, o_7, o_5, o_8, $\cdots$, and so on. Figure 8(b) shows the intermediate results for each step. In the first list, the checking stops at o_4 (Step 5), since its distance to the query point in the x-dimension (i.e., 8) is already longer than *min_dis* (i.e., 6). Similarly, the checking stops at o_8 (Step 4) in the second list. As a result, only five objects rather than all nine objects are evaluated. The improvement is expected to be significant when the width and height of the grid cell differ greatly.



(a)

Step	Object Checking	*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. 8 An Example for Detecting NN

3.4 Cost Models

In our previous work [29], we proposed three basic approaches to partitioning the search spaces into grid cells, namely, *fixed partition (FP)*, *semi-adaptive partition (SAP)*, and *adaptive partition (AP)*. The basic ideas are

briefly summarized as follows. FP divides the search space into fixed-size grid cells and hence is very simple for implementation. However, it does not take into account the distribution of objects and their VCs. Thus, it is not easy to utilize the index pages efficiently, especially when the objects' distribution is non-uniform. SAP adopts the fixed-size partition along one dimension, while keeps the partition along other dimension dynamic. It allows the grid cells to expand or shrink according to the objects' distribution. AP adaptively partitions the space using a kd-tree-like partition method. It recursively partitions the search space into two complementary subspaces such that the number of objects associated with each subspace is nearly the same. The partitioning process stops when the objects associated with each subspace could be fitted into one page. Please refer to [29] for the detailed partitioning schemes.

In this subsection, we derive the cost model of the grid-partition index under three partitioning approaches. In general, index search cost and index size are two basic measures of an index. Since the index size can be easily obtained when an index is built, we only derive the search cost in terms of the number of page accesses. We start by defining some notations in Table 1.

Notation	**Definition**
s_p	page capacity
N	number of objects
n_g	number of grid cells
n_s	number of strips (for SAP)
p_i	query probability in grid cell i
$(n_{i,1}, \cdots, n_{i,m_i})$	path in upper-level index from root to the leaf pointing to grid cell i (for AP)
s_i	size of lower-level index for grid cell i (in terms of # pages)
t_i	cost of locating cell i in upper-level index (in terms of # pages)
$t_{i,j}$	average search cost from the root to node $n_{i,j}$ (in terms of # pages)
T_u	search cost of upper-level index (in terms of # pages)
T_l	search cost of lower-level index (in terms of # pages)

Table 1 Definition of Notations

Generally, the search cost consists of the costs searching the upper-level index and the lower-level index:

$$T = T_u + T_l = \sum_{i=1}^{n_g} p_i(t_i + s_i). \tag{2}$$

The parameter s_i is a constant when an index is constructed, and p_i could be approximated by the ratio of the grid cell's area to the area of

the original search space. We only need to analyze t_i for different partition schemes. For the FP scheme, we obtain t_i in (3), assuming the page capacity is larger than the overhead for the header:

$$t_i = \begin{cases} 1 \text{ if grid cell } i \text{ is on the first index page;} \\ 2 \text{ otherwise.} \end{cases} \tag{3}$$

For the SAP scheme, we have t_i in (4), assuming the page capacity is larger than the overhead for the header in the first upper-level index page or the discriminators in the extra index nodes:

$$t_i = \begin{cases} 1 \text{ if the strip for grid cell } i \text{ is on the first index page and only} \\ \quad \text{contains 1 grid cell;} \\ 2 \text{ if the strip for grid cell } i \text{ is on the first index page and grid} \\ \quad \text{cell } i \text{ is on the first page of the extra node, or if it is not} \\ \quad \text{on the first index page but only contains 1 grid cell;} \\ 3 \text{ if the strip for grid cell } i \text{ is on the first index page but grid} \\ \quad \text{cell } i \text{ is not on the first page of the extra node, or if it is} \\ \quad \text{not on the first index page but the grid cell } i \text{ is on the first} \\ \quad \text{page of the extra node;} \\ 4 \text{ otherwise.} \end{cases} \tag{4}$$

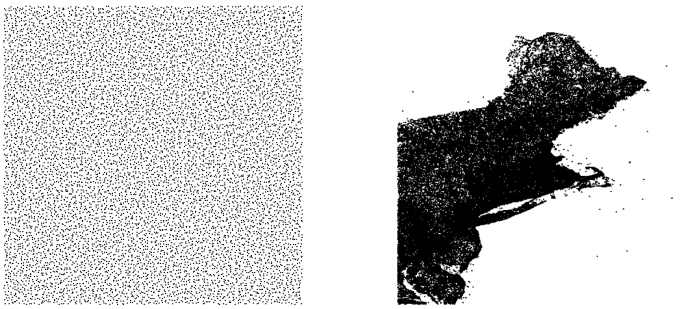
For the AP scheme, the index search cost for a node $n_{i,j}$ is denoted by $t_{i,j}$. Thus, we have $t_i = t_{i,m_i}$, which is obtained recursively:

$$t_{i,j} = \begin{cases} 1 & \text{if } j = 1; \\ t_{i,j-1} & \text{if } n_{i,j} \text{ and } n_{i,j-1} \text{ are on the same page;} \\ t_{i,j-1} + 1 & \text{otherwise.} \end{cases} \tag{5}$$

4 Performance Evaluation

This section presents the performance comparison of the proposed grid-partition index to the existing indexes for NN search. Both synthetic and real data are used in the evaluation. For synthetic data, a series of UNIFORM datasets is generated by randomly distributing the points (up to 1M points) in a square Euclidean space; several SKEWED datasets are also generated following the Zipf distribution, where the skewness parameter θ is varied from 0.0 to 0.9. The REAL dataset contains more than 100,000 postal addresses in three metropolitan areas (New York, Philadelphia, and Boston) [4].

We compare the grid-partition index to typical solution-based indexes and object-based indexes. For solution-based indexes, it was shown in [27] that D-tree outperforms other existing index structures. As such, D-tree is employed to index the solution space for NN search. It is referred to as *solution-based index* in later discussions. For object-based indexes, we evaluate the NN search algorithm based on R-tree, which is denoted as *object-based index*. Since all the objects are available before the index is



(a) UNIFORM Dataset (b) REAL Dataset

Fig. 9 Datasets for Performance Evaluation

built, the STR packing scheme [14] is employed to build R-tree to optimize the performance.

We evaluate the indexes in both air indexing and disk indexing application scenarios. In data broadcast, D-tree can be broadcast in either depth-first or breadth-first order. They have the same performance. However, as explained in Section 2, the R-tree-based NN search algorithm does not work well for air indexing as it would cause an extremely large access latency. We make the following revision to improve its access latency. No matter where the query is issued, the index pages are accessed sequentially, while branches that are guaranteed to fail are pruned according to the *mindist* and *minmaxdist* based heuristics as in the original algorithm (see [17] for details). In addition, R-tree is broadcast in depth-first order to reduce access latency. Although width-first order has been proved more efficient, it asks the client to maintain a queue to remember the distance information between the query point and all the nodes at the same level. Therefore, the client must have enough memory space to keep all these necessary information. Since this requirement cannot be guaranteed for small portable clients, we adopt the width-first order to traverse R-tree. In a traditional disk indexing environment, the best-first ordering search is employed to process NN queries, which selects the next node to search the NN from the set of MBRs in all nodes visited rather than from those in the current node only [9].

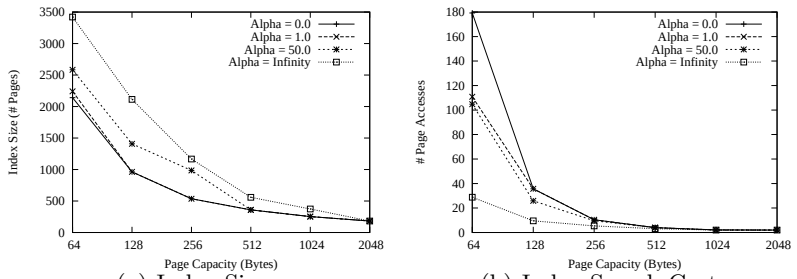
In the data broadcast scenario, for simplicity, a *flat* broadcast scheduler is employed (i.e., data objects are broadcast in a round-robin manner). To multiplex the index and data on the broadcast channel, we employ the $(1, m)$ interleaving technique (see Section 2.1 for details). The optimal value of m depends on the index size. It is calculated for each index structure separately based on the technique presented in [11].

The system parameters are set as follows. In each page, two bytes are allocated for the page id. Two bytes are used for one pointer and four bytes are used for one coordinate. The page capacity is varied from 128 bytes to

8K bytes.[2] In the following, first, we discuss the indexing efficiency metric; next, since the air indexing and disk indexing scenarios have different access characteristics in various aspects (e.g., access modes, performance goals, and page capacities), we report the experimental results for them respectively. In the air indexing scenarios, the page capacity is set to 256 bytes by default; while in the disk indexing scenarios, it is set to 4K bytes. The results to be reported were obtained for $1,000,000$ randomly generated queries on a PC with Pentium 4, 1.8G CPU and 1G main memory.

4.1 Indexing Efficiency Metric

The metric of indexing efficiency has been used in the FP and SAP grid partition schemes in order to determine the best grid partition. This subsection evaluates the effect of parameter α in the metric, which, set to a non-negative number, weighs the importance of the saved page accesses and the index overhead.



(a) Index Size (b) Index Search Cost

Fig. 10 Performance under Different Settings of α (UNIFORM(N=10,000), FP)

Figure 10 shows the performance for the UNIFORM (N=$10,000$) dataset when the FP partition scheme is employed. Similar results are obtained for the SAP scheme and/or other datasets. From the results, we can observe that the value of α has a significant impact on the performance, especially for small page capacities. In general, the larger the value of α, the better the index search cost and the worse the index size since a larger value of α assigns more weight to reduce search cost. As expected, the best index size is achieved when α is set to 0, and the best search cost is achieved when α is set to infinity. The setting of α can be adjusted based on requirements of the applications and systems. In the disk indexing scenario, the index size is not a big concern and the search cost is more important. Therefore, α is

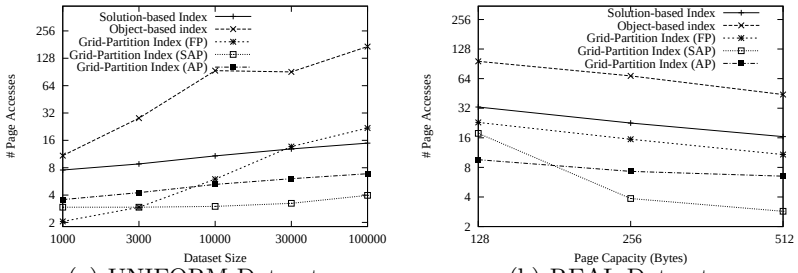
[2] For the disk storage, the page capacity is normally assumed in the order of 1K bytes [17,28]; for the wireless channel, the page capacity is normally assumed in the order of 100 bytes [10,11].

set to infinity to optimize the search cost. In the air indexing scenario, the index size is also important as it affects the access latency. Thus, the value of α is set to 1 in order to strike a balance between index size and index search cost.

4.2 Performance in the Air Indexing Scenarios

This subsection evaluates the indexes in an air indexing environment, where both index size (transferred to access latency) and index search cost (transferred to tuning time) are concerned with the performance of an index structure. In what follows, first, we report the results in terms of tuning time and access latency; then, we evaluate the indexes' adaptiveness to skewed object distributions.

4.2.1 Tuning Time As explained in Section 2.1, in wireless broadcast environments, improving the index search cost saves tuning time and hence power consumption and connection costs. Figure 11 shows the tuning times of different indexes. For UNIFORM datasets, the performance is evaluated under different dataset sizes with the default page capacity of 256 bytes; for REAL dataset, it is evaluated with different page capacities varying from 128 bytes to 512 bytes.



(a) UNIFORM Dataset (b) REAL Dataset

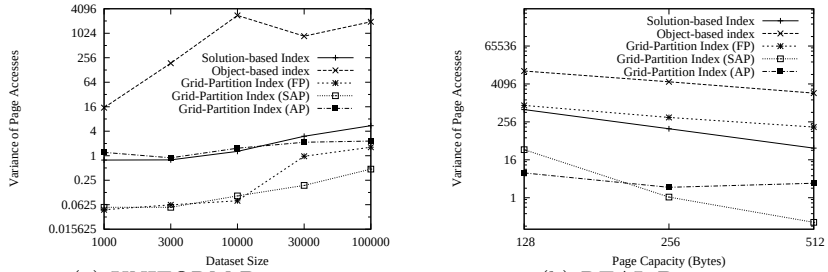
Fig. 11 Tuning Time in Air Indexing Environments

It can be observed that the proposed grid-partition index (with different grid partition approaches) substantially outperforms the existing indexes in most cases. In particular, the improvement of the proposed index over the object-based index is over a factor of 10 on average. Among the three grid partition approaches, the SAP gives the best performance in most cases. The main reason for this is that the SAP approach adapts better to the object distributions and their solution space compared to the FP, while it has a simpler data structure for the higher-level index (i.e., the index used for locating grid cells) compared to the AP. As a result, in most cases, the

SAP accesses one or two pages to locate grid cells and another one to detect the NN.

It is interesting to note that the FP deteriorates significantly with increasing database size. When the dataset size becomes larger than 100,000, it even works worse than the solution-based index (see Figure 11(a)).[3] This can be explained as follows. Recall that we set the parameter of α in indexing efficiency to 1 in air indexing, which means that the index size and index search cost have equal weights in deciding the final grid cell size. When the database becomes larger, the index overhead is increased. Hence, to achieve the best indexing efficiency, the index search cost is sacrificed a little bit to render a smaller index size.

We also evaluate the variance of tuning time in Figure 12. The object-based index has the worst variance since it relies on a branch-and-bound search algorithm. The solution-based index can provide a competitive performance since the underlying D-tree was designed to be a balanced tree. The grid-partition index has the best overall performance (less than 4 for most cases). Among the three partition approaches, as expected, the FP achieves a quite good performance for the UNIFORM datasets but a poor performance for the skewed REAL dataset; both the SAP and AP have small variance for both types of datasets.



(a) UNIFORM Dataset (b) REAL Dataset

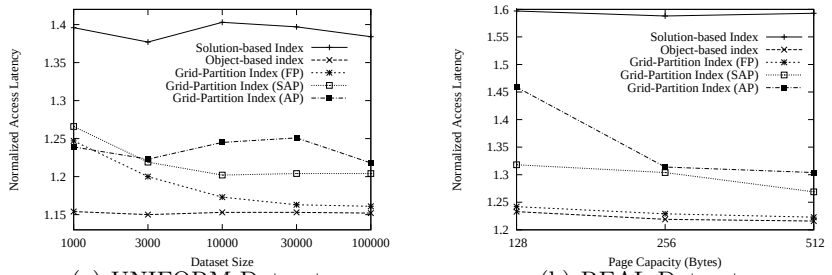
Fig. 12 Variance of Tuning Time in Air Indexing Environments

4.2.2 Access Latency This subsection evaluates the indexes in terms of access latency. As explained in Section 2.1, the access latency is determined by the index size. The larger the index size, the worse the access latency. The result is shown in Figure 13, where the latency is normalized to the average latency without any index (i.e., half the time needed to broadcast all objects in the dataset).

We can see that the solution-based index gives the worst performance because of its largest index size, which is caused by the fact that it indexes the VCs (polygons) rather than the objects. It is also clear that the

[3] However, as we shall show in Section 5.2.2, the large size of solution-based index diminishes its competitiveness.

object-based index provides the best performance since, there's no object duplication and we use the packing algorithm to ensure full usage of each page when constructing R-tree index. However, as we saw in the last subsection, its search performance is not good enough to be a practical index for NN search in wireless broadcasting environments. The performance of the grid-partition index is not bad. Compared to the object-based index, they introduce only a small latency overhead (within 10% in most cases) due to object duplication in different grid cells. Given its superior performance in tuning time, there is no doubt that it is the best index overall. Among the three partition approaches, the AP works worse than the other two due to the large overhead in maintaining the paged kd-tree structure.

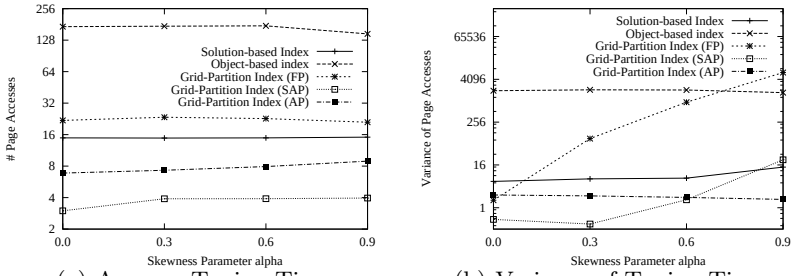


(a) UNIFORM Dataset (b) REAL Dataset

Fig. 13 Access Latency in Air Indexing Environments

4.2.3 Effect of Object Distributions We now evaluate the robustness of the proposed index with respect to various object distributions, which are simulated by the skewness parameter θ as mentioned in the very beginning of this section. The higher the value of θ, the more skewed the object distribution. As shown in Figure 14(a), the average tuning time of all indexes is almost not affected by the value of θ. However, Figure 14(b) shows that the variance is really affected. As explained in Section 4.1, the page occupancy with the FP approach is not uniform especially for highly skewed datasets, since it does not consider the object distribution when partitioning the grid. Hence, with increasing skewness, the variance of the FA increases dramatically. The other grid partition approaches and indexes are not affected a lot. This implies that they are able to adapt to different object distributions.

To summarize, compared to the solution-based index, the proposed grid-partition index achieves a much shorter access latency and a better tuning time in most cases; compared to the object-based index, the proposed index improves the tuning time, on average, by over a factor of 10, while maintaining a slightly worse access latency. Therefore, the proposed grid-partition index strikes a better balance between tuning time and access latency.



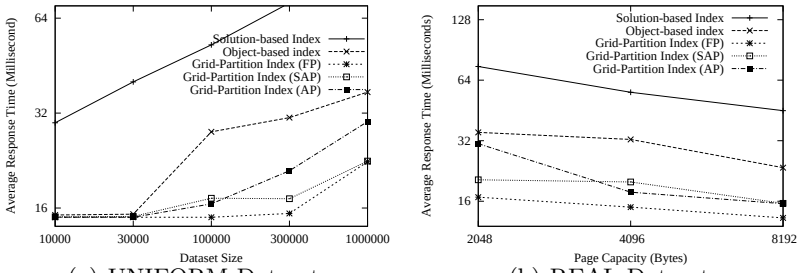
(a) Average Tuning Time (b) Variance of Tuning Time

Fig. 14 Performance for Datasets of Different Skewness Levels in Air Indexing Environments (PageSize=256)

4.3 Performance in the Disk Indexing Scenarios

In on-demand access, disk indexing can be employed to improve the efficiency of query processing on the server. This subsection evaluates the access latencies of the indexes. As the disk space is not an important concern, the primary performance goal in this scenario is to optimize the query latency. Thus, in deciding the grid partition for the FP and SAP approaches, the value of α in indexing efficiency is set to infinity to allocate the entire weight to the saved index search cost.

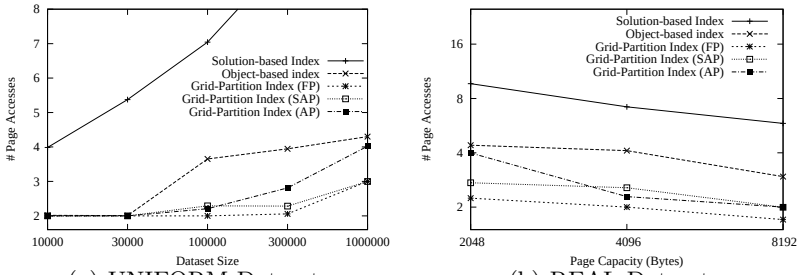
4.3.1 Access Latency Figure 15 shows the average query latency, which includes the disk I/O cost as well as the CPU cost. The improvement of the grid-partition index over the other two indexes is significant. The solution-based index has the worst performance. This is partly because its relatively large index size increases the depth of the tree and, hence, worsens the index search performance. Since random access is possible in the on-demand access environment, the object-based index now works better than the solution-based index, but still falls behind the proposed grid-partition index.



(a) UNIFORM Dataset (b) REAL Dataset

Fig. 15 Access Latency in Disk Indexing Environments

To obtain more insights into disk I/O cost and CPU cost, we show in Figure 16 the index search cost in terms of the number of page accesses. As can be seen, the relative performance of the various indexes is the same as that shown in Figure 15. This implies that the I/O cost (i.e., index search cost) dominates the CPU cost in the overall access latency for all indexes. As observed in the experiments, the disk I/O delay accounts for more than 90% of the overall latency, which is consistent with the previous studies.



(a) UNIFORM Dataset (b) REAL Dataset

Fig. 16 Index Search Cost in Disk Indexing Environments

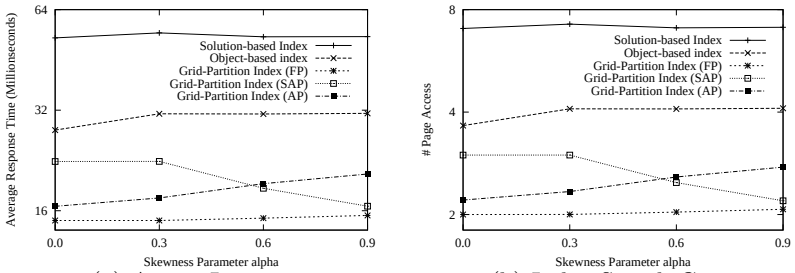
Among the three grid partition approaches, the FP provides the best performance in this scenario. This is mainly because we set α to infinity to give full weight to the index search cost. Consequently, the FP constructs the index in a way such that the objects associated with each grid cell can fit into one page, though the average page occupancy might be low. As such, any NN query can be answered by accessing two or three index pages (one or two for the upper-level index and another one for the lower-level index).

4.3.2 Effect of Object Distributions As in the air indexing scenario, we evaluate the robustness of indexes in terms of adaptiveness to various object distributions. As shown in Figure 17, the grid-partition index outperforms the existing indexes consistently. In particular, the FP-based grid-partition index achieves the best performance in all cases. Its improvement is 70% over the solution-based index and about 50% over the object-based index.

To summarize, the grid-partition index offers the best access latency with various datasets in a disk indexing scenario. The performance improvement over the solution-based index and the object-based index reaches a factor of 4. Since the index search cost is the only concern in this experiment, the FP approach performs better than the SAP and AP approaches.

5 Continuous-Nearest-Neighbor Search

We have shown the superiority of the grid-partition index for NN search; this section proceeds to discuss answering *continuous-nearest-neighbor search*



(a) Access Latency (b) Index Search Cost

Fig. 17 Performance for Datasets of Different Skewness Levels in Disk Indexing Environments (PageSize=4K)

(CNN) on the grid-partition index. A CNN search retrieves the nearest neighbor for every point on a given line segment which a mobile client may traverse (Definition 2) [19,22]. An example of CNN search is "finding the nearest gas stations along the route from my current location to Boston on Highway I-93.[4]" Since it is natural for mobile clients to issue queries while they are moving, such CNN queries have many important applications in mobile computing. Sistla et al. was the first to identify the CNN problem. Since then, it has received considerable attention thanks to the continuously rapid development of location-based services and mobile computing technologies [20,22,23].

Definition 2 *For a given query line segment from point s to point e, denoted by $\overline{se}$, CNN search returns an answer set $cnn(s,e)$ that contains data objects from the dataset, D, and satisfies the following conditions:*
$\forall p \in \overline{se}, \forall o' \in (D - cnn(s,e)), \exists o \in cnn(s,e)$ *such that* $dis(p,o') \geq dis(p,o)$

Here, every object o in the answer set *dominates* a part of the line segment; i.e., o is the NN of any query point lying on that partial line segment. An illustrative example is depicted in Figure 18, where $cnn(s,e)$ contains three objects, namely, o_1, o_2, and o_4. o_1 dominates the shadowed line segment $\overline{sp_1}$; i.e., o_1 is the NN of any point lying on $\overline{sp_1}$. Similarly, o_2 dominates $\overline{p_1p_2}$, and o_4 dominates $\overline{p_2e}$. p_1 and p_2 are called *split points* [23], since they are the points at which the NN objects along the line segment change.

The first proposal to address the CNN problem appeared in [20]. It employed a sampling technique to obtain an approximate answer. Given a query line segment, an NN query can first be processed for the pre-defined sampling points. Although the nearest objects found are local and are not necessarily the real nearest-neighbor objects, a search range to bound the real nearest-neighbor objects can be determined according to these local

[4] While a route may not be a line segment, it can be decomposed into multiple line segments.

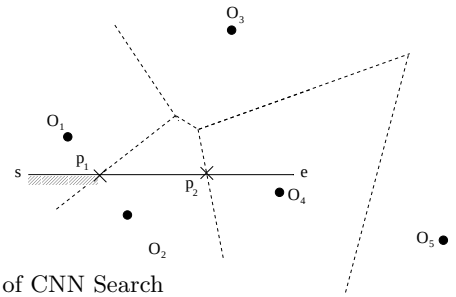


Fig. 18 Example of CNN Search

nearest-neighbor objects. Consequently, its accuracy heavily depends on the pre-defined sampling points on the query line.

Tao et al. carried out an in-depth study on CNN search and devised two algorithms for CNN queries based on R-tree [22, 23]. The first is based on the concept of time-parameterized (TP) queries [22]. Treating a query line segment as the moving trajectory of a query point, the nearest object to the moving query point is valid only for a limited duration. Consequently, CNN queries can be considered as TP queries. In this case, a new TP query was issued to retrieve the next nearest object once the valid time of the current query expired; i.e., when a split point was reached. While the TP approach avoids the drawbacks of sampling, it is an incremental algorithm that needs to issue n NN queries in order to obtain the final answer set, where n is the set cardinality.

The second algorithm is based on some heuristics to obtain the whole answer set within a single navigation of R-tree [23]. The main heuristic employs a metric of $mindist(E, q)$; i.e., the minimum distance between an MBR E and a query point or query segment q. The algorithm also maintains a list SL to record the split points found so far, together with their NNs. The search starts from the root and traverses the R-tree according to the following principles: i) when a leaf node is visited, SL is updated accordingly if some objects have shorter distances to the split points than that of the recorded nearest neighbor; ii) for an intermediate node, it is visited only if the MBR may contain a qualified data point. The qualification conditions are two-fold. First, $mindest(E, q)$ should be smaller than SL_{max}; i.e., the maximum distance between a split point and its NN. Second, there must exist at least one split point s_i, $s_i \in SL$, such that $distance(s_i, s_i.NN) > mindist(E, q)$.

5.1 Supporting CNN on the Grid-Partition Index

Recall that the grid-partition index ensures that the nearest neighbor to a given query point q is associated with the grid cell containing q. Consequently, the answers to a CNN query for a given line segment l must be associated with all of the grid cells that overlap with l. There are two steps involved in the search process of a CNN query based on the grid-partition

index. The first is to identify all the grid cells overlapping with the given query segment and to retrieve all the objects associated with these grid cells. Note that the segment can be divided into several parts, with each part lying within only one grid cell. Thus, the second step is to find the exact answer set for each part within a single cell, which can be achieved by Algorithm 2.

Algorithm 2 *Finding_NN_Objects_of_a_Segment*

Input: a query segment $\overline{se}$, the located grid cell i, the answer set $result$;
Output: nearest neighbors of all the points along $\bar{se}$;
Procedure:
1: NN_s = NN(s); NN_e = NN(e);
2: $result \cup= \{NN_s, NN_e\}$;
3: **if** $NN_e = NN_s$ **then**
4: return $result$;
5: **end if**
6: let m be the object returned by SplitPoint($\overline{se}$, NN_s, NN_e);
7: NN_m = NN(m);
8: **if** ($NN_m = NN_s$) or ($NN_m = NN_e$) **then**
9: return $result$;
10: **else**
11: $Finding\_NN\_Objects\_of\_a\_Segment(\overline{sm}, i, result)$;
12: $Finding\_NN\_Objects\_of\_a\_Segment(\overline{me}, i, result)$;
13: **end if**

Before we explain Algorithm 2, we describe some functions to be used. The function $NN(q)$ is to return the NN object for the query point q, which is introduced in Section 3.3. The function $SplitPoint(\overline{se}, NN_s, NN_e)$ is to find the split point along the query line segment $\overline{se}$ based on the two detected NN objects, NN_s and NN_e. As shown in Figure 19, the split point is the intersection of the segment $\overline{se}$ and the bisector of the segment $\overline{NN_sNN_e}$.

The basic idea of Algorithm 2 is to employ a recursive search algorithm similar to binary search. It first finds the NNs of the endpoints of the query segment, namely NN_s and NN_e. If those two NNs are the same, the whole line segment definitely falls within the VC of NN_s, which can be easily proven by the convexity of VCs. Therefore, the CNN search process stops. Otherwise, the split point m is determined based on NN_s, and NN_e; and m's NN, NN_m, is found. If NN_m is equal to NN_s (or NN_e), both NN_s and NN_e must be the NN of m. This can be guaranteed by the function SplitPoint. As a result, segment $\overline{sm}$ is dominated by NN_s and the segment $\overline{me}$ is within the VC of NN_e. Thus, the search is terminated. Otherwise, the search algorithm is invoked again based on the segments $\overline{sm}$ and $\overline{me}$. Suppose the final answer set contains n objects, the client needs to conduct $O(n)$ NN searches to complete the query.

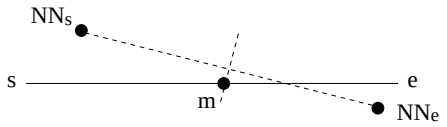
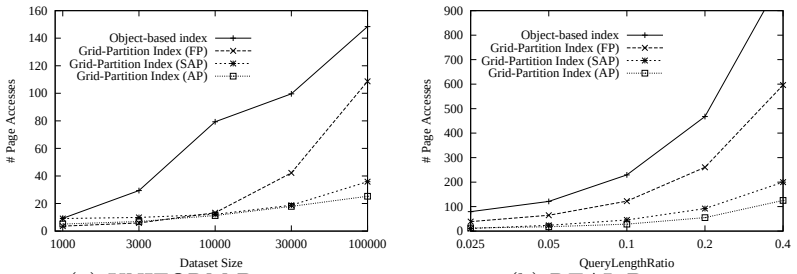


Fig. 19 Example of MidPoint for a Given Segment $\overline{se}$

5.2 Simulation Results

This subsection compares the proposed algorithm based on the grid-partition index with the state-of-the-art algorithm based on R-tree [23] for CNN search. The parameter settings are similar to those used in Section 4. We use a new parameter *QueryLengthRatio* to define the ratio of the length of query line segment to the width of the search space. Due to the reasons explained in Section 4, the depth-first order is employed to traverse R-tree in air indexing environment; whereas the best-first order is used in the disk indexing environment.

5.2.1 CNN Search Performance in the Air Indexing Scenarios Figure 20(a) shows the tuning time under different UNIFORM datasets, with the page capacity set at 256B and *QueryLengthRatio* at 0.1. As in the case of NN search, by enabling CNN search via a single linear scan, the grid-partition index improves the performance over R-tree remarkably. On average, the grid-partition indexes with FP, SAP, and AP outperform R-tree by 61%, 62%, and 74%, respectively. Among them, SAP and SP provide a more stable performance than FP throughout the datasets tested, which is consistent with the results observed in Section 4.



(a) UNIFORM Dataset (b) REAL Dataset

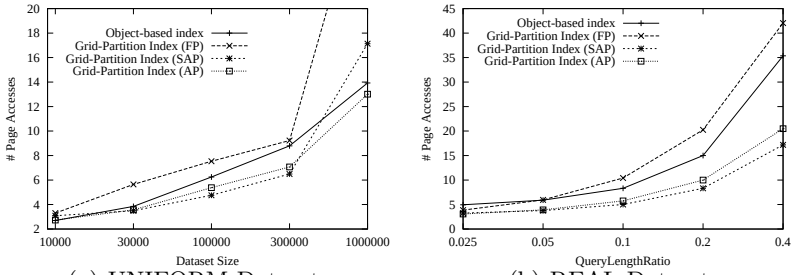
Fig. 20 Tuning Time for CNN Search in Air Indexing Scenarios (PageSize=256B)

Figure 20(b) plots the tuning time of different indexes under the REAL dataset with the parameter *QueryLengthRatio* varying from 0.025 to 0.4. For all indexes, we can observe that the longer the query line segment, the higher the tuning time. The grid-partition index outperforms R-tree in all cases tested. In particular, the improvement increases with lengthening the

query segment. This is because with a longer query segment, more NNs are return and, hence, more accumulative improvement can be observed.

5.2.2 CNN Search Performance in the Disk Indexing Scenarios We show in Figure 21(a) the disk search performance under various UNIFORM datasets with the page capacity set at 4KB and *QueryLengthRatio* at 0.1. Figure 21(b) shows the results of different indexes under the REAL dataset with various *QueryLengthRatio* settings.

Recall that the best-first order is employed in the R-tree-based search algorithm in the disk indexing scenarios. This significantly improves the performance of R-tree for CNN search. As a result, the R-tree now performs better than the grid-partition index with FP; but still worse than the grid-partition indexes with SAP and AP. Note that this is opposite to our previous observation that the grid-partition index with FP has best performance for NN search (Section 4.3.1). This could be explained as follows. As mentioned before, the FP approach partitions the space as much as possible such that the associated objects of each cell can be accommodated in a single page. Thus, most of the NN queries need to access only a single page in the lower-level index, which, together with the simple hashing function in the upper-level index, optimizes the performance for NN search. However, for CNN search, as the FP approach has smaller grid cells, much more cells need to be checked to find all NNs of a given line segment. Consequently, the overall performance of FP is not good for CNN search.

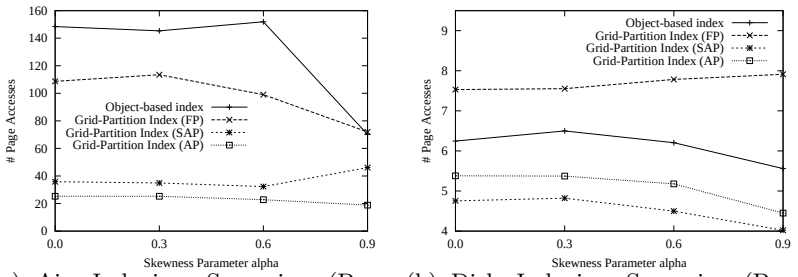


(a) UNIFORM Dataset (b) REAL Dataset

Fig. 21 Search Time for CNN Search in Disk Indexing Scenarios (PageSize=4096B)

5.2.3 CNN Search Performance under Various Object Distributions This set of experiments examines the impact of different objects distributions. We vary the skewness parameter θ from 0.0 to 0.9. The search performances for the air indexing and disk indexing scenarios are plotted in Figure 22(a) and 22(b), respectively. We can see that the relative performances of different indexes are affected little by the setting of skewness level. For the air indexing scenarios, R-tree performs the worst in almost all cases; the

grid-partition indexes with FP, SAP, and AP improve the performance by 15.1%, 64.7%, and 79.8%, respectively. For the disk indexing scenarios, the grid-partition indexes with SAP and AP consistently outperform R-tree.



(a) Air Indexing Scenarios (Page-Size=256B)

(b) Disk Indexing Scenarios (Page-Size=4096B)

Fig. 22 Performance for Datasets of Different Skewness Levels

6 Conclusion

NN search is a very important and practical application among the promising LDIS applications. In this paper, we have proposed an index, called the grid-partition index, to address the NN search problem in mobile computing environments. The grid-partition index combines the advantages of existing index structures, such as the small size of the object-based index and the fast search of the solution-based index, to provide a flexible structure that is suitable for both disk indexing and air indexing environments. An algorithm was devised to associate all the potential objects to a grid cell such that the maintenance cost is minimized for object updates.

The performances of the grid-partition index and existing object-based and solution-based indexes were evaluated using both synthetic and real datasets. The results showed that the grid-partition index substantially outperforms both the object-based and solution-based indexes. We summarize the results as follows. For air indexing scenarios, the grid-partition index outperforms the solution-based index in terms of both tuning time and access latency; it also strikes a better balance between tuning time and access latency than does the object-based index. For disk indexing, the grid-partition index offers a much better access latency than the object-based and solution-based indexes. Furthermore, we have extended the grid-partition index to support CNN search.

In this paper, the grid-partition index was proposed to efficiently answer NN search. Moreover, the idea of the grid-partition index can be applied to k-nearest-neighbor (KNN) search, where the challenge is how to efficiently associate all relevant objects to a grid cell. We shall leave this issue to a

future study. In addition, we are examining generalized NN search such as finding the nearest hotel with a room rate of less than $200.

7 Acknowledgments

The authors would like to thank Profs. Sunil Arya and Mordecai J. Golin, the editor Ralf Hartmut Güting, and anonymous reviewers for their valuable comments and suggestions that helped to improve the quality of this paper. The research was supported in part by Wharton-SMU Research Center, Singapore Management University (Grant No. C220/MSS3C101 and C220/T050011), grants from the Research Grant Council, Hong Kong SAR, China (Grant No. HKUST6179/03E and HKUST6225/02E). Jianliang Xu's work was supported by a grant from Hong Kong Baptist University (Grant FRG/02-03/II-34). Wang-chien Lee's work was supported in part by the National Science Foundation under Grant IIS-0328881.

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, USA, 1996.
3. K. L. Cheung and W.-C. Fu. Enhanced nearest neighbour search on the r-tree. *SIGMOD Record*, 27(3):16–21, 1998.
4. Spatial Datasets. Website at http://www.rtreeportal.org/spatial.html.
5. H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. E. Abbadi. Approximate nearest neighbor searching in multimedia databases. In *Proceedings of the 17th IEEE International Conference on Data Engineering (ICDE'01)*, April 2001.
6. J. Goldstein and R. Ramakrishnan. Contrast plots and p-sphere trees: Space vs. time in nearest neighbor searches. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)*, pages 429–440, Cairo, Egypt, September 2000.
7. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD'84)*, pages 47–54, 1984.
8. A. Hinneburg, C. C. Aggarwal, and D. A. Keim. What is the nearest neighbor in high dimensional spaces? In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)*, September 2000.
9. Gí. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.
10. 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'00)*, pages 157–166, San Diego, CA, USA, February 2000.

11. 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.
12. I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 500–509, Santiago de Chile, Chile, September 1994.
13. 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.
14. 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.
15. J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems (TODS)*, 9(1):38–71, 1984.
16. S. Pramanik and J. Li. Fast approximate search algorithm for nearest neighbor queries in high dimensions. In *Proceedings of the 15th International Conference on Data Engineering (ICDE'99)*, page 251, March 1999.
17. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*, pages 71–79, May 1995.
18. Computer Science and Telecommunications Board. *IT Roadmap to a Geospatial Future*. The National Academies Press, 2003.
19. A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 422–432, Birmingham, UK, April 1997.
20. Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *Proceedings of the 7th International Symposium on Spatial and Temporal Databases (SSTD'01)*, pages 79–96, Los Angeles, CA, USA, July 2001.
21. R. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.
22. Y. Tao and D. Papadias. Time parameterized queries in spatio-temporal databases. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02)*, pages 334–345, Madison, Wisconsin, USA, 2002.
23. Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Hong Kong, 2002.
24. R. Weber and S. Blott. An approximation based structure for similarity search. Technical Report 24, ESPRIT Project HERMES (No. 9141), October 1997.
25. R. Weber, H. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB'98)*, pages 194–205, NewYork, August 1998.
26. G. T. Wilfong. Nearest neighbor problems. In *Symposium on Computational Geometry*, pages 224–233, June 1991.
27. 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.
28. C. Yu, B. C. Ooi, K.-L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to KNN processing. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 421–430, Roma, Italy, September 2001.
29. B. Zheng, J. Xu, W. C. Lee, and D. L. Lee. Energy-conserving air indexes for nearest neighbor search. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT'04)*, Heraklion - Crete, Greece, March 2004.