Proceedings of the Third IASTED International Conference
ADVANCES IN COMPUTER SCIENCE AND TECHNOLOGY
April 2–4, 2007, Phuket, Thailand
ISBN CD: 978-0-88986-656-0

T^{LINK}-TREE: MAIN MEMORY INDEX STRUCTURE WITH CONCURRENCY CONTROL AND RECOVERY

Jan Lindström
Solid Information Technology Ltd
Itlahdenkatu 22 B
00210 Helsinki Finland
email: jan.lindstrom@solidtech.com

ABSTRACT
While B+trees are the most popular index structure for disk databases, the T-Tree has been widely accepted as a promising index structure for main memory databases. For over two decades the speed of CPUs has increased faster than the speed of the main memory. Therefore, the speed gap of the CPU and memory has rapidly become steeper, and memory access has become the new bottleneck in databases. In this paper we explore concurrency control and recovery in a new index structure T^{link}-tree which enhances the cache usage, reduces the number of rotate operations and the overhead required for balancing the tree by delayed split and deletion.

KEY WORDS
index, data structures, database systems, main-memory, cache-conscious

1 Introduction

Database index structures have been researched for decades. Disk databases use hard disk as storage and the main issue is to keep the number of disk operations as low as possible. This has led to the popularity of low, wide tree-structures like B+tree [1]. The B+tree suits well for disk databases and are the most common index structure used.

Since a relatively large main memory has become available, partially or fully memory resident databases have became feasible. This has motivated the research of index structures especially designed for main-memory databases (MMDB) [2]. Unfortunately, over two decades the speed of CPUs has increased faster than the speed of main memory. Therefore, the speed gap between the CPU and main memory has rapidly become steeper, and main memory access has become the new bottleneck in main memory databases [3]. Thus, it is crucial to minimize both the number of main memory accesses and the latency incurred from the main memory accesses.

The B+tree have many benefits such as short access paths with equal length and high node fan-out, i.e., the number of children per node, is high. However, these benefits lose their importance if main memory access is relatively fast [4]. Therefore, T-tree was proposed [5]. T-Tree offers efficient CPU-cycle usage, enhances the poor storage properties of the AVL-tree [6]. Each T-tree node has multiple elements, thus reducing the need for rotations typical to balanced binary trees. To achieve better scanning properties T*-tree [7] was proposed where data is stored on leaf nodes.

Main-memory structure organization and cache usage is topical due to the increase in the memory processor speed gap. Therefore, within the past few years many cache-conscious data structures [8, 9, 10], programming conventions [11] and design issues [12] has been proposed.

In this paper, we will extend T_{link}-tree proposed in [13] with a concurrency control and recovery methods. We will use similar methods as in [14]. In our knowledge this is the first time when concurrency control and recovery algorithms are presented to the T-Tree and its variants.

The remaining part of the paper is organized as follows. In Section 2 we describe T-tree structure. In section 3 we propose a new T_{link}-tree structure in detail and present basic algorithms. Finally, Section 4 gives the conclusions of this paper.

2 T-tree

The T-tree [5] is the most widely used index structure for main memory database system (e.g. Starburst [15], Dali [16], and Oracle TimesTen). The T-tree is a combination of the AVL-tree [6] and B-tree [1]. Every node of the T-tree contains several pairs of data items containing the key and the pointer to a actual data. All items are sorted by the key value within the node. Structure of the T-tree is similar to binary trees (see Figure 1).

Definition 2.1 *Maximum key value of the leaf node of left sub-tree of the node p is called Greatest Lower Bound (GLB).*

Definition 2.2 *Minimum key value of the leaf node of right sub-tree of the node p is called Least Upper Bound (LUB).*

Example 1 *Consider T-tree in the Figure 1(b). GLB of the root node is the key 15 and LUB of the root node is the key 32.*

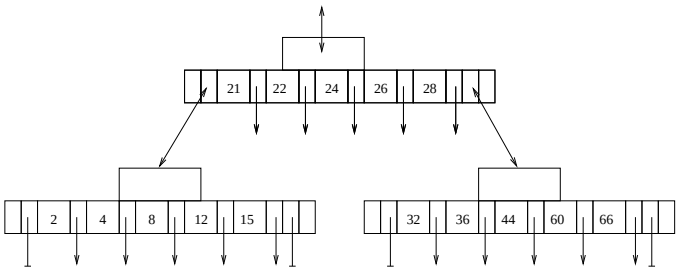

Figure 1. Node structure and organization of the T-tree

T-tree has shown to be effective index structure for main memory databases [5] but it has three drawbacks. Firstly, due to its in-order traversal the T-tree performs unnecessary traversal up and down of the tree in range queries and when pages split or merge. Secondly, when a data item is inserted or delete into/from the internal node, access to the GLB node is needed. This causes unnecessary traversal up and down of the tree. Additionally, the possibility for a new node insertion and/or deletion becomes high because it test the GLB node only in case of node overflow and underflow. Finally, the height of the tree needs to be always calculated in order to check the balance of the tree. If tree is out of balance costly rotation operation is triggered.

3 The T^{link}-tree

The most important problems of the original T-tree are the height of the tree and bad performance in range queries. Therefore, we have designed methods to decrease the height of the tree significantly. Additionally, we have added successor and predecessor pointers to every node in the T^{link}-tree to enhance speed of the range queries.

Database record of the T^{link}-tree reside on record pages where records are in a sorted order (see Figure 2). T^{link}-tree has a structure similar to a B-link tree [17]. The Figure 2 shows the node organization of the T^{link}-tree. A node contains a left and right child pointer, a parent pointer, successor pointer, and a predecessor pointer. Furthermore, a node contains a low key value of records pointed by a key pointer. Thus, every key value represents a range of records between key value and next key value. Last key value of the node represents the high-value of the left most record page. Logical structure of the T^{link}-tree is similar to T-Tree and shown in Figure 2.

Formally, a T^{link}-tree is an array $T[0, ..., n]$ of pages $T[P]$ indexes by unique *page identifiers* $P = 0, ..., n$. We define our T^{link}-tree as a sparse database index to the database so that all pages store only the page identifier to actual database records. An *index* page P is a page and contains a list of *index records* of the form $(k_1, P_1), (k_2, P_2), ..., (k_n, P_n)$ where $k_1, k_2, ..., k_n$ are key values and $P_1, P_2, ..., P_n$ are page identifiers to database records. Note that the index page can be either a non leaf or a leaf page.

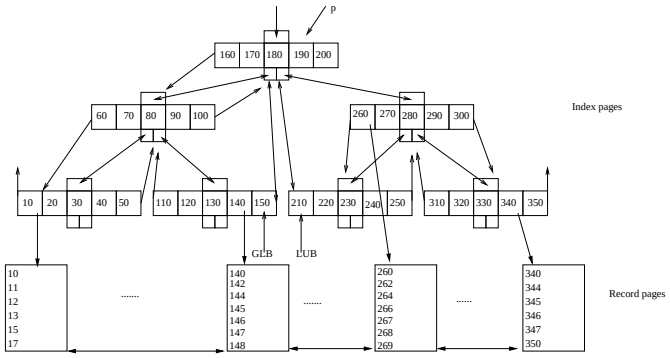

Figure 2. Organization of the T^{link}-tree

T^{link}-tree is *structurally consistent* if it satisfies the basic definition of the T-tree, so that each page can be accessed by following a child link to the eldest child and then following the sideways links, level by level. A structurally consistent T^{link}-tree can contain underflow pages and chains of successive sibling pages that are indirect children of their parent. Structurally consistent T^{link}-tree is *balanced* if (1) none of its non root pages is underflow, (2) no indirect child page has a right sibling page that is also an indirect child, and (3) difference between the height of the right and left sub trees of all nodes in the tree is never more than one.

3.1 Concurrency control and Recovery

In a centralized database system, $locks$ are usually used to ensure the logical consistency of the database under a concurrent history of transactions. Additionally, $latches$ are used to ensure the physical consistency of a database page under a single operation. A latch is implemented by a low-level primitive (semaphore) that provides a cheap synchronization mechanism but no deadlock detection. Lock request may be made with the conditional or the unconditional option. A *conditional* request means that the requester (transaction) is not willing to wait if the lock cannot be granted immediately. An *unconditional* request means that the requester is willing to wait until the lock is granted. We use *physiological logging* [18] as a compromise between physical logging and logical logging. We assume that the buffer manager employs the *steal* and *no-force* buffering policies [18].

Physical redo means that, when a page update needs to be redone at recovery time, the page identification fields of the log record are used to determine uniquely the affected pages. Similarly, *physical undo* means that, when a page update needs to be undone during transaction rollback, the page identification fields of the log record are used to determine the effected pages. The pages are then accessed and the update is undone on these pages. Physical redo and undo provide faster recovery because only the pages mentioned in the log record are accessed.

We use a redo and undo recovery protocol for handling the system failures. Restart recovery supports physical redo, physical undo (if possible), logical undo, concurrency control at the record level, and the steal-and-no-force policies for the buffer management. The restart recovery protocol is base on ARIES [18].

3.2 Search algorithm

Any successful index structure should provide support for a two kinds of search operations. These search operations are random search for a particular item and a sequential search of a range of items. To perform a search $Search[key, \theta value]$ to fetch the first matching record $(key, page)$. Given a key value $key < \infty$, find the last key value key and the associated record page identifier $page$ such that key satisfies $key\theta value$ and the record $(key, page)$ is in the database. Here θ is one of the comparison operators.

Sequential search algorithm for the T^{link}-tree utilizes the successor and predecessor pointers to reduce the overhead. Note that in the T-Tree sequential search requires moving up and down in the tree. To simulate the fetch-next operation on a key range $[min, max]$, the search operation is used as follows. The search the first record in the key range, a transaction T issues $Search(key_1, \geq min)$. To fetch the next record in the key range, T issues $Search(key_2, \geq key_1)$ and so on. The sequential search algorithm operates as follows:

1. The search always starts at the root of the tree.
2. If the node is not a leaf node of the tree and the search value is less than the minimum value of the node, then search continues down the sub tree pointed by the left-child pointer. Else, if the search value is greater than the maximum value of the node, then search continues down the sub tree pointed by the right-child pointer.
3. If the node is a leaf node and the search value is greater than the maximum values of the node, then search continues left the tree pointed by the successor pointer. Else, search the current node.

To simplify the presentation of our algorithms, we will use the following notations.

- **slatch(P):** Fix page P and acquire an shared latch on P.
- **xlatch(P):** Fix page P and acquire an exclusive latch on P.
- **unlatch(P):** Release the latch on page P and unfix P.
- **slock(r):** Acquire an unconditional shared lock on record r.
- **xlock(r):** Acquire an unconditional exclusive lock on record r.
- **unlock(r):** Release the lock on record r.

To perform $Search[key, \theta value]$ operation, a transaction T takes as input a key value key and uses $Search(key, P)$ algorithm to find the target leaf page P that covers the database record with key value key. The algorithm traverses the tree using the latch-coupling method with S latches and returns the page identification of the leaf page that covers key.

```
Search(key, P) {
    P = root page; slatch(P);
    do {
        h = high-key(P);
        l = low-key(P);
        if (key < l) {
            Q = P.left;
            slatch(Q);unlatch(P);
            P = Q;
        } else if (key > h) {
            if (P is a leaf page) {
                Q = P.successor;
            } else {
                Q = P.right;
            }
            slatch(Q);unlatch(P);
            P = Q;
        } else {
            search key from P;
            if (bounding key value found) {
                slatch(P.ptr); R = P.ptr; unlatch(P);
                search key from R;
            }
            return R or NULL if not found;
        }
    } while (P);
}
```

Figure 3. Search algorithm.

3.3 Insert algorithm

The insert algorithm of the T^{link}-tree is also very similar with that of the T-Tree. A major difference is that the predecessor and successor pointers are used to access neighbor nodes. Additionally, the overflow handling method is different. When node overflow occurs in the T-Tree the minimum value of the current node is inserted into the GLB node if there is free space. GLB node is founded traversing the tree up and down. If there is no space available for insertion a new leaf node is created and inserted.

In the T^{link}-tree the LUB node is first checked for available space using the successor pointer. If it has enough space, the maximum value of the current node will be inserted there. Otherwise, the GLB node is checked for available space using the predecessor pointer. If it has enough space, the minimum value of the current node will be inserted there. If both nodes do not have enough space, then two new nodes are created below the overloaded node. Keys are evenly distributed between these three nodes.

As this approach uses both the GLB and LUB nodes, it reduces the possibility of creating a new node compared to T-Tree. This reduces check and rotates operations for tree balancing. Additionally, this also increases space utilization in the T^{link}-tree. We call this algorithm split-delay insert algorithm (see also Figure 4):

1. Search the node where the value belongs.

2. When a node is found, then we check for room for another entry. If the insert value fit, then insert it into this node and stop. Else, check weather there is room in GLB node (this can be accessed using predecessor pointer. If there is room, then remove the minimum value from the node and insert this to GLB node and insert the insert value the original node. If GLB node has no room, check weather LUB node has room using successor link. If there is room, then remove the maximum value from the node and insert this to LUB node and insert the insert value to the original node. Else, create a new leaf and insert the insert value to this node.

3. If the search exhausts the tree and no node is found, then insert the value in the last node on the search path. If the insert value fits, then insert the value to the node. Else, check weather there is room in GLB node (this can be accessed using predecessor pointer. If there is room, then remove the minimum value from the node and insert this to GLB node and insert the insert value the original node. If GLB node has no room, check weather LUB node has room using successor link. If there is room, then remove the maximum value from the node and insert this to LUB node and insert the insert value to the original node. Else, create a new leaf and insert the insert value to this node.

4. If a new leaf was added, then check the tree for balance by following the path from the leaf to the root. For each node in the search path, if the two sub trees of a node differ in depth by more than one level, the rotation (see Section 3.4 must be performed. Once one rotation has been done, the tree is rebalanced and processing stops.

The Insert(key, x, T) algorithm implements the $Insert[key, x]$ database operation in the forward-rolling phase of transaction T. The given record $R = (key, x)$ is inserted into the database. The insertion is redo-undo-logged for transaction T, and the inserted record r is X-locked for T for commit duration.

```
Insert(key, x, T) {
    P = Search(key);
    if (free space) {
      xlatch(P); insert key to P;
      log(n,<T,insert,P,(key,x),Last-LSN(T)>)
        unlatch(P);
    } else {
      Q = P.predecessor; xlatch(Q);
      if (free space) {
        move min-key from P to Q;
        insert key to P;
        log(n,<T,delete,P,(min-key,r), Last-Lsn(T)>)
        log(n,<T,insert,P,(key,x),Last-LSN(T)>)
        log(n,<T,insert,Q,(min-key,r),Last-LSN(T)>)
        unlatch(P); unlatch(Q);
      else {
        Q = P.successor; xlatch(Q);
        if (free space) {
         move max-key from P to Q;
          insert key to P;
          log(n,<T,delete,P,(max-key,r), Last-Lsn(T)>)
          log(n,<T,insert,P,(key,x),Last-LSN(T)>)
          log(n,<T,insert,Q,(max-key,r),Last-LSN(T)>)
          unlatch(P); unlatch(Q);
       else {
          unlatch(Q);
          R=Split(P,Q)
          Link(P,Q,R)
          Rebalance(P);
          unlatch(P);
       }
      }
    }
}
```

Figure 4. Insert algorithm.

As a example consider a T^{link}-tree in the Figure 2. Inserting a new key with value 143 causes a page overflow. There is no space both on GLB and LUB nodes. Therefore, a high value of the node 148 is removed and inserted to a new node (see Figure 5).

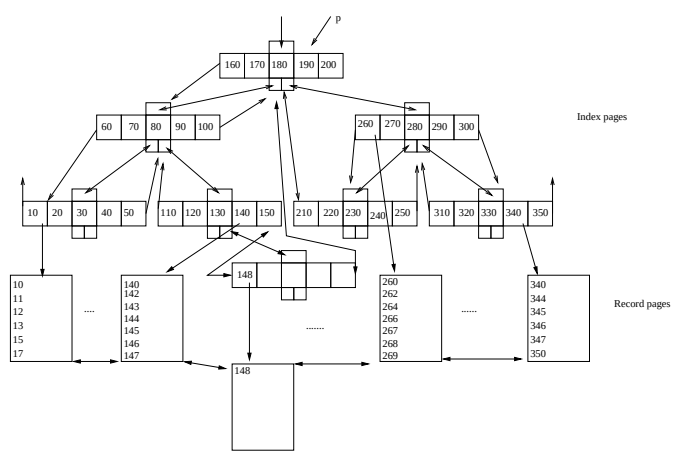

Figure 5. T^{link}-tree after key value 56 is inserted.

The operation $split(Q)$ is used to split a full page Q. The operation allocates a new page R, moves the upper half

of the records from Q to R, and makes R a right sibling of Q.

```
Split(Q) {
    allocate(R);
    move half of the records from Q to R;
    let records to be list of moved records;
    log(n,<split, Q, R, (records)>);
    Page-LSN(Q)=PageLSN(R)=n;
  }
```

Figure 6. Split algorithm.

The operation $link(P, Q, R)$ is used to link the right sibling page R of page Q to the parent page P.

```
Link(P,Q,R) {
    if (leaf) {
        insert the index record (v,R) into P
        change index record (v,Q) to (u,Q) in P
        log(n,<link,P,(u,Q),(v,R)>)
        Page-LSN(P)=n;
    } else {
        if (insert on left) {
    R.successor = Q;
            R.precessor = Q.precessor;
    Q.precessor.successor = R;
            Q.precessor = R;
    R.parent = Q;
    Q.left = Q;
            log(n,<plink,left,R,Q)>)
            Page-LSN(Q)=n
            Page-LSN(R.precessor)=n;
        } else {
            R.successor = Q.successor;
            Q.successor.precessor = R;
            Q.successor = R;
            R.precessor = Q;
            R.parent = Q;
            Q.right = R;
            log(n,<plink,right,R,Q)>)
            Page-LSN(Q)=n
            Page-LSN(R.successor)=n;
        }
    }
  }
```

Figure 7. Link algorithm.

3.4 Delete algorithm

We will use delayed delete method where a node is removed from a tree only when whole node group becomes empty. This leads to a reduction in the number of check and rotate operations for tree resulting improved performance. Firstly, search the node where the deleted value is. Then, delete the value from the node. If the node is empty we mark node to be removed. If a node is removed, then check the tree for balance by following the path from the leaf to the root. For each node in the search path, if the two sub trees of a node differ in depth by more than one level, the rotation must be performed. Once one rotation has been done, the tree is rebalanced and processing stops.

3.5 Rebalancing algorithm

Despite the additional successor and predecessor pointers, the rotate algorithm of the T^{link}-tree is the same that of T-Tree. Specifically, the rotate operation works as follows. When a leaf node group is inserted or deleted, the rebalancing mechanism of the T^{link}-tree checks the difference of the depth of the two sub trees. This depth is stored control structure of the node group on every node. If the difference is larger than one, rebalancing is required.

4 Conclusion

While B+tree is the most popular index structure for disk databases, the T-Tree has been widely accepted as a promising index structure for main memory databases. In this paper we explored a new index structure T^{link}-tree which enhances the cache usage, reduces the number of rotate operations and the overhead required for balancing the tree by delayed split and deletion.

We have extended our proposed method with a concurrency control and recovery methods. This is the first time to our knowledge when concurrency control and recovery algorithms are presented to the T-Tree and its variants.

More studies are needed to verify the proposed method by fully implementing our algorithm and comparing it to other cache sensitive methods.

References

[1] D. Comer, "The ubiquitous B-tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–136, June 1979.

[2] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 509–516, Dec. 1992.

[3] S. Manegold, P. Boncz, and M. Kersten, "Optimizing database architecture for the new bottleneck: memory access." *VLDB J.*, vol. 9, no. 3, pp. 231–246, 2000.

[4] T. J. Lehman and M. J. Carey, "Query processing in main memory database management systems," in *Proceedings of ACM SIGMOD 1986 Annual Conference*. ACM Press, 1986, pp. 239–251.

[5] ——, "A study of index structures for main memory database management systems," in *Proceedings of the 12th VLDB Conference*, Kyoto, Aug. 1986, pp. 294–302.

[6] G. M. Adelson-Velskii and E. M. Landis, "An algorithm for organization of information," *Soviet Math. Doclady 3*, pp. 1259–1263, 1962.

[7] K.-R. Choi and K.-C. Kim, "T*-tree: a main memory database index structure for real time applications," in *RTCSA*, 1996, pp. 81–.

[8] G. Graefe and P. Larson, "B-tree indexes and cpu caches," in *ICDE*, 2001, pp. 349–359.

[9] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," in *VLDB*, 1999, pp. 78–89.

[10] ——, "Making B+-trees cache conscious in main memory," in *SIGMOD*, 2000, pp. 475–486.

[11] T. Chilimbi, M. Hill, and J. Larus, "Making pointer-based data structures cache conscious," *IEEE Computer*, vol. 33, no. 12, pp. 67–74, 2000.

[12] S. Manegold, P. Boncz, and M. Kersten, "Generic database cost models for hierarchical memory systems," in *VLDB*, 2002, pp. 191–202.

[13] J. Lindström, "T$_{link}$-tree: Main memory index structure," in *Proceedings of the 2nd International Advanced Database Conference (IADC-2006)*, San Diego, California, USA, 2006.

[14] J. Jaluta, S. Sippu, and E. Soisalon-Soininen, "Concurrency control and recovery for balanced b-link trees," *VLDB*, vol. 14, no. 2, pp. 257–277, 2005.

[15] T. Lehman, E. Shekita, and L. Cabrera, "An evaluation of Starburst's memory resident storage component," *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 6, pp. 555–566, 1992.

[16] H. V. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan, "Dalí: A high performance main memory storage manager," in *Proceedings of the 20th VLDB Conference*, 1994, pp. 48–59.

[17] P. Lehman and S. Yao, "Efficient locking for concurrent operations on B-trees," *ACM Transactions on Database Systems*, vol. 6, no. 4, pp. 650–670, Dec. 1981.

[18] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.