Broadcasting Indexed Multidimensional Data*†

Susanne Hambrusch Chuan-Ming Liu Walid G. Aref
Sunil Prabhakar

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA

{seh,liucm,aref,sunil}@cs.purdue.edu

July 5, 2001

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. This minimizes the tuning time (i.e., the amount of time spent listening to the channel) and thus the power consumption experienced by the client. This paper investigates the execution of queries on broadcasted multidimensional index structures. The execution of such queries corresponds to a partial traversal of the tree, not a single-path root-to-leaf search. Our solutions minimize the tuning time and the latency (i.e., the time elapsed between issuing and termination of the query). They differ in assumptions on the format of the broadcasted index tree, how a mobile client makes use of its limited memory, and the data structures employed by a client. Experimental work on real and synthetic data shows that different broadcast schedules and client algorithms lead to different tuning times and latencies. Our solutions also provide efficient methods for starting the execution of a query in the middle of a broadcast cycle.

Keywords: Data dissemination, multidimensional index trees, latency and tuning time, query processing, wireless broadcasting.

*Work Supported by NSF CAREER Grant IIS-9985019, NSF Grant 9988339-CCR, a Gift from Microsoft Corp., and the Purdue Research Foundation.

†A preliminary version of this paper appears in the Proceedings of *7th International Symposium on Spatial and Temporal Databases,* July 2001.

1 Introduction

The continuous broadcast of data together with an index structure is an effective way of disseminating data in a wireless mobile environment [2, 4, 5, 7]. The index allows mobile clients executing a query to tune into a continuous broadcast only when relevant data is available on the channel and thus minimizes power consumption. A client experiences latency (the time elapsed elapsed between issuing and termination of the query) and tuning time (the amount of time spent listening to the channel). This paper considers the execution of queries on broadcasted multidimensional index structures. Such queries require a partial traversal of the index structure, not a single-path root-to-leaf search. Examples of queries are searching in multidimensional R-trees and R^*-trees and range queries in quad-trees and k-d-trees [8, 9].

Assume that an n-node multidimensional index tree is broadcast in a wireless environment. A server schedules the tree for the broadcast and may or may not be creating the tree. Scheduling a tree for broadcast involves determining the order in which nodes are sent out, deciding whether and which nodes of the tree are broadcast more than once in a cycle (i.e., whether repetition is allowed within a broadcast cycle), and adding other data entries to improve performance (in particular the tuning time). Mobile clients execute queries by tuning into the broadcast at arbitrary times and traversing parts of the broadcasted tree. We assume that mobile clients operate independently of each other. This paper focuses on the execution of queries in R- and R^*-trees. Queries we consider include searching for a specific record and range queries. In either query type, 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 papers which consider only root-to-leaf searches [2, 4]. We point out that our work applies to other tree-based index structures as well, in particular to queries on broadcasted quad- and k-d trees.

Traversal and partial traversal of a multidimensional index tree is straightforward when the mobile client happens to tune in at the beginning of a broadcast cycle and the client can locally store received addresses of nodes to be tuned into later. However, this may cause a client to store information of hB nodes, where h is the height of the index tree and B is the maximum number of children. When a client's memory is limited or a query starts executing during the on-going broadcast cycle, performance depends on what information is maintained

at a client, what entries are made available by the scheduler, and how the client makes use of the broadcasted information.

We present and analyze three algorithms for executing queries on a broadcasted index tree. Our algorithms differ on how a mobile client decides which data to delete when no more additional data can be stored, the degree of repetition of nodes in the broadcast, and the type of data structures employed by a client. The algorithms are presented in Section 3. Our experimental work compares these algorithms on real and synthetic point and rectangle data sets for R^*- and R-trees. Experimental work is discussed in Section 4. Our experimental results show that these methods lead to different tuning times and latencies. We 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 all previously read data. We also explore the impact of the structure of the tree on the overall performance. This includes comparing the broadcasting of packed (i.e., static) and dynamic trees and a relationship between packet size and index node size. Section 5 contains concluding remarks.

2 Assumptions and Preliminaries

Our solutions do not make any assumption about the structure of the tree broadcast. Trees can range from highly structured multidimensional index trees, like R^*- and R-trees, to random trees. We use B to denote the fanout (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. For any index node v, we assume the broadcast contains the entries generally present in the corresponding index structure. This includes an identifier and data of node 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 at the point in time when v' appears in the broadcast. The address corresponds to the offset used in [5, 7] to capture the time a packet is broadcast.

The broadcast of one instance of the tree is called a *cycle*. The cycle length of any broadcast is at least cn, for some constant $c \geq 1$. We assume throughout the paper that a cycle contains the nodes of the tree according to a preorder traversal, possibly with some nodes being broadcast

more than once in case of repetition in the schedule. This assumption allows a client to (i) complete a query in one cycle when tuning in at the begin of a cycle and to (ii) minimize the number of nodes to be stored at a client while minimizing the tuning time. We point out that a query can always be executed in one cycle when a mobile client tunes in at every node, thus achieving a latency and tuning time equal to the cycle length. A broadcast of the data without an index structure requires a client to tune in at every data item, thus making the tuning time equal to the latency. The use of an index allows a reduction of the tuning time, allowing clients to minimize their power consumption.

We say a client *explores* node u when the client tunes into the broadcast to receive u and examines all of u's entries. When none of u's children needs to be explored further, u is an *unproductive node*, otherwise u is called a *productive* node. The scheduler includes additional entries to the broadcasted index nodes, as discussed in Section 3.

The objective 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 as to be explored. Not being able to store all these addresses results in an increase in the number of unproductive nodes explored and thus increases the tuning time. When comparing index trees with the 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 present three algorithms for executing a query during the broadcast of an index tree when the mobile client has memory of size m, $m < hB$. We assume that each "unit" of memory can store the address of a node and optionally a small number of other entries. Limited memory implies that a client is not always able to store all relevant information received earlier and losing data can increase the tuning time. For each child of an index node v, the broadcast

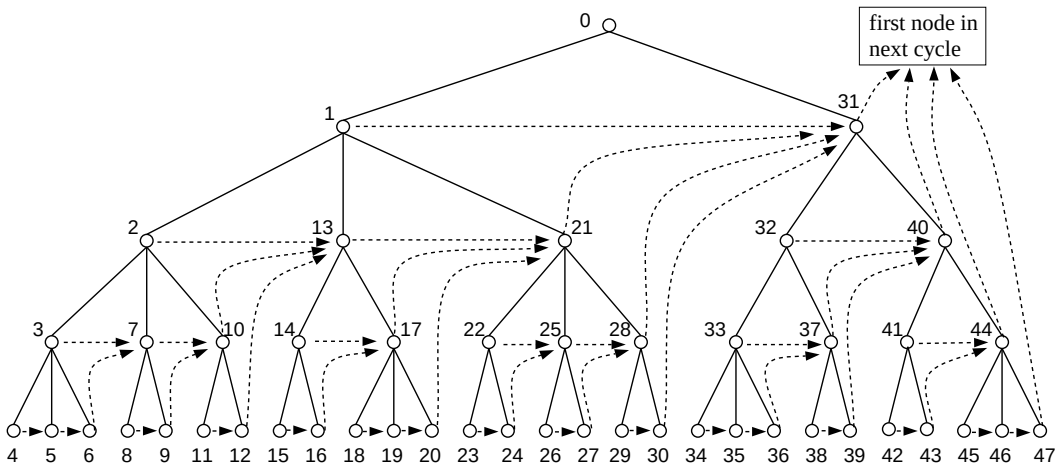


Figure 1: A tree with $n = 48$, $B = 3$, and *next*-entries (dashed)

schedule contains the address of the child. In addition, for every node v, the schedule contains an entry $next(v)$ which is the address of the node in the broadcast after all descendents of v have been broadcast. This can be a sibling of v or a higher-degree cousin, as shown in Figure 1.

The algorithms of Sections 3.1 and 3.2 assume that the index tree is broadcast in preorder, without repetition of nodes (i.e., every node appears once in the broadcast cycle), and with a *next*-entry for every node. The two algorithms differ on how the client decides what entries to delete when required to free up memory. The index tree is scheduled by the server without knowledge of m, the space available at a client. The program executed by the client is tailored towards its own available memory.

The algorithm described in Section 3.3 assumes a broadcast schedule in which selected nodes are repeated in the cycle. The amount of node repetition is determined by the server with the goal of broadcasting more frequently needed index nodes more often. The memory size m of a client is independent of the amount of repetition in the schedule. Our work shows how latency and tuning time depend on the relationship between m and the degree of repetition.

3.1 Using Next Entries

Assume an index tree is broadcast by the server using a preorder schedule with a *next*-entry for every node, as shown in Figure 1. A mobile client starts the execution of a query at an arbitrary point during the broadcast cycle. The client tunes in to receive and explore the next node in the broadcast, say node v.

The client maintains a deque Q [3]. The deque is empty at the start of a query and can contain at most m entries. The top of the deque experiences insertion and deletions, while the bottom experiences only deletions. If node v is a productive node, let $v_1, ..., v_k$ be the children of v to be explored. Nodes $v_1, ..., v_k$ are inserted at the top of Q in reverse order of their address in the broadcast, along with the address. 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 deleted from Q will be recreated using *next*-entries of nodes explored later in the broadcast. While the bottom of Q deque experiences deletions when space is needed, the top of Q experiences a delete operation right before the exploration of a node. Figure 2 gives a high-level description of the algorithm executed by the client.

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 in this case. After the exploration of an unproductive node, the next node to be tuned into is either found in the deque Q or, if Q is empty, it is entry $next(v)$. Finding the next node to tune into is described in Algorithm FindNext given in Figure 2.

Figure 3 shows the traversal of a tree for $m = 3$ when the query executed generates the data stored in leaves $4, 5, 18, 20, 34$ and 35. After the exploration of node 1, deque Q contains three nodes, namely 2, 13, and 31. Node 2 is explored and deleted from Q. Nodes 3, 7, and 10 are to be added. Since $m = 3$ and Q contains already two entries (13 and 31), a deletion to free up space takes place. Nodes 13 and 31 are deleted and nodes 3, 7, and 10 are added. This is the status of Q before node 3 is explored. Exploring node 3 results in the deletion of node 10. The address of node 10 is re-obtained from the entries of node 7 for which $next(7)$=10. In total, *next*-entries are used four times, as indicated in the figure.

Exploring a node v having k children costs $O(k)$ time. Assume v is unproductive. If the deque is not empty, deleting the most recently added node gives the next node to tune into. This node had a productive parent, but it can be productive or unproductive. When the deque is empty, we tune in at $next(v)$. This node may have had an unproductive parent. For a given query, there exist trees and queries for which a client tunes in at $\Theta(B^{h-\frac{s}{B}})$ unproductive nodes. Minimizing the unproductive nodes explored corresponds to minimizing the tuning time. The algorithm described in the next section uses a different metric to delete nodes from the queue with the goal of minimizing unproductive nodes.

When a mobile client tunes into the broadcast at an arbitrary point during the cycle, it starts

Algorithm ExploreQ(v)
(1) explore node v;
(2) **if** node v is a leaf node **then**
determine the relevance of the data stored in the leaf node 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, arranged
in the order they appear in the broadcast;
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=FindNext(v);
(4) ExploreQ(u)
End Algorithm ExploreQ

Algorithm FindNext(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 FindNext

Figure 2: Client algorithm for query execution during a tree broadcast with *next*-entries

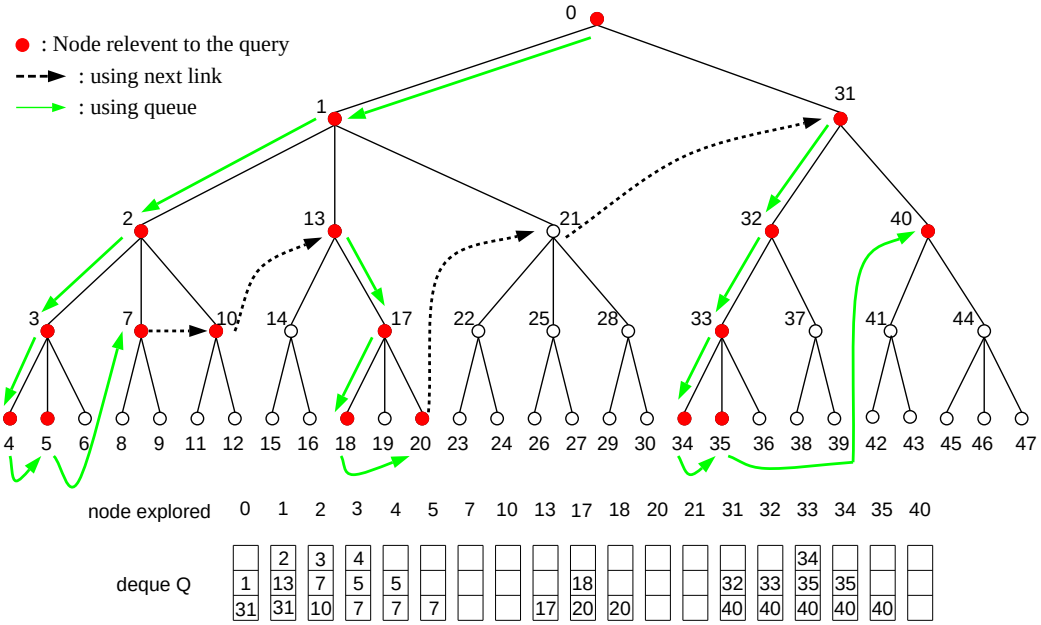


Figure 3: Processing a query issued at the begin of the cycle and using *next*-entries, $m = 3$. The entries below the tree show deque Q after the exploration of the corresponding node.

executing the program with an empty deque. 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. Hence, the latency experienced is at most one cycle length.

The *next*-entries allow a client to reach parts of the tree at higher levels without having seen the higher level nodes and without having to tune in and explore every node. Tuning time can be minimized by skipping the on-going cycle and beginning the processing of the query at the start of the next cycle, at the expense of a larger latency.

3.2 Using an exploration-based cost function

In the previous algorithm, a client uses a deque Q of size at most m to hold the most recent nodes identified as index nodes to be explored. This section describes a solution in which a client keeps 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 and that do **not** need to be explored by the query.

Quantity $cost(u)$ measures the loss of u in terms of the additional unproductive nodes a client needs to tune into in case the node is deleted from the queue. Entry $cost(u)$ is determined by the client, at no additional cost, while node u is explored. The tree is scheduled for broadcast as in Section 3.1; i.e., a preorder schedule with every node having a *next*-entry.

We discuss the algorithm using the implementation underlying our experimental work. 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 tuned into and explored, in the order of the children in 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. In L, deletions occur in a number of ways. First, when space is needed, a node and its entire child-list are deleted (which node is determined by the *cost*-entries). After a node in a child-list has been explored, the node is deleted from the child-list. Finally, when a node has an empty child-list, it is deleted from list L. Insertions to L happen when a node is explored at which point the node and its child-list are inserted. 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.

The entry for node v in list L contains the following:

- node v's id
- the child-list containing the children of v not yet broadcast and to be explored
- $next(v)$
- a pointer to node v in queue PQ

If node v is in L with k children in the child-list, we consider node v to be using $k+1$ memory locations. Node v has an entry in PQ containing $cost(v)$ and a pointer to node v in list L.

When a mobile client tunes in for node v, it first explores node v and computes $cost(v)$, as done in step (1) of Algorithm ExplorePQ in Figure 4. When v is productive with children $v_1, ..., v_k$, 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 nodes based on *cost*-entries: The node with minimum

Algorithm ExplorePQ(v)
(1) explore node v and compute $cost(v)$;
(2) **if** node v is a data node **then**
    determine the relevance of the data to the query
  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:
      (1.1) insert v into PQ with $cost(v)$;
      (1.2) **while** there is not enough space for v and its k children **do**
        (a) determine node u in PQ having minimum cost;
        (b) delete all entries of node u and u's child-list from L
        endwhile
      (1.3) insert v and v's child-list to L
    endif
  endif
(3) u = FindNextNode(v);
(4) ExplorePQu)
End Algorithm ExplorePQ

Algorithm FindNextNode(v)
if list L is empty **then**
  return $next(v)$
else
  let u be the most recently added node in L;
  let w be the first node in the child-list of u;
  (1) **while** w has been broadcast in the cycle and the child-list is not empty **do**
    (1.1) delete w from u's child-list and update $cost(u)$;
    (1.2) update w to be the new first node in the child-list of u;
    endwhile
  (2) **if** the child-list of u is not empty **then**
    return w, if w arrives before $next(v)$ in the broadcast,
    otherwise **return** $next(v)$
    else
      (2.1) delete entries related to u from L and PQ;
      (2.2) FindNextNode(u)
    endif
endif
End Algorithm FindNextNode

Figure 4: Client algorithm for query execution during the tree broadcast using next-entries and a priority-based rule for freeing up memory.

cost-value is deleted from PQ, along with this node's entry and child-list in L. Then, Algorithm FindNextNode is invoked to find the the next node to be explored.

Since nodes are not deleted according to the FIFO management, a non-empty list L does not imply that the address of the node to be tuned into next can be obtained from L. Let u be the most recently added node in L. Step (1) of FindNextNode scans u's child-list to identify the first child not yet broadcast. Let w be this node. If u is the parent of v, this is the next sibling of v to be explored. However, w can be the child of a non-parent ancestor of v. Observe that we do not need to identify whether u is a parent of v. FindNextNode determines whether to tune it at w or at $next(v)$ by comparing their addresses and tuning it at the one arriving earlier. If node u had no more children to explore, u is deleted from L and the search for the next node continues recursively.

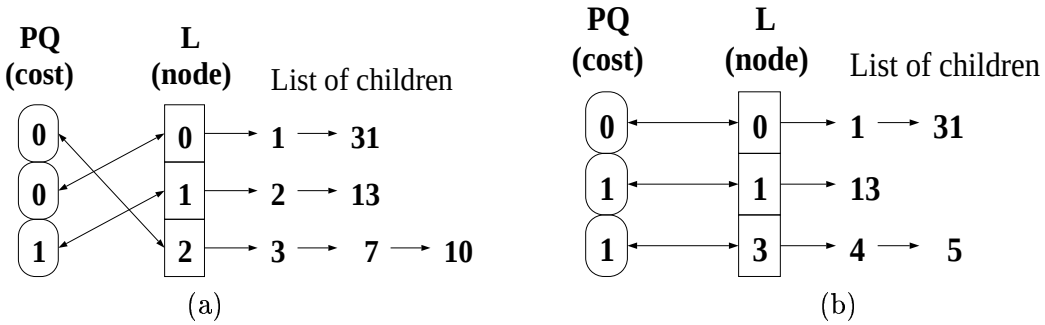


Figure 5: Illustrating the use of L and PQ for the tree of Figure 1.

We use the tree given in Figure 3 to demonstrate this process. After node 2 has been explored, L and PQ are as shown in Figure 5(a). Node 3 is explored and node 3 is to be inserted with a child-list containing two nodes. 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 Figure 5(b). After node 5 is explored, the child-list of node 3 is empty and node 1 is considered for finding the next node to tune into. Step (1) of FindNextNode returns node 13 of node 1's child-list. The addresses of nodes 13 and $next(3) = 7$ are used to determine that 7 is the next node to tune into.

Adding a node with k productive children costs $O(k)$ time for list L and $O(\log m)$ time for queue PQ. The update costs are $O(1)$ per update for list L and $O(\log m)$ time for queue PQ. Observe that one could use a balanced tree structure built on the node id's and augmented

with a cost entry. This would support all operations needed in $O(\log m)$ time. However, we get the same asymptotic time and a simpler implementation using a list and a priority queue (implemented as a heap).

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 an exploration-based cost metric.

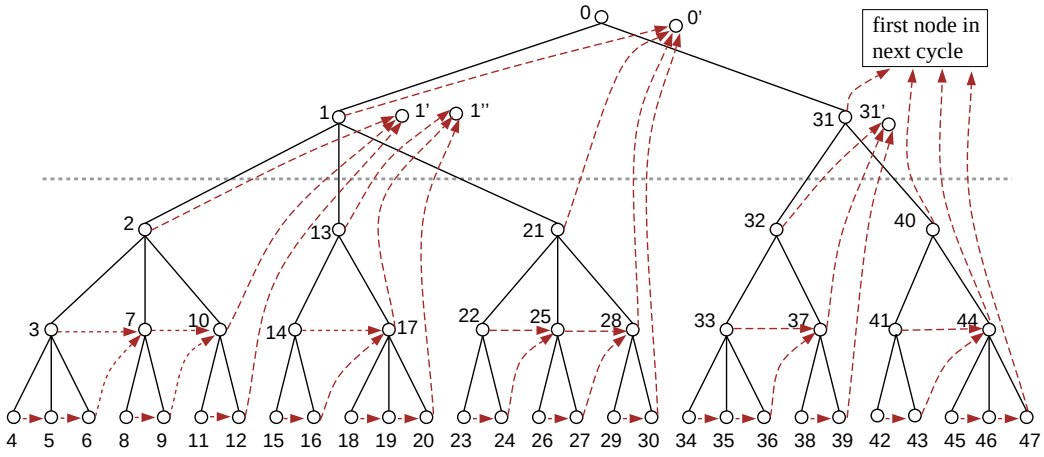
3.3 Broadcasting with repetition

Repetition can be used to improve broadcast performance, as demonstrated in [5, 7]. The algorithm described in this section repeats selected nodes within a broadcast cycle. The repetition of nodes increases the cycle length and can thus increase the latency. Our experimental work shows that the use of repetition in the broadcast schedule can result in clients experiencing smaller tuning time compared to schedules without repetition.

The tree is scheduled for broadcast using the following format. Let l be a parameter selected by the server, $1 \leq l < h$. Note that the root of tree T is on level 0 and the leaves are on level $h-1$. 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. Next, consider the remaining nodes by decreasing level numbers, starting with level $l-1$. Let u be such a node and let $v_1, ..., v_k$ be its children. The schedules for the subtrees rooted at $v_1, ..., v_k$ have already been generated. Let S_i be the schedule of the subtree rooted at child v_i, $1 \leq i \leq k$. Then, the schedule for the subtree rooted at node u is $uS_1uS_2u...uS_k$. The schedule can be viewed as a combination of an Euler tour on the top l levels and preorder traversals on the remaining subtrees [3]. The schedule is similar to the distributed indexing method described in [5] and used for single root-to-leaf searches of broadcasted data files.

Figure 6 shows a schedule for $l = 2$. The number of times a node is repeated is equal to its number of 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. For the tree in Figure 6 this comes to 7 nodes.

Every node in the broadcast schedule has one additional entry, which we again call *next*.



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

Figure 6: Tree and broadcast schedule with node repetition for $l = 2$; pointers in the tree indicate *next*-entries.

Entry $next(v)$ is again defined as the address of the node in the broadcast after all descendents of v have been broadcast. However, $next(v)$ can now be the address of a repeated copy of a node. Let u be a node on level $l-1$ of T. The path from u to the rightmost leaf in its subtree is called the *rightmost path*. For nodes on levels l or larger and not on a rightmost path, the *next*-entries point to non-repeated nodes. For all other nodes, the next-entries point to repeated nodes. For example, in Figure 6, $next(13)$ is the address of the third copy of node 1 (indicated as $1''$) and $next(3)$ is the address of the only copy of node 7.

Nodes to be explored are handled similarly as in Section 3.1: they are inserted into queue 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 tunes into node $next(u)$ and then explores $next(u)$. New nodes to be explored are inserted into queue Q. Tuning in at repeated nodes allows a client to restore lost entries and leads to a reduction in the tuning time. The ideal situation occurs when the value of l matches the memory size of a client so that nodes without repetition are never deleted. This happens when $\lfloor \frac{m}{B} \rfloor + 1 = h - l$. When a client can stored repeated nodes, the client experiences none of the benefits from the repetition,

only a disadvantage in terms of a larger latency. When a client is forced to delete nodes broadcast without repetition, the performance is as for broadcast schedules without repetition. Our experimental results clearly indicate this behavior. Observe that the relationship between $\lfloor \frac{m}{B} \rfloor$ and $h - l$ determines which case applies.

In the following, we compare the repetition-based broadcast with the algorithm of Section 3.1. Consider the scenario in which u is a productive node having B children. Assume that p of these children are productive nodes. Once the current copy of node u has been lost from the queue, we access its next copy in the broadcast. This next copy allows us to determine the productive children of u. If node u is lost B times, the broadcast tunes in for each of the B copies of u. Compared to the situation when no node is lost, the replication algorithm spends additional $O(B^2)$ time ($O(B)$ per lost instance of node u) on "rediscovering" the productive children. The algorithm tunes in at B additional nodes (the copies of u). Consider now the algorithm of Section 3.1. Assume node u is lost and its p productive children are determined using next-fields. The client will tune in for every child of u. This means it tunes in at $B - p$ unproductive nodes and the increase in the tuning time is proportional to the number of unproductive children of u. If each child of u has B children itself, the computation cost is $O(B^2)$. If u had not been lost, it would only be $O(pB)$.

In this worst-case example, repetition does not reduce the tuning time and it increases latency as wells as computation time for the client. Intuitively, one expects repetition of nodes to reduce the tuning time. The advantage of repetition is that even after a node has been lost, we can recover the information about the children. In some sense, we are able to restore data lost. As will be discussed in more detail in Section 4, node repetition results in small tuning time when at least half the levels of T have repeated nodes (i.e., $l > h/2$).

4 Experimental Results

Our experimental work focuses on the execution of range queries on broadcasted R^*-, R-, and quad-trees. We compare the performance of four algorithms: Algorithms **NoInfo**, **Next**, **Double**, and **Repeat-l**. Algorithm **NoInfo** assumes the tree is broadcast in preorder without additional entries and without node repetition. A mobile client maintains nodes to be explored as long as it has memory available. Once nodes are lost, the client tunes in at every node until

information accumulated allows again a more selective tuning. We include this approach to highlight the gain in terms of latency and tuning time for the other algorithms. Algorithm **Next** corresponds to the solution described in Section 3.1. Algorithm **Double** corresponds to the priority-based approach described in Section 3.2. In Algorithm **Repeat-l** the first l levels are broadcast with repetition as described in Section 3.3.

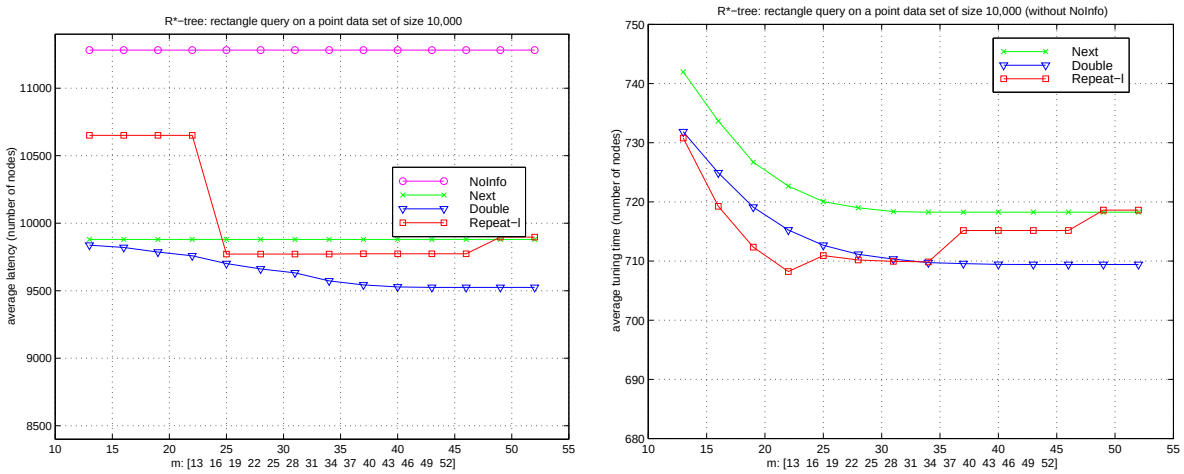
We considered trees having between 5,000 and 150,000 leaves and a fanout B (i.e., number of children) between 4 and 30. Page sizes used ranged from 128 to 512 bytes. Data stored at the leaves corresponds to either points or rectangles. Data is created either by using a uniform distribution or using the 2000 TIGER system of the U.S. Bureau of the Census. For random point data, points were generated using a uniform distribution within the unit square. For random rectangle data, the centers of the rectangles were generated uniformly in the unit square and the sides of the rectangles were generated with a uniform distribution between 10^{-5} and 10^{-2}. For 2000 TIGER data, we used data files on counties in U.S. 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 dynamically (i.e., through the insertion of points or rectangles) or statically. We refer to the static creation as the packed version [6]. The R^*-trees used in Section 4.1 are created dynamically. 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 do not have to be created through the insertion of points, but can be created in one step. Such packed trees have a smaller number of total nodes and utilize the space in the nodes in a better way (i.e., almost all nodes have B children). In Section 4.2, we discuss the differences in performance between packed and dynamic 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 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 [1]. We consider both synthetic and real data on points as

well as on rectangles.



(a) (b)

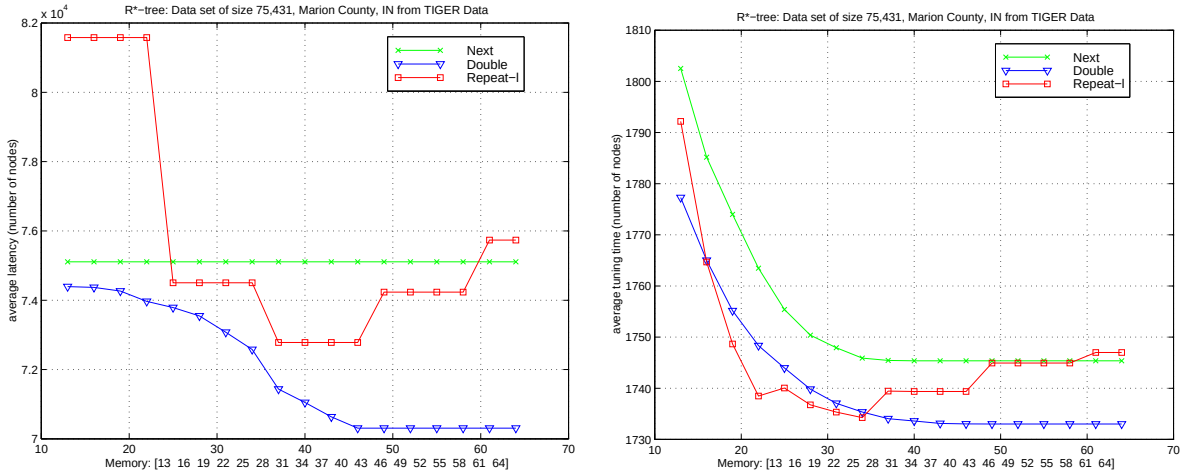
Figure 7: Latency and tuning time for an R^*-tree with $B = 12$ and 10,000 leaves; $n = 11,282$ and $h = 6$.

Our first set of experiments, illustrated in Figure 7, uses an R^*-tree with 10,000 leaves corresponding to random points and $B = 12$. The tree has a height of 6 and 11,282 nodes. The results shown are for a fixed tree and the 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.

Figure 7 shows a typical 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 address 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 point of the query and the length of time a query continues executing when no more relevant data is to be found. Algorithm Double generally achieves the best latency. A reason for this lies in the fact that Algorithm Double is not so likely to delete "expensive' nodes. For the latency this means nodes whose loss results in extending the time for recognizing the termination of the query. For the tuning time this means nodes whose loss results in unproductive nodes to be explored.

To maintain expensive nodes, Double uses a priority queue and an ordered list structure. For Algorithm Repeat-l shown in Figure 7 we have $\lfloor \frac{m}{B} \rfloor = h - l$; i.e., a mobile client is able to store non-repeated nodes. The relationship between l and m is explored further later in the section.

For the tuning time shown in Figure 7(b) we omit Algorithm NoInfo. It averages 6,000 nodes and changes little as memory size changes. The tuning times of the other algorithms show the impact of the optimizations done by the algorithms. Algorithm Next 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 Next.



(a) (b)

Figure 8: Latency and tuning time for rectangle queries on an R^*-tree with $B = 12$ and 75,431 leaves corresponding to road segments in Marion County, Indiana.

Figure 8 shows latency and tuning times for rectangles generated from line segments associated with roads the in Marion County, Indiana. The underlying data is shown in Figure 9. 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 random data sets.

The remainder of this section focuses on random data sets. However, the conclusions hold for all the Tiger data sets we considered. We start with a comparison of the effect of different starting times of a query within a cycle. Figure 10 compares the number of unproductive nodes

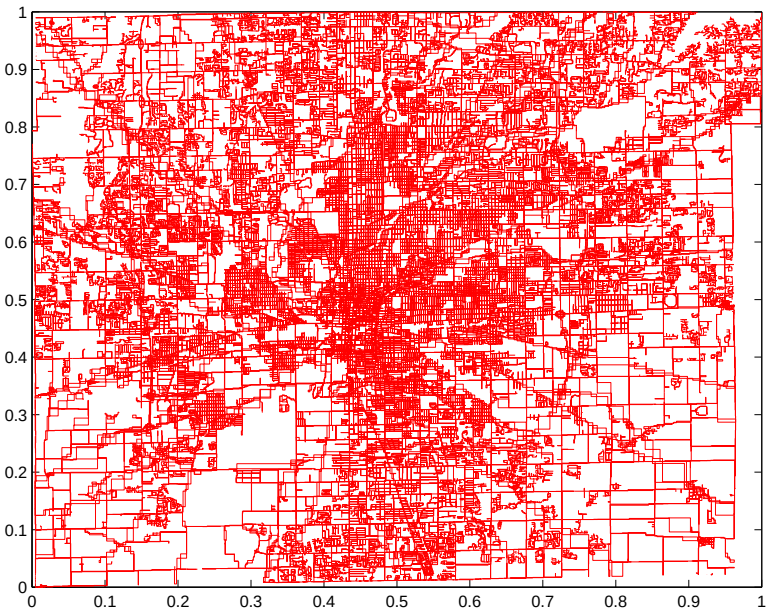
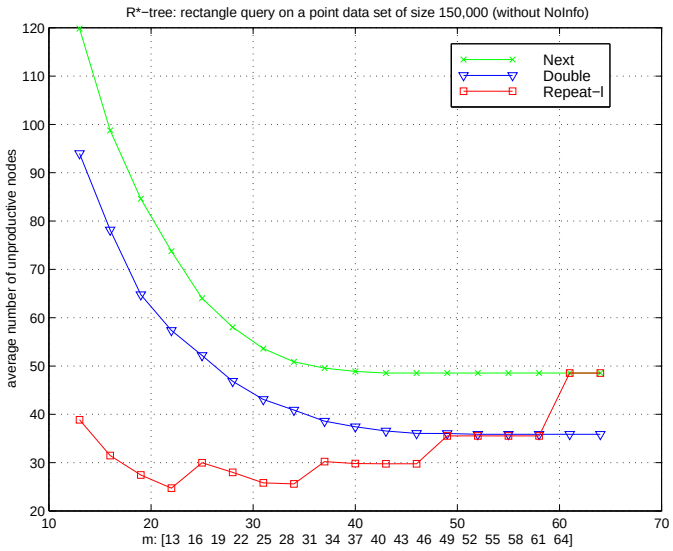
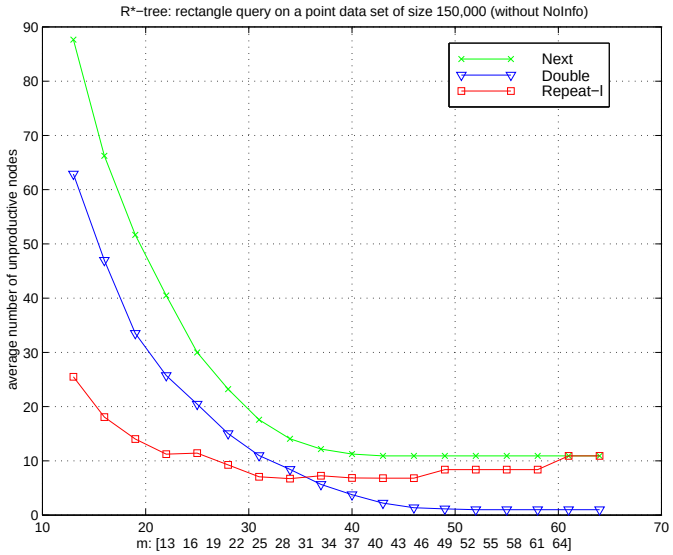


Figure 9: 2000 TIGER data on roads in Marion County, Indiana.

for queries issued in (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 (it is always bounded by the length of one cycle). The figure echos and amplifies the trend already observed for the tuning time and the number 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 Double performs well especially when m is large. For large m, Algorithm Double performs well not because it keeps nodes whose loss is expensive (no algorithm loses many nodes for large m), but because Double 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 Double and Next. In summary, Figure 10 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.



(a) Queries issued at first leaf in the broadcast cycle



(b) Queries issued at root (i.e., first node) in the broadcast cycle

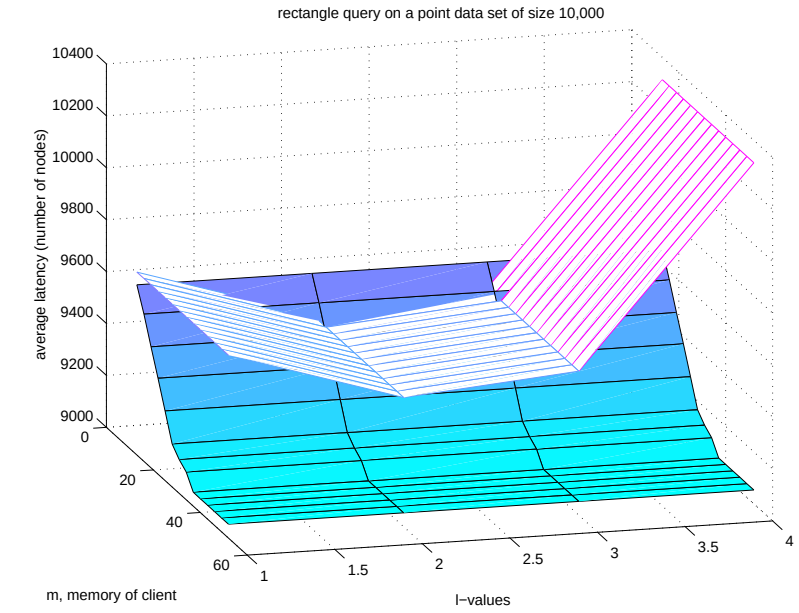
Figure 10: Unproductive node comparison for an R^*-tree with $B = 12$ and 150,000 leaves.

The discussion on Algorithm Repeat-l so far considered the case when $\lfloor \frac{m}{B} \rfloor + 1 = h - l$. In this situation the repetition of nodes in the schedule matches the client's memory size; i.e., a client has enough memory to store addresses of nodes to be explored in the future and broadcast without repetition. Next, we explore the relationship between l and m on the performance of Algorithm Repeat-l. The discussion uses a point data set of size 10,000. The queries executed by the client are rectangles having sides between 0.001 and 0.25. Figure 11 gives the latency and the tuning time for different m and l values. Algorithm Repeat-l corresponds to the white (non-filled grid). Note that for $l = 0$, Repeat-l behaves like Algorithm Next (and is not shown). Results shown are $1 \leq l \leq h - 1$. For the sake of better comparison, Algorithm Double is included. Note that Double is independent of the l-values chosen by the server for Repeat-l.

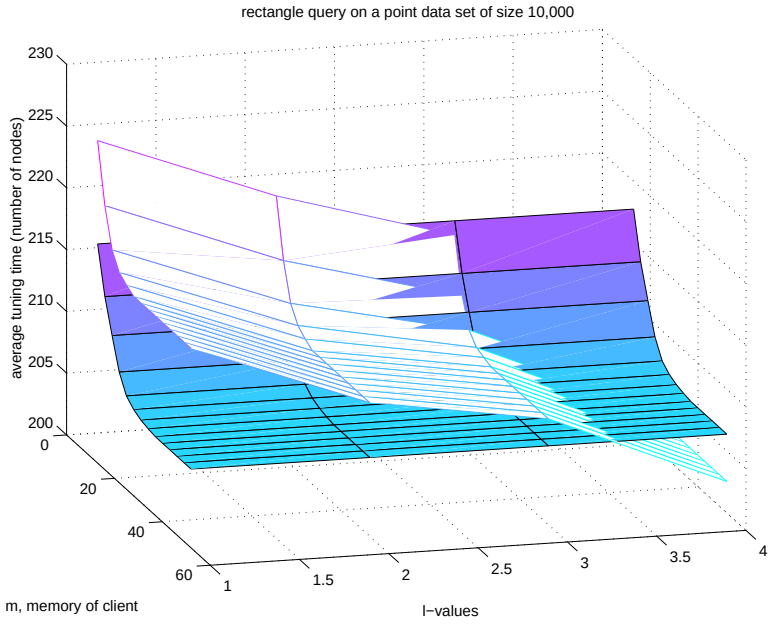
As l increases, the cycle length of the tree broadcast for Repeat-l increases. Increased cycle length translates to higher latency. The latency is also influenced by the ability of a query to detect termination. In Figure 11(a) we observe both of these properties. As l increases, the latency increases 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 increases somewhat for $l = 2$, the increase does not yet impact the latency as much as for larger values of l. For Algorithm Repeat-l, the tuning time decreases as l increases, as shown in Figure 11(b). In addition, the tuning time decreases for both Repeat-l and Double as a client's memory size increases. From our experimental results we can conclude that for $l \geq h/2$, Repeat-l achieves a better tuning time than Algorithm Double. When considering the latency, Double is always better than Repeat-l.

4.2 Packed R-trees versus dynamic R^*-trees

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 a dynamically created R^*-tree versus broadcasting a packed R-tree. We use the technique proposed in [6] 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 [6], we select the one recommended as the best and which sorts the data by Hilbert values. When the data



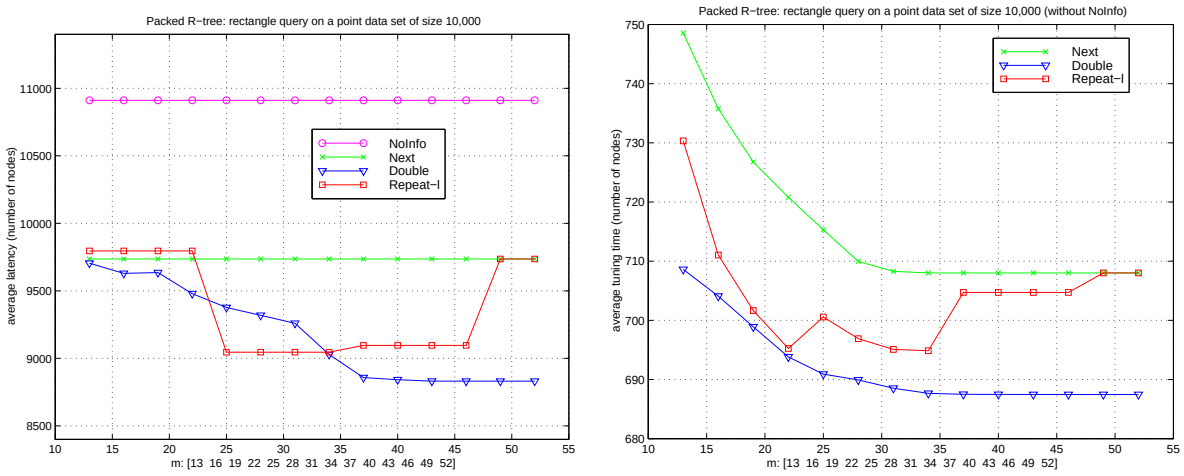
(a) Latency.



(b) Tuning time.

Figure 11: The latency and tuning time for Algorithm Repeat-l (non-filled grid) and Algorithm Double (filled grid) with different m- and l- values for an R^*-tree of height h=6 and having 10,000 leaves.

are rectangles, we sort the rectangles according to the Hilbert value of the center point of the rectangles.



(a) (b)

Figure 12: Latency and tuning for a packed R-tree with $B = 12$ and 10,000 leaves; $n = 10,911$, $h = 5$.

Figure 12 shows the latency and the tuning time for a packed R-tree, using the same data as in Figure 7. The fanout is fixed to 12. The packed R-tree has a height of 5 and 10,911 nodes, compared to a height of $h = 6$ and $n = 11,282$ for the dynamically created R^*-tree. 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 for packed trees is as observed for dynamically created trees. Algorithm Double achieves the best overall performance. The improvement in the latency compared to the dynamic tree creation is due to the smaller number of nodes in the packed tree. Figure 13 compares the number of unproductive nodes between the packed and the dynamic tree. 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 a more efficient "searching" for productive nodes. In summary, a packed tree does achieve a smaller latency. The improvement in the tuning time is not significant.

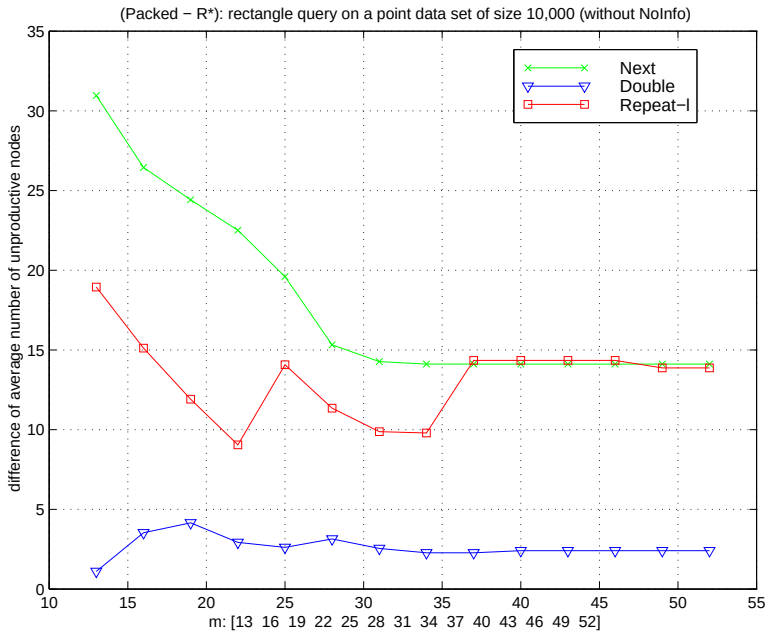
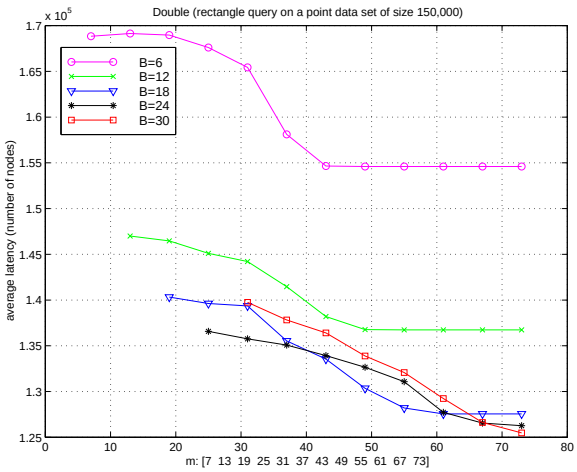


Figure 13: Differences of the number of unproductive nodes for the three algorithms using Packed R-tree and the three algorithms using R^*-tree.

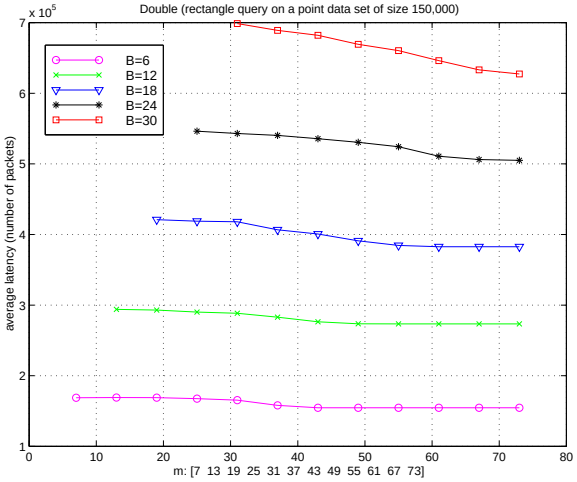
4.3 Packet and index node size

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. We consider 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 a node uses at least one packet and 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 (a node with fanout $B = 6p$ corresponds to p packets).

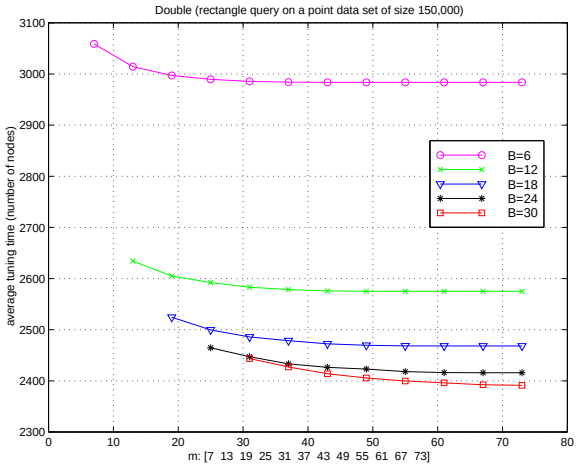
Figure 14 shows the results of the node-based and packet-based metrics for R^*-trees on Algorithm Double. The trees are created dynamically for a data set of 150,000 points. The



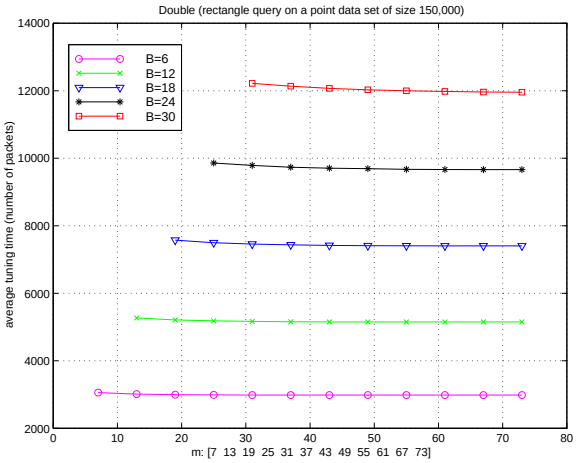
(a.1) Lateny (number of nodes)



(a.2) Latency (number of packets)



(b.1) Tuning time (number of nodes)



(b.2) Tuning time (number of packets)

Figure 14: Latency and tuning time based on nodes and packet metric for 5 different B-values. Results shown are achieved by Algorithm Double on an R^*-tree with 150,000 leaves.

sizes of the trees in terms of the number of nodes are 194,117 for $B = 6$, 169,243 for $B = 12$, 162,558 for $B = 18$, 159,212 for $B = 24$, and 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, performance for the latency as well as tuning time is predictable and changes only slightly as the memory size increases. In summary, the fanout of the broadcasted index tree should be tailored towards the packet size of the mobile environment as well as the memory size of the clients.

5 Conclusions

We considered the efficient execution of queries on broadcasted R- and R^*- trees, focusing on the generation of the broadcast schedule and the query algorithm executed by a mobile client. Efficiency is measured in terms of latency and tuning time experience by a client while not exceeding given memory constraints. We presented three client algorithms that differ on how a mobile client decides which data to delete when no more additional data can be stored, the degree of repetition of nodes in the broadcast, and the type of data structures employed. Our experimental results show that these algorithms lead to different tuning times and latencies and 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. Our results also showed that using a packed index tree compared to a dynamically created one has little impact on the performance, while tailoring the fanout of a node to the packet size in the wireless environment can lead to considerable improvement.

References

[1] N. Beckmann, H. Kriegel, R. Schneider, and 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*, pages 322–331, May 23-25 1990.

[2] M.-S. Chen, P.S. Yu, and K.-L. Wu. Indexed sequential data broadcasting in wireless mobile computing. In *Proceedings of the 17-th International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society Press, May 1997.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[4] T. Imieliński, S. Viswanathan, and B. R. Badrinath. Energy efficient indexing on air. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the International Conference on Management of Data*, pages 25–36. ACM Press, May 1994.

[5] T. Imieliński, S. Viswanathan, and B. R. Badrinath. Data on air: Organization and access. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):353–372, May/June 1997.

[6] I. Kamel and C. Faloutsos. On packing r-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM)*, November 1993.

[7] S.-C. Lo and A.L.P. Chen. An adaptive access method for broadcast data under an error-prone mobile environment. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):609–620, 2000.

[8] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[9] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.