Acta Informatica 26, 597–614 (1989)

Acta Informatica
© Springer-Verlag 1989

Concurrency and Trie Hashing

W. Litwin[1], Y. Sagiv[2,*] and K. Vidyasankar[3,**]

[1] I.N.R.I.A., Rocquencourt, B.P. 105 F-78153 Le Chesney Cedex, France
[2] Department of Computer Science, Hebrew University, Givat Ram, IL-91904 Jerusalem, Israel
[3] Department of Computer Science, Memorial University of Newfoundland, St. John's, Newfoundland, Canada A1C 5S7

Summary. The *Trie Hashing* (TH), defined by Litwin[1], is one of the fastest access methods for dynamic and ordered files. The hashing function is defined in terms of a *trie*, which is basically a binary tree where a character string is associated *implicitly* with each node. This string is compared with a prefix of the given key in the search process, and depending on the result either the left or the right child is chosen as the next node to visit. The leaf nodes point to buckets which contain the records. The buckets are on a disk, whereas the trie itself is in the core memory. In this paper we consider concurrent execution of the TH operations. In addition to the usual search, insertion and deletion operations, we also include range queries among the concurrent operations. Our algorithm locks only leaf nodes and at most two nodes need to be locked simultaneously by any operation regardless of the number of buckets being accessed. The modification required in the basic data structure in order to accommodate concurrent operations is very minor.

1. Introduction

The *Trie Hashing* (TH) is one of the fastest access methods for dynamic and ordered files [7]. The hashing function is defined in terms of a *trie*, which is basically a binary tree where a character is associated with each internal node including the root. (In [4], it is defined as an M-ary tree.) This character enables the computation of a string, called *split string*, for each node. The split string is compared with a prefix of the given key in the search process, and depending on the result either the left or the right child is chosen as the next node to

* The work of this author was supported in part by grant 2545-2-87 from the Israeli National Council for Research and Development

** This research was done while visiting I.N.R.I.A., and is supported in part by the Natural Sciences and Engineering Research Council of Canada, Individual Operating Grant A-3182

[1] (*Proc. SIGMOD'81*, pp. 19–29)

Offprint requests to: K. Vidyasankar

visit. The leaf nodes point to buckets which contain the records. The buckets are on a disk, whereas the trie itself is in the core memory.

In this paper we consider concurrent execution of the TH operations. In addition to the usual search, insertion and deletion operations, we also include range queries among the concurrent operations. In terms of the number of nodes that may be locked simultaneously, our algorithm compares favourably with the concurrency algorithms for other dynamic search structures, in particular, [6, 10] for B-trees, [5] for binary search trees and [1] for linear hashing. Only leaf nodes are locked, and at most two nodes need to be locked simultaneously by any operation regardless of the number of buckets being accessed. The above mentioned algorithms use three simultaneous locks. The modification required in the basic data structure in order to accommodate concurrent operations is very minor.

In Sect. 2 we describe the properties of TH briefly. Section 3 discusses concurrency without range queries, and Sect. 4 with them. We compare our algorithm with the other algorithms mentioned above in Sect. 5.

2. Trie Hashing

2.1. File Structure

For TH, a *file* is a set of records identified by primary keys. *Keys* are character strings of fixed maximum length with characters, called *digits*, from a finite alphabet. The digits are assumed to be lexicographically ordered. Hence the set of all possible keys is totally ordered. The smallest digit of the alphabet is *space* and will be denoted "–", while the largest digit will be denoted ":". The part of the record besides the key is irrelevant to access computation. Records are stored on a disk in *buckets* that are units of transfer between the disk and the core memory. Buckets are assumed to be of fixed capacity. Each bucket has an *address*. The addresses are successive integers starting with 0.

A *trie* can be thought of as a binary tree in which a string, called *split string*, is associated *implicitly* with each internal node (including the root) and a bucket address or *nil* value is associated *explicitly* with each leaf. The search for (the record of) a key starts at the root. At each internal node, a prefix of the key is compared with the split string of that node. If the prefix is less than or equal to the split string the left branch is traversed; otherwise the right branch is chosen. The traversal stops when a leaf is reached. If the leaf contains a bucket address, then the corresponding bucket will contain the record of the key, if the record is in the file. If the leaf has a *nil* value, the record is not in the file. Thus a trie defines a (partial) hashing function that maps keys to bucket addresses. Unlike the usual hashing functions, a trie preserves the lexicographical order. That is, when the buckets are ordered according to the *inorder* traversal of the trie, if b and b' are the buckets that keys C and C' are mapped into, then $b \leqq b'$ if and only if $C \leqq C'$. Hence range queries can also be processed efficiently.

As mentioned above, a split string is associated only implicitly, with each internal node. Only a *digit field*, consisting of a pair of attributes called *digit value* and *digit number*, denoted (d, n), is associated explicitly. The digit field

of the root is fixed as (:, 0). The digit field of each other node is determined at the time of the creation of that node, by the splitting algorithm *A2* described later. It remains unchanged until the deletion of that node (in a merge operation). The split string of the root is ":", and that of each other node is computed from the digit fields of that node and of some of its ancestors. This computation is done as the trie is traversed, from the root node, in each key search operation. Thus at the expense of recomputing the (same) split string in each traversal, the storage space for internal nodes is drastically reduced since only one character (and a number) is stored irrespective of the size of the split string. This facilitates storing the entire trie in the core memory. (It is estimated in [8] that, when 6 bytes are used for each internal node, a trie size of 10 k bytes would suffice for a file growing to about 100000 records, and a 64 k byte trie would suffice for more than 750000 records. For files whose tries are too large to store in the core memory, a multilevel trie hashing scheme, where only a part (the top part) of the trie is kept in the core memory and the other parts on disk, is devised in [9]. For this paper, we assume that the entire trie is in the core memory.)

We now discuss the computation of split strings. First we introduce some terminology. Throughout this paper, internal nodes will be denoted by small letters a and b. Leaf nodes, also called *external* nodes, will be denoted by capital letters A and B. We use x and y to denote nodes which may be internal or external. A leaf with a *nil* value will be called a *nil leaf* or *nil node*. The bucket referred to by A, if any, will be denoted $bucket(A)$. A string T will denote $t_0 t_1 \dots t_m$ where each t_i is a digit and $m \leq k$, where $k+1$ is the limit on the key length, usually imposed by the file management system. We will call t_0 the first digit, t_1 the second digit, etc. The prefix $t_0 t_1 \dots t_i$ of T will be denoted T_i and also as $[T]_i$.

We will associate, again implicitly, another string called *maximal string*, denoted M, with each (internal and external) node. This will be used in computing the split string which will be denoted S. The string M will be of length less than or equal to $k+1$, The maximum length, and S will be of length $n+1$ where n is the digit number in the digit field (d, n) of that node.

A0: Computation of M and S.

1. For the root node, M equals ":".

Let a be an internal node for which $M(a)$ is already defined. Let (d, n) be its digit field, x its left child and y its right child. Each of x and y may be an internal or external node. In the following, "$\cdot$" refers to string concatenation.

2. $S(a)$ is d if n is 0, and $[M(a)]_{n-1} \cdot d$ otherwise.

3. $M(x)$ is $S(a)$, and $M(y)$ is $M(a)$ itself. □

Figure 1 shows a TH file of 31 most used English words [4, p. 482]. In Fig. 1(c), M and S values are shown next to the internal nodes, and M values next to the external ones.

2.2. Key Search

Any search starts at the root node. It then follows a path determined by comparison of a prefix of the key with the split string S of the currently visited trie

a the, of, and, to, a, in, that, is, i, it, for, as, with, was, his, he, be, not, by, but, have, you, which, are, on, or, her, had, at, from, this.

b

0	1	2	3	4	5	6	7	8	9	10
	to	or			you		her			
are	this	on	it	by	with		he			
and	the	of	is	but	which		have		at	from
a	that	not	in	be	was	i	had	his	as	for

c

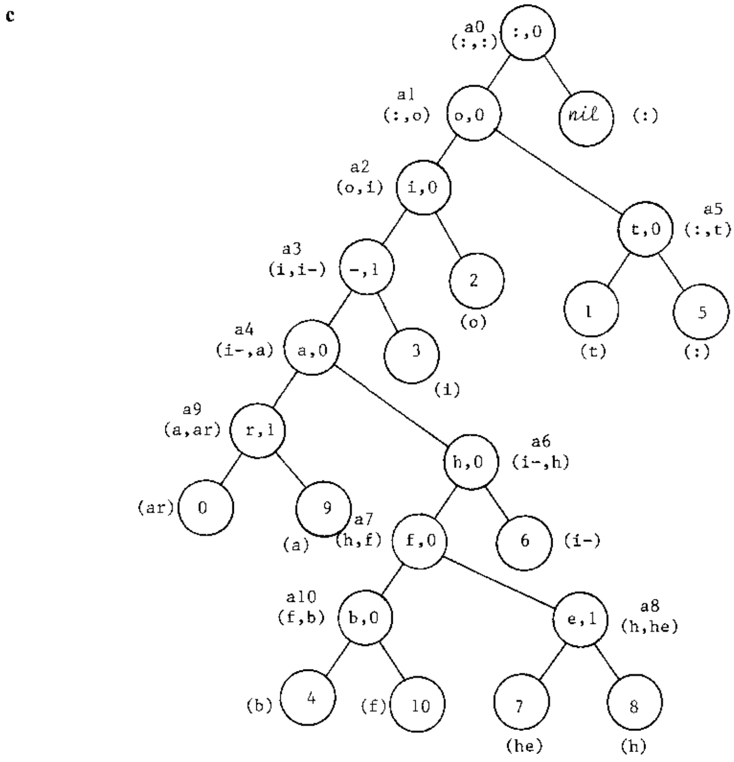

Fig. 1. Example file. **a** Insertions with words causing splits underlined; **b** Buckets; **c** Trie with (M, S) values for internal nodes and (M) values for external nodes

node, computed as per *A0*. As the result of the comparison, either the left or the right child is chosen as the next node to visit. The traversal stops when a leaf is reached. This leaf has the address of the bucket that should contain the record corresponding to the key, if such record is in the file, or a *nil* value in which case the record is not in the file. For the sake of clarity, we omit the details of the computation of S in the following description.

A1: Key Search. Let C be the searched key.

1. Start the search at the root node.

Node a	Digit number n	C_n	$S(a)$	Traversal direction
a0	0	h	:	Left
a1	0	h	o	Left
a2	0	h	i	Left
a3	1	ha	i–	Left
a4	0	h	a	Right
a6	0	h	h	Left
a7	0	h	f	Right
a8	1	ha	he	Left

Fig. 2. Search for C = "hat"

2. While the current node is an internal node a, say with digit field (d, n), traverse the left child if $C_n \leq S(a)$, and traverse the right child otherwise.

3. If the current node is an external node A, then search for the record in the bucket pointed to, if any, by the node. □

Figure 2 traces the above algorithm for C = "hat" in our example trie. It leads to $A = 7$. The key "gun" also leads to the same external node, while "s" leads to $A = 1$.

2.3. Insertion

Let C be the key to be inserted. Execute the key search algorithm *A1* to locate the external node, say A, corresponding to C. If it is a *nil* node, a new bucket containing C is created and the external node modified to contain the address of this bucket. Otherwise if *bucket*(A) is not full, the insertion simply adds the record to the bucket. If it is full, a new external node B is added to the trie and a new bucket *bucket*(B) is added to the file. The set of records that were mapped to *bucket*(A), and that of the current key C, is then split into two parts whose sizes are usually almost equal. One part, containing the records with smaller key values is assigned to *bucket*(A), and the other part containing the remaining records to *bucket*(B). The split is therefore order preserving.

The splitting algorithm is presented below. It inserts, between leaf A and its parent, a new internal node with A as the left child and B as the right child. In some cases, however, it is necessary to insert more than one internal node. The algorithm then generates some *nil* leaves also.

A2: Splitting. Let *bucket*(A) be the overflowed bucket determined through *A1*. Let SEQ be the ordered sequence of $b+1$ keys to be split, that is, all the keys in *bucket*(A) and the new key. Let *middle key* be the key in the $[(b+1)/2]$ position of SEQ. We choose some key Q as the *split key*. Normally, the split key is the middle key, but any key in SEQ, except the last one, is an admissible choice. Finally, let L be the last key in SEQ. Note that L is not necessarily the new key C. Let $q_0 q_1 \ldots q_k$ be Q and $l_0 l_1 \ldots l_k$ be L. If either string is of

length less than $k+1$, then sufficiently many spaces are padded at the right end.

1. Find the smallest i for which $Q_i < L_i$, that is, $q_i < l_i$.
2. If $i > 0$, then go to step 4.
3. Replace leaf A with a new internal node with digit field (q_i, i), q_i being the $(i+1)$st digit of Q. Attach leaf A as the left child of this node and a new external node B as the right child. Add a new bucket to the file as *bucket*(B) and move into it all the keys R in SEQ such that $R_i > Q_i$. Move the remaining keys into *bucket*(A). Return.
4. Let m be the largest value less than i such that $Q_m = [M(A)]_m$; if no such value exists, let m be -1. If $m = i-1$, then go to step 3.
5. For each $j = m+1, \ldots, i-1$ do: replace leaf A with a node with digit field (q_j, j) and attach leaf A as the left child and a *nil* leaf as the right child of this node.
6. Go to step 3. □

The steps 1 to 4 correspond to the case where only one internal node is created. Step 5 creates several internal nodes and *nil* leaves.

In our example trie, let "hat" be a key to be inserted. The search (Fig. 2) leads to bucket 7 which needs to be split. Here L is "her". If the middle key "have" is chosen as the split key Q, step (1) gives $i=1$. In step (4), since $M(A)$ is "he", $m=0$. Then in step (3), leaf 7 would be replaced with a node with digit field $(a, 1)$ and children: leaf 7 itself now containing "had", "have" and "hat" on the left, and leaf 11 with "he" and "her" on the right. This is illustrated in Fig. 3a. A case where step 5 is to be used occurs when the initial contents of leaf 7 are "had", "ham", "hate" and "hated", as illustrated in Fig. 3b. Here L is "hated" and Q is "hat". Step (1) gives i equal to 3. In step (4), $M(A)$ is "he" and hence m is 0. Step (5) gives rise to the internal nodes with digit fields $(q_1, 1) = (a, 1)$ and $(q_2, 2) = (t, 2)$. Then step (3) yields the node with digit field $(q_3, 3) = (-, 3)$.

We now define another string called *maximal key*, and denoted MX, for each (internal and external) node as follows: MX is of length equal to $k+1$ and is obtained by padding sufficiently many largest digits ":" at the end of M. From algorithms *A0*, *A1* and *A2* it can be verified easily that for each (internal and external) node x, all the keys in the buckets of the leaves of the subtrie rooted at x will be less than or equal to $MX(x)$. For each internal node the M and MX values remain unchanged but for external nodes they may change. We show now that the MX value decreases when the corresponding bucket is split. Note that Q_i is the split string of the internal node created in step 3 of *A2*, and hence also the new M value of A. Denoting the new M and MX values of A as $M'(A)$ and $MX'(A)$, we have $[MX'(A)]_i = M'(A) = Q_i < L_i \leqq [MX(A)]_i$. Thus $MX'(A) < MX(A)$. A merge operation would increase the MX value.

2.4. Range Queries

We note that when the leaves of a trie are visited according to the *inorder* traversal, the successive corresponding MX values are strictly in ascending order.

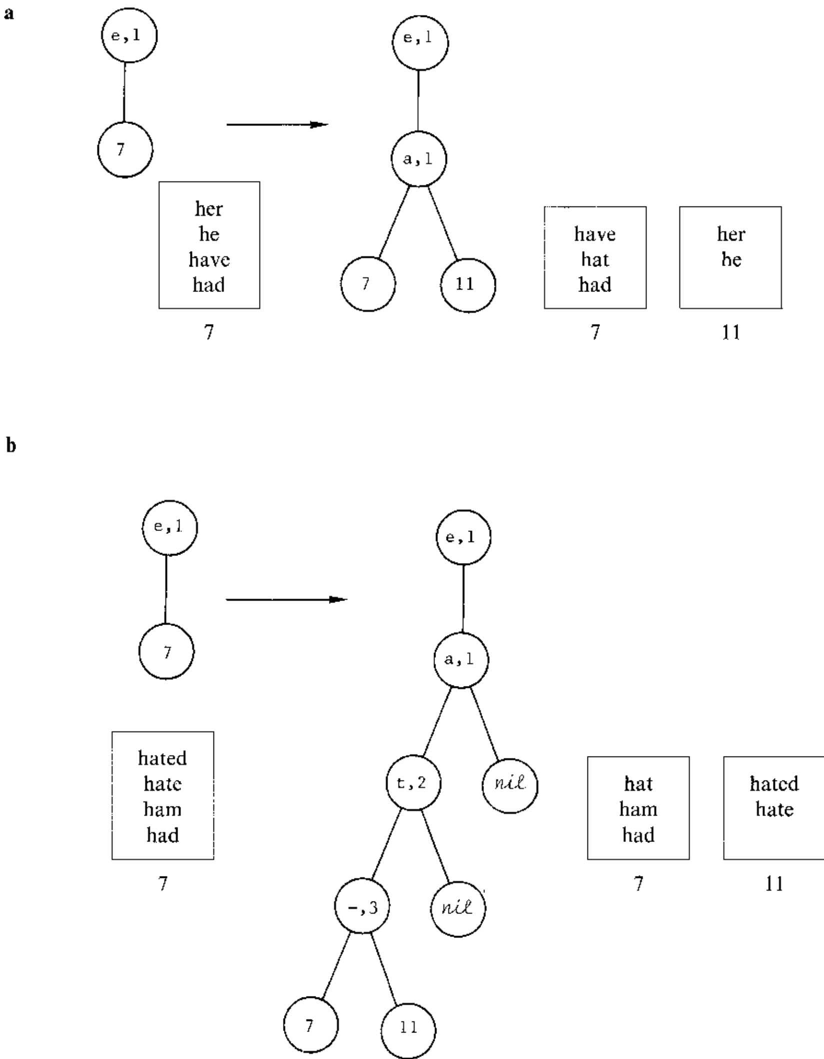

Fig. 3. Splits

In our example trie, the successive *MX* strings for the buckets are shown in Fig. 4, assuming k equals 4. Therefore all keys may be examined in ascending order through only one access of each bucket and hence range queries such as "find all records whose keys are within some interval $[C1, C2]$, $C1 < C2$" can be processed efficiently. They need to access only the buckets pointed to by the leaf corresponding to $C1$ and the leaves that follow in inorder, up to the leaf corresponding to $C2$. The corresponding sequence of addresses results also from *preorder* and *postorder* traversals, as all these traversals visit the leaves in the same order.

MX	Bucket #
ar : : :	0
a: : : :	9
b: : : :	4
f : : : :	10
he : : :	7
h: : : :	8
i– : : :	6
i : : : :	3
o: : : :	2
t : : : :	1
: : : : :	5

Fig. 4. *MX* strings for the buckets

2.5. Deletion

First the bucket that should contain the record corresponding to the given key is found using the algorithm *A1*, and the record deleted if it is there. If on deletion the bucket becomes empty, then it is released and the corresponding external node in the trie is made a *nil* node.

If both the children of an internal node *a* are leaf nodes then, if possible and if *a is not the root*, their buckets are merged into a single bucket, namely that of the left leaf. In that case, the *M* and *MX* values for the new bucket will be the same as those of the original right child. The corresponding external node is then made the child of the parent of *a*, and *a* itself is deleted.

3. Concurrency

3.1. Preliminaries

In this section, we consider concurrent execution of all the TH operations, except the range queries which we will include in Sect. 4. The following points influence our concurrency algorithm. (i) The operations that modify the trie do so only "near" the leaves. The upper part of the trie is not affected. (ii) Once an internal node is created its digit field remains unchanged until the node is deleted. (iii) The entire trie is in the core memory.

In the concurrent execution, all the operations are executed in the normal fashion. All of them start accessing the nodes of the trie from the root onwards, and have a search phase in which the appropriate nodes are located for further operations. Due to concurrent split and merge operations, the search may be prolonged. For example, suppose it is found that the left pointer of an internal node *a* refers to an external node *A*. Before *A* is accessed (that is, locked), an insertion operation might split *bucket*(*A*) thus creating an internal node in place of *A*. Then the traversal of the trie would continue from the new node

on. On the other hand, if a concurrent merge operation deletes A, and the parent a also, then we make the search process to "back up". By going up the trie one node (or more in case of further deletions due to other concurrent merges), the unmodified portion of the trie can be reached and the search continued from there on. Eventually the relevant external node will be found. We describe the search procedure after discussing the data structure.

We will assume the following *standard representation* discussed in [8] for storing the trie. The external nodes are not stored explicitly. The internal nodes are stored as an array of cells of fixed size, one cell per node. The array index will be the node address. Thus the internal node addresses are successive integers starting with 0. The bucket addresses, which are also successive integers starting with 0, are taken as the addresses of the corresponding external nodes also. In a cell, negative pointer values refer to internal nodes, and positive values to external nodes (and buckets). A typical size of a cell, discussed in [8], is 6 bytes, 2 for each pointer field and 2 for the digit field: 1 for digit value and 1 for digit number. In the 2 bytes representing a pointer value, the left most bit is taken as the sign bit. Then negative values -1 to $-(32\text{k}-1)$ refer to internal nodes (the internal node 0 will be the root node which will not be pointed to by any other node); positive values 0 to $32\text{k}-1$ refer to external nodes; and -0 is taken to mean a *nil* node. We will reserve one of the values, say $(32\text{k}-1)$ to indicate that the corresponding external node is deleted; *this is a modification in the data structure for concurrent operations.*

An external node is deleted in a merge operation, and at that time its parent internal node is also deleted. For example, let A be the left, and B the right, child external nodes of internal node a, and let the parent of a be b. Suppose that a is the left child of b. When A and B are merged, A becomes the left child of b, and a and B are deleted. We will implement this as follows: the address of A will be entered in the left pointer field of b as a positive value and $(32\text{k}-1)$ will be entered in *both* the left and the right pointer fields of a. Thus if one pointer field of an internal node contains $(32\text{k}-1)$, then the other pointer field will also contain $(32\text{k}-1)$, and the entire internal node is considered to be deleted.

The deleted nodes will have to be garbage-collected later on, after ensuring that they will not be encountered by any operation. Any of the garbage-collection methods, like those in [5], can be used. We require only that the array index of the cell representing an internal node never changes from the time of creation of that internal node to that of deletion and subsequent garbage-collection. We note also that only the deleted (internal) nodes will have to be kept for some time. The deleted buckets can be garbage-collected right away.

We assume that the individual fields of the nodes (digit field and two pointer fields, each of the same size, namely 2 bytes, in the example above) can be accessed independently and atomically. Therefore when one process is updating one pointer field, another process can update the other pointer field of the same internal node concurrently, without fear of losing the updates. Any operations in buckets are done in the core memory, that is, the bucket is read into the core memory, the necessary modifications done and the updated bucket written back (atomically) on a disk. As in [6], we assume that each process

has its own private workspace, not shared with other processes. A process reading a bucket stores it in its workspace. Hence it is possible that the same bucket is in more than one private workspace at the same time.

The search procedure is described below. It locks and returns the external node corresponding to the *input-key*. We note that the internal nodes are never accessed completely. Only the relevant fields are accessed individually. The maximal string of *current-node* is referred to by M, and that of *next-node* by M'. We stack M' also so as not to recompute it, and hence not to access the digit field again, if we have to backtrack to that node later. Whether *next-node* is deleted, and if not, whether it is internal or external, can be found from the *next* pointer value. We have to check this value again, after locking the *next-node*. This is to ensure that between the time the node was located and the time it is locked, no concurrent split or merge operation has altered that node: a split would change the *next* pointer value to refer to an internal node, and a merge would change it to refer to a deleted node. Recall that, in our implementation, the merge "deletes" both the left and the right children.

```
procedure search-external-node(input-key);
begin
current:=root;
M:=":";
childdir:="left";
M':=":";
next:=current.childdir;
found:=false;
while not(found) do
    begin
    while next-node is not deleted and is internal do
        begin
        push(stack, current, childdir, M, M');
        current:=next;
        M:=M';
        get current.digitfield and compute childdir and M';
        next:=current.childdir
        end;
    if next-node is not deleted and is external then
        begin
        lock next-node
        next: = current.childdir;
        if next-node is not deleted and is external
            then found:=true and exit
            else unlock next-node
        end;
    if next-node has been deleted then
        begin
        pop(stack, current, childdir, M, M');
        next:=current.childdir
```

```
        end
    end;
end;
```

3.2. Locking Protocols

We now describe the locking protocols. *Only external nodes are locked*, and all the locks are *exclusive* ones. The buckets are not explicitly locked. But at any time only the process that has locked A can access *bucket*(A).

We distinguish the various operations as follows. A *key-search* would refer to searching for a key. A *simple-insertion* is the insertion of a record in an already existing nonfull bucket. The *first-insertion* is inserting a record as the first record of a new bucket and modifying the corresponding external node from a *nil* node to one pointing to this bucket. A *split-insertion* is the insertion causing splitting of a bucket. Whether the split creates one internal node or several internal nodes is irrelevant for our discussion. We have distinguished the three types of insertions only for clarity. The locking sequences are exactly the same for all of them; only the operations are different. Hence the insertion process need not know the type of insertion it is going to encounter. Similar properties hold also for the two deletions which we describe now. A *simple-deletion* is the deletion of a record from a bucket resulting in a nonempty bucket. The *last-deletion* deletes the last record from a bucket, releases the bucket and makes the corresponding external node a *nil* node. A *merge* merges the left and the right leaf buckets and deletes the parent internal node. A merge operation is *not* part of a deletion operation. Even though it is usually triggered by a deletion, it is to be performed separately, not necessarily immediately after the deletion. In this paper we will not be concerned with when and how a merge operation is initiated. Any two external nodes that are children of the same internal node which is not the root are candidates for a merge operation. Several merge operations may be executed concurrently.

In protocols (1)–(6) below, only one external node is referred to. It is found and locked by the *search-external-node* procedure. That node is denoted A. Its parent is denoted a. In (7), two external nodes, two children of the same internal node, are referred to. The left child is denoted A and the right one B. Their parent is a and the parent of a is b. We do the merge only when a is not the root. Hence b always exists. As in the *search-external-node* procedure, for each of A and B, after locking the nodes we have to make sure that it is nondeleted external node; if not, (both) the lock(s) are to be released and the merge operation restarted.

Protocols

1. Key-search.
    Lock A; search for the key (and the record) in *bucket*(A); unlock A.
2. Simple-insertion.

Lock A; insert record in *bucket*(A); unlock A.

3. First-insertion.
   Lock (the *nil* node) A; create new bucket and insert record in it; let A point to the new bucket; unlock A.
4. Split-insertion.
   Lock A; do the split creating new external and internal nodes; update the appropriate pointer of a; unlock A.
5. Simple-deletion.
   Lock A; delete record from *bucket*(A); unlock A.
6. Last-deletion.
   Lock A; delete record from *bucket*(A) and release the bucket; change A to a *nil* node; unlock A.
7. Merge.
   Lock (the left child) A first and then lock (the right child) B; with each node, after locking make sure that it is a nondeleted external node, and if not (both) the lock(s) are to be released and the merge operation restarted; (on realizing that the buckets pointed to by A and B can be merged) do the merge (all records are put in *bucket*(A)); release *bucket*(B); delete and unlock B; update the appropriate pointer of b to refer to A; delete a; unlock A. □

We note that in protocol (1), A could be unlocked as soon as the bucket is brought to the core memory. The search for the key in the bucket can be done afterwards. That is, as long as the bucket is read into the core memory atomically, no lock is needed for searching for the key.

We assume that the lock manager refers to a node, that is locked or to be locked, in terms of the *current.childdir* pointer location, that is, for example as "the external node whose address appears in the left pointer field of the internal node a". (This method is possible since the array index of an internal node never changes, as per our assumption.) Therefore whether the address is a bucket address or it refers to a *nil* node is irrelevant. This method assures also unique identification of each *nil* node, when there are several *nil* nodes in the trie.

3.3. Correctness

Ordinary search, insertion and deletion operations involve only one key and hence only one external node. (The split-insertion also involves only one *already existing* node and, of course, some new nodes.) These operations may be initiated in some (time) order, but may effectively be done or completed in some other order. For example, "Insert P", "Insert Q" and "Insert R" may be initiated in that order. But actual insertions might take place in the order R, Q and P. Likewise "Insert P" and "Search P" initiated in that order may indeed be executed in the reverse order giving the result that "P is not found". We still accept the executions as correct. As long as the effects of the operations are

reflected in the search structure in some serial order, we are satisfied with the concurrent execution.

Several attempts have been made to formalize the above idea of correctness (for example, [2, 3]). In this paper, we follow the proposal in [3], which is as follows. We will call the operations on trie, such as "Insert P" and "Search P", *trie-operations*. These consist of several *node-operations* on individual nodes. Some node-operations are designated as *decisive* operations, and the others as *nondecisive* ones; the latter help to locate the node(s) where the former ones are to be performed. For example, "Insert P" requires the execution of the procedure search-external-node, involving several nondecisive "search node" operations, to locate the external node pointing to the bucket where P must be inserted, and the decisive operation which actually inserts P. The nondecisive operations do not alter the search structure and hence their free interleaving can be allowed.

Definition. The concurrency control mechanism is *correct* if it allows serializable executions of the trie-operations assuming that they consist only of the decisive node-operations. □

Theorem 1. *The locking protocols* (1)–(7) *constitute a correct concurrency control mechanism.*

Proof. We recall that once an internal node is created its digit field will not change until the deletion of the node, only its pointer values may change. Before a process tries to modify the left (right) pointer field, it must lock the left (right) child external node. Since at most one process can do that, there is no danger of more than one process modifying the same pointer field simultaneously. Furthermore, in merge operations both the left and the right child external nodes need to be locked before performing the merge and deleting the parent internal node. Hence mutual exclusion between processes modifying a pointer field of an internal node and those deleting the same internal node is guaranteed due to the locking of the respective external nodes themselves. Therefore we do not lock the internal nodes.

For all operations except merge, only one node has to be locked. For merge, always the left leaf node is locked before the right one. Hence there is no possibility of deadlock.

The *search-external-node* procedure locates the appropriate node, locks it and then verifies that the locked node is external and has not been deleted. (The only possible modification in the pointer field, by a concurrent operation, is to change it to refer to an internal node or to a deleted node.) The nodes locked for the merge operation are also verified similarly. Thus it is guaranteed that the correct nodes are locked for the decisive operations.

Ordinary search, insertion and deletion operations have *only one* decisive node-operation. Hence the serial order of these decisive operations determines the equivalent serial order of the respective trie-operations. It is straightforward to verify that the above locking protocols guarantee the mutual exclusion of the decisive operations (including the single decisive operation of the merge which locks two nodes), and assure serializability of all the trie-operations. □

4. Range Queries

In this section we consider concurrent execution of range queries also along with the other TH operations. The correctness criterion mentioned in the last section is extended to range queries as follows. The decisive operations of a range query will consist of the node-operations on the external nodes that lie within the range of the query; all other operations will be nondecisive ones. Then, here also, a concurrent execution is *correct* if the trie-operations are serializable, assuming that they consist only of the decisive node-operations.

Locking Protocol

The range queries first find the leftmost external node whose bucket is in the range with respect to inorder traversal. From there on, they read one bucket at a time, from left to right, until the rightmost bucket in the range. To find each one of them, some backtracking may be required, due to concurrent execution of the other operations. To achieve serializable order among range queries, we require the range queries to acquire locks on the external nodes and force the "conflicting" range queries to acquire locks on the common nodes in the same order (that is, no overtaking is allowed). Any node can be locked by at most one process (be it a range query or another operation) simultaneously. We use "lock-coupling" technique [3] and ensure that any range query has locks on at most two nodes at any time. These locks also provide mutual exclusion of range queries and other operations.

begin
search and *lock* the leftmost external node, say, *current-node*;
compute;
while there exists a relevant next node do
    begin
    search and *lock* the next external node (with respect to inorder traversal)
        next-node;
    unlock current-node;
    current-node:=*next-node*;
    compute
    end;
unlock current-node
end;

By "compute" we mean "perform the necessary computation with the contents of *bucket*(*current-node*), if such a bucket exists". The *search-external-node* procedure described in Sect. 3 is to be used for searching and locking the nodes. The locks need to be applied whether the external node points to a bucket or a *nil* node. The lock on the node *current-node* has to be kept until *next-node* is locked. (This is necessary only to ensure that another range query does not overtake the current one. This can be accomplished also by (i) degrading the

lock to, say, no-overtake-lock, and then allowing another range query or some other operation to acquire a (process) lock on this node and (ii) ensuring that the no-overtake-locks on each node are released in the same order as they were acquired. Here several no-overtake-locks may exist simultaneously on a node along with at most one process lock.) There exists a relevant next node if *current-node* is not the rightmost node in the range, which can be determined from its maximal string M.

Theorem 2. *The locking protocols* (1)–(7) *and the range query locking protocol constitute a correct concurrency control mechanism.*

Proof. Each operation that locks more than one node does so in the left to right order (with respect to inorder traversal of the trie). Hence there is no possibility for deadlock.

To show serializability, given a trie T and a concurrent execution involving a set P of TH operations on T, we derive a binary tree T' and a set P' of operations on T' as follows. The tree T' will contain all the nodes that were created in T. Each node of T' will have a distinct label. A node of T' with label x may correspond to an external node A of T at some stage (say stage 1). If A is split into A and B creating internal node a, then x would correspond to a, and its children, say y and z, would correspond to the new A and B respectively (stage 2). The nodes y and z are guaranteed to be in T' by its construction. Some time later, A and B may be merged into A. Then x would correspond to A again (stage 3). If A is split again into, say A and B', creating new internal node a', then x, y and z would correspond to a', A and B' respectively (stage 4). Thus the same node of T' may correspond to different nodes of T at different stages. We can see that an upper part of T' corresponds to T at some stage, with different upper parts at different stages, and the leaves of that upper part correspond to the external nodes of T at that stage.

Corresponding to each operation p on T, we define an operation p' on T' such that p' would lock the node on T' that corresponds to the node on T locked by p at that stage. For example, if p locks A in stage 1 above, then p' will lock x; if it is stage 2, then p' will lock y. Likewise, if p locks B in stage 2 or B' in stage 4, p' will lock z. Now locking node x of T' can be considered to be equivalent to locking all the leaves of the subtree of T' rooted at x *simultaneously*. Then we note that for each operation on T that is considered in this paper the corresponding operation on T' locks a set of leaves that occur consecutively in the ordering of the leaves according to the inorder traversal. Thus each operation in P' can be treated as a range query.

We now construct a directed graph G with vertices corresponding to P' and edges as follows: there is a directed edge from p'_i to p'_j if both these operations lock some leaf of T' and p'_i locks it before p'_j. Suppose two operations p_1 and p_2 on T lock a common (external) node A. Let x and y be the corresponding nodes in T' locked by p'_1 and p'_2 respectively. Then x must be either the same as y, or a descendent or ancestor of y. Hence p'_1 and p'_2 will lock a common leaf of T', and there will be a directed edge between p'_1 and p'_2. (We note that even if p_1 and p_2 do not lock a common node, p'_1 and p'_2 may lock a common leaf. One such instance is p_1 locking A in stage 1 above and p_2 locking B in

stage 2.) Then the acyclicity of G will imply the serializability of the concurrent execution of P' on T', and hence the serializability of the concurrent execution of P on T. The acyclicity of G follows from the fact that the operations P' lock the leaves of T' in the left to right order, that is, if z is to the right of y, then z is locked at the same time or after, but not before, y. We note that the locking sequences are very much similar to that of the non-two-phase locking protocol of [11] for hierarchically structured data items, also called tree protocol in [12]. □

5. Discussion

In this paper, we have given locking protocols for concurrent execution of the Trie Hashing operations. In addition to the normal search, insertion and deletion operations, we have also included range queries among the concurrent operations. Locks are applied only to external nodes. At most two nodes need to be locked simultaneously by any operation regardless of the number of buckets it accesses. Thus our algorithm compares favourably with the concurrency algorithms for other dynamic search structures, in particular [10] for B-trees, [5] for binary search trees and [1] for linear hashing, each requiring upto three simultaneous locks. (Three nodes are locked in the first two, and the "root", consisting of variables *level* and *next*, and two buckets are locked in the last one, where the appropriate hashing function is defined as:

$$bucket \leftarrow h_{level}(key);$$
$$\textbf{if } bucket < next \textbf{ then } bucket \leftarrow h_{level+1}(key).$$

We note that linear hashing does not allow efficient processing of range queries.

The techniques employed in our algorithm for enhancing concurrency have been used in the other algorithms also:

(i) The merge operation is decoupled from deletion and "postponed" to a later time in our algorithm. Splits and merges are postponed in [1, 10] and rotation in [5].

(ii) The idea of backtracking from a deleted or a wrong node is used in [1, 5, 10] also.

(iii) The lock-coupling technique has been used in [1, 3, 6].

The enhanced concurrency (namely, locking at most two nodes simultaneously) and the simplicity that we have been able to achieve in our locking protocols seem to be due to the inherently simpler characteristics of the (sequential) TH operations compared to those of the others.

(a) In binary search trees a search may terminate in any node. New nodes are inserted only in the leaf level. But any node may have to be deleted. Hence the tree is rotated so that the node to be deleted becomes a leaf. Rotation is done to balance the tree also. In tries, all searches terminate in the leaf nodes. Node insertions as well as deletions occur only in the leaf level, the affected internal nodes being parents of external leaf nodes. The trie is not balanced. (Clearly the cost of processing a few extra nodes residing in the core memory

is insignificant compared to that of accessing the buckets from the disk.) Hence rotation procedures, requiring more simultaneous locks (three in [5]), are not needed. The fact that all the "critical" operations occur in the leaf level enables *not locking* the internal nodes of the trie in our protocols.

(b) The simplicity of TH over B-trees is primarily with internal nodes.

(i) In TH, merges can occur only in the leaf level. In B-trees (sibling) internal nodes in any level can also be merged, and hence need to be locked during the merge.

(ii) In TH, only the pointer values of an internal node can keep changing. Our algorithm is able to make these changes just by locking the appropriate leaf nodes, and without locking the internal node itself. In B-trees (the bucket corresponding to) an internal node is treated as a single atomic unit, and hence is locked to perform any updates.

(iii) The entire trie and hence all the internal nodes are in the core memory. Hence when a wrong or deleted node is encountered, it is easy to go up the trie and come down to the current node. The same approach in B-trees would require additional disk accesses, to bring the higher level nodes from disk to the core memory. To avoid these accesses, at least in the case where the correct node is to the right of the present node, a link to the next node (in the same level) is provided in the data structure in [6, 10]. Such a link is unnecessary in tries, adding to the simplicity of our protocols.

(c) In linear hashing, the root values *level* and *next* determine the hashing function. Hence when one process is using the root, no other concurrent process can change the root. This curtails the concurrency considerably. For example, a search process has to lock the root until the bucket is located and locked. Then a merge operation, which again has to lock the root and the two buckets to be merged, cannot operate concurrently with a search. Also at most one restructuring operation (split or merge) can be executed at any time. The hashing function in linear hashing can be thought of as a tree of height one, all the buckets attached directly to the "root". Any restructuring operation changes the root (values *level* and/or *next*) and hence the entire tree. In contrast, in TH, a restructuring operation changes only a part of the trie. Hence several restructuring operations, each changing different parts of the trie, can be executed concurrently, along with other operations accessing the unmodified portions of the trie as well.

We note that we have made almost no change at all in the TH data structure (of [8]) in order to accommodate concurrent operations. The only significant change is using a pointer value to denote a deleted node. One other change is adding the root node with digit field (:, 0). The left child of our root node is the root node in [8]. This change is not essential and made only for convenience to check easily whether an internal node is a root node (while deciding on a merge operation). Clearly our changes are very minor compared to those made in the other algorithms: for example, storing "high value" [6, 10] or "locallevel" value [1] in buckets, and using deletion bits [6, 10] or color field [5] to denote deleted nodes.

The advantages of TH over other access methods for dynamic and ordered files are discussed in [8] for nonconcurrent executions. Our algorithm maintains

this edge in concurrent executions also since the possible overhead in any operation, due to concurrent execution, is longer search phase(s) involving more accesses of the trie in the core memory, but no increase in the number of bucket accesses.

Acknowledgement. It is our pleasure to thank the referees whose various comments improved the presentation of this paper considerably.

References

1. Ellis, C.S.: Concurrency in Linear Hashing. ACM TODS **12**, 195–217 (1987)
2. Ford, R., Calhoun, J.: Concurrency Control Mechanisms and the Serializability of Concurrent Tree Algorithms. Proc. ACM PODS'84, pp. 51–60
3. Goodman, N., Shasha, D.: Semantically-based Concurrency Control for Search Structures. Proc. ACM PODS'85, pp. 8–19
4. Knuth, D.E.: The Art of Computer Programming. Vol. 3: Sorting and Searching. Reading, Mass.: Addison-Wesley 1974
5. Kung, H.T., Lehman, P.L.: Concurrent Manipulation of Binary Search Trees. ACM TODS **5**, 354–382 (1980)
6. Lehman, P.L., Yao, S.B.: Efficient Locking for Concurrent Opertions on B-Trees. ACM TODS **6**, 650–670 (1981)
7. Litwin, W.: Trie Hashing. Proc. ACM SIGMOD'81, pp. 19–29
8. Litwin, W.: Trie Hashing: Further Properties and Performance. Foundations of Data Organization. New York: Plenum Press 1986
9. Litwin, W., Zegour, D., Levy, G.: Multilevel Trie Hashing. Proc. Extending Database Technology Conference, Venice, 1988
10. Sagiv, Y.: Concurrent Operations on B-Trees with Overtaking. J. Comput. Syst. Sci. **33**, 275–296 (1986)
11. Silberschatz, A., Kadem, Z.: Consistency in Hierarchical Database Systems. J. ACM **27**, 72–80 (1980)
12. Ullman, J.D.: Principles of Database Systems. Rockville, Md: Computer Science Press 1982

Received October 7, 1987/March 22, 1989