Continuous Monitoring of Spatial Queries in Wireless Broadcast Environments

Kyriakos Mouratidis, Spiridon Bakiras, *Member, IEEE,* and Dimitris Papadias

***Abstract*— Wireless data broadcast is a promising technique for information dissemination that leverages the computational capabilities of the mobile devices, in order to enhance the scalability of the system. Under this environment, the data are continuously broadcast by the server, interleaved with some indexing information for query processing. Clients may then tune in the broadcast channel and process their queries locally without contacting the server. Previous work on spatial query processing for wireless broadcast systems has only considered snapshot queries over static data. In this paper we propose an air indexing framework that (i) outperforms the existing (i.e., snapshot) techniques in terms of energy consumption, while achieving low access latency, and (ii) constitutes the first method supporting efficient processing of continuous spatial queries over moving objects.**

***Index Terms*— Spatial databases, query processing, location-based services, wireless data broadcast, air indexes.**

I. Introduction

Mobile devices with computational, storage, and wireless communication capabilities (such as PDAs) are becoming increasingly popular. At the same time, the technology behind positioning systems is constantly evolving, enabling the integration of low-cost GPS devices in any portable unit. Consequently, new mobile computing applications are expected to emerge, allowing users to issue location-dependent queries in a ubiquitous manner. Consider, for instance, a user (mobile client) in an unfamiliar city, who would like to know the 10 closest restaurants. This is an instance of a k *nearest neighbor* (kNN) query, where the query point is the current location of the client and the set of data objects contains the city restaurants. Alternatively, the user may ask for all restaurants located within a certain distance, i.e., within 200 meters. This is an instance of a *range* query.

Spatial queries have been studied extensively in the past, and numerous algorithms exist (e.g., [10], [11], [21]) for processing snapshot queries on static data indexed by a spatial access method. Subsequent methods [22], [24], [30] focused on moving queries (clients) and/or objects. The main idea is to return some additional information (e.g., more NNs [22], expiry time [24], validity region [30]) that determines the lifespan of the result. Thus, a moving client needs to issue another query only after the current result expires. These methods focus on single query processing, make certain assumptions about object movement (e.g., static in [22], [30], linear in [24]) and do not include mechanisms for maintenance of the query results (i.e., when the result expires, a new query must be issued).

Recent research considers *continuous monitoring* of multiple queries over arbitrarily moving objects. In this setting, there is a central server that monitors the locations of both objects and queries. The task of the server is to report and continuously update the query results as the clients and the objects move. As an example, consider that the data objects are vacant cabs and the clients are pedestrians that wish to know their k closest free taxis until they hire one. As the reverse case, the queries may correspond to vacant cabs, and each free taxi driver wishes to be continuously informed about his/her k closest pedestrians. Several monitoring methods have been proposed, covering both range (e.g., [4], [7], [18]) and kNN (e.g., [19], [26], [29]) queries. Some of these methods [19], [18], [26], [29] assume that objects issue updates whenever they move, while others [4], [7] consider that data objects have some computational capabilities, so that they inform the server only when their movement influences some query.

In the aforementioned methods, the processing load at the server side increases with the number of queries. In applications involving numerous clients, the server may be overwhelmed by their queries or take prohibitively long time to answer them. To avoid this problem, [14] propose *wireless data broadcast*, a promising technique that leverages the computational capabilities of the clients' mobile devices, and pushes the query processing task entirely to the client side. In this environment, the server only monitors the locations of the data objects, but is unaware of the clients and their queries. The data objects are continuously broadcast by the server, interleaved with some indexing information. The clients utilize the broadcast index, called *air index*, to tune in the channel only during the transmission of the relevant data and process their queries locally. Thus, the server load is independent of the number of clients.

Previous work on location-dependent spatial query processing for wireless broadcast systems has only considered snapshot queries over static data. On the other hand, existing spatial monitoring techniques do not apply to the broadcast environment, because they assume that the server is aware of the client locations and processes their queries centrally. In this paper we propose the *Broadcast Grid Index* (BGI) method, which is suitable for both snapshot and continuous queries. Furthermore, BGI extends to the case that the data are also dynamic. Figure 1 shows an example of continuous monitoring using BGI. The data objects are taxis that issue location updates to a central server using unicast uplink messages. The server processes the location updates, and continuously broadcasts the object information along with an up-to-date index using a wireless (e.g., 3G) network. Finally, the interested clients (e.g., mobile devices) listen to the broadcast channel and process their queries locally. Note that since the server tasks are independent of the number and the positions of the clients, this architecture may theoretically support infinite concurrent clients/queries. On the other hand, high object (e.g., taxi) cardinality increases both the server load (for processing the updates) and the length of the broadcast cycle.

BGI, and broadcast techniques in general, are preferable for

K. Mouratidis is with the Singapore Management University. S. Bakiras is with John Jay College, CUNY. D. Papadias is with the Hong Kong University of Science and Technology.

applications where the number of clients is large with respect to the number of data objects. As an example [33], Microsoft's MSN Direct (www.msndirect.com) uses broadcasting as an information dissemination method. Subscribers can receive live information regarding traffic conditions, stock quotes, gas prices, movie times, weather reports, etc. Even though location-based queries are not supported, we believe that this will be the next step, i.e., allowing the user to filter out unnecessary information using selective tuning (thus reducing the power consumption).

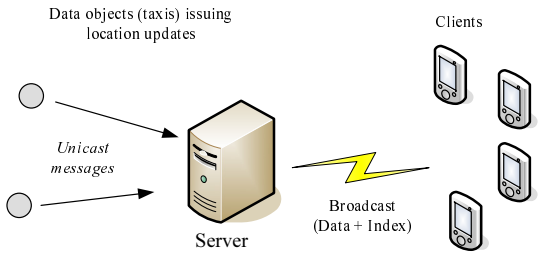


Fig. 1. Example of BGI

BGI indexes the data with a regular grid. The grid structure is beneficial in broadcasting environments because the spatial extents of its cells are implicit, leading to a small index size (and, thus, less broadcast information). Moreover, a grid supports fast object updates (as opposed to a more complicated index), avoiding server overloading in the presence of numerous updates [15]. In the case of static data, the index information is broadcast in two parts. The first one contains the cell cardinalities, and the second one contains the coordinates of the objects falling in each cell. This allows for efficient query processing at the client side. For applications where the objects are moving, the server broadcasts a *dirty grid* in the beginning of each cycle. The dirty grid indicates the regions of the data space that have received updates since the last cycle. The clients re-evaluate their queries only if the affected regions can potentially invalidate their current result.

The remainder of the paper is organized as follows. Section II briefly describes wireless broadcast systems, and overviews previous work on air indexes. Sections III and IV present the BGI method for snapshot and continuous kNN queries over static data, respectively. Section V extends the proposed framework to dynamic data objects. Range queries are collectively discussed in Section VI, due to their relative simplicity. Section VII experimentally evaluates our algorithms and, finally, Section VIII concludes the paper.

II. Related Work

Section II-A introduces the main concept of wireless broadcasting and discusses techniques for query processing on one-dimensional data. Section II-B overviews previous work on spatial query processing using air indexes.

A. Wireless Broadcasting and Air Indexes

The transmission schedule in a wireless broadcast system consists of a series of *broadcast cycles*. Within each cycle the data are organized into a number of index and data *buckets*. A bucket (which has a constant size) corresponds to the smallest logical unit of information, similar to the page concept in conventional storage systems. A single bucket may be carried into multiple network *packets* (i.e., the basic unit of information that is transmitted over the air). However, they are typically assumed to be of the same size (i.e., one bucket equals one packet).

The most common data organization method is the $(1, m)$ *interleaving scheme* [14], as shown in Figure 2. The data objects are divided into m distinct segments, and each data segment in the transmission schedule is preceded by a complete version of the index. In this way, the access latency for a client is minimized, since it may access the index (and start the query processing) immediately after the completion of the current data segment. [14] also introduces an alternative distributed index that reduces the degree of replication in order to further improve the performance. Specifically, instead of the entire index being replicated prior to each data segment, only the index that corresponds to the subsequent segment is included (i.e., replication occurs at the upper levels of the index tree).

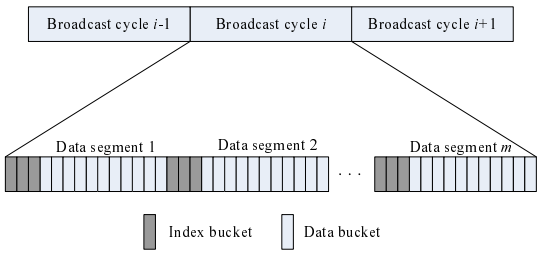


Fig. 2. The $(1, m)$ interleaving scheme

The main motivation behind air indexes is to minimize the power consumption at the mobile client. Although in a broadcast environment the uplink transmissions are avoided, receiving all the downlink packets from the server is not energy efficient. For instance, the Cabletron 802.11 network card (wireless LAN) was found to consume 1400 mW in the *transmit*, 1000 mW in the *receive*, and 130 mW in the *sleep* mode [5]. Therefore, it is imperative that the client switches to the sleep mode (i.e., turns off the receiver) whenever the transmitted packets do not contain any useful information. Based on the data organization technique of Figure 2, the query processing at the mobile client is performed as follows: (i) the client tunes in the broadcast channel when the query is issued, and goes to sleep until the next index segment arrives, (ii) the client traverses the index and determines when the data objects qualifying its query will be broadcast, and (iii) the client goes to sleep and returns to the receive mode only to retrieve the corresponding data objects.

To measure the efficiency of an indexing method, two performance metrics have been considered in the literature: (i) *tuning time*, i.e., the total time that the client stays in the receiving mode to process the query, and (ii) *access latency*, i.e., the total time elapsed from the moment the query is issued until the moment that all the corresponding objects are retrieved. In other words, the tuning time is a measure of the power consumption at the mobile client, while the access latency reflects the user-perceived quality of service.

Most of the existing work on query processing for wireless data broadcast focuses on conventional data and one-dimensional indexes. Imielinski et al. [13] introduce two methods, namely *hashing* and *flexible* index, for retrieving records based on their key values. The same authors [14] propose the aforementioned $(1, m)$ and distributed index techniques, and study their performance under the B$^+$-tree index. Hu et al. [12] consider multi-

attribute queries, and investigate the performance of three indexes (index tree, signature, and hybrid index) under this scenario. In a recent study, Xu et al. [27] propose the exponential index, which offers the ability to adjust the trade-off between access latency and tuning time.

Acharya et al. [1] propose Broadcast Disks, a method that assumes knowledge of the user access patterns. Objects with high access frequency are replicated within the broadcast cycle to reduce access latency. Ref. [2] complements the Broadcast Disks approach with a set of update dissemination policies. Chen et al. [6] also address the problem of skewed data access; they build unbalanced tree structures based on the object frequencies. In this paper, we assume no knowledge about the user access patterns (i.e., objects are equally likely to be queried). In this setting, a flat broadcast program (where each object appears once in the broadcast cycle) achieves the best expected performance [2].

B. Spatial Query Processing using Air Indexes

Hambrusch et al. [9] explore the possibility of broadcasting spatial data together with a data partitioning index. They present several techniques for spatial query processing that adjust to the limited memory of the mobile device. The authors evaluate their methods experimentally for range queries (using the R*-tree [3] as the underlying index), and illustrate the feasibility of this architecture.

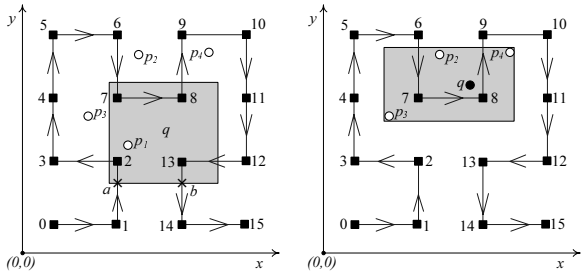
Xu et al. [28] and Zheng et al. [34] focus on *single* nearest neighbor (NN) search in broadcast environments. Both methods utilize a pre-computed Voronoi diagram that can answer any NN query by identifying the Voronoi cell that encloses it. Specifically, the Voronoi diagram of the data objects is built prior to the construction of the air index. The D-tree [28] then recursively partitions the data space (containing the Voronoi cells) into areas with a similar number of cells. This procedure is repeated until each area contains exactly one cell. On the other hand, the *grid-partition* index [34] divides the space into disjoint grid cells, each intersecting multiple Voronoi cells. Both methods are found superior to broadcast solutions based on R*-trees.

Zheng et al. [33] investigate another class of NN queries, namely *linear* NN (LNN) queries, in the context of wireless broadcast systems. A LNN query retrieves the NNs of every point on a line segment, i.e., the solution comprises of a series of points, each being the NN of a particular segment of the line. The authors adjust the methods introduced by Tao et al. [25] to fit the broadcast environment, and show that their techniques outperform the naïve solution where there is no index available.

Most relevant to our work are the techniques related to kNN search on the air. [31] proposes an *approximate* kNN query processing algorithm that is not guaranteed to always return k objects. The idea is to use an estimate r of the radius that is expected to contain at least k points. Using this estimate, the search space can be pruned efficiently at the beginning of the search process. The authors also introduce a learning algorithm that adaptively re-configures the estimation algorithm to reflect the distribution of the data. Regarding the query processing phase, two different approaches are proposed: (i) the standard R*-tree index enhanced with the aforementioned pruning criterion, and (ii) a new *sorted list* index that maintains a sorted list of the objects on each spatial dimension. The sorted list method is shown to be superior to the R*-tree only for small values of k.

Gedik et al. [8] describe several algorithms to improve kNN query processing in sequential-access R-trees. They investigate the effect of different broadcast organizations on the tuning time, and also propose the use of histograms to enhance the pruning capabilities of the search algorithms. Park et al. [20] focus on reducing the access latency of kNN search by accessing the data segment of the broadcast cycle. In particular, they propose a method where the data objects are sorted according to one spatial coordinate. In this way, the client does not need to wait for the next index segment to arrive, but can start query processing immediately by retrieving the actual data objects.

The Hilbert Curve Index (HCI) [32] is a general framework for processing both range and kNN queries in wireless broadcast systems. HCI is based on the $(1, m)$ interleaving scheme; it exploits the linear access of the broadcast channel by transforming the two-dimensional space into a one-dimensional one, using the Hilbert space-filling curve [16]. Once the objects are mapped onto the Hilbert curve, they are indexed with a B$^+$-tree which is then broadcast on the air (as the index segment). Range queries are processed as follows. Consider Figure 3(a) where the Hilbert values range from 0 to 15, and the query region is the shaded rectangle. The client first determines the first (a) and the last (b) points on the Hilbert curve that intersect the query window (illustrated as crosses in the figure). Letting $H(a)$ and $H(b)$ be the Hilbert values of a and b, the client listens to the first broadcast index segment, and retrieves all objects inside the Hilbert range $[H(a), H(b)]$. In our example, $H(a) = 2$ and $H(b) = 13$ (note that the Hilbert value of a point is the integer that corresponds to its closest solid square in the two-dimensional space). Objects p_1, p_2, p_3, p_4 are identified (with $H(p_1) = 2, H(p_2) = 6, H(p_3) = 7, H(p_4) = 9$), but not all of them satisfy the query. Thus, they are mapped back to the two-dimensional space, and their associated data are received (from the corresponding data segment) only if they are inside the query region; in our example, the result includes only p_1.



(a) Range query (b) 2-NN query

Fig. 3. HCI examples

In HCI, kNN queries are processed with a two-step approach. In the first step, the query point q is mapped onto the Hilbert curve, and the k objects closest to q (on the curve) are determined. In the second step, the tightest rectangle that encloses these k objects is calculated, and a range query is processed (in the way described above) to retrieve a set of candidate neighbors. Within this set, the k closest objects to q are identified by comparing their individual distances from q. Figure 3(b) exemplifies this procedure for a 2-NN query q, where $H(q) = 8$. In the first step, the client identifies the $k = 2$ data objects with the closest Hilbert values to $H(q) = 8$; these are p_3 and p_4, with $H(p_3) = 7$ and $H(p_4) = 9$,

respectively. In the second step, the client additionally retrieves p_2 since it falls inside the bounding rectangle of p_3 and p_4 (shown shaded). Among these candidate objects, the $k = 2$ closest ones (i.e., p_2 and p_4) are selected and their contents are retrieved from the corresponding data segments. To conclude the description of HCI, [32] includes an optimization (applicable to both range and kNN queries) that improves the accuracy of the Hilbert curve by partitioning the original space into smaller sub-grids.

The Distributed Spatial Index (DSI) [17] is another general air index, supporting both range and kNN queries. DSI is a distributed index that aims at minimizing the access latency at the cost of an increased tuning time. Similar to HCI, it uses the Hilbert curve to order the data. The broadcast cycle is constructed as follows. The ordered data are partitioned into a number of *frames*. Each frame contains a fixed number of consecutive objects (on the Hilbert curve) and an index table. Each entry of the index table contains a pointer to a subsequent frame, along with the minimum Hilbert value inside that frame. Specifically, the ith entry points to the e^ith future frame, where e is a system parameter. Figure 4 illustrates a situation where each frame contains 2 data objects, $e = 2$, and the subscripts of the objects coincide with their Hilbert order. The index table of every frame contains pointers (and the corresponding minimum Hilbert values) to the 1st, 2nd, 4th, and 8th subsequent frame; the arrows in the figure represent the index entries in Frame 1. These exponentially increasing frame intervals enable fast access to both nearby and distant frames. To identify the object with a specific Hilbert value, the client listens to the current frame, and follows the pointer to the furthest frame that does not exceed the target Hilbert value (i.e., goes to sleep until that future frame is broadcast). The procedure is repeated for this coming frame and terminates when the search converges to the frame that contains the target object. Query processing in DSI is similar to HCI, relying on the locality preservation of the Hilbert curve. The improved access latency of DSI stems mainly from the fact that the client retrieves indexing information directly when it first tunes in the channel (instead of waiting for the next index segment to be broadcast).

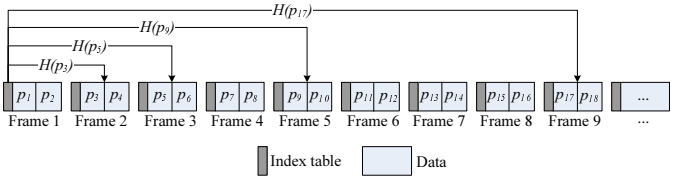


Fig. 4. DSI example

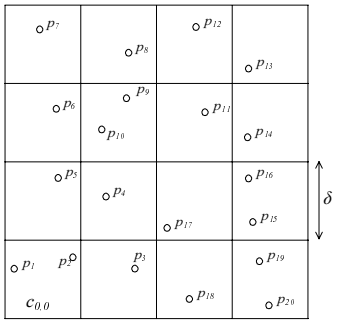
HCI and DSI are the most general and efficient indexes for data on air and, thus, we use them as benchmarks in our experimental evaluation. Note, however, that these methods are designed for snapshot queries over static data. Their adaptation to continuous queries and dynamic data would be inefficient, since they cannot utilize previous results and require query re-computation from scratch at the beginning of each broadcast cycle.

III. Snapshot kNN Queries

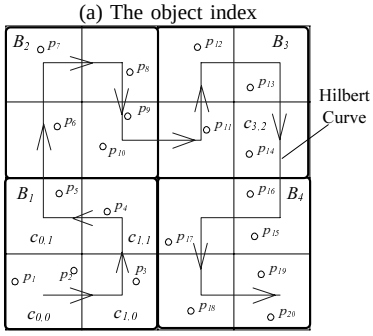
For ease of presentation, we first describe the Broadcast Grid Index (BGI) method focusing on snapshot kNN queries over static data. Section III-A introduces the data index and the procedure that forms the index segment. Section III-B discusses the query processing algorithm at the client side. The application of our techniques to continuous queries, moving objects and range search is presented in Sections IV, V and VI respectively. Although our examples are two-dimensional, the extension to higher dimensionalities is straightforward.

A. Air Index Structure

BGI indexes the data objects with a regular grid, i.e., a partitioning of the data space into square cells of equal size with side-length δ (a system parameter). Each cell stores the object coordinates falling inside, and maintains their total number. Consider Figure 5(a), where the data objects in the system are p_1 to p_{20}, and δ is set as shown. In this example, cell $c_{0,0}$ contains the coordinates of objects p_1 and p_2 and has cardinality 2. Given an object with coordinates x and y, its covering cell is $c_{i,j}$, where $i = \lfloor x/\delta \rfloor$ and $j = \lfloor y/\delta \rfloor$. Similarly, given a cell $c_{i,j}$, its corresponding region is $[i \cdot \delta, i \cdot \delta + \delta) \times [j \cdot \delta, j \cdot \delta + \delta)$. The grid information is placed into packets to form the index segment of BGI. Note that the index segment contains only[1] the object coordinates to keep its size small. Following the $(1, m)$ scheme, the full object information is broken into m data segments, each preceded by a copy of the index segment. The value of m is determined using the analysis of [14].



(a) The object index



(b) Forming the index segment

Fig. 5. The grid index and the index segment

In BGI, the first packet of the index segment provides general system information. It is called *header* packet and contains (i) the data space spatial extents, (ii) the cell side-length δ of the grid, (iii) the data object size in bytes, (iv) the number m of index/data segments per broadcast cycle, (v) the order of the current index segment in the broadcast cycle (i.e., an integer between 1 and

[1]Returning to the restaurant example, each object record may contain several items such as type, menu, prices, etc. Such attributes are not stored in the index segment, but are transmitted in the data segments of the broadcast cycle.

m), and (vi) the cell side-length of the dirty grid (used only in the case of moving objects, as discussed in Section V). The rest of the index segment consists of two parts. The first one, called the *upper level*, contains the cell cardinalities appearing in a specific order (to be discussed next). The second part of the index segment is called the *lower level*, and contains the object coordinates. The lower level is formed by scanning the upper level cells in the specified order, and sequentially placing their contents (object coordinates) into a list. The upper level is transmitted first, followed by the lower one. The detailed structure of the index segment is depicted in Figure 6.

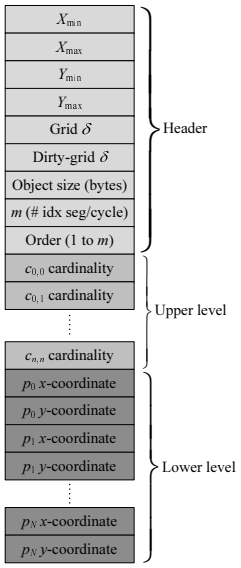


Fig. 6. The structure of the index segment

Concerning the upper level, since the bucket[2] size is typically small, the list of cell cardinalities has to be divided into multiple packets. Let C be the maximum number of integers that fit in a packet. Each bucket B_i consists of a number, called *offset*, followed by $C-1$ cell cardinalities. BGI benefits from compact packet MBRs (minimum bounding rectangles), as it will become clear from the kNN computation algorithm at the client side (described in Section III-B). Therefore, the cells of the grid are visited according to some space-filling curve. In our implementation we use the Hilbert curve, but BGI can be applied with other space-filling curves as well (e.g., the Peano curve). The cell cardinalities are stored into packets in this order, in blocks of size $C-1$. The offset $B_i.offset$ of a bucket B_i is set to the sum of all the cell cardinalities contained in the preceding buckets (i.e., in buckets B_j, where $j < i$).

Returning to our running example, Figure 5(b) shows how the upper level is formed, assuming that the packet size is $C = 5$. Initially, we sort the cells according to the Hilbert value of their centroids, and consider them in this order. Every $C-1$ ($=4$) of them are stored in a bucket. For example, the cardinalities of cells $c_{0,0}$, $c_{1,0}$, $c_{1,1}$, and $c_{0,1}$ form bucket B_1. The offset of B_1 is 0 because it constitutes the first bucket, and $B_1 = \{0, 2, 1, 1, 1\}$. The upper level construction continues with B_2. Its offset equals the sum of cardinalities in B_1, i.e., $B_2.offset = 5$ and $B_2 = \{5, 1, 1, 1, 2\}$. Similarly, $B_3 = \{10, 1, 1, 1, 1\}$, and $B_4 = \{14, 2, 1, 1, 2\}$. The boundaries of the upper level buckets appear bold.

Regarding the lower level of the index, we first form the list of object coordinates. In our example, this order coincides with the object name subscripts. Since $C = 5$ and each object p has a pair of coordinates $p.x$ and $p.y$, the first lower level packet contains $\{p_1.x, p_1.y, p_2.x, p_2.y, p_3.x\}$, the second one $\{p_3.y, p_4.x, p_4.y, p_5.x, p_5.y\}$, and so on. Note that when a client receives an upper level bucket B, it can determine when the contents of each cell c in B will be broadcast. Assume for simplicity that $m = 1$, i.e., the broadcast cycle consists of one index segment followed by a single data segment. If a client receives packet B_3 (when the upper level is transmitted) and determines to process the positions of the objects in cell $c_{3,2}$, then it can acquire the contents of $c_{3,2}$ as follows. The client knows how many more upper level packets remain to be broadcast (their number is fixed). Given that $B_3.offset + 3$ ($= 13$) objects precede the contents of $c_{3,2}$ in the lower level, it can compute when to wake up and enter the receiving mode.

Concerning the data segments, the order of the full information of an object in the broadcast cycle is the same as its order in the lower index level. To conclude the broadcast cycle description, each index or data packet contains a pointer to the first packet (i.e., the header packet) of the next index segment, so that a client that first tunes in the channel knows when to wake up for query processing. We omit these pointers from our examples for the sake of simplicity. After local query processing (i.e., after receiving the necessary packets of the index segment), the client knows the order of its result objects in the lower level and, thus, in the subsequent data segments. Since the full object information has fixed size, it can determine when to tune in to receive the corresponding data packets.

BGI has several advantages compared to other indexing methods. First, the index segment is concise, leading to a shorter broadcast cycle, and lower tuning time and access latency. This is because: (i) the spatial regions of the cells are implicit and they do not have to be transmitted, and (ii) the ordering of the cells and the objects can be easily inferred by the clients, avoiding the extra cost of previous methods that store pointers within their index segment, and from the index segment to the data objects. Finally, the index building and index segment creation procedures are very fast because of the simplicity of the grid. Even though for static objects this is a one-time cost spent when the system starts functioning, in the case that the objects are dynamic (as in Section V) it is essential to keep this cost low; when objects move frequently, the index has to support fast updates so that the new broadcast cycle can be designed on-the-fly.

B. Query Processing

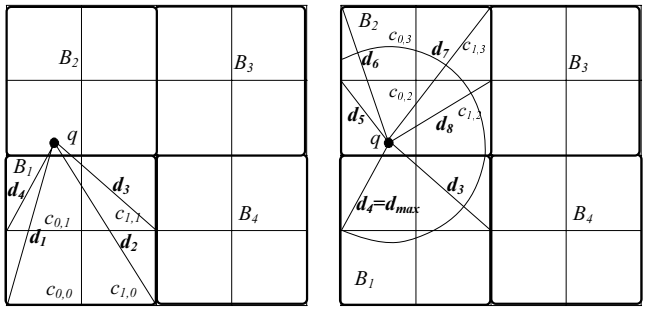
The kNN computation runs completely at the client side. Let q be the client location. Given a cell c, $maxdist(c)$ is the maximum possible distance between any point in c and q. If the cardinality of c is $c.card$, then at least $c.card$ objects lie within distance $maxdist(c)$ from q. Similarly, $mindist(c)$ is the minimum possible distance between any point in c and q. If there are at least k objects within distance d_{max} from q, then a cell c (or bucket) does not have to be considered if $mindist(c) \geq d_{max}$, since it cannot contain any better neighbor.

Based on the above observations, the NN computation algorithm consists of two steps. During the first step, the client

[2]The words bucket and packet are used interchangeably, since they are assumed for simplicity to have equal size.

receives (some) upper level buckets. According to the cardinalities and the $maxdist$ of the contained cells, it computes a conservative upper bound d_{max} of the radius around q that contains at least k objects. During the second step, the client listens to the contents of cells c (in the lower level) that have $mindist(c) < d_{max}$; cells (and buckets) with $mindist$ above d_{max} are skipped. After the second step, the client already knows the coordinates and the packets containing the full information of its k NNs. An important remark is that during each step, the bound d_{max} keeps decreasing, excluding more unnecessary packets from consideration.

Continuing the example of Figure 5(a), Figure 7(a) shows the first step of a 2-NN query at point q. The client initializes d_{max} to infinity, tunes in the broadcast channel, and listens to the packets of the upper level, starting with $B_1 = \{0, 2, 1, 1, 1\}$. Recall from Figure 5(b) that 0 is the number of objects preceding B_1, 2 is the cardinality of $c_{0,0}$, 1 is the cardinality of $c_{1,0}$, and so on. The first cell $c_{0,0}$ has $maxdist(c_{0,0})$ equal to d_1 and cardinality 2. Two virtual entries $<c_{0,0}, d_1>$ are inserted in a list $best\_NN$ that stores the k NNs ordered according to their (actual or conservative) distance from q. The d_{max} equals the key of the kth entry in $best\_NN$, i.e., after the consideration of $c_{0,0}$, d_{max} becomes equal to d_1, because at least $k = 2$ objects lie within distance d_1 from q. The next cell $c_{1,0}$ has $maxdist$ $d_2 > d_{max}$ and it is ignored. The following cell (in the Hilbert order) $c_{1,1}$ has $maxdist$ $d_3 < d_{max}$ and cardinality 1. A new entry $<c_{1,1}, d_3>$ is inserted in $best\_NN$ and an entry $<c_{0,0}, d_1>$ is deleted, i.e., $best\_NN = <c_{1,1}, d_3>, <c_{0,0}, d_1>$. The d_{max} remains d_1, i.e., the distance of the kth (2nd) entry. The last cell $c_{0,1}$ updates $best\_NN$ to $<c_{0,1}, d_4>$, $<c_{1,1}, d_3>$ and d_{max} becomes equal to d_3.

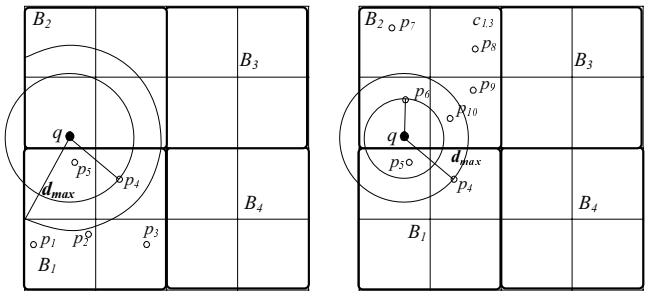


(a) Processing B_1 (b) Processing B_2

Fig. 7. The first step of a 2-NN computation

The next upper level bucket is $B_2 = \{5, 1, 1, 1, 2\}$. Since $mindist(B_2) < d_{max}$ ($= d_3$), B_2 has to be processed. Its cells ($c_{0,2}$, $c_{0,3}$, $c_{1,3}$, and $c_{1,2}$) and their cardinalities are considered like before. As shown in Figure 7(b), d_{max} is updated to d_4, which is the minimum $maxdist$ guaranteed to contain 2 objects (one from $c_{0,2}$ and one from $c_{0,1}$). The first step terminates here and the client sleeps, because $mindist(B_3) \geq d_{max}$ and $mindist(B_4) \geq d_{max}$, meaning that they cannot lead to a better bound.

In the second step (and while the lower level is transmitted), the client listens to the contents of the cells with $mindist$ less than d_{max}. Whenever some new object enters the $best\_NN$ list, the d_{max} is updated to the distance of the kth element in $best\_NN$. Thus, d_{max} shrinks as more objects are considered, enhancing the pruning of the algorithm and skipping unnecessary packets. Continuing our example in Figure 8(a), after processing the objects (coordinates) in cells $c_{0,0}$, $c_{1,0}$, $c_{1,1}$, $c_{0,1}$, $best\_NN = \{p_5, p_4\}$, and $d_{max} = dist(p_4)$. Next, the contents of cells $c_{0,2}$, $c_{0,3}$, $c_{1,3}$, and $c_{1,2}$ are broadcast (Figure 8(b)). Object p_6 in $c_{0,2}$ updates $best\_NN = \{p_5, p_6\}$ and $d_{max} = dist(p_6)$. Processing continues with $c_{0,3}$ but the result does not change. The next cell contents in the lower level correspond to $c_{1,3}$. Since $mindist(c_{1,3}) > d_{max}$, the client does not listen to the objects in $c_{1,3}$. Finally, the objects in $c_{1,2}$ are considered without, however, altering the result. The algorithm terminates here with $best\_NN = \{p_5, p_6\}$. When the information about p_5 and p_6 is broadcast in the data segment, the client wakes up and receives it.



(a) Processing $c_{0,0}$, $c_{1,0}$, $c_{1,1}$, $c_{0,1}$ (b) Processing $c_{0,2}$, $c_{0,3}$, $c_{1,2}$

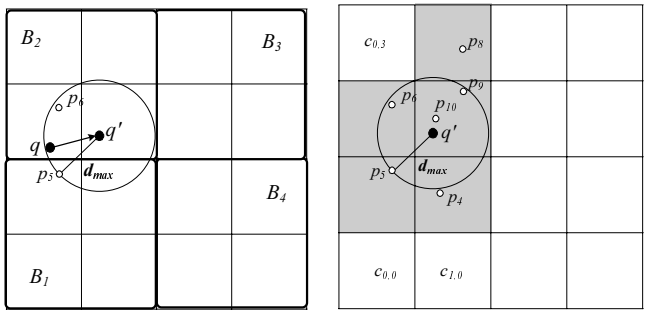
Fig. 8. The second step of a 2-NN computation

The kNN computation algorithm is shown in Figure 9. Lines 1 to 8 implement step one, while lines 9 to 15 implement step two. The first step uses the $best\_NN$ list to store the cells with the lowest $maxdist$ values. In particular, each cell c in a considered bucket B, generates up to $c.card$ virtual entries $<c, maxdist(c)>$ into the $best\_NN$ list, where $c.card$ is the cardinality of c. In the second step, when a cell c is considered, we delete from $best\_NN$ all its virtual entries ($<c, maxdist(c)>$) and insert its actual objects p (if $dist(p) \leq d_{max}$). Maintaining the same $best\_NN$ list in both steps enhances the pruning power of the method; the $maxdist$ of a cell not transmitted so far is used to prune cells in line 9, even though its exact contents are not known.

kNN Computation

// Client at q goes online, and listens to the first index segment
// Step 1: The upper level is broadcast
1. $best\_NN = \varnothing$; $d_{max} = \infty$
2. **for** each bucket B
3. **if** $mindist(B) < d_{max}$ // Prune upper level buckets
4. **for** each cell c in B
5. **for** $iter = 1$ to $c.card$
6. **if** $maxdist(c) < d_{max}$
7. Delete the kth entry of $best\_NN$
8. Insert $<c, maxdist(c)>$ to $best\_NN$; Update d_{max}
// Step 2: The lower level is broadcast
9. **for** each cell c with $mindist(c) < d_{max}$
10. Delete all entries of c from $best\_NN$
11. **for** each object p in c
12. **if** $dist(p) \leq d_{max}$
13. Delete the kth entry of $best\_NN$
14. Insert $<p, dist(p)>$ into $best\_NN$; Update d_{max}
15. **return** $best\_NN$

Fig. 9. The kNN computation algorithm



(a) First step (b) Second step

Fig. 10. 2-NN monitoring over static data

IV. Continuous kNN Queries on Static Data

In this section we consider continuous kNN queries from moving clients over static data objects. In this scenario, the server broadcasts exactly the same information as above, but the clients continuously monitor their k NNs as they move. A straightforward processing method is to compute from scratch the k NN set whenever the client changes position. This, however, may be very expensive. We propose an alternative monitoring algorithm that re-uses the previous query result in order to reduce the number of packets received.

Assume that a client moves from point q to point q', and let $best\_NN$ be its previous result (i.e., the kNN set at q). Since the objects are static, the old NNs of the client are still at their previous location. Thus, we can directly compute an upper bound d_{max} as the maximum distance between the old NNs and the new client location q'. This bound allows the client to start pruning index buckets immediately. The kNN monitoring algorithm for static objects is the same as in Figure 9, the difference being in line 1, where d_{max} should be initialized as $d_{max} = \max_{p \in best\_NN}(dist(p, q'))$. Figure 10(a) shows an example, where a 2-NN query moves from q to q'.

The old NNs are p_5 and p_6 and d_{max} is initialized to $dist(p_5, q')$. When the upper level is broadcast (Figure 10(a)), the client receives bucket B_1 but the cells therein do not have sufficiently small $maxdist$ to further decrease d_{max}. Bucket B_2 is also considered. Similar to B_1 its cells have $maxdist$ greater than d_{max}. On the other hand, buckets B_3 and B_4 are ignored since their $mindist$ is larger than d_{max}. The client sleeps and waits until the lower level is broadcast to perform the second step of the NN computation algorithm (Figure 10(b)). During this step, cells $c_{0,0}$, $c_{1,0}$, $c_{0,3}$ are pruned and only the contents of the shaded cells are considered. The algorithm finally returns $best\_NN = \{p_6, p_{10}\}$ as the result.

V. Continuous kNN Queries on Moving Data

In this section we address the problem of continuous kNN queries, where both the query points and the data objects may move arbitrarily. Section V-A describes the basic algorithm, while Section V-B introduces an optimization that utilizes some stored information at the client side.

A. Basic Algorithm

We now assume that the data objects may move, appear or disappear; e.g., in our taxi example, cabs may move, new ones may enter service or existing ones may go off duty. When an object moves, it sends an update to the server including its id, its old and its new coordinates. When an object appears (disappears), it informs the server of this event, providing also its id and (expired) location. The task of the server is to update the data grid and restructure the broadcast cycle. If the object lists of the cells are implemented as hash-tables, the grid supports updates (i.e., object insertions and deletions) in constant expected time. Note that an object movement is equivalent to a deletion from its old location and an insertion to its new one.

The object updates that arrive to the system during a broadcast cycle are buffered and take place on the grid at the end of the cycle. That is, the index segments during a broadcast cycle are identical, and the up-to-date information is broadcast when the next cycle commences. The creation of the index segment for the next broadcast cycle is performed on-the-fly. The upper level buckets B_i contain the same cells in the same order, and their new offset and cell cardinalities are found at transmission time. The lower level simply places into buckets the new contents of the cells. Similarly, the full object information is placed in the same order into consecutive packets, forming the data segments.

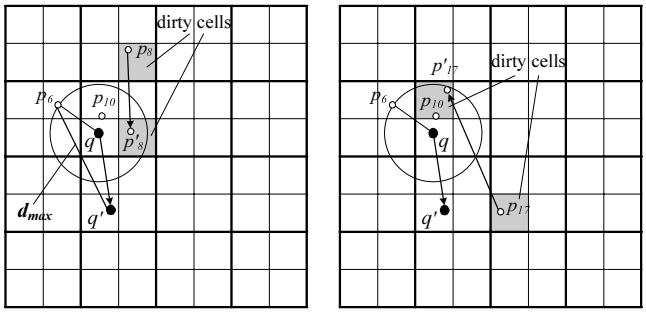
Concerning the query processing at the client side, the first-time result of a continuous kNN query is computed in the way described in Section III, by listening to the next broadcast index segment. The kNN monitoring, however, is more complex; the ordering of the previous NNs might have changed since the previous broadcast cycle (timestamp), and some outer objects may now lie closer to the query than the old NNs. If the client also changes position, the problem becomes even more complicated. To avoid frequent NN computations from scratch, BGI re-evaluates only the queries that may be affected by the updates in the last timestamp.

In particular, a *dirty grid* is broadcast in the beginning of each cycle (prior to the upper level of the first index segment), indicating the regions of the data space that received updates during the cycle. The dirty grid is a regular grid, whose granularity is typically finer than that of the object grid (i.e., its cell side-length is smaller). Each cell of the dirty grid contains one bit (and no actual objects or cardinality information). Before applying the updates received in a broadcast cycle, all the cells of the dirty grid are initialized to contain 0. If some object insertion or deletion takes place inside the region covered by a cell of the dirty grid, then its bit is set to 1, and the cell is said to be dirty. Note that, even if an object moves within the boundaries of its dirty cell, the cell is still marked as dirty (since each object movement is equivalent to a deletion followed by an insertion).

Each client tunes in the broadcast channel and listens to the dirty grid in the beginning of each broadcast cycle. It re-evaluates its query only if: (i) some dirty cell overlaps with the circular disk $circ(q)$ with center at the client location q and radius equal to the distance of its previous kth NN, or (ii) if the client moves to a new position q'. Otherwise, the result is unaffected by the updates in the last timestamp, and the client can sleep until the beginning of the next broadcast cycle. Even if re-evaluation is necessary, existing information may be re-used to facilitate efficiency.

Consider the example of Figure 11(a), where the client moves to a new location q' and object p_8 moves to p'_8. Since the dirty cells do not contain any of the previous NNs p_6 and p_{10}, these objects have not moved. This provides an initial bound d_{max}=$\max(dist(p_6, q'), dist(p_{10}, q'))$ =$dist(p_6, q')$ for the NN

computation at q'. On the other hand, in Figure 11(b), the update of p_{17} affects the cell of p_{10}. The client does not know whether p_{10} is at its previous position (e.g., it may have been deleted), and no bound d_{max} can be computed before the call of the NN computation algorithm.

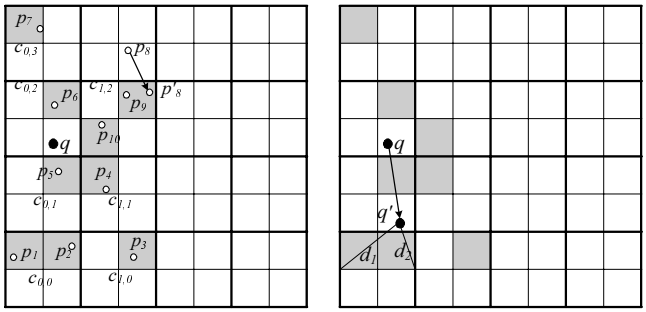


(a) p_8 issues an update (b) p_{17} issues an update

Fig. 11. 2-NN monitoring over dynamic data

B. Optimization

In order to provide "tight" d_{max} values for NN re-computations in highly dynamic environments, we may store some extra information at the client side. Since for the received cells we have the concrete object locations, the maintained information is kept at the finest, i.e., dirty grid granularity. We illustrate this optimization using the running example of Figure 8, where the client computes from scratch its 2 NNs. During the second step of the algorithm, the client listens to the contents of cells $c_{0,0}$, $c_{1,0}$, $c_{0,1}$, $c_{1,1}$, $c_{0,2}$, $c_{1,2}$ and $c_{0,3}$. We map the received object coordinates into dirty grid cells. Then, we insert each such non-empty cell into a list, along with the number of objects inside[3]. In our example, we store the id and the cardinality of the shaded cells in Figure 12(a).



(a) Dirty cells maintained (b) q moves to q'

Fig. 12. NN re-computation optimization

In the subsequent broadcast cycles, (i) if some stored cells become dirty, we remove them from the list, and (ii) if a re-computation is necessary, we acquire an initial d_{max} according to the cardinalities and $maxdist$ of the stored cells, prior to the NN retrieval. Continuing the example of Figure 12(a), assume that in the next timestamp object p_8 moves to p'_8. Its old and new cells are marked as dirty. The client receives the dirty grid information and removes the cell of p'_8 from its list, but it does not resort to NN re-computation because its NN list $\{p_5, p_6\}$ is not affected by the update. Assume now that in the next timestamp, the client moves from q to q' (Figure 12(b)) and has to re-evaluate the query. Before invoking the algorithm of Figure 9, it considers the cardinalities of the cells in its list (i.e., the shaded cells). For each such cell, it assumes that all the objects inside lie at distance $maxdist$ from q', and computes a bound d_{max} accordingly. In our example, all the cells have cardinality 1 and the client sets as d_{max} the second smallest $maxdist$ (i.e., $d_{max} = d_1$). The complete kNN monitoring algorithm is shown in Figure 13.

kNN Monitoring
// Client at q has completed the kNN computation of Figure 9,
// and maintains a list L of dirty grid cell cardinalities
1. Listen to the contents of the next dirty grid
2. **for** each dirty grid cell c with its bit set
3. Delete c from L
4. **if** no dirty cell overlaps with $circ(q)$ **and** $q' = q$
5. **return** // result set has not changed
6. **else**
7. $best\_NN = \varnothing$; $d_{max} = \infty$
8. **for** each object p in the current result
9. **if** p's cell is not dirty
10. Insert $<p, dist(p, q')>$ to $best\_NN$
11. **for** each cell c in L
12. **for** $iter = 1$ to $c.card$
13. **if** $maxdist(c) < d_{max}$
14. Delete the kth entry of $best\_NN$
15. Insert $<c, maxdist(c)>$ to $best\_NN$; Update d_{max}
16. Invoke the kNN computation algorithm of Figure 9,
17. using the current value of d_{max}
18. **return**

Fig. 13. The kNN monitoring algorithm

The overhead of the dirty grid is expected to be low since for each cell we send a single bit. Furthermore, its size can be reduced by the *run-length* compression scheme. In this scheme, large blocks of consecutive bits with the same value (i.e., all 0 or all 1) are represented by a single integer. To achieve high compression ratios, the order of bits follows the Hilbert order of the corresponding cells in the dirty grid. Since object updates in most real-world applications exhibit locality, ordering according to a space-filling curve leads to long blocks of ones or zeros, when the corresponding spatial region is "hot" (in terms of updates) or not. Finally, as the header packet is typically not full, the free space can be occupied by (part of) the dirty grid information.

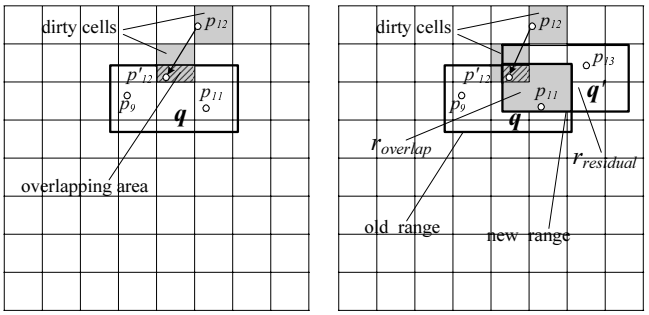
VI. Range Queries

In this section we discuss how BGI extends to range queries. For simplicity we assume that the ranges are rectangular. Snapshot queries over static data can be answered trivially. The client tunes in the channel and waits until the first index segment is broadcast. Among the buckets B_i of the upper level, it listens only to the ones whose MBRs overlap with its range. Similarly, it considers only the cells in the lower level that intersect the range.

Next, we consider continuous range monitoring over static data. The first-time result of the query is computed with the algorithm described above. When the client moves, we make use of the previous result of the query. In particular, the client already knows the objects falling in the intersection of the old and the new ranges. Therefore, it computes only the objects falling in the remaining part of the new range.

[3]Note that since the memory of the client is limited, we store the cardinality and not the actual contents of the cell.

Regarding continuous ranges queries over dynamic data, BGI uses the dirty grid technique. Assume that the query is static. If no dirty cell intersects its range, then re-computation is avoided. If some dirty cells overlap the range, then partial re-computation is performed. Figure 14(a) illustrates such a case, where a dirty cell intersects the query rectangle. The client computes the overlapping area (appearing striped) between the range and the dirty cell, and evaluates a range query for this area.



(a) p_{12} issues an update (b) q moves to q'

Fig. 14. A continuous range query over dynamic data

In the case that the query moves, then its new range r_{new} is partitioned into: (i) the overlapping area $r_{overlap}$ between the old and the new range, and (ii) the residual area $r_{residual} = r_{new} - r_{overlap}$. The client has to compute all the objects in $r_{residual}$. Additionally, if some dirty cells intersect the $r_{overlap}$, then it also has to compute the objects falling in the intersection of these dirty cells with $r_{overlap}$. Consider Figure 14(b) where q moves to q'. Object p_{12} moves to a new position p'_{12}, resulting in two dirty cells. One of these cells overlaps with $r_{overlap}$ and their intersection is the striped rectangle. The client tunes in the channel and listens to all upper level buckets whose MBRs intersect either the striped area or $r_{residual}$. Similarly, from the lower level it listens to the contents of the cells intersecting one of the aforementioned areas.

VII. Experimental Evaluation

In this section we evaluate the performance of the proposed methods under various system parameters. We use a real[4] spatial dataset (REAL) containing the locations of 5,848 cities and villages in Greece (available at *www.rtreeportal.org*). Additionally, in order to investigate scalability, we generated 5 skewed datasets (SKEW) where the object locations follow a Zipf distribution with parameter 0.8. All datasets are scaled to fit in a [0,10000]^2 workspace. Assuming that the data objects are distributed on a 50km×50km area, the average density of the objects varies between 2.3 and 80 objects/km^2.

For static data, each object corresponds to a point in the dataset. For generating moving data, we randomly select the initial position and the destination of each object from the spatial dataset. The object then follows a linear trajectory (with constant velocity) between the two points. Upon reaching the endpoint, a new random destination is selected and the same process is repeated. Distance is defined according to the Euclidean metric. To further control the object movement, only a certain fraction (which we call *agility*) of the objects move during each timestamp. The same pattern is also adopted for the moving queries.

In each experiment we generate and evaluate 10,000 random queries. In the continuous case the queries are evaluated for a period of 100 timestamps. We use the tuning time and access latency as the performance metrics. The results correspond to the average measurements over all queries, expressed in number of packets. The size of each data object is fixed to 128 bytes. Table I summarizes the parameters under investigation, along with their ranges. Their default (median) values are typeset in boldface. In each experiment we vary a single parameter, while setting the remaining ones to their default values. Section VII-A evaluates snapshot queries, Section VII-B deals with continuous queries over static objects, while Section VII-C considers the case where both the queries and the data objects are mobile. Finally, Section VII-D investigates the effect of packet losses on the performance of our methods.

TABLE I
System parameters

Parameter	Range
SKEW DB size	10K,50K,**100K**,150K,200K
Packet size (bytes)	64,128,**256**,512,1024
Range query area (km^2)	12.5,25,**50**,100,200,400
Number of NNs (k)	1,2,**4**,8,16,32
Object/Query speed (km/h)	5,**25**,125
Object/Query agility (%)	0,10,20,30,40,**50**,60,70,80,90,100

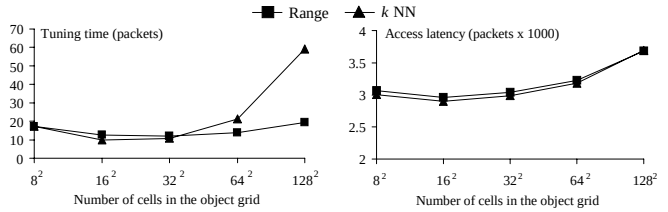
A. Snapshot Queries

In this section we compare BGI against HCI and DSI, which, as discussed in Section II-B, are the current state-of-the-art frameworks in terms of tuning time and access latency, respectively. For BGI and HCI we employ the $(1, m)$ interleaving scheme, where the value of m is set according to the methodology in [14]. For HCI we use the space partitioning optimization with a 4×4 grid (the default setting used in [32] for the REAL dataset).

The first experiment studies the effect of the grid granularity on the performance of BGI using the REAL dataset. We divide each axis into a number of equal intervals that varies between 8 and 128. Figure 15(a) shows the tuning time for range and kNN queries as a function of the grid size. A grid of 16×16 cells is the best choice overall, and we use this value in the remainder of this section. A coarser grid (e.g., 8×8) reduces the size of the upper level index (and, thus, the size of the index segment), but query processing considers more objects (equivalently, receives more lower level packets). On the other hand, a finer grid is more expensive, because the increased index size offsets any potential benefit in precision. Figure 15(b) illustrates the access latency for the same experiment. Again, BGI performs best for a 16×16 grid. Even though the index segment is smaller for a 8×8 grid, the latency is larger because the objects are less clustered and the full data of the result objects are placed further away from each other in the broadcast cycle.

Figure 16(a) (16(b)) depicts the tuning time for range (kNN) queries as a function of the packet size. As the packet size increases, the clients receive fewer packets with any method. The tuning time of BGI is 2 to 3 times lower than HCI because the index (a B$^+$-tree) of HCI is large, and query processing on the one-dimensional Hilbert curve is less efficient than processing with a grid in the original space. BGI is around 10 times better

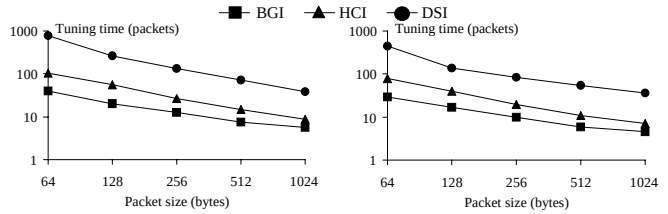
[4]The same dataset is used in the evaluation of [17] and [32].



(a) Tuning time (b) Access latency

Fig. 15. Performance versus grid granularity (REAL)

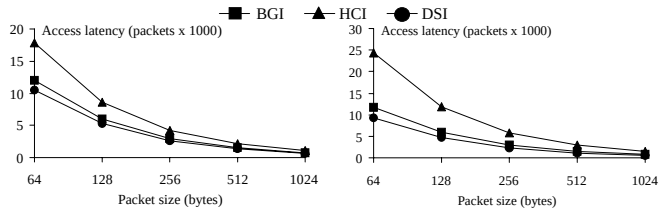
than DSI, mainly because DSI is designed to sacrifice tuning time for the sake of lower latency, and also because of its Hilbert curve based query evaluation. Note that the tuning time of HCI is considerably smaller than that of DSI. To avoid confusion, this fact does not contradict the results shown in [17], since [17] evaluates DSI against a *distributed* version of HCI (which sacrifices tuning time for access latency compared to the $(1, m)$ version used here).



(a) Range queries (b) kNN queries

Fig. 16. Tuning time versus packet size (REAL)

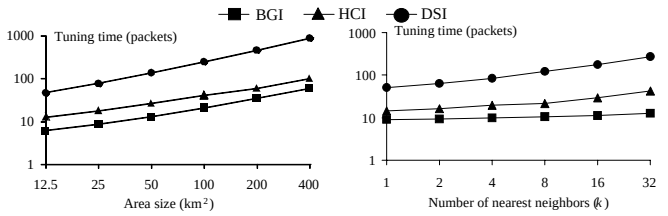
Figure 17 shows the access latency for the same experiment. For range queries, HCI has around 50% higher latency than our method, because of its larger index and the Hilbert ordering that may place the full information of the result objects far from each other in the broadcast cycle. For kNN queries, the latency of HCI is even higher (around 2 times worse than BGI), due to the two-step kNN retrieval that performs a kNN search on the Hilbert curve, followed by a range query on the returned window. DSI has 20% and 12% lower latency than BGI, for range and kNN queries, respectively, because of its distributed nature. Note that even though DSI achieves slightly lower access latency than our method, its tuning time is an order of magnitude higher (as demonstrated in Figure 16).



(a) Range queries (b) kNN queries

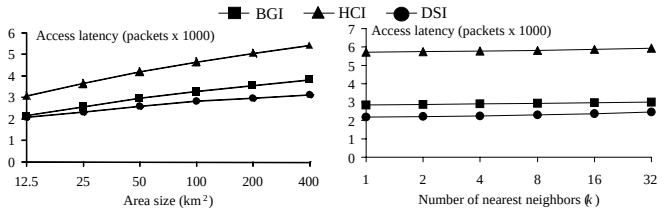
Fig. 17. Access latency versus packet size (REAL)

Figure 18 investigates the effect of the query selectivity on the tuning time. As the selectivity increases, the performance of all algorithms deteriorates due to the larger search region and result size. BGI is consistently better than both its competitors,



(a) Range queries (b) kNN queries

Fig. 18. Tuning time versus query selectivity (REAL)



(a) Range queries (b) kNN queries

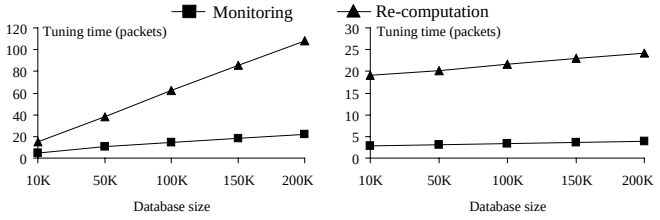
Fig. 19. Access latency versus query selectivity (REAL)

for the reasons explained in the context of Figure 16. For kNN queries, our method is less sensitive to the parameter k because multiple nearest neighbors may be found in the same cell of the grid. Figure 19 shows the access latency for the same experiment. The latency equals to the number of packets between the first time tuning in of the client and the position of the furthest result object in the broadcast cycle. As the selectivity (equivalently, the result size) increases, so does the latency. The latency for ranges is larger than kNN because they return more objects; e.g., for the default settings of Table I, a range query with a 50 km^2 area returns on the average 117 objects, while a 4-NN one retrieves only 4.

B. Continuous Queries over Static Data

Next, we evaluate the monitoring version of BGI, as discussed in Section IV. Specifically, we consider static objects and continuous moving queries. Since both HCI and DSI are designed for snapshot queries, we compare our method with a naïve *re-computation* approach that re-evaluates each query from scratch at the beginning of every broadcast cycle (using the basic BGI algorithm of Section III). To verify the generality of our results we use the synthetic datasets (SKEW) in our experiments, in addition to the REAL one. Since the database size and distribution of SKEW are very different from those of REAL, we fine-tuned the object grid granularity in a manner similar to Figure 15. The results indicated that a 64×64 grid provides good results for all the tested database sizes of SKEW. Concerning the query movement, unless otherwise specified, the queries are assumed to move with a medium speed and have an agility of 50%.

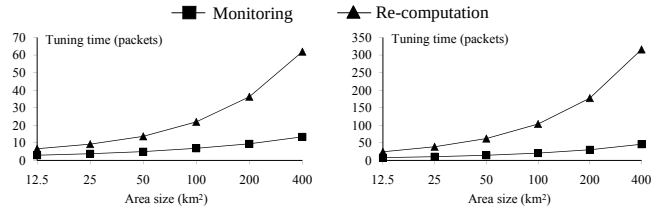
In the first experiment we study the effect of the DB size (for SKEW) on the performance of BGI. Figure 20 shows the tuning time for various database sizes, ranging from 10K to 200K objects. For range (kNN) queries, the monitoring method is 3 to 5 (6 to 7) times better than re-compute, due to the re-use of previous results. For kNN queries the gap is larger, because NN processing is more complicated than the range one.



(a) Range queries (b) kNN queries

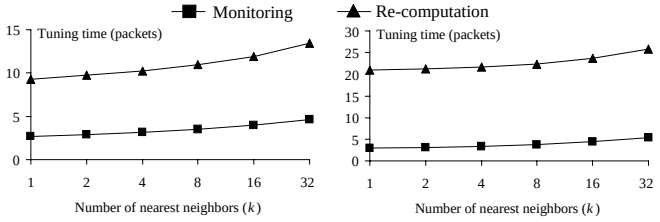
Fig. 20. Tuning time versus database size (SKEW)

Figures 21 and 22 illustrate the tuning times of the monitoring and re-computation versions of BGI versus the query selectivity for range and kNN queries, respectively. Similar to Figure 18, higher selectivity implies higher tuning time. The monitoring technique reduces the number of received packets by a factor of 2 to 7 compared to re-compute. For range queries, as the selectivity increases, monitoring becomes even better than re-computation because there is larger overlapping area between the old and new range of the moving queries. On the other hand, for kNN queries, the relative performance of the algorithms is practically independent of k, since, contrast to range monitoring, the previous NNs cannot be inserted into the new result directly.



(a) REAL (b) SKEW

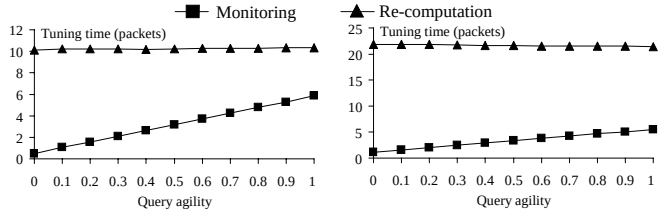
Fig. 21. Tuning time versus query selectivity (range)



(a) REAL (b) SKEW

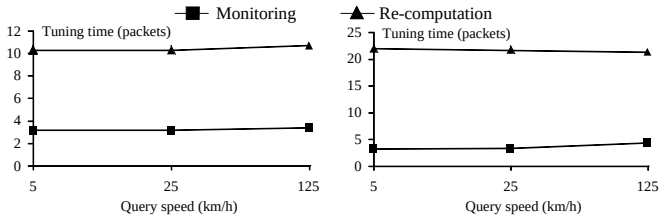
Fig. 22. Tuning time versus number k of NNs

Finally, Figures 23 and 24 investigate the effect of the moving behavior (i.e., agility and speed) of the query on the tuning time, for kNN queries. Re-computation is unaffected by query movements, since it computes the NNs from scratch at every cycle in any case. For monitoring, a high query agility leads to increased tuning time, as it performs more frequent partial re-computations. Nevertheless, it achieves savings of 43% (for REAL) and 74% (for SKEW) even for constantly moving queries (i.e., agility 1). When the query moves faster, the performance of monitoring slightly deteriorates, because d_{max} becomes looser. The results for range queries follow the same trends and are omitted.



(a) REAL (b) SKEW

Fig. 23. Tuning time versus query agility (kNN)

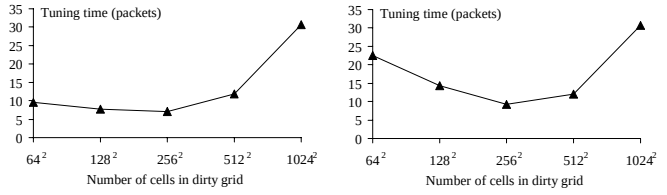


(a) REAL (b) SKEW

Fig. 24. Tuning time versus query speed (kNN)

C. Continuous Queries over Dynamic Data

In this section we evaluate BGI (as described in Section V) in an environment where both queries and objects are mobile. Again, due to the absence of a competitor, we use the naïve (re-computation) approach as the baseline. To fine-tune the dirty grid granularity for the monitoring technique, in Figure 25 we measure the tuning time for various grid sizes. A 256×256 dirty grid achieves the best trade-off between grid size and precision. We use this granularity for all the remaining experiments. We focus solely on kNN, since range queries do not benefit from a monitoring algorithm in a system with a high rate of updates. This is because range queries need to access only a small number of cells (i.e., the ones intersecting the query range), and the savings of the dirty grid are counter-balanced by its overhead. Thus, the diagrams for the monitoring and re-computation approaches are almost identical.



(a) REAL (b) SKEW

Fig. 25. Tuning time versus dirty grid granularity (kNN)

Figure 26 shows the tuning time as a function of the database size. Clearly, the monitoring version of BGI scales well with increasing database size, and is almost 2 times better than re-computation in all cases. Figure 27 investigates the effect of k. Comparing the results to Figure 22 (for static objects) it is evident that frequent object updates decrease the pruning power of the BGI monitoring algorithm. Nevertheless, it performs considerably better than re-compute, as it significantly benefits from re-using the previous NNs and the dirty grid information.

Figures 28 and 29 illustrate the effect of the object moving

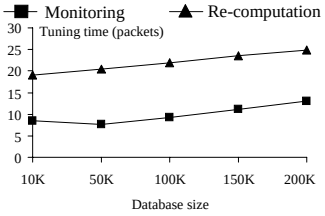
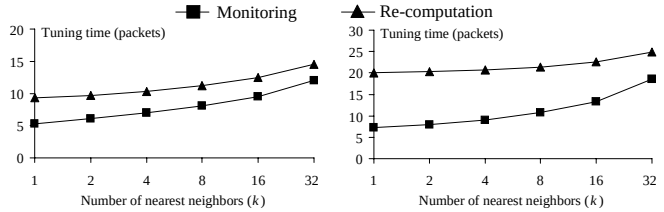


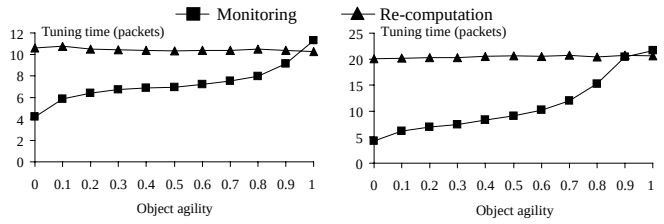
Fig. 26. Tuning time versus database size (SKEW, kNN)



(a) REAL (b) SKEW

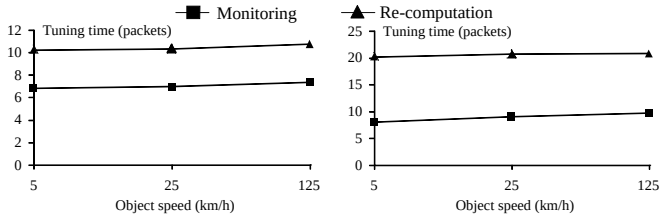
Fig. 27. Tuning time versus number k of NNs

behavior on the tuning time. Clearly, re-computation is unaffected by object movements. The tuning time of monitoring degrades with object agility, since more query results are re-computed per timestamp. For constantly moving objects (i.e., agility 1), all queries are affected and monitoring is worse than re-computation due to the overhead of the dirty grid. On the other hand, its performance is independent of the object speed, since an object movement is treated as a deletion from its old position and an insertion to the new one, and the probability that any of these positions affects some query is independent of their distance (i.e., the object moving distance).



(a) REAL (b) SKEW

Fig. 28. Tuning time versus object agility (kNN)



(a) REAL (b) SKEW

Fig. 29. Tuning time versus object speed (kNN)

Next, we study the feasibility of implementing continuous monitoring over the existing cellular infrastructure. Recall that updates occurring during the current broadcast cycle are transmitted at the next one. Thus, clients do not always know the most up-to-date location of all objects. In order to minimize this inaccuracy, the broadcast cycle should be short. Figure 30 illustrates the duration of the cycle (in seconds) as a function of the database size (the number of clients does not affect the duration) for wireless networks supporting data rates of 384 Kbps and 2 Mbps. These rates are chosen because 2 Mbps is the maximum rate for the 3G standard, while 384 Kbps is the rate of the current systems.

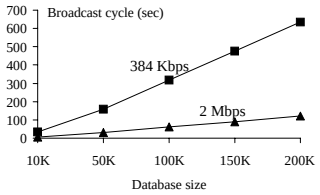


Fig. 30. Duration of broadcast cycle versus database size (SKEW, kNN)

For a 384 Kbps bandwidth, 10K objects are supported with a minimum accuracy of around 30 sec. On the other hand, a 2 Mbps rate can provide an accuracy of 1 minute for 100K objects. We believe that these delays are acceptable since the targeted applications (e.g., taxi example of Figure 1) involve a relatively small number of data objects (in the order of thousands) and a large number of clients (in the order of millions). Furthermore, as the infrastructure for wireless broadcasting improves, the supported database sizes will increase accordingly. Finally, note that the inaccuracy is not specific to BGI, but inherent to all broadcasting methods since the broadcast cycle cannot be restructured to accommodate object updates on-the-fly. This is the case also for snapshot queries [14].

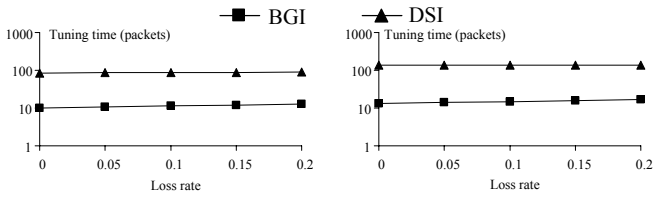
D. Robustness to Packet Loss

The results reported in the previous sections assume a perfect channel where no packet losses occur. However, the typical wireless channel is error-prone due to several factors, such as radio interference, fading, attenuation, etc. Specifically, packet error rates in wireless networks are reported to range from 1% up to 10% [23]; resilience to link errors is, thus, a very desirable property for any air indexing scheme. In this section we investigate the performance of our methods in lossy environments.

First, we consider snapshot queries. We compare only against DSI, because HCI is not designed to deal with packet loss, while its distributed version (which does take packet losses into account) is worse than DSI in both access latency and tuning time [17]. In BGI, errors are handled in a straightforward manner: a lost packet in the current index segment has to be recovered in the subsequent index segment. However, query processing may continue in the current segment even after a packet loss. Specifically, for range queries, BGI continues normally in the current index segment, but the lost packet is received in the next segment. If the lost packet is in the upper level, then the corresponding lower level ones must also be accessed in the next segment. For kNN queries, error handling in BGI is similar, except for the case where an upper level packet is lost; in this situation BGI halts and resumes when the lost packet is broadcast again (i.e., in the next index segment).

Figure 31 shows the tuning time of BGI and DSI (for snapshot queries) under different packet loss rates for the REAL dataset. DSI is affected to a lesser degree than BGI, because its distributed structure offers multiple search paths towards any frame of the index. Nevertheless, BGI retains its efficiency achieving a 7-10

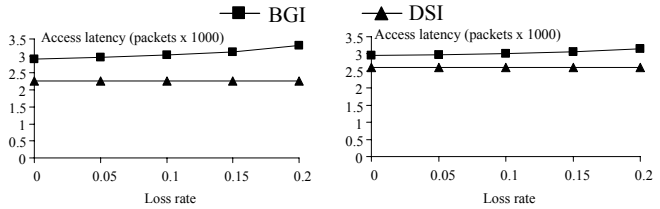
times smaller tuning time than DSI in all cases, because it needs to (successfully) receive only a few index packets (see Section VII-A).



(a) kNN (b) Range

Fig. 31. Tuning time versus packet loss rate (snapshot queries, REAL)

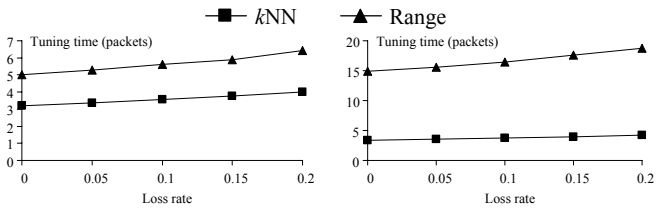
Figure 32 illustrates the access latency for the same experiment. DSI remains practically unaffected by the link errors, due to its distributed nature that enables query processing to resume directly after a packet loss. BGI also performs very well, and the degradation due to errors is below 14% in all cases. Note that, similar to the performance evaluation in [17], we assume that only index packets may be received erroneously. When a data packet is lost, it has to be recovered in the next broadcast cycle, and its effect on the access latency is independent of the underlying air indexing method.



(a) kNN (b) Range

Fig. 32. Access latency versus packet loss rate (snapshot queries)

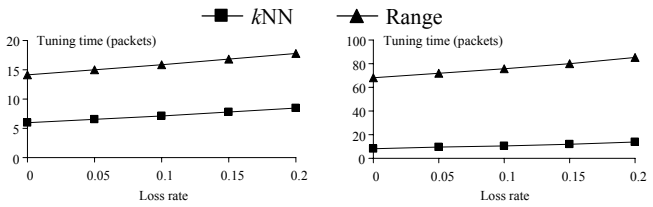
Figures 33(a) and 33(b) depict the effect of the packet loss rate on the tuning time of our monitoring algorithm for continuous queries over static data (using REAL and SKEW, respectively). BGI is not affected significantly by link errors, and the maximum performance degradation compared to a perfect channel (no packet loss) is 28%.



(a) REAL (b) SKEW

Fig. 33. Tuning time versus packet loss rate (continuous queries, static data)

Finally, Figure 34 shows the tuning time of our monitoring algorithm (for dynamic data/queries) as a function of the loss rate. Recall that, in the case of dynamic data, we use a dirty grid to mark the cells that have been affected by the object movements. In the presence of errors, our algorithm works as follows: if the first packet of the broadcast cycle (that contains the dirty grid) is corrupted, the client just re-evaluates the query from scratch (without having to wait for the next broadcast cycle); otherwise, query processing is performed according to the algorithm of Figure 13. Range queries are affected less by packet losses, and the increase of the tuning time for a 20% loss rate (compared to a perfect channel) is 25% for both datasets. On the other hand, the maximum performance degradation for kNN queries is 41% and 65% for the REAL and SKEW datasets, respectively.



(a) REAL (b) SKEW

Fig. 34. Tuning time versus packet loss rate (continuous queries, dynamic data)

VIII. Conclusions

In this paper we study spatial query processing in wireless broadcast environments. A central server transmits the data along with some indexing information. The clients process their queries locally, by accessing the broadcast channel. In this setting, our target is to reduce the power consumption and the access latency at the client side. We propose an on-air indexing method that uses a regular grid to store and transmit the data objects. We design algorithms for snapshot and continuous queries, over static or dynamic data. To the best of our knowledge, this is the first study on air indexing that (i) addresses continuous queries, and (ii) considers moving data objects. We demonstrate the efficiency of our algorithm through an extensive experimental comparison with the current state-of-the-art frameworks for snapshot queries, and with the naïve constant re-computation technique for continuous queries. A challenging problem is to devise cost models for continuous monitoring of spatial queries in wireless broadcast environments. Such models could reveal the best technique given the problem settings, help fine-tune several system parameters (e.g., grid size) and potentially lead to better algorithms. Another interesting direction for future work is to study different types of spatial queries, such as reverse nearest neighbors, and to extend our framework to process their snapshot and continuous versions.

Acknowledgments

We would like to thank Baihua Zheng for providing us with the implementations of HCI and DSI. This work was supported by grant HKUST 6184/05 from Hong Kong RGC.

References

[1] S. Acharya, R. Alonso, M. J. Franklin, and S. B. Zdonik. Broadcast disks: Data management for asymmetric communications environments. In *SIGMOD*, 1995.

[2] S. Acharya, M. J. Franklin, and S. B. Zdonik. Disseminating updates on broadcast disks. In *VLDB*, 1996.

[3] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[4] Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *MDM*, 2004.

[5] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. In *MOBICOM*, 2001.
[6] M.-S. Chen, P. S. Yu, and K.-L. Wu. Indexed sequential data broadcasting in wireless mobile computing. In *ICDCS*, 1997.
[7] B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, 2004.
[8] B. Gedik, A. Singh, and L. Liu. Energy efficient exact kNN search in wireless broadcast environments. In *GIS*, 2004.
[9] S. E. Hambrusch, C.-M. Liu, W. G. Aref, and S. Prabhakar. Query processing in broadcasted spatial index trees. In *SSTD*, 2001.
[10] A. Henrich. A distance scan algorithm for spatial access structures. In *GIS*, 1994.
[11] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM TODS*, 24(2):265–318, 1999.
[12] Q. Hu, W.-C. Lee, and D. L. Lee. Power conservative multi-attribute queries on data broadcast. In *ICDE*, 2000.
[13] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Power efficient filtering of data an air. In *EDBT*, 1994.
[14] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Data on air: Organization and access. *IEEE TKDE*, 9(3):353–372, 1997.
[15] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed and Parallel Databases*, 15(2):117–135, 2004.
[16] I. Kamel and C. Faloutsos. On packing R-trees. In *CIKM*, 1993.
[17] W.-C. Lee and B. Zheng. DSI: A fully distributed spatial index for location-based wireless broadcast services. In *ICDCS*, 2005.
[18] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, 2004.
[19] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.
[20] K. Park, M. Song, and C.-S. Hwang. An efficient data dissemination schemes for location dependent information services. In *ICDCIT*, 2004.
[21] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.
[22] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, 2001.
[23] I. Stojmenovic. *Handbook of wireless networks and mobile computing*. John Wiley & Sons, 2002.
[24] Y. Tao and D. Papadias. Spatial queries in dynamic environments. *ACM TODS*, 28(2):101–139, 2003.
[25] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, 2002.
[26] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable processing of continuous K-nearest neighbor queries in spatio-temporal databases. In *ICDE*, 2005.
[27] J. Xu, W.-C. Lee, and X. Tang. Exponential index: A parameterized distributed indexing scheme for data on air. In *MobiSys*, 2004.
[28] J. Xu, B. Zheng, W.-C. Lee, and D. L. Lee. Energy efficient index for querying location-dependent data in mobile broadcast environments. In *ICDE*, 2003.
[29] X. Yu, K. Q. Pu, and N. Koudas. Monitoring K-nearest neighbor queries over moving objects. In *ICDE*, 2005.
[30] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD*, 2003.
[31] B. Zheng, W.-C. Lee, and D. L. Lee. Search k nearest neighbors on air. In *MDM*, 2003.
[32] B. Zheng, W.-C. Lee, and D. L. Lee. Spatial queries in wireless broadcast systems. *Wireless Networks*, 10(6):723–736, 2004.
[33] B. Zheng, W.-C. Lee, and D. L. Lee. On searching continuous k nearest neighbors in wireless data broadcast systems. *IEEE Transactions on Mobile Computing*, 6(7):748–761, 2007.
[34] B. Zheng, J. Xu, W.-C. Lee, and D. L. Lee. Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services. *VLDB J.*, 15(1):21–39, 2006.

Kyriakos Mouratidis is an Assistant Professor at the School of Information Systems, Singapore Management University. He received his BSc degree from the Aristotle University of Thessaloniki, Greece, and his PhD degree in Computer Science from the Hong Kong University of Science and Technology. His research interests include spatiotemporal databases, data stream processing, and mobile computing.

Spiridon Bakiras received his BS degree (1993) in Electrical and Computer Engineering from the National Technical University of Athens, his MS degree (1994) in Telematics from the University of Surrey, and his PhD degree (2000) in Electrical Engineering from the University of Southern California. Currently, he is an Assistant Professor in the Department of Mathematics and Computer Science at John Jay College, CUNY. Before that, he held teaching and research positions at the University of Hong Kong and the Hong Kong University of Science and Technology. His research interests include high-speed networks, peer-to-peer systems, mobile computing, and spatial databases. He is a member of the ACM and the IEEE.

Dimitris Papadias is a Professor at the Computer Science and Engineering, Hong Kong University of Science and Technology. Before joining HKUST in 1997, he worked and studied at the German National Research Center for Information Technology (GMD), the National Center for Geographic Information and Analysis (NCGIA, Maine), the University of California at San Diego, the Technical University of Vienna, the National Technical University of Athens, Queen's University (Canada), and University of Patras (Greece). He has published extensively and been involved in the program committees of all major Database Conferences, including SIGMOD, VLDB and ICDE. He is an associate editor of the VLDB Journal, the IEEE Transactions on Knowledge and Data Engineering, and on the editorial advisory board of Information Systems.