CONCURRENT OPERATIONS IN LARGE ORDERED INDEXES(1)

Y. S. Kwong(2) and D. Wood(2)

In this paper we present a new solution to the problem of supporting concurrent operations in B-trees, using a technique called side-branching to provide a higher degree of concurrency than previous solutions in the same category. We also propose a new data structure, T-trees, as an alternative to B-trees for representing very large ordered indexes in database applications. A T-tree offers not only an elegant structure for storing a huge amount of data, it also permits a consistent view and uniform treatment of concurrency at both the page tree and the page node levels.

---

(1) : Work supported under Natural Sciences and Engineering Research Council of Canada Grants Nos. A-3042 and A-7700.

(2) : Unit for Computer Science, McMaster University, Hamilton, Ontario, Canada, L8S 4K1.

1. INTRODUCTION

With the ever-growing influence of databases in our everyday life, their effective organization and maintenance which guarantee efficient access are becoming increasingly important. Balanced search trees such as 2-3-trees and AVL-trees are usually used for a small amount of data to ensure logarithmic performance in searching, insertion and deletion. However, when an enormous amount of information is involved, data are often organized into pages and stored in direct access media such as disks. If these pages are linked together as a multiway tree and are transferred into main memory when necessary, we refer to such a structure as a page tree. One such versatile data structure called the B-tree was proposed in [Bayer and McCreight] for representing large ordered indexes of dynamic random access files. In view of the rapid decline of hardware costs it is very unnatural and much too inefficient to restrict large databases to sequential operation. The problems of how best to introduce concurrency and to what extent and at what cost should the degree of concurrency be maximized are challenging and interesting topics for investigation. Their satisfactory solution will have considerable practical implications in concurrent programming, database design and applications.

In this paper we investigate the problem of permitting an arbitrary number of processes to operate concurrently on a database stored as a B-tree structure. A new solution is proposed together with some techniques which might also be applicable for handling concurrency in other data structures. We also show that this solution can be extended to a tree-of-trees structure called the T-tree without any increase in overhead but allowing a uniform view and consistent treatment of concurrency at both the page tree and the page node levels.

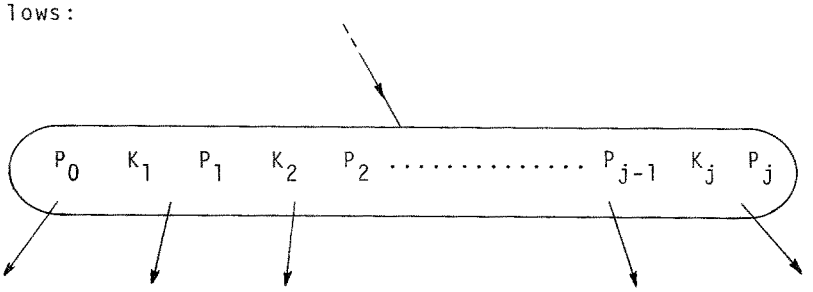
2. PRELIMINARIES

In this section we first define the B-tree structure and then present some preliminaries about the problem of supporting concurrent operations in B-trees.

2.1. B-TREES

Definition 1: A B-tree of order m (≥ 3) is a page tree which satisfies the following properties: (1) Every node has at most

m sons. (2) Every node, except for the root and the leaves, has at least $\lceil m/2 \rceil$ sons. (3) The root has at least two sons. (4) A nonleaf node with j+1 sons contains j keys and can be represented as follows:



where the P_i's are pointers to its sons and the K_i's are keys such that all keys in the subtree pointed to by $P_{i-1}(P_i)$ are less than (greater than) K_i for $1 \leq i \leq j$. (5) All leaves appear at the same level and they are the same as nonleaf nodes except that all pointers are null.

In the literature (e.g. [Bayer and McCreight], [Knuth]) B-trees are often assumed to have sequential allocation of keys and pointers in page nodes. In this paper we shall also adopt this convention. However, it should be noted that B-trees as used here correspond to B-S-trees in [Kwong and Wood, 1978].

2.2 THE CONCURRENCY PROBLEM

We want to allow an arbitrary number of processes to operate concurrently on a database stored as a B-tree. These processes are assumed to be asynchronous, each of which is progressing at a finite, but undetermined speed and is performing one of the following operations:

SEARCH (K,T): to determine whether the key K is in some node of the tree T.

INSERT (K,T): to add the key K to the tree T if K is not already present in the tree.

DELETE (K,T): to remove the key K from the tree T if K is present in the tree.

The problem we are facing is how to support concurrent operations so that the degree of concurrency can be increased as far as is reasonable subject to the following constraints:

(1) The integrity of the data and the structure of the B-tree must be preserved.
(2) There should be no major modifications of the data structure.
(3) Implementation details must be invisible.

For obvious reasons condition (1) must be satisfied. It should be noted that condition (2) only stipulates that no major modifications of the B-tree structure are permitted, however, it does not rule out minor and natural extensions which simplify the solutions to difficult concurrency problems. Condition (3) is imposed to enforce the distinction between an algorithm and its implementations.

Basically a SEARCH process reads the nodes along a path from the root towards some particular leaf node. It makes no changes whatsoever to the data or the structure of the tree. For this reason we feel justified in calling it a pure-reader. However, an INSERT or a DELETE process may modify the data as well as the structure of the tree. We therefore call it an updater. Usually an updater has to go through the following two phases:

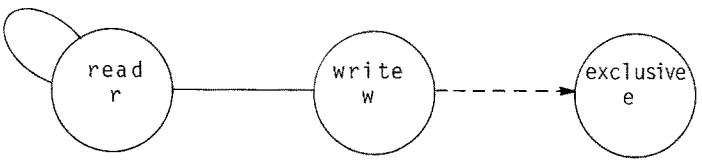
(1) searching: read in a top-down manner from the root in search of the appropriate place for adding or removing a key;
(2) restructuring: add or remove a key and then rebalance the tree if necessary.

An updater in its searching phase is referred to as an updating-reader, to distinguish it from a pure reader which is also reading. However, once it gets into its restructuring phase, we call it a writer. For convenience, we also call a process a reader if it is either a pure-reader or an updating-reader and the path from the root to a leaf as determined by a process on its passage to the frontier the access path of the process.

2.3 CONCURRENCY CONTROL

In order to regulate concurrent accesses we need a concurrency control which associates with each node several types of locks to be used by processes operating on the node and a compatibility and convertibility graph (CCG) specifying a relation which must hold among the various types of locks. The CCG is a directed graph whose vertices are labelled with the different types of locks and whose edges are used to represent the compatibility

and convertibility relation (CCR) among the locks. For any two vertices α and β (which need not be distinct) a solid edge directed from α to β means that a process with a β-lock on a node would permit another process to put an α-lock on that node. A broken edge from α to β indicates that a process holding an α-lock on a node may convert it into a β-lock. An undirected edge between α and β represents two directed edges from α to β and from β to α. In our solution we shall use the three types of locks (read-locks, write-locks and exclusive-locks) proposed in [Bayer and Schkolnick] with its CCG as shown below. In the sequel we shall refer to these locks as r-locks, w-locks and e-locks respectively.



A process can manipulate the locks on a node N via three types of indivisible operations, viz. lock, unlock and convert:

(1) α-lock (N): If the granting of an α-lock on node N does not violate the CCR then the process is allowed to continue; otherwise, it is put to sleep.

(2) α-unlock (N): the process simply wakes up some sleeping or suspended process in accordance with the CCR and some fair scheduling discipline.

(3) convert (N, α, β): It denotes a request to change an α-lock on a node N into a β-lock. If the conversion can be done without violating the CCR then it is performed. Otherwise the operation is undefined.

3. CONCURRENCY IN B-TREES

In this section we shall propose a new solution for supporting concurrent operations in B-trees. Basically our solution is an improved version of the solutions in [Bayer and Schkolnick] and [Ellis], using a new restructuring technique which we call side-branching. As we shall see it supports a fairly high degree of concurrency among processes. Moreover, it requires no modification to the B-tree structure and all implementation details remain

invisible, thereby satisfying all criteria of a desirable solution specified in Section 2.2. For brevity we shall consider concurrent searching and insertion only, however, it is quite straightforward to extend our solution to permit deletion.

As in [Samadi], [Parr], [Bayer and Schkolnick] and [Ellis], our solution makes use of a simple observation: for an updater U operating in a sequential environment there exists a node which is the root of a subtree beyond which all changes in data and structure due to U cannot propagate. We call it a safe node for U. We also call the safe node which is deepest in the tree as determined by U in its passage to the frontier the deepest safe node (or dsn) for U and the path from it to a leaf determined by U the scope of U.

Definition 2: A node in a B-tree is insertion-safe or i-safe if it is unsaturated, i.e. it has less than m-1 keys.

In the following we first give an informal description of our solution. A SEARCH process is a pure-reader which searches down the tree for the argument key K by first r-locking the root and then on its passage to the frontier it r-unlocks a node only after it has r-locked its son. On the other hand an INSERT process first locks its scope by w-locking the root and then on its passage to the frontier, the appropriate nodes are w-locked and examined. When a safe node is found, all its ancestors are then w-unlocked. If the argument key K is found the process is terminated after releasing all its locks on nodes. Otherwise on reaching a leaf node its scope must be w-locked and it can then begin restructuring. We shall refer to these methods employed by the pure- and updating-readers as lock-coupling techniques.

In all proposed solutions in the literature, a key and a pointer are added to a node at each restructuring step, starting at the leaf node of the scope. If the node is not i-safe it becomes oversaturated and is split into two "halves". This leads to a key and a pointer being pushed upward to be added to the father node. After adding to its dsn the writer will then have completed its restructuring. This common approach requires two additional fields per node for the extra key and pointer since a node can become oversaturated. Moreover, in adding a key and a pointer to a saturated node, keys and pointers must be shifted to make space for the new entries. Such shifting is often redundant because "half" of

the node has to be copied into a new node and deleted soon afterwards.

Instead of actually adding a key and a pointer to a node at each restructuring step, a writer in our solution uses the key to determine whether it should be added to the left or right "half" It then copies the appropriate "half" into a newly created node to which the key and pointer are added and pushes a key and a pointer upward while leaving the node in its scope intact. The writer continues restructuring upward in this way until the leading node of its scope is reached. It then adds a key and a pointer to this node which must be i-safe. Observe that before adding to the safe node the writer is simply reading keys and pointers in the nodes of its scope and building a side-branch which in inaccessible to other processes. Obviously pure-readers on its scope are not affected. We shall refer to this restructuring technique as side-branching. It should be noted that when a key and a pointer are added to the dsn by a writer, we only require that the node is being w-locked by the writer, which means that there can be many concurrent pure-readers on that node. This is made possible by using a generalized version of the RW-addition technique which will be discussed later.

After adding the side-branch to the leading node of the scope, the remaining task for the writer is to remove the appropriate "half" of every node in its scope other than the leading one. The writer can drive off pure-readers operating on a node N or waiting in its associated queue by executing the following operations: convert (father of N, w, e); e-unlock (father of N); convert (N, w, e). It can then remove the appropriate "half" of that node and repeat this for every node in the scope. The major steps of an INSERT process can be summarized as follows:
(i) w-lock its scope. (ii) Build a side-branch and add it to its dsn. (iii) Remove the appropriate "half" of each node in its scope.

Note that four major techniques, viz. lock-coupling, driving off readers, side-branching and RW-addition, are used in our solution. The first two techniques were used in various proposed solutions in the literature but have not been explicitly identified. The most important technique that we introduce is side-branching, which is based on the simple notion of making extra

copies of keys and pointers while leaving nodes in the scope of a writer intact until the very last moment. The RW-addition technique that allows an arbitrary number of pure-readers to operate on a node to which a writer is adding a key and a pointer concurrently was first introduced in [Ellis] to increase "intra-nodal" concurrency among processes. In their operations on a node readers and writers are required to proceed in opposite directions under a rather severe restriction -- the reading (writing) of a key and a pointer must be indivisible. In our version of RW-addition we show that this restriction can be removed by requiring pure-readers to perform repeated reading whenever necessary. Note that the RW-addition technique specified below can be viewed as an application of the results in [Lamport] to the particular situation of B-trees.

```
procedure READ (K, node);
comment determine whether K is in the node and if not then return
   pointer to the appropriate son which should be searched;
begin
   comment search from left to right and remember the key at each
      step;
   i := 1; key := K_i;
L: do (K > key and K_i not rightmost)
      i := i+1 & key := K_i;
   if K = key then report K is found else
   begin
      comment read the pointer and check its version number;
      if K < key then
         begin
            pointers := P_i-1;
            if key ≠ K_i then i := i+1 & go to L
         end
      else
         begin
            pointer := P_i;
            if K_i not rightmost then i := i+1 & go to L
         end;
      report K not found; return pointer
   end
end
```

```
procedure ADD (K, P, node);
comment add the key K and the pointer P to the node;
begin
   comment search from right to left;
   let i be the index of rightmost key;
   do (K < K_i and i ≥ index of leftmost key)
      comment shift by copying pointer first and then key;
      P_{i+1} := P_i;  K_{i+1} := K_i;  i := i-1
   od;
   comment add K and P;
   P_{i+1} := P_i;  K_{i+1} := K;  add P
end
```

We can now formally present the algorithms for searching and insertion. For simplicity we assume that the root of a B-tree is always i-safe.

```
procedure SEARCH (K,T);
comment determine if the key K is in some node of the tree T;
begin
   comment search from top down using lock-coupling technique;
   r-lock (root of T);
   current := root of T;
   READ current to determine whether K is in it and if not then
      find the appropriate son;
   do (current not a leaf and K not found)
      r-lock (son);  r-unlock (current);
      current := son;  READ current
   od;
   report if K has been found;
   r-unlock (current)
end
```

```
procedure INSERT (K,T);
comment add the key K to the tree T if K is not already present;
begin
   comment search for K and proceed to lock scope by lock-coupling;
   w-lock (root of T); current := root of T;
   DETERMINE the appropriate son and whether K is in current;
   do (current not a leaf and K not found)
      w-lock (son);  current := son;
      if current is i-safe then w-unlock ancestors of current;
      DETERMINE
   od;
   comment check if K is already present;
   if K has been found then
      w-unlock curent and its ancestors & report K already present
   else
   begin
      comment restructure by side-branching;
      do (current not i-safe)
         CREATE side branch & current := father of current;
      ADD side branch to current;
      comment remove redundant copies of keys and pointers;
      convert (current, w, e);  e-unlock (current);
      do (current not a leaf)
         current := son of current on access path;
         convert (current, w, e);
         REMOVE appropriate "half" of current;
         e-unlock (current)
      od
   end
end
```

In our algorithms for searching and insertion, five procedures are used, viz. READ, ADD, DETERMINE, CREATE and REMOVE. READ and ADD are used to permit RW-addition and have already been specified. The other three procedures are used either by an updater in a sequential environment or by a reader in the presence of concurrent pure-readers but no other updaters. Hence they can be specified in a straightforward manner regardless of concurrency.

4. EXTENSION TO T-TREES

It was pointed out in [Kwong and Wood, 1978] that a major deficiency of the B-tree structure is that "intranodal" operations, i.e. those operations within a page node such as searching and key addition (removal) that do not cause overflow (underflow), require O(m) time. This could be a serious problem when there are several hundred keys in a page node -- a situation which is not unusual. However, all these operations can be done in O(log m) time if the page nodes are organized as trees. We are therefore led to the tree-of-trees structure called the T-tree. Besides guaranteeing logarithmic performance for intranodal operations, this tree-of-trees structure is very attractive in supporting concurrent operations: when multiple users are allowed to search and update in parallel, the searching and updating algorithms used at the page tree level can also be employed, with some simplifications, at the page node level. In this section we shall examine a subclass of T-trees which offers a uniform view of the data structure and investigate how to extend our solution in Section 3 to this subclass in order to achieve a consistent view and a uniform treatment of concurrency at both the page tree and the page node levels.

Definition 3: Let $f:N \to N$ be a function on the set of positive integers such that $2 \leq f(m) \leq \lceil m/2 \rceil$ holds for all $m \geq 3$. An f(m)-tree is a page tree which satisfies properties (1), (3), (4), (5) of a B-tree in Definition 1 and also the following: (2') Every node, except for the root and the leaves, has at least f(m) sons.

Basically, f(m)-trees are a generalization of B-trees in which the minimal number of sons a nonleaf node (except the root) must have is between 2 and $\lceil m/2 \rceil$, depending on the function f. Note that no assumptions are made about the internal organization of page nodes.

Definition 4: A tree-of-trees (or simply a T-tree) of order m is an f(m)-tree in which all page nodes are represented by balanced trees of a particular type. Thus we obtain, for example, T_{AVL}- and T_{2-3}-trees when AVL-trees and 2-3-trees respectively, are used.

In our solution for B-trees, r-locks and w-locks are employed by pure-readers and updaters respectively in their operations on the page tree. Because of side-branching, w-locks are converted by an updater into e-locks only at the final steps of restructuring. Note that at any time there can be at most one page

node being e-locked by an updater and most nodes in the page tree are usually not e-locked. Intranodal concurrency is therefore usually permitted for any nodes among an arbitrary number of pure-readers and at most one updater. Although no two updaters can operate on a node concurrently and this might appear to be an undesirable restriction, however, it is this special feature that allows the searching and updating algorithms at the page tree level to be applied within page nodes to support intranodal concurrency without any increase in overhead, provided that page nodes are organized as B-trees. Observe that our solution in Section 3 requires a process to lock a page node before it can operate on it. Therefore, an updater will always have the whole B-tree inside the page node w-locked before it starts adding (removing) a key and a pointer to (from) the node. Obviously it is not necessary for an updater to even attempt to carry out any further w-locking. Because of side-branching an updater can allow pure-readers to be present in the page node until its side-branch has been completed. At this point the updater must drive away all pure-readers before removing redundant copies of keys and pointers. This can be done easily by using the concurrency control already available -- converting the w-lock on the page node into an e-lock. Since it is unnecessary for pure-readers to provide a means for a concurrent updater to get rid of them from nodes within page nodes, no locking inside page nodes, let alone lock-coupling, is necessary for pure-readers. From this informal description it should be clear that algorithms for searching and updating developed for a page tree can also be used, after simplicication, within page nodes. We must emphasize that this uniform treatment of concurrency at the page tree and the page node levels is achieved without introducing another level of concurrency control for intranodal operations. As we shall show the algorithms for searching and updating at the page tree level need only slight modifications.

Definition 5: A T_B-tree of order (m,n) is a T-tree of order m in which the page nodes are organized as B-trees of order n.

Note that the number of null pointers in the leaves of a B-tree is always one more than the number of keys in the tree. When a page node is organized as a B-tree we replace these null pointers with pointers to the sons of the page node, giving a very natural representation of keys and pointers.

In the following we shall present the algorithms for searching and insertion in a T_B-tree. Consider applying the algorithms SEARCH and INSERT developed in Section 3 to the page tree of a T_B-tree. It should be evident that we need to modify only those procedures in the algorithms which assume sequential allocation within page nodes. Hence the new algorithm for searching in a T_B-tree is identical to SEARCH except that READ must be replaced by t-READ, where the prefix t indicates that t-READ is a "tree" version of READ. Basically t-READ can be derived from SEARCH by discarding all locking and unlocking operations.

```
procedure t-READ (K,T);
comment determine whether K is in some node of T and if not
   return a pointer to the son which should be searched;
begin
   current := root of T;
   READ current to determine whether K is in it and if not then
      find the appropriate son;
   do (current not a leaf and K not found)
      current := son & READ current;
   if K has been found then report K is found
   else report K not found & return son
end
```

We can derive the algorithm for insertion in a T_B-tree from INSERT as follows: (1) Substitute the procedures DETERMINE, CREATE and REMOVE with t-DETERMINE, t-CREATE and t-REMOVE. (2) Replace {ADD side-branch to current; convert (current, w, e); e-unlock (current)} with {t-ADD side-branch to current}.

Recall that procedure DETERMINE is the same as READ except that it is used by a writer on a node that it has w-locked. Therefore t-DETERMINE is the same as t-READ except that READ is replaced by DETERMINE for efficiency.

As in CREATE and REMOVE, the procedures t-CREATE and t-REMOVE are employed by a writer operating either in a sequential environment or as a reader in the presence of concurrent pure-readers but no other writers. Hence they can be specified in a straightforward manner provided that the new page node created by t-CREATE and the page node left behind after its appropriate "half" has been removed by t-REMOVE are at least minimal. As noted in [Kwong and Wood, 1978] t-CREATE and t-REMOVE for T_{2-3}-trees can be

very simple provided that $f(m) = 2^{\lceil \log_3(m+1) \rceil - 1}$. The former consists of copying the left (or right) subtree from a page node into a new page to which a key and a pointer are then added, whereas the latter consists of simply removing the subtree which has been copied and forming a new root if necessary.

Basically procedure t-ADD can be derived from INSERT by discarding all locking and unlocking operations. In addition, checking for the presence of the argument key is not necessary because this procedure will be used only when the key is absent. After a side-branch has been built within a page node, all pure-readers must be driven away before redundant keys and pointers can be removed. As indicated earlier, this can be carried out by converting the w-lock on the page node into an e-lock. To reduce e-locking this convert operation is combined with that at the page tree level, thereby justifying step (2) in our derivation of an insertion algorithm from INSERT.

```
procedure t-ADD (K,P,T);
comment add key K and pointer P to the tree T inside a page node;
begin
   comment search for the appropriate place to add K and P;
   current := root of T;  DETERMINE the appropriate son;
   do (current not a leaf)
      current := son & DETERMINE
   comment restructure by side-branching
   do (current no i-safe)
      CREATE side-branch & current := father of current;
   ADD side-branch to current; convert (T, w, e);
   do (current not a leaf)
      current := son of current & REMOVE appropriate "half" of
      current;
   e-unlock (T)
end
```

Two important features of side-branching as a restructuring technique inside page nodes organized as B-trees should be noted:

(1) For restructuring at the page tree level, side-branching might cause extra page fetches because an updater needs an extra pass over its scope. However, this disadvantage vanishes when side-branching is applied within page nodes since no page fetches are involved.

(2) If restructuring within a page node is carried out by adopting any other proposed solution in the literature then either the page node must be e-locked during the entire restructuring phase or the page node must be e-locked and e-unlocked many times.

5. CONCLUDING REMARKS

In this paper we have given a new solution to the problem of supporting concurrent operations in B-trees, using a technique called side-branching. We have also proposed a new data structure, namely T-trees, as an alternative to B-trees for representing large ordered indexes and illustrated how our solution for B-trees can with slight modification be applied to T-trees so that a consistent view and treatment of concurrency is obtained.

How does our solution for B-trees compare with proposed solutions in the literature? As shown in [Kwong and Wood, 1979] all but one solution in the literature can be classified into one of two categories (types 1 and 2) depending on whether the number of nodes that must be locked at each restructuring step is unbounded or bounded, respectively. The solution presented here is of type 1 and it provides a higher degree of concurrency than all previous type 1 solutions. In [Kwong and Wood, 1979] a type 2 solution is also given. However, it is not at all clear whether a type 2 solution is always better than a type 1 solution, unless it is reasonably simple and no more than 2 or 3 nodes need to be locked at each restructuring step. In the case of T-trees, even if a type 2 solution is adopted at the page tree level, it still appears preferable to adopt a type 1 solution to support intranodal concurrency so that an extra level of concurrency control is not necessary.

REFERENCES

Bayer, R. and McCreight, E.: Organization and maintenance of large ordered indexes. Acta Informatica 1 (1972), pp. 173-189.

Bayer, R. and Schkolnick, M.: Concurrency of operations on B-trees. Acta Informatica 9 (1977), pp. 1-21.

Ellis, C. S.: Concurrent search and insertion in 2-3 trees. Technical Report 78-05-01, Dept. of Computer Science, University of Washington (May, 1978).

Guibas, L. J. and Sedgewick, R.: A dichromatic framework for balanced trees. Proc. 19th Annual Symposium of Foundation of Computer Science (Oct., 1978), pp. 8-21.

Knuth, D. E.: The Art of Computer Programming, Vol. 3: Sorting and Searching. Addison-Wesley, Reading, Mass. (1973).

Kwong, Y. S. and Wood, D.: T-trees: a variant of B-trees. Technical Report 78-CS-18, Unit for Computer Science, McMaster University (Nov., 1978).

Kwong, Y. S. and Wood, D.: Concurrency in B-trees, S-trees and T-trees. Technical Report 79-CS-17, Unit for Computer Science, McMaster University (August, 1979).

Lamport, L.: Concurrent reading and writing. Comm. ACM 22, 11 (1977), pp. 806-811.

Lehman, P. and Yao, S. B.: Efficient locking for concurrent operations on B-trees. In preparation.

Miller, R. E. and Snyder, L.: Multiple access to B-trees. Proc. Conference on Information Sciences and Systems (March, 1978).

Parr, J. R.: An access method for concurrently sharing a B-tree based indexed sequential file. Technical Report 36, Dept. of Computer Science, University of Western Ontario (April, 1977).

Samadi, B.: B-trees in a system with multiple users. Information Processing Letters 5, 4 (1976), pp. 107-112.