Robust Search Methods for B–Trees

Kikuo Fujimura
Pankaj Jalote

Computer Science Department
University of Maryland
College Park MD 20742 USA

ABSTRACT. The B–tree and its variations are commonly used to support multi–level indexing for organizing storage structures. If an index in a B–tree gets corrupted, the data retrieval becomes unreliable, as the corrupted index may force a search for a record to take a wrong branch in the tree. In this paper, we propose search methods for B–trees that correctly perform the search despite the presence of corrupted indices. If in a search, the desired record with a given key is not found, our robust search methods determine if this is caused by a corrupted index. The corrupted index is then detected, and corrected such that the value of the index is in a right range. We first present a method to handle a single error in the tree, and then generalize the method to cope with multiple errors. Unlike the previous attempts for robust data structures, our methods do not require redundancy to be added to the data structure, and make use of the constraints by which the indices are organized.

1. INTRODUCTION

Different techniques have evolved over the years for supporting software fault tolerance [1]. These include recovery block based schemes [10], n–version programming [2], exception handling [5, 9], and robust data structures [13, 14]. The first three categories focus on handling software design faults, while the robust data structures are primarily concerned with preserving the integrity of the data structures. In this paper we consider the problem of supporting reliable software with potentially unreliable data structures. In particular, we focus on search methods for B–trees.

The B–tree and its variations have been widely used in physical implementation in database systems [4, 6]. If an index in a tree gets corrupted, the data retrieval becomes substantially unreliable. If a search is diverted to a wrong branch by the corrupt index, the record with a given key may not be found, even when it actually exists in the database. In this paper, we discuss reliable search methods which can function correctly even in the presence of corrupt indices.

Much of the previous work on robust data structures has concentrated on detecting and correcting structural errors like corrupted pointers [3, 11, 12, 14, 15, 16]. The general approach is to add redundancy to the data structures to make them robust against structural corruption. Such redundancy may include an extra identifier field in a node and extra pointers (like threads in a tree or backward links). The redundancy is then used to detect and correct structural corruptions.

The methods we propose aims to handle corruptions in the semantic information, in particular the value of the indices in the B–tree, and assume that the structure is correct. Hence, the methods we propose are complementary to the approaches that handle structural corruptions. Similar methods for detecting errors were proposed for sorted arrays and lists in [8].

We consider a variation of the B–tree which is called B^+–tree. In a B^+–tree all the records are stored in the leaves, and the intermediate nodes are used solely as a hierarchical index to the records at the bottom. B^+–trees have been used as the access method in VSAM [7, 17], and have some advantages over B–trees. An index in a B^+–tree is considered to be correct as long as its value stays between the maximum key value in its left subtree and the minimum key value in its right subtree. If the index is in this range, a search on the tree will not be misdirected. An index error occurs when a value is found outside this range.

In this paper, we present techniques to detect and correct errors in indices. We assume that the record keys of the leaves are error–free, and that structural elements, like pointers, are error–free. The proposed search methods reach the record with a given key, if it is present, even if some of the indices on the search path are corrupt. Our *robust search method* never reports "record not found" when the record in fact exists in a leaf of the tree. A key feature of the proposed methods is that they do not require adding any redundancy to the data structure. Instead, we make use of the "built–in" redundancy that exists due to the relationship between indices and keys in the records.

The remainder of the paper is organized as follows.

This work was supported in parts by NSF grants DCI–8610337 and DCR–8605557. K. Fujimura is also a member of the Center for Automation Research, University of Maryland. P. Jalote has a joint appointment with the Institute of Advanced Computer Studies, University of Maryland.

0731-3071/88/0800/0194$01.00 © 1988 IEEE

In Section 2, we illustrate our method for error detection with an example. In Section 3, we present our robust search method in the presence of a single error. An extension to this search method to handle multiple errors is given in Section 4.

2. ERROR DETECTION STRATEGY

In a B^+-tree an internal node (i.e. not a leaf) can contain many indices, up to a pre-specified maximum. A node with k indices has exactly $k+1$ children. Let $I_1, I_2, \cdots, I_k$ be the k indices in a node, which are kept in the node in ascending order. Let $P_0, P_1, \cdots, P_k$ be the pointers to the $k+1$ children of the node. For an index I_i we will use the term L-subtree to denote the tree pointed by P_{i-1}, and R-subtree to denote the tree pointed by P_i. The following B^+-tree property is of importance to us in the subsequent discussions.

> **BTP:** For an index I_i in a node, I_i is greater than any index or key in its L-subtree, and is smaller than or equal to any index or key in its R-subtree.

We consider an index I_i to be *corrupt* if I_i does not satisfy the BTP. That is, I_i is either equal to or smaller than the largest key in the L-subtree, or is larger than the smallest key in the R-subtree. An example of a B^+-tree is given in Figure 1. In this example the index 35 in the node $n1$ is corrupt, since there are keys in the R-subtree of this index that are smaller than 35. Suppose that we are to search for key 31 in this tree.

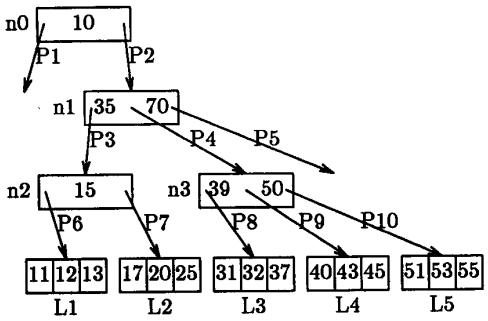

Figure 1: An example of a B^+-tree with a corrupt index

During the search, the tree is traversed in a top-down fashion from the root ($n0$). Indices 35 and 15 are referenced successively (pointers P2, P3, and P7 are traversed), which leads us to leaf L2, where key 31 is not found. However, at this point, we cannot conclude that key 31 does not exist, since there may be a corrupt index on the search path, $n0$–$n1$–$n2$, which may have misdirected the search. Any one of the indices, 10, 35, or 15, can be corrupt. For the case in which at most one corrupt index is present, our strategy is to narrow down the possibility to at most a single index by using the B-tree property. Let us consider each of these indices.

Could index 15 in $n2$ have misdirected the search? This is impossible, observing that the largest key in L2, the R-subtree of 15, is smaller than 31. Since all keys in leaves are assumed to be correct, the direction given by the index in $n2$ must be correct.

Could 10 be the corrupt index? The answer is no. If it is corrupt then the search should have actually gone to the L-subtree, instead of the R-subtree. This means that in the R-subtee the search should choose the leftmost index in each node. This is not the case (R-subtree of $n2$ is chosen). Hence, 10 cannot be corrupt.

This leaves us with only one index – 35. In order to resolve whether or not this index 35 is corrupt, we now try the other branch at $n1$ (pointer P4). Subsequently, key 31 is found in record L3, which makes 35 the corrupt index. If the key is not found there too, then we can conclude correctly that key 31 does not exist.

After the corrupt index is located, the index is corrected by replacing the corrupt value by a value that preserves the B-tree property. In this example, 35 in $n1$ can be any value satisfying $25 < x \leq 31$; this is the range of correct indices for this subtree.

Our strategy is to maintain a set of indices encountered during the search that may have caused a wrong branching decision. As the search proceeds, the entries from this set are deleted, if they are no longer considered suspicious, and new entries are added. The basic issue for this approach is how to maintain this set such that minimum entries remain in the set when the search terminates. We first consider the case in which only one index can be corrupt, and then generalize our scheme.

3. ROBUST SEARCH WITH SINGLE ERROR

The following is a regular top-down search function, where I and P represent index and pointer in a node, respectively. SEARCH_0 assumes that the tree contains no error.

```
function SEARCH_0(n: node; K: key;):boolean
/*return TRUE if K is found, FALSE if not*/
begin
(1) while n not a leaf do begin
(2)    find i such that n.I_i ≤ K < n.I_{i+1}
(3)    n := n.P_i
   end
(4) if K found in leaf then return TRUE
      else return FALSE
end
```

3.1. Maintaining the Suspicious–Set

We call an index *suspicious* if it may have caused an incorrect branching decision during a search. Note that this does not imply that a corrupt index on the search path must necessarily be suspicious. Indices which correctly direct a search are *not* suspicious, whether or not they are corrupt. Also note that selecting i in line 2 is based on I_i and I_{i+1}, and if either of them is corrupt, the search may be misdirected. This means that both should be initially considered suspicious. We say that the index I_i directs the search to R, while I_{i+1} directs the search to L. We assume that an error in an index maintains the ordering property of the indices in a node but violates the BTP. If the ordering property in a node is violated, the error can easily be detected, for the single error situation, by checking the index I_{i+2}. Once detected, the error can be corrected using the error correction scheme described later.

In order to maintain suspicious indices which direct a search to R and L, we use two queues of indices, SS_L and SS_R, respectively (together termed the *suspicious–set*). An element of the queues is a triplet of the form (*Node,Index,Dir*), where *Node* is the node in which a suspicious index is found, *Index* is the suspicious index (we will use both the value of the index and its position in the node for *Index*), and *Dir* is the branching direction given at the index, which is either R (right) or L (left). Both queues are initially empty. As a search descends the tree, indices are added to the queues in a FIFO manner. (Operation *add* appends an element to the end of the queue, *delete* removes an element from the front, if the queue is not empty). These indices stay in the queue until they are judged to be no longer suspicious. We use the procedure UPDATE_SS to maintain the two queues during the search. This procedure is executed at each iteration of the search loop and once when the search terminates at a leaf.

```
procedure UPDATE_SS(n: node; i:index)
/* SS_L and SS_R are queues of size one. */
begin
(1) if I_i not the smallest index then begin
(2)       delete (SS_R)
(3)       if n is not a leaf then add (SS_R, (n, i, R))
    end
(4) if I_i not the largest index then begin
(5)       delete (SS_L)
(6)       if n is not a leaf then add (SS_L, (n, i+1, L))
    end
end
```

If a previous index which directs the search to R was corrupt (that is, the search should have actually gone to L), then during the rest of the search, indices chosen in the nodes must be the smallest. This means that if an index encountered during the search is not the smallest one, the direction R given by the previous index must be correct. Thus, the previous index can be removed from SS_R, as is done in Line 2 of UPDATE_SS. A similar property for a corrupt index that misdirects the search to L (instead of R) is used to delete entities in SS_L in Line 5. We have the following claims.

Claim 1. During a search, elements in SS_R and SS_L are the only possible indices that could have misdirected the search.

Proof: Suppose that a search for key K is at node m in which index b directs the search to R. Suppose that current SS_R contains an entry $s = (n, a, R)$.

If the direction of s is incorrect, then the correct value of a (denoted as a') must be greater than K.

$$K < a'$$

Since the search chose the right branch of index b, K must be equal to or greater than b.

$$b \leq K$$

Since index b is found in the right branch of node m,

$$a' \leq b$$

From these equations we get $K < a' \leq b \leq K$, which is a contradiction. Therefore, the direction given by a in node n is reliable, and the entry s can be removed from the set, as is done in line (2) of the procedure. In turn, if m is not a leaf, the current direction (m, b, R) is added to SS_R, since it is not yet proved to be reliable. The same argument applies for the direction L.

○

Claim 2. For an unsuccessful search, SS = (SS_R + SS_L) contains at most one index.

Proof: During each iteration of the search loop whenever an entry is added to a queue, one is also deleted. Hence, on termination of the search loop, at most one entry exists in each of the queues. On termination (i.e. n is a leaf) the entry from at least one of the queues is removed (and maybe both), and none is added. Hence the claim.

○

An empty SS means that there is no suspicious index on the search path. For an unsuccessful search, this happens when (1) the search ends with i in a leaf node such that i is not the largest or the smallest, or (2) the largest (smallest) index is chosen at every node in a search.

3.2. Error Detection

If the suspicious–set is empty after the initial search, then the result of the search is correct, whether or not the desired record is found. Otherwise, we are left with an unreliable "record not found" response. For a correct response, we need to examine the alternate branch of the suspicious index simply by redoing SEARCH_0 from that node. Due to our Claim 2, we need to check at most one subtree. If the key is found after the second search, the index in SS *must* be corrupt.

If the key is not found during this second search, then we can be sure that the key *in fact* does not exist in the tree. The index *may*, however, be corrupt and may have misdirected the original search. One way to determine whether or not the suspicious index in node n is corrupt is to look at the leaf whose parent is n in post–order traversal (or in–order). In other words, if (n, i, L) is in the suspicious set, we compare I_i with the leftmost key in the R subtree of I_i. If I_i is greater than the key, then the index I_i is corrupt and actually misdirected the search. The situation where (n, i, R) is in the suspicious set is handled similarly. Combining, we have the following function for error detection.

```
function ERROR_DETECT(S):boolean
/* returns TRUE when S is an error */

begin
 if S.Dir=L then
     return (SmallestKeyInRsubtree≤S.index)
 if S.Dir=R then
     return (LargestKeyInLsubtree>S.index)
end
```

3.3. Error Correction

If an error is detected, then the error correction phase takes over. The index that has been determined to be corrupt is replaced with an appropriate value. The correct value must be greater than the rightmost key of the left subtree (called *lval*), and equal to or smaller than the left most key of the right subtree (called *rval*). If SS_R contains an element (n, i, R) after an unsuccessful search, the search must have chosen the leftmost direction at all the nodes below n. This means that the search has reached the leftmost node in the right branch of index i of node n, where *rval* for i is found. Similarly, if SS_L contains an element (n, i, L), *lval* for j must be known after an unsuccessful search. Hence, either *rval* or *lval* is known after the first search. These can be used to correct the corrupt index as shown below.

```
procedure ERROR_CORRECT(S)
begin
  if lval is known then
      S.Node[S.Ind]:=lval+1
  else
      S.Node[S.Ind]:=rval
end
```

It should be pointed out that a corrupt index need not necessarily misdirect a search. The goal for robust search is to detect (and correct) those errors that misdirect a search. The scheme for detection and correction that we have described above always detects and corrects an error in an index if a search is affected by the error. As a side benefit, sometimes an error that does not affect a search (i.e. the search would have proceeded the same way even if the index was not corrupt), is also corrected. Combining the functions for detecting and correcting errors, we have the complete *robust search* scheme for single errors.

```
function ROBUST_SEARCH_1(n, K): boolean
/* Returns TRUE when key K is found */
/* Tree can contain at most one error */
  begin
(1)  while n not leaf do begin
(2)    find i such that n.I_i≤K<n.I_{i+1}
(3)    UPDATE_SS(n, i)
(4)    n:=n.P_i
     end

(5)  find smallest j such that n.key_j≤K≤n.key_{j+1}
(6)  UPDATE_SS (n,j)
(7)  SS := SS_R + SS_L

(8)  if K found in n then return(TRUE)
(9)  if SS = φ then return(FALSE)

(10) S := the entry in SS

(11) found := SEARCH_0(alternate_branch(S), K)
(12) if found then ERROR_CORRECT(S)
(13) elseif ERROR_DETECT(S)
           then ERROR_CORRECT(S)
(14) return(found)
  end
```

The first four statements are the regular search, except for UPDATE_SS, which is executed in every loop iteration. UPDATE_SS is also executed in line 6, to handle the case when the node is a leaf. At line 7, SS will have at most one element (Claim 2). If the key is found (line 8) or if SS is empty (line 9), we are sure that there is no corrupt index that has misdirected the search, and the robust search ends. Otherwise, we focus on the only entry in SS. In line 11 a standard search is performed in the alternate_branch of this index, i.e. the subtree other than the one that was taken earlier. If a key is found in this search, we know that the index is corrupt, and an error correction is performed. Otherwise, we still check if the index is corrupt by using the ERROR_DETECT routine, and if an error is found it is corrected.

4. ROBUST SEARCH WITH MULTIPLE ERRORS

The idea in the previous section can be extended to handle multiple corruptions. In this section, we assume that a tree can contain at most m corrupt indices. For the sake of simplicity, we also assume that a node in a tree contains at most one corrupt index.

For updating the suspicious set we use the procedure UPDATE_SS_m. This procedure is similar to the procedure UPDATE_SS with the difference that now we delete an element from the queue only if the number of elements in the queue is equal to m. Furthermore, on termination of the search, if I_i is not the smallest (largest), the SS_R (SS_L) queue is totally cleared. With this we have the following claim.

Claim 3. For an unsuccessful search, SS_R + SS_L contains at most $min(m,h)$ indices, where h is the height of the tree.

Proof: Analogous to the proof of claims 1 and 2 and is given in the appendix.

Now we present our R-SEARCH_m procedure for multiple corruptions. Since there are many corrupt indices possible, we must perform the alternate search many times. The basic idea is to keep searching for a key until the key is found or SS becomes empty. For the multiple error case, the procedure ERROR_DETECT returns true if and only if the the index is found to be corrupt and the corruption is such that the search was misdirected. This is easily determined by comparing the search key K to the maximum or minimum of the R or L subtree of the index under suspicion.

```
function R_SEARCH_m(n; K; m):boolean
/* Returns TRUE if K exists in the tree. */
/* Tree may contain at most m corrupt indices. */

found: boolean   /* initially false */
SS, SS_R, SS_L: queue  /* initially empty */

begin
(1) while n not leaf do begin
(2)    find i such that n.I_i ≤ K < n.I_{i+1}
(3)    if m ≠ 0 then UPDATE_SS_m (n, i, m)
(4)    n := n.P_i
    end

(5) find i such that n.Key_i ≤ K < n.Key_{i+1}
(6) UPDATE_SS_m (n,i,m)
(7) SS := SS_R + SS_L

(8) if K found then return(TRUE)
(9) if SS = φ or m=0 then return(FALSE)

(10) while SS ≠ φ and not found do begin
(11)   S := first element of SS
(12)   delete(SS)
(13)   if ERROR_DETECT(S) then begin
(14)      ERROR_CORRECT(S + all elments in SS)
(15)      found := R_SEARCH_m
              (alternate_branch(S), K, m-1-|SS|)
       end
     end
(16) return(found)
end
```

After the initial search, SS contains at most $min(h,m)$ indices with the same Dir, where h is the height of the tree. We examine whether or not these have misdirected the search, starting from the one nearest to the root (Line 11). If the index is determined to have misdirected the search, all the other indices remaining in SS must also be corrupt. The reason for this is best explained by an example. Suppose the entries in SS are of $Dir=L$, and I_i is the suspicious index nearest to the root. If ERROR_DETECT for this node is true (i.e. the index I_i is corrupt and actually misdirected the search), it means that the key K is greater than the correct value of I_i. This implies that K must be greater than all the indices in the left branch of i, which implies that all the indices that have directed a search to L in the original search are also corrupt. Hence when an index is found to be corrupt, all the entries in SS must also corrupt. All these indices are corrected in line 13. The ERROR_CORRECT procedure is suitably modified to handle a set of nodes.

If an error of index I_i is detected, we examine the branch that has been missed in the first search by using R_SEARCH_m recursively (Line 14) with the new m being $m-1-|SS|$. If the direction given by i is determined to be correct, then we move to the next element of SS, the second nearest index from the root.

5. DISCUSSION AND CONCLUSION

In this paper we have presented robust search methods for B-trees that operate correctly even if some of the indices in the tree are corrupted. Moreover, the corrupt indices can also be corrected such that future searches are not affected.

The proposed schemes require no modifications to the data structure, and have little storage overhead. Extra storage is needed basically to maintain the variables like the queues, which are all bounded. The cost for robust search is primarily in the increased processing time. In most situations, we would like to have minimal overhead due to the activities for supporting fault tolerance when there is no fault in the system. High recovery overhead when the fault is detected may be acceptable.

In the proposed schemes some processing time overhead is due to the maintenance of the suspicious set. However, the major overhead comes due to the possibility that if a search fails, a subtree may need to be searched (for the case where at most one index is corrupt). However, in case there are no corruptions in the tree, this extra search time overhead is needed only if a failed search ends at an index in a leaf node which is either the smallest or the largest (otherwise the SS is empty). If the size of the leaves is large, we can expect that the probability of an unsuccessful search ending at the smallest or the largest entry in the leaf is small, and consequently the average overhead for search in error-free trees will be small.

For error detection, in case the key is not found in the tree (even in the alternate search), the major overhead results from finding the maximum or minimum value in order to detect if the index is corrupt. This overhead can be reduced by checking for the BTP property when the alternate search is proceeding. In most cases, the indices in the nodes during the search will themselves reveal that the index is corrupt, and a final traversal of the subtree to determine the maximum or the minimum can be avoided.

This work is complementary to the robust data structure work, particularly the robust B–tree implementation [3, 16], where structural corruption of the tree can be handled. For highly reliable systems, both of these techniques should be used. We are currently studying the integration of the two techniques.

REFERENCES

[1] T. Anderson and P. A. Lee, *Fault Tolerance, Principles and Practice*, Englewood Cliffs, NJ: Prentice Hall International, 1981.

[2] A. Avizienis, "The N–version approach to fault tolerance", *IEEE Tran. on Software Engg.*, Vol. SE–11, No. 12, Dec. 1985, pp. 1491–1501.

[3] J. P. Black, D. J. Taylor and D. E. Morgan, "A robust B–tree implementation", *Int. Conf. on Software Engineering*, March, 1981, pp. 63–70.

[4] D. Comer, "The ubiquitous B–tree", *ACM Computing Surveys*, Vol. 11, No. 2, June 1979, pp. 121–137.

[5] F. Cristian, "Exception handling and software fault tolerance", *IEEE Tran. on Computers*, Vol. C–31, no. 6, June 1982, pp. 531–540.

[6] C. J. Date, *An Introduction to Database Systems* Vol.1, Addison–Wesley, 1986.

[7] D. Keehn and J. Lacy, "VSAM data set design parameters", *IBM System Journal*, Vol. 3, 1974, pp. 186–212.

[8] K. Kuspert, "Ein effizientes verfahren zur fehlererkennung in sortierten feldern and listen", Informatik–Fachberichte, Vol 83, Springer–Verlag, pp. 51–62.

[9] Liskov, B. H. and A. Snyder, "Exception handling in CLU", *IEEE Tran. on Software Engg.*, Vol. SE–5, No. 6, Nov. 1979, pp. 203–210.

[10] B. Randell, "System structure for software fault tolerance", *IEEE Tran. on Software Engineering*, June 1975, Vol. SE–1, No. 2, pp. 220–232.

[11] M. C. Sampaio and J. P. Sauve, "Robust Trees", *15th Int. Symposium on Fault–Tolerant Computing*, June 1985, pp. 42–47.

[12] S. C. Seth and R. Muralidhar, "Analysis and design of robust data structures", *15th Int. Symp. on Fault–Tolerant Computing*, June 1985, pp. 14–19.

[13] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: Improving software fault tolerance", *IEEE Tran. on Software Engineering*, Vol. SE–6, No. 6, November 1980, 585–594.

[14] D. J. Taylor, D. E. Morgan and J. P. Black, "Redundancy in data structures: Some theoretical results", *IEEE Trans. on Software Engineering*, Vol. SE–6, No. 6, November 1980, 595–602.

[15] D. J. Taylor, and J. P. Black, "Guidelines for storage structure error correction", *15th Int. Symp. on Fault–Tolerant Computing*, June 1985, pp. 20–22.

[16] D. J. Taylor and J. P. Black, "A locally correctable B–tree implementation", *The Computer Journal*, Vol. 29, no. 3, June 1986.

[17] R. E. Wagner, "Indexing Design Considerations", *IBM System Journal*, Vol. 12, No. 4, 1973, pp. 351–367.

Appendix

Proof of Claim 3: Suppose that a search for K is at node n_{m+1} where index a_{m+1} directs the search to R. Suppose that SS_R (a queue of size m) contains m elements, (n_1,a_1,R), (n_2,a_2,R), $\cdots$,(n_m,a_m,R). If a_1 misleads the search, then the correct value of a_1 (denoted as a_1') must be greater than K.

$$K < a_1'$$

Since K chose the right branch at nodes n_2, $\cdots$, n_{m+1},

$$a_i \leq K, \text{ for } i=2,3,\cdots,m+1$$

must hold. Since the tree can not contain $m+1$ corruptions, one of $a_2,a_3,\ldots,a_{m+1}$, must be correct. Therefore, one of the following must hold from the B–tree property.

$$a_1' \leq a_j, \text{ for } j=2,3,\cdots,m+1$$

From these, we have $K < a_1' \leq K$, which is a contradiction. Therefore, the direction given by a_1 is correct, thus a_1 can be removed from SS_R. The same argument applies to direction L. At a leaf, at least one (and maybe both) of SS_R and SS_L is cleared. Since a search visits at most h nodes, the size of the suspicious–set after a search is at most $min(h,m)$.