A Concurrency Control Scheme over T-tree in Main Memory Databases with Multiversion

*Ying Xia[1] Sook-Kyoung Cho[1] Young-Hwan Oh[1] June Kim[2] Hae-Young Bae[1]
[1] Dept. of Computer Science & Engineering, Inha University
[2] Dept. of Internet Service, Computer & Software Technology Lab., ETRI
g1992131@inhavision.inha.ac.kr

Abstract

In this paper, we present a concurrency control scheme over the index structure in main memory databases using multiversion mechanism, and implement it on T-tree. As a well-known idea for concurrency control, multiversion allows multiple transactions to read and write different versions of the same data item, each transaction sees a consistent set of versions for all the data items it accesses [1]. Logical versioning and physical versioning techniques are used to keep versions of data item and versions of index node respectively. The main features of this multiversion indexing approach are (1) update operations and rotations on T-tree can take place concurrently, (2) the number of locking and latching requirement is sharply reduced because read-only transactions do not obtain any locks or latches and update transactions obtain latches only when actually performing the update, (3) it reduces storage overhead for tracking version and reclaims storage in time, and (4) it provides complete isolation of read-only transactions from update transactions, so the read-only transactions can get response information without any block.

1 Introduction

Since disk access in main memory database system is only needed for logging, in concurrent environment, what dominates the cost of database access and effects system response time is latching and locking, because they might cause other transactions blocked [7]. Here, we use multiversion technique to reduce such cost and provide unblocked data access for read-only transactions and highly concurrent execution level for update transactions.

Logical versioning is applied to data item. New version is created only when data is updated, the overhead of versioning space is zero. As for which version should be read is decided by timestamp of the transaction. No locking is needed when read-only transactions access a data item.

Physical versioning is applied to index structure. When index structure is updated, a new version of all affected nodes are created and linked in the tree. It enables read-only transactions to traverse t he index structure without acquiring latches even when other transactions is updating on it. Update transactions also do not get latches during it traversals the index structure until the node that actually update is performed on.

Here, we assume that relations have only a single primary key index and zero or more secondary key indices, and the primary key values cannot be updated

The rest of the paper is organized as follows. Section 2 explains the multiversion technique implemented in our design. Section 3 gives the algorithms of concurrent operations on the T-tree with versioning. Conclusions are shown in Section 4.

2 Multiversion on T-tree

In system with multiversion, transactions are classified at startup time as read-only transactions (which includes only read items) and update transactions (which will update or write some items or which simply want access to the most current data). Each transaction is assigned a timestamp when it starts [1].

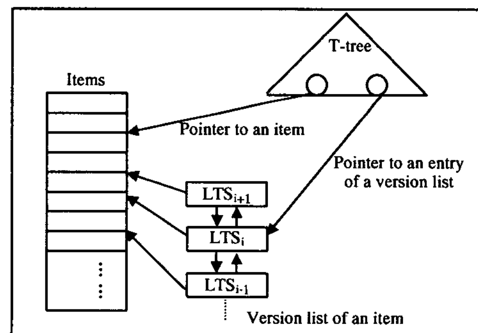
2.1 Logical versioning

Logical versioning is the versioning applied to data items. When a transaction updates a data item, a new version of that item is created. When transaction commits, a logical timestamp (LTS) is obtained. The transaction stamps each version it has created with this LTS.

A version list is defined in which all entries are doubly linked and ordered by LTS. Each entry of version list includes a LTS and a pointer pointing to the version data.

For each item, read-only transaction reads the latest version whose LTS is less than or equal to its timestamp.

A version of a data item will be deleted when it is no longer needed by any read-only transaction which has a timestamp equal to or larger than LTS of the version, but smaller than LTS of the next newer version of the item.



(Figure 1. pointers to data items)

In T-tree, data pointer in the tree node could be point to a data item if there is only one version or to a version entry if there are many versions. When an update transaction commits, pointers in tree node is updated to point to the new version entry. As Figure 1 shows, the transaction, which create the new version stamped as

*This Research is supported by Ministry of Information and Communication under work of university S/W research center and by ETRI under work of High Performance Real-time DBMS.

LTS_{i+1}, is still not commit.

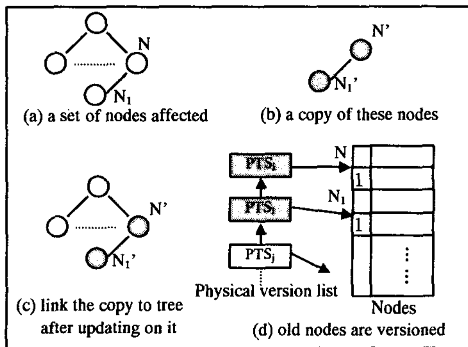
When traverses T-tree, if the pointer point to an item directly, then return this pointer. Else, version list of the item is traversed to determine an entry with the largest LTS less than or equal to the transaction timestamp, and return this version entry pointer.

2.2 Physical versioning

Physical versioning is the versioning applied to index structure, here refers to nodes in T-tree. A new version of a tree node is created only when a key is involved in insertion, deletion, or rotation. No new version is created if there is only a directly updating performed on a node.

Let N be the root of nodes set affected by an operation, physical versioning first copy these nodes and let N' as new root while change pointers to the nodes in original nodes set to point to new copies, then update are performed on the new copy, finally atomically change the pointer to N to point to N'. By doing so, what read-only transactions read is always the final update before it starts but not partial updates.

In T-tree, each node contains a version bit to indicate if the node is (physically) versioned. A physical timestamp (PTS) of the node is obtained after it is versioned. After linking a newer version of a node into the tree, mark the version bit of this nodes as 1, and append an entry containing a pointer to the node and its PTS into a physical version list. Figure 2 shows the four steps of physical versioning.



(Figure 2. four steps of physical versioning)

Once there is no timestamp of update transaction is smaller than the version's PTS, the older physical version will be deleted.

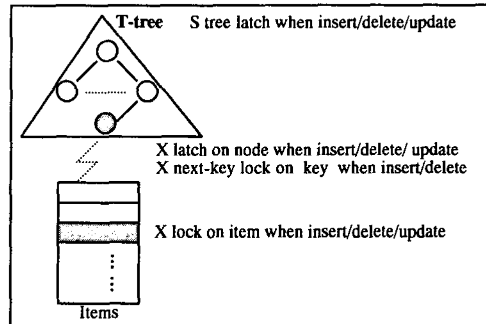
3 Operations on T-tree with multiversion

3.1 Search

As a basic operation, Search can be called by reader or invoked by insert, delete or update. We design a general Search algorithm which includes two important parameters: (1) *lock-mode*, a flag which indicates whether or what next-key lock (exclusive or shared) should be obtained on the key value returned by Search, and (2) *latch-mode*, a flag which if True indicates that the terminating node (bounding node or a leaf or a semi-leaf node that should contain the key) holds exclusive latch and the tree holds shared latch. A stack is used to record all nodes on access path. Search will get the proper version of data. It is relatively simple, here we ignore the details, refer to [4,5].

When Search is called on behalf of a read-only transaction, lock-mode is None, latch-mode is False. While, when Search is called on behalf of an update transaction, lock-mode is Shared, latch-mode is False. Whether called on behalf of updaters or readers, Search does not obtain latches on its way down and does not check if a node has been versioned until reaching the terminating node.

When Search is invoked in update transactions, as figure 2 depicts, after Search return, three levels of locking and latching is involved, they are latch on tree, latch/lock on node and lock on data item. If lock is needed, lock is obtained before latch is obtained to prevent deadlocks [6]. After obtaining appropriate locks and latches, validation is performed by checking if node has been versioned to determine if the key value is indeed the key value to be returned because the concurrent updaters may have inserted or deleted index entries while Search was obtaining locks or latches.



(Figure 3. different locks/latches for different update operations after Search return)

3.2 Insert

Insert a key first invokes Search with the key to insert, lock-mode is Exclusive, latch-mode is True. This ensures that an exclusive next-key locking in the index is obtained in order to avoid phantom problem and support repeatable read [6], and an exclusive latch on the node involved in insert is also held. Let N be the exclusively latched node after Search, three cases should be considered:

Case 1: N bounds key and has room. A copy of N, say N', is created and key is inserted into it. Then exclusive latch on N's parent and check if it has versioned. If so, release latch, re-traverse the tree from root to N to determine N's most current parent and latch it. Repeat this process until N's parent is found to be not versioned. Then the pointer to N is updated to point to N'. N then is versioned and all latches released.

Case 2: N does not bound key. In this case the node is a leaf or a semi-leaf. If there is room in N, then key is inserted as described in Case 1. Else, a new node which containing key is allocated, an exclusive latch on the node is obtained and the left or right child of N is set to point to the newly allocated node.

Case 3: N bounds key but no room. If the left child of N is null, then two node N1 and N2 are allocated and exclusively latched, N1 is a copy of N containing key but not containing the leftmost key in N and the left child of N1 is set to N2, while N2 simply contains the leftmost key in N. After exclusively latching N's parent, the pointer to N is updated to point to N1. N then is versioned and all latches released.

If the left child of N is not null, release the latch on N, and obtain exclusive tree latch. If N has been versioned or its left child has become null in between, release tree latch and Insert restart again by invoking Search from the highest but without been versioned node in stack. Otherwise, the following actions are taken:

Let N1 be the node that contains the largest key value in the left

subtree of N. If N1 has room, then a copy of N1 is made, the leftmost key value in N is inserted into the copy, change the pointer in N1's parent to point to the new version, and mark N1 as versioned. If N1 has no room, then a new node containing only the leftmost key value in N is allocated and N1's right child is set to point to the newly allocated node, and then new copy of N is made but delete the leftmost key value, inserted the key and change N's parent's pointer to N to point to the new copy, then node N is versioned.

The lock on the key is released at the end of the insertion once the key has been inserted.

3.3 Delete

Like Insert, Delete first invokes Search with the key to delete, lock-mode equal to Exclusive and latch-mode equal to True.

If N is bounding node and contains more than the minimum number of key values, then make a copy of N, delete key from the copy and update the pointer to N in its parent to point to the copy, finally mark N as versioned and release latch on N.

If deletion could cause the number of key values in N to be less than the minimum number of key values, then release the latch on N but obtain the tree latch in exclusive mode. Once the tree latch has been obtained, if N has been versioned in between, release tree latch and restart delete by invoking Search from the highest but without been versioned node in stack, else one of the following cases is considered:

Case 1: N is a leaf. If N contains a single key value, then exclusively latch its parent and set the pointer to N as null. Else, make a copy of N, and delete the key from the copy, update the pointer to N in its parent point to the new copy, mark N as versioned and release the tree latch.

Case 2: N has a left subtree. Let N' be the node in the left subtree containing the largest key value. Copy all nodes in between N and N'. If N' contains more than one key, then also make a copy of N' and delete the largest key value from the copy. If N' has only a single key value, exclusively latch N's parent, then set the pointer to N' in the copy of N's parent to null and release latch. In the copy of N, delete requested key and insert the largest key value of N'. Finally, update the pointer to N in N's parent to the new copy of N, mark node N, N' and all nodes between N and N' as versioned. The tree latch is released if N' contained more than one key because rotation is not required.

Case 3: N has no left child and has a right child. Actions are similar as Case 2. If necessary, rotations are performed until the tree is balanced, the tree latch is held until all are completed.

3.4 Update

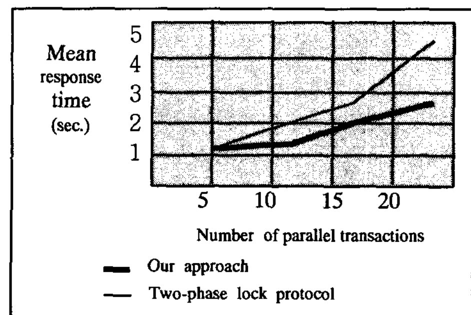
Update an index entry for a key value first invokes Search with the key value, lock-mode is None, latch-mode is True. Once the node containing the key value is exclusively latched by Search, the index entry is over-written by a new pointer, and then latch on the node is released. No physical versioning is needed.

3.5 Rotate

Rotate is done by traversing the tree upwards from the lowest unbalanced node. Details refer to [4]. If a exclusively tree latch is held, upwards traversal is performed and every time a parent node is accessed, check if it is versioned, if so, the tree is re-traversed. If the exclusively tree latch is not held, latch it before a parent node is examined to determine if it can be rotated, if it has been versioned, re-traversal the tree. While performing a rotation, physical versioning only requires that three nodes involved in the rotation is copied.

4 Performance and Conclusions

We did a simulation work to compare the performance of this approach with that of two-phase lock protocol over T-tree. We assume that each transaction is a set of 1000 accessing operations in which 30% are update and the others are read-only. As Figure 4 depicts, by using our approach, 1) the mean response time is shorter than that of the two-phase lock protocol at number of parallel transaction, which means we can serve more users at a means response level. And 2) the response time values are distributed in a narrow value range than that of two-phase lock protocol, which means transaction do not block each other. This scheme is especially suitable for applications which involving many read-only transactions.



(Figure 4. Comparison of the performance of our approach with that of two-phase lock protocol)

5 References

[1] P. M. Bober and M. J. Carey Indexing Alternatives for Multiversion Locking. In Proc. of the 4th International Conf. on Extending Database Technology. 1994

[2] P. Bohannon, D. Lieuwen, R. Rastogi, S. Seshadri, A. Silberschatz and S. Sudarshan. The architecture of the Dali main-memory storage manager. In Journal of Multi-media Tools and Applications, 4/2, 1997.

[3] V. Gottemukkala, T. J. Lehman. Locking and Latching in a Memory-Resident Database System. In Proc. of the 18th VLDB conf. 1992

[4] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. In Proc. of the Int'l Conf. on VLDB, 1986

[5] H. Lu, Y. Y. Ng, Z. Tian. T-Tree or B-Tree: Main Memory Database Index Structure Revisited. In Institute of Electrical and Electronics Engineers, Inc. 1998

[6] C. Mohan and F. Levine. Aries/IM: an efficient and high concurrency index management method using write-ahead logging. In Proc. of the ACM SIGMOD Conf. on VLDB, 1990

[7] R. Rastogi, S. Seshadri, P. Bohannon, D. Leinbaugh. Logical and Physical Versioning in Main Memory Databases. In Proc. of the 23rd VLDB conf. Athens, Greece, 1997