T-Tree or B-Tree: Main Memory Database Index Structure Revisited*

Hongjun Lu[1] *Yuet Yeung Ng* *Zengping Tian*[2]
Hong Kong University of Science & Technology Fudan University
Hong Kong, China Shanghai, China
{luhj,enoch}@cs.ust.hk zptian@cs.ust.hk

Abstract

While the B-tree (or the B^+-tree) is the most popular index structure in disk-based relational database systems, the T-tree has been widely accepted as a promising index structure for main memory databases where the entire database (or most of them) resides in the main memory. However, most work on the T-tree reported in the literature did not take concurrency control into consideration. In this paper, we report our study on the performance of the main memory database index structure that allows concurrent accesses of multiple users. Two concurrency control approaches over the T-tree are presented. The results of a simulation study indicate that the B-link tree, a variant of the widely used B-tree index will outperform the T-tree if concurrency control is enforced. This is due to the fact that concurrency control over a T-tree requires more lock operations than that of a B-link tree, and the overhead of locking and unlocking is high.

1. Introduction

For the past decade, an important assumption of database research and development is that most of the data of a database are on disk. With the advent of the hardware technology, computer systems with main memory size in the order of magnitude of gigabytes are available nowadays. The increasing availability of large and relatively cheap memory makes it possible to have main memory databases (MMDB) where all data reside in main memory, which provides significant additional performance benefits as shown in [8, 16]. In fact, MMDB has been receiving the attention of database researchers for the past decade [6, 1, 3, 10, 17, 4].

When the entire database resides in the main memory, related techniques developed under the assumption of disk I/O as the main cost of database operations should be re-examined. Among the others, index structures that affect the overall system performance heavily has been one of the research focuses. In the early 90's, Lehman and Carey proposed the T-tree as an index structure for main memory database [15]. Because of its good overall performance, the T-tree has been widely accepted as a major MMDB index structure. It was adopted by several systems, including the main memory relation manager (MMM) of the Starburst system from IBM Almaden Research Center [17] and the Dali system from the AT&T Bell Laboratories [11, 4, 22].

The work reported in this paper is motivated by our observation that, in contrast to a large amount of research work on the concurrent B-tree [5, 19, 23, 18, 20, 12, 13, 24], little work has been reported on the study of the concurrent T-tree [14, 17, 8]. As pointed by Lehman et al. [17] and Gottemukkala et al. [8], once the I/O bottleneck of paging data into and out of the main memory are removed, some other factors such as latching and locking dominate the cost of database access. Since the concurrent access of the T-tree may require latching and locking intensively, the performance of the T-tree should be somehow affected in such an environment. Although previous work has demonstrated that the T-tree provides a better performance than the B-tree [15], the performance study did not consider the effects of concurrent access of the indexes.

In this study, we have investigated the performance issue of the T-tree when the concurrent access from multiple users is allowed. Motivated by the good performance of the B-link tree [19, 23, 18], we modified the T-tree into the *T-tail tree*. The T-tail tree allows an extra tree node to be linked to a T-tree node when the T-tree node overflows, to delay the tree rotating operation. Two concurrency control mechanisms were proposed. A performance study was conducted to compare the performance of the T-tail tree and the B-link tree. To our surprise is that the T-tail tree, and hence the T-tree, does not provide a better performance than the B-link tree

* The second author's work is partially supported by a grant from the Research Grant Council of Hong Kong Special Administrative Region, China (No. HKUST758/96E). The third author's work is partially supported by a grant from Sino Software Research Institute (No. SSRI97/98.EG02).
[1] On leave from the School of Computing, The National University of Singapore.
[2] Currently a visiting scholar at the Department of Computer Science, The Hong Kong University of Science and Technology.

because of the high cost of locking and unlocking required to enforce concurrency control.

The rest of this paper is organized as follows. Section 2 describes the T-tail tree index structure and its concurrent access algorithms. A simulation model, experiments conducted, and the results are given in Section 3. Finally, Section 4 concludes the paper.

2. T-tree, T-tail tree and concurrent operations

In this section, we first briefly describe the structure of the T-tree index and its variation, the T-tail tree. Afterwards, we propose two mechanisms that allow concurrent operations including both search and modification on the tree while maintaining the consistency.

2.1. T-tree and T-tail tree

The T-tree [15], rooted in the AVL tree [2] and the B-tree [5], is a balanced *binary* tree whose nodes contain more than one item. Figure 2.1 (a) depicts a T-node, a node of a T-tree. A T-node consists of a number of data pointers, three data fields, 1 parent pointer, 0-1 tail pointer, and 0-2 child pointers. An internal T-node has two child pointers

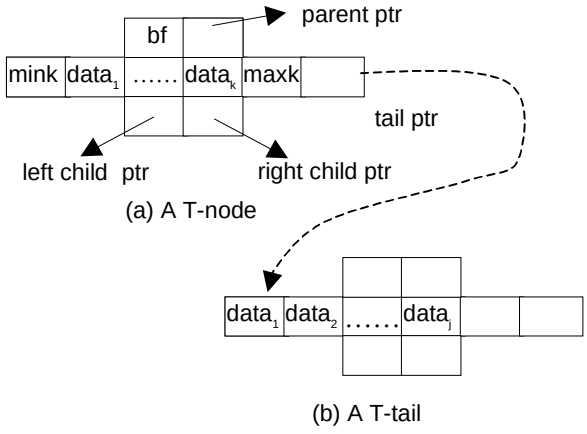

Figure 2.1 Structure of a T-node

pointing to its left and right subtrees, respectively. A *leaf* T-node has no child pointers. A T-node may have only one child pointer and is called *half-leaf* node. The data pointers in a T-node point to the corresponding data entries in the memory, thus through the data pointers, the corresponding data entries and their keys can be accessed. There are also two special fields *minK* and *maxK* in each T-node that store the minimum and maximum key values in the node, respectively. For a node *T* and a value *K*, if $minK \le K \le maxK$, then we say that node *T bounds* the value *K*. Another special data field is the balance factor, *bf*, which is the value of the right subtree height minus the left subtree height. The height of a tree is defined to be its maximum level, the length of the longest path from the root to a leaf node. Since a T-tree is a balanced binary tree, the balance factor, *bf*, can only be +1, 0, or –1. For each internal node *A*, there exists a corresponding leaf (or half-leaf) node that holds a data pointer to the data entry with a key value that is the predecessor to the *minK* of *A*. This node is named the *predecessor* of *A*. Similarly, the node holding the data pointer to the data entry with a key value that is the successor to *maxK* of *A*, is called the *successor* of *A*.

A minimum count and a maximum count are associated with a T-tree. Internal nodes keep their occupancy (i.e. the number of data pointers in the node) in this range. When the number of data pointers in a node is smaller than the minimum count or larger than the maximum count, the node is said to be *underflow* or *overflow*, respectively. In general, a node is not full. That is, the number of data pointers in it is kept less than the *node size*, the maximum number of data pointers a node can have. During the insertion operation, the corresponding data pointer will be inserted into the node that bounds the key value of the entry. When a data pointer is to be inserted into a node that is full, it may cause inserting a new node into the tree. Inserting a new node not only requires redistributing data pointers between the overflowed node and the new node, but also require moving the nodes around, i.e., to rotate the tree to keep the tree balance. Similarly, deletions may cause node underflow or empty, which may give rise to the deletion of a node. Deleting a node may also cause the tree to be rotated for keeping the balance.

Frequent tree rotations will degrade system performance. In observation of that tree rotations are induced by node overflow and underflow, which are consequences of insertion or deletion in nodes with fixed size, we allow the node size to be changed dynamically. In our implementation, a T-node is allowed to have one *T-tail*, in which some data pointers can be inserted. A T-tail bears the same structure as a T-node, but only its data pointer storage space is used. During an insertion operation, if the node is full, a tail is created for it, and then all insertions on the node can be operated on its tail. A T-node is said to be *completely full* if both the T-node itself and its tail are full. Fully tail will be inserted into the tree as a new T-node later. If a deletion causes the node underflow, then data pointers in its tail will be moved into it and empty tail will be deleted. Note that data pointers in the T-node and its tail are sorted by the key values of the corresponding data entries. Also *minK* and *maxK* of the T-node should bound all the key values of the corresponding data entries with pointers in the T-node and its tail. We name such index structure *T-tail tree*. A T-tail is given in Figure 2.1 (b). It is expected that our implementation will reduce the

possibility of tree rotations, hence increasing the performance [21].

To enforce concurrent access over a T-tail tree, we employ a locking mechanism. For simplicity, we use only three types of locks, the shared lock (S-lock), the shared and intention exclusive lock (SIX-lock), and the exclusive lock (X-lock), which were originally used in hierarchical locking protocols [9]. Their compatibility relations are the same as in the original work. That is, shared locks are compatible with themselves and shared and intention exclusive locks. Exclusive locks are incompatible with themselves and all other locks. Shared and intention exclusive locks are compatible with shared locks but incompatible with themselves.

We propose two approaches to enforce concurrent access over a T-tail tree; one is pessimistic and the other is optimistic. For the pessimistic approach, it is assumed that conflicts among tree operations are inevitable and lead to undesirable situations, such as creating inconsistent data or deadlock. Each concurrent operation tries to prevent the happening of such situations. In this approach, search operation use *lock-coupling* in their descent from the root to the *bounding node*, the node that bounds the key value of the data entry in the operation. During update operation (insertion or deletion), all nodes on the way from the parent of the *critical node* to the bounding node are locked using SIX-locks to prepare for a possible tree rotation. A *critical node* is the nearest ancestor of the bounding node whose balance factor equals to 1 or –1. If a tree rotation does occur later, all these SIX-locks will be converted to X-locks. Actually, tree rotations rarely happen over the T-tail tree. Thus this approach handles concurrent access over the tree in a pessimistic way. In the other extreme, the optimistic approach assumes that concurrent operations over a T-tail tree do not interfere with each other, and it allows operations to complete without worrying about possible conflicts. If the bounding node is completely full during insertion, the whole tree will be exclusively locked by the operation and the tree is fixed. During tree fixing, all the T-tails are inserted to its successor as a T-node, and empty nodes and tails are deleted. In the following two sections, we will present the algorithms for the two above-mentioned approaches.

2.2. The pessimistic approach

As the T-tree, and hence the T-tail tree, is evolved from the AVL tree, the pessimistic concurrent operations over them are similar to those on the AVL tree [7]. A search gets an S-lock on the root first. Then it searches down from the root to the bounding node and lock-couples its way with S-locks. Finally it searches the bounding node. During an update operation, the updater first takes the root as a potential critical node and gets an SIX-locks on it. Then it searches down from the root and gets an SIX-lock on each node on the way. If a new node whose balance factor is not equal to 0 is found, then this node is taken as the potential critical node. All SIX-locks on the ancestors of the parent of this potential critical node are released. Finally, an X-lock should be obtained on the bounding node and the operation is performed on it. Later on, if a tree rotation occurs, all the SIX-locks on the nodes from the parent of the critical node to the bounding node will be converted to X-locks and the tree is rotated. The algorithms are described in the followings.

Search. The concurrent searching in a T-tail tree is similar to searching in an AVL tree. The main differences are that the locking mechanism has to be used and the tail has to be searched. The algorithm works as follows:

(1) The search always starts at the root of the tree, and an S-lock should be gotten on the root first.
(2) Search down from the root and lock-couple the way using S-locks.
(3) If the search key value is less than the *minK* of the current node, then search down its left-subtree. Else, if the search key value is greater than the *maxK* of the current node, then search down its right-subtree. Else, search the current node and its tail (if any).
(4) After the search is performed, the S-lock on the current node should be released.

The search fails when a node and its tail (if any) are searched and the corresponding data pointer cannot be found or when a node that bounds the search key value cannot be found.

Insertion. The insertion operation uses SIX-locks and X-locks to enforce concurrency control. It begins with a search to locate the bounding node. If the bounding node is not completely full, the data pointer to the data entry is inserted into the bounding node. Else, if the bounding node is completely full, then its tail will be removed from the bounding node and inserted into its successor as a new T-node. The pointer is inserted into either the original bounding node or the new T-node, depending on which one bounds its key value. The balance of the tree is checked then. If the T-tail tree is unbalanced as a consequence of the insert operation, a tree rotation will be performed. During this process, many locking, unlocking, and lock converting steps are involved. We describe the insertion algorithm in more details as follows:

(1) The search starts at the root of the tree, an SIX-lock is placed on the root, and takes the root as a potential critical node.
(2) Search down the tree for the bounding node. During this process, all nodes from the parent of the critical node to the bounding node are locked using SIX-locks. If some node whose balance factor is not 0 is found on the way, this node becomes the potential critical node. All the SIX-locks on the ancestors of the parent of this new potential critical node will be released. If the search exhausts the tree and no node

bounds the key value of the data entry, the last node on the search path is assigned as the bounding node.

(3) If the bounding node is found and is not completely full, the SIX-lock on the bounding node is converted to the X-lock and the corresponding data pointer is inserted into it. Release all the locks and terminate.

(4) If the bounding node is completely full, then search for the successor of the bounding node like in (2) but the lock on the bounding node is kept. Then the SIX-locks on the bounding node and the successor are converted into the X-locks. The tail of the bounding node is moved off and inserted into the successor as a leaf node. The data pointer will be inserted into the bounding node or the new child node, depending on which one bound the searched key value. Then the balance of the tree is checked.

(5) If a new leaf was added, then check the tree for balance by following the path from the leaf to the critical node. For each node on the way from the leaf to the critical node, if the two subtrees of a node differ in depth by more than two levels, then the SIX-locks on the nodes from the parent of the critical node to the current node are converted into X-locks and a tree rotation must be performed. Once one rotation is done, the tree is rebalanced. Release all the locks and terminate.

Deletion. The deletion algorithm works similar as the insertion algorithm in the way of locking and processing. During the operation, the data pointer to be deleted is searched for, the operation is performed, and then rebalancing is done if necessary. If the deletion does not cause an underflow, then simply delete the data pointer. If it causes an underflow in an internal node, then borrow the data pointer to the data entry with maximum key value in its predecessor. Otherwise, if a deletion makes some leaf node empty, the node will be deleted, the tree should be rebalanced, and rotated if necessary. The algorithm works as follows:

(1) and (2) are the same as in the insertion algorithm. If the bounding node cannot be found, end with failure.

(3) If the bounding node is found and the deletion will not cause it underflow, the SIX-lock on the bounding node converted into the X-lock. The data pointer to the corresponding data entry is deleted from it. Release all the locks and terminate.

(4) If the bounding node is an internal node and the deletion will makes it underflow, then search for the predecessor of the bounding node like in (2) but the lock on the bounding node is kept. All the SIX-locks on the nodes from the bounding node or the parent of the critical node to the predecessor are converted into the X-locks. The data pointer to the data entry with the maximum key value in the predecessor is moved to the bounding node. If the predecessor is a leaf and the deletion makes it empty, then the operation deletes it and the balance of the tree is checked.

(5) If the bounding node is not an internal and the deletion makes it empty, then delete the leaf node or replace the half-leaf with its left child. Then the balance of the tree is checked.

(6) If a leaf was removed, then check the tree for balance as in (5) of the insertion algorithm.

The tree balancing and rotating are the same as in [15], thus we omit it here.

2.3. Optimistic approach

For the optimistic approach, at most one node is locked at a time by an update operation, and search operation does not lock any node at all. It allows empty nodes existing in the tree temporarily. Under such condition, operations are running with maximum concurrency. Only when we try to insert a data pointer into a completely full T-node, the whole tree will be exclusively locked by the operation and the tree is fixed. During the fixing phase, all the tail nodes created after the last tree fixing are adjusted; empty nodes and tails are deleted. Tree balance is checked and rotation is done if necessary.

In this approach, some additional data structures are used. One is the *fixing flag*. It is a logical variable to indicate whether the tree is being fixed (fixing flag equals to *TRUE*) or normal used (fixing flag equals to *FALSE*). The *count* is used to record the number of operations currently operating over the tree. A semaphore is used to protect fixing flag and count from being operated by more than one operation at a time. Three node pointer pools are used. One is the New Node Pool (NNP) that is used to record the newly created tails after the last tree fixing. The second one is the Deallocated Node Pool (DNP) that is used to record the empty tails during the normal tree operations. The third one is the Empty Node Pool (ENP) that is used to record the empty T-nodes.

Search. The search operation is quite straightforward. Before an operation performs on the tree, it will first acquire a semaphore and check the fixing flag. If the fixing flag is set to *TRUE*, then the operation releases the semaphore and waits. Otherwise, it increases the count and releases the semaphore. It then starts from the root and searching for the bounding node. If the data entry has been found, then it acquires the semaphore and decreases the count. After releasing the semaphore, the operation ends with success. Otherwise, it fails.

Insertion. The algorithm works as follows:

(1) The insertion operation searches for the bounding node in the same way as the search operation.

(2) If the bounding node is found, then get an X-lock on it. If the node is not completely full, then insert the corresponding data pointer into it. If there is a newly created tail, record it in the NNP. After releasing the

lock and decreasing the count, the operation ends with success.

(3) If the node is completely full, then release the lock and decrease the count. After setting the fixing flag to *TRUE* and releasing the semaphore, wait until no operation performs on the tree. Then fix the tree. Finally, after resetting the fixing flag to *FALSE*, resume the operation.

(4) If the search exhausts the tree and no node bounds the key value of the data entry, then the last node on the search path is locked using X-locks. If it is not completely full, then process it like in (2). Otherwise, conduct work as in (3).

Deletion. The deletion works in a straightforward fashion. After finding the bounding node, it deletes the corresponding data pointer directly. When the deletion empties a T-node, it is recorded into the ENP. If the deletion makes the tail of the node empty, then record the tail node to the DNP to process it in later tree fixing. Note that the tail cannot be deleted immediately since there may be other operations working on it. It works as follows:

(1) The deletion operation searches for the bounding node in the same way as the search operation.

(2) If the bounding node is found, then get an X-lock on it. Delete the corresponding data pointer from the node. If the deletion makes a T-node empty, it is recorded in the ENP. If the deletion empties its tail, then record the tail into the DNP. After releasing the lock and decreasing the count, end with success.

(3) If the bounding node cannot be found, then release the lock and decrease the count. End with failure.

Tree Fixing. The tree fixing algorithm rearranges the tree. It checks the nodes recorded in the NNP to see if the node can be merged in some way. Empty nodes in the DNP and ENP are deleted. Check the tree balance and conduct tree rotation if necessary. The algorithm works as follows:

(1) For each node in the NNP, try to merge it with its host node. If all the data pointers in the tail can be merged into its host node, then merge them into the host node and the tail is deleted. Else, if the tail cannot be merged into its host node, then insert it to the successor of its host node. Check the tree for balance and rotate the tree if necessary.

(2) For each node in the ENP, if it is empty, then replace it with its predecessor and delete the predecessor. Check the tree for balance and rotate the tree if necessary.

(3) For each node in the DNP, if it is empty, then delete it.

3. A Performance study

In order to investigate the performance of concurrent access over the T-tail tree, a performance study was conducted. We implemented the two concurrency control approaches described in the previous section. As a comparison, operations over the B^+-tree [5] and the B-link tree algorithms [23] were also implemented. Two groups of experiments were conducted. The first group dedicated to the performance of operations over the T-tail tree and the B^+-tree without concurrency control, and the second one studied their performance with concurrency controls. The algorithms were implemented in C++. All experiments were conducted on a Pentium 233 computer with 64M memory, running Linux OS in the single user mode.

3.1. Simulation model

Simulation Process. The simulation work consisted of a number of experiments. Each experiment conducted in three steps. In the first step, a tree was built (for different experiment, the tree might be a T-tail, B^+, or B-link tree) to provide basic data structure for the experiment. A set of operations (searches, insertions, and deletions) is then performed on the tree. Finally, the experiment results were collected.

Tree Initialization. Before each simulation, a tree was built by inserting 0.5M (Million) data pointers to data entries, whose keys were randomly selected from the key space of 1 to 1M. To make the tree more realistic, 0.5M update operations were then performed. In those operations insertion and deletion had even distributions, thus about 0.25M insertions and the same number of deletions were performed on the tree.

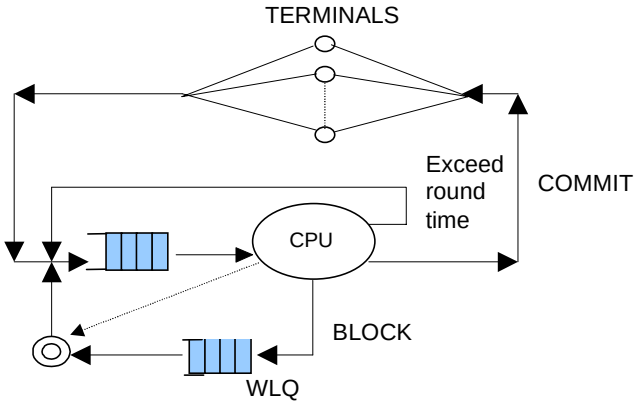

Figure 3.1 Simulation model

Simulation Model. Our model was a closed-queuing model as outlined in Figure 3.1. There are a number of terminals and each one performs some operations on the tree. Each terminal issues a request, which is one of the tree operations (search, insert, and delete). The request is inserted into the Waiting CPU Queue (WCQ). The CPU is scheduled using a round-robin discipline without preemption. When the CPU becomes free, the first request in the WCQ is assigned to it. If an operation cannot be

finished in a round, it will be sent to the WCQ again. After an operation is committed, the terminal waits for some time and submits the next operation again.

For concurrent operations with concurrency control, a lock manager [9] is used to maintain the locks. An operation is blocked if it cannot obtain the required lock on some resource. Then it is sent to the Waiting Lock Queue (WLQ). It will be re-sent to the WCQ when the required lock is available.

Measurements. During the experiments, we mentioned the processing time, the tree height, and the number of tree nodes. The processing time is the total time requested to complete a certain number of operations.

3.2. The CPU cost for each basic operation

A tree operation consists of a number of basic operations such as data comparison, pointer assignment, arithmetic operation, acquisition and release of semaphore, and locking and unlocking. Since all tree operations are performed in memory, the processing times for each of these basic operations decide the cost of each tree operation. Thus it is important to identify the most expensive basic operation. We conducted a set of simple tests to measure the CPU costs for the five basic operations mentioned above.

Table 3.1 CPU cost for each operation

Basic operations	Time ($x10^{-9}$ seconds)
Data Comparison	128
Pointer Assignment	4
Arithmetic Operation	55
Acquire and Release semaphore	8947
Lock and Unlock	17466

We employed a main function that looped to execute each operation 1 giga (10^9) times. Also, an empty function was called those times so as to measure the cost of a function call. Finally, the cost of the function call was subtracted from the cost of the executions of each operation. The net cost of each operation is listed in Table 3.1

3.3. Experiment results

The experiments were classified into two groups. In the first group we investigated the performance of the T-tail tree and the B^+-tree without concurrency control. The performance of concurrent accesses on the T-tail tree and the B-link tree was studied in the second group of experiments.

Performance of Tree Operations without Concurrency Control. In this group of experiments, operations were completed one by one without concurrency. Only one operation was working on the tree at any time. Under such settings, the algorithms operated on the T-tail tree were almost the same as the original ones proposed in [15]. The results of the T-tail tree are denoted as the *T-tail tree* in the figures. For the B-tree, we used the standard B^+-tree algorithms in [5] (denoted as the *B^+-tree* in the figures). The purpose of this group of experiments was to study the processing time, the number of nodes, and the height of a tree while there was no concurrency control enforced. The results are reported as follows.

Figure 3.2 presents the total processing time required for 0.5M ($5x10^5$) operations. We used a workload that the update ratios were varied from 0 to 100%. For update, there was the same number of insertions and deletions. In this experiment, the fan-out and the size of the leaf node for the B^+-tree were 10, and the node size of the T-tail tree had the same number. From the figure it can be seen that, operations on the B^+-tree always costs more than those on the T-tail tree. The underlying reason is that, during a search over the B^+-tree, a linear (binary) search is performed in every node before going to the next level. In the T-tail tree, only two data comparisons (*minK* and *maxK*) are conducted before going to the next level. Note that, without concurrency control, data comparison is the most expensive basic operation among the other basic operations. With the increase of the number of the update operations, node splits and concatenations occur frequently. A node split or concatenation will cause a recursive insertion or deletion of keys in some nodes, which may involve a lot of data comparisons, thus imposes much additional cost for the operation. For the T-tail tree, rotations occur relatively infrequent and the search cost is lower.

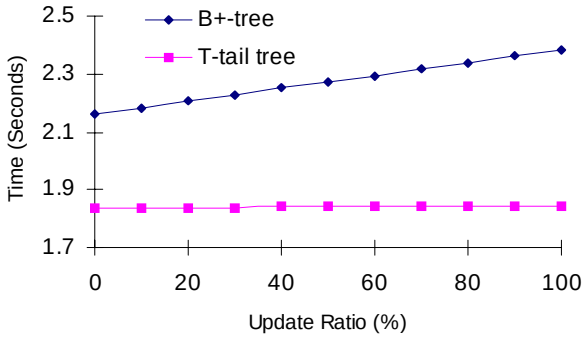

Figure 3.2 Processing time

Given 0.5M data entries, Figure 3.3 depicts the numbers of nodes for both the T-tail tree and the B^+-tree, with varying node sizes from 5 to 15. From the figure, we observe that the B^+-tree has far more nodes than the T-tail tree when the node size is small. It is due to the fact that the B^+-tree puts all data in leaves and all internal nodes are taken as index. Compared with the T-tail tree, the B^+-tree

holds additional nodes for indexing. With small node size, there should be more leaf nodes than those with larger node size. With the increase of node size, the node numbers for both trees become smaller.

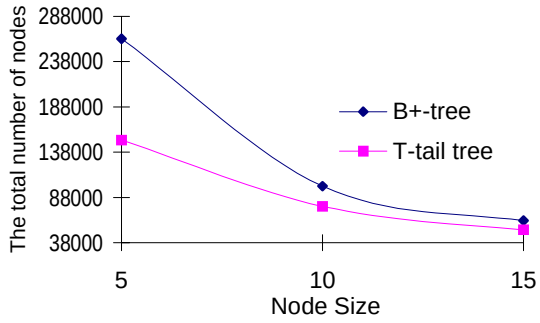

Figure 3.3 Number of nodes

For varying node size, Figure 3.4 gives the tree heights of both trees of 0.5M data entries. It is obvious that the T-tail tree is much higher than the B^+-tree. The reason is straightforward. Since the T- tail tree is a balanced binary tree and at each level there are only two subtrees. In that sense, nodes 'pile up' much faster than the B^+-tree for its fanout and leaf node size were 10 in our experiment. For operations without concurrency control, it is faster to search in the T-tail tree than to search in the B^+-tree, since the former involves many relatively cheap pointer assignments and fewer expensive data comparisons, the later employs many expensive data comparisons. While operations concurrent perform on the T-tail tree, the higher tree height will reduce its performance heavily for the sake of locking and unlocking. We will show this point in the following group of experiments.

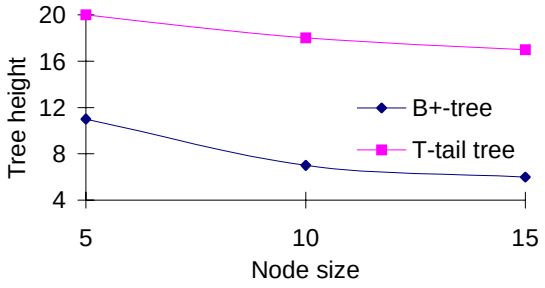

Figure 3.4 Tree height

Performance of Tree Operations with Concurrency Control. In this group of experiments we investigated into the performance of both the T-tail tree and the B-link tree with presence of concurrent operations. A lock manager was employed to maintain the lock resources. For the T-tail tree, we implemented all the algorithms in both the pessimistic and the optimistic approaches presented in section 2. The experiment results of the pessimistic approach and the optimistic approach are denoted as the *T-tree pessimistic* and the *T-tree optimistic* in the figures, respectively. For the B-tree, we implemented the B-link tree algorithms [23], whose experiment results are denoted as the *B-link tree* in the figures. In this group of experiments, we studied the effects on the processing time by varying the update ratio, number of operation, and tree size.

Figure 3.5 shows the effect on the processing time by varying the update ratio. In the experiment, both the T-tail tree and the B-link tree had the same number of data pointers to data entries, 0.5M. The total number of operations conducted on the trees were 0.5M. The node size and the fan-out were 10. From the figure, it can be seen that, the B-link tree algorithms and the T-tail tree optimistic approach (optimistic approach hereafter) are much better than the T-tail tree pessimistic approach (pessimistic approach hereafter). For different update ratio, the processing time for the pessimistic approach varied from 136.6 seconds to 140.4 seconds. The time for the optimistic approach was from 10.8 seconds to 19.6 seconds. For the B-link tree algorithms, it was only from 2.2 seconds to 11.3 seconds. That is, 5 or 10 times more efficient than the algorithms on the T-tail tree. It is because pessimistic approach uses many expensive locking and unlocking primitives in each operation. Even though the optimistic approach uses fewer locking and unlocking, it employs acquiring and releasing semaphore, the second most expensive primitives. Moreover, it makes the tree exclusively locked by some insertion operations, which reduces the concurrency of the operations. On the contrary, the B-link tree algorithms lock only one node simultaneously.

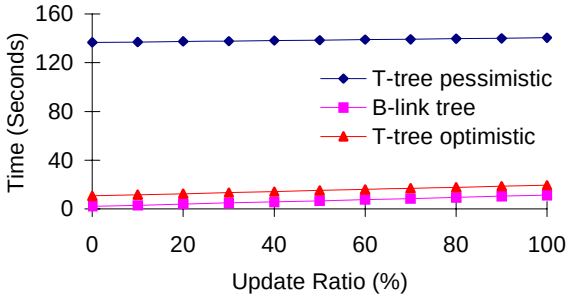

Figure 3.5 Effect of update ratio

Modifying the number of operations, we re-conducted the experiments. The experimental results are presented in Figure 3.6. The number of operations was varied from 0.1M to 1M. The workload consisted of search 80% and 10% each of insertion and deletion. From the figure, it can be observed that the relations among these three curves are the same as in the previous figure. For different number of operations, the processing time for the pessimistic approach varied from 27.4 seconds to 274.5 seconds, the time for the optimistic approach from 2.5 seconds to 25.1 seconds, and

that for the B-link tree algorithms was from 0.8 seconds to 8.0 seconds. The reason is that, since the workload and the number of data entries in the tree are kept unchanged during the experiment, each operation has the same cost in each approach. But they are different for different approaches. Thus the three curves increase linearly, but with different trends.

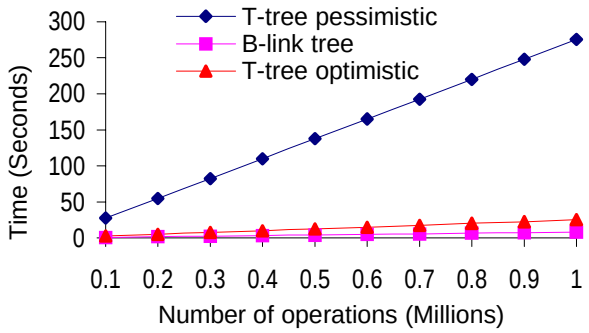

Figure 3.6 Effect of the number of operations

We further varied the tree size and re-tested the algorithms on the trees. The result is given in Figure 3.7. In this experiment, the tree size was varied from 0.1M to 1M data entries. The number of operations was 0.5M, and the workload was the same as in the previous experiment. From the figure, we observe that the relations among the three curves are unchanged also. That is, the B-link tree algorithms are the best, then the optimistic approach, and the last one is the pessimistic approach. The reason is that, with the increment of the tree size, the height of the T-tail tree increased, thus giving rise to more overheads of the locking and unlocking. On the other hand, the numbers of the locking and unlocking in the other two approaches are almost not affected by the varying of tree size. Therefore, the processing time for the pessimistic approach is increased with the growing of the tree size. But the times for the other two approaches are almost unchanged.

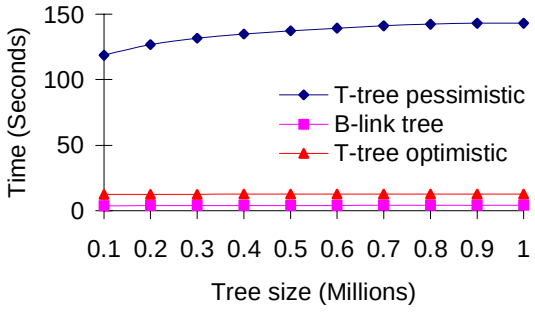

Figure 3.7 Effect of the tree size

Discussions. The experiments in the first group gave the same conclusion as [15]. That is, the T-tail tree performs better than B^+-tree without concurrency control. However, with concurrency control, it seems the B-link tree performs better than the T-tail tree, and hence the T-tree. It can be explained in the followings.

Lehman et al. [17] and Gottemukkala et al. [8] pointed out that, locking and unlocking are the most expensive operations in the main-memory data management system. This is actually our test result presented in Table 3.1. Even the values may be different for different system, but locking and unlocking are several orders of magnitudes more expensive than the other operations in our test. A natural consequence of this observation is that, in main-memory environment, any concurrency control mechanism with more locking or unlocking should not perform better than those with fewer locking and unlocking.

The pessimistic approach employs many locking and unlocking operations. In the search phase, each operation locks and unlocks every node on the way once. For update operations, they will locks all the nodes from the critical node to the bounding node simultaneously. However, for the B-link tree algorithms, one update operation locks far fewer nodes than that in the pessimistic approach. Even the optimistic approach locks fewer nodes also, but it has to acquire and release semaphore. Moreover, some insert operations may exclusively lock the tree for fixing, which decreases the performance. When the whole tree is taken as a single node and each operation exclusively locks the whole tree during its processing, the B-link tree still outperforms the T-tail tree if the update ratio is not very high [21].

The tree height is another factor that affects the performance of the locking and unlocking in the T-tail tree. Given a fixed number of data entries, a T-tail tree is much higher than its B-link counterpart (see Figure 3.4). This always leads to, the existence of a large number of nodes on the way between the critical node to the bounding node, which imposes much cost for the update operations. The third factor that makes it difficult for designing concurrency control mechanism on a T-tree is that, a tree rotation always involves a critical node, which may be far away from the bounding node. This makes the locking mechanism hard to use and sometimes it has to lock more nodes to avoid deadlock or lock conflicts.

To design concurrency control mechanism over T-trees, two factors should be given much concentration. One is the number of locks one operation should put on nodes and the other is the degree of concurrency. These two factors may affect the performance heavily as shown in our experiments.

4. Conclusions

In this paper we studied the performance of concurrent operations over a T-tree, a well-known main memory database indexing structure. Two concurrency control approaches, the pessimistic approach and the optimistic approach were presented. A simulation model was built to investigate the performance of the B-tree and the T-tree with different concurrency control approaches.

Without enforcing concurrency control mechanisms, our results conform to the previous reported work. That is, when the data reside in memory, the T-tree index does outperform the B-tree index. However, when concurrency control mechanisms are enforced, both pessimistic and optimistic approaches make the T-tree index a worse index structure than the B-tree index. The basic reason is that, although the T-tree reduces comparison time within a node, the overhead of a large number of locking and unlocking is too high. Therefore, unless we can provide better algorithms to reduce the number of locks, the concurrent B-trees will outperform the T-trees.

References

[1] A. C. Ammann, M. B. Hanrahan, R. Krishnamurthy, "Design of a Memory Resident DBMS", *Proc. IEEE COMPCON Conf.* Los Alamitos, CA, 1985 pp.54-57.

[2] A. Aho, J. Hoperoft, J. D. Ullman, The Design and Analysis of Computer Algorithms, *Addison-Wesley Publishing Company*, 1974.

[3] D. Bitton, M. B. Hanrahan, C. Turbyfill, "Performance of Complex Queries in Main Memory Database Systems", *Proceedings of the International Conference On Data Engineering*, Los Angeles, California, 1987, pp. 72-81.

[4] P. Bohannon, D. Lieuwen, R. Rastogi, A. Silberschatz, S. Seshadri, S. Sudarshan, "The Architecture of the Dalí Main-Memory Storage Manager", *Multimedia Tools and Applications*, Volume 4, Number 2, 1997, pp. 115-151.

[5] D. Comer, "The Ubiquitous B-tree", *ACM Computer Surveys*, Volume 11, Number 2, June 1979, pp. 121-137.

[6] D. J. DeWitt et all., "Implementation Techniques for Main Memory Database Systems", *Proceedings of ACM-SIGMOD Int'l Conference on Management of Data*, Boston, MA, June 1984, pp.1-8.

[7] C. S. Ellis, "Concurrent Search and Insertion in AVL-trees", *IEEE Transactions on Computers*, Volume 29, Number 9, September 1980, pp. 811-917.

[8] V. Gottemukkala, T. Lehman. "Locking and Latching in a Memory-Resident Database System", *Proceedings of the 18th International Conference on Very Large Databases*, Vancouver, British Columbia, Canada. 1992, pp.533-544.

[9] J. Gray, "Notes on Database Operating Systems", *Operating Systems: An Advanced Course*, Volume 60, Springer-Verlag, New York, 1979.

[10] H. Garcia-Molina, K. Salem, "Main Memory Database Systems: An Overview", *IEEE Transactions on Knowledge and Data Engineering*, Volume 4, Number 6, December 1992, pp. 509-516.

[11] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, S. Sudarshan, "Dali: A high Performance Main-Memory Storage Manager", *Proceedings of the 20th International Conference on Very Large Databases*, Santiago, Chile, 1994, pp. 48-59.

[12] T. Johnson, D. Shasha, "A Framework for the Performance Analysis of Concurrent B-tree Algorithms", *Proceedings of ACM-SIGMOD Int'l Conference on Management of Data*, Atlantic City, New Jersey, 1990, pp. 273-287.

[13] T. Johnson, D. Shasha, "The Performance of Current B-Tree Algorithm", *ACM Transactions on Database Systems*, Volume 18, Number 1, 1993, pp.51-101.

[14] V. Kumar, "Concurrency Control and Recovery in Main Memory Databases", *The Journal of Computer Information Systems*, Vol. 30, No. 3, USA, Spring 1990, pp. 24-30.

[15] T. J. Lehman, M. J. Carey, "A Study of Index Structures for Main Memory Database Management Systems", in *Proceedings 12th Int. Conf. On Very Large Database*, Kyoto, August 1986, pp. 294-303.

[16] T. J. Lehman, V. Gottemukkala, "The Design and Performance Evaluation of a Lock Manager for a Memory-Resident Database System", V. Kumar (Ed.), *Performance of Concurrency Control Mechanisms in Centralised Database Systems*, Prentice-Hall, 1996. pp.406-428

[17] T. Lehman, E. J. Shekita, L. Cabrera, "An Evaluation of Starburst's Memory Resident Storage Component", *IEEE Transactions on Knowledge and Data Engineering*, Volume 4, Number 6, December 1992, pp.555-566.

[18] V. Lanin, D. Shasha, "A Symmetric Concurrent B-Tree Algorithm". *Proceedings of the Fall Joint Computer Conference*, Washington, DC, USA, 1986, pp. 380-389.

[19] P. L. Lehman, S. B. Yao, "Efficient Locking for Concurrent Operations on B-trees", *ACM Transactions on Database Systems*, Volume 6, Number 4, 1981, pp.650-670.

[20] C. Mohan, "ARIES/KVL: A key Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes", *Proceedings of the16th International Conference on Very Large Databases*, Australia,1990, pp.392-405.

[21] Y. Y. Ng, "Concurrency Control on Index Structures for Main Memory Database Management System", Mphil thesis, Department of Computer Science, the Hong Kong University of Science and Technology, June 1999.

[22] R. Rastogi, S. Seshadri, P. Bohannon, D. Lieuwen, A. Silberschatz, S. Sudarshan, "Logical and physical versioning in main memory databases", *Proceedings of the Twenty-Third International Conference on Very Large Databases*, Athens, Greece, 1997, pp.86-95.

[23] Y. Sagiv, "Concurrent Operations on B-Tree with Overtaking", *Proceedings of ACM SIGACT/SIGMOD Symposium on the Principles of Database Systems*, ACM, New York, 1985, pp. 28-37.

[24] V. Srinivassan, M. J. Carey, "Performance of B+ Tree Concurrency Control Algorithms", *VLDB Journal*, Volume 2, Number 4, 1993, pp. 361-406.