Volume 60, issue 3, March 2007
ISSN 0169-023X
ELSEVIER
3
DATA & KNOWLEDGE ENGINEERING
DKE

Available online at www.sciencedirect.com

ScienceDirect

Data & Knowledge Engineering 60 (2007) 511–529

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Efficient query execution on broadcasted index tree structures [☆]

Susanne Hambrusch [a,*], Chuan-Ming Liu [b], Walid G. Aref [a], Sunil Prabhakar [a]

[a] *Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, USA*
[b] *Department of Computer Science and Information Engineering, National Taipei University of Technology, Taipei 106, Taiwan*

Received 27 March 2006; accepted 30 March 2006
Available online 15 May 2006

Abstract

The continuous broadcast of data together with an index structure is an effective way of disseminating data in a wireless mobile environment. The index allows a mobile client to tune in only when relevant data is available on the channel and leads to reduced power consumption for the clients. This paper investigates the execution of queries on broadcasted index trees when query execution corresponds to a partial traversal of the tree. Queries exhibiting this behavior include range queries and nearest neighbor queries. We present two broadcast schedules for index trees and two query algorithms executed by mobile clients. Our solutions simultaneously minimize tuning time and latency and adapt to the client's available memory. Experimental results using real and synthetic data compare results for a broadcast with node repetition to one without node repetition and they show how a priority-based data management can help reduce tuning time and latency.
© 2006 Elsevier B.V. All rights reserved.

Keywords: Data dissemination; Index tree structures; Latency and tuning time; Query processing; Wireless broadcast

1. Introduction

The continuous broadcast of data together with an index structure is an effective way of disseminating data in a wireless mobile environment [4,5,9,10,12,13,18,22]. The index allows mobile clients to tune into a continuous data broadcast only when data of interest and relevance is available. This allows a client to minimize power consumption. A client executing a query experiences latency (i.e., the time elapsed between issuance and termination of the query) and tuning time (i.e., the amount of time spent actively listening to the channel).

This paper considers the execution of queries on broadcasted tree index structures. The index tree can be a spatial index structure like *R*-trees and R^*-trees, quad-trees, or *k*-*d*-trees [15,16] or another searchable tree data structure. The query executed by a mobile client can be any query whose execution results in a partial traversal

[☆] Work supported by NSF CAREER Grants IIS-0209120, IIS-9985019 and IIS-0093116, NSF Grants CCR-9988339 and CCR-0010044, and a Gift from Microsoft Corp., and the Purdue Research Foundation.
[*] Corresponding author.
E-mail addresses: seh@cs.purdue.edu (S. Hambrusch), cmliu@csie.ntut.edu.tw (C.-M. Liu), aref@cs.purdue.edu (W.G. Aref), sunil@cs.purdue.edu (S. Prabhakar).

0169-023X/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2006.03.011

of the tree. This covers range queries, point queries, and nearest neighbor queries, all of which require a partial tree traversal, not a single-path root-to-leaf traversal.

Assume that an n-node index tree is broadcast in a wireless environment. A server schedules the tree for the broadcast and the broadcast of one instance of the tree is referred to as one broadcast cycle. Scheduling the tree for broadcast involves determining the order in which index nodes are sent out, deciding which nodes (if any) of the tree appear more than once in one cycle, and adding data entries to index nodes to improve performance (in particular the tuning time). When a client starts the execution of a query, it tunes into the broadcast at an arbitrary point and starts query execution. We assume throughout that mobile clients operate independently of each other and that they can have different characteristics with respect to their available memory.

This paper focuses on the execution of queries in tree index structures when more than one path from the root to leaves may need to be traversed, resulting in a partial exploration of the broadcasted index tree. This feature distinguishes our work from related work which considers single-path root-to-leaf searches [4,5,9] and work focusing on the creation of the best index structure [4,21–23]. The use of an index for broadcasting is complementary to the work on dynamic broadcasts which disseminate data according to metrics like demand frequencies [1,3,7,17,19,20]. A preliminary and abbreviated version of our paper appeared in [8].

We present two broadcast schedules and two query processing algorithms and show that they minimize the latency and tuning time for a query and adapt to the memory available at the client. The contribution of our work lies in optimizing these parameters simultaneously. We note that not considering latency, tuning time, and memory together makes the problem easier. For example, starting a query at the beginning of a cycle and tuning in at every node during the broadcast of one cycle minimizes the latency, but not the tuning time (i.e., it achieves latency and tuning time equal to the cycle length). On the other hand, minimizing the tuning time is straightforward when (a) the index tree is broadcast in preorder, (b) the mobile client tunes in at the beginning of a broadcast cycle, and (c) the client is able to store all received addresses of nodes to be tuned into later. Starting query processing at the beginning of a cycle is restrictive and may unnecessarily increase the latency (as the client needs to wait for the beginning of the next cycle). Storing all necessary addresses of index nodes makes the amount of data to be stored dependent on the size of the tree broadcast. Our assumption is that clients with different memory sizes—which are independent of the characteristics of the tree—tune into the broadcast to execute their queries. The effective use of index structure and the available memory leads to a reduction in the tuning time and latency for a query.

We present and analyze two client algorithms for executing queries on a broadcasted index tree. The two algorithms differ on how the mobile client decides which data to delete when it can store no more additional data relevant to the query. We compare the execution of these two algorithms on a broadcast schedule in which the tree contains no node repetition. We consider the execution of the simpler of the two client algorithms on a broadcast schedule in which the tree does contain node repetition. The algorithms and the associated broadcast schedules are presented in Section 3.

Our experimental work, discussed in Section 4, compares the algorithms on 2-dimensional data sets with clients executing range queries. The data includes real and synthetic point and rectangle data sets for R^*- and R-trees. Our experimental results show that the algorithms achieve different tuning times and latencies. Overall, the algorithm using a priority-based data management of nodes to be explored and a broadcast schedule with no node repetition achieves the best performance. Our results show that being able to handle limited memory effectively results in efficient methods for starting the execution of a query in the middle of a broadcast cycle. This relationship exists since starting in the middle of a broadcast cycle corresponds to having lost relevant and previously broadcast data. Our experimental work also explores the impact of the structure of the index tree on the overall performance. We show that when the fanout of the broadcasted index tree is tailored towards the packet size and memory size of a client, performance improves considerably. However, we observed little change in performance between a packed tree (i.e., a tree created off-line) versus a tree created on-line through the random insertion of data.

2. Assumptions and preliminaries

The query processing algorithms executed by the clients do not make any assumption about the structure of the broadcasted index tree. We use B to denote the fanout of the tree index nodes (i.e., maximum number of children of a node), h to denote the height of the tree, and n to denote the total number of nodes. Data items are stored at index nodes or at the leaves of the tree. Every index node v contains information allowing a client to determine which of v's children could be roots of subtrees containing data relevant to the query. A query can terminate at any index node. For any index node v, the broadcast contains v's identifier, the data content of v, and information on v's children. We assume that for every child v' of v, the broadcasted index node for v contains the address of v' in the schedule. The address allows a client to tune in when v' appears in the broadcast. The address is crucial to reducing the tuning time and is commonly used [10,13].

The broadcast of one instance of the index tree represents one *cycle*. The length of a cycle is at least n. We assume throughout the paper that a cycle contains the nodes of the tree according to a preorder traversal, with some nodes being broadcast more than once when the schedule repeats nodes. The preorder assumption is a natural one. It implies that for a node v, the children of v appear after v in the broadcast cycle.

Our query algorithms can be used for a wide range of queries. A client can execute any query which, after retrieving an index node from the broadcast, knows which children of this index node need to be tuned into in order to generate the results of the query. The details on how an index node is handled are query-specific and are not relevant for the purpose of executing the query during the tree broadcast. We use the term "explore" to refer to the actions taken when an index node is processed in a particular query. We say a client *explores* node u when the client tunes into the broadcast to receive and process all the entries stored with u. After exploring node u, the client knows which of u's children need to be explored and it knows their addresses in the broadcast. If none of u's children needs to be explored further, u is an *unproductive node*, otherwise u is called a *productive* node.

We use two queries to illustrate the process of exploring nodes of the tree. The execution of a *range query* corresponds to a partial traversal of the tree. A node v of the index tree on which the range query is executed contains, for each child of v, information about the rectangular region containing the data items in the subtree rooted at the child. In this case, exploring node v corresponds to (i) comparing the range of each child with the query range, (ii) determining the overlap between the query range and the bounding rectangle covering all data points in the subtree, and (iii) determining which children of node v could contain data relevant to the query.

The execution of a *nearest-neighbor* (NN) query also conforms to the required assumptions. To execute an NN query, the client can use the algorithm proposed in [14] by traversing the tree in preorder. The client needs to record the minimum distance to a data object observed from the explored broadcast. For each node that is processed, the client can simply compute the values for *mindist* and *minmaxdist*, the minimum possible distance to any point in the subtree of that node, and the guaranteed maximum distance to at least one point in the node, respectively. These two distances are used by the client to make pruning decisions regarding the forthcoming children of a node being processed. Details of these distances, and the pruning algorithm are a natural extension to the algorithm presented in [14]. The algorithm can also be easily extended to process k-NN queries by recording distances to the k closest data objects observed from the explored broadcast.

The objective for a client is to minimize the tuning time and the latency. We use two metrics to measure tuning time and latency, a node-based and a packet-based metric. In the *node-based metric*, the tuning time counts the number of index nodes explored during the execution of one query and the latency counts the total number of nodes broadcast by the scheduler during the execution of the query. In the *packet-based metric*, the packet size is fixed. We count the number of packets tuned into, where an index node consists of a number of packets. In either metric, tuning time is minimized when the client is able to store addresses of nodes already identified for exploration. Not being able to store all these addresses due to memory constraints results in an increase in the number of unproductive nodes explored and thus increases the tuning time. When comparing index trees with the same fanout, we use the node-based metric. Index trees with different fanout values are compared using the packet-based metric.

3. Algorithms for mobile clients with limited memory

In this section we describe two broadcast schedules for index trees and two client algorithms for executing queries during the broadcast. A client can execute either one of two algorithms: **NextQ** or **DoublePQ**. The two broadcast schedules differ on whether node repetition is allowed and thus have different cycle lengths.

Let m denote a client's memory size. We assume that each "unit" of client memory can store the address of a node and optionally a small, constant number of other entries. Throughout, clients can have different memory sizes, and the query algorithm a client executes will adjust to its memory size. We assume that a client cannot store all addresses of nodes to be explored later. In the worst case, a broadcast based on a preorder traversal needs to store hB node storage units. Hence, we assume $m < hB$.

In the broadcast schedule without node repetition, the index tree is broadcast in preorder. When node v is broadcast, the schedule contains, (i) for each child of v, the address of the child in the broadcast, and (ii) an entry *next*(v) which is the address of the first node in the broadcast which is not a descendent of v. Next(v) can be a sibling of v or a higher-degree cousin. Next-entries are natural and have been used for single-path queries in [12,18]. Fig. 1 shows an example of a 48-node tree used throughout this paper. The tree happens to be balanced and has a fanout of $B = 3$. The next-entries are indicated in the tree by dashed arrows. The broadcast schedule corresponds to a list of the 48 nodes in the order from 0 to 47.

Sections 3.1 and 3.2 describe Algorithms NextQ and DoublePQ operating on the broadcast schedule described above. The two algorithms differ on how a client decides what entries to delete when it runs out of memory. The metric used by DoublePQ requires different data management at the client, but results in the exploration of fewer unproductive index nodes after entries are lost. Clients listening to the broadcast can execute either one of the two algorithms.

Section 3.3 describes the broadcast schedule with node repetition and sketches how a client executes algorithm NextQ on this schedule. For both broadcast schedules, the index tree used is generated by the server without knowledge of the amount of memory available at clients. Each client algorithm is tailored towards the client's available memory. For the schedule with node repetition, the amount of node repetition is determined by the server with the goal of broadcasting more frequently needed index nodes more often. Section 3.4 concludes with an asymptotic worst-case comparison of the latency, the cost of exploring a node, and the number of unproductive nodes explored (which captures the tuning time).

3.1. Algorithm NextQ: using next entries

Throughout this section assume that the index tree is broadcast in preorder with next-entries and no node repetition. A mobile client starts the execution of a query at an arbitrary point during the broadcast cycle. Assume a client tunes in at next node in the broadcast, say node v. The client explores node v. Details of the exploration depend on the query executed and are not the focus of this work (we refer to Section 2 for a discussion on two query types). In algorithm NextQ, the client maintains a deque Q [6]. The deque is empty at the start of a query and contains, at any point, at most m entries. The top of the deque experiences insertion and deletions, while the bottom experiences only deletions.

Assume node v has been explored. If v is a productive node, let $v_1, \ldots, v_k$ be the children of v to be explored later. Nodes $v_1, \ldots, v_k$, along with their address in the broadcast, are inserted at the top of Q in reverse order of

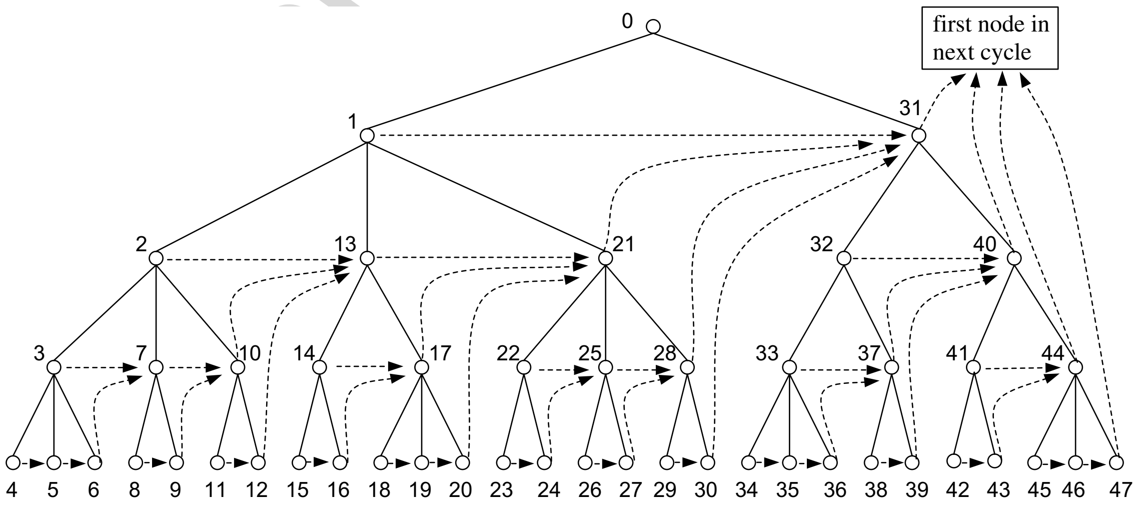


Fig. 1. A tree with $n = 48$, $B = 3$, and next-entries (dashed arrows).

their address (i.e, appearance) in the broadcast. Should Q become full during the addition of these children, nodes are deleted at the bottom of Q; i.e., using FIFO management. The addresses of nodes to be explored and deleted from Q will be "restored" by using the *next*-entries of nodes, generally at the cost of exploring more nodes. While the bottom of deque Q experiences deletions when space is needed, the top of Q experiences a delete operation right before the exploration of a node. Fig. 2 gives a high-level description of Algorithm NextQ.

Observe that after the exploration of a productive node v, the next node to be explored is always found in Q (since Q will not be empty). After the exploration of an unproductive node, the next node to be tuned into is either found in Q or, if Q is empty, it is *next*(v).

Fig. 3 shows an example of NextQ for a client with $m = 3$ and a query that needs to retrieve data stored in leaf nodes 4, 5, 18, 20, 34, and 35 of the broadcasted tree. All nodes explored during the query execution are marked as filled nodes. The deque status shown underneath the tree shows Q only for nodes explored during query execution. The status of the deque Q is shown after the node appearing on top of the queue has been explored. For the sake of simplicity, we first assume that node 0 is the first node tuned into.

The exploration of node 0 results in two nodes being added to Q: nodes 1 and 31. After the exploration of index node 1, deque Q contains three nodes, 2, 13, and 31. After node 2 is deleted from Q and explored: nodes 3, 7, and 10 are to be inserted to Q. Since $m = 3$ and Q already contains two entries (13 and 31), nodes 13 and 31 are deleted and nodes 3, 7, and 10 are added to Q. This is the status of Q before node 3 is explored. Exploring node 3 results in the deletion of node 10, and Q now contains nodes 4, 5, and 7. Node 4 is a leaf and no nodes are added. After node 5 is explored, Q contains only node 7. Node 7 is found to be an unproductive node. Since Q is empty at that point, node next(7) = 10 is tuned in and explored. Node 10 is found to be an unproductive node and next(10) = 13 is explored. Node 13 has one productive child, node 17, which is added to Q. In total, *next*-entries are used four times during the execution of the query, as indicated in Fig. 3 through dashed arrows. The query terminates after node 40 is explored.

If query processing would start at a different node, the exploration of the nodes would follow the same pattern. For example, if query processing were to start at node 21, Q would be empty, node 21 would be found unproductive, and the next node explored would be node 31. The same number of nodes would be explored and latency and tuning time would be as before. If query processing were to start at node 23, algorithm NextQ

```
Algorithm NextQ($v$)
(1)   explore node $v$ using query dependent processing;
(2)   if node $v$ is a leaf node then
              determine the relevance of $v$'s data to the query
      else if node $v$ is a productive node then
              • let $v_1, v_2, \ldots, v_k$ be the children of $v$ to be explored;
              • insert children $v_1, v_2, \ldots, v_k$ at the top of deque $Q$ in reverse order of their arrival
              in the broadcast schedule;
              • should $Q$ become full, delete nodes from the bottom of $Q$ until there is space
              for the $k$ new nodes to be inserted;
      endif
(3)   $u$=FindNextNodeQ($v$);
(4)   NextQ($u$)
End Algorithm NextQ

Algorithm FindNextNodeQ($v$)
if queue Q is empty then
      return $next(v)$
else
      return the node returned when deleting the most recently added node from $Q$
endif
End Algorithm FindNextNodeQ
```

Fig. 2. Client algorithm NextQ for query execution during a tree broadcast with *next*-entries.

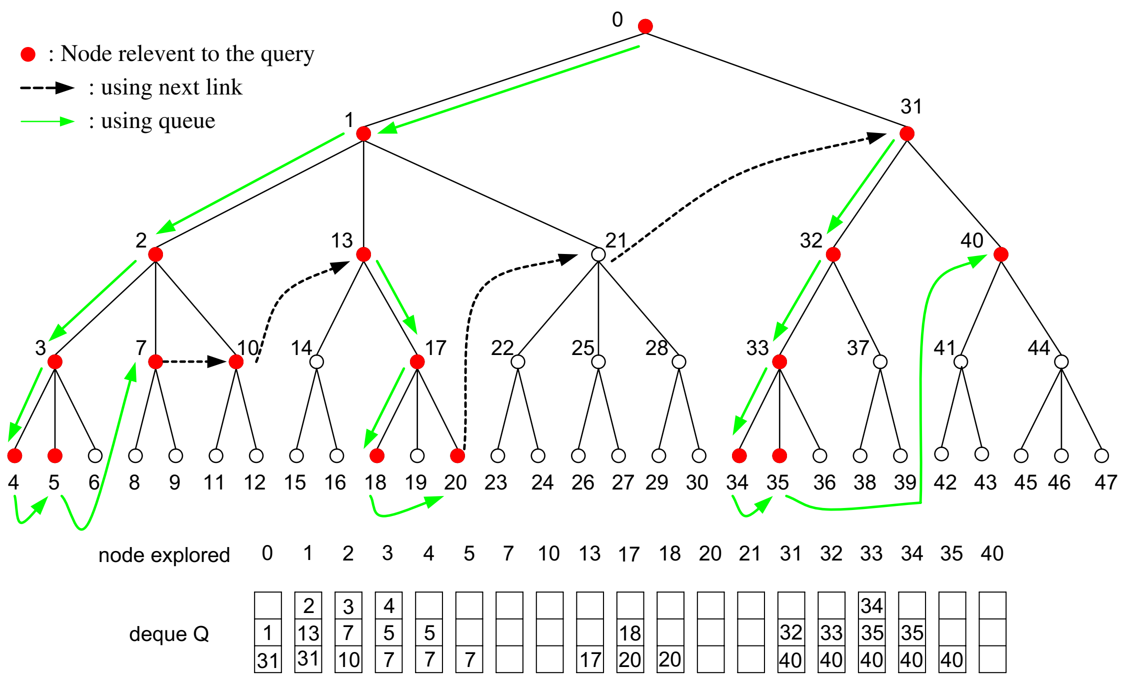


Fig. 3. Processing a query issued at the begin of the cycle and using *next*-entries, $m = 3$. The entries below the tree show deque Q after the exploration of the corresponding node.

would explore nodes 23, 24, 25, and 28, before reaching productive node 31 (see Fig. 1 for all next-links of the tree).

Hence, when a mobile client tunes into the broadcast at an arbitrary point during the cycle, query execution starts with an empty deque and proceeds just as though entries have been lost due to space limitations. To complete the query within one cycle, the client needs to remember the address of the first node obtained after tuning in. Let *f* be this node. Should node *f* be received again in the next broadcast cycle, the query terminates (if it did not already terminate). In the case when node *f* is not tuned into in the next broadcast cycle, the address of *f* is used to determine that one cycle has passed and the query can terminate. The latency experienced is thus at most one cycle length.

3.2. Algorithm DoublePQ: using an exploration-based cost function

This section describes Algorithm DoublePQ, a client algorithm processing queries by employing different data management from NextQ. Assume again that the tree is broadcast in preorder with next-entries and no node repetition. Recall that in NextQ, a client uses a deque of size at most m to hold the most recent nodes identified as index nodes to be explored. When a client runs out of memory, NextQ deletes nodes from the bottom of the deque. In contrast, Algorithm DoublePQ deletes nodes based on a priority. For a node u, let $cost(u)$ be the number of children of u not yet broadcast in the on-going cycle that do *not* need to be explored by the query. For the tree shown in Fig. 3, node 1 has $cost(1) = 1$ since node 21 does not need to be explored. Quantity $cost(u)$ measures the additional unproductive nodes a client needs to tune into in case information is lost.

We discuss Algorithm DoublePQ using the implementation underlying our experimental work described in Section 4. This implementation uses a priority queue PQ and an ordered linked list structure L. List L contains productive nodes already explored, along with a child-list for every node. The child-list contains the children to be explored, in the order of the broadcast. In some sense, L is a "list of lists." Note that when we refer to the "nodes in L", we do not mean the nodes in the child-list of a node, but the nodes themselves. Every node in L has an entry in priority queue PQ. Nodes in PQ are arranged according to the *cost*-entries, along with pointers between PQ and L. Note that if node v is in L with k children in its child-list, we consider node v to be using $k + 1$ memory locations. For DoublePQ to be meaningful, we assume $m \geqslant B$; i.e., there is at least enough

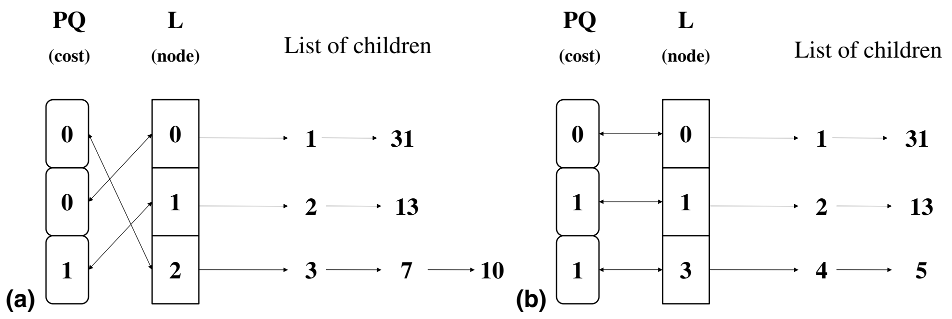


Fig. 4. Illustrating the use of data structures L and PQ in DoublePQ for the tree of Fig. 1.

memory to hold one node and its children. Fig. 4(a) shows the data structures after nodes 0, 1, and 2 have been explored (we assume $m \geqslant 7$).

Let u be a productive node. During the exploration of u, cost(u) is computed. The cost-entry is computed on the fly when u is explored. This incurs no additional costs since for every child we need to determine whether it needs to be explored. Node u is added to L and its children to be explored form the child-list of u in L. Node u is inserted with $cost(u)$ into priority queue PQ. Deletions in L occur in a number of ways: (i) when a node has an empty child-list, the node is deleted from L, (ii) a node in a child-list is deleted at some point after it has been explored, (iii) when the client needs more space, a node with minimum cost-entry is deleted (along with its child-list). At any point in time, the child-list of a node in L contains at least one node. We point out that as nodes are deleted from the child-list of a node u, the value of $cost(u)$ changes. Cost-entries can be maintained dynamically by maintaining, for every node u' in the child-list of u, a count of the number of siblings not to be explored between u' and the next node in u's child-list. As $cost(u)$ changes, updates are made on the priority queue structure.

Fig. 5 gives a high-level description of Algorithm DoublePQ. When a mobile client tunes in and receives node v, it explores node v and computes $cost(v)$ (step (1)). When v is productive with children $v_1, \ldots, v_k$, we insert node v with key $cost(v)$ in queue PQ. We create an entry for node v in list L and place the k children of v into its child-list. When there is not enough memory to store node v and its k children, we delete (before the insertion) nodes based on $cost$-entries: The node with minimum cost-value is deleted from PQ, along with this node's entry and child-list in L.

Algorithm DoublePQ differs from NextQ on how the next node to be explored is determined. For the case when v is a productive node, v_1 is the next node to tune into and explore. In the other situations, FindNextNodePQ(v) is invoked. Since nodes in list L are not deleted according to the FIFO management, the node returned is either $next(v)$, a node in a child-list, or the $next$ value of a node. FindNextNodePQ determines this node.

First, if L is empty, we tune in at $next(v)$ (case 1). Let m be the node most recently added to L. FindNextNodePQ deletes nodes in m's child-list that were broadcast earlier. Any such deleted node corresponds to a productive node deleted from L due to space considerations. If the child-list ends up empty, we continue with node m. Otherwise, let w_1 be the first child in m's child list. If $w_1 = v$, then m is v's parent. If the child-list of m contains only node $v = w_1$, then no sibling of v remains to be explored (case 2). We delete m from L and make a recursive call using m. If the child-list of m contains a second node w_2, we tune in at w_2 and delete w_1 from the child-list (case 3). Note that this allows us to skip siblings between v and w_2 that do not need to be explored. Finally, when $w_1 \neq v$, v's parent was removed from L and we tune in at $next(v)$ (case 4). Tuning in at w_1 would miss node $next(v)$ and lead to incorrect query execution.

We use the tree given in Fig. 3 to demonstrate DoublePQ and FindNextNodePQ. Fig. 4(a) shows L and PQ after node 2 has been explored. Nodes 0, 1, and 2 are in list L, along with their child-lists. Node 3 and its two productive children are to be added. Since there is not enough space, a node with minimum PQ-value is deleted. Node 2 is selected for deletion and node 3 is added. The result of this update is shown in Fig. 4(b). Node 4 is not a productive node; we have FindNextNodePQ(4) = 5 and node 4 is deleted from the child-list of node 3. Node 5 is also not productive and FindNextNodePQ(5) invokes FindNextNodePQ(3) (case 2). Node 1 is the most recently added node. Node 2 is deleted from the child-list of node 1 and node 13 is now the first node in the child-list of 2. Case 4 applies and we continue with node $next(3)$, i.e. node 7.

```
Algorithm DoublePQ($v$)
(1) explore node $v$ using query dependent processing and compute $cost(v)$;
(2) if node $v$ is a data node then
        determine the relevance of $v$'s data to the query
        $u$ = FindNextNodePQ($v$)
    else
        if node $v$ is a productive node then
            let $v_1, v_2, \ldots, v_k$ be the children to be explored, arranged in the order they
            appear in the broadcast:
            (2.1) insert $v$ into $PQ$ with $cost(v)$;
            (2.2) while there is not enough space for $v$ and its $k$ children do
                    (a) determine node $u$ in $PQ$ having minimum cost (break ties arbitrarily);
                    (b) delete all entries of node $u$ and $u$'s child-list from $L$ and $PQ$
                    endwhile
            (2.3) insert $v$ and its $k$ children into $L$
            (2.4) $u = v_1$
        else
            $u$ = FindNextNodePQ($v$)
        endif
    endif
(3) DoublePQ($u$)
End Algorithm DoublePQ

Algorithm FindNextNodePQ($v$)
(1) if list $L$ is empty then return $next(v)$ endif
let $m$ be the node most recently added to $L$;
scan the child-list of $m$ and delete nodes already broadcast until the scan reaches
a node not yet broadcast, node $v$, or the list is empty;
if the child-list of $m$ is empty then
    FindNextNodePQ($m$);
else
    let $w_1$ be the first node in the child-list after the scan;
    (2) if $w_1 = v$ and $w_1$ is the only node in the child-list of $m$ then
            delete all entries of node $m$ and $m$'s child-list from $L$ and $PQ$;
            FindNextNodePQ($m$);
        endif
    (3) if $w_1 = v$ and the child-list of $m$ contains at least 2 nodes then
            let $w_2$ be the second node in the child-list of $m$;
            delete node $w_1$ from the child-list of $m$;
            return $w_2$;
        endif
    (4) if $w_1 \neq v$ then return $next(v)$ endif
endif
End Algorithm FindNextNodePQ
```

Fig. 5. Client algorithm DoublePQ for query execution using next-entries and a priority-based rule for managing memory.

When a client tunes into the broadcast cycle at some arbitrary point, the algorithm is initiated with empty data structures. Like the previous algorithm, a client needs to remember the address of the first node seen in order to terminate the query with a latency not exceeding the length of one cycle. Starting the algorithm in an on-going cycle reduces the benefit gained by the exploration-based cost metric.

3.3. Operating on a broadcast with node repetition

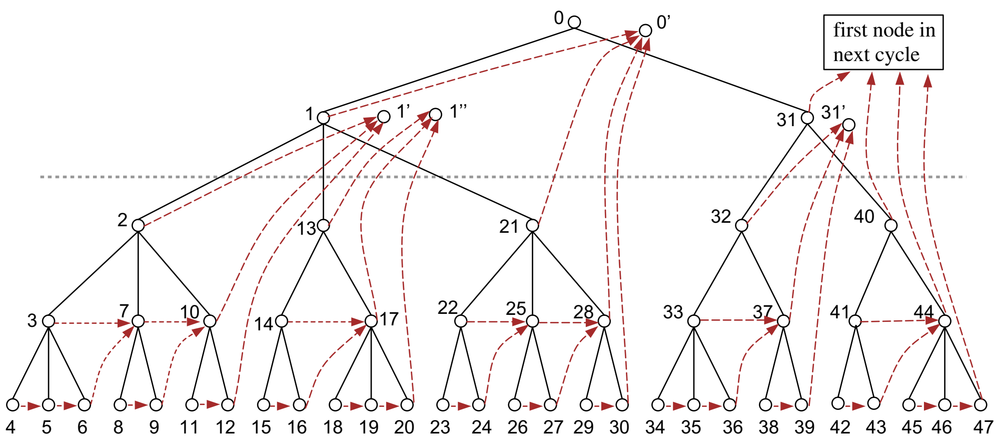
In this section we consider query execution on a broadcast schedule in which selected nodes are repeated within a cycle. Repetition can improve broadcast performance, as demonstrated in [10,13]. Repetition of nodes increases the cycle length and can thus increase the latency. However, our experimental work shows that the use of repetition in the broadcast schedule can result in clients experiencing smaller tuning time compared the schedule without repetition.

Let l be a parameter chosen by the server. To be meaningful for a tree of height h, we have $1 \leqslant l < h$. Parameter l determines the amount of node replication in the schedule. Note that a tree of height h consists of $h+1$ levels, with the root being on level 0. For every node u on level l of T, the server generates the preorder traversal of the subtree rooted at node u. This is the schedule for the subtree rooted at node u on level l. Next, consider the remaining nodes by decreasing level numbers, starting with level $l-1$. Let v be a node on level j with children $v_1, \ldots, v_k$, $0 \leqslant j \leqslant l-1$. The schedules for the subtrees rooted at nodes $v_1, \ldots, v_k$ on level $j+1$ have already been generated. Let S_i be the schedule of the subtree rooted at child v_i, $1 \leqslant i \leqslant k$. Then, the schedule for the subtree rooted at node v is $vS_1vS_2v \ldots vS_k$. The schedule can be viewed as a combination of an Euler tour on the top l levels and preorder traversals on the remaining subtrees [6]. The schedule is similar to the distributed indexing method used for single root-to-leaf searches of broadcasted data files [10].

In the schedule generated, nodes on levels $0, 1, \ldots, l-1$ are replicated. The number of times a node is repeated is equal to the number of its children. Since the nodes on the first l levels are repeated, the total number of repeated nodes in the broadcast is equal to the number of nodes on the first $l+1$ levels of the tree minus 1. Fig. 6 shows a schedule for $l = 2$. This trees has three nodes on the first two levels, and in the schedule these three nodes appear as seven repeated nodes: 0, 0′, 1, 1′, 1″, 31, 31′.

Every node in the broadcast schedule has one additional entry, which we again call *next*. Entry *next*(v) is the address of the node in the broadcast after all descendents of v have been broadcast. Observe that *next*(v) can now be the address of a repeated copy of a node. Let u be a node on level l of T. The path from u to the rightmost leaf in its subtree is called the *rightmost path*. For all nodes on a rightmost path, the next-entries point to repeated nodes. For all other nodes, the *next*-entries point to non-repeated nodes. For example, in Fig. 6, *next*(13) = 1″ which is the address of the third copy of node 1 and *next*(3) = 7 which is the address of the only copy of node 7.

We refer to a client executing algorithm NextQ on the schedule with a node repetition of l as executing Algorithm **Repeat-*l***. In Repeat-l, nodes to be explored are, as in NextQ, inserted into deque Q according to their arrival in the broadcast cycle and are deleted when space is needed according to the FIFO rule. When a node u has been explored and Q is empty, the client continues with exploring *next*(u). Tuning in at repeated nodes allows a client to restore lost entries and leads to a reduction in the tuning time.



Broadcast schedule:

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 1' | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 1" | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30

0' | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 31' | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47

Fig. 6. Tree T of height 4 and the broadcast schedule with node repetition for $l = 2$; pointers in the tree indicate *next*-entries.

Note that l, which determines the amount of replication in the schedule, is independent of the client's memory. Clearly, a client makes the best use of the schedule with node repetition when it never has to delete a node to be explored which is not replicated. Losing a non-replicated node increases the tuning time and the algorithm's tuning time converges to that of Algorithm NextQ. The loss of a replicated node increases the tuning time only slightly. However, re-exploring a replicated node increases the computation time experienced by the client. When T is a complete B-ary tree, a client does not lose a non-replicated node when $l = h - \lfloor \frac{m}{B} \rfloor - 1$.

The data management used by algorithm DoublePQ can also be applied to the broadcast schedule with repetition. Such a client algorithm would need a modified cost metric for replicated nodes as their loss should have a lower cost that of a non-replicated node. The experimental work does not consider this implementation. It focuses on whether the two forms of data management at a client lead to a difference in performance and whether replication does reduce tuning time.

3.4. Asymptotic comparisons of NextQ, DoublePQ, and Repeat-l

In this section we give a brief asymptotic comparison of the costs arising for a client during the execution of a query. Assume, as before, that T is an n-node index tree of height h, with every index node having a maximum fanout of B. The bounds are given for the packet-based metric (i.e., one packet contains all the data associated with an index node).

In the packet-based metric, the cycle length of the broadcast schedule without repetition is n. Since we assume that every index node has at least two children, the increase in the cycle length for the schedule with node repetition equals the number of nodes on level $l + 1$, which is $O(B^l)$. For all queries, the latency is bounded by the cycle length. In the best case, the query corresponds to a single path, and latency and tuning time equal the height of the tree. Two other relevant metrics are the computation costs a client experiences when exploring a node and the maximum number of unproductive nodes explored during the execution of a query. The cost of exploring a node is linear in the number of children for Algorithms NextQ and Repeat-l. For Algorithm DoublePQ, the cost of maintaining the priority queue needs to added, as shown in Fig. 7. The number of unproductive nodes explored depends on the structure of the tree, the data retrieved by the query, the start of the query, and the amount of memory available at a client.

Fig. 8 shows a worst-case scenario on the number of unproductive nodes explored for Algorithm NextQ (the examples for the other two algorithms are similar). Assume ϵ is chosen such that exploring subtrees of

Algorithm	**Cycle Length**	**Latency**	**Client's computation time exploring a node**	**Number of unproductive nodes**
NextQ	n	$O(n)$	$O(B)$	$\Theta(B^{h-\frac{m}{B}})$
DoublePQ	n	$O(n)$	$O(\max\{B, \log m\})$	$\Theta(B^{h-\frac{m}{B}})$
Repeat-l	$n + O(B^l)$	$O(n + B^l)$	$O(B)$	$\Theta(B^{h-\frac{m}{B}} + B^l)$

Fig. 7. Asymptotic worst-case bounds for a client executing Algorithms NextQ, DoublePQ, and Repeat-l; m denotes the client's memory size.

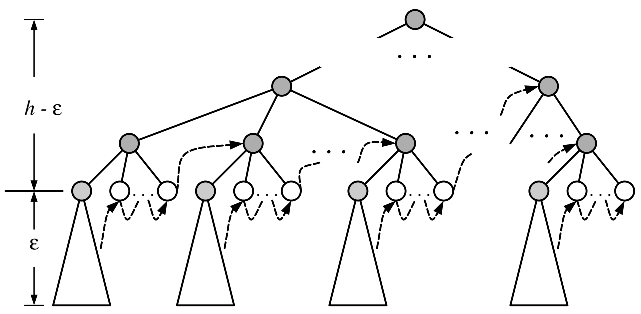


Fig. 8. B-ary tree exploring the maximum number of unproductive nodes.

height ϵ results in the loss of all nodes to be explored at higher levels. For complete B-ary tree this implies $\epsilon = \frac{m}{B}$. If the query needs to explore each one of these subtrees and all other nodes on level $h - \epsilon - 1$ would not need to be explored, the number of unproductive nodes is equal to the non-shaded nodes on level $h - \epsilon - 1$, as shown in Fig. 8. In this case, the number of unproductive nodes is proportional to $B^{h-\frac{m}{B}}$.

4. Experimental results

Our experimental work focuses on mobile clients executing queries on broadcasted R^*- and R-trees. The trees are built on point or rectangle data, and the queries executed by clients are either point queries (i.e., for a given point, determine all rectangles containing this point) or rectangle queries (i.e., for a given rectangle, determine all rectangles overlapping with the query rectangle or all points lying in the query rectangle). Such queries are traditionally called range queries. We compare the performance of Algorithms **NextQ** and **DoublePQ** executing the queries on a broadcast schedule without node repetition, Algorithm **Repeat-*l*** executing the queries on a schedule with node repetition as described in Section 3.3, and Algorithm **NoInfo**. Algorithm NoInfo assumes the index tree is broadcast in preorder without additional entries and without node repetition. A mobile client maintains nodes to be explored using FIFO management. Once nodes are lost, the client tunes in at every node until information accumulated allows again a more selective tuning. Algorithm NoInfo thus shows the loss in latency and tuning time resulting from the lack of next-entries.

We consider trees having between 5000 and 150,000 leaves and a node fanout B between 4 and 30. Node sizes range from 128 to 512 bytes. Both real and synthetic datasets are used. For synthetic point data, points are generated using a uniform distribution within the unit square. For synthetic rectangle data, the centers of the rectangles are generated uniformly in the unit square and the sides of the rectangles are generated with a uniform distribution between 10^{-5} and 10^{-2}. Real data was taken from the 2000 TIGER dataset from the US Bureau of the Census. We used data files on counties in US and extracted line segments from the road information. These line segments were used to generate rectangles (in the form of minimum enclosing rectangles) or points (using centers of line segments). Section 4.1 presents the performance of the four algorithms for R^*-trees on different data sets.

Index trees can either be created on-line (i.e., through the insertion of data) or off-line (having all data available and being able to preprocess it). Using common terminology, we refer to the tree created off-line as the packed tree [11]. The R^*-trees used in Section 4.1 are created on-line. This reflects the situation when the server has no control over the structure of the tree to be broadcast and it only generates the schedule. In some scenarios, index trees are created using data known in advance. In this case, data can be preprocessed (for example, by sorting it). This allows the creation of packed trees that have a smaller number of total nodes and better space utilization (i.e., almost all nodes have B children). In Section 4.2, we discuss the differences in performance observed for these two different ways of creating trees for R-trees. Section 4.3 considers the case when the server can choose the fanout of the tree broadcast and we examine the relationship between packet size and fanout of index nodes.

4.1. Comparisons for R^*-trees

This section compares the different client algorithms for the broadcast of dynamically created R^*-trees; i.e., the server broadcasts a tree created through the insertion of data. The R^*-trees were generated using code available from [2]. We consider both synthetic and real data on points as well as on rectangles.

4.1.1. General trends on latency and tuning time

Our first set of experiments, illustrated in Fig. 9, uses an R^*-tree with $B = 12$ and 10,000 leaves containing synthetic point data. The tree has a height of 6 and a total of 11,282 nodes. The results shown are for a fixed tree as queries vary as follows. A mobile client tunes in at a random point in the broadcast cycle and starts executing a rectangle query (i.e., report all points inside the given rectangle). The coordinates of the rectangle center of a query are chosen according to a uniform distribution and the sides are uniform between 0.002 and 0.5. Data reported is the average of 100 queries and each query is issued at 50 random time points within a broadcast cycle.

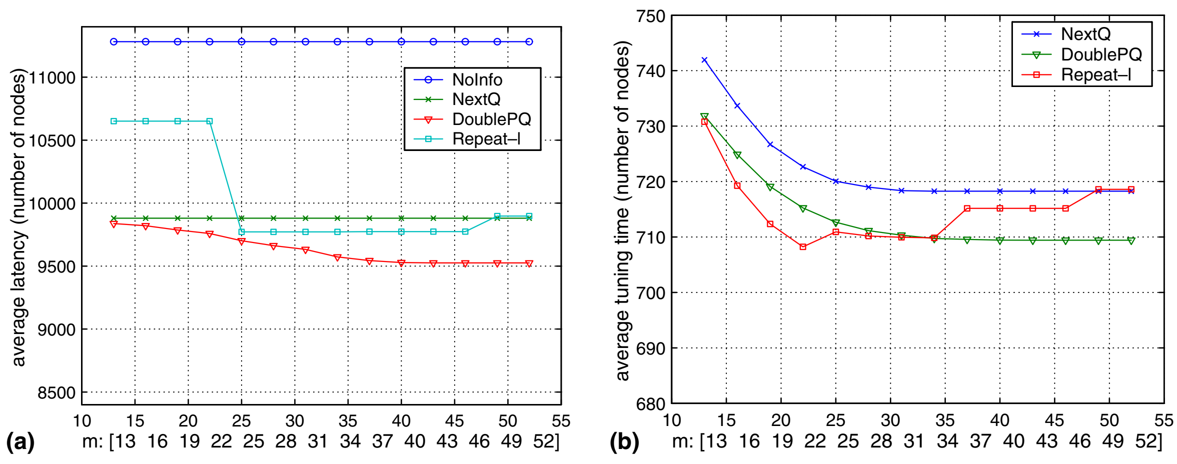


Fig. 9. Latency and tuning time for R^*-trees ($B = 12$, $n = 11{,}282$, $h = 6$, 10,000 leaves) for rectangle queries on point data.

Fig. 9 shows a comparison of the latency and tuning time. The x-axis reflects the size of the available memory. A memory of, say $m = 19$, means there is space for the addresses of 19 index nodes and for node-related entries needed by the algorithm (this number varies only slightly between different algorithms). The latency is influenced by the starting time of the query and the length of time a query continues executing when no more relevant data is to be found. Algorithm DoublePQ generally achieves the best latency. A reason for this lies in the fact that Algorithm DoublePQ is not so likely to delete "valuable" nodes. For the latency this means keeping nodes whose loss results in extending the time for recognizing the termination of the query. For the tuning time this means keeping nodes whose loss results in more unproductive nodes to be explored. The results for Algorithm Repeat-l shown in Fig. 9 are for the case when the repetition factor l chosen by the server matches m, the memory available in the client. This means the client never deletes non-repeated nodes to be explored (i.e., we have $l = h - \lfloor \frac{m}{B} \rfloor$). This is the best situation for a client with respect to latency and tuning time.

For the tuning time shown in Fig. 9(b) we omit Algorithm NoInfo since it consistently tunes in for about 6000 nodes. The tuning times of the other algorithms show the impact of the optimizations used by the algorithms. Algorithm NextQ has the highest tuning time (and thus the highest number of unproductive nodes). The tuning time for Repeat-l reflects that as memory increases and repetition decreases, the tuning time becomes identical to that of NextQ.

We conclude this section with latency and tuning time results for TIGER data. Fig. 10 shows latency and tuning times for rectangles generated from line segments associated with roads in Marion County, Indiana. The underlying data is shown in Fig. 11. The index tree was built for 75,431 leaves with $B = 12$. The rectangle center of query is randomly selected from the rectangle centers of data and the sides of the query rectangle are distributed uniformly between 0.001 and 0.25. This non-uniform data shows the same trend and characteristics as discussed for synthetic data sets. The remaining experiments focus on synthetic data sets. However, the conclusions hold for all the Tiger data sets we considered.

4.1.2. Effect of different query starting times on performance

Our solutions achieve the best performance when query processing starts at the begin of a broadcast cycle. For the sake of simplicity, the algorithms were explained assuming this scenario. However, this assumption is not relevant to the correctness of the described query algorithms. To show the difference in performance when a query starts at different points in the broadcast cycle, we compare two scenarios: starting query processing at the begin of the cycle and at the first leaf in the cycle. Starting at the first leaf is one of the worst-case situations, as no information about nodes on the path from the root to the leaf is available.

Fig. 12 compares the number of unproductive nodes for queries starting (a) at the first leaf, and (b) at the beginning of the broadcast. The results given are for an R^*-tree having 150,000 leaves (corresponding to points) and $B = 12$. The sides of queries range from 0.001 to 0.25. Observe that the latency is not impacted by the starting point of a query. Our algorithms ensure that latency is always bounded by the length of one cycle. The figure echoes and amplifies the trend already observed for the tuning time and the number

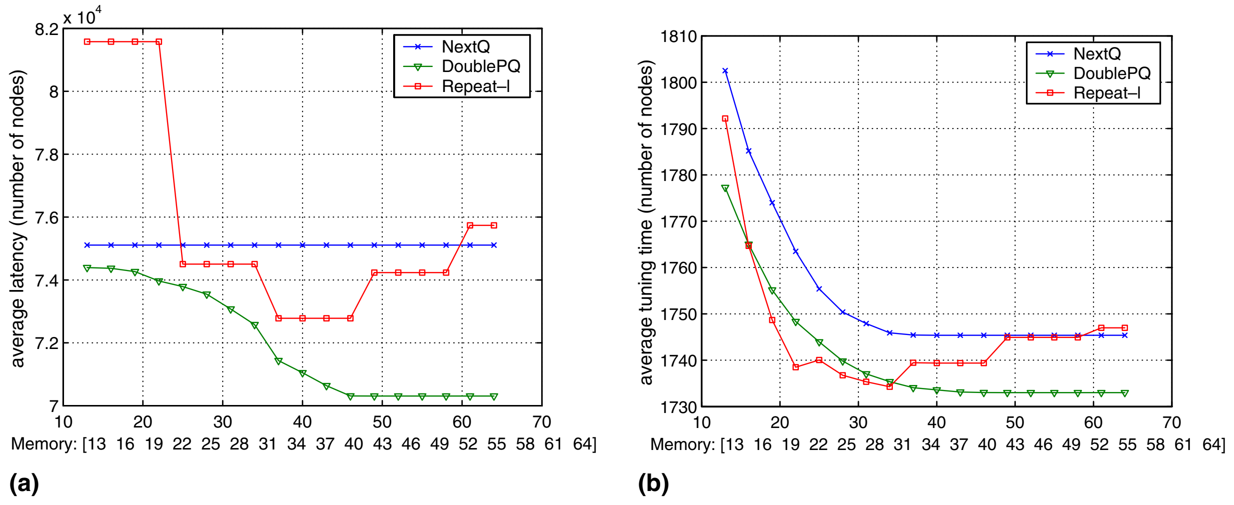


Fig. 10. Latency and tuning time for R^*-trees ($B = 12$, 75,431 leaves) for rectangle queries on rectangle data induced by road segments in Marion County, Indiana.

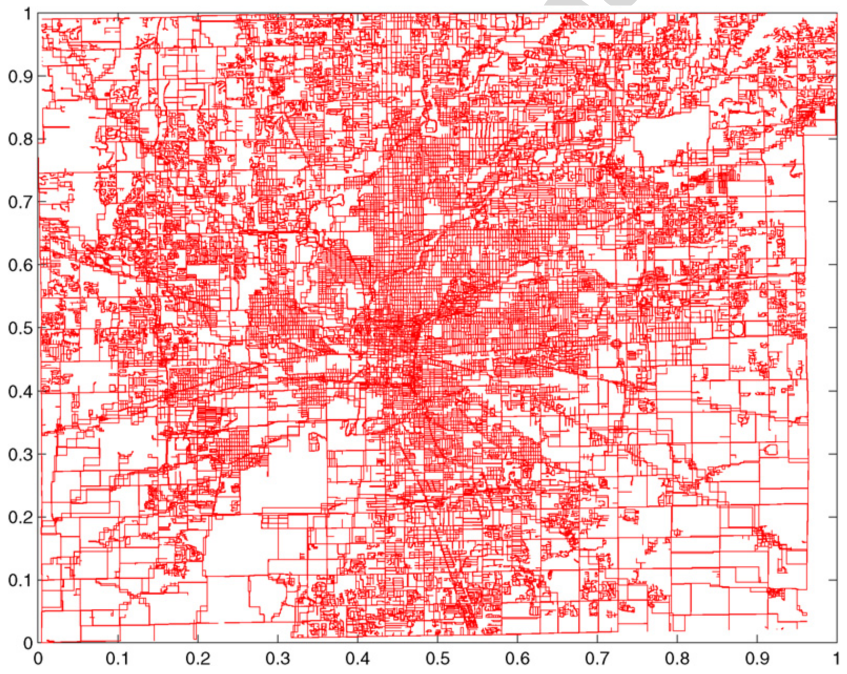


Fig. 11. 2000 TIGER data on roads in Marion County, Indiana.

of unproductive nodes. Starting a query at a leaf results in higher tuning times and more unproductive nodes for all algorithms. As memory increases to the point that no (or very few) nodes to be explored are lost, the differences in the number of unproductive nodes among the three algorithms become evident. Algorithm DoublePQ performs well especially when m is large. For large m, Algorithm DoublePQ performs well not because it keeps nodes whose loss is expensive (no algorithm loses many nodes for large m), but because Algorithm DoublePQ stores nodes together with its children (recall that a node in list L has a list of children to be explored). Algorithm Repeat-l is again shown for $\lfloor \frac{m}{B} \rfloor = h - l$. When m is small, the schedule broadcast for Repeat-l contains a larger number of repeated nodes. This is the reason for Repeat-l experiencing fewer unproductive nodes compared to DoublePQ and NextQ. In summary, Fig. 12 reflects that being able to handle small memory at a client efficiently leads to efficient handling of a query starting in the middle of a cycle.

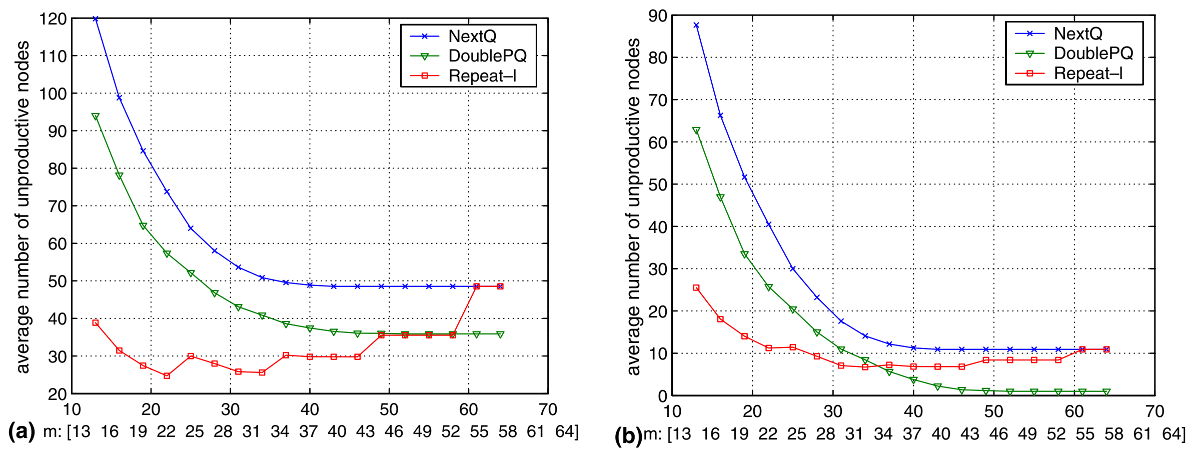


Fig. 12. Comparing the number of unproductive nodes explored for R^*-trees ($B = 12$, 150,000 leaves) for rectangle queries on point data: (a) queries issued at first leaf in the broadcast cycle and (b) queries issued at root (i.e., first node) in the broadcast cycle.

4.1.3. Relationship between *l* and *m* for Algorithm Repeat-*l*

The discussion on Algorithm Repeat-l in the previous two subsections considered the case when $l = h - \lfloor \frac{m}{B} \rfloor$. This corresponds to the situation when the repetition of nodes in the schedule matches the client's memory size; i.e., a client has enough memory to store the addresses of non-repeated nodes to be explored. Next, we explore the relationship between l and m on the performance when a client executes Algorithm Repeat-l. The experiments considered are for index trees built for point data of size 10,000. The queries executed by the client are rectangles having sides between 0.001 and 0.25. Fig. 13 gives the latency and the tuning time for different m and l values with $1 \leqslant l \leqslant h - 1$. Algorithm Repeat-$l$ corresponds to the white (non-filled) grid. Note that for $l = 0$, Repeat-l behaves like Algorithm NextQ and this value of l is not considered. For the sake of comparison, the plots include the performance of Algorithm DoublePQ on the same data sets using a broadcast schedule without repetition. Note that since the value of l is not relevant in this setup, the performance of DoublePQ is identical for different l values.

As l increases, the cycle length increases. The increased cycle length translates to higher latency. The latency is also influenced by the ability of a query to detect termination. In Fig. 13(a) we observe both of these properties. As l increases, the latency increases for Repeat-l. However, the latency decreases when the schedule switches from repeating the root (i.e., $l = 1$) to repeating the first two levels. This decrease in the latency from $l = 1$ to $l = 2$ is due to the ability of detecting termination of a query faster for $l = 2$. While the cycle length

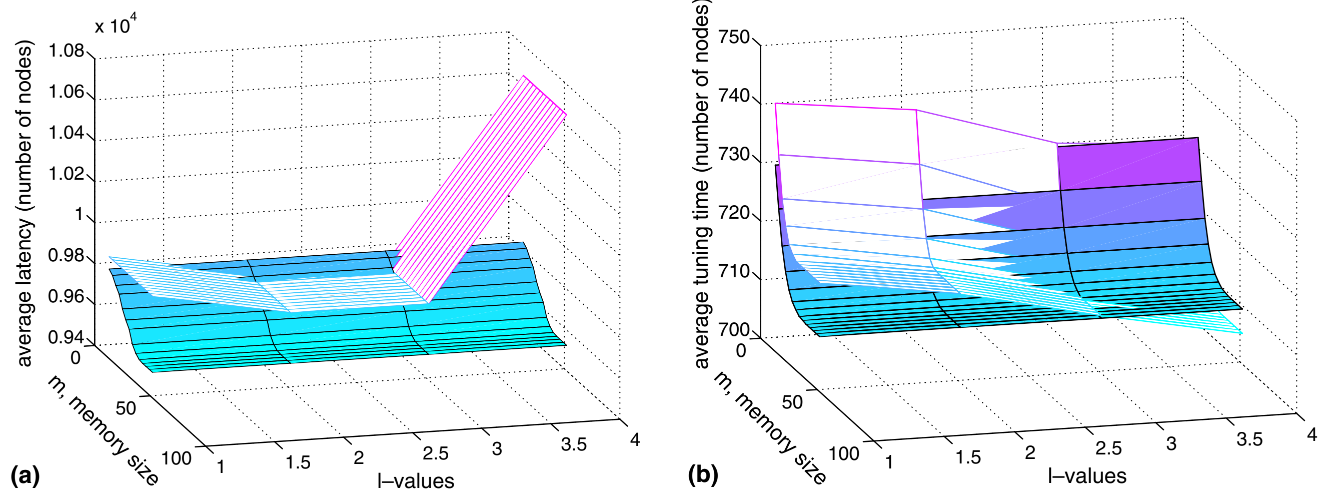


Fig. 13. The latency and tuning time for Algorithm Repeat-l (non-filled grid) and Algorithm DoublePQ (filled grid) with different m- and l-values for an R^*-tree of height $h = 6$ and having 10,000 leaves: (a) latency and (b) tuning time.

increases somewhat for $l = 2$, the increase does not yet impact the latency. This impact is, however, seen for larger values of l. We point out that in Fig. 9(a) and other figures showing the latency of Repeat-l as the available memory changes, we see a change in the latency for different m values. On the other hand, there is no change in Fig. 13(a). The reason lies in the different assumptions for l and m. In Fig. 13(a), l is fixed as m changes. This results in the cycle length not changing and thus the latency not changing. In Fig. 9(a) and the other plots on the latency, we adjust l give the best performance for the chosen m, thus resulting in a different cycle length and different latency.

For Algorithm Repeat-l, the tuning time decreases as l increases, as shown in Fig. 13(b). In addition, the tuning time decreases for both Repeat-l and DoublePQ as a client's memory size increases. From our experimental results we can conclude that for $l \geqslant h/2$, Repeat-l achieves a better tuning time than Algorithm DoublePQ. When considering the latency, DoublePQ is always better than Repeat-l.

4.2. Packed versus on-line R^*-tree creation

When the server creates the index tree only for the broadcast, it should create a tree best suited for query processing in the mobile environment. In this section we compare the performance when broadcasting an R^*-tree created through the insertion of data versus broadcasting a packed R-tree. We use the technique proposed in [11] for generating a packed R-tree. For a given data set, we first sort the data and then build an R-tree bottom-up, grouping the data and proceeding level by level. Among the variations for generating a packed R-tree proposed in [11], we select the one recommended as the best and which sorts the data by Hilbert values. When the data are rectangles, we sort the rectangles according to the Hilbert value of the center point of the rectangles.

Fig. 14 shows the latency and the tuning time for a packed R-tree, using the same data as in Fig. 9. The fanout is fixed to 12. The packed R-tree has a height of $h = 5$ and $n = 10{,}911$ nodes, compared to a height of $h = 6$ and $n = 11{,}282$ for the R^*-tree created through random insertions. The packed tree achieves a smaller total number of nodes and a smaller height by giving almost all nodes 12 children. The trend among the algorithms using a packed trees is as observed for trees created in an on-line fashion. Algorithm DoublePQ achieves the best overall performance. The improvement in the latency compared to the on-line tree creation is due to the smaller number of nodes in the packed tree. Fig. 15 compares the number of unproductive nodes between the packed R-tree and the on-line R^*-tree (the value shown is packed minus on-line). For all three algorithms, the packed tree explores more unproductive nodes (the difference shown in the graph is always positive). This has been observed for all the data we considered. The reason for the packed tree exploring more unproductive nodes lies in the fact that in the packed tree almost all nodes have B children. When a node is identified as productive, it tends to have more children that need to be explored and thus more unproductive nodes overall. Index nodes with a smaller fanout allow more efficient "searching" for productive nodes. In summary, a packed tree achieves a smaller latency, but the improvement in the tuning time is not significant.

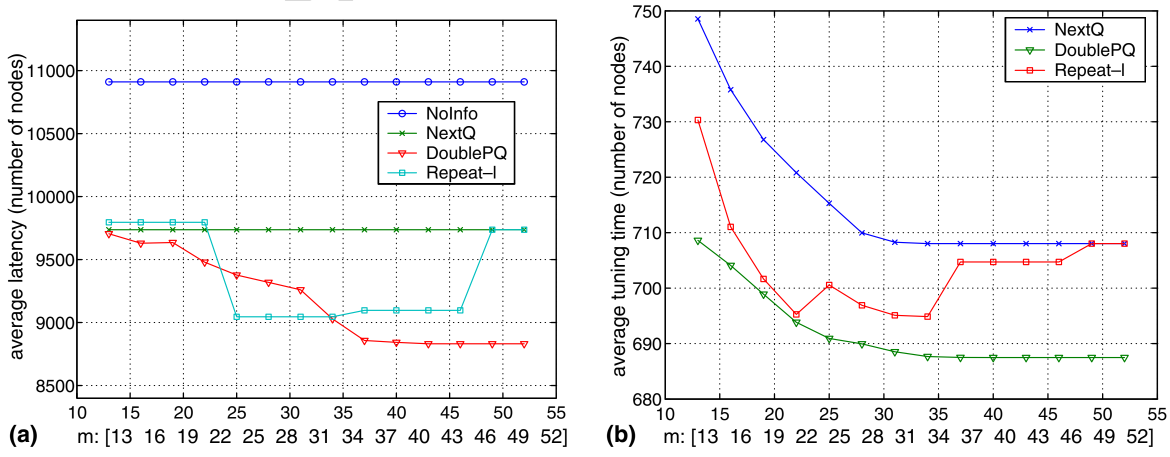


Fig. 14. Latency and tuning time for a packed R-tree ($B = 12$, $n = 10{,}911$, $h = 5$, and 10,000 leaves) for rectangle queries on point data.

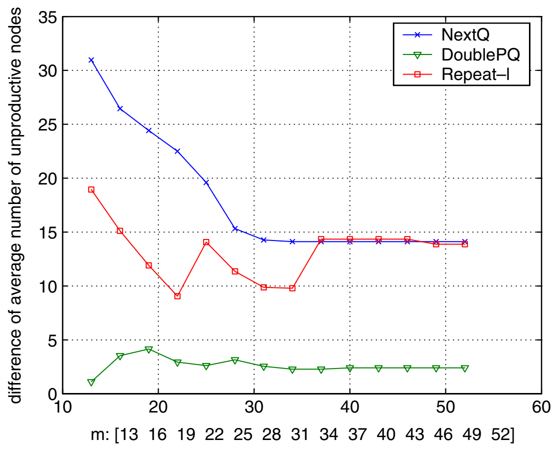


Fig. 15. Difference in the number of unproductive nodes explored between a packed R-tree and an on-line R^*-tree. Both are created for the same data sets on 10,000 point data items.

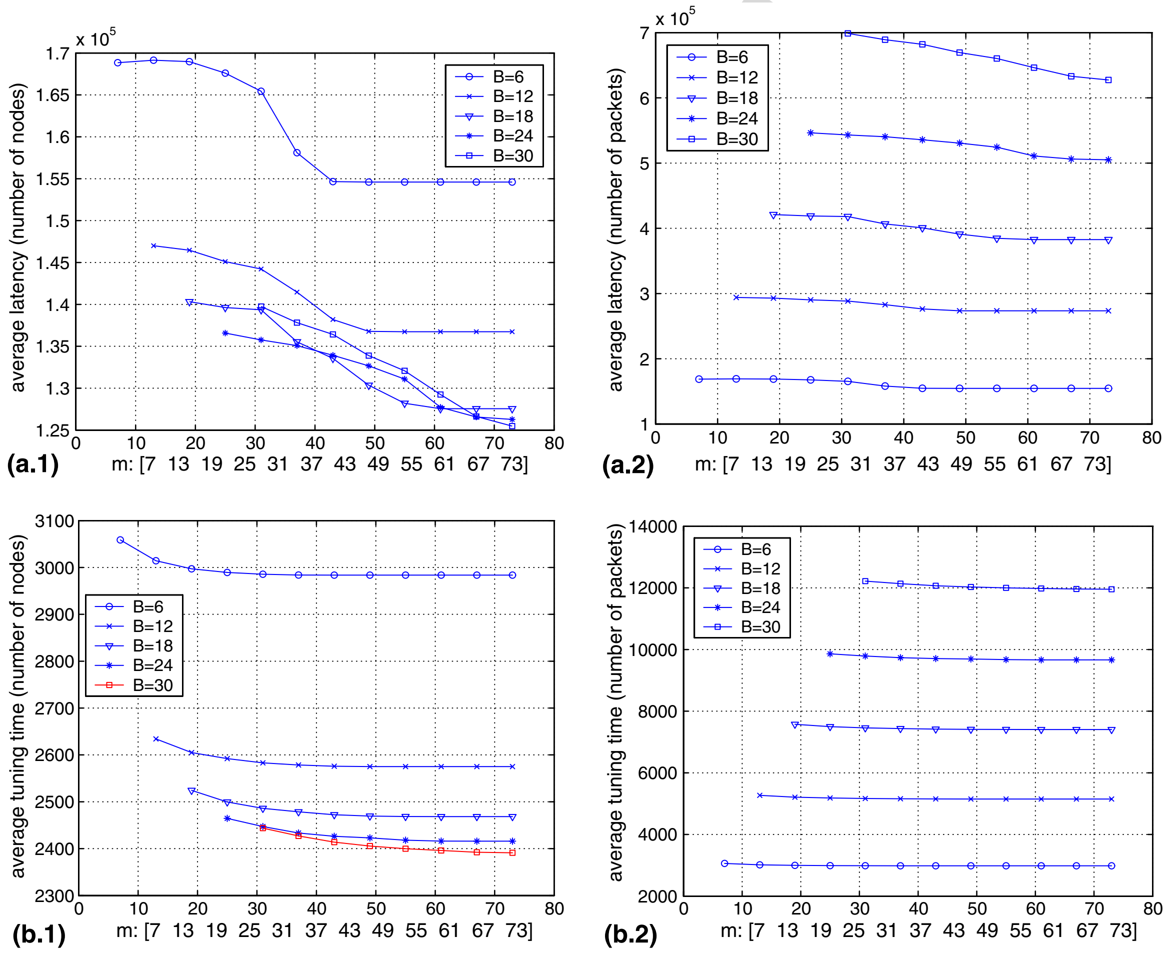


Fig. 16. Latency and tuning time based on nodes and packet metric for five different B-values. Results shown are achieved by Algorithm DoublePQ on an R^*-tree with 150,000 leaves: (a.1) latency (number of nodes), (a.2) latency (number of packets), (b.1) tuning time (number of nodes) and (b.2) tuning time (number of packets).

4.3. Packet and index node size

The previous section showed that broadcasting a packed R-tree leads to only a minor improvement in the tuning time compared to broadcasting a dynamically created tree. Another parameter impacting the performance is the fanout, B, of the index tree. In this section we consider the broadcasting of the same data set using index trees with different fanouts. Ref. [10] discusses the impact of the packet (bucket) size when each packet may contain a number of nodes. A conclusion is that a larger number of nodes in a packet has a positive effect on latency and the impact of the packet size on the tuning time and latency depends on the indexing method.

We assume a node uses at least one packet when broadcast. We use two metrics when measuring latency and tuning time: the number of nodes explored and the number of packets tuned into. When counting packets, the cost of exploring a node is proportional to the number of packets used by the node. We assume that nodes do not share packets. We considered B-values for which one index node uses one packet as well as B-values for which one node corresponds to a chosen, fixed number of packets. Our discussion assumes that one packet can hold a fanout of $B = 6$ (corresponds to 156 bytes). The B-values we consider are $B = 6$, 12, 18, 24, and 30. Note that a node with fanout $B = 6p$ corresponds to p packets.

Fig. 16 shows the results of the node-based and packet-based metrics for R^*-trees on Algorithm DoublePQ. The trees are created dynamically for a data set of 150,000 points. The sizes of the trees in terms of the number of nodes are $n = 194{,}117$ for $B = 6$, $n = 169{,}243$ for $B = 12$, $n = 162{,}558$ for $B = 18$, $n = 159{,}212$ for $B = 24$, and $n = 157{,}245$ for $B = 30$. The height of the trees ranges from 9 for $B = 6$ to a height of 5 for $B = 30$. Surprisingly, the largest B-value fails to generate the best latency in the node-based metric. The tuning time is only slightly better. The reason lies in the relationship between B and m, the memory size of a client. If m cannot grow with the fanout B, performance deteriorates since many nodes to be explored are identified, but their addresses cannot be stored. In the packet-based metric, the index size corresponding to one packet (i.e., $B = 6$) gives, not surprisingly, the best performance. As B increases, the latency as well as tuning time are predictable and change only slightly as the memory size increases. The results indicate that the best choice for the fanout of the index tree depends on the packet size. In summary, our conclusion is that the fanout of the broadcasted index tree should be tailored towards the packet size of the broadcast environment as well as the memory size of the clients.

5. Conclusions

We considered the efficient execution of queries by mobile clients on broadcasted index trees, focusing on the generation of the broadcast schedule generated by a server and the query algorithm executed by a client. Efficiency is measured in terms of latency and tuning time experienced by a client, and clients cannot exceed given memory constraints. We presented two client algorithms that differ on how a mobile client decides which data to delete when no more additional data can be stored and two broadcast schedules which differ on whether nodes are repeated in the broadcast. Our experimental study was carried out on R- and R^*-trees built on 2-dimensional data sets with clients executing range queries. Overall, we found that algorithm DoublePQ on the schedule without node repetition achieves the best performance, suggesting that priority-based data management is effective. The impact of repeated nodes in the broadcast schedule could be limited due to the data sizes considered. Our results also showed that using a packed index tree compared to one created through random insertions has little impact on the performance, while tailoring the fanout of a node to the packet size in the wireless environment can lead to overall improvement.

Acknowledgements

We thank the referees for numerous constructive comments and suggestions.

References

[1] S. Acharya, R. Alonso, M.J. Franklin, S.B. Zdonik, Broadcast disks: data management for asymmetric communications environments, in: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data 22–25 May 1995, pp. 199–210.

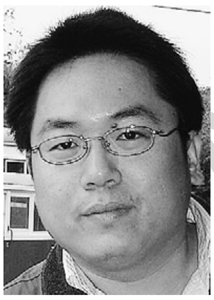
[2] N. Beckmann, H. Kriegel, R. Schneider, B. Seeger, The R^*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, 23–25 May 1990, pp. 322–331.

[3] Y.-I. Chang, C.-N. Yang, A complementary approach to data broadcasting in mobile information systems, Data Knowl. Eng. (2002) 181–194.

[4] M.S. Chen, K.-L. Wu, P.S. Yu, Optimizing index allocation for sequential data broadcasting in wireless mobile computing, IEEE Trans. Knowl. Data Eng. 15 (1) (2003) 161–173.
[5] M.-S. Chen, P.S. Yu, K.-L. Wu, Indexed sequential data broadcasting in wireless mobile computing, in: Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS), IEEE Computer Society Press, 1997.
[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, McGraw-Hill, New York, 1990.
[7] M. Franklin, S. Zdonik, A framework for scalable dissemination-based systems, in: OOPSLA '97 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications, ACM Press, 1997, pp. 94–105.
[8] S. Hambrusch, C.-M. Liu, W. Aref, S. Prabhakar, Query processing in broadcasted spatial index trees, in: Proceedings of 7th International Symposium on Spatial and Temporal Databases SSTD 2001, July 2001, pp. 502–521.
[9] T. Imieliński, S. Viswanathan, B.R. Badrinath, Energy efficient indexing on air, in: Richard T. Snodgrass, Marianne Winslett (Eds.), Proceedings of the International Conference on Management of Data, ACM Press, 1994, pp. 25–36.
[10] T. Imieliński, S. Viswanathan, B.R. Badrinath, Data on air: organization and access, IEEE Trans. Knowl. Data Eng. 9 (3) (1997) 353–372.
[11] I. Kamel, C. Faloutsos, On packing *r*-trees, in: Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM), November 1993, pp. 490–499.
[12] S. Khanna, S. Zhou, On indexed data broadcast, J. Comput. Syst. Sci. 60 (2000) 575–591.
[13] S.-C. Lo, A.L.P. Chen, An adaptive access method for broadcast data under an error-prone mobile environment, IEEE Trans. Knowl. Data Eng. 12 (4) (2000) 609–620.
[14] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, in: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA, 1995, pp. 71–79.
[15] H. Samet, Applications of Spatial Data Structures: Computer Graphics Image Processing GIS, Addison-Wesley, Reading, MA, 1990.
[16] H. Samet, The Design and Analysis of Spatial Data Structures, Addison-Wesley, Reading, MA, 1990.
[17] C.-J. Su, L. Tassiulas, V.J. Tsotras, Broadcast scheduling for information distribution, ACM/Baltzer J. Wireless Network 5 (2) (1999) 137–147.
[18] K.L. Tan, B.C. Ooi, On selective tuning in unreliable wireless channels, Data Knowl. Eng. 28 (2) (1998).
[19] K.L. Tan, J.X. Yu, P.K. Eng, Supporting range queries in a wireless environment with nonuniform broadcast, Data Knowl. Eng. 29 (2) (1999) 201–221.
[20] J.X. Yu, K.L. Tan, An analysis of selective tuning schemes for nonuniform broadcast, Data Knowl. Eng. 22 (3) (1997) 319–344.
[21] J. Zhang, L. Gruenwald, An access time cost model for spatial range queries on broadcast geographical data over air, in: Proceedings of the 14th International Conference on Database and Expert Systems Applications (DEXA), 2003, pp. 371–380.
[22] J. Zhang, L. Gruenwald, Optimizing data placement over wireless broadcast channel for multi-dimensional range query processing, in: 5th IEEE International Conference on Mobile Data Management (MDM 2004), January, Berkeley, CA, USA, 2004, pp. 256–265.
[23] B. Zheng, W.-C. Lee, D.L. Lee, Spatial queries in wireless broadcast systems, Wireless Networks 10 (6) (2004) 723–736.

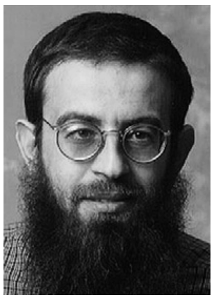
Susanne E. Hambrusch is Professor and Head of the Department of Computer Sciences at Purdue University. She received the Diplom Ingenieur in Computer Science from the Technical University of Vienna, Austria, in 1977, and a Ph.D. in Computer Science from the Penn State in 1982. In 1982 she joined the faculty at Purdue University. She has served at the head of the department since 2002.

Dr. Hambrusch's research interests are in parallel and distributed computation, data management and data dissemination in wireless environments, and analysis of algorithms. Her research has been supported by NSF, AFO, ONR, Microsoft Corp., Intel, and DARPA. Dr. Hambrusch is a member of the Editorial Board for Parallel Computing and Information Processing Letters and she serves on the IEEE Technical Committee on Parallel Processing. Her recognition's include a 2003 Outstanding Engineering Alumni Award from Pennsylvania State University, and 2004 TechPoint Mira Education Award.



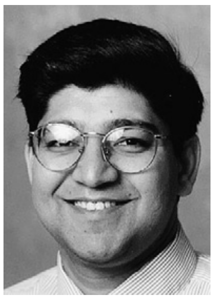
Chuan-Ming Liu is an Assistant Professor in the Department of Computer Science and Information Engineering, National Taipei University of Technology (NTUT). He received his Ph.D. in Computer Science from Purdue University in 2002 and B.S. and M.S. degrees both in Applied Mathematics from National Chung-Hsing University, Taiwan, in 1992 and 1994, respectively.

Dr. Liu is a member of Upsilon Pi Epsilon Honor Society in Computer Science. His research interests include parallel and distributed computation, data management and data dissemination in wireless environments, and analysis and design of algorithms.



Walid G. Aref is Professor of Computer Science at Purdue University. He received an M.Sc. in Computer Science in 1986 from Alexandria University, Egypt, and a Ph.D. in Computer Science in 1993 from the University of Maryland at College Park.

Professor Aref's interests are in developing database technologies for emerging applications, e.g., spatial, spatio-temporal, multimedia, genomics, and sensor databases. He is also interested in indexing, data mining, and geographic information systems (GIS). Professor Aref's research has been supported by the National Science Foundation, Purdue Research Foundation, CERIAS, Panasonic, and Microsoft Corp. In 2001, he received the CAREER Award from the National Science Foundation and in 2004, he received a Purdue University Faculty Scholar award. Professor Aref is a member of Purdue's Discovery Park Bindley Bioscience Center. He is on the editorial board of the VLDB Journal, a senior member of the IEEE, and a member of the ACM.



Sunil Prabhakar is an Associate Professor of Computer Science at Purdue University. Received his Ph.D. in Computer Science from the University of California at Santa Barbara in 1998, and a B.Tech in Electrical Engineering from the Indian Institute of Technology, Delhi. He joined Purdue's CS Department in 1998 as an Assistant Professor.

Dr. Prabhakar's research focuses on performance and privacy issues in large-scale, modern database applications. He is currently working on uncertainty management and efficient execution for high-update database applications such as sensor databases, enhancing privacy and digital rights management for databases, and the development of advanced databases for bioinformatics. Prior to joining Purdue, Dr. Prabhakar held a position with Tata Unisys Ltd. from 1990 to 1994.