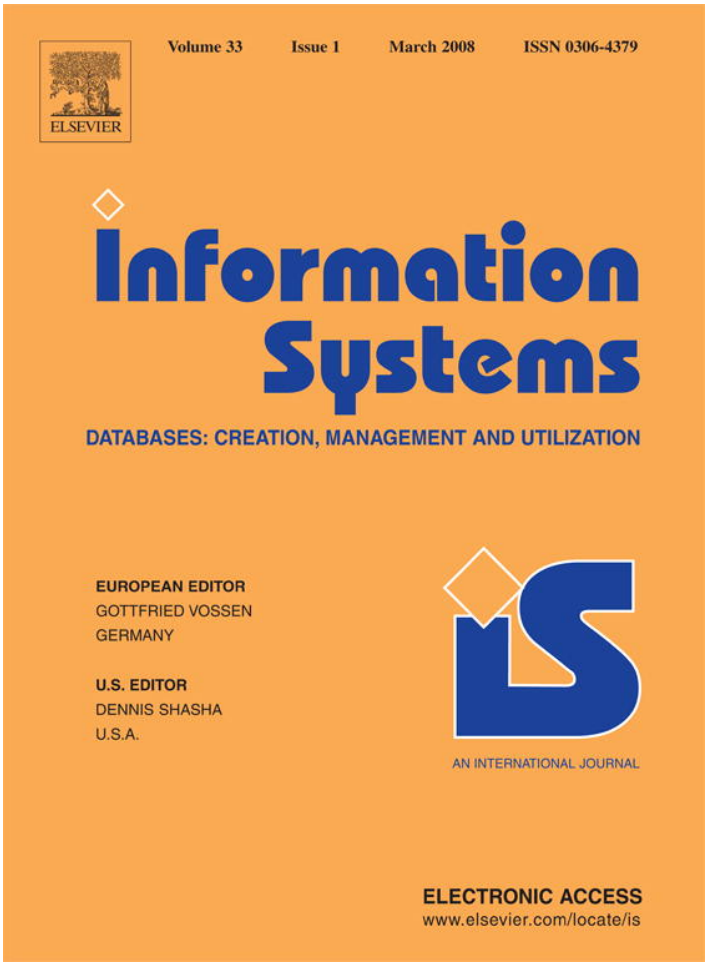
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.



This article was published in an Elsevier journal. The attached copy is furnished to the author for non-commercial research and education use, including for instruction at the author's institution, sharing with colleagues and providing to institution administration.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

Author's personal copy

Available online at www.sciencedirect.com
ScienceDirect
Information Systems 33 (2008) 18–35
www.elsevier.com/locate/infosys

Effective protocols for *k*NN search on broadcast multi-dimensional index trees

Chuan-Ming Liu*, Shu-Yu Fu

Department of Computer Science and Information Engineering, National Taipei University of Technology, Taipei 106, Taiwan

Received 9 December 2005; received in revised form 24 April 2007; accepted 30 April 2007
Recommended by K.A. Ross

Abstract

In a wireless mobile environment, data broadcasting provides an efficient way to disseminate data. Via data broadcasting, a server can provide location-based services to a large client population in a wireless environment. Among different location-based services, the *k* nearest neighbors (*k*NN) search is important and is used to find the *k* closest objects to a given point. However, the *k*NN search in a broadcast environment is particularly challenging due to the sequential access to the data on a broadcast channel. We propose efficient protocols for the *k*NN search on a broadcast *R*-tree, which is a popular multi-dimensional index tree, in a wireless broadcast environment in terms of latency and tuning time as well as memory usage. We investigate how a server schedules the broadcast and provide the corresponding *k*NN search algorithms at the mobile clients. One of our *k*NN search protocols further allows a *k*NN search to start at an arbitrary time instance and it can skip the waiting time for the beginning of a broadcast cycle, thereby reducing the latency. The experimental results validate that our mechanisms achieve the objectives.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Data dissemination; Multi-dimensional index trees; Latency; Tuning time; *k*NN search; Memory usage

1. Introduction

Advanced technologies in communications, positioning systems, and networking make it possible for mobile clients to ubiquitously access different kinds of information services, such as traffic information, stock-price information, electronic news, etc. However, the bandwidth in such a wireless mobile environment is asymmetric [1–3]. In other words, the downlink (server-to-client) bandwidth is much greater than the uplink (client-to-server) bandwidth. The conventional client–server model will hence be a poor match with the wireless mobile environment when the group of mobile clients is large due to bottlenecks of the uplink. Instead, *data broadcasting* provides an effective approach for a server to disseminate data to a large pool of clients.

Via data broadcasting, a mobile client can access the information and execute a query by tuning into the broadcast. A mobile client executing a query experiences a *latency*, which is the time elapsed between issuing and termination of the query, and the *tuning time*, which is the amount of time spent on listening to the broadcast. The latency indicates the Quality of Service (QoS) provided by the system and the tuning time represents the power consumption

*Corresponding author. Tel.: +886 2 27712171.
E-mail addresses: cmliu@csie.ntut.edu.tw (C.-M. Liu), bobby@mcse.csie.ntut.edu.tw (S.-Y. Fu).

0306-4379/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2007.04.002

Author's personal copy

of mobile clients. In general, when the broadcast consists of only data, these two cost measures are equivalent. Broadcasting data with an index structure was introduced [4,5] in order to further alleviate energy consumption. The index allows a mobile client to selectively tune into a broadcast according to the index [6–14]. This leads to a reduction in tuning time and therefore distinguishes these two cost measures. Moreover, since the memory on a mobile device is also limited, the amount of memory used when a mobile client executing a query by listening to the broadcast should be considered [6,8].

The k nearest neighbors (kNN) search is one of the important and classic problems in computer science. Given a query point p, the kNN search is to find the k closest objects to p in a multi-dimensional space. Using the kNN search, a mobile client can have a kNN query, such as "please give me 5 nearest hotels" or "please find the 3 nearest gas stations". The shaded nodes in Fig. 1(a) are the 3NN at the query point p. In [15], the authors proposed a kNN search algorithm using an R-tree [16] as the index structure. However, such an algorithm does not fully fit in a wireless broadcast environment due to the sequential access nature of the broadcast. By adapting the algorithm, Gedik et al. [17] provided a kNN search algorithm on a broadcast R-tree.

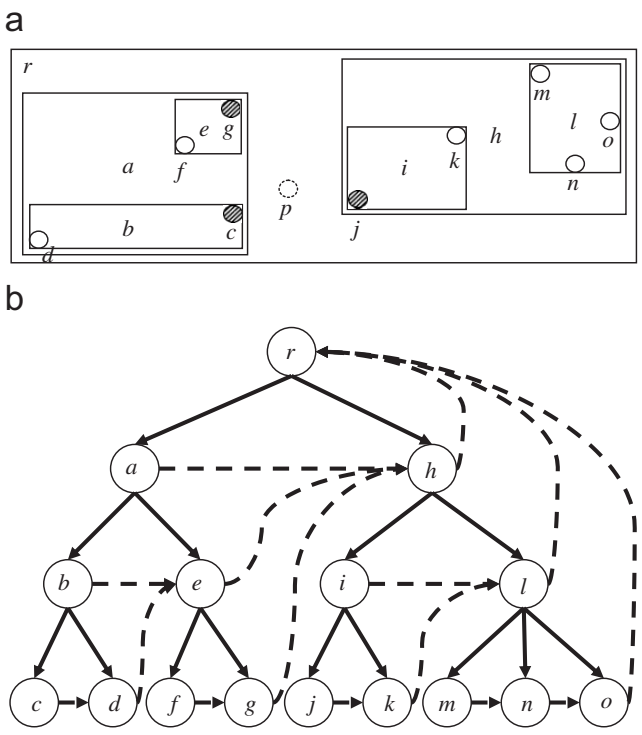


Fig. 1. The rectangles in (a) representing the MBRs of the index nodes in the 16-node R-tree in (b). (a) The 3NN (shaded nodes) of the query point p (dashed circle) among all the nodes (circles). (b) A 16-node R-tree and the dashed lines indicating the *next*-entries (defined in Section 4) of the nodes.

This paper considers the kNN search on a broadcast index tree that is an R-tree or one of its variations in wireless mobile environments. We use an R-tree as the index structure for the following reasons. First, the R-tree has been well studied and recognized as an efficient index structure [18]. Second, many types of queries have been studied on broadcast R-trees in recent years, including point query and range query [6–8,11]. This allows us to avoid designing a new index structure, for which all kinds of queries should be reconsidered. Our work considers one broadcast R-tree for different kinds of queries.

In order to simultaneously optimize the latency, tuning time, and memory usage at the client, we investigate how a server schedules the index tree for broadcast and what the query processing is with the corresponding broadcast at the client side. Almost all previous papers on the kNN search in a broadcast environment assumed that the execution of a search starts from the root (i.e., the beginning of a broadcast cycle). Such a mechanism will have the clients wait for the beginning of the broadcast to start the search, and therefore leads to a longer latency. In this paper, we further provide a mechanism to allow the execution of a kNN search to start at an arbitrary time instance and each search execution can be done within one broadcast cycle length.

In the rest of this paper, we first overview the related work and present the preliminaries in Sections 2 and 3, respectively. In Section 4, we present and discuss the broadcast schedules. Scheduling the index tree for broadcast involves determining the order by which the index nodes are sent out and adding additional entries to the index nodes for improving performance. A mobile client tunes into the broadcast and operates independently according to the broadcast schedule. An algorithm for executing a kNN search on the corresponding broadcast R-tree is proposed and analyzed in Section 5. In Section 6, by adapting our technique from Section 5, we propose a mechanism to allow a kNN search to start at any time instance, and thus reduce the latency further. The experimental evaluation is discussed in Section 7. We use R^*-trees [19] as the index trees on point data sets and compare our approach on the tuning time, latency, and memory usage with the mechanisms in [17]. Our experimental results show that our

Author's personal copy

proposed algorithms achieve a shorter latency and smaller tuning time with less memory usage on the mobile client side. Section 8 concludes this paper.

2. Related work and background

R-trees and their variations are widely used to index multi-dimensional points or rectangles [18,19]. The index node of an *R*-tree uses the *minimum bounding rectangle* (MBR) as its index which surrounds the MBRs of its children and contains the information of its children, including the MBRs of the children. The leaf node in an *R*-tree only contains the MBRs of data objects. Fig. 1 shows a 16-node *R*-tree and the corresponding MBRs of the index nodes. The *R*-tree has been extensively studied and widely used in many applications including the *k*NN search service in recent decades.

The continuous broadcast of data together with an index structure is an effective way of disseminating data in a wireless mobile environment [4–13]. The index allows mobile clients to tune into a continuous data broadcast only when data of interest and relevance is available. This leads to a reduction in tuning time. The use of an index for broadcasting is complementary to the work on dynamic broadcasts that disseminate data according to metrics like access frequencies [1,20–24] or access patterns [25–30]. We do not consider the data access frequencies since we focus on spatial data broadcast where the index is the key part for query processing.

Research on broadcasting spatial data with an index structure has attracted much attention in recent years. Based on building a Voronoi diagram, the authors of [31] and [32] proposed grid-partition index and *D*-tree index structures, respectively, which are used especially for broadcast environments. They also discussed the query processes for many types of queries on these two index structures, including nearest neighbor search. However, their approach could not be extended to the *k*NN search. In [6,8], the authors investigated the execution of queries on broadcast index trees when query execution corresponds to a partial traversal of the tree and mobile clients have limited memory. The authors of [17] provided an energy-efficient *k*NN searching approach on broadcast *R*-trees based on the conventional *k*NN search algorithm [15]. However, their work focused on minimizing the tuning time only. In our work, the proposed *k*NN search protocols minimize not only the tuning time but the latency and the amount of memory used.

3. Preliminaries and assumptions

The first NN search algorithm on an *R*-tree [15] followed the branch-and-bound algorithmic design pattern and could be generalized to *k*NN search ($k>1$). The algorithm searches the *R*-tree in a depth-first fashion and prunes the irrelevant nodes using two distance metrics, *mindist* and *minmaxdist*, defined below. Suppose a query point p is given. Then, for a node v,

- *mindist*(v) is the minimum distance from p to v's MBR; and
- *minmaxdist*(v) is the minimum distance of the maximum distances from p to each face of v's MBR.

If v is a point data, then *mindist*(v) = *minmaxdist*(v). *mindist*(v) is the minimum possible distance to a child of v and *minmaxdist*(v) is the upper bound of the distance from the query point to the closest object within the MBR of v. Using these two distances, the query process can prune the nodes which are guaranteed not to be in the *k*NN of p. This conventional *k*NN search algorithm also maintains two queues, *ItemQueue* and *ResultQueue*. The *ItemQueue* keeps the nodes which should be examined later and are sorted by *mindist*, and *ResultQueue* stores the candidate nodes to be in the *k*NN and sorted by *minmaxdist*. Let *k*thdist be the *minmaxdist* of the *k*th node in *ResultQueue* and initially ∞. If *mindist*(v)$\leqslant$*k*thdist for node v, then node v is inserted into the *ItemQueue* and will be checked later; otherwise, node v can be ignored.

Most of the existing algorithms for NN search and its variants follow this design technique used in the conventional *k*NN search algorithm. In a wireless broadcast environment, the conventional *k*NN search algorithm can be adapted to a broadcast *R*-tree, but it results in a larger tuning time due to the sequential access nature of a broadcast tree [17]. In [17], in addition to *mindist* and *minmaxdist*, the authors used *maxdist* to adapt the conventional *k*NN search algorithm to the wireless broadcast environment and provided an energy-efficient *k*NN searching approach. When a query point p is given, for a node v,

- *maxdist*(v) is the maximum distance from p to v's MBR.

In order to reduce the tuning time (i.e., the number of nodes to be checked), their approach

Author's personal copy

uses a conservative way to predict the $k$thdist and can prune the index nodes effectively in the earlier stages. Hence, the tuning time is reduced.

Our proposed kNN search algorithms follow the design technique. We use *maxdist*, *mindist*, and additional entries in the index node to prune the nodes irrelevant to the resulting kNN as well as to find the next node in broadcast to continue the search process. In the broadcast, we assume that each packet in the broadcast corresponds to a node in the R-tree for simplicity. For any index node v, the packet basically contains v's identifier and information on v's children. The information for each child v' of v includes the address of v' in the broadcast and the index for v'. The unshaded area in Fig. 2 shows the basic content of index node h in Fig. 1(b) in the broadcast. The address allows a client to tune in when the relevant node appears in the broadcast and is crucial to reducing the tuning time [6–11]. The leaf node in the broadcast contains only the node's identifier and the data content.

We say a client *explores* a node v when the client tunes into the broadcast to receive and process all the entries stored with v. Two auxiliary lists, *C-List* and *R-List*, are maintained during the search process. After exploring node v, the client knows which children of v need to be explored and *C-List* keeps all the nodes to be explored later. *R-List* stores the nodes in the current result of kNN during the search process. The functions of *C-List* and *R-List* are similar to those of *ItemQueue* and *ResultQueue* used in [17], respectively. Our proposed client query process manages the nodes in both lists according to their *maxdist* values in a non-decreasing order. To facilitate the management and processing, we use *list* where the stored nodes are ordered by *maxdist* instead of *queue* to store the nodes. In the rest of this paper, when we discuss our proposed algorithms, we use *C-List* and *R-List*; whereas, we will use *ItemQueue* and *ResultQueue* when referring to the other algorithms.

Since one packet in the broadcast corresponds to a node in the broadcast R-tree, all the cost measures

Node ID	Child's ID	Child's MBR	Child's l-entry	Next-entry
h	l	MBR	2	r
	i	MBR	3	

Fig. 2. The corresponding packet of index node h in the R-tree Fig. 1(b); the unshaded part representing the basic content; the shaded part indicating the additional entries used in this paper and defined in Section 4.

use the *node-based metric* where the number of nodes is counted [6,11]. Therefore, the tuning time counts the number of nodes explored during the execution of a kNN query, the latency measures the total number of nodes that appeared in the broadcast during the execution of a kNN search, and the memory usage takes the maximum number of nodes stored in *C-List* during the execution of a kNN search.

4. Data broadcast scheduling

There are two aspects which should be considered when designing a data broadcasting protocol in a wireless environment. One is the broadcast schedule at the server and the other is the corresponding query process at the mobile clients. In this section, we discuss the broadcast schedules at the server. A broadcast consists of a sequence of packets (nodes). The schedules based on the breadth-first traversal (BFS) can achieve a better tuning time for kNN search [17], but they result in a large memory usage at the mobile clients. Furthermore, most of the existing algorithms for different types of queries on broadcast R-trees use the broadcast schedules based on the depth-first traversal (DFS) [6–9,11,17,21,27]. We thus consider the broadcast schedules based on the DFS. Furthermore, in order to reduce the tuning time, latency, and memory usage and allow a kNN search to start in the middle of a broadcast cycle, each packet in our broadcast schedule has some additional entries.

4.1. Broadcast index trees

We basically consider two broadcast schedules, based on the DFS, and these two broadcasts differ in the order of the children of each node in the R-tree. The first broadcast schedule, pDFS, organizes the broadcast simply by the depth-first order. Such a broadcast schedule for R-trees has been used and studied in many papers [6–9,11,17,21,27]. However, such a broadcast schedule does not consider any factor that might affect the performance in terms of the tuning time, latency, and memory usage. We thus consider a variation of the pDFS, called wDFS. The wDFS broadcast schedule will rearrange an R-tree by the subtree sizes in a non-increasing order and then place the nodes in the broadcast according to the depth-first order. Using wDFS broadcast schedule, higher fan-out nodes are broadcast earlier, so *R-List* (or *ResultQueue*) is filled

Author's personal copy

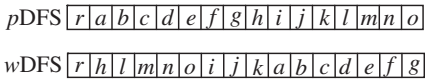


Fig. 3. The *p*DFS and *w*DFS broadcast schedules for the *R*-tree in Fig. 1(b).

quickly during the query processing. Thus, the broadcast generated by *w*DFS schedule can improve the performance of the *k*NN search process at the mobile clients, particularly in terms of the tuning time. The experimental results in Section 7 illustrate the impacts.Fig. 3 shows the broadcasts of the *R*-tree in Fig. 1 generated by the *p*DFS and *w*DFS schedules, respectively.

4.2. Additional entries

Recall that a broadcast packet corresponds to a node in the *R*-tree. A broadcast node v contains v's identifier and the information of v's children. For simplicity, we use the address of v, $addr(v)$, in the broadcast as v's identifier. The information of each child v' of v includes the address of v' in the broadcast and the MBR (index) of v'. The unshaded area in Fig. 2 indicates the basic content of a broadcast index node. In our broadcast, two kinds of additional entries are added in each node: the *l-entry* for each child and the *next-entry* of the node. The shaded area in Fig. 2 shows the additional entries of index node h in the *R*-tree in Fig. 1(b).

The *l*-entry of a child v' of v, $l(v')$, is the number of leaves in the subtree rooted at v'. Using the *l*-entry, the proposed *k*NN search algorithms can effectively prune the nodes which are irrelevant to the resulting *k*NN when exploring a node. As shown in Fig. 2, the number of leaves rooted at child node i is 2 (i.e., $l(i) = 2$). The *l*-entry plays an important role when exploring a node and we will discuss it in more details in Section 5.

In order to allow the *k*NN search to start in the middle of a broadcast cycle, we refer to the broadcast schedule in [8] and use an additional entry, *next*-entry, for each broadcast node. The *next*-entry of node v is the address of the node in the broadcast after all the descendents of v have been broadcast. The root's *next*-entry is defined as *null*. We use $next(v)$ to denote the *next*-entry of node v. The dashed lines in Fig. 1(b) indicate the next entries of the nodes in the *R*-tree. We will first introduce the algorithm which mainly uses the *l*-entries to execute the search process in Section 5. Then, we employ the *next*-entry and adapt the algorithm to provide an algorithm which allows a *k*NN search to start any time instance in Section 6.

5. Exact *k*NN search algorithm

This section introduces our exact *k*NN search algorithm, *w-disk*, on a broadcast *R*-tree. Algorithm *w-disk* follows the branch-and-bound algorithmic design pattern and starts from the beginning of the broadcast cycle (i.e., the root of the broadcast *R*-tree). We will show that algorithm *w-disk* can find the exact *k*NN efficiently and analyze the time complexity for exploring a node in a broadcast *R*-tree.

Suppose that a *k*NN query is issued at query point p. In order to prune the nodes irrelevant to the resulting *k*NN, algorithm *w-disk* uses *maxdist* to determine a circle C centered at the query point p when exploring a node. With such a circle and the *l*-entry added in the child's information in the broadcast, the algorithm can decide which node and its descendants are irrelevant to the *k*NN and exclude them for further exploring to reduce tuning time and latency. Relevant nodes will be stored in *C-List* or *R-List*. All the nodes in *C-List* and *R-List* are ordered by *maxdist*, respectively.

Consider the union U of *C-List* and *R-List*. For each node v in U, using $maxdist(v)$ as the radius, one can generate a circle C_v centered at the query point p. Suppose that there are n MBRs inside C_v and the corresponding nodes are $v_1, \ldots, v_n$. We associate node v with the value, S_v, which is the total number of leaves in the subtrees rooted at $v_1, \ldots, v_n$ and can be expressed as

$$S_v = \sum_{i=1}^{n} l(v_i), \tag{1}$$

where $l(v_i)$ is the number of leaves in the subtree rooted at v_i. Please note that, in all the nodes in U, there is at least one node u having $S_u \geqslant k$. Since all the nodes in *C-List* and *R-List* are ordered by *maxdist*, for any two nodes v and w in U with $maxdist(v) < maxdist(w)$, all the MBRs in C_v are also in C_w. Hence, by considering the nodes in U according to their *maxdist* values in a non-decreasing order, we can determine a node whose *maxdist* is minimum among all the nodes having their associated *S*-values greater than or equal to k. We refer to such a node as the *Pnode*. The *Pnode* is used to prune the nodes irrelevant to the *k*NN during the execution of the search and can be calculated

Author's personal copy

incrementally, similar to the prefix sum for a number sequence.

In the rest of this Section, we first present algorithm *w-disk* in Section 5.1. Then, an example to illustrate how the algorithm proceeds is given in Section 5.2. Last, in Section 5.3, we discuss the correctness and the time complexity of algorithm *w-disk*.

5.1. Algorithm w-disk

Initially, the Pnode is a pseudo-node and the radius of the corresponding circle is ∞. The algorithm starts with receiving the root and then explores the root. During the exploration, all the children of the root are inserted into *C-List* since all the MBRs of the children are in the corresponding circle of the Pnode. This represents that the resulting kNN might reside in the MBRs of the children. After the insertion, a new Pnode is calculated and used to prune the irrelevant nodes during the exploration on the current and next nodes. The next node to be explored is the node closest to the currently explored node in the broadcast in *C-List*. We now describe how to explore a node in general.

Suppose that node v is the next node to be explored. Algorithm *w-disk* works as follows. Assume that the Pnode determined in the previous explored node is u and the corresponding circle C_u is centered at p with radius $maxdist(u)$. Consider that node v is received from the broadcast. There are two cases for node v. First, when node v is a leaf node, we insert v into *R-List*. If *R-List* is full, we remove the node having the maximum *maxdist* among all the nodes in *R-List* including node v. Then, we consider the next node to be explored from *C-List* as before and the process continues.

The other case is that node v is an index node. For each child v' of v, the algorithm first uses C_u to decide whether v' can be ignored. If $mindist(v')$ is greater than $maxdist(u)$, then child v' can be ignored since it is impossible for the leaves in the subtree rooted at v' to be in the kNN; otherwise, insert v' into *C-List*. The algorithm then calls FINDPNODE() to find a new Pnode u' among all the nodes in the current *C-List* and *R-List*. It is not difficult to find a new Pnode u' since both of the lists are sorted by *maxdist* in a non-decreasing order. We can simply scan the nodes in both lists according to *maxdist* in a non-decreasing order until the first node u is met where $S_u \geqslant k$. With the Pnode u' and its corresponding circle $C_{u'}$, the algorithm deletes all the nodes in *C-List* and *R-List* whose MBRs are outside $C_{u'}$ and keeps all the nodes whose MBRs intersect with $C_{u'}$. Then, the algorithm extracts the next node to be explored from *C-List*. Fig. 4 shows the high-level description for exploring a node in algorithm *w-disk*.

5.2. An example

Algorithm *w-disk* starts from the beginning of the broadcast cycle (i.e., the root) and stops when *C-List* is empty (i.e., there is no next node to be explored). We now use the *R*-tree in Fig. 1 to illustrate how algorithm *w-disk* works. Consider a 3NN search at the query point p with the *p*DFS data

Algorithm *k*NN-Explore (v)
/* u is the Pnode decided when exploring the previous explored node. */
(1) **if** node v is a leaf node **then**
  (1.1) INSERT v into *R-List*
  else /* node v is an index node */
  (1.2) **for** each child v' of v **do**
    if $mindist(v') \leftarrow maxdist(u)$ **then**
      INSERT v' into *C-List*
  (1.3) u'=FINDPNODE();
  (1.4) DELETE the nodes of which $mindist > maxdist(u')$ from the both lists
(2) w=FINDNEXT(), where w is the node closest to the currently explored node in *C-List*
(3) kNN-Explore(w)
End Algorithm kNN-Explore

Algorithm FINDPNODE( )
(1) Scan the nodes in *C-List* and *R-List* according to $maxdist$ in a non-decreasing order using the way like the merge sort until the first node u whose $S_u \geq k$;
(2) **return** u
End Algorithm FINDPNODE

Fig. 4. Client algorithm for exploring a node in the kNN search.

Author's personal copy

broadcast schedule shown in Fig. 3. Algorithm *w-disk* first explores the root r. During the exploration, nodes a and h are placed into *C-List* with the order of a and h since $maxdist(a) < maxdist(h)$ (Step 1.2). Then the algorithm needs to find a new Pnode from *C-List* (Step 1.3). Node a is the new Pnode because $maxdist(a) < maxdist(h)$ and $S_a > k$ and the corresponding circle is C_a. Having the new Pnode a, the algorithm examines all the nodes in *C-List* to discard the irrelevant nodes (Step 1.4). However, the MBR of h intersects with C_a and h is thus kept in *C-List*. After exploring the root r, the next node to be explored is node a since node a is broadcast earlier than node h (Step 2), and the algorithm waits for node a in the broadcast to proceed with the search process.

When exploring node a, both of a's children are inserted into *C-List* since the *mindist* of each child of a is smaller than $maxdist(a)$. After this insertion, the order of nodes in *C-List* is e, b, and h. Recall that the nodes in *C-List* are ordered by *maxdist* in a non-decreasing order. The new Pnode b with the corresponding circle C_b can be easily found as follows. The algorithm first considers node e since $maxdist(e)$ is smallest among all the nodes in *C-List* and *R-List*. However, since $S_e = l(e) = 2 < k = 3$, the algorithm next considers node b for the new Pnode. Because the MBR of e is in C_b and C_b contains more than $k = 3$ leaves (i.e., $S_b = l(e) + l(b) = S_e + l(b) = 4 > 3$), node b is the new Pnode and the algorithm then uses C_b to decide the irrelevant nodes in both *C-List* and *R-List*. The relation among nodes e, b, h, and C_b is shown in Fig. 5. Since b is broadcast before e and h, the algorithm then extracts node b from *C-List* to be the next node to be explored.

After receiving node b from the broadcast, both children of b are inserted into *C-List* since their *mindist*s are smaller than or equal to $maxdist(b)$. Then, by considering nodes c, e, d, and h in both lists, the new Pnode is e and node d can be deleted from *C-List* since $mindist(d) > maxdist(e)$. The process then proceeds in the same way.

Recall that, when exploring a leaf node, we insert the leaf node to *R-List* instead of *C-List*. For example, node c is inserted into *R-List* after receiving it. After the insertion, *C-List* contains nodes e and h and *R-List* contains node c, respectively. The next node to be explored is e since e is broadcast earlier than the other nodes in the current *C-List*. When receiving node e, all the children of e are inserted into *C-List*. Then, *C-List* contains nodes g, f, and h and *R-List* has node c, respectively. The new Pnode is determined from nodes c, g, f, and h in both lists. As shown in Fig. 6, the new Pnode is node f. Algorithm *w-disk* then continues. When *C-List* is empty, the algorithm stops and the resulting kNN is c, g, and j. Please note that node h is not discarded after the new Pnode f is determined when exploring node e since the MBR of h intersects with C_f and some of the leaf nodes in the subtree rooted at f may be in the resulting kNN. This relates to the correctness of the algorithm and will be discussed next.

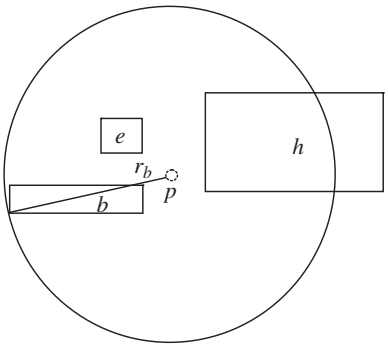


Fig. 5. Relation among node e node h, and the Pnode b with its corresponding circle C_b when algorithm *w-disk* is exploring node a on the boradcast *R*-tree in Fig. 1.

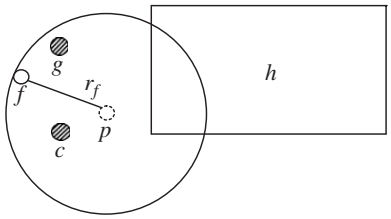


Fig. 6. Relation among the nodes in *C-List* and *R-List* when exploring node e where the new Pnode is node f since $S_f = 3$.

5.3. Correctness

Before we claim the correctness of algorithm *w-disk*, we first consider the case in Fig. 7, where node c is the Pnode. The MBRs of node b and c are inside C_c. However, the resulting kNN may contain some leaves in the MBR of node a, which intersects with C_c but is not fully inside C_c. This observation points to the reason why we only prune the nodes whose MBRs are outside the corresponding circle of the Pnode.

Author's personal copy

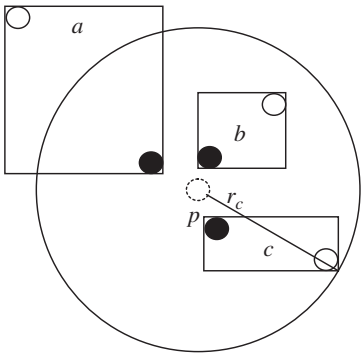


Fig. 7. A scenario where a leaf node in the resulting 3NN (bullets) at query point p resides in an MBR (the MBR a) which intersects with the circle C_c of the Pnode c but is not inside C_c.

Theorem 1. *Given a kNN search at query point p, algorithm w-disk can find the exact k nearest neighbors.*

Proof. Consider a Pnode u and its corresponding circle C_u. When using C_u to prune nodes, the nodes whose MBRs are inside C_u are retained and $S_u \geqslant k$. Furthermore, recalling steps (1.2) and (1.4) of algorithm *w-disk*, the node whose *mindist* is greater than *maxdist(u)* is pruned. In other words, the nodes whose MBRs intersect with C_u are also kept. Hence, after the pruning, C_u still includes the query result. We therefore can conclude that algorithm *w-disk* will not prune a node that contains an object that is part of the query result during the search. This completes the proof. □

5.4. Time complexity

Recall that algorithm *w-disk* uses two lists: *C-List* and *R-List*. Nodes in *C-List* and *R-List* are ordered by the *maxdist* value. The operations on these two lists include INSERT, DELETE, FINDPNODE, and FINDNEXT. Using operation INSERT, algorithm *w-disk* inserts a node into *C-List* or *R-List* at the correct position in the lists depending on whether the node is an index node or a leaf. When a node v in the lists is irrelevant to the kNN, the algorithm calls DELETE to delete v from the lists. FINDPNODE operation will return the Pnode from these two lists. FINDNEXT operation is performed in Step (2) and finds the next node to be explored in *C-List*. The time to explore a node is determined by these four operations and is defined in the following theorem.

Theorem 2. *For a broadcast R-tree having height h and degree B, it takes* $O(B \cdot \max\{k, hB\})$ *time to explore a node.*

Proof. The number of nodes in *C-List* and *R-List* when exploring a node is at most $O(\max\{k, hB\})$, where k is the given number for the kNN search. We will consider each of the four operations. Operations INSERT and DELETE clearly run in $O(\max\{k, hB\})$ time since the nodes in both lists are ordered. Because the degree of the broadcasted tree is B and the algorithm needs to insert at most B children into the lists when exploring a node, the total running time for all INSERT operations is $O(B \cdot \max\{k, hB\})$. Operation FINDNEXT can be done in $O(hB)$ time by simply scanning through *C-List*. Because the nodes in both lists are ordered and the subtree rooted at each node in both lists has at least one leaf node, operation FINDPNODE can be done by examining the nodes in these two lists incrementally one by one. In other words, one can use the result obtained from the previous scanned nodes in the lists to check whether a node is the Pnode or not. This examination of a node takes only constant time. Hence, operation FINDPNODE takes $O(k)$ time to generate the Pnode since the corresponding circle of the kth examined node during the processing must contain at least k leaf nodes. Therefore, the overall running time to explore a node is $O(B \cdot \max\{k, hB\})$. □

6. Search in the middle of a broadcast cycle

Most of the related papers have assumed that the query process always starts with the root of the broadcast *R*-tree (i.e., at the beginning of broadcast cycle). This assumption increases the latency because clients must wait for the root to appear in the broadcast after the queries are issued. In order to avoid waiting for the root in the broadcast, in this section we adapt algorithm *w-disk* and propose a kNN search mechanism, *w-disk**, which allows (1) the kNN search process to start right after a query is issued and (2) the search process to be performed within one broadcast cycle length. This mechanism leads to a shorter latency. In addition to using the l-entry to prune the irrelevant nodes, algorithm *w-disk** uses the *next*-entry defined in Section 4 to make it possible to start a search in the middle of a broadcast cycle. Algorithm *w-disk** also uses *C-List* and *R-List* to store the candidate nodes to be

Author's personal copy

explored later and the nodes in the current result of the kNN search, respectively.

6.1. Algorithm w-disk*

When a client issues a kNN search at a query point p and tunes in to the broadcast, algorithm *w-disk** starts by receiving a node f from the broadcast. Algorithm *w-disk** retains the way to explore a node used in algorithm *w-disk*. In order to allow the search process to start at an arbitrary time instance and to be performed within one broadcast cycle length, algorithm *w-disk** considers in different ways the nodes received before the root and the nodes received after the root.

We first consider a node v received before the root. Recall that *C-List* stores the candidate nodes to be explored later and algorithm *w-disk* stops when *C-List* is empty. However, when exploring a node v received before the root, that *C-List* is empty does not mean that the whole tree has been explored. If *C-List* is not empty, then the next node to be explored is the node closest to v in *C-List*. If *C-List* is empty, in order to continue the search process, we instead consider the *next*-entry of v, *next*(v), as the next node to be explored.

Following the *next*-entry to proceed with the search process allows the search process can start in the middle of a broadcast cycle. However, such a policy may explore some nodes irrelevant to the resulting kNN. In order to reduce the tuning time and accommodate the middle of the broadcast search capability into *w-disk* style kNN search, we make the following changes in *w-disk** when exploring a node u received before the root: *next*(u) will be stored into *C-List* before the process of the exploration on node u if *C-List* is empty. Note that the l-entry of *next*(u) is unknown and we set the l-entry of *next*(u) as 0 to represent the unknown. In order to keep *next*(u) in *C-List* to the very end, we further set *next*(u)'s MBR as p. Hence, *next*(u) will be extracted from *C-List* only when the subtree rooted at u has been explored.

On the other hand, when exploring the root or a node w received after the root, the emptiness of *C-List* indicates the termination of the search process. In order to have the search process be completed in one broadcast cycle, the algorithm will exclude the nodes which have been explored from consideration. To achieve this, the search process will keep the address, *addr*(f), of the first node f received when the client tunes in. Suppose node w is received after the root and about to be explored. If w is a leaf node, then w is inserted into *R-List* as in algorithm *w-disk*. If node w is an index node, algorithm *w-disk** decides whether a child w' of w can be ignored using the current Pnode and the address of the first node *addr*(f). If a child w' having an address greater than or equal to *addr*(f), child w' should have been explored before the root and therefore can be excluded from consideration.

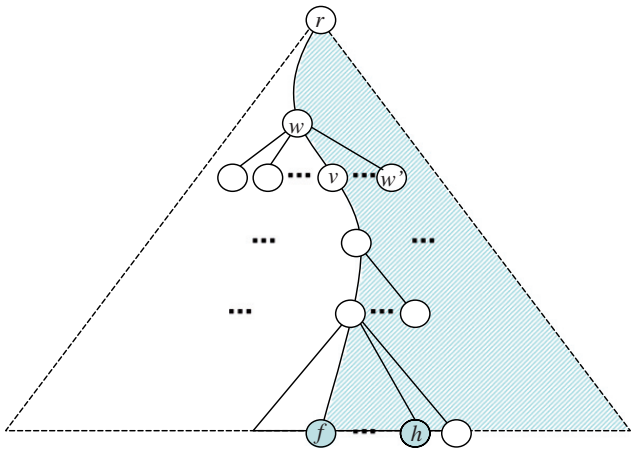


Fig. 8. Snapshot of a kNN search which starts from node f and now is exploring node w; nodes with a thick circle represent the nodes that have been explored and shaded nodes represent the nodes currently in *R-List*.

Using the above condition can easily exclude the explored nodes. However, such an exclusion complicates the calculation for a Pnode. Recall that, to decide a Pnode, the algorithm needs to count the number of leaves in the subtree rooted at each node in both *C-List* and *R-List*. However, since the algorithm starts in the middle of a broadcast, part of the subtree rooted at some node in both of the lists might have been explored. Some leaves in that subtree may therefore be counted twice when deciding a new Pnode. Consider the subtree rooted at node v in Fig. 8 when node w is being explored. The nodes on the right of the path from w to f (shaded area) had been explored before exploring node w. The leaf nodes f and h will be counted twice when calculating S_v for deciding the Pnode. From the way the Pnode is decided, we make the following observation.

Property 1. *Suppose a kNN search starts from a node f and node v is an ancestor of f. Then, node v must be considered after the root has been explored. Furthermore, suppose that nodes f and h are two leaf nodes in R-List and node v is a common ancestor of f*

Author's personal copy

and h and in C-List. If v is a Pnode, then $S_f < k$ and $S_h < k$ *as well as nodes f and h are counted twice when computing* S_v.

Recall that nodes in *C-List* and *R-List* are sorted by *maxdist* in a non-decreasing order and the Pnode is decided according to that order. Node f is considered before node v since $maxdist(f) \leqslant maxdist(v)$. To maintain consistency with algorithm *w-disk* and have a more accurate value of S_v when deciding if v is a Pnode, algorithm *w-disk** will also use the value of S_v to determine a Pnode and eliminate the duplication when computing S_v by checking each leaf node in *R-List*. If the address of a leaf node u in *R-List* is between $addr(v)$ and $next(v)$, node u is duplicated and then S_v will be adjusted by deducting the amount of the duplicated nodes. The $next(v)$ of node v can be obtained easily by the addresses of v's siblings' stored in the packet of v's parent.

However, after deducting the number of the duplicated nodes in *R-List*, the value of S_v still includes the number of the leaf nodes which had been explored but not in *R-List* (the leaf nodes with think circles in the shaded area in Fig. 8). Including such nodes when computing S_v may affect the validation of the condition, $S_v \geqslant k$, to derive a new Pnode. The pruning circle of the Pnode may be biased by the number of such leaf nodes. To ensure that algorithm *w-disk** can correctly find the exact kNN by using the condition, $S_v \geqslant k$, to decide a Pnode v in *C-List*, we claim that the condition $S_v \geqslant k$ is still valid in the following theorem.

Theorem 3. *Suppose the root has been explored. Assume that node v is in C-List and some leaf nodes in the subtree* T_v *rooted at v have been explored. If* $S_v \geqslant k$ *and v is the Pnode, then the nodes in the resulting kNN are within the disk* C_v *centered at the query point p.*

Proof. We classify the leaf nodes in T_v into three sets: L_e is the set of exclusive leaf nodes which had been excluded from the kNN; L_n is the set of non-explored leaf nodes which have not been explored; and L_r is the set of candidate leaf nodes in *R-List*. Suppose the current Pnode is w and S'_v is the S-value just before node v is considered to be the new Pnode and $S'_v < k$. Then $S_v = S'_v + l(v) - |L_r|$, where $l(v)$ is the number of leaf nodes in T_v. Hence, we can derive $S_v = S'_v + (|L_r| + |L_n| + |L_e|) - |L_r| = S'_v + |L_n| + |L_e|$. We then claim that, when deciding the Pnode v, including $|L_e|$ in S_v will not affect the result by showing that if $S_v \geqslant k$ then $|L_n| + S'_v \geqslant k$. This will guarantee that the nodes in the resulting kNN are within the disk C_v centered at the query point p. If $|L_n| + S'_v \geqslant k$, then we are done. Now, suppose $|L_n| + S'_v < k$, i.e., $|L_e| > 0$. This means that a leaf node under v, say u, was pruned at an earlier stage and the radius of C_w is less than the *mindist* of u. Since u is counted when deciding v is the new Pnode, the *maxdist* of v is larger than or equal to the *mindist* of u. However this is a contradiction to the fact that the pruning circle only shrinks as the algorithm progresses. We therefore complete the proof. □

6.2. An example

Fig. 9 shows the high-level description for exploring a node in algorithm *w-disk**. We now use the *R*-tree in Fig. 1 to illustrate how the algorithm works. Consider a kNN search at the query point p with $k = 3$, and suppose that the query starts at node g with a pDFS broadcast schedule. Algorithm *w-disk** receives the node g from the broadcast and keeps the address of g, $addr(g)$. Since g is received before the root and *C-List* is empty, algorithm *w-disk** inserts $next(g) = h$ into *C-List* with the MBR as p (Step 1). Since g is a leaf node, after the exploration, node g is placed into *R-List* (Step 2.1). Then, the next node to be explored is node $h = next(g)$ which is popped from *C-List* (Step 3). When exploring node h, which is also received before the root, since *C-List* is empty, then node $r = next(h)$ (i.e., the root) is inserted into *C-List*. While exploring node h, nodes i and l are placed into *C-List* in the order of r, i, and l since $0 = maxdist(r) < maxdist(i) < maxdist(l)$ (Step 2.2).

The algorithm then finds a new Pnode i since $S_i = l(r) + l(g) + l(i) = 3 \geqslant k$ (Step 2.3) as shown in Fig. 10. Having the Pnode, l is pruned from *C-List* because $maxdist(i) < mindist(l)$ (Step 2.4) and the next node to be explored is node i. The process continues. Before the root is received, *R-List* contains the nodes in the order of j, g, and k and *C-List* contains $r = next(h)$. The next node to be explored is therefore the root r.

Algorithm *w-disk** then receives the root r. Afterwards, the *next* entry will not be used and the search process stops when *C-List* is empty. During the exploration of r, node a is placed into *C-List*. On the contrary, node h will not be considered since h has been considered before the root is received ($addr(h) > addr(g)$), even though the MBR of node h intersects with the disk C_i of Pnode

Author's personal copy

Enhanced Algorithm k**NN-Explore***(v)
/* $begin$ is the address of the first node received from the broadcast */
/* u is the Pnode decided when exploring the previous explored node. */
(1) **if** v is broadcast before the root **and** *C-List* is empty **then**
INSERT $next(v)$ into *C-List* with l-entry=0 and p as the MBR
(2) **if** node v is a leaf node **then**
(2.1) INSERT v into *R-List*
else /* node v is an index node */
(2.2) **for** each child v' of v **do**
if v is broadcast before the root **or** the address of $v' < begin$ **then**
if $mindist(v') \leq maxdist(u)$ **then**
INSERT v' into *C-List*
(2.3) u'=FINDPNODE*();
(2.4) DELETE the nodes of which $mindist > maxdist(u')$ from the both lists
(3) w=FINDNEXT(), where w is the node closest to the currently explored node in *C-List*
(4) kNN-Explore*(w)
End Algorithm k**NN-Explore***

Algorithm FINDPNODE*()
(1) Scan the nodes in *C-List* and *R-List* by $maxdist$ in a non-decreasing order using the way like the merge sort until the first node u whose (S_u − the number of the duplicated nodes in *R-List*) $\geq k$;
(2) **return** u
End Algorithm FINDPNODE*

Fig. 9. High-level description of exploring a node in algorithm *w-disk**.

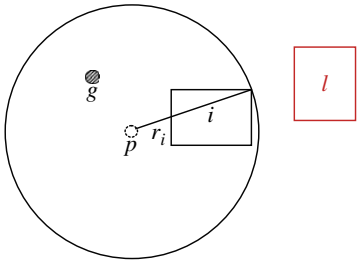


Fig. 10. Relations among the nodes in *C-List* (nodes *r*, *i*, and *l*) and *R-List* (node *g*) when exploring node *h* for a 3NN search at query point *p* which starts from node *g* on the broadcast *R*-tree in Fig. 1.

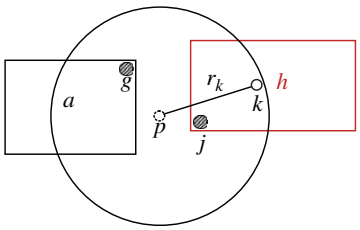


Fig. 11. When exploring the root, the MBRs of nodes *a* and *h* intersect with the circle C_i of Pnode *i* and child *a* is inserted to *C-List* but child *h* is not since *h* has been explored.

i as shown in Fig. 11. Node *k* then is the Pnode and no node will be pruned further.

Since node *a* is the node closest to the root in *C-List*, the next node to be explored is node *a* (Step 3). When exploring node *a*, both children *b* and *e* are inserted into *C-List* (Step 2.2). The nodes in *C-List* and *R-List* are in the order of *e*, *b* and *j*, *g*, *k*, respectively, according to the *maxdist*.

The algorithm then calls FINDPNODE*() to find the new Pnode (Step 2.3), node *e*. Recall that the duplicated nodes can be determined by checking the address of each leaf node in *R-List*. When calculating S_e for determining node *e* as a Pnode, node *g* is duplicated because $addr(e) \leqslant addr(g) \leqslant next(e)$. S_e is therefore modified by deducting the amount of the duplicated nodes and $S_e = 3 = l(j) + l(e)$. Since $S_e = 3 \geqslant k$, node *e* is the new Pnode. The following step (Step 2.4) will prune node *k* from *R-List* since $mindist(k) > maxdist(e)$. The algorithm then explores node *b* and the search process continues. In the end, when *C-List* is empty, the algorithm stops and the resulting *k*NN is *c*, *j*, and *g*.

6.3. Analysis

Algorithm *w-disk** basically retains the way used in algorithm *w-disk* to explore a node. However, the way of excluding the explored nodes used in algorithm *w-disk** complicates the calculation for a Pnode. This will affect the running time on exploring a node, as shown in the following theorem.

Author's personal copy

Theorem 4. *For a broadcasted R-tree having height h and degree B, algorithm w-disk* takes at most* $O(\max\{B \cdot \max\{k, hB\}, k^2\})$ *time to explore a node.*

Proof. Recalling Theorem 2, we will consider four operations, INSERT, DELETE, FINDPNODE*, and FINDNEXT. Clearly, the INSERT and DELETE operations run in $O(\max\{k, hB\})$ time since both are the same as the corresponding ones used in *w-disk*. Hence, the total running time for all INSERT operations is also $O(B \cdot \max\{k, hB\})$. Similarly, operation FINDNEXT can be done in $O(hB)$ time as shown in Theorem 2. Operation FINDPNODE* in *w-disk** is more complicated than operation FINDPNODE used in *w-disk*. For the nodes received before the root, FINDPNODE* has the same running time, $O(k)$, as FINDPNODE. However, for the nodes received after the root, FINDPNODE* needs to check the duplicated nodes in *R-List* one by one. To check each node, the time it takes is proportional to the size of *R-List*, which is k. Therefore, operation FINDPNODE* takes $O(k^2)$ time to check the duplicated nodes. By considering the running time of each operation, the overall running time for exploring a node is $O(\max\{B \cdot \max\{k, hB\}, k^2\})$ for algorithm *w-disk**. □

7. Experimental results

In this section, we present our experimental results. We first compare our kNN search algorithm ***w-disk*** with the revised conventional approach ***w-conv*** and the improved algorithm ***w-opt*** from [17]. The cost measures include the tuning time, latency, and memory usage. As mentioned in Section 2, we use the node-based metric where the number of nodes is counted in the experiments. Then, we discuss the impact on the performance resulting from different broadcast schedules. In addition to pDFS and wDFS, we also consider two broadcast schedules based on the BFS, pBFS and wBFS, in our experiments for comparing the performance of *w-disk* with *w-opt* further. Broadcast schedules pBFS and wBFS follow the same principles as pDFS and wDFS, respectively. We last discuss the performance of algorithm ***w-disk****, which allows a kNN search to start in the middle of a broadcast cycle.

The index trees used in the experiments are R^*-trees that are built on point data and generated using the codes available from [19]. We consider trees having 100,000 or 150,000 leaves and degree B between 12 and 25. For the point data set, points are generated using a uniform distribution within the unit square and correspond to the leaves. The value of k varies from 1 to 210. For each value of k, data reported are the average of 2000 different kNN searches with different query points selected uniformly.

7.1. Comparisons for different algorithms

When comparing the algorithms, we consider the performance of each algorithm in terms of tuning time, latency, and memory usage. We focus on the comparisons in tuning time and memory usage since all the three algorithms have the same latency when the query process starts at the root of the broadcast tree. Recall that all three of the compared algorithms keep the candidate nodes to be explored later. The mechanism used in these algorithms will explore the relevant nodes only. When the resulting kNN are found and retained, the algorithm will stop and ignore the nodes to be explored later. So, with the same broadcast schedule, all the compared algorithms get the same latency.

In this subsection, we use pDFS and pBFS broadcast schedules to illustrate our findings. Similar trends can be found when using wDFS and wBFS, respectively, in our experiments. The results given are for an R^*-tree having 100,000 leaves and degree 12. The tree has a height of 6 and 112,802 nodes. The kNN search starts from the beginning of a broadcast cycle.

7.1.1. Using pDFS

Fig. 12 shows a comparison of the tuning time and memory usage for algorithms, *w-disk*, *w-opt*, and *w-conv*. The x-axis reflects the different values of k, and the tuning time is influenced by the value of k. As the value of k increases, the tuning time increases for all the compared algorithms. This is because more nodes are relevant during the search processing when the value of k is larger. Among the three algorithms, algorithm *w-disk* always achieves the best tuning time. Recall that *w-opt* uses a conservative approach to predict the nodes in the *ResultQueue* such that the $k$thdist can be determined in an earlier stage, and therefore *w-opt* can prune the nodes effectively. Using the l-entries in the broadcast node, our algorithm *w-disk* can always decide the range of the kNN more accurately. Hence, *w-disk* can avoid exploring more nodes

Author's personal copy

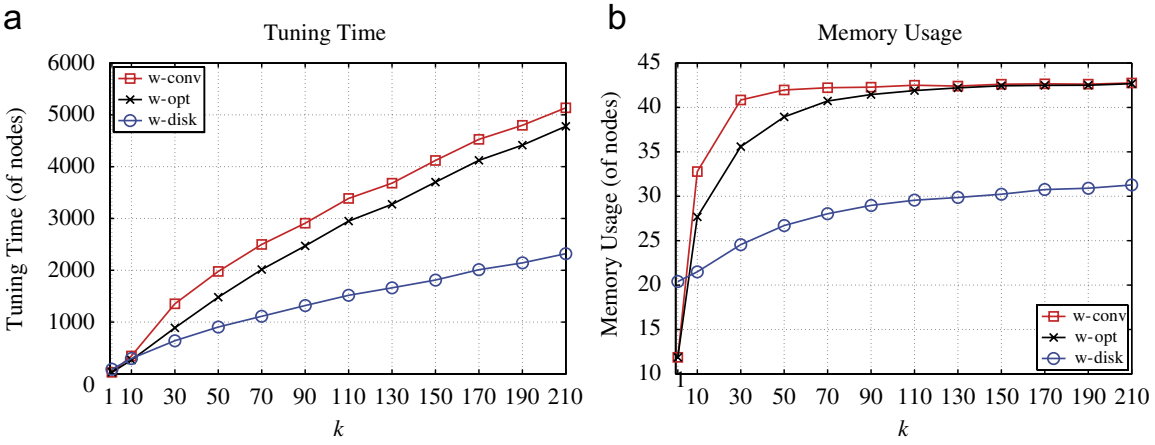


Fig. 12. Comparisons for algorithms *w-conv*, *w-opt*, and *w-disk* on an R^*-tree with degree 12 and 100,000 leaves with *p*DFS broadcast schedule.

irrelevant to the resulting *k*NN and it therefore requires a smaller tuning time. This trend becomes obvious as the value of k increases. When $k = 1$, the *k*NN search becomes an NN search and our algorithm may result in a larger tuning time since the circles used to prune nodes may contain too many leaves in the initial steps. In general, the difference in the tuning time becomes larger as k increases.

Recall that the size of *C-List* (or *ItemQueue* for *w-opt* and *w-conv*) denotes the maximum amount of memory used during the *k*NN search processing. Fig. 12(b) shows the comparison of the maximum amount of memory used. The results indicate that *w-disk* uses the least amount of memory among these three algorithms, and that *w-opt* performs better than *w-conv*. Algorithm *w-disk* uses the Pnode to prune irrelevant nodes when exploring a node. Such a Pnode is derived by considering the number of leaves in its corresponding circle, and this leads to a better approximation to the resulting *k*NN. Furthermore, *w-disk* uses the previous Pnode to delete the irrelevant children and the new Pnode to delete the irrelevant nodes in *C-List* and *R-List*. As a result, *w-disk* needs less storage to execute a *k*NN search.

7.1.2. Using pBFS

In [17], the authors concluded that using the broadcast schedule based on BFS can achieve a better tuning time than using the broadcast schedule based on DFS, although it requires more memory space. They used *w-opt* to illustrate this conclusion. With the broadcast schedule based on BFS, *w-disk* follows the same trends but can result in a smaller tuning time with less memory used. In this subsection, we discuss the performance of the algorithms using *p*BFS broadcast schedule.

Fig. 13 compares algorithms *w-conv*, *w-opt*, and *w-disk* using the same R^*-tree as in Fig. 12 with a *p*BFS broadcast schedule in terms of the tuning time and memory usage. As with the results using the *p*DFS broadcast schedule, *w-disk* outperforms the other two algorithms and the tuning time and memory usage increase as the value of k increases, respectively, for all the algorithms. We then further discuss the difference between using *p*DFS and using *p*BFS. By comparing Figs. 13(a) and 12(a), we can conclude that all the algorithms can have a smaller tuning time when using *p*BFS, respectively. Using *p*BFS, the query process explores the broadcast tree level by level. The algorithms can decide the candidate nodes more precisely and effectively on each level, therefore avoid exploring some subtrees whose roots are at higher levels. Thus, the tuning time can be reduced dramatically in comparison with *p*DFS.

As for the memory usage, we compare Figs. 13(b) and 12(b) and find that using *p*BFS will result in more memory usage than using *p*DFS. Using *p*BFS, the query process explores the broadcast tree level by level. Such an exploration will store more nodes to be explored than *p*DFS, where the broadcast tree is explored according to the depth-first principle.

7.2. Comparisons for broadcast schedules

In the previous section, we have shown and compared the performance when applying *p*DFS

Author's personal copy

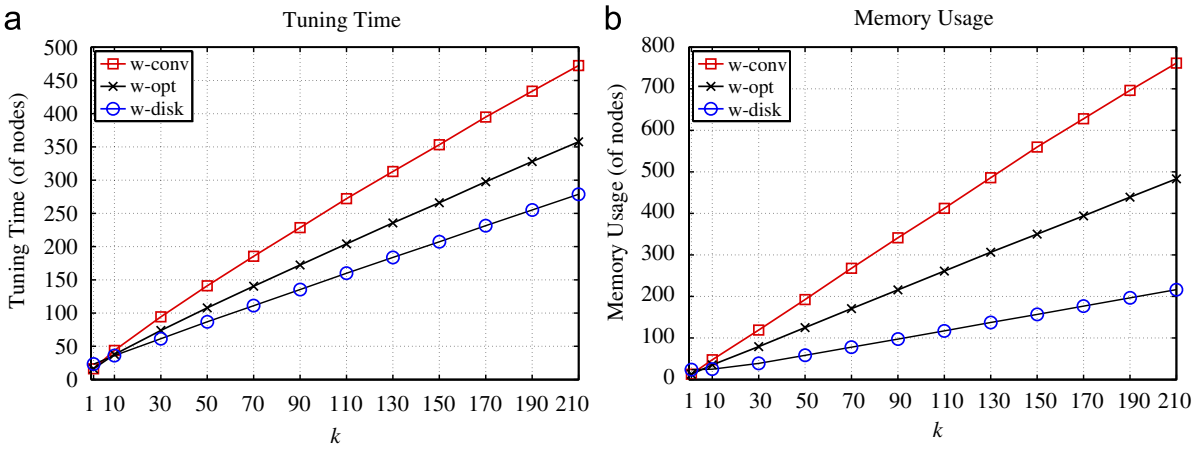


Fig. 13. Comparisons for algorithms *w-conv*, *w-opt*, and *w-disk* on the R^*-tree used in Fig. 12 with *p*BFS broadcast schedule.

and *p*BFS, respectively. Similar trends can be found when applying *w*DFS and *w*BFS broadcast schedules. In this section, we further discuss the effects resulting from different broadcast schedules and focus on the impacts of the rearrangement of the broadcast trees. We will show the results of using algorithm *w-disk* on different broadcast schedules. Using the other two algorithms will lead to the same trends on the performance. To illustrate, we consider the R^*-tree used in Fig. 12 and show the results in Fig. 14.

Fig. 14(a) presents the comparison of the tuning time using *w*DFS, *p*DFS, *w*BFS, and *p*BFS broadcast schedules, respectively, when applying algorithm *w-disk*. As mentioned in Section 7.1, using the broadcast schedules based on DFS may lead to a larger tuning time than the ones based on BFS. However, the broadcast schedules based on BFS will have almost the same tuning time due to the level-by-level node exploration.

After exploring the nodes on the same level, both *w*BFS and *p*BFS will have the same pruning circle. The only difference between these two broadcast schedules is the order to be explored among the nodes on the same level. Hence, the rearrangement of the broadcast tree does not have too much impact on the tuning time for the broadcast schedules based on BFS. As for the broadcast schedules based on DFS, broadcasting the node which has a larger subtree size first allows the mobile clients to have a better approximation for the *k*NN in an earlier stage since more MBRs can be obtained earlier. This impact results in a smaller tuning time, and so using *w*DFS yields a smaller tuning time than *p*DFS.

In general, using BFS broadcast schedules causes a longer latency than using DFS broadcast schedules and Fig. 14(b) shows this trend. The broadcast schedule based on BFS disseminates the tree level by level. The leaves are broadcast after all the index nodes have been broadcast. This will lengthen the latency in comparison with the DFS broadcast schedule. However, the rearrangement of the broadcast tree does not have impact on the latency according to Fig. 14(b) where the curves of *p*DFS and *w*DFS as well as the curves of *p*BFS and *w*BFS are tangled together, respectively. In general, the latency increase as the value of k increases. However, the structure of the broadcasted tree also impacts the latency.

For memory usage, using the broadcast schedules based on DFS can result in less memory usage than the schedules based on BFS. The results shown in Fig. 14(c) support this conclusion. From an asymptotic viewpoint, using DFS needs $O(hB)$ memory space to store the candidate nodes, where h is the height of the index tree; whereas using BFS needs $O(B^h)$ memory space. When considering the rearrangement of the broadcast trees, the conditions for DFS and BFS broadcast schedules differ. For DFS broadcast schedules, broadcasting the node which has a larger subtree size first forces the clients to store more nodes during the DFS exploration. Hence, using *p*DFS will use less memory than *w*DFS. On the other hand, when using BFS broadcast schedules, the nodes on each level are the same and broadcasting the nodes having larger subtree sizes make the approximation for the *k*NN more precisely and earlier during the search process. Hence, using *w*BFS leads to less memory usage than *p*BFS.

Author's personal copy

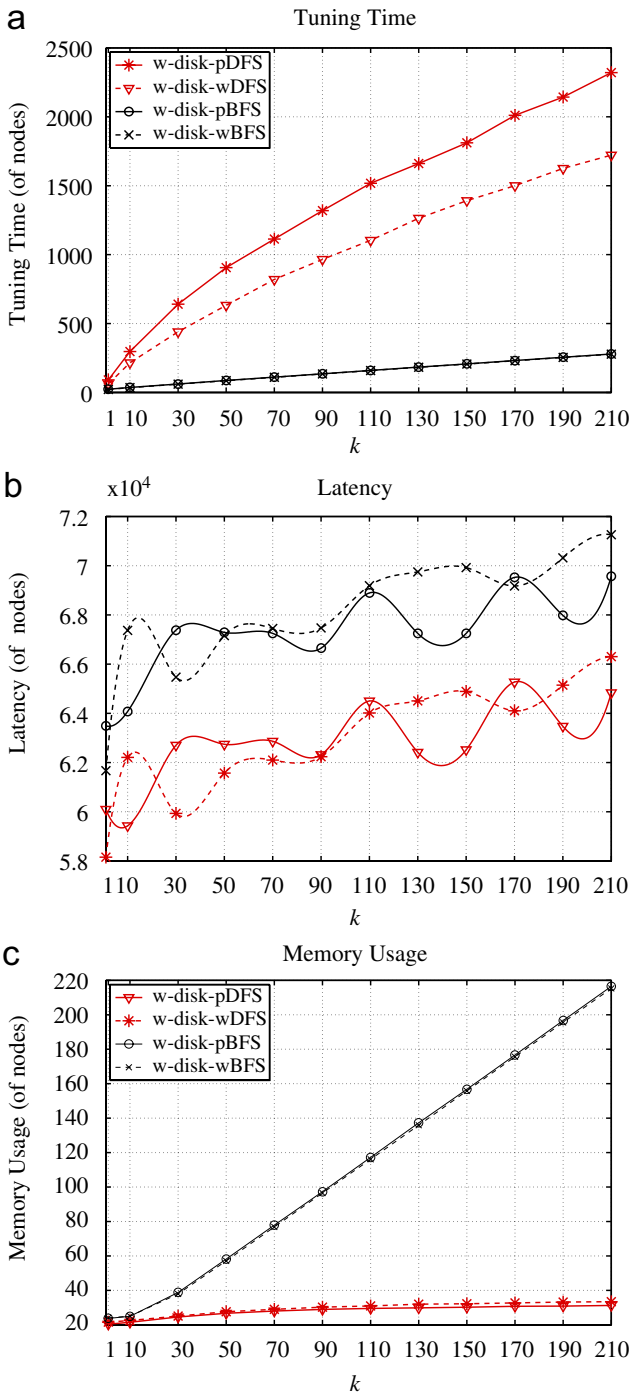


Fig. 14. Tuning time (a), latency (b), and memory usage (c) of the kNN search for different value of k using algorithm *w-disk* on an R^*-tree with different broadcast schedules.

7.3. Search at an arbitrary time instance

We now present the results of algorithm *w-disk** which allows a kNN search to start in the middle of a broadcast cycle. The starting time of each kNN search is selected randomly within a broadcast cycle. Because algorithm *w-disk* outperforms algorithms *w-conv* and *w-opt* and the latter two algorithms do not allow a kNN search to start in the middle of a cycle, we will compare algorithm *w-disk** with algorithm *w-disk* which always starts from the beginning of a broadcast cycle. As we indicate in Section 6, algorithm *w-disk* wastes time in waiting for the beginning of a broadcast cycle after a kNN search is issued, and this leads to a longer latency. We therefore first discuss the latency and then consider the tuning time. It is expected that algorithm *w-disk** will cause a larger tuning time since the algorithm may explore more irrelevant nodes. Last, we discuss the memory usage about these two algorithms. The results given in this section are for an R^*-tree having 150,000 leaves and the degree 25. All the conclusions hold in all of the experiments.

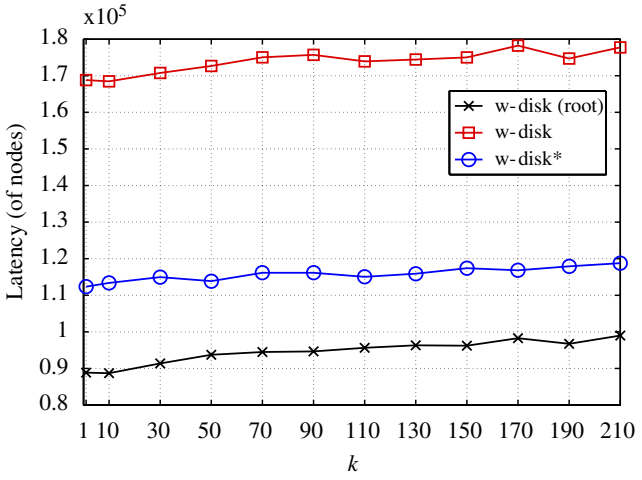


Fig. 15. Latency of the kNN search for different values of k issued at different time instances; the plot of *w-disk*(root) indicating the latency measured from the root by applying algorithm *w-disk*.

Fig. 15 shows the latency for different k values using algorithm *w-disk* and *w-disk** with different starting times. Algorithm *w-disk** leads to a shorter latency than algorithm *w-disk* since algorithm *w-disk* must wait for the beginning of the broadcast cycle. In order to see how much time is wasted in waiting for the root for algorithm *w-disk*, we further measure the latency (*w-disk*(root) in the figure) for the same kNN search which always starts from the root. The figure demonstrates that the time for waiting for the root is almost the half of the broadcast cycle length, which is 158,761. On the other hand, using algorithm *w-disk** can start the search process immediately after the search is issued and need not spend time on waiting. Furthermore, algorithm *w-disk** can guarantee that a kNN search

Author's personal copy

can be performed within one broadcast cycle length. Therefore, algorithm *w-disk** outperforms algorithm *w-disk* on latency.

Recall that algorithm *w-disk** starts in the middle of a broadcast cycle and uses the *next* entry to proceed with the search process when *C-List* is empty before the root is explored. The algorithm hence explores more irrelevant nodes than algorithm *w-disk* does. Such unnecessary explorations result in a larger tuning time. Fig. 16 presents this trend. Note that the difference on the tuning time of these two algorithms is at most 800, which is relatively small when compared to the cycle length of 158,761. However, using algorithm *w-disk** can have a much shorter latency than using algorithm *w-disk*. As shown in Fig. 15, the difference in the latency of these two algorithms is almost 60,000. As a result, using a little more tuning time, algorithm *w-disk** can achieve a much shorter latency.

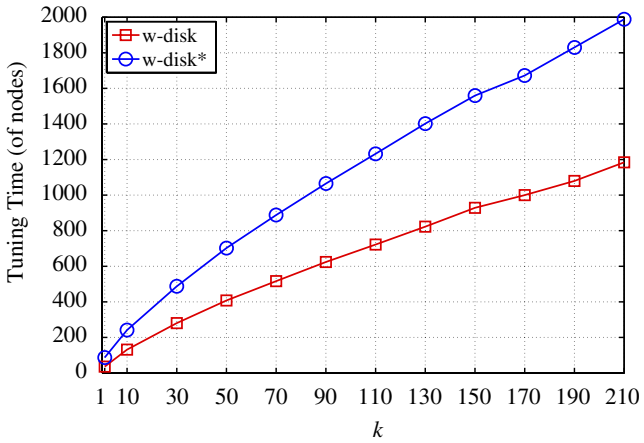


Fig. 16. Tuning time of the *k*NN search for different values of *k* issued at different time instances.

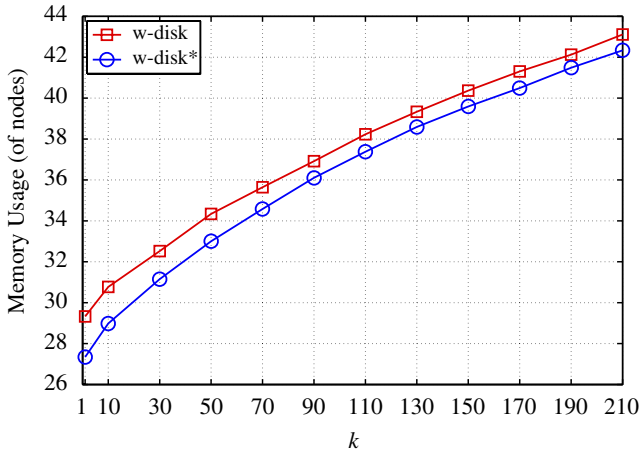


Fig. 17. Memory usage of the *k*NN search for different values of *k* issued at different time instances.

Fig. 17 shows the memory usage of two algorithms. Surprisingly, algorithm *w-disk** uses less memory when executing a *k*NN search process than does algorithm *w-disk*. Recall that algorithm *w-disk** starts the search in the middle of a cycle. This allows the search process to explore some smaller subtrees at lower levels before the root is broadcast by following the *next*-entries. Thus, the process needs fewer memory to store the candidate nodes. After the root is broadcast, *w-disk** already has a smaller pruning circle to continue the search process. As a result, *w-disk** consumes less space for storing the nodes. In contrast, algorithm *w-disk* starts the search from the root and considers the larger subtrees first, and so it needs more space to store nodes to be explored during the search.

8. Conclusions

We considered effective protocols for *k*NN search on broadcasted multi-dimensional index trees, focusing on the search algorithms executed by the clients and the broadcast schedules generated by the server. By adding additional entries, called *l*-entries, into the broadcasted R-tree nodes, our proposed *k*NN search algorithm, called *w-disk*, can find pruning circles to avoid the exploration of irrelevant nodes during the search process to achieve good performance in terms of tuning time. We also provide an algorithm to start the *k*NN search in the middle of a broadcast, without the need to wait for the root of the tree to appear in the broadcast. This is achieved by adding further entries, called *next*-entries, into the broadcast *R*-tree nodes and by adapting the search algorithms to take advantage of these entries to start the search immediately, significantly improving the result latency.

In the experiments, we compared our proposed algorithms with two other algorithms, *w-conv* and *w-opt*, on broadcast *R**-trees using different broadcast schedules. We first considered that the query process starts at the root of the broadcast tree. As the results show, all the three compared algorithms have the same latency due to the similar condition for terminating the query processing, and our proposed *k*NN search algorithm performs better than the other two in terms of tuning time and memory usage. We have further discussed the effects resulting from the different broadcast schedules. In comparison with the broadcast schedules based on DFS, the broadcast schedules based on

Author's personal copy

BFS result in smaller tuning time but need longer latency and more memory usage. The experimental results also show that the rearrangement on the broadcast trees reduces the tuning time and results in less memory usage when the broadcast schedule is based on BFS. The experiments on algorithm *w-disk**, which allows the *k*NN search in the middle of a broadcast, illustrate that using a little more tuning time *w-disk** achieves a much shorter latency with less memory usage than *w-disk*.

In the future, we plan to continue the study on efficient data broadcasting protocols for different types of queries. Our work is directed toward a unified framework which can support different types of queries on broadcast multi-dimensional index trees in wireless broadcast environments.

Acknowledgments

We thank the referees for numerous constructive comments and suggestions.

References

[1] S. Acharya, R. Alonso, M. Franklin, S. Zdonik, Broadcast disks: data management for asymmetric communication environments, in: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, 1995, pp. 199–210.

[2] S. Acharya, M. Franklin, S. Zdonik, Balancing push and pull for data broadcast, in: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, 1997, pp. 183–194.

[3] J. Jing, A.S. Helal, A. Elmagarmid, Client–server computing in mobile environments, ACM Comput. Surv. 31 (2) (1999) 117–157.

[4] T. Imielinski, S. Viswanathan, B.R. Badrinath, Energy efficient indexing on air, in: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, 1994, pp. 25–36.

[5] N. Shivakumar, S. Venkatasubramanian, Efficient indexing for broadcast based wireless systems, Mobile Networks Appl. 1 (4) (1996) 433–446.

[6] S. Hambrusch, C.-M. Liu, W.G. Aref, S. Prabhakar, Efficient query execution on broadcasted index tree structures, Data Knowledge Eng. 60 (3) (2007) 511–529.

[7] S. Hambrusch, C.-M. Liu, S. Prabhakar, Broadcasting and querying multi-dimensional index trees in a multi-channel environment, Inf. Syst. 31 (8) (2006) 870–886.

[8] S.E. Hambrusch, C.-M. Liu, W.G. Aref, S. Prabhakar, Query processing in broadcasted spatial index trees, in: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases, Springer-Verlag, 2001, pp. 502–521.

[9] T. Imieliński, S. Viswanathan, B.R. Badrinath, Data on air: organization and access, IEEE Trans. Knowledge Data Eng. 9 (3) (1997) 353–372.

[10] S. Khanna, S. Zhou, On indexed data broadcast, J. Comput. Syst. Sci. 60 (3) (2000) 575–591.

[11] C.-M. Liu, Broadcasting and blocking large data sets with an index tree, Ph.D. Thesis, Purdue University, West Lafayette, IN, 2002.

[12] J.X. Yu, K.-L. Tan, An analysis of selective tuning schemes for nonuniform broadcast, Data Knowledge Eng. 22 (3) (1997) 319–344.

[13] J. Zhang, L. Gruenwald, Optimizing data placement over wireless broadcast channel for multi-dimensional range query processing, in: Proceedings of the 5th IEEE International Conference on Mobile Data Management (MDM 2004), 2004, pp. 256–265.

[14] B. Zheng, W.-C. Lee, D.L. Lee, Spatial queries in wireless broadcast systems, Wireless Networks 10 (6) (2004) 723–736.

[15] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, in: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, 1995, pp. 71–79.

[16] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, 1984, pp. 47–57.

[17] B. Gedik, A. Singh, L. Liu, Energy efficient exact knn search in wireless broadcast environments, in: Proceedings of the 12th Annual ACM International Workshop on Geographic Information Systems, 2004, pp. 137–146.

[18] V. Gaede, O. Günther, Multidimensional access methods, ACM Comput. Surv. 30 (2) (1998) 170–231.

[19] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The *R**-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, 1990, pp. 322–331.

[20] M. Franklin, S. Zdonik, A framework for scalable dissemination-based systems, in: OOPSLA '97 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications, 1997, pp. 94–105.

[21] S. Hameed, N.H. Vaidya, Efficient algorithms for scheduling data broadcast, Wireless Networks 5 (3) (1999) 183–193.

[22] H.-P. Hung, M.-S. Chen, On exploring channel allocation in the diverse data broadcasting environment, in: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05), 2005, pp. 729–738.

[23] C.-J. Su, L. Tassiulas, V.J. Tsotras, Broadcast scheduling for information distribution, ACM/Baltzer J. Wireless Network 5 (2) (1999) 137–147.

[24] B. Zheng, X. Wu, X. Jin, D.L. Lee, Tosa: a near-optimal scheduling algorithm for multi-channel data broadcast, in: Proceedings of the 6th International Conference on Mobile Data Management, 2005, pp. 29–37.

[25] A. Bar-Noy, J. Naor, B. Schieber, Pushing dependent data in clients–providers–servers systems, Wireless Networks 9 (2003) 421–430.

[26] J.-L. Huang, M.-S. Chen, W.-C. Peng, Broadcasting dependent data for ordered queries without replication in a multi-channel mobile environment, in: Proceedings of the 19th International Conference on Data Engineering, 2003, pp. 692–694.

[27] A.R. Hurson, Y. Jiao, Data broadcasting in a mobile environment, in: Wireless Information Highway, IRM Press, 2004, pp. 96–154, (Chapter 4).

Author's personal copy

[28] C.-M. Liu, K.-F. Lin, Disseminating dependent data in wireless broadcast environments, Distributed Parallel Databases, available on-line, January 2007.
[29] K.-F. Lin, C.-M. Liu, Schedules with minimized access latency for disseminating dependent information on multiple channels, in: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC 2006), vol. 1, 2006, pp. 344–351.
[30] C.-M. Liu, K.-F. Lin, Efficient scheduling algorithms for disseminating dependent data in wireless mobile environments, in: Proceedings of the IEEE 2005 International Conference on Wireless Networks, Communications and Mobile Computing (WirelessCom 2005), 2005, pp. 375–380.
[31] B. Zheng, J. Xu, W.-C. Lee, L. Lee, Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services, VLDB J. 15 (1) (2006) 21–39.
[32] J. Xu, B. Zheng, W.-C. Lee, D.L. Lee, The *D*-tree: an index structure for planar point queries in location-based wireless services, IEEE Trans. Knowledge Data Eng. 16 (12) (2004) 1526–1542.