Query Processing in Broadcasted Spatial Index Trees*

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

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907, USA
{seh,liucm,aref,sunil}@cs.purdue.edu

Abstract. The broadcasting of spatial data together with an index structure is an effective way of disseminating data in a wireless mobile environment. Mobile clients requesting data tune into a continuous broadcast only when spatial data of interest and relevance is available on the channel and thus minimize their power consumption. A mobile client experiences latency (time elapsed from requesting to receiving data) and tuning time (the amount of time spent listening to the channel). This paper studies the execution of spatial queries on broadcasted tree-based spatial index structures. The focus is on queries that require a partial traversal of the spatial index, not only a single-path root-to-leaf search. We present techniques for processing spatial queries while mobile clients are listening to a broadcast of the tree. Our algorithms can handle clients with limited memory, trees broadcast with a certain degree of replication of index nodes, and algorithms executed at the clients may employ different data structures. Experimental work on R*-trees shows that these techniques lead to different tuning times and different latencies. Our solutions also lead to efficient methods for starting the execution of a query in the middle of a broadcast cycle. Spatial query processing in a multiple channel environment is also addressed.

1 Introduction

The broadcasting of spatial data together with an index structure is an effective way of disseminating data in a wireless mobile environment [2–4, 6]. Mobile clients requesting data tune into a continuous broadcast only when spatial data of interest and relevance is available on the channel and thus minimize their power consumption. A client experiences latency (the time elapsed from requesting to receiving data) and tuning time (the amount of time spent listening to the channel). This paper studies the execution of spatial queries by mobile clients on broadcasted tree-based spatial index structures. The focus is on queries that require a partial traversal of the spatial index, not only a single-path root-to-leaf search. Examples of such queries arise in R-trees, R*-trees, quad-trees, or k-d-trees [8, 9].

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

Assume that an n-node index tree is broadcast by a server. The 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 replication 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 appropriate times and traverse parts of the broadcasted tree. We assume that mobile clients operate independently of each other. For the quadtree, k-d-tree, R-tree, and R*-tree indexes, we consider the execution of range queries that determine all objects containing a given query point or overlapping a given query rectangle. Typically, in a range query, more than one path from the root to leaves is traversed. Hence, a partial exploration of the tree is performed. This feature distinguishes our work from related papers which consider only root-to-leaf searches [2, 3].

Traversal and partial traversal of an 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 addresses of nodes to be tuned into later. However, a client may have to store information of hB nodes, where h is the height of the index tree and B is the maximum number of children. When memory is limited or a query starts executing during the on-going broadcast cycle, performance depends on what information is maintained at a client and how the client makes use of the information. In Section 3 we propose three different solutions for dealing with mobile clients having limited memory. Our solutions differ on how a mobile client decides which data to delete, the degree of replication of nodes in the broadcast, and the type of data structures employed by the clients. Our experimental work on real and synthetic data sets is discussed in Section 4. Our results show that our methods lead to different tuning times and somewhat different latencies. We also 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 cycle. Observe that starting in the middle of a cycle corresponds to having lost all previously read information.

In Section 5 we consider query processing of broadcasted index trees in a 2-channel environment. We assume that a mobile client can tune into either of the channels at each time instance, but that the channel number tuned into needs to be specified. The use of multiple channels can lead to a significant reduction in the latency. We present two heuristics which differ in the type of conflict that can happen when a query is executed and the bounds on the cycle length of the generated schedule.

2 Assumptions and Preliminaries

Our solutions do not make any assumption on the structure of the tree broadcast. Trees can range from highly structured (like R^*- or B-trees) to random trees. We use B to denote the 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 a list of v's children. In addition, the addresses of children in the broadcast schedule are included.

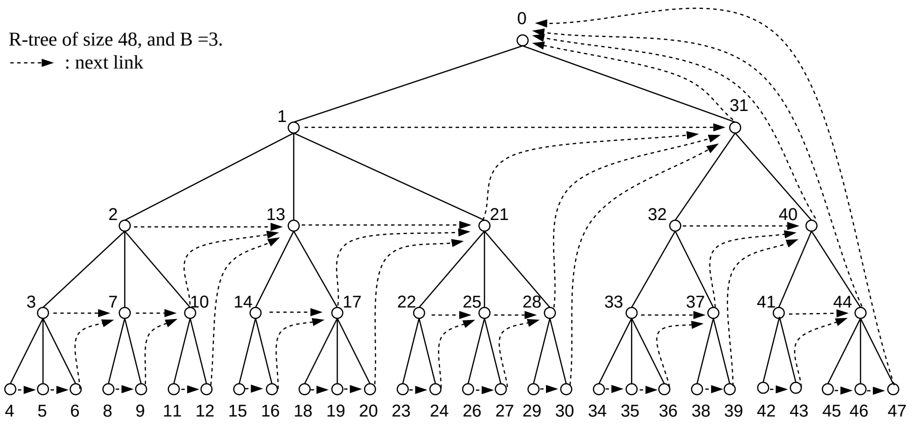


Fig. 1. A tree and its next entries

The broadcast of one instance of the tree is called a *cycle*. When the tree is broadcast in a single channel environment, the cycle length is at least cn, for some constant $c \geq 1$. We assume throughout that a broadcasted tree obeys the *ancestor property*: when node v is broadcast for the first time in a cycle, the parent of v was broadcast earlier in the same cycle. Hence, the root is always the first node broadcast. We say a client *explores* node u when the client tunes into the broadcast to receive u and examines the entries about u and u's children. When none of u's children needs to be explored further, u is an *unproductive node*, otherwise u is called a *productive* node.

An objective of our work is to minimize two parameters: the *tuning time* which counts the number of index nodes explored during one query and the *latency* which counts the total number of nodes broadcast by the scheduler during the execution of the query. Minimizing the number of unproductive nodes corresponds to minimizing the tuning time.

As already stated, the scheduler may add additional entries to the broadcasted nodes. One such entry used in our solutions is $next(v)$: $next(v)$ 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.

An index node is viewed as one packet in the broadcast. An alternate way of measuring would be to fix the packet size and count the number of packets tuned into (which now contain a number of index nodes). We find an evaluation based on the number of index nodes more meaningful for comparing our strategies.

Having to tune in for a node is more relevant to the performance than the number of bytes received for a node.

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 s, $s < hB$. 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 may not be able to store all relevant information received earlier and losing data can increase the tuning time.

The algorithms of Sections 3.1 and 3.2 assume that the index tree is broadcast in preorder, without replication (i.e., every node appears once in the broadcast cycle), and with a *next*-entry for every node. The two algorithms differ on how they decide what entries to delete when required to free up memory. The index tree is generated without knowing s and thus different clients can have different memory sizes. Only the program executed by the client is tailored towards its s. The algorithm described in Section 3.3 assumes the index tree is broadcast with node replication. The amount of node replication is determined by s, the size of the memory. A client with a memory size smaller than the chosen s does not get the advantage of node replication. Additional entries can be added to the broadcasted index tree to allow better performance for such clients.

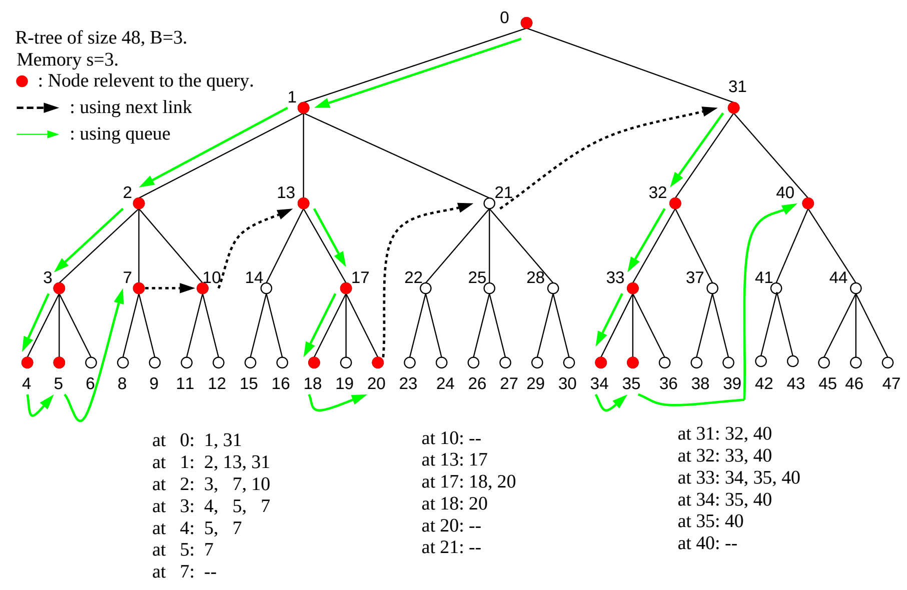
3.1 Using Next Fields

We assume that every mobile client maintains a queue Q. The queue is initially empty and at any time it contains at most s entries. We assume that deletions can be done on both ends of the queue. Assume the mobile client started the execution of a query and is tuning in to receive and explore node v. If v is a productive node, then let $v_1, ..., v_k$ be the children of v to be explored. Nodes $v_1, ..., v_k$ are put into Q by the order of their position in the broadcast. For each node v_i, we record the address of v_i in the broadcast as well as entry $next(v)$. Should Q become full, nodes are deleted using FIFO management. After the exploration of v and possibly its children, the next node to be explored is either found in the queue or in entry $next(v)$. A high level description of the algorithm executed by a client is given in Figure 2. Figure 3 shows the traversal of a tree when the query needs the data stored in leaves $4, 5, 18, 20, 34$ and 35. For $s = 3$, next links are used four times.

Exploring a node v having k children costs $O(k)$ time. Assume v is unproductive. If the queue is not empty, then deleting the most recently added node from the queue gives the next node to tune into. This node had a productive parent, but it can be productive or unproductive. When the queue is empty, we tune in at $next(v)$. This node may have had an unproductive parent. For a given query, there exist scenarios for which a client tunes in at $\Theta(B^{h-\frac{s}{B}})$ unproductive nodes.

```
**Algorithm ExploreQ($v$)**
(1) **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 of $v$ to be explored, arranged
            in the order they appear in the broadcast;
            insert children $v_1, v_2, \ldots, v_k$ into queue $Q$ of size $s$ in order of broadcast,
            deleting using first-in/first-out rule;
            ExploreQ($v_1$)
        **endif**
    **endif**
(2) **if** queue Q is empty **then**
        ExploreQ($next(v)$)
    **else**
            let $w$ be the node returned when deleting the most recently added
            node from $Q$;
            ExploreQ($w$)
    **endif**
**End Algorithm ExploreQ**
```

Fig. 2. Client algorithm for query execution during a tree broadcast with next fields



at 0: 1, 31
at 1: 2, 13, 31
at 2: 3, 7, 10
at 3: 4, 5, 7
at 4: 5, 7
at 5: 7
at 7: --

at 10: --
at 13: 17
at 17: 18, 20
at 18: 20
at 20: --
at 21: --

at 31: 32, 40
at 32: 33, 40
at 33: 34, 35, 40
at 34: 35, 40
at 35: 40
at 40: --

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

When a mobile client tunes into the broadcast at an arbitrary time during the cycle, it starts executing the program with an empty queue. The client needs to remember the first node obtained after tuning in. Let f be this node. When node f is received again in the next broadcast cycle, the query terminates (if it did not already terminate). Hence, the latency experienced is at most one cycle length. Next-fields 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.

3.2 Using an exploration-based cost function

The previous algorithm uses queue Q to hold the most recent nodes identified by the client as index nodes to be explored. This section describes a solution that 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 removed from the queue. The tree is scheduled for broadcast as in the algorithm of Section 3.1; i.e., we assume that every node in the broadcast has a next-entry.

We discuss the algorithm using the implementation underlying our experimental work. In this implementation, we use two queues: a queue Q in which nodes are added in the order they are encountered during the broadcast and a priority queue PQ in which nodes are arranged according to *cost*-entries, along with pointers between entries for a node in the two queues. The reason for using two queues is simplicity. 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 $O(\log s)$ time. However, we get the same asymptotic time and a simpler implementation using two queues, with PQ implemented as a heap.

The entry for node v in Q contains the following:

- node v's id
- the list of 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 Q with k children, 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 queue Q.

When a mobile client tunes in for node v, it explores node v. When v is productive with children $v_1, ..., v_k$, we proceed as described in Algorithm ExploreDQ given in Figure 4. Observe that when there is not enough memory to add node v and its k children to the queues, we delete nodes along with the information of children to be explored based on *cost*-entries. Algorithm FindNextNode describes how the next node to be explored is determined. In two situations the next node explored is $next(v)$, namely when queue Q is empty or when v's parent is no longer in the queue PQ. When Q is not empty and v's parent is still

```
**Algorithm ExploreDQ($v$)**
(1) **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) compute $cost(v)$;
            (1.2) insert $v$ into $PQ$ with $cost(v)$;
            (1.3) **while** not enough space for $v$ and its $k$ children in the queues **do**
                    (a) determine node $u$ in $PQ$ having minimum cost;
                    (b) delete entries related to $u$ from the queues
                    **endwhile**
            (1.4) insert $v$ and $v$'s child list into $Q$
        **endif**
    **endif**
(2) $u$ = FindNextNode($v$);
(3) ExploreDQ($u$)
**End Algorithm ExploreDQ**

**Algorithm FindNextNode($v$)**
**if** queue Q is empty **then**
    **return** $next(v)$
**else**
    let $u$ be the node returned when deleting the most recently added node from $Q$;
    **if** $u$ is not the parent of $v$ **then**
        **return** $next(v)$
    **else**
        delete $v$ from the child list of $u$ and update $cost(u)$;
        **if** the child list of $u$ is not empty **then**
            **return** the first node in the child list of $u$
        **else**
            (1) delete entries related to $u$ from the queues;
            (2) FindNextNode($u$)
        **endif**
    **endif**
**endif**
**End Algorithm FindNextNode**
```

Fig. 4. Client algorithm for query execution during the tree broadcast using next fields and two queues.

present, the next node to be explored is determined from the child list of v's parent, as described in Figure 4.

Adding a node with k productive children to the queues costs $O(k)$ time for queue Q and $O(\log s)$ time for queue PQ. The updates to the queues are $O(1)$ per update for queue Q and $O(\log s)$ time for queue PQ. Observe that cost-entries for nodes in PQ can only decrease.

When a client tunes into the broadcast cycle at some arbitrary point, the algorithm is initiated with empty queues. Like the previous algorithm, a client needs to remember 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 replication

Replication can be used to improve broadcast performance, as demonstrated in [4, 6]. The algorithm described in this section replicates selected nodes within a broadcast cycle. The replication of nodes increases the cycle length and can thus increase the latency. Again, assume that a client uses a queue of maximum size s. Our experimental work shows that replication achieves smaller tuning time for queries starting their execution in the middle of a cycle (in comparison to the two algorithms presented in the previous sections).

The tree is scheduled for broadcast using the following format. For each node that is the root of a subtree of height $\lfloor s/B \rfloor$, the broadcast contains the nodes in the subtree in preorder traversal with no replication. Consider a node u being the root of a subtree of height larger than $\lfloor s/B \rfloor$. Assume the children are $v_1, ..., v_k$ and the schedules for the subtrees rooted at these children 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 $uS_1uS_2u...uS_k$. Figure 5 shows such a schedule when the nodes of the first three levels are replicated (nodes on the last two levels appear exactly once in the broadcast cycle). The number of times a node is replicated is equal to its number of children. Hence, if the nodes on the first k levels are replicated, the total number of replicated nodes in the broadcast is equal to the number of nodes on the first $k+1$ levels of the tree minus 1. For the tree in Figure 5 this comes to 19 nodes.

All nodes in the broadcast cycle have one additional entry, *next_repl*. Entry *next_repl*(v) gives the address of the next relevant replicated node corresponding to the closest ancestor of v. Figure 5 shows the *next_repl*-entries in the array representing the schedule. For example, *next_repl*(13) is the third copy of node 1 for both copies of node 13. Observe that if queries were to start at the begin of the cycle and clients were to have the same amount of memory, *next_repl*-entries would have to be added only for the nodes on the replicated levels.

Nodes to be explored are handled as in Section 3.1: they are put into queue Q according to their arrival in the cycle and are deleted when space is need according to the FIFO rule. A subtree rooted at a node not replicated can be explored without a loss of productive nodes. A loss of productive nodes happens only for replicated nodes or when the query starts in the middle of the cycle.

When a node u has been explored and Q is empty, we tune at node $next\_repl(u)$. When this node arrives, we determine which of its children yet to arrive in the broadcast cycle need to be explored and put those into Q.

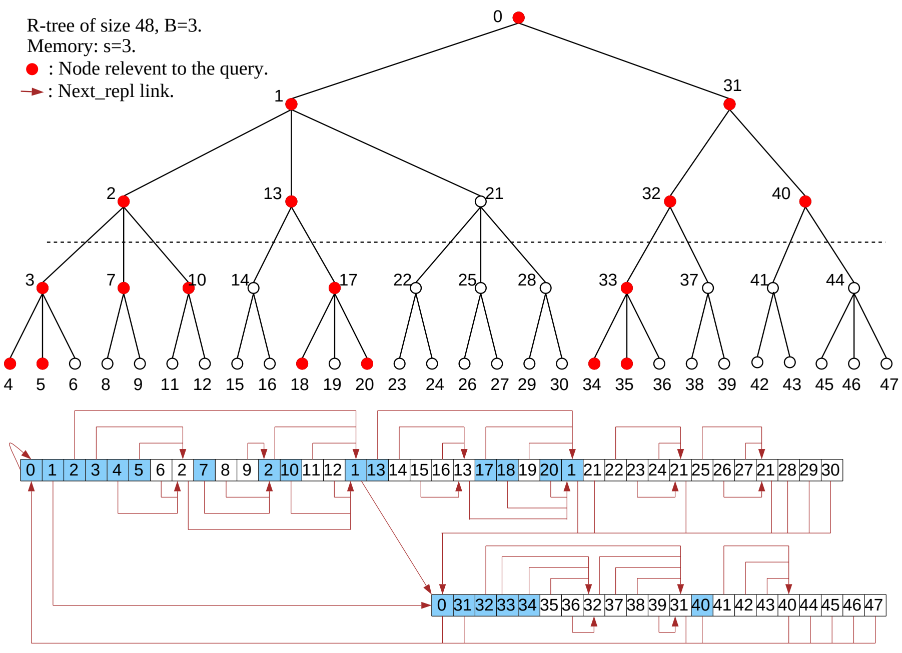


Fig. 5. Tree and broadcast schedule with node replication for $s = 3$; pointers in schedule indicate *next_repl*-entries.

In the following, we compare the replication-based broadcast with the algorithms using *next*-fields. 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 $O(pB)$.

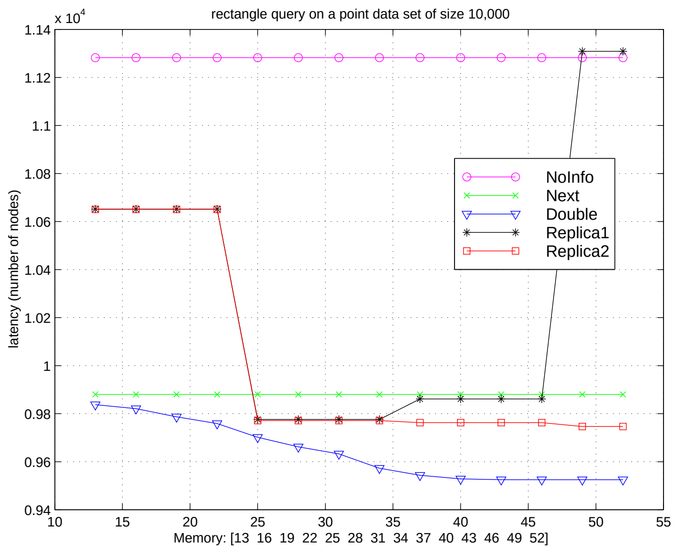
In this example, replication does not reduce the tuning time and it increases computation costs and latency. Intuitively, one expects replication of nodes to reduce the tuning time. An advantage of replication is that even after a node has been lost, we can recover the information about the children. In some sense, we are able to restore data lost. As will be discussed in more detail in Section 4, the replication approach results in small tuning time, but the latency depends on how queries starting within a cycle are handled.

4 Experimental Result for Limited Memory

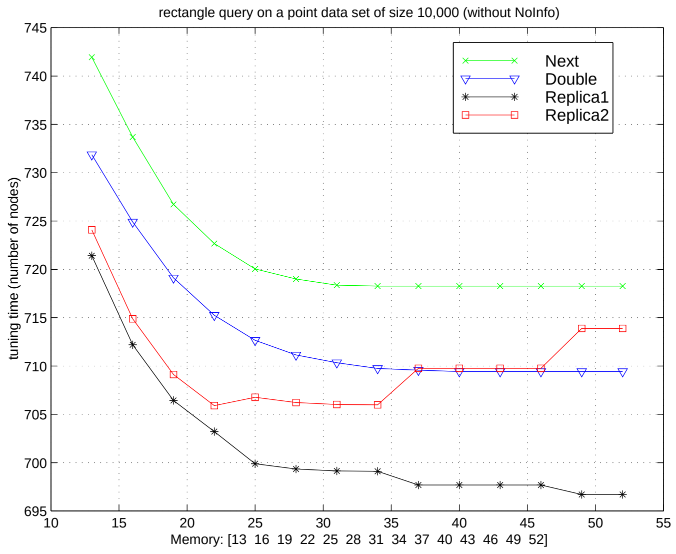
Our experimental work focuses on the execution of spatial range queries on broadcasted R^*-trees. We compare the performance of five algorithms: Algorithms **NoInfo**, **Next**, **Double**, **Replica1**, and **Replica2**. Algorithm **NoInfo** assumes the tree is broadcast without additional entries and without replication. A mobile client maintains nodes to be explored as long as it has memory available. When such nodes are lost, the client is forced to tune 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 four 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. We include two implementations of the replication approach: Algorithms **Replica1** and **Replica2**. In Replica1 every node is broadcast with its *next_repl*-entry and in Replica2 the *next*-entry is added to each node. The availability of *next*-entries leads to significantly better latencies when few nodes are replicated.

The R^*-trees were generated using code available from [1]. Trees were created through the insertion of points (resp. rectangles). We considered trees having between 5,000 and 150,000 leaves and a fanout (i.e., number of children) between 4 and 30. Page sizes used ranged from 128 to 512 bytes. Point and rectangle data were created either by using a uniform distribution or data from the 2000 TIGER system of the U.S. Bureau of Census. For R^*-trees based on 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; 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 Indiana and extracted line segments from the road information. These line segments were used to generate rectangles (enclosed minimum bounded rectangle) or points (center of line segment).

Our first set of experiments is for 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 data shown is 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. 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



(a)



(b)

Fig. 6. Latency and tuning time comparison for an R^*-tree with $B = 12$ and 10,000 leaves.

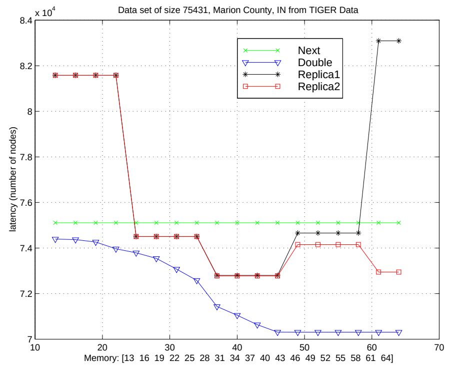
reported is the average of 100 queries and each query is issued at 50 different time points within a broadcast cycle.

Figure 6(a) shows a typical comparison of the latency and tuning time experienced by the five algorithms. The x-axis reflects increases in the memory. A memory of, say 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 consistently achieves the best latency. A reason for this lies in the fact that Algorithm Double is not so likely to delete nodes whose loss is "expensive". 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. The price for this is the maintenance of two queues by a mobile client. As expected, the two replication-based algorithms have a higher latency for small memory sizes. Note that for the graphs shown the s chosen by the scheduler is equal to the memory size of the client. As s increases, the latency of Replica2 behaves like that of Algorithm Next and the latency of Replica1 behaves like that of Algorithm NoInfo. This happens since Replica1 does not have $next$-entries needed to reduce the latency of a query issued within a cycle.

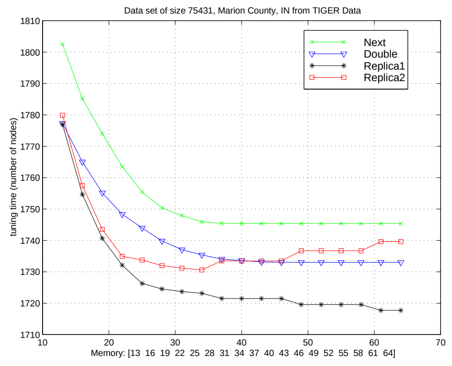
For the tuning time plot of Figure 6(b) we omit Algorithm NoInfo. It averages 6,000 nodes and changes little as memory size changes. The tuning times of the other four algorithms show the impact of the optimizations done by the algorithms. Algorithm Next has the highest tuning time (and thus the highest number of unproductive nodes). The tuning time for Replica2 reflects that as memory increases and replication decreases, the tuning time becomes identical to that of Next. The tuning time of Replica1 reflects that as memory increases and replication decreases, a query issued in the middle of the cycle may do very little processing in the on-going cycle. Completing the query in the next cycle results in low tuning time, but higher latency.

Figure 7 shows latency and tuning time for rectangles generated from line segments associated with roads in Marion County. The underlying data is shown in Figure 8. This non-uniform data shows the same trend and characteristics as discussed for random data sets.

The remainder of this section shows results obtained for random data sets. However, the conclusions hold for all Tiger data we considered. We start with a comparison of the effect of different starting times of a query. The results given are for an R^*-tree having 150,000 leaves (corresponding to points) and $B = 12$. Figure 9 compares the unproductive nodes for queries issued in (a) at the first leaf and (b) at the begin of the broadcast. The queries correspond to smaller rectangles than in the previous figures: the sides range from 0.001 to 0.25. The figure echos the trend already discussed for the different algorithms. Note that there is no difference in the tuning time between the two replication-based solutions when the query starts at the beginning of a broadcast cycle. Starting a query at a leaf results in higher tuning times and more unproductive nodes for



(a)



(b)

Fig. 7. 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.

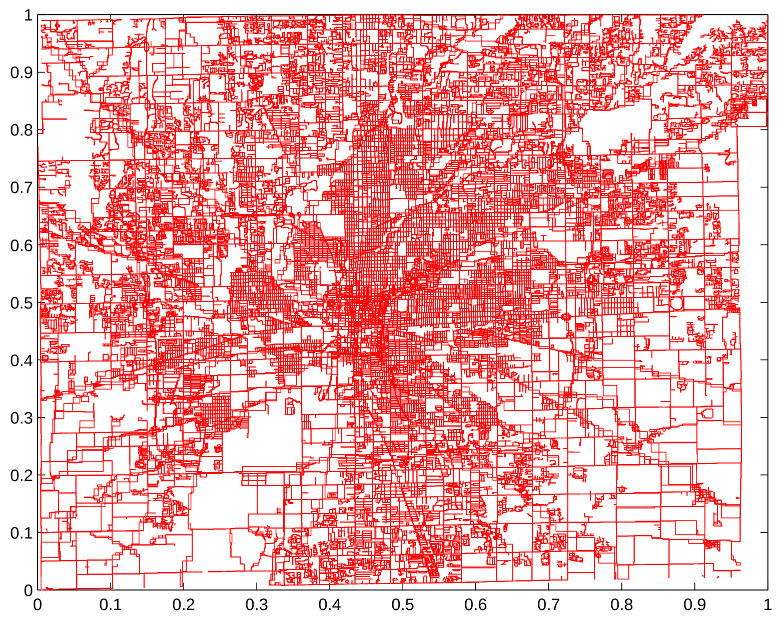
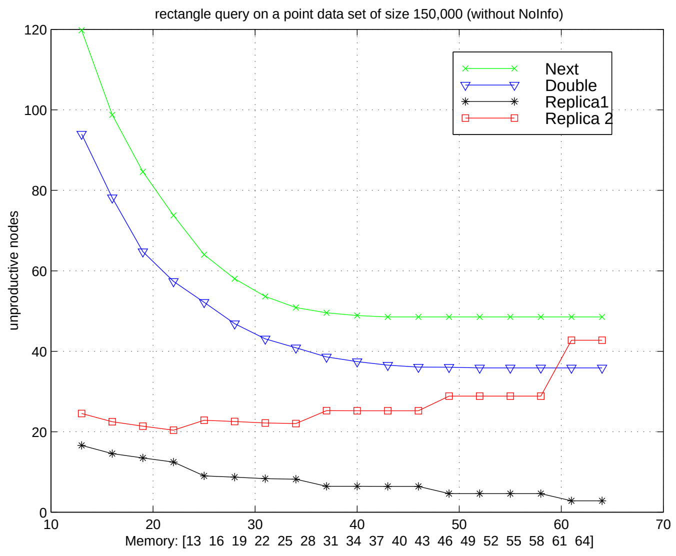
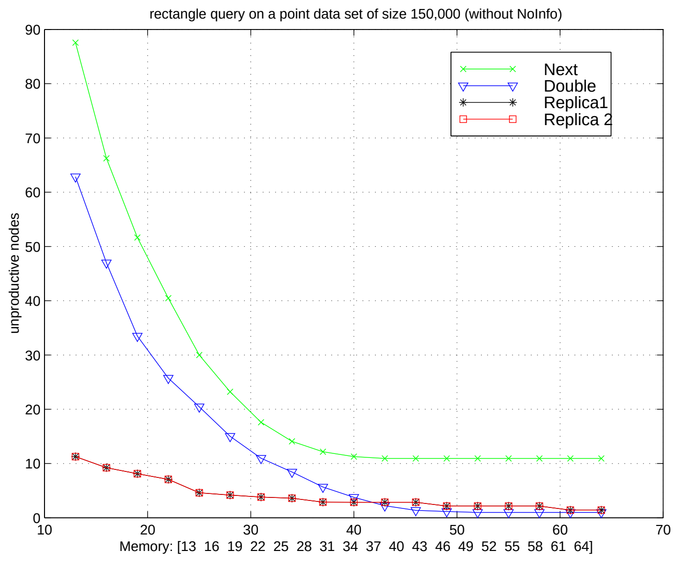


Fig. 8. 2000 TIGER data on roads in Marion County, Indiana.



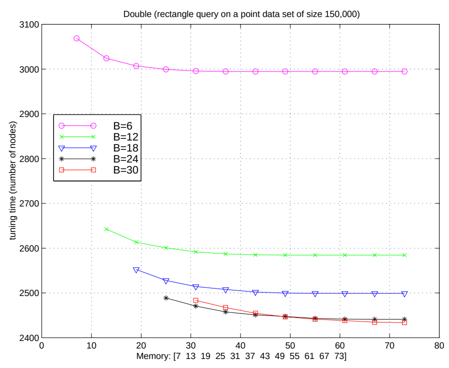
(a) Queries issued at first leaf in the broadcast cycle



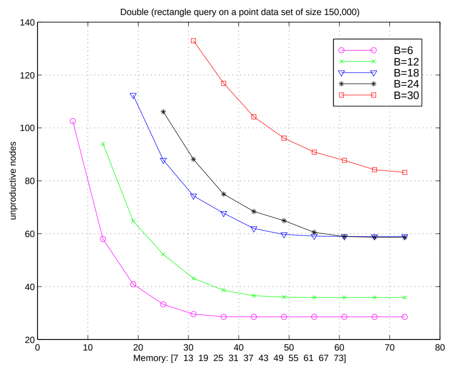
(b) Queries issued at root (i.e., first node) in the broadcast cycle

Fig. 9. Unproductive node comparison for an R^*-tree with $B = 12$ and 150,000 leaves.

all algorithms. As memory increases to the point that no (or very few) nodes to be explored are lost, the differences in the unproductive nodes among the four algorithms become evident. We observe that Replica1 achieves the best results, but one pays a price in the form of a higher latency. Algorithm Double performs well not because it keeps expensive nodes, but because it stores nodes together with its children (recall that a node in queue Q has a list of productive children). We point out that as the average size of the rectangle query decreases, the ratio of unproductive nodes to the tuning time increases.



(a)



(b)

Fig. 10. Tuning time and unproductive nodes for Algorithm Double for 5 different B-values on trees with 150,000 leaves.

We conclude this section with a comparison of different B-values for trees with a fixed number of leaves. When the broadcasted index tree is generated explicitly for the broadcast, the way the tree is formed should be influenced by what features of the tree result in better latency and tuning times. For all algorithms we observed that, as B increases, the tuning time decreases. At the same time, as B increases, we see an increase in the number of unproductive nodes. This behavior is observed independent of the size of the memory at the mobile client, as shown in Figure 10 for Algorithm Double.

5 2-Channel Broadcasting

The availability of multiple channels can lead to a significant reduction in the latency [7, 10]. The assumption is that a mobile client can tune into any one of the channels at each time instance and that the channel number tuned into needs to be specified. However, poorly designed algorithm for multiple channels may result in an increase in the latency and tuning time. In this section we present two methods for scheduling a spatial index tree in a 2-channel environment. The broadcast schedules we describe assume that a node is broadcast only once in a cycle (i.e., no node replication takes place). The two methods differ in the type of conflict that can happen when a query is executed by a client and the bounds on the cycle length of the generated schedule. The actions taken by a client when executing a query are extensions of the work described in Sections 3.

Before giving details on the two algorithms, we state assumptions and definitions used. The tree scheduled for broadcast can have arbitrary structure. We assume that every index node has at least two children and that each index node has the entries of an R^*-tree. The queries executed by a client are either point- or range-queries. We note that scheduling a balanced tree is easier than scheduling an arbitrary tree. Since our results hold for arbitrary trees, we present them in this more general framework.

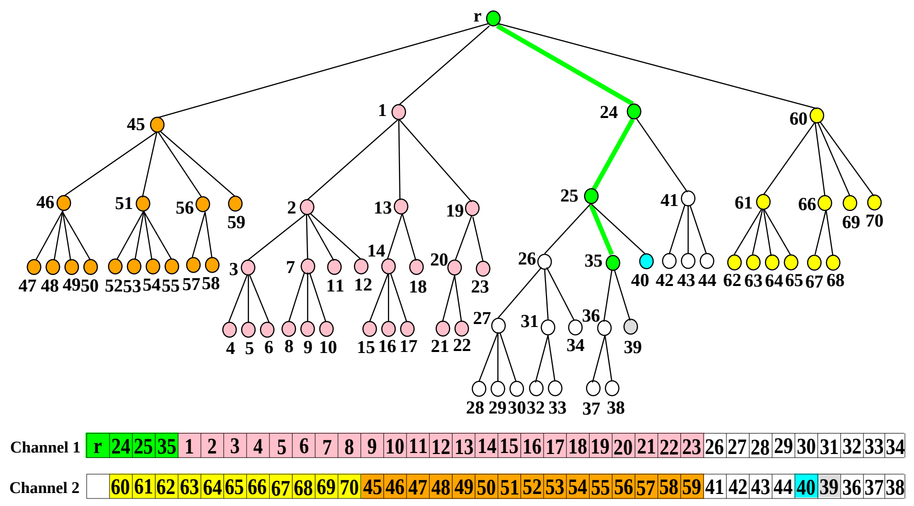
Assume that an n-node tree T is broadcast in a 2-channel environment. Generating a schedule of cycle length k, $k \geq n/2$, corresponds to generating an assignment of nodes to channel positions 1 and k. Clearly, placing two nodes in the same channel position in the cycle can cause a conflict. A conflict happens when a query needs to explore both nodes placed in the same position. Since only one node can be accessed, the client needs to wait for the next broadcast cycle to explore the second node and latency can increase.

We distinguish between two forms of conflicts, a data-conflict and a query-conflict. A *data-conflict* occurs when channels 1 and 2 contain nodes whose corresponding rectangles are not disjoint. A *query-conflict* occurs when channels 1 and 2 contain nodes whose corresponding rectangles are disjoint, but both nodes are to be explored by a mobile client executing a query. Our presented methods are heuristics minimizing data-conflicts, cycle length, and latency. Our results complement the work on achieving maximum parallelism for R-trees [5] where the goal is to maximize data-conflicts since they minimize access to disks.

5.1 Achieving optimum cycle length

This section presents an algorithm generating a 2-channel broadcast schedule of minimum cycle length (i.e., $n/2$) based on a weighted traversal of the tree. The latency of any query is bounded by n, the number of nodes in the tree. The schedule satisfies the ancestor property and it can contain data-conflicts. In addition, the schedule satisfies the *one-switch property* which captures the fact that a path from root r to any leaf switches channel positions at most once. More precisely, let P be the path from r to a leaf v. Then, either all nodes on P are assigned to channel 1 positions or there exists a node x, such that the nodes on the path from r to x are assigned to channel 1 positions and the remaining nodes on the path are assigned to channel 2 positions. One can show that when the broadcast schedule has the one-switch property, the latency of a query is bounded by n (i.e., a query is completed within two cycles).

To determine the assignment of nodes to channel positions in the channel, the algorithm performs a preorder traversal of T considering the children of a node in the following order. For a node u with children $u_1, \ldots, u_l$, let $size(u_i)$ be the number of nodes in the subtree rooted at node u_i, $1 \leq i \leq l$. The children of u are traversed in order of non-increasing $size$-entries. Assume the preorder numbering of this traversal has been generated. Let m be the node having received preorder number $\lceil n/2 \rceil$. Let P be the path from r to m with $P = < r = p_1, p_2, \ldots, p_k = m >$. In the broadcast schedule, nodes on the path from r to m are assigned to k consecutive channel 1 positions, positions 1 to k. The remaining nodes with preorder numbers between 1 and $\lceil n/2 \rceil$ are assigned to consecutive channel 1 positions immediately following position k. Nodes with a preorder number larger than $\lceil n/2 \rceil$ are assigned to consecutive channel 2 positions starting at position 2 and ending at position $\lfloor n/2 \rfloor$. Nodes are placed into channel 2 by considering subtrees rooted at nodes whose parent is on path P, handling these subtrees by decreasing preorder numbers, and within each subtree placing nodes according to increasing preorder numbers. Figure 11 shows the channel assignments for one tree. In this example, m corresponds to the node labeled 35; we have $k = 4$ and nodes $r, 24, 25, 35$ form path P.



Channel 1	r	24	25	35	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	26	27	28	29	30	31	32	33	34
Channel 2		60	61	62	63	64	65	66	67	68	69	70	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	41	42	43	44	40	39	36	37	38

Fig. 11. A tree with $n = 71$ and its 2-channel schedule with a cycle length of 36; the integers next to the nodes are the weighted preorder numbers.

Let S be the broadcast schedule generated by this procedure. It is easy to see that S satisfies the one-switch property and that the cycle length is $\lceil n/2 \rceil$. The following theorem shows that the ancestor property holds.

Theorem 1 *Broadcast schedule S satisfies the ancestor property.*

Proof: Let node m and path P be defined as above. By traversing path P from root r towards m we show that a node placed on channel 2 always has its parent placed earlier. Assume that we are at node p_i, $i \geq 2$. If the number of nodes already placed in channel 2 positions (these nodes are not in the subtree rooted at p_i) is at least $i - 1$ (and this was true for all smaller i's), the ancestor property is satisfied so far.

Assume now that the number of nodes so far placed in channel 2 positions and not in the subtree rooted at p_i is smaller than $i-1$. The node placed at position i in channel 2 would not have its parent placed earlier, thus violating the ancestor property. We show that this situation cannot arise. Consider the smallest i for which this situation occurs. Node p_{i-1} has no child assigned to a channel 2 position. Let x be the number of nodes in the subtree rooted at p_{i-1} assigned to channel 1 positions - excluding node p_{i-1} and any nodes in the subtree rooted at p_i. Let y be the number of nodes in the subtree rooted at p_i, with y_1 nodes assigned to channel 1 and y_2 nodes assigned to channel 2. From the way the preorder numbers are generated, we have $x \geq y$. Finally, let a be the total number of nodes assigned to channel 1 before node p_{i-1} was reached. Then, $i + a + x + y_1 = n/2$ and $i - 1 + y_2 = n/2$. Hence, $a + x + y_1 = y_2 - 1$, which is not possible for $a \geq 0$ and $x \geq y_1 + y_2$. Hence, such a situation cannot occur and the ancestor property is satisfied for the schedule. $\square$

The schedule generated does not place nodes with an awareness of data-conflicts. Indeed, the algorithm is a general method for assigning nodes of a tree to two channels satisfying the ancestor and one-switch property. However, the way assignments of subtrees to channels are made results in a good latency even when data-conflicts occur. A point- and range-query starting at the begin of a cycle is always completed within two cycles. Thus, for R^*-trees, the latency is no worse than the cycle length in a 1-channel environment. In many situations, it will be better. A client executing a query needs to maintain nodes to be explored according to their position in the channel. If there is a data-conflict, the first broadcast cycle explores the node in channel 1. The subsequent broadcast cycle explores remaining nodes in channel 2. The memory needs for a client are the same as for the 1-channel case. This means that a client may need hB memory in order to be able to store all the relevant index nodes to be explored. When clients have limited memory, the approaches described in Section 3 can be employed with minor modifications.

5.2 Broadcasting without data-conflicts

In this section we describe an algorithm which generates, from a given index tree T, a 2-channel broadcast schedule without data-conflicts. Using this schedule, a mobile user can execute a point query in one cycle and a range query in at most two cycles. The cycle length depends on the amount of overlap between index nodes. Within a cycle, every channel 1 position is assigned a node. Whether a channel 2 positions contains a node depends on whether the algorithm was able to identify subtrees whose corresponding regions have no overlap.

Assume T is an R-tree with root r. The scheduling algorithm assigns root r to the first position in channel 1 and then considers the children of r. Let $v_1, \cdots, v_k$ be these children. An *overlap graph* $G = (V, E)$ is created as follows: $V = \{v_1, \cdots, v_k\}$ and $(v_i, v_j) \in E$ iff the rectangles corresponding to v_i and v_j overlap , $1 \leq i, j \leq k$. For a node $v_i \in V$, let $size(v_i)$ be the number of nodes in the subtree rooted at v_i. We use G to determine an independent set IS of maximal size. For the algorithm to use the independent set, set IS needs to have the following two properties:

1. $|IS| > 2$ and
2. for all v_i in IS, $size(v_i) \leq \frac{m}{2}$, where $m = \sum_{v_i \in IS} size(v_i)$.

If such an independent set IS exists, the algorithm assigns the nodes in the subtrees rooted at the nodes in IS to the two channels. The assignment is such that every channel receives $m/2$ nodes. We first arrange the nodes in IS by non-increasing associated size-entries. Assume that $j-1$ nodes of IS have already been handled and let v_j be the j-th node. Let l_1 and l_2 be the last channel positions filled in channels 1 and 2, respectively. When starting to process an independent set, we have $l_1 = l_2$. Without loss of generality, assume $l_1 \leq l_2$. Then, node v_j and all nodes in the subtree rooted at v_j are assigned to channel 1 positions, starting with position $l_1 + 1$. The nodes in the subtree are assigned using a preorder numbering of the nodes (within the subtree). After all $|IS|$ nodes have been assigned, let l_1 and l_1 be the last channel positions filled in channels 1 and 2, respectively, $l_1 \leq l_2$. If $l_1 = l_2$, we are done processing the independent set. If $l_1 \neq l_2$, let $l_2 - l_1 = \epsilon$ and let v_r be the last node in IS placed into channel 2. The first $\frac{\epsilon}{2}$ nodes in the subtree rooted at node v_r are reassigned to the first positions in channel 1 (first with respect to the processing of set IS). This reassignment achieves $l_1 = l_2$, maintains the ancestor property and the one-switch property.

The nodes of graph G not in the independent set IS form the new graph G. The process of finding a maximal independent set continues until G is empty or no such set can be found. Assume that this situation occurred and G is not empty. Let $w_1, \cdots, w_l$ be nodes in G and let $l_1 = l_2$ be the last channel positions filled. The algorithm next assigns nodes $w_1, \cdots, w_l$ to l channel 1 positions starting at position l_1+1. The corresponding channel 2 positions receive no node assignment. Next, consider the children of $w_1, \cdots, w_l$ in the same way as the children of root r were considered. The algorithm continues this process until all nodes are assigned to channel positions.

Clearly, the cycle length depends on the number of nodes whose rectangles overlap. An extreme case occurs when all nodes are assigned to channel 1 positions. The best cycle length possible is $n/2$. Since there are no data-conflicts in the generated broadcast schedule, a client can execute a point query in a single cycle. It is easy to show that the broadcast schedule satisfies the one-switch property and that a range query can be executed in at most two cycles. However, the generated broadcast may result in a large number of paths leading from root r to leaves and switching channels. To execute a range query, client may need to store $O(n)$ nodes to be explored in the next cycle. In contrast, the previous schedules require $O(hB)$ space. We are currently exploring ways to reduce the space requirements for data-conflict free schedules. Experimental work measuring the latency and tuning time as well as the number of data-conflicts for both 2-channel algorithms is on-going.

6 Conclusions

We presented algorithms for scheduling a spatial index tree for broadcast in a 1- and 2-channel environment. The generated broadcast schedules differ on whether nodes are broadcast once or multiple times in a cycle and the choice of entries added by the scheduler. Mobile clients execute a spatial query by tuning into the broadcast. The algorithms executed by the clients aim to minimize latency and tuning time. They depend on the type of broadcast as well as on the client's available memory and chosen data structures. Our experimental work on real and synthetic data shows the tradeoffs and performance differences between the algorithms and broadcast schedules. All our solutions achieve a significant improvement over a straightforward broadcast without additional entries. Our experimental work shows that broadcast schedules using node replication achieve smaller tuning times for queries that begin during a cycle. We also show that achieving good performance when tuning-in during a cycle is related to processing a query with limited memory at the client.

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)*, 1997.
3. T. Imieliński, S. Viswanathan, and B. R. Badrinath. Energy efficient indexing on air. In *Proceedings of the International Conference on Management of Data*, pages 25–36, 1994.
4. Tomasz 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.
5. I. Kamel and C. Faloutsos. Parallel R-trees. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 195–204, 1992.
6. 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.
7. S.-C. Lo and A.L.P. Chen. Optimal index and data allocation in multiple broadcast channels. In *Proceedings of 2000 IEEE International Conference on Data Engineering*, pages 293–304, 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.
10. N. Shivakumar and S. Venkatasubramanian. Efficient indexing for broadcast based wireless systems. *MONET*, 1(4):433–446, May/June 1996.