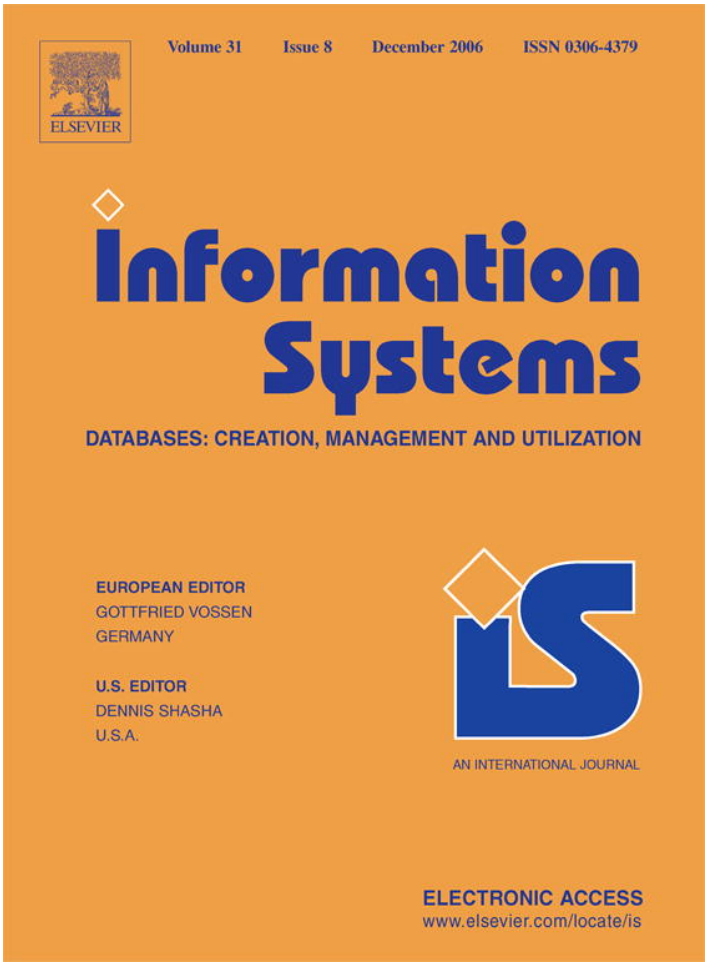


Available online at www.sciencedirect.com
ScienceDirect
Information Systems 31 (2006) 870–886
www.elsevier.com/locate/infosys

Broadcasting and querying multi-dimensional index trees in a multi-channel environment ☆

Susanne Hambrusch[a,*], Chuan-Ming Liu[b], Sunil Prabhakar[c]

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

Received 13 September 2004; accepted 11 May 2005

Recommended by F. Carino Jr

Abstract

The continuous broadcast of data together with an index structure is an effective way of disseminating data in a wireless, mobile environment. The availability of an index allows a reduction in the tuning time and thus leads to lower power consumption for a mobile client. This paper considers scheduling index trees in multiple channel environments in which a mobile client can tune into a specified channel at one time instance. Let T be an n-node index tree of height h representing multi-dimensional index structure to be broadcast in a c-channel environment. We describe two algorithms generating broadcast schedules that differ in the worst-case performance experienced by a client executing a general query. A general query is a query which results in an arbitrary traversal of the index tree, compared to a simple query in which a single path is traversed. Our first algorithm schedules any tree using minimum cycle length and it executes a simple query within one cycle. However, a general query may require $O(hc)$ cycles and thus result in a high latency. The second algorithm generates a schedule of minimum cycle length on which a general query takes at most $O(c)$ cycles. For some queries this is the best possible latency.

Keywords: Data broadcast; Latency; Mobile computing; Query processing; Power consumption; Selective tuning

1. Introduction

The continuous broadcast of data together with an index structure is an effective way of disseminating data in a wireless mobile environment [1–8]. The index enables a mobile client to tune in only when relevant data is available on the channel. This leads to a reduction in the tuning time (i.e.,

☆ Work supported by NSF Grants 9988339-CCR, 9985019, and 0010044-CCR and a Gift from Microsoft Corporation.
*Corresponding author. Tel.: +1 765 494 1831.
E-mail addresses: seh@cs.purdue.edu (S. Hambrusch), cmliu@csie.ntut.edu.tw (C.-M. Liu), sunil@cs.purdue.edu (S. Prabhakar).

0306-4379/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2005.05.002

the amount of time spent listening to the channel) and thus reduces the power consumption experienced by the client. Data broadcast with an index should also minimize the latency experienced by each client (i.e., the time elapsed between the issuing and the termination of the query). In a multiple channel environment, a client can tune into any one of the channels at each time instance, but the channel number needs to be specified. The use of multiple channels can lead to a significant reduction in the latency [4,6,9,10], but poorly designed broadcasting algorithms can lead to an increase in the latency as well as tuning time.

This paper presents two algorithms for scheduling multi-dimensional index trees in a multiple channel environment. Let T be an n-node multi-dimensional index tree of height h broadcasted in a c-channel environment, $c \geqslant 2$. The same schedule is broadcast repeatedly and we refer to the broadcast of one instance of the tree as one *cycle*. The schedules generated by our algorithms broadcast every node of tree T exactly once in each cycle. Generating a broadcast schedule thus corresponds to assigning every node to a position within the cycle and to a channel.

The broadcast schedules generated allow a mobile client to efficiently execute typical range-like queries [11]. Executing such queries corresponds to a partial traversal of the index tree. We differentiate between simple and general queries. In a simple query, a single path from the root to a leaf or interior node is traversed, while a general query results in an arbitrary traversal of the index tree. The objective is to minimize the tuning time and latency for a client when executing a query. Both of our algorithms execute a simple query within one cycle. The algorithms differ in the number of cycles needed when executing a general query.

Executing a simple query within one cycle is possible when a schedules satisfies the ancestor property defined as follows. Let u and v be any two nodes of T with u being the parent of v. A broadcast schedule satisfies the *ancestor property* when parent u is broadcast before v in the same cycle. Nodes v and u can be broadcast on different channels. The ancestor property allows a client to (i) tune in to receive node u, (ii) determine from the information available at u whether node v needs to be tuned into, and then (iii) tune in for node v in the same broadcast cycle. A schedule satisfying the ancestor property has root r of T as the only node assigned to position 1 in the broadcast schedule. In our schedules, root r is always assigned to position 1 in channel 1. Satisfying the ancestor property implies that the cycle length is at least $\gamma = \lceil (n-1)/c \rceil + 1$. We point out that our broadcast schedules are generated independent of any data access frequencies. Broadcasting solutions using access frequencies and minimizing the latency include [12–16].

In Section 2 we describe how to generate a schedule satisfying the **a**ncestor **p**roperty and having **m**inimum **c**ycle length. This schedule allows executing a **s**imple **q**uery in one cycle and we refer to the algorithm generating it as algorithm APMC-SQ. Algorithm APMC-SQ places no restriction on tree T. Clearly, if the ancestor property is to be satisfied, there exist trees that require a cycle length larger than $\gamma = \lceil (n-1)/c \rceil + 1$. We present a characterization, which we call the L-condition, that allows us to determine whether a given tree can be scheduled with the ancestor property and a cycle length of γ. Given an n-node tree, algorithm APMC-SQ generates a schedule of minimum cycle length in $O(n)$ time. The disadvantage of the schedule generated by APMC-SQ is the latency experienced when executing a general query. We show that there exist general queries for which algorithm APMC-SQ uses $O(hc)$ cycles, whereas at most c cycles are used by other schedules.

In Section 3 we describe an $O(n)$ time algorithm generating a schedule satisfying the **a**ncestor **p**roperty, having **m**inimum **c**ycle length, and allowing the execution of a **g**eneral **q**uery in at most c cycles. We refer to this algorithm as algorithm APMC-GQ. The algorithm produces the claimed schedule for a tree satisfying a condition on the minimum number of children of a node. This condition, which we call the C-condition, relates the minimum number of children to the minimum number of channels needed to place the nodes in the subtree. It allows schedules to satisfy a certain form of "compactness" with respect to how subtrees are handled. This compactness is the key to executing general

queries efficiently. The algorithm processes the tree starting at the root and uses a hybrid approach which assigns nodes either level-by-level assigns entire subtrees, once they have become "small enough". For trees satisfying the C-condition, the generated schedule has minimum cycle length of $\gamma = \lceil (n-1)/c \rceil + 1$. Section 4 discusses the algorithm a client would use for efficiently executing a general query. We conclude in Section 5.

2. Algorithm APMC-SQ

This section describes algorithm APMC-SQ which schedules a given tree T tree on c channels so that the **a**ncestor **p**roperty is satisfied and **m**inimum **c**ycle length is achieved. These properties allow a client to execute a **s**imple **q**uery in one cycle and minimize the tuning time. Recall that $\gamma = \lceil (n-1)/c \rceil + 1$ is the minimum cycle length for any tree. We first consider trees for which the minimum cycle length is γ. The algorithm for trees requiring a cycle length exceeding γ is basically the same. We make the distinction for arguing correctness.

Let $l(k)$ be the number of nodes on level k, $0 \leqslant k \leqslant h$. We say tree T of height h satisfies the *L-condition* for c channels if we have

$$\sum_{i=0}^{k} l(i) \geqslant kc + 1$$

for every level k, $0 \leqslant k \leqslant h$. To simplify the description, we assume $(n-1) \bmod c = 0$. Hence, exactly c nodes are assigned to channel position γ. Note that level h of any tree can be augmented with leaves to fill position γ and the generated schedule can then be altered to only partially fill position γ. The ancestor property requires that a node u on level k is placed at a channel position greater than or equal to $k+1$. To fill the channel with a cycle length of γ, at least $(k-1)c+1$ nodes on levels $0, 1, \ldots, k-1$ need to be placed in channel positions to the left of where node u is placed. The L-condition captures this requirement and ensures enough nodes for filling the channel. Note that a tree not satisfying the L-condition for a given c cannot be scheduled with a cycle length of γ.

Consider the 25-node tree of height 6 shown in Fig. 1(a). It satisfies the L-condition for three channels, but not for four channels. The two columns with inequalities shown on the right of the tree represent the number of nodes up to the indicated level compared to the minimum number of nodes needed for satisfying the L-condition. For $c = 4$, the inequalities are violated at levels 4 and 5 (indicated by x's). Fig. 1(b) shows two schedules, with the left one being a 3-channel schedule having a minimum cycle length of $\gamma = 9$. The right one shows a schedule of cycle length 8, which is the best possible for $c = 4$ for this particular tree. Note that for a 25-node tree and $c = 4$ the best possible cycle length for any tree is 7.

When tree T satisfies the L-condition, algorithm APMC-SQ generates a schedule having a cycle length of γ by processing the tree level by level, starting with level h. The schedule is generated by assigning nodes level by level and filling the channel in a right to left pattern, starting with position γ. The rule for placing the nodes is simple: consider the nodes on a level from left to right and assign each node to the largest channel position without violating the ancestor property. Within the same channel position, nodes are assigned to the smallest channel index possible. A node u on level k thus needs to know the smallest channel position assigned to one of its children and is then placed to the first available position to the left of this position. Using standard tree processing techniques, the n nodes can be placed $O(n)$ time using to the rules stated.

We next show that when tree T satisfies the L-condition, the algorithm generates a schedule of length γ. More specifically, position 1 contains the root r and every other position is assigned exactly c nodes. A tree satisfying the L-condition can have levels with fewer than c nodes and we say level k is *deficient* if $l(k) < c$. The L-condition guarantees that there are nodes closer to the root "making up" deficiencies of deficient levels. After having handled a level, the channel can contain partially filled channel positions. Partially filled channel positions are caused by deficient levels and form a staircase pattern, as discussed below. For the tree

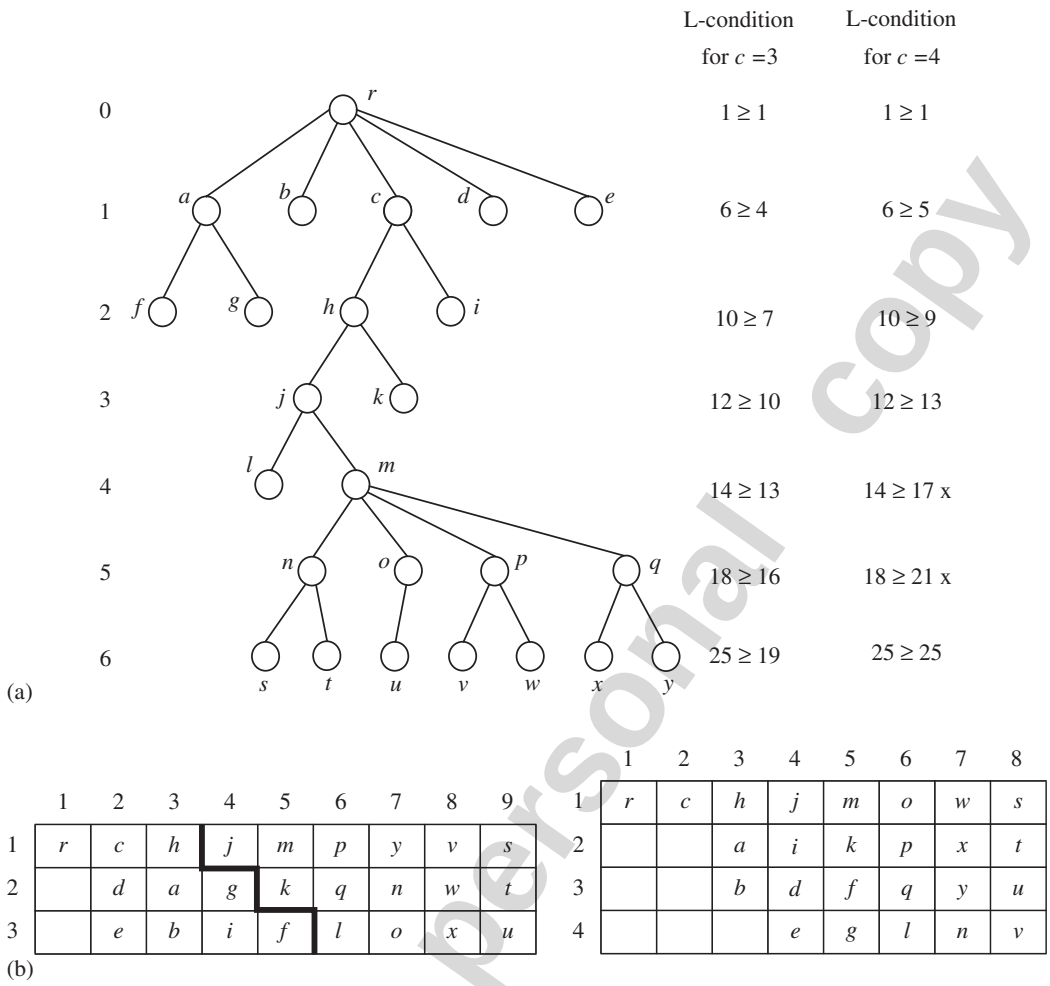


Fig. 1. Illustrating the *L*-condition. (a) A 25-node tree of height 6; columns on right state *L*-conditions inequalities for three and four channels, respectively. (b) A schedule of cycle length $\gamma = 9$ for $c = 3$ (left) and a schedule of cycle length 8 for $c = 4$ (right).

shown in Fig. 1 and $c = 3$, levels 3 and 4 are deficient, but the *L*-condition is satisfied. After level 3 has been handled, the channel contains a 2-staircase, as indicated by the bold lines in the 3-channel schedule.

Lemma 1. *For every deficient level k there exist* $c - l(k)$ *unique nodes from levels* $k - 1$ *or higher that can be associated with level k to form a set of c nodes.*

Proof. To determine the nodes associated with deficient levels, imagine scanning the tree level by level, starting with level 1. For a non-deficient level, mark all but c nodes to be available for "association" with a deficient level. Clearly, one marked node can be associated with only one level. For every deficient level k, identify $c - l(k)$ nodes marked for association to form a set of size c. To determine these nodes, use marked nodes at level $k - 1$ or smaller on levels closest to level k. However, it is not crucial which nodes are used, only that such $c - l(k)$ unique nodes exist. Since the tree satisfies the *L*-condition, the levels up to and including level k contain at least $kc + 1$ nodes. Hence, the $c - l(k)$ nodes needed by level k can always be identified and every deficient level k can be associated with $c - l(k)$ unique nodes closer to the root. □

Let $a(j)$ be the number of nodes assigned to channel position j at any time during the algorithm, $1 \leqslant j \leqslant \gamma$. Let i be the smallest channel

position with $a(i)>0$. We say that the channel contains an s-staircase if there exist s partially filled channels and $0<a(i)\leqslant\cdots\leqslant a(i+s-1)<a(i+s)=c$. In the s-staircase we can identify s nodes $x_1,x_2,\ldots,x_s$ with $x_1,x_2,\ldots,x_s$ assigned to channel positions $i,i+1,\ldots,i+s-1$, respectively, such that tree T contains a path from x_1 to x_s. This is illustrated in Fig. 2(a) where the path from node x_1 to x_6 contributes to the gray-shaded 6-staircase. The next two properties show that partially filled channel positions create a staircase pattern and that non-deficient levels alone can only create a 1-staircase.

Property 1. *After handling level k we have $a(1)\leqslant a(2)\leqslant\cdots\leqslant a(j)\leqslant\cdots\leqslant c$, $1\leqslant k\leqslant h$.*

Proof. The condition $a(1)\leqslant a(2)\leqslant\cdots\leqslant a(j)\cdots\leqslant c$ induces a staircase pattern. Assume that after handling level k, the channel does not contains a staircase pattern. Then, there exist positions $j-1$ and j such that $a(j-1)>a(j)$ and $a(j)<c$. From the way nodes are assigned it follows that re-assigning any node assigned to position $j-1$ to position j would violate the ancestor property. Hence, every node at position $j-1$ is a parent of a node at position j. However, there can be at most $a(j)$ parents and since $a(j-1)>a(j)$ there exists at least one node at $j-1$ that can be assigned to position j. This contradicts the way the algorithm assigns nodes and the staircase pattern follows. □

Property 2. *Assume levels $h,\ldots,k+1,k$ are not deficient. After having all nodes on levels up to and including level k, the channel contains at most a 1-staircase.*

Proof. The condition is satisfied after level h has been handled since there are no precedence constraint among leaves. Assume it holds before level k is handled and let position i be the only partially

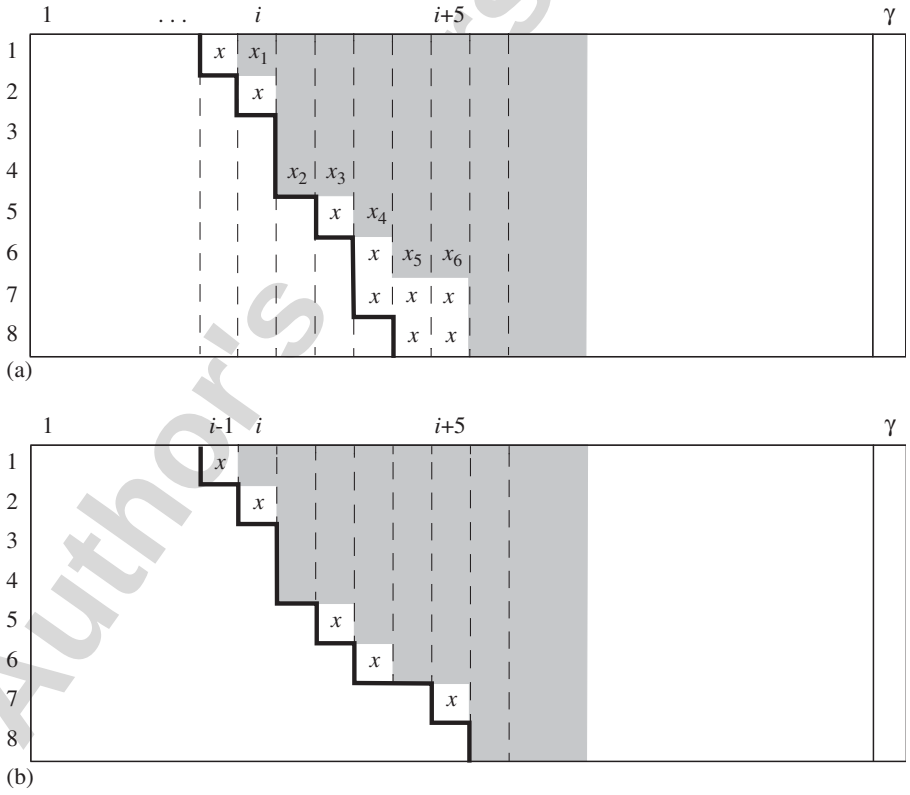


Fig. 2. Showing two possible extensions for a channel with $c=8$ containing a 6-staircase (gray-shaded area) before level k is processed. (a) Level k is not deficient: nine nodes on level k change the 6-staircase into a 5-staircase. (b) Level k is deficient: five nodes on level k change the 6-staircase into a 7-staircase.

filled channel positions containing $a(i)$ nodes (coming from level $k-1$). Level k contains at least $l(k)-a(i) \geqslant c-a(i)$ nodes that are not a parent of a node assigned to position i. Placing $c-a(i)$ of these nodes fills position i with c nodes. The positions to the left of i are filled with nodes on level k without any precedence constraint. This results in at most a 1-staircase pattern and the claim follows. □

Assume the channel contains an s-staircase before level k is handled. We next discuss how the channel changes after the nodes on level k have been placed. When level k is a deficient level, the algorithm places at least one of the $l(k)<c$ nodes into channel position $i-1$ (i.e., the rightmost empty channel position). If channel position $i+s-1$ (i.e., the rightmost partially filled position) is not filled by nodes from level k, we continue with an $(s+1)$-staircase. If position $i+s-1$ is filled, we continue with an s-staircase, or possibly a smaller staircase.

When level k is not a deficient level, assigning the nodes on level k results in an s'-staircase, with $s' \leqslant s$. At least one node on level k is again placed at position $i-1$. Since we place at least c nodes, the channel can only contain an s-staircase or a smaller one after handling level k. This follows from the way nodes are assigned and from Property 1.

Fig. 2 shows two situations for assigning the nodes on level k and extending a 6-staircase for $c=8$. Assume for (a) that level k is not deficient and contains nine nodes. The placement of the nine nodes is indicated by the x's without a subscript. One node is a parent of the node at position i and is placed at position $i-1$. The remaining eight nodes are placed into rightmost positions creating the 5-staircase shown. In (b), level k is a deficient level containing five nodes. Placing them results in the 7-staircase shown.

We can conclude that if the channel contains an s-staircase after handling level k, the algorithm encountered at least $s-1$ deficient levels. Further, the $l(k)$ nodes on a non-deficient level k fill the area below the existing staircase (with the exception of the nodes placed in position $i-1$). Note that each one of $l(k)-c$ nodes is either associated with a deficient level (as stated in Lemma 1) or is a surplus node with respect to the L-condition.

Theorem 1. *For a tree T satisfying the L-condition, algorithm APMC-SQ generates a schedule having cycle length γ and satisfying the ancestor property.*

Proof. The algorithm clearly generates a schedule satisfying the ancestor property. To show that the cycle length is γ we show that placing the root generates a 1-staircase. Assume that before handing level k the channel contains an s-staircase with i being the leftmost partially filled position, $s \geqslant 2$. Let $free(i)=\sum_{j=i+1}^{i+s-1} c-a(j)$. Quantity $free(i)$ does not include unfilled positions at channel position i and is a lower bound on the number of nodes associated with deficient levels yet to be assigned (these nodes are on level k or higher).

When level k is a deficient level, we place between one and $a(i)$ nodes into position $i-1$. Placing fewer than c nodes on level k into rightmost possible positions can result in $free(i-1)$ being larger, smaller or identical to $free(i)$. The important observation is that $free(i)$ can increase and that the staircase can increase (by at most 1).

When level k is not deficient, nodes in excess of c fill partially filled channel positions below the staircase. Since at most $c-a(i)$ positions are added and at least $l(k)-a(i)$ positions are filled, we have $free(i-1) \leqslant free(i)$. Indeed, compared to $free(i)$, $free(i-1)$ reduces by at least $l(k)-c$. In Fig. 2(a), we have $free(i)=15$ before non-deficient level k is handled. After placing the nine nodes on level k, we have $free(i-1)=14$.

After the nodes on level 1 have been placed, we have placed for every deficient level k the $c-l(k)$ associated nodes. Placing these nodes filled the staircase and there could be at most a 1-staircase. Since we assumed $(n-1) \bmod c = 0$, we have a 0-staircase consisting of $\gamma-1$ channel positions, each assigned exactly c nodes, and the $free(2)=0$. Hence, placing the root into channel position 1 creates a 1-staircase and the cycle length γ follows. □

Assume now that tree T does not satisfy the L-condition and that the algorithm makes assignments of nodes to channel positions as before. Not satisfying the L-condition means there exist deficient levels that cannot be associated with "surplus" nodes on higher levels. The tree shown in Fig. 1(a) does not satisfy the L-condition for

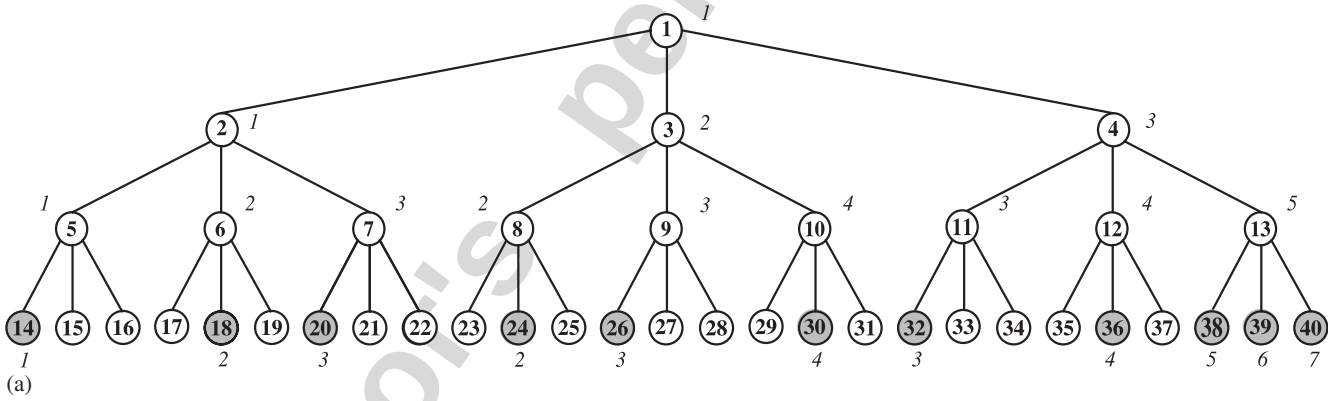
$c = 4$. Deficient levels 4 and 5 would need a total of four nodes on higher level, but only one node is available to be associated with a deficient level. After the nodes on level 1 have been placed, the channel contains a 2-staircase (with node c at position 2) and placing root r extends the 2-staircase into a 3-staircase. When the tree does not satisfy the L-condition, the channel does not contain a 0-staircase after level 1 is processed. Hence, the cycle length will exceed γ. The following result follows.

Theorem 2. *For an arbitrary n-node tree T, a broadcast schedule satisfying the ancestor property and having minimum cycle length can be generated in $O(n)$ time.*

The final discussion focuses on the latency needed to execute a query. A simple query can be executed in one cycle. A general query can require a number of cycles to execute. Fig. 3(a) and (b) show a tree and a broadcast schedule generated by Algorithm APMC-SQ, respectively. Assume a general query is executed and the query results are in the 11 shaded leaf nodes of the tree. Executing this query on the schedule shown in (b) uses seven cycles. The italic integer next to a node indicates the cycle a node is explored in. The schedule shown in (c) also satisfies the ancestor property and also has minimum cycle length. However, it allows the same query to be executed in three cycles. This example illustrates the disadvantages of the schedule generated by APMC-SQ: there exist trees of height h for which a general query uses $h(c-1)+1$ cycles. At the same time, there exist schedules for which any general query can be executed in at most c cycles. The next section describes an algorithm generating a schedule for which any general query can be executed in at most c cycles.

3. Algorithm APMC-GQ

This section describes algorithm APMC-GQ generating, in $O(n)$ time, a schedule that satisfies



(a)

(b)

channel														
1	1	2	5	8	11	14	17	20	23	26	29	32	35	38
2		3	6	9	12	15	18	21	24	27	30	33	36	39
3		4	7	10	13	16	19	22	25	28	31	34	37	40

(c)

channel														
1	1	2	5	14	15	16	6	17	18	19	7	20	21	22
2		3	8	23	24	25	9	26	27	28	10	29	30	31
3		4	11	32	33	34	12	35	36	37	13	38	39	40

Fig. 3. General queries executed by Algorithm APMC-SQ. (a) A 40-node tree (node names inside circles; italic numbers indicate the cycle a node is explored in schedule shown in (b). (b) Schedule generated by algorithm APMC-SQ; a query with results in the shaded leaves uses 7 cycles. (c) A schedule for which the same query uses 3 broadcast cycles.

the **a**ncestor **p**roperty, has **m**inimum **c**ycle length, and allows a **g**eneral **q**uery to be executed by a client in at most c cycles. The algorithm does not assume that $(n-1) \bmod c = 0$, as was done, w.l.o.g., for algorithm APMC-SQ. In the broadcast schedule generated, position 1 contains only the root, position γ contains $(n-1) \bmod c$ nodes, and every other position contains exactly c nodes.

We assume that tree T has been preprocessed so that every node u has its children arranged by non-increasing subtree sizes, with the largest subtree being the leftmost child. Throughout, we use s_u to denote the size of the subtree rooted at node u (including u). The schedule is generated in two phases:

Phase 1: The channel assignment phase, done by algorithm ASSIGN, assigns every node to a channel. Selected nodes are not only assigned to a channel, but also to their position in the channel. We refer to these nodes as *cp-assigned* nodes and the others as *c-assigned* nodes.

ASSIGN traverses the tree level-by-level, starting at the root. Channel positions are filled from the left to right with *cp*-assigned nodes. For a subtree identified as being assigned to one channel, all nodes in the subtree are identified as *c*-assigned nodes (i.e., not yet assigned to positions in the channel).

Phase 2: The positioning phase assigns all *c*-assigned nodes to a position in the channel determined in ASSIGN. This is done in procedure POSITION.

Algorithm APMC-GQ assumes tree T satisfies a condition which allows *compact* scheduling of subtrees within a cycle length of γ, $\gamma = \lceil (n-1)/c \rceil + 1$. This condition, called C-condition, is defined next. The compact schedules are the key to being able to execute any general query in at most c cycles, as is discussed in Section 4. To fill the c channels at positions 2, the root r has to have at least c children. Let u be any node on level $j-1$ and assume u is the root of a subtree of size s_u. To ensure compact scheduling, we require that node u has at least $\lceil (s_u - 1)/(\gamma - j) \rceil$ children. When this condition holds for every interior node of the tree, we say that the tree satisfies the *C-condition*. The next paragraph gives an intuitive justification of the C-condition.

A node u on level $j-1$ is placed at a position k with $k \geqslant j$. Channel position k cannot be assigned to any other node in the subtree rooted at u. The number of positions available in one channel for nodes in the subtree rooted at u is at most $\gamma - k$. The schedule we generate assigns all non-root nodes of the subtree rooted at u to channel positions $k+1, k+2, \ldots, \gamma$. Only the smallest and the largest channel index containing nodes from the subtree do not use all $\gamma - k$ of these positions. In some sense, we pack channels with nodes in the subtree rooted at u, starting where a previous subtree ended, completing filling one channel after the other, and filling a last channel possibly only partially. At position $k+1$ we can only place children of node u. The C-condition ensures that u has enough children to fill all channels used by the subtree at position $k+1$.

Consider the tree sketched in Fig. 4. Node a_1 is the root of a subtree of size 48 and assigned to position 2 or later in the channel. For $c = 5$, once node a_1 is assigned to a channel in position 2, at most 20 nodes remaining in the subtree rooted at a_1 can be assigned to one channel. In general, the 47 non-root nodes may get assigned to four channels: two channels will always be assigned 20 nodes and the remaining 7 nodes may get partitioned among two channels. The C-condition requires the root of a subtree of size 48 to have at least 3 children. This allows the algorithm to pack nodes while satisfying the ancestor property. In the particular example, it turns out we need to assign only two of a_1's children to position 3 (children b_1 and b_2), but three children are needed if the first channel would not be channel 1.

Before ASSIGN starts, algorithm APMC-GQ does the following preprocessing. The children of every node are arranged by non-increasing sizes of subtrees. Root r is assigned to position 1 in channel 1 and labeled as *cp*-assigned. For every channel k determine its channel capacity t_k; i.e., the number of nodes yet to be assigned to channel k. After assigning the root, $(n-1) \bmod c$ channels have a capacity of $\lfloor (n-1)/c \rfloor$, while the remaining channels have a capacity of $\lfloor (n-1)/c \rfloor - 1$. Algorithm APMC-GQ then invokes ASSIGN$(r, 1, c)$.

Recall that in phase 1 nodes are either labeled as *cp*-assigned or as *c*-assigned. Assume node u is on level $j-1$. A *cp*-assigned node u on level $j-1$ is always assigned to channel at position j. When the root u of a subtree is labeled as *c*-assigned and the parent of u is *cp*-assigned, we refer to u as a *root-node*. We describe ASSIGN assuming it was invoked as ASSIGN(u, nxt, c_u). The following holds at that point in time:

- node u is *cp*-assigned to a channel at position j and
- position $j+1$ in channel nxt has no *cp*-assigned node assigned to it.

ASSIGN will assign the nodes in the subtree rooted at u (excluding u) to channels $nxt, nxt+1, \ldots, nxt+c_u-1$, where c_u is the number of channels needed by the subtree.

Let $v_1, v_2, \ldots, v_l$ be the children of u arranged by non-increasing subtree sizes. ASSIGN considers u's children in the order $v_1, v_2, \ldots, v_l$. Assume ASSIGN(u, nxt, c_u) is processing child v_i which is on level j. The subtree rooted at v_i is handled depending on whether or not v_i has been *cp*-assigned earlier. Node v_i can be *cp*-assigned or unassigned (it will never be *c*-assigned). These two cases are similar, but we describe them separately for the sake of clarity.

Case 1: Node v_i is unassigned.

Recall that an invariant condition is that channel nxt has no *cp*-assigned node at position $j+1$. Earlier calls could have identified *c*-assigned nodes for channel nxt. These subtrees will be assigned to positions in channel nxt in phase 2. The target entry t_{nxt} represents the number of available positions yet to be assigned. Depending on the size of the subtree, one of two situations applies.

(A) If the subtree rooted at node v_i *does fit* into channel nxt; i.e., $s_{v_i} \leqslant t_{nxt}$, then

- Mark v_i a root-node for channel nxt.
- Mark all nodes in the subtree as *c*-assigned to channel nxt.
- Update the capacity of channel nxt (i.e., $t_{nxt} = t_{nxt} - s_{v_i}$).
- Set $c_i = 1$ (c_i represents the number of channels used).
- Node v_i makes no further calls to ASSIGN.

(B) If, on the other hand, the subtree rooted at node v_i *does not fit* into channel nxt (i.e., $s_{v_i} > t_{nxt}$), the subtree rooted at v_i is assigned to channels as follows.

- Compute c_i, the number of channels receiving nodes from the subtree rooted at node v_i; $c_i \geqslant 2$. Computing c_i takes into account that node v_i will be placed at position $j+1$ and that none of its children can be placed at position $j+1$. Using the compact channel approach, computing c_i assumes channel nxt and subsequent channels, with the exception of the last one, are filled completely. Channel $nxt+c_i-1$ is thus the largest channel index receiving a node from the subtree rooted at v_i.
- Nodes $v_i, v_{i+1}, \ldots, v_{i+c_i-2}$ become *cp*-assigned nodes assigned to channels $nxt, nxt+1, \ldots, nxt+c_i-2$, respectively, at position $j+1$. Lemma 3 shows that in a tree satisfying the *C*-condition, these sibling nodes are always present. Note that while channel $nxt+c_i-1$ received nodes from the subtree, it does not receive a *cp*-assigned node at position $j+1$ at this point (this means the invariant condition is satisfied for the next sibling).
- Invoke ASSIGN(v_i, nxt, c_i) to determine the assignment of the subtrees rooted at the children of v_i.

Case 2: Node v_i is *cp*-assigned.

When handing an earlier sibling of node v_i, node v_i was identified as a *cp*-assigned node to fill position $j+1$ in a channel needed by the subtree of this sibling. Let c' be the channel index node v_i is assigned to. Since channel nxt does not hold a *cp*-assigned node, we have $c' < nxt$. As for Case 1, one of two situation applies, depending of the size of the subtree rooted at v_i.

(A) If the subtree rooted at node v_i *does fit* into channel nxt (i.e., $s_{v_i} - 1 \leqslant t_{nxt}$), then

- Mark all children of node v_i as root nodes for channel nxt.

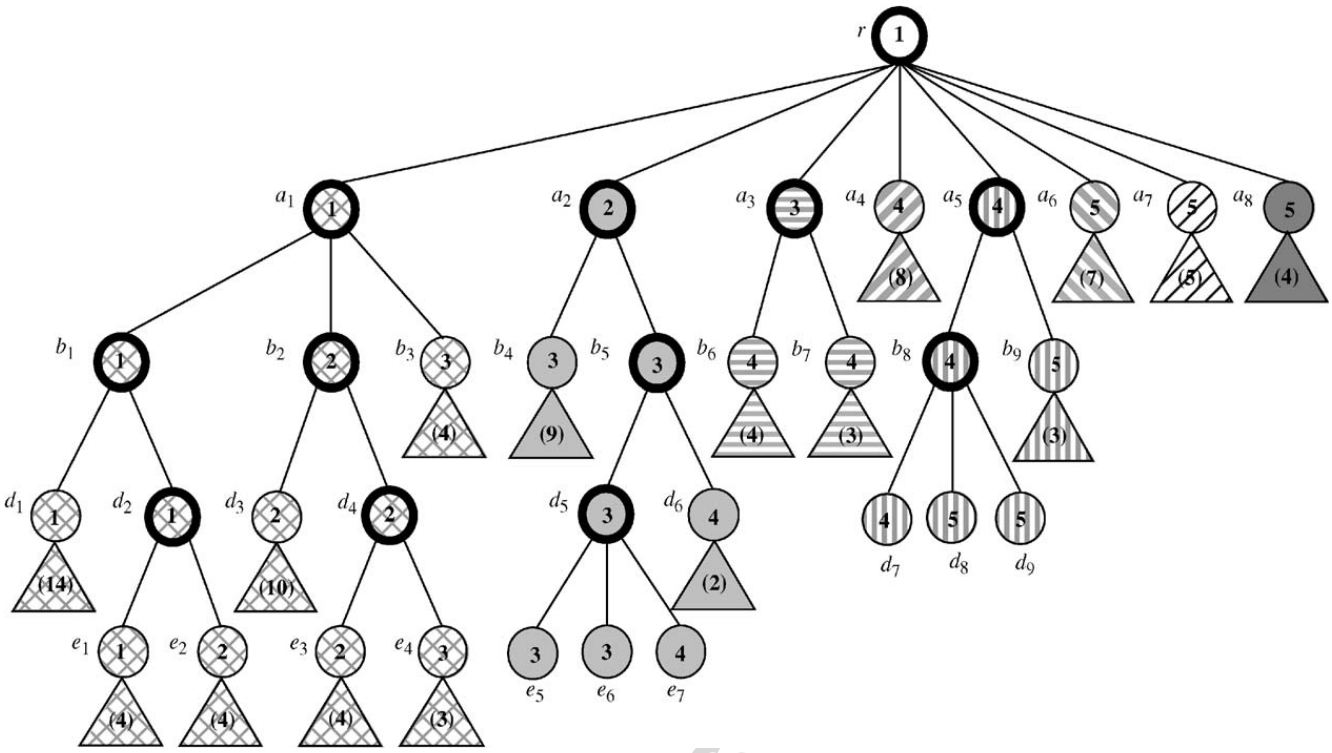


Fig. 4. Algorithm APMC-GQ on a 106-node tree and 5 channels ($\gamma = 22$). Nodes with thick (resp. thin) cycles represent *cp*-assigned nodes (resp. root-nodes). The number inside a node represents the assigned channel; the number in a triangle representing a subtree represents the number of nodes in the subtree. The sizes of the subtrees rooted at the nodes on level 1 are 48, 17, 8, 8, 8, 7, 5, and 4, respectively.

- Mark all nodes in the subtrees as *c*-assigned nodes for channel *nxt*.
- Update the target capacity for channel *nxt*.
- Set $c_i = 1$.
- Node v_i makes no further calls to ASSIGN.

(B) If the subtree rooted at node v_i *does not fit* into channel *nxt*, the subtree rooted at v_i is assigned to channels as follows. Let $v_{l'}$ be the right-most the child of node u assigned to a channel. Node $v_{l'}$ is a *cp*-assigned at position $j+1$.

- Compute c_i, the number of channels needed to place the subtree rooted at v_i. This is done as described for case 1 (use the target capacities, the fact that no descendent of v_i can be placed into the position $j+1$, and the compact schedule approach).
- Nodes $v_{l'+1}, v_{l'+2}, \ldots, v_{l'+c_i-1}$ become *cp*-assigned nodes for channels $nxt, nxt+1, \ldots, nxt+c_i-2$, respectively, at position $j+1$. Lemma 3 shows that in a tree satisfying the *C*-condition, these sibling nodes are always present.
- Invoke ASSIGN(v_i, nxt, c_i).

Before proceeding with sibling v_{i+1}, procedure ASSIGN makes updates to provide the correct channel environment. Node v_{i+1} needs a new channel *nxt* that has no *cp*-assigned node at position $j+1$ and at least one free location to the right of position $j+1$. Channel $nxt+c_i-1$ is the last channel to have received a node when handling v_i. Two situations are possible:

- Channel $nxt+c_i-1$ *is filled* to capacity; i.e., the total number of *cp*- and *c*-assigned nodes equals the initial channel capacity. In this case we set $nxt = nxt+c_i$ and continue with sibling v_{i+1}. Observe that the new channel *nxt* does not have a *cp*-assigned node at position $j+1$.
- Channel $nxt+c_i-1$ *is not filled* to capacity. In this case we continue with $nxt = nxt+c_i-1$ and sibling v_{i+1}.

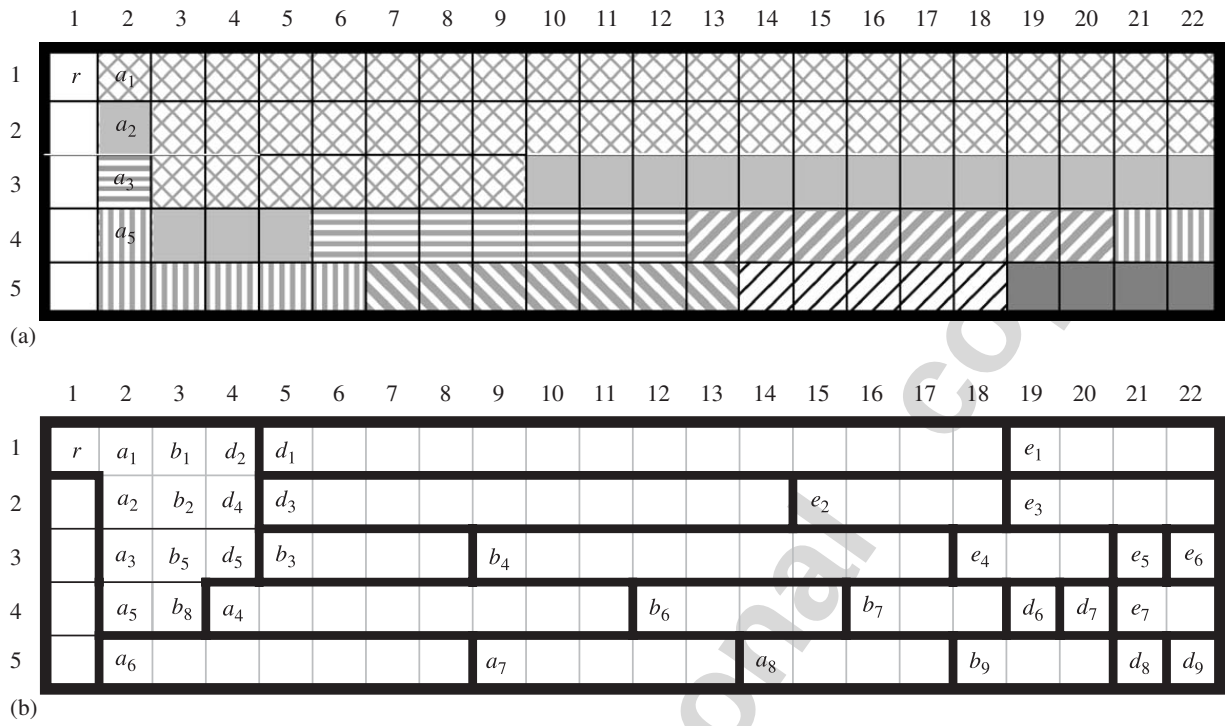


Fig. 5. Determining the broadcast schedule for the tree shown in Fig. 4; $n = 106$, $c = 5$ and $\gamma = 22$. (a) Channel position 2 shows the *cp*-assigned nodes on level 1 of the tree (they are identified when the children of the root are processed in ASSIGN). The shading patterns indicate which channels receive nodes from subtrees rooted at level 1. Shading indicates number of positions, not actual positions. For example, the darker solid gray on channel 5 matches the shading of node a_8 and indicates that the subtree rooted at a_8 will be assigned to four positions in channel 5. (b) The final schedule showing all *cp*-assigned nodes and the root-nodes along with their position determined by POSITION; nodes in root-node subtrees are not explicitly shown, but the positions used are indicated.

After procedure ASSIGN, every node of the tree is either *cp*- or *c*-assigned. Procedure POSITION assigns the *c*-assigned nodes to positions in the designated channels. Consider the root-nodes by increasing level in T. Root-nodes on the same level are considered from left to right (actually, any order would work). Let $v_1, \ldots, v_{m_i}$ be the resulting sequence of root-nodes for some level i. Let T_j be the subtree rooted at v_j, $1 \leqslant j \leqslant m_i$ and let c_j be the channel identified for v_j. Starting with the next available position in channel c_j, place the nodes of subtree T_j in pre-order in consecutive locations in channel c_j.

Figs. 4 and 5 illustrate algorithm APMC-GQ for a tree with $n = 106$ and a channel with $c = 5$. The schedule generated has length $\gamma = 22$. Fig. 4 shows the tree with selected subtrees drawn as shaded triangles. The integer in the triangle indicates the number of nodes in the subtree. Note that the subtrees rooted at the nodes on level 1 have sizes $48, 17, 8, 8, 8, 7, 5$, and 4, respectively. The shading of nodes and subtrees is used to illustrate where the nodes in the subtrees rooted at the nodes on level 1 (the a's) are placed.

After the root r has been assigned, ASSIGN$(r, 1, 5)$ is invoked. Node a_1 is the root of a subtree of size 48 and placing this subtree requires 3 channels (Case 1(B)). This results in nodes a_1 and a_2 becoming *cp*-assigned to position 2 in channels 1 and 2, respectively. ASSIGN$(a_1, 1, 3)$ is invoked next. The pattern used in Fig. 4 for the subtree rooted at a_1 is also used in Fig. 5(a) and it shows the number of locations in channels $1, 2$, and 3 assigned nodes from this subtree. Note that channels 1 and 2 will be filled completely with nodes from this subtree. Channel 3 receives 7 nodes. Fig. 5(a) suggests that seven positions in channel 3 are being used by

nodes in the subtree rooted at a_1, not which positions. The final positions of these seven nodes are shown in Fig. 5(b): they are the subtree rooted at b_3 (four nodes) and the subtree rooted at e_4 (three nodes).

When the second child of the root, a_2, is handled, we have $nxt = 3$ and position 2 in channel 3 is free. Since node a_2 has already been *cp*-assigned and it is the root of a subtree of size 17, Case 2(B) holds. 13 of these nodes are assigned to channel 3 and seven nodes are assigned to channel 4. Before ASSIGN is invoked on node a_2 to determine the channel for each node in the subtree, node a_3 is *cp*-assigned to channel 3 at position 2.

We then reach the third child of the root, a_3, which is the root of a subtree of size 8. We have $nxt = 4$ and the subtree rooted at a_3 fits into channel 4: Case 2(A) holds. Nodes b_6 and b_7 are marked as root-nodes. These two root-nodes and the nodes in their respective subtrees are *c*-assigned to channel 4. Fig. 5(a) indicates place holders for these 7 nodes through the horizontally striped pattern (a_3 has the same pattern).

Continuing with the children of the root, we assign the subtree rooted at a_4 to channel 4 and mark a_4 a root-node (Case 1(A)). The nodes in the subtree rooted at a_5 are partitioned between channels 4 and 5 (a_5 is *cp*-assigned to channel 4). The remaining three subtrees are assigned to channel 5 and their roots are marked as root-nodes.

Fig. 5(b) shows the assignment of nodes to channels and positions in the channel. Nodes in subtrees rooted at root-nodes are indicated by the position of the root-node and the number of spaces. For example, the seven nodes in the subtree rooted at a_1 assigned to channel 3 are the nodes in the subtree rooted b_3 (four nodes in positions 5–8) and the nodes in the subtree rooted at e_4 (three nodes in positions 18–20). Nodes in subtrees are placed following a pre-order traversal.

We now turn to the correctness of the algorithm. Two crucial properties, one being the ancestor property, follow from the way nodes are assigned to channels by ASSIGN and positioned within the assigned channel by POSITION. For any node v, let $chan(v)$ be the assigned channel and $pos(v)$ be the position of v in $chan(v)$.

Lemma 2. *Let v be a node with parent u. Then, $pos(u) < pos(v)$ and $chan(u) \leqslant chan(v)$.*

Proof. Assume parent u is on level $j-1$. Consider first the condition on the positions. When both u and v are *cp*-assigned nodes, we have $j = pos(u) = pos(v) - 1 = j + 1$ and the condition follows. When u and v are nodes in a common subtree whose root is a root-node, u and v are assigned to the same channel using a pre-order ordering and $pos(u) < pos(v)$ follows. Finally, when u is a *cp*-assigned node and v is a root-node, the algorithm excludes position $pos(u)$ for any descendent of u and the condition follows. Hence, the ancestor property is satisfied.

Consider now the condition on the channel indices. When u and v are *c*-assigned nodes, they belong to a common subtree whose root is a root-node. We have $chan(u) = chan(v)$ from the way such a subtree is handled. Assume that parent u is a *cp*-assigned node. Then, from the way a subtree rooted at a *cp*-assigned node is handled to achieve a compact schedule, it follows that $chan(u) \leqslant chan(x)$ for any descendent x of u. Hence, the condition $chan(u) \leqslant chan(v)$ follows. We note that this condition will allow a general query to be executed in at most c cycles, as will be discussed in Section 4. □

Lemma 3. *Every channel position between positions 2 and γ is assigned c nodes of the tree.*

Proof. Algorithm APMC-GQ packs the nodes into channels while satisfying the ancestor property. In order to show that every position (except the first and the last) are assigned c nodes, we need to show that steps 1(B) and 2(B) in procedure ASSIGN are always able to place the required number of *cp*-assigned nodes.

Let u be again a *cp*-assigned node on level $j-1$ of the tree. After ASSIGN(u, nxt, c_u) is invoked, the children $v_1, v_2, \ldots, v_l$ of u are considered. The channels are filled starting with channel nxt and position $j+1$ is the leftmost position used. The *C*-condition states that u has at least $\lceil (s_u - 1)/(\gamma - j) \rceil$ children. The *cp*-assigned children of node u are placed at position $j+1$ starting with channel nxt. Once all children of u have been handled, the last channel receiving nodes from the

subtree rooted at u is assigned a *cp*-assigned node at $j+1$ only when the last channel is filled completely. Since the number of children is at least $\lceil (s_u - 1)/(\gamma - j) \rceil$, ASSIGN$(u, nxt, c_u)$ has enough children to satisfy the demand for *cp*-assigned nodes. Hence, steps 1(B) and 2(B) can always be executed and the desired schedule is generated. □

The $O(n)$ running time of algorithm APMC-GQ follows from using straightforward tree processing methods. The tree is traversed once in a preprocessing. ASSIGN traverses the tree a second time and POSITION traverses the subtrees rooted at root-nodes. The following theorem summarizes the result of this section.

Theorem 3. *Algorithm APMC-GQ generates, in $O(n)$ time, a broadcast schedule satisfying the ancestor property and having a cycle length of γ.*

Before turning in the next section to the query processing done by a client on the broadcast schedule, we illustrate why the correctness of the algorithm requires the children to be considered by non-increasing subtree sizes. We do this by giving a simple counterexample. Consider $c = 3$ and a tree in which the root r has three children: two, a_1 and a_2, are single nodes and the third child, a_3, is the root of a large subtree. If the children are considered in the order a_1, a_2, a_3, the subtrees of size 1 rooted at a_1 and a_2, respectively, are *c*-assigned to channel 1. Node a_3 is *cp*-assigned to position 2 in channel 1 and the nodes in the subtree rooted at a_3 use positions starting with position 3. This means channels 2 and 3 have no node assigned to position 2 and the desired schedule is not generated. On the other hand, when the nodes are considered in the order a_3, a_2, a_1, all three nodes become *cp*-assigned to position 2 and all channels at position 2 are filled. Considering subtrees by non-increasing sizes guarantees that the *cp*-assigned nodes captured by the *C*-condition are being used as needed.

4. Query processing

This section describes how a mobile client executes a query using a broadcast schedule generated by one of the described algorithms. We discuss how the client selects the channel number if it needs data from identical positions on different channels, how the client manages information related to the broadcast, as well as performance in terms of latency, tuning time, and space requirements. We also discuss how a client can execute multiple queries simultaneously.

4.1. Query processing protocol

Assume first that query execution starts at the beginning of a broadcast cycle. We say a client *explores* node u when the client tunes into the broadcast, retrieves node u, and examines all of u's entries. If none of u's children need to be explored further, u is an *unproductive node*, otherwise u is called a *productive* node. Productive nodes can generate query responses and thus every search strategy needs to access all productive nodes. The client maintains two dictionary structures, T_{act} and T_{next}. T_{act} contains cycle positions holding a node to be explored in the on-going cycle. T_{next} stores cycle positions of nodes to be explored in later broadcast cycles. Both structures are organized and indexed by cycle positions. Each entry associated with a channel position stores at most c channel numbers containing nodes to be explored at the corresponding position.

Query execution starts with the client tuning in and exploring root r at position 1. The client uses T_{act} to determine the next node to explore as follows. The client deletes from T_{act} the smallest cycle position containing a node to be explored. Let e be this position. If there is more than one node to be explored at position e, the node at the smallest channel number is explored and the remaining entries for position e are inserted into T_{next}. Let u be the node explored next. If u is a productive node, let $v_1, \ldots, v_l$ be the children of u to be explored. Nodes $v_1, \ldots, v_l$ are then inserted into T_{act}. Since the ancestor property holds, all associated positions are larger than the current position. There may be more than one node at a single position and the complete exploration may require a number of broadcast cycles.

After the exploration of a productive node u, T_{act} is not empty. Hence, the next node to be explored is again determined using T_{act}, as

discussed above. After the exploration of an unproductive node, one of the following occurs:

(1) T_{act} is not empty.
The next node to be explored is determined from T_{act} as discussed above.
(2) T_{act} is empty, but T_{next} is not empty.
When this happens, the client tunes out for the remainder of the current broadcast cycle. The client makes T_{next} the new T_{act}, T_{next} is initialized as being empty, and the client continues query processing at the beginning of the next cycle.
(3) T_{act} and T_{next} are both empty.
No more nodes are to be explored and the query terminates.

Structures T_{act} and T_{next} can be implemented, for example, as binary search trees with a priority queue associated with each node. The operations executed correspond to *Extract-Min*, *Search*, *Insert*, and *Delete*. The maximum number of entries in T_{act} and T_{next} are $\lceil (n-1)/c \rceil + 1$ and the number of channel numbers associated with each position is c. Hence, each operation costs $O(\log n/c)$ time on the structure for T_{act} and T_{next} and $O(\log c)$ time for handling the channel numbers associated with one cycle position. In total, the client's cost of determining the next node to explore is $O(\log n)$.

4.2. Performance

Since all nodes explored have the potential for producing a query result, the tuning time a client experiences is the best possible. The latency experienced by a client depends on the number of cycles needed. This number is related to the number of cycle positions containing more than one node to be explored. If a query uses more than c cycles in a c-channel environment, the resulting latency is exceeds the worst-case latency in an 1-channel environment. Solutions for multiple channels should avoid such a situation.

A simple query corresponding to one path from root r to a leaf can be executed within one cycle and thus has a latency of at most one cycle. This holds since both of our algorithms generate broadcast schedules satisfying the ancestor property. Consider now a general query requiring a partial traversal of the index tree. We have already shown in Section 2 that algorithm APMC-SQ has a poor worst-case performance for a general query. In particular, there exists trees and general queries for which $h(c-1)+1$ cycles are needed, whereas only c cycles are needed by other schedules. We next show that algorithm APMC-GQ uses at most c cycles for any general query.

A general query results in a partial traversal of the tree in which a number of paths are to be explored. Paths may overlap, with each path starting at the root and ending at an internal node or a leaf. Because the nodes in the subtree rooted at a root-node are arranged in preorder in the same channel, all paths sharing a common root-node can be traversed in the same cycle. We view such a set of paths as one path. The number of paths containing different root-nodes or ending at a *cp*-assigned node can have significant impact on the latency. Indeed, it is not hard to show that each such path may require one broadcast cycle. We say a query requires an m-search if the query processing results in a partial traversal containing m paths ending at root-nodes or *cp*-assigned nodes. Fig. 6 shows a 4-search which has two paths ending at *cp*-assigned nodes, b_1 and c_5, and two paths endings at root-nodes, d_4 and c_7.

We next show that the broadcast schedule generated by algorithm APMC-GQ and the given client protocol allow a general query resulting in an m-search to be executed within $\min\{m, c\}$ cycles. We first give the bound of c cycles.

Lemma 4. *For a schedule generated by algorithm APMC-GQ, a client executes a general query within c broadcast cycles.*

Proof. Consider a query resulting in an m-search. Assume query processing is currently operating on the ith cycle, $i \geqslant 1$. For each cycle already elapsed, the client has tuned in at least once. For the ith cycle, let k_i be the smallest channel index tuned into. We first show, by induction on i, that

(i) $i \leqslant k_i$ and
(ii) after the ith cycle, all nodes relevant to the query and at channels $1, 2, \ldots, k_i$ have been explored.

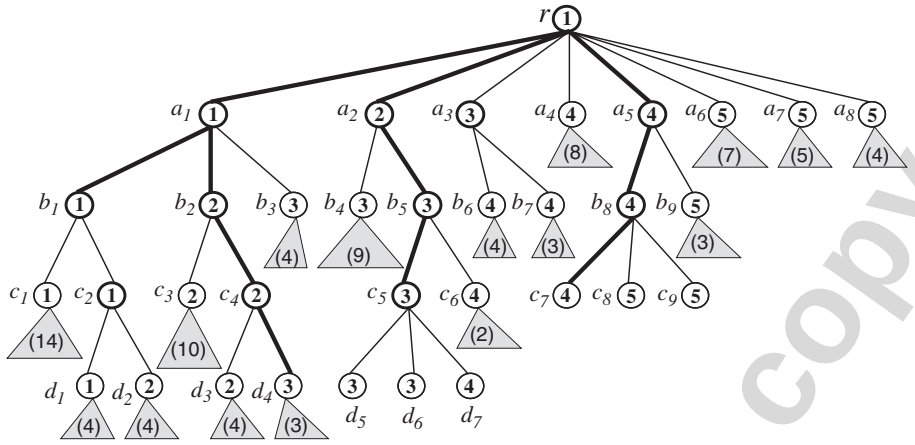


Fig. 6. A partial traversal (think lines) results in a 4-search on the index tree (see Fig. 4 for an explanation of the entries inside the nodes).

For $i = 1$, we have $k_1 = 1$. By the ancestor property, the condition $chan(u) \leqslant chan(v)$ when u is a parent of v (shown in Lemma 2), and the client protocol used, all the nodes relevant to the query on channel 1 have been explored after the first cycle. Note that $chan(u) \leqslant chan(v)$ does not require the query to return to channel 1 at some later point in time. Assume the above claim holds for cycles $1, \ldots, i-1$ and consider cycle i. Since all relevant nodes on channels $1, 2, \ldots, k_{i-1}$ have been explored after the completion of the $(i-1)$th cycle, we have $k_{i-1} < k_i$. Hence, $i \leqslant k_{i-1} + 1 \leqslant k_i$ and (i) follows. By Lemma 2 and our policy for selecting the next node to explore, all relevant nodes on channel k_i are explored during the ith cycle. Hence, all the relevant nodes on channels $1, 2, \ldots, k_i$ have been explored after the ith cycle and (ii) follows. Since $k_i \leqslant c$, the lemma follows. □

The number of cycles needed for an m-search is smaller than c when $m < c$. Consider a path ending at a root-node v. The nodes on the path from the root to v and in the subtree rooted at v can be traversed in one cycle. Every path ending at a cp-assigned node requires at most one cycle. Therefore, at least one of the m paths is processed in one cycle and at most m cycles are needed. We conclude with the following theorem:

Theorem 4. *For a schedule generated by algorithm APMC-GQ, a client executes a query corresponding to an m-search within* $\min\{m, c\}$ *broadcast cycles, where c is the number of channels.*

4.3. Client space requirement

The query processing protocol described in Section 4.1 assumes that the client can store all the relevant information acquired during the execution of the query. We briefly discuss a worst-case bound on the space requirement for the mobile client when no information about index nodes is to be lost. As the results for 1-channel broadcasts presented in [11] indicate, the actual requirements are considerably smaller, but hard to predict. Assume the tree has height h and index nodes have a fanout of B. In a 1-channel environment, there exist index trees and queries for which $\Theta(hB)$ index nodes need to be stored at a client. Intuitively, a client needs to store, for a path of length h, the siblings of the nodes on this path. All these nodes may have to be explored later. In a c-channel environment, the worst cast bound increases to $\Theta(chB)$. It is achieved, for example, by a search that generates c cp-assigned nodes when processing root r. Then, it is possible to schedule nodes on channels so that c paths are explored simultaneously, with each path requiring $O(hB)$ nodes to be stored. In some sense, this corresponds to a breadth-first-search traversal with a "width" of c.

If a client does not have enough space to store all nodes to be explored later, it must employ a

mechanism to manage the available space. Deleting information results in an increase in tuning time and latency. Approaches for deciding on which index nodes to keep and which to delete are described in [11]. The techniques for managing space are identical to methods used for handling queries that start executing in the middle of a cycle. The next sections discusses solutions which apply to these two problems.

4.4. Starting a query during a cycle

The client protocol described assumes that query processing starts at the beginning of a cycle. In this section we sketch solutions for starting query execution in the middle of the cycle. As mentioned in the previous section, these solutions are also relevant to managing limited space at a client.

A trivial solution is to simply wait for the beginning of the next cycle. Doing so assumes that the beginning of the next cycle can be determined. This approach maintains minimum tuning time in an asymptotic sense, but increases the latency. In addition, this solution is not helpful for managing limited space at a client. In a 1-channel environment, another approach is for a client to tune in when query processing starts and keep on tuning in at consecutive cycle positions until a more selective tuning is possible (from the information about the tree that becomes available). In a c-channel environment, a client needs to decide also on the channel number. One possibility is to start with channel number 1 and tune in at channel number 1 until a more selective decision is possible. A second possibility is to use one of the channels, say channel c, as a "look-ahead" channel. We then use only $c-1$ channels for scheduling the index tree, but we can significantly improve performance when data is lost due to limited space or the query starts in the middle of the cycle. The details of this approach are given next.

Assume the tree has been scheduled for broadcast on $c-1$ channels, using one of the algorithms we presented. For any position i, channel c contains information about the nodes broadcast in the first $c-1$ channels at position $i+1$. Each entry in the look-ahead channel also includes a pointer to the beginning of the next cycle. When a query starts in the middle of the broadcast cycle, the client uses the information in the look-ahead channel to determine which channel to tune into next. If there are two (or more) nodes in the next position to be explored, the one with smallest channel number is selected for tuning in and the others are stored by the client in T_{next}. If the entry in the look-ahead channel cannot provide any guidance, the client tunes in at the next entry in the look-ahead channel.

4.5. Multiple queries by one client

Our previous discussions assumed that a client executes one query at a time. However, clients may need to execute a number of queries simultaneously. For example, in the PLACE system [17], a client issues a number of range queries (which in the current prototype it sends to a server).

Assume a client issues multiple queries. The client again uses T_{act} and T_{next} to store the information received from the broadcast and to organize cycle positions to tune into. If only these two data structures are used, the following problem arises. When the client tunes in to receive and explore a node v, the node may be relevant to more than one query. Since the client does not know which queries need node v, we need to check node v against all currently executing queries. This approach can be improved by the client building an index on the queries. Then, after node v is explored, the index structure on the queries allows a client to efficiently determine the relevant queries. In addition to managing the different queries with respect to received nodes, the support for multiple queries also needs to adjust the rules used for selecting the channel number to tune in. In the single query situation, the client selects the node in T_{act} having smallest channel number. We point out that this policy can lead to starvation when multiple queries are executed simultaneously and corresponding adjustments need to be made.

5. Conclusions

We presented two efficient methods for scheduling a given n-node index tree for broadcast in a

c-channel environment. The generated schedules have minimum cycle length and queries are executed by tuning in over a number of cycles. Our first algorithm can handle any index tree and the generated schedule allows the execution of a simple query within one cycle. However, a general query may require $O(hc)$ cycles and thus result in a high latency. The second algorithm assumes that the tree satisfies a condition allowing a compact schedule and the generated schedule allows a general query to be executed in at most $O(c)$ cycles. For some queries this is the best possible latency in a c-channel environment.

References

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

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

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

[4] H.V. Leong, A. Si, Data broadcasting strategies over multiple unreliable wireless channels, in: Proceedings of the 1995 International Conference on Information and Knowledge Management, Baltimore, MD, November 1995, ACM, pp. 96–104.

[5] S.-C. Lo, A.L.P. Chen, An adaptive access method for broadcast data under an error-prone mobile environment, IEEE Trans. Knowledge Data Eng. 12 (4) (2000) 609–620.

[6] S.-C. Lo, A.L.P. Chen, Optimal index and data allocation in multiple broadcast channels, in: Proceedings of 2000 IEEE International Conference on Data Engineering, February 2000, pp. 293–304.

[7] K.L. Tan, B.C. Ooi, On selective tuning in unreliable wireless channels, Data Knowledge Eng. 28 (2) (1998).

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

[9] W.-C. Peng and M.-S. Chen, Dynamic generation of data broadcasting programs for a broadcast disk array in a mobile computing environment, in: Proceedings of ACM 9th International Conference on Information and Knowledge Management, ACM, Washington DC, USA, November 2000, pp. 38–45.

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

[11] S. Hambrusch, C.-M. Liu, W. Aref, S. Prabhakar, Query processing in broadcasted spatial index trees, in: Advances in Spatial and Temporal Databases—7th International Symposium, SSTD, 2001, Lecture Notes in Computer Science, vol. 2121, Springer, Berlin, July 2001, pp. 502–521.

[12] S. Acharya, R. Alonso, M.J. Franklin, S.B. Zdonik, Broadcast disk: data management for asymmetric communications environments, in: M.J. Carey, D.A. Schneider (Eds.), Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA, 22–25 May, 1995, pp. 199–210.

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

[14] Y.-I. Chang, C.-N. Yang, A complementary approach to data broadcasting in mobile information systems, Data Knowledge Eng. 40 (2) (2002) 181–194.

[15] S. Hameed, N. Vaidya, Efficient algorithms for scheduling data broadcast, ACM/Baltzer J. Wireless Network 5 (3) (1999) 183–193.

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

[17] S. Hambrusch, W. Aref, S. Prabhakar, Place: pervasive location aware computing environments, 2002, Purdue University, http://www.cs.purdue.edu/place/.