A Concurrent B^{link}-Tree Algorithm Using a Cooperative Locking Protocol*

Sung-Chae Lim[1], Joonseon Ahn[2], and Myoung Ho Kim[3]

[1] WST Lab., Korea Wisenut Inc., Yangjae-dong, Seocho-gu, Seoul, 137-130, Korea, sclim@dbserver.kaist.ac.kr

[2] Hankuk Aviation University, Hwajundong, Koyang, Kyounggido 412-791, Korea jsahn@mail.hangkong.ac.kr

[3] Korea Advanced Institute of Science and Technology 373-1, Kusung-dong, Yusung-gu, Taejon, 305-701, Korea, mhkim@dbserver.kaist.ac.kr

Abstract. We present a new concurrent B^{link}-tree algorithm that provides a concurrent tree restructuring mechanism for handling underflow nodes as well as overflow nodes. Our algorithm does not require any lock for downward searching and preserves bottom-up tree restructuring without deadlock. To this end, we develop a new locking mechanism for inserters and deleters and a node update rule that preserves the semantical tree consistency during tree restructuring. Our analytical experiment shows that the overhead of additional disk I/O is acceptable.

1 Introduction

While various index structures have been proposed for high performance transaction processing, B-tree indexing has been typically used by many commercial database systems [1]. Therefore, many concurrent B-tree algorithms have been proposed to deal with concurrent accesses to B-trees efficiently [2,3,4,5,6,7,8,9].

Among the concurrent B-tree algorithms, it has been indicated that the B^{link}-tree [3,5,6] which provides non-blocked downward searching and bottom-up node splitting is among the best choices considering transaction throughput [13,14]. However, they have no concurrent mechanism for restructuring underflow nodes. When underflow nodes cannot be handled, B-trees become sparse, which leads to the degradation of performance [10,11]. In [6], underflow nodes are restructured by a background mode process that retrieves all the tree nodes. This method suffers from heavy retrieval cost and tree compaction time.

In this paper, we propose a concurrent B^{link}-tree algorithm that can handle underflow nodes concurrently. We first present a new locking protocol which provides a deadlock-free locking sequence to updaters competing for lock grants on the same nodes. Also, we provide a node update rule for key transfer and

* This research was supported by IRC (Internet Information Retrieval Research Center) in Hankuk Aviation University. IRC is a Kyounggi-Province Regional Research Center designed by Korea Science and Engineering Foundation and Ministry of Science & Technology.

A. James, B. Lings, M. Younas (Eds.): BNCOD 2003, LNCS 2712, pp. 253–260, 2003.

node merging to guarantee concurrent search operations to access nodes that are under restructuring and maintain the consistency of B-trees [7,9].

The rest of this paper is organized as follows. Section 2 revisits the problem of B^{link}-tree concurrency control (CC). In Section 3, we present a new algorithm for the concurrent access to B^{link}-trees. Section 4 describes the node update rule for deleters. Section 5 proves deadlock-freeness of our algorithm and Section 6 addresses performance issues. Finally, Section 7 gives a conclusion.

2 Preliminaries

2.1 Backgrounds on B^{link}-Tree Concurrency Control

B^{link}-tree is a modification of the B-tree such that sibling nodes at each level are linked from the left to the right [3,6]. An internal node with n index entries has the format $< p_1, k_1, \ldots, k_{n-1}, p_n, k_{max}, siblinglink >$. $siblinglink$ points to the right sibling. p_i points to the subtree having keys k such that $k_{i-1} < k \leq k_i$, where $k_n = k_{max}$. In leaf nodes, p_i points to the record with key value k_i.

If an inserter makes a node overflow, the inserter moves the right-half portion of the node to a new node, updates the sibling links and k_{max} of the two nodes and inserts a new index entry into the parent node [5]. If a process arrives at a node searching for a target key greater than k_{max} of the node, which means the node has been split, the reader moves to the next node by following the sibling link. In this way, search operations can execute concurrently with node splits not requesting any locks. This non-blocked downward search improves the concurrency of B-trees and reduces the CPU cost for locking operations [13,14].

2.2 The Basic Idea of the Proposed B^{link}-Tree CC Algorithm

In B^{link}-tree, multiple updaters, each of which is an inserter or a deleter, may get to their target leaf nodes following the same path. Handling underflow nodes needs to exclusively access their parent and sibling nodes to transfer index entries or merge half-full nodes. Also, handling overflow nodes needs to exclusively access their parent nodes. Therefore, multiple updaters restructuring the same nodes can cause deadlock.

In our locking protocol, we use two kinds of locks, the X mode and IX mode locks. The X lock is not compatible with any lock, while the IX lock is compatible with itself[12]. At the lock grant time of an IX lock, the function for lock request returns a value, Shared or Alone. If there exist other IX-lock holders on the node, Shared is returned. Otherwise, Alone is returned.

A lock holder can change the kinds of its locks. If a process P holding an X lock on a node N requests a conversion to an IX lock, IX lock is immediately granted to P. Also, IX lock is granted to other processes that have been blocked requesting IX locks for N if such processes exist. If P holding an IX lock on a node N requests a conversion to an X lock, P has the highest priority to hold an X lock on the node. If there were other IX-lock holders on the node, P is inserted to the waiting queue. Otherwise, P is granted the X lock immediately.

```
procedure search_for_leaf(kvalue, Leaf) /* read the leaf with a key value of kvalue */
begin
  Ptr ← the pointer to the root node; Height ← the height of the tree;
  while ( Height > 1) do /* pass down internal nodes */
    Read the node pointed to by Ptr into a local memory node, N;
    while ( kvalue > N.largest_key ) do /* The node was split */
      Ptr ← N.siblinglink;
      Read the node pointed to by Ptr into the memory area N again;
    endwhile
    Ptr ← the pointer to the next child node; /* search down to the child level */
    Decrease variable Height by 1;
  endwhile
  get_node(Ptr, kvalue, Leaf ,Lockmode); /* lock and read the leaf node */
end.
procedure get_node(Ptr, kv, Node, Lockmode)
begin
  State ← lock(Ptr, Lockmode); /* lock node Ptr with a given lock mode, Lockmode */
  label1: Read the node pointed to by Ptr into the local memory node, Node;
  if ( kv > Node.largest_key ) then
    unlock(Ptr); Ptr ← Node.siblinglink; State ← lock(Ptr, Lockmode);
    goto label1;
  endif
end.
```

Fig. 1. The algorithm for procedures search_for_leaf() and get_node()

Our locking protocol is based on lock stratification and lock cooperation. The former defines a rule which prevents deadlocks associated with nodes at multiple levels. The rule forces an updater which locks both a parent and child nodes to have an X lock for the parent and IX locks for the child. When a deleter observes that a sibling node to be restructured already has been locked by another updater, it follows lock cooperation, which prevents deadlocks associated with nodes at the same level and keeps trees consistent not losing any update.

3 The Proposed B^{link}-Tree CC Algorithm

3.1 The Key Search Algorithm

Fig. 1 shows procedures for key search. In our algorithm, all processes use the procedure search_for_leaf() for their downward searching. Arriving at a leaf node, they calls get_node() to lock and read the node having a given key value. If an IX lock is requested, lock() returns Shared or Alone at the lock grant time. Otherwise, its return value has no meaning. In the procedures search_for_leaf() and get_node(), examining the largest key and chasing along sibling links are necessary because the node can be split from overflow right before the arrival.

3.2 Algorithm for Inserting a New Index Entry

The insertion algorithm performed by an inserter is as follows:

(1) Search the target leaf node and lock it in X mode; then, insert an index.
(2) Unless the node overflows, write the node and exit after releasing the lock. Otherwise, go to the next step.
(3) Create a new node and lock the new node in IX mode.
(4) Perform a half-splitting by using the newly created node, and then convert the X lock on the overflow node to IX mode.
(5) X Lock and read the parent node by using get_node(). Then, release IX locks on the half-splitting nodes.
(6) Insert an index entry pointing to the new node into the parent node. If the parent node does not overflow, write it and exit after releasing all the locks. Otherwise, go to step (3) for key insertion into the parent node.

In step (4), the X lock is converted to an IX lock to observe the lock stratification rule. Unlike [5], we retain locks on the half-splitting nodes until an X lock is granted on the parent node. Otherwise, a deleter may delete one of the half-splitting nodes for node merging while the inserter is blocked.

3.3 Algorithm for Deleting an Index Entry

The deletion of an index entry is performed as follows.

(1) Search the target leaf node and lock it in X mode; then, delete the index.
(2) Unless the node underflows, write the node and exit after releasing the lock. Otherwise, go to the next step.
(3) Convert the X lock on the underflow node to IX mode. Then, X lock and read the parent node by using get_node().
(4) Choose a sibling for key transfer (or node merging) and IX lock the node.
(5) If the lock request on the sibling returns Alone, update nodes according to the steps described in Section 4. Otherwise, i.e., if the return value is Shared, follow the lock cooperation procedure given in the next subsection.

3.4 Lock Cooperation

If IX locks are shared between a deleter and an updater, we call such case a cooperation demanding situation (CDS). Only deleters can detect CDS when they receive Shared from the IX lock request for a sibling node. To enable deleters to deal with CDS, we use a *state* field in each node. When a deleter makes a node N underflow and subsequently detects a CDS on the left(or right) sibling of N, it sets the *state* field in N to LS(or RS). Otherwise, the *state* field is NULL.

Suppose a deleter Pd making node N underflow detects a CDS on the left sibling Ns on which an updater P1 holds an IX lock. Then Pd checks the *state* field in Ns. We describe the actions by Pd and P1 based on the two categories.

(a) If Ns has value RS, it means that P1 is a deleter which already detected a CDS on Pd and thus has a completely overlapped scope with Pd. In this case, Pd terminate after releasing all its locks and P1 completes tree restructuring.
(b) Otherwise, it can be one of three cases: (i) P1 is an inserter, (ii) P1 is a deleter that will detect a CDS on N, or (iii) P1 is a deleter that does not use N for tree restructuring. For all cases, Pd releases the X lock on the parent node after setting *state* of N with LS and suspends until P1 releases its lock on Ns by converting the IX lock on Ns to X mode. Then, Pd will resume tree restructuring after P1 performs its actions described below. In cases of (i) and (iii) P1 completes its tree restructuring after acquiring the X lock on the parent node. In (ii), P1 will later detect the completely overlapped situation with Pd, and hence will leave the tree as Pd does in (a).

In case of a CDS on a right sibling, the same rules can be applied analogously. The followings are steps for lock cooperation by Pd that detects a CDS on Ns.

(1) If Ns.*siblinglink* is not N, which means that Ns has been split, place a new IX lock on the node pointed to by Ns.*siblinglink* and release the IX lock on Ns; the newly locked node is regarded as Ns from now on.
(2) If the *state* field in Ns indicates a completely overlapped situation(i.e., the *state* field is LS(RS) and Ns is the right (left) sibling), then write N into the disk and exit after releasing all the locks; otherwise, go to the next step.
(3) Set N.*state* to LS or RS appropriately, and then write N into the disk.
(4) Release the X lock on the parent node and then request lock conversion on Ns from IX mode to X mode (self blocking).
(5) Convert the X lock on Ns to IX mode, then lock and read the parent Np of N. At this point, Ns may not be a correct sibling. For instance, an inserter may split the parent and make N and Ns have different parents.
(6) Check N and Ns are adjacent in Np. If Ns is a correct sibling, read Ns and restructure the three nodes N, Ns and Np following the update rule described in Section 4. Otherwise, Pd releases the IX lock on Ns and goes to step (4) of the deletion algorithm in Section 3.3. Note that lock cooperation is a part of step (5) of the deletion algorithm in Section 3.3.

4 Restructuring an Underflow Node

Suppose a deleter P has locked and read nodes N, Ns and Np using the procedure in Section 3.3. Here, N is an underflow node, Ns is a sibling of N and Np is their parent. Currently, P is the unique process that can update these nodes.

4.1 Transferring the Index Entries

If Ns has sufficient index entries, P moves some index entries of Ns to N. This is straightforward when Ns is the left sibling. That is, some rightmost index entries in Ns are inserted into N and these index entries are deleted from Ns, and then Np is properly updated. We should be careful when Ns is the right sibling and

leftmost index entries in Ns are deleted because such deletion may spoil other search operations which use the pointer to Ns. Therefore, we create a new node which stores the remaining entries of Ns and replace Ns with the new node.

(1) Create a new node Nnew and write the right portion of entries in Ns into Nnew. Nnew contains those entries of Ns which are not transferred to N.
(2) Write the left entry of Ns into N and update N.*siblinglink* to point to Nnew.
(3) Update the maximum key value for N in Np so that it reflects the index transfer to N and replace the pointer to Ns with the pointer to Nnew.
(4) Update the first pointer of Ns to point to N and mark Ns invalid so that any process arriving at this node can follow the pointer to N.
(5) Release all the locks and exit.

The invalidated node needs to be kept temporarily for processes that have the pointer to Ns by reading the old value of its parent node.

4.2 Merging the Half-Full Nodes

Unless Ns has sufficient index entries, node merging is performed. In node merging, we always transfer entries in a right sibling Nr into a left sibling Nl.

(1) All the index entries of Nr are inserted into Nl and the sibling link of Nl are updated with that of Nr. And we set the *state* field of Nl with NULL.
(2) The pointer to Nr and the old maximum value of Nl in Np are deleted. The previous maximum value of Nr becomes that of Nl.
(3) Nr becomes invalidated as in the case of the leftward key transfer.
(4) If Np underflows, P performs (3) of the algorithm in Section 3.3 after releasing locks on Nl and Nr. Otherwise, P releases all its locks and exits.

5 Deadlock-Freeness of the Proposed Locking Protocol

To prove deadlock freeness of our protocol, we use a lock-wait-for graph described below. When a certain updater P requests an IX or X lock for node N and is blocked due to lock conflict, we draw arcs heading for N from every node for which P already holds any lock. We remove these arcs heading for N when the lock is granted to P. For the proof, we have only to show that any cycle cannot be formed in this lock-wait-for graph using the following lemmas.

Lemma 1. *Any cycle composed of nodes at more than one level cannot be formed in the lock-wait-for graph.*

Lemma 2. *Any cycle composed of nodes at only one level cannot be formed in the lock-wait-for graph.*

The first lemma can be easily proved from the lock stratification rule. We can prove second lemma as follows. In our protocol, a process can hold only an IX lock on a node N when it requests a lock for a sibling of N. Therefore,

because IX locks are compatible with IX locks, a cycle composed of nodes at the same level can be constructed from X lock requests only. Before a lock holder of N requests an X lock for a left(right) sibling of N, it always confirms that the state field of the sibling is not RS(LS) and sets the state field of N with LS(RS), holding an X lock for the parent of the two nodes. From this, we can show that any cycle composed of nodes at one level cannot be formed in the lock-wait-for graph. The complete proof is omitted here because of limited space.

6 The Performance Overview

Because a deleter which detects a CDS has to set the *state* field in the underflow node and re-read the parent node and the sibling node after the self-blocking state, our lock cooperation has the overhead of additional one page (i.e. node) write and two page reads. Because the overhead of disk I/Os may degrade the performance, we investigate how often the CDS may occur.

Let the number of nodes in a tree be N_T and suppose each update operation can exist at a certain node with the probability of $1/N_T$. When two sibling nodes are updated by two updaters, we call the updaters are adjacent. The mean number of the pairs of updaters that are adjacent is denoted by N_{adj}.

Suppose N_U number of updaters come into a tree in sequence. We define I_k such that $I_k = 1$, if the k-th updater is adjacent to one of k-1 other updaters, otherwise, $I_k = 0$. Then, the expectation of $Y_k = \sum_{i=1}^{k} I_i$ is given as follows:

$$E(Y_k) = E(\sum_{i=2}^{k} I_i) < \sum_{i=2}^{k} E(I_i | I_{i-1} = 0, \ldots, I_2 = 0) = \sum_{i=2}^{k} E(I_i | Y_{i-1} = 0) \quad (1)$$

Since $N_{adj} = E(Y_{N_U})$ and $E(I_k | Y_{k-1} = 0) = \frac{2*(k-1)}{N_T}$, we have the following.

$$N_{adj} < \sum_{k=2}^{N_U} \frac{2*(k-1)}{N_T} = \frac{N_U*(N_U-1)}{N_T},\ N_U = 2, 3, 4, \ldots \quad (2)$$

Let each node have between $2D-1$ and D index entries and X_n be the probability that a given node has n index entries where $D \leq n \leq 2D-1$. Equation (3) shows X_n whose complete description can be found in [17].

$$X_n = \frac{1}{(n+1)}(H(2D) - H(D))^{-1},\ \text{where } H(D) = \sum_{i=1}^{D} 1/i \approx lnD \quad (3)$$

X_D is the probability that a deleter incurs an underflow and X_{2D-1} is the probability that an inserter incurs an overflow. Assuming that the frequencies of insertions and deletions are the same, the probability that an updater causes tree restructuring is $\frac{X_D + X_{2D-1}}{2}$. Then, the probability that a pair of adjacent updaters results in a CDS is less than $\frac{(X_D + X_{2D-1})^2}{4}$. Thus, the upper bound on the mean number of CDS occurrences N_{CDS} is driven as follows:

$$N_{CDS} < \frac{(X_D + X_{2D-1})^2}{4} * N_{adj} \approx (\frac{3}{4ln2})^2 * \frac{N_U*(N_U-1)}{N_T*(D+1)^2} \quad (4)$$

From this, we can see N_{CDS} is very small. For instance, N_{CDS} is $2.6 * 10^{-4}$ when there are 300 concurrent updaters in a B^{link}-tree with 5 M index entries.

7 Conclusion

We have presented a deadlock-free B^{link}-tree algorithm that can handle overflows and underflows concurrently while supporting non-blocked downward searches. To this end, we have developed a locking mechanism composed of lock stratification and lock cooperation and methods for restructuring underflow nodes.

Since lock cooperation requires additional disk accesses, we have analyzed the overhead based on a probability model. This shows that the overhead from the lock cooperation is acceptable.

References

1. D. Comer: The Ubiquitous B-tree. *ACM Computing Surveys*, **11(2)** (1979) 121–137
2. Bayer, R. and Schkolnick, M.: Concurrency of Operations on B-Trees. *Acta Informatica* **9** (1977) 1–21
3. Philip L. Lehman and S. Bing Yao: Efficient Locking for Concurrent Operations. *ACM Transactions on Database Systems* **6(4)** (1981) 650–670
4. Udi Manber and Richard E. Ladner: Concurrency Control In a Dynamic Search Structure. *ACM Transactions on Database Systems* **9(3)** (1984) 439–455
5. Yat-Sang Kwong and Derick Wood: A New Method for Concurrency in B-Trees. *IEEE Transactions on Software Engineering* **8(3)** (1982) 211–222
6. Yehoshua Sagiv: Concurrent Operations on B^*-Tree with Overtaking. *Journal of Computer and System Science* **33(2)** (1986) 275–296
7. Shasha, D. and Goodman, N.: Concurrent Search Structure Algorithms. *ACM Transactions on Database Systems* **13(1)** (1988) 53–90
8. C. Mohan: ARIES:IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. *ACM SIGMOD* **21** (1992) 371–380
9. Ragaa Ishak: Semantically Consistent Schedules for Efficient and Concurrent B-Tree Restructuring. *International Conference on Data Engineering* (1992) 184–191
10. Chendong Zou and Betty Salzberg: On-line Reorganization of Sparsely-populated B^+-trees. *ACM SIGMOD* **25** (1996) 115–124
11. Jan Jannink: Implementing Deletion in B^+-Trees. *ACM SIGMOD* **24** (1995) 33–38
12. Gray, J. and Reuter, A.: Transaction Processing: Concepts and Techniques. *Reading Mass* (1993) 449–490. Morgan Kaufmann Pub.
13. Johnson, T. and Shasha, D.: The Performance of Current B-Tree Algorithms. *ACM Transactions on Database Systems*, **18(1)** (1993) 51–101
14. V. Shrinivasan and Michael J. Carey: Performance of B^+ Tree Concurrency Control Algorithms. *VLDB Journal* **2** (1993) 361–406
15. Johnson, T. and Shasha, D.: The Performance of Current B-Tree Algorithms. *ACM Transactions on Database Systems* **18(1)** (1993) 51–101
16. Jayant R. Haritsa and S. Seshadri: Real-Time Index Concurrency Control. *SIGMOD Record* **25(1)** (1996) 13–17
17. Theodore Johnson and Dennis Shasha: B-trees with Inserts and Deletes: Why Free-at-Empty is Better than Merge-at-Half. *Journal of Computer and System Science* **40** (1993) 45–76