Real-Time Systems, 19, 245–282 (2000)

Real-Time Access Control and Reservation on B-Tree Indexed Data*

TEI-WEI KUO ktw@csie.ntu.edu.tw
Department of Computer Science and Information Engineering, National Taiwan University, #1 Roosevelt Rd., Sec. 4, Taipei, Taiwan 106, ROC

CHIH-HUNG WEI ywr84@cs.ccu.edu.tw
Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan 621, ROC

KAM-YIU LAM cskylam@cityu.edu.hk
Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong

Abstract. This paper proposes methodologies to control the access of B^+-tree-indexed data in a batch and firm real-time fashion. Algorithms are proposed to insert, query, delete, and rebalance B^+-tree-indexed data based on the non-real-time algorithms proposed in Kerttu, Eljas, and Tatu (1996) and the idea of priority inheritance (Sha, Rajkumar, and Lehoczky, 1990). We propose methodologies to reduce the number of disk I/O to improve the system performance without introducing more priority inversion. When the schedulability of requests with critical timing constraints is highly important, we propose a mechanism for data reservation based on the idea of preemption level and the Stack Resource Policy (Baker, 1990). The performance of our methodologies was evaluated by a series of experiments, from which we have obtained encouraging results.

Keywords: real-time data access, B-tree index, batch operations, data reservation, priority inversion.

1. Introduction

With the advances in communication and multimedia technologies, new database applications are emerging in recent years. Many of the new applications, such as internet programmed stock trading systems, tele-medicine systems, digital library, and computer-integrated manufacturing systems (Andresen et al., 1996; Choy and Morris, 1996; Grossman et al., 1995; Lam, Kuo, and Shu, 1998), require "real-time" access to different kinds of information such as stock data, traffic conditions, and patient status. Although the transactions in these applications may not have hard real-time deadlines, they usually have different degrees of real-time response time requirements. It is of paramount importance to have a predictable system performance to make such application systems useful. For example, in a programmed stock trading system, a late response to a trading query may result in a loss of a good trading opportunity (Lam, Kuo, and Shu, 1998). How to meet the response time constraints of transactions is becoming an important concern of these application systems. In addition to the deadlines associated with the transactions, another important characteristic of the applications is that they usually need to manage a large number of data items.

* A preliminary version of this paper appeared in the *Proceedings of IEEE 15th International Conference on Data Engineering, 1999.*

In the past decades, different index structures have been proposed for database systems (Bernstein, Hadzilacos, and Goodman, 1987; Elmasri and Navathe, 1994). Amongst the proposed index structures, B-tree is one of the most popular and efficient index structures. Mechanisms based on locking are usually used to control concurrent access of a B-tree index in order to maintain database consistency. Various B-tree access algorithms, e.g., (Bayer and Schkolnick, 1977; Lehman and Yao, 1981; Mohan and Levine, 1992; Mond and Raz, 1985), have been proposed with the objective to increase the concurrency in accessing the index and the data items so that the mean data access delay can be minimized. Although the B-tree index structure and the proposed algorithms have been shown to be very efficient for traditional database systems such as banking systems, airline reservation, and accounting systems, they have not been successfully applied to database systems with stringent response time requirements. As shown in many previous research in real-time database systems (RTDBS), e.g., Kuo (1994), Sha et al. (1991), a high concurrency of transaction execution may not guarantee the satisfaction of transaction deadlines or the minimization of deadline violations. B-tree algorithms which aim at maximizing the concurrency of transaction execution may not be able to guarantee the urgency of transaction deadlines. The impacts of priority inversion can seriously increase the chance of deadline violations as the blocking time of a transaction execution is often unbounded for the B-tree algorithms proposed for traditional database systems, where priority inversion is a situation in which a higher priority (or urgent) transaction is blocked by a lower priority (or less urgent) transaction. Priority cognitive access algorithms for B-tree indexed data are needed to manage the priority inversion problem and meet the urgency of transaction executions.

Although many researchers have proposed various real-time concurrency control protocols, e.g., (Abbott and Garcia-Molina, 1988; Bestavros, 1994; Dipippo and Wolfe, 1993; Kuo and Mok, 1993; Liang, Kuo, and Shu, 1997; Lin and Son, 1990; Peng and Lin, 1996; Sha et al., 1991; Xiong et al., 1996), most of them assume that a real-time database consists of data items or objects without any index structure. Many of proposed concurrency control protocols can only be applied to RTDBS consisting of a fixed set of data items or/and a fixed set of transactions, e.g., Kuo and Mok (1993), Liang, Kuo, and Shu (1997), Sha et al. (1991). As the demand of timing constraints is increasing in traditional (or modern) database systems in recent years, such as banking systems, digital library, and stock trading systems, many researchers have started to consider how to provide real-time performance to transactions in database systems which have stringent response time requirements. Such systems, e.g., digital library, stock trading systems, or banking systems, may have a tremendous number of disk-resident data objects/records and usually have indices to facilitate and speed up data accesses. The issue of real-time index concurrency control was first proposed and addressed in Goyal et al. (1995), and a performance evaluation of a variety of B-tree concurrency control protocols was made for single-key index operations. In this paper, we extend the prior work to consider real-time access control for batch (multi-key) operations. Batch algorithms based on non-real-time algorithms in Kerttu, Eljas, and Tatu (1996) and the idea of priority inheritance (Sha, Rajkumar, and Lehoczky, 1990) are proposed to significantly improve the performance and the schedulability of real-time disk-resident database systems. A relaxed B^+-tree structure (Kerttu, Eljas, and Tatu, 1996) is adopted to reduce the blocking time of urgent transactions because of more flexible rebalance conditions and

deferred local rebalance operations. To further reduce the number of disk I/O and improve the system performance, methodologies to share nodes among batch algorithms are proposed without introducing more priority inversion. When the response time of highly critical transactions must be considered, we propose a mechanism to reserve data based on the idea of preemption level and the Stack Resource Policy (Baker, 1990), where highly critical transactions are often more valuable and have urgent deadlines. The performance of our methodologies is evaluated by a series of simulation experiments, for which we have obtained very encouraging results.

The major contributions of this work are as follows: (1) A relaxed and less blocking B^+-tree index structure is explored for real-time and batch processing of user requests.(2) Real-time batch algorithms with data sharing mechanisms are proposed to improve the performance of a firm real-time disk-resident B^+-tree-indexed database without introducing more priority inversion. The value of a firm real-time transaction will be totally lost after its deadline and it has to be aborted immediately. (3) The idea of preemption level is explored for data reservation to improve or guarantee the performance of highly critical real-time transactions.

The rest of the paper is organized as follows: Section 2 reviews the previous work on the real-time concurrency control protocols and the B-tree algorithms. Section 3 defines the B^+-tree index structure, the transaction model, and important terminology. In Section 4, we propose basic insertion, deletion, query, and rebalance algorithms for firm real-time and batch processing of user requests. We also provide methodologies to reduce the number of disk I/O. Section 5 describes a data reservation and scheduling mechanism based on the idea of preemption level (Baker, 1990) for highly critical real-time transactions. Section 6 provides simulation results to demonstrate the capability of the proposed methodologies. Section 7 is the conclusion.

2. Related Work

2.1. Concurrency Control Algorithms for B-Tree Indexed Data

B-tree indexing and retrieval has been a preferred access method for large-scale commercial database systems. A number of highly efficient B-tree algorithms, e.g., (Bayer and Schkolnick, 1977; Lehman and Yao, 1981; Mohan and Levine, 1992; Mond and Raz, 1985), for data insertion, deletion, and query have been proposed. The algorithms can be classified as either top-down or bottom-up. Algorithms in the top-down class (e.g., Bayer and Schkolnick, 1977; Mond and Raz, 1985), traverse a B-tree from the root to the leaf using lock-coupling and different locking modes (such as exclusive, shared, and intention shared locks). In order to improve the concurrency, insert and delete operations often use weaker locks such as shared locks for index nodes. However, when splitting or merging of any node may be resulted, an insert or delete operation is restarted with a restrictive locking mode such as “shared and intention exclusive” locks.

Algorithms in the bottom-up class (e.g., Lehman and Yao, 1981), usually first traverse a B-tree from the root to the leaf using shared locks. Insertion and deletion algorithms need to lock the leaf node in an exclusive mode. When splitting or merging may be resulted,

insertion and deletion algorithms will traverse backward from the leaf to some proper node, e.g., the root, using exclusive locks to restructure the B-tree. Some algorithms (e.g., Lehman and Yao, 1981; Mohan and Levine, 1992), depend on relaxed B-tree structures to increase the concurrency. Performance study in Johnson and Shasha (1990), Srinvasan and Carey (1991) shows that the B-Link algorithms in Lehman and Yao (1981) have the best performance for single index operations, where a single index operation inserts, deletes, or queries one data item at a time. B-Link algorithms rely on a relaxed B-tree structure which uses links to chain all the nodes at each level. Insert (and delete) operations involving node splitting (and merging) are done in two phases. In other words, a newly split node is inserted into the appropriate parent after the insertion of the data. An appropriate entry deletion occurs later at the next higher level of the B-tree after the deletion of data.

2.2. Real-Time Concurrency Control for B-Tree Indexed Data

Although a number of researchers have proposed protocols for real-time access control of data items (Abbott and Garcia-Molina, 1988; Dipippo and Wolfe, 1993; Haritsa, Carey, and Livny, 1990; Kuo and Mok, 1992; Kuo and Mok, 1993; Liang, Kuo, and Shu, 1997; Lin and Son, 1990; Peng and Lin, 1996; Sha et al., 1991; Xiong, 1996), the only prior work that we are aware of on real-time access control is Goyal et al. (1995), Haritsa and Seshadri (2000). In these papers, the performance of various B-tree algorithms was studied. They show that the B-Link algorithms performed better than many other well-known B-tree algorithms in reducing the number of deadline violations, especially when a load control component was adopted. The load control component prevents new operations from entering the system whenever the utilization of some bottleneck resource, e.g., disks, exceeds some threshold. Although effective algorithms for single index operations have been proposed, it is always more profitable to insert, delete, or query data items in a batch if several keys are provided at a time. There are typical applications, such as digital library, full-text retrieval, and newspaper house application (Andresen et al., 1996; Choy and Morris, 1996; Grossman et al., 1995; Lam, Kuo, and Shu, 1998), in which batch operations are badly needed.

3. B-tree Index Structure and Transaction Model

3.1. B^+-Tree Index

Let a database consist of a collection of data objects indexed in B^+-tree structures. B^+-tree is a B-tree-based search tree, which is used to speed up the retrieval of records or data objects in response to various search conditions (Please see Figure 1). Each node in a B^+-tree of *order p*, except for a special node called the *root*, has one *parent* node and at most *p child* nodes. A node which has no child node is called a *leaf* node. A non-leaf node is called an *internal* node. Each internal node in a B^+-tree contains at most $p - 1$ search values and p pointers in the order

$$\langle P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q \rangle$$

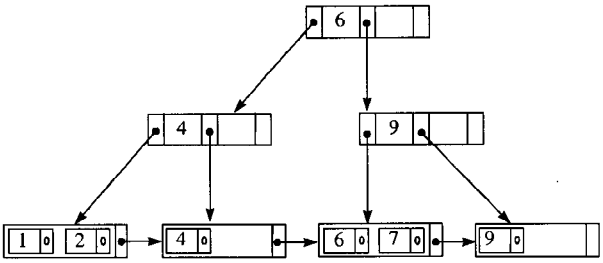


Figure 1. A B$^+$-tree of order 3.

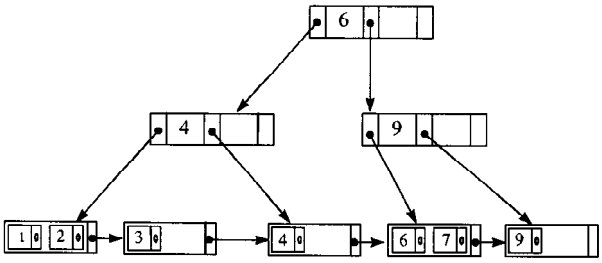


Figure 2. A relaxed B$^+$-tree.

where $q \leq p$. For all search field values X in the subtree pointed by P_i, we have $K_{i-1} \leq X < K_i$ for $1 < i < q$, $X < K_i$ for $i = 1$, and $K_{i-1} \leq X$ for $i = q$ (Elmasri and Navathe, 1994). Each leaf node is of the form

$$\langle\langle K_1, Ptr_1\rangle, \langle K_2, Ptr_2\rangle, \ldots, \langle K_{q-1}, Ptr_{q-1}\rangle, P_{next}\rangle\rangle$$

where $K_1 < K_2 < \cdots < K_{q-1}$. Each ptr_i is a data pointer which points to the record or data object whose search field value is K_i (Elmasri and Navathe, 1994). A leaf or internal node is said to have k empty slots when $k = p - q$. A node is of *level i* if there are i internal nodes (excluding the node itself) between the root and the node. The height of a B$^+$-tree is the maximum level of nodes in the tree plus one. For example, the root and the leaf nodes in Figure 1 are of levels 0 and 2, respectively. The height of the tree is 3. For the purpose of this paper, we use the terms "record" and "data object" interchangeably.

In this paper, we aim at maximizing the system concurrency and reducing the blocking time of transaction executions. A relaxed B$^+$-tree (Nurmi, Soisalon-Soininen, and Wood, 1987) is adopted, as shown in Figure 2. A relaxed B$^+$-tree is a B$^+$-tree in which some balance conditions of a B$^+$-tree are relaxed so that the tree can later be easily rebalanced with local operations.

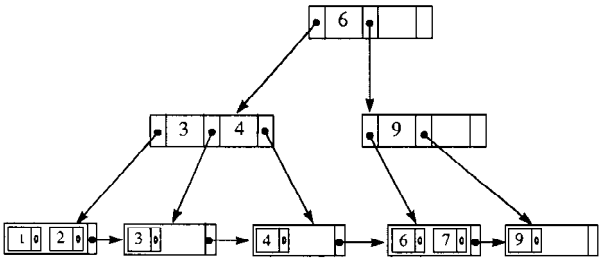


Figure 3. A rebalanced relaxed B^+-tree.

Insertions may generate overflow temporary nodes which must be balanced later. An overflow temporary node is a node at which no parent node is directly pointing to. For example, the insertion of data with key 3 in the B^+-tree shown in Figure 1 causes the overflow of the leaf node which contains data pointers for data with keys 1 and 2. The insertion overflow results in the creation of an overflow temporary node which contains a data pointer for data with key 3, as shown in Figure 2. The tree will be rebalanced later by modifying the proper internal nodes and become a conventional B^+-tree, as shown in Figure 3. The search of a key is carried out exactly in the same way as in conventional B^+-trees except that searching over overflow temporary nodes might be needed. For example, the search of data with key 3 in the B^+-tree shown in Figure 2 will first reach the leaf node which contains data pointers for keys 1 and 2 and, then, go right on a sibling link to the overflow temporary node which contains a data pointer for key 3. We refer interested readers to Nurmi, Soisalon-Soininen, and Wood (1987) for details. For the rest of this paper, our usage of the term "B-tree" refers to the "relaxed B^+-tree" variant.

3.2. Transaction Model

We are interested in real-time access control of a database system, such as stock trading system or news server, which contains a large amount of disk-resident data indexed in a B^+-tree structure. The system contains a dynamic workload which consists of a large number of query and insert transactions. An insert transaction is responsible for inserting a collection of data objects into the database. A query transaction retrieves data objects from the database according to the key values provided by users. Transactions are assumed to be generated by users in terms of simple interfaces, such as home pages via a web-based browser. For example, a stock broker can look up the face values of some stocks by supplying codes to the system. A student may look up the scores of his/her courses taken in this semester via a browser, and the personnel of the registration office can insert records of course scores for students. Transactions in the above examples may need different degrees of response time requirements. Even for a student's inquiry on his/her course scores, it should not take a long time. Otherwise, the student will either quit from the inquiry, or complain to the registration office or the computation center. With the motivations of the

above example systems, we propose to define a transaction as a simple collection of keys to have good flexibility in proposing effective data access control protocols.

A query transaction Q_i is formally defined as a tuple of key values

$$(key_1, key_2, \ldots, key_{n_i}),$$

where each key value is for index search of a specific data object interested to a user, and $key_j \neq key_k$ for $j \neq k$. The results of a query transaction is a tuple

$$(data_1, data_2, \ldots, data_i),$$

where $data_j$ is the image of the data object with a key value equal to key_j. If there is no data object with a key value equal to key_j, then $data_j$ is *null*.

An insert transaction I_i is formally defined as a tuple of pairs

$$((key_1, data_1), (key_2, data_2), \ldots, (key_{n_i}, data_{n_i})),$$

where each $data_j$ is associated with a key key_i for insertion into a B^+-tree index structure, and $key_j \neq key_k$ for $j \neq k$. The execution of an insert transaction not only causes the inclusion of the data objects into the database but also results in correct modification of the B^+-tree index structure. A delete transaction is a tuple of key values

$$(key_1, key_2, \ldots, key_{n_i}),$$

where each key value key_j is for deleting a specific data object with the key value, and $key_j \neq key_k$ for $j \neq k$. The execution of a delete transaction may not only cause the removal of the data objects from the database but also result in the modification of the B^+-tree index structure.

For the purpose of this paper, only real-time access control on a single B^+-tree index structure is considered. The access control proposed in this section can be applied to multiple B^+-tree index structure straightforwardly.

3.3. Definitions

For the purpose of this paper, we shall first define the following terminology:

Definition 1 (Overflow Node). An *overflow node* is a node at which no parent node is directly pointing to, and it is not the root. A *non-overflowed* node is not an overflow node.

An overflow node can be a leaf node, e.g., the node which contains a data pointer for data with key 3 in the B^+-tree shown in Figure 2, or an internal node.

Definition 2 (Sibling Node). A node is the *right sibling node* of another node if the latter node has a link, e.g., p_{next}, directly pointing at the former. The latter is called the *left sibling node* of the former. These two nodes are *sibling nodes* of each other.

Definition 3 (Control Set). The *control set* of a non-overflowed node $node_i$ is the set of nodes $CS(node_i) = \{node_j\}$ in which

1. The only non-overflowed node in $CS(node_i)$ is $node_i$.

2. For any overflow node $node'$ in $CS(node_i)$, there must exist a sequence of nodes $node_{n_1}$, $node_{n_2}, \ldots, node_{n_m}$ in $CS(node_i)$ such that $node_{n_{k+1}}$ is the right sibling node of $node_{n_k}$ for $1 \leq k < m$, and $node_{n_1}$ and $node_{n_m}$ are $node_i$ and $node'$, respectively.

Note that, when the size of a control set is one, the control set only contains a non-overflowed node. A control set consists of a non-overflowed node and zero or more overflow nodes.

Definition 4 (Ancestor Node). A node $node_i$ is an *ancestor node* of another node $node_j$ in a B^+-tree if there exists a sequence of nodes $node'_1, \ldots, node'_n$ such that $node_i$ is either the parent node of $node_{i+1}$ or the left sibling node of $node_{i+1}$ for $i < n$, and the level of $node_i$ is lower than the level of $node_j$, where $node_i$ and $node_j$ are $node'_1$ and $node'_n$, respectively.

The definition of a control set is to simplify the lock management of nodes. Under the algorithms proposed in the following sections, a transaction can lock all the nodes in the control set of a non-overflowed node by locking the non-overflowed node. We define the key set of a control set as the set of keys which can possibly appear in the leaf nodes of the sub-trees pointed by the nodes in the control set. When a set of keys to be accessed by a transaction overlaps with the key set of a control set, the transaction should lock the control set.

Definition 5 (Key Set of a Control Set). The *key set* $KS(CS_i)$ of the control set CS_i of a non-overflowed node $node_i$ is defined as follows:

- If $node_i$ is the root, then $KS(CS_i)$ equals to the universe of the keys.
- If $node_i$ has a parent node $node_j = \langle P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_q \rangle$, and P_k points to $node_i$, then $KS(CS_i)$ is defined as follows:
  - If $1 < k < q$, then $KS(CS_i)$ equals to the set of keys between K_{k-1} and K_k including K_{k-1} but excluding K_k.
  - Suppose $k = 1$. Let $node'_j = \langle P'_1, K'_1, P'_2, K'_2, \ldots, P'_{q'-1}, K'_{q'-1}, P'_{q'} \rangle$ be the closest ancestor node of $node_i$ which satisfies either one of the following conditions first:
    1. $node'_j$ has a pointer P'_x pointing at a subtree which contains $node_i$ and $x > 1$.
    2. $node'_j$ is an overflow node and has a pointer P'_x pointing at a subtree which contains $node_i$ and $x = 1$.

    If Condition 1 is satisfied first, then $KS(CS_i)$ equals to the set of keys between K'_{x-1} and K_1 including K'_{x-1} and excluding K_1. If Condition 2 is satisfied first, then $KS(CS_i)$ equals to the set of keys between $K"_{q"-1}$ and K_1 including $K"_{q"-1}$ and excluding K_1, where $node_y = \langle P"_1, K"_1, P"_2, K"_2, \ldots, P"_{q"-1}, K"_{q"-1}, P"_{next} \rangle$ is the left sibling node of $node'_j$, and $P"_{next}$ is a node pointer pointing to the right sibling node, if it exists, or a null pointer. If there does not exist such ancestor node satisfying either of the conditions, then $KS(CS_i)$ equals to the set of keys less than K_1.

– If $k = q$, and $node'_j = \langle P'_1, K'_1, P'_2, K'_2, \ldots, P'_{q'-1}, K'_{q'-1}, P'_{q'} \rangle$ is the closest ancestor node of $node_i$ which has a pointer P'_x pointing at a subtree which contains $node_i$ and $x < q'$, then $KS(CS_i)$ equals to the set of keys between K_{q-1} and K'_x including K_{q-1} and excluding K'_x. If there does not exist such ancestor node, then $KS(CS_i)$ equals to the set of keys no less than K_{q-1}.

The key set $KS(CS_i)$ of a control set CS_i is exclusively partitioned into the key sets of the nodes in the control set $KS(CS_i)$ according to definitions of relaxed B^+-tree nodes (Nurmi, Soisalon-Soininen, and Wood, 1987). For example, let $node_i = \langle P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_{next} \rangle$ be a non-overflowed node and has a right sibling node in the control set CS_i. The key set of $node_i$ is the set of keys in the key set $KS(CS_i)$. If $node_i = \langle P_1, K_1, P_2, K_2, \ldots, P_{q-1}, K_{q-1}, P_{next} \rangle$ has a left sibling node $node_j = \langle P'_1, K'_1, P'_2, K'_2, \ldots, P'_{q'-1}, K'_{q'-1}, P'_{next} \rangle$ and a right sibling node, then the key set of $node_i$ is the set of keys in the key set $KS(CS_i)$ which is no less than $K'_{q'-1}$ (including $K'_{q'-1}$). If the size of the control set $KS(CS_i)$ is one, then the key set of the only node in the control set CS_i equals to the key set $KS(CS_i)$.

4. Real-Time Batch Algorithms

4.1. Overview

The purpose of this section is to first propose basic insertion, query, deletion, and rebalancing algorithms for batch firm real-time access of B^+-tree-indexed data. We shall then further improve the algorithms in reducing the number of disk I/O since disk I/O is notoriously slower than main memory data access. Mechanisms will be proposed to share data and locks among transactions. In Section 5, additional scheduling disciplines will be proposed to guarantee the schedulability of highly critical real-time transactions.

The basic insertion, query, deletion, and rebalancing algorithms are based on the non-real-time algorithms in Kerttu, Eljas, and Tatu (1996). A breadth-first-search (BFS) traversal mechanism with priority inheritance (Sha, Rajkumar, and Lehoczky, 1990) is adopted to improve the system performance and manage the priority inversion problem. With priority inheritance, if there is a lock conflict, and the priority of the lock-holding transaction is lower than the priority of the lock-requesting transaction, then the priority of the lock-holding transaction will be raised up to that of the lock-requesting transaction. The lock-requesting transaction will be blocked until the lock-holding transaction releases the lock. We adopt the B^+-tree structure in Kerttu, Eljas, and Tatu (1996) except that each non-overflowed child node has a pointer pointing back to the parent node for the efficiency of the proposed algorithms. No splitting of any node is needed unless it is overflowed, and no merging of any node will occur unless the node is empty.

Let the system be associated with a queue *Work-Queue* of ready transactions sorted in an increasing order of their absolute deadlines, where each transaction is running the query, insertion, deletion, or rebalance algorithms proposed in the following sections. The transaction at the beginning of the *Work-Queue* is assigned the processor. The earliest deadline first (EDF) scheduling algorithm (Liu and Layland, 1973) is adopted to schedule

transactions for execution. There are two kinds of locks in the system: read and write locks. A transaction is required to read-lock (or write-lock) a node before it initiates any disk I/O to read the node from (or update the node into) the disk. Read locks are compatible to each other. Locks of nodes are managed by the lock manager of the system. After a transaction initiates a disk I/O, and before the disk I/O completes, the transaction is considered as *blocked* and will release the processor. Other ready transactions in the *Work-Queue* may compete for the processor. When the disk I/O request of a transaction is satisfied, the transaction is inserted into the *Work-Queue*.

It is assumed that disk I/O requests are serviced in a non-preemptive priority-ordered fashion. Theorems regarding priority inversion will not count the number of priority inversions due to disk I/O. (The number of priority inversions for each I/O request is one in this paper if disk I/O requests are serviced in a non-preemptive priority-ordered fashion. In other words, the number of priority inversions for an entire transaction for disk I/O requests can be as large as the number of disk accesses. Such a situation exists when the I/O system is very busy.)

4.2. Basic Protocols

4.2.1. Batch-Query Algorithms

When a query request $(key_1, key_2, \ldots, key_{n_i})$ with deadline d_i arrives in the system, a query transaction τ_i^q, which will execute Algorithm $Batch-Query((key_1, key_2, \ldots, key_{n_i}), ROOT, d_i)$, is inserted into the *Work-Queue*. τ_i^q will be assigned to use the processor if it is the ready transaction with the highest priority (in the *Work-Queue*).

```
Procedure Batch-Query(key_set, node, deadline)
    Input: key_set = (key_1, key_2, ..., key_{n_i})
        node: starting node in the tree.
    Begin
        node_list = φ;
        R-Lock(node);
        EnQueue(node_list, (key_set, node));
        While node_list is not empty
            request = DeQueue(node_list); /* request = (key_set, node) */
            Read_Node(request.node); /* disk I/O for a tree node */
            If request.node is a leaf node
              Then
                  For each c-node_i in the control set of request.node Do
                      Read_Data(request.key_set, c-node_i); /* disk I/O for data */
              Else
                  For each c-node_j in the control set of request.node (from left to right) Do
                      Child_Set = {node_i | node_i is a child node of c-node_j
                          which includes a subset key_set_i of keys in request.key_set};
                      For each child_node_i in Child_Set (from left to right) Do
```

```
                    R-Lock(child_node_i);
                    EnQueue(node_list, (key_set_i, child_node_i));
                EndFor;
            EndDo;
        EndIf;
        Unlock(request.node);
    EndWhile;
End
```

Algorithm *Batch-Query* is generalized from the query algorithm in (Kerttu, Eljas, and Tatu, 1996) by considering batch retrievals of data. Algorithm *Batch-Query* starts searching data with keys in *key_set* from node *node* which is usually the root of the B^+-tree in the basic protocols. The algorithm traverses the B^+-tree and read-locks nodes in a breadth-first search (BFS) fashion starting from *node*. In the beginning of the algorithm, *node* is inserted into a first-in-first-out BFS queue. During each iteration, a node *node* is popped out of the queue. If *node* is a leaf node, then *node* is read from the disk to retrieve the needed data. If *node* is an internal node, then all child nodes of the nodes in the control set of *node* are locked and inserted into the first-in-first-out BFS queue for node locking and accessing in the subsequent iterations. Note that each node must be read-locked before it is read from the disk. Algorithm *Batch-Query* only read-locks non-overflowed nodes. When Algorithm *Batch-Query* read-locks a non-overflowed node, all overflow nodes in the control set of the non-overflowed node are implicitly read-locked. Algorithm *Batch-Query* unlocks a node when all of its child nodes are read-locked. Note that the BFS traversal style may result in the blocking of a higher-priority transaction by an entire lower-priority transaction. The deployment of the priority inheritance scheme is necessary to reduce the entire blocking time of a transaction as other middle-priority transactions may arrive. Suppose that the priority inheritance scheme is not adopted, then many middle-priority transactions may arrive and preempt the execution of the lower-priority transaction which blocks the higher-priority transaction. Such arrivals of middle-priority transactions can be unlimited. In other words, a higher-priority transaction may be blocked by an unbounded number of priority inversions. It will seriously damage the response time of a real-time system.

As astute readers may notice, Algorithm *Batch-Query* can be slightly modified to support range queries. The *key_set* of a query transaction must be sorted and revised as a set of key ranges, and the algorithm should look for the overlapping of key_set_i and the keys of each node in the B^+-tree. The complexity and properties of the algorithm will remain.

4.2.2. Batch-Insert Algorithms

When an insertion request $\tau_i^I = ((key_1, data_1), (key_2, data_2), \ldots, (key_{n_i}, data_{n_i}))$ with deadline d_i arrives in the system, an insert transaction τ_i^I which will execute Algorithm $Batch-Insert(((key_1, data_1), (key_2, data_2), \ldots, (key_{n_i}, data_{n_i})), ROOT, d_i)$ is inserted into the priority-sorted *Work-Queue* of the system. τ_i^I will be assigned the processor if it is the highest-priority ready transaction (in the *Work-Queue*).

Algorithm *Batch-Insert* traverses the B^+-tree index and locks nodes in a BFS fashion starting from *node*. During each iteration, a node *node* is popped out of the BFS queue, and its child nodes are visited. Different from Algorithm *Batch-Query*, nodes of the highest two levels, including some leaf nodes, are write-locked because data may be inserted into leaf nodes. Algorithm *Insert-Data* is called whenever some data is inserted into a leaf node. If any overflow node is created because of the insertion, then a rebalance transaction with a proper deadline must be inserted into the *Work-Queue*. The batch-delete algorithm is similar to the batch-insert algorithm except that data are deleted. The algorithm is included in Appendix B.

```
Procedure Batch-Insert(key&data_set, node, deadline)
    Input: key&data_set = ((key_1, data_1), (key_2, data_2), ..., (key_{n_i}, data_{n_i}))
        node: starting node in the tree.
    Begin
        If node is a leaf node
          Then
              W-Lock(node);
              Inset-Data(key&data_set, node, deadline);
              Return;
          Else IF node is the parent node of a leaf node
              W-Lock(node);
          Else
              R-Lock(node);
          EndIF;
        node_list = φ;
        EnQueue(node_list, (key&data_set, node));
        While node_list is not empty
          request = DeQueue(node_list); /* request = (key_set, node) */
          Read_Node(request.node); /* disk I/O for a tree node */
          For each c-node_j in the control set of request.node Do
              Child_Set = {node_i | node_i is a child node of c-node_j
                  which includes some keys in request.key&data_set};
              Let key&data_set_i be a subset of request.key&data_set
                  whose keys are in node_i;
          EndDo;
          If request.node is the parent node of a leaf node
              Then
                  For each child_node_i in Child_Set (from left to right) Do
                      W-Lock(child_node_i); /* child_node_i is a leaf node. */
                      Inset-Data(key&data_set_i, child_node_i, deadline);
                      Unlock(child_node_i);
                  EndFor;
              Else
                  If request.node is the grand parent node of a leaf node
                      Then
```

```
                    For each child_node_i in Child_Set (from left to right) Do
                        W-Lock(child_node_i);
                        EnQueue(node_list, (key&data_set_i, child_node_i));
                    EndFor;
                Else
                    For each child_node_i in Child_Set (from left to right) Do
                        R-Lock(child_node_i);
                        EnQueue(node_list, (key&data_set_i, child_node_i));
                    EndFor;
                EndIf;
            EndIf;
        Unlock(request.node);
    EndWhile;
End
```

```
Procedure Insert-Data(key&data_set, leaf_node, deadline)
    Input: key&data_set = ((key_1, data_1), (key_2, data_2), ..., (key_{n_i}, data_{n_i}))
        leaf_node: a w-locked leaf node.
    Begin
        If the size of the control set of leaf_node is larger than one
          Then
            Insert data and keys in key&data_set to the proper node in the control set of leaf_node;
            Split any overflowed nodes and link them together;
                /* All data are sorted, and all nodes are linked
                according to the definitions of B+-tree control set */
          Else
            Insert data and keys to leaf_node;
            If leaf_node needs splitting
                Then
                    Split leaf_node and link them together;
                        /* All data are sorted, and all nodes are linked
                        according to the definitions of B+-tree control set */
                        parent_node = the parent node of leaf_node;
                        Add-Work-Queue(Job(Rebalance,parent_node, leaf_node, deadline));
                        /* A new request is inserted! */
                EndIf;
          EndIf;
        Unlock(leaf_node);
    End
```

4.2.3. Rebalance Algorithms

Due to insertion and deletion of data, overflow nodes may be created, and some nodes may become empty. The existence of overflow and empty nodes incurs extra overheads in the traversal of the B^+-tree. Although delaying the rebalancing of tree is allowed in a relaxed B^+-tree (Nurmi, Soisalon-Soininen, and Wood, 1987), the deterioration of B^+-tree, because of the excessive number of overflow nodes and empty nodes, may offset the advantages of the relaxed B^+-tree in reducing the number of blocking of transactions. This section is meant to describe the rebalance algorithm.

The rebalance algorithm, called Algorithm *Rebalance*, is the same as the non-real-time rebalance algorithm in Kerttu, Eljas, and Tatu (1996). As required in Kerttu, Eljas, and Tatu (1996), the system only allows one rebalance transaction executing at a time. In other words, the execution of a rebalance transaction will block the executions of other rebalance transactions. The creation of overflow nodes by batch-insert transactions and empty nodes by batch-delete transactions will insert a proper rebalance transaction in the *Work-Queue*. Each rebalance transaction is assigned a deadline, which is, in general, very loose. A rebalance transaction only targets at rebalancing a node and its child nodes' control set. If further rebalancing is needed at a higher level, another rebalance transaction is inserted in the *Work-Queue* again.

Please see Appendix A for detailed definitions of the rebalance algorithm.

4.2.4. Properties

THEOREM 1 *The system is deadlock-free.*

Proof: Since all transactions lock nodes in the B^+-tree are from the root to the leaves and from the left to the right, the system is deadlock-free. ■

THEOREM 2 *The maximum number of priority inversion for a higher-priority batch-query transaction is not larger than the number of transactions which have write-locked some nodes.*

Proof: Since a batch-query transaction only read-locks nodes in the B^+-tree, and write-locks are incompatible with any write-locks or read-locks, a higher-priority batch-query transaction can only be blocked by transactions which have write-locked some nodes. ■

Note that the number of transactions which have write-locked any B^+-tree node is limited by one plus the number of nodes which point to leaf nodes, where one stands for the write-lock of a rebalance transaction.

THEOREM 3 *The maximum number of priority inversion for a higher-priority batch-insert (or batch-delete) transaction is no larger than the sum of the number of transactions which have write-locked some nodes and the number of lower-priority query transactions which have read-locked any nodes of the highest two levels.*

Proof: The correctness of this theorem follows directly from the fact that a batch-insert (or batch-delete) transaction may write-lock nodes of the highest two levels and read-lock the other nodes. In other words, a batch-insert (or batch-delete) transaction can only be blocked by transactions which have locked any nodes of the highest two levels or write-locked any nodes. ■

THEOREM 4 *The maximum number of priority inversion for a higher-priority rebalance transaction is no larger than the number of active lower-priority non-rebalance transactions plus one.*

Proof: The correctness of this theorem follows directly from the following fact: A rebalance transaction must write-lock a parent node and a child node (to implicitly write-lock its control set) to restructure pointers and keys between the parent node and the nodes in the control set of the child node. Since write-locks are incompatible to any read-locks and write-locks, and only one rebalance transaction execution is allowed at a time, a rebalance transaction may be blocked by any active lower-priority non-rebalance transaction (which may either read-lock or write-lock some nodes) and one rebalance transaction. ■

4.2.5. Remark

As astute readers may point out, since each non-overflowed child node has a pointer pointing back to the parent node, an action related to node splitting or merging may result in the parent pointers of a large number of nodes to be updated if the tree has a large fan-out. Such updating can seriously damage the performance of the system. However, the proposed algorithms can be slightly modified to handle the situation where no node in the B^+-tree is associated with a parent node (Obviously, the definitions of overflowed and non-overflowed nodes will need to be changed accordingly). The modifications of the algorithms are as follows:

- No change is necessary for Algorithm Batch-Query because the traversal of the tree is from the root to the leaves.

- Supply the parent node of the leaf node in each invocation of Procedure Insert-Data in Algorithm Batch-Insert. In other words, replace Insert-Data(*key*&*data_set*$_i$, *child_node*$_i$, deadline) with Insert-Data(*key*&*data_set*$_i$, *child_node*$_i$, *request.node*, deadline). Then remove the statement "parent-node = the parent node of a leaf node" from Procedure Insert-Data because the parent node of a leaf node is provided in the invocation.

- Supply the parent node of the leaf node in each invocation of Procedure Delete-Data in Algorithm Batch-Delete. In other words, replace Delete-Data(*key*&*data_set*$_i$, *child_node*$_i$, deadline) with Delete-Data(*key*&*data_set*$_i$, *child_node*$_i$, *request.node*, deadline). Then remove the statement "parent-node = the parent node of a leaf node" from Procedure Delete-Data because the parent node of a leaf node is provided in the invocation.

- Insert new statements at the very beginning of Algorithm Rebalance to search for the parent node of parent_node (starting from the root). Note that it must be done before Algorithm Rebalance write-locks *parent_node* and *child_node* to prevent the existence of any deadlock. Note that in the searching of the grand parent node, Algorithm Rebalance should not unlock a parent node before it locks the proper child node until it finds the grand parent node. We should also remove the following statement in Algorithm Rebalance because it does not need to exist any more:

  "grand_parent_node = the parent node of parent_node"

4.3. Starting-Search-Node Transformation and Node Sharing

The purpose of this section is to further improve the performance of the basic protocols in reducing the number of disk I/O to meet the timing constraints of transactions since disk is the slowest component in a computer system. Slight saving in disk I/O may boost the performance of the system significantly. Two major methodologies are proposed: (1) The beginning node to start the processing of a transaction may be moved from the root of the B^+-tree to some node of a higher level. (2) The contents of a node retrieved by a transaction from the disk can be cached for other transactions. No modification of any basic protocols proposed in the previous sections will be needed.

4.3.1. Starting-Search-Node Transformation

The basic protocols proposed in the previous sections start the processing of each transaction from the root of the B^+-tree to the leaf nodes which point to the data needed by the transaction. The number of disk I/O for accessing each data is proportional to the height of the tree, the location and number of overflow nodes. As astute readers may notice, many transactions tend to repeatedly lock and read the same nodes, e.g., the root and lower-level nodes. Since lower-level nodes are usually read-locked by transactions and may be needed by some other transactions as well, the system performance can be largely improved if the nodes retrieved by other transactions can be used to start the processing of another transaction from some node of a higher level instead of the root.

When a transaction is initiated, it is inserted into the *Work-Queue* with the starting search node as the root, e.g., *Batch-Query*(*key_set*, *root*, *deadline*) or *Batch-Insert*(*key*&*data_set*, *root*, d_i). Suppose that transaction τ read-locks $node_i$ and retrieves $node_i$ from the disk. If the system finds out that $node_i$ contains the set of keys needed by some other transaction τ' in the *Work-Queue*, e.g., the *key_set* in *Batch-Query*(*key_set*, $node_j$, *deadline*) or the *key*&*data_set* in *Batch-Insertion*(*key*&*data_set*, $node_j$, d_i), and the level of $node_i$ is higher than the level of the current starting search node $node_j$ of transaction τ', then the starting search node of transaction τ' is replaced with $node_i$. For example, if transaction $\tau' =$ *Batch-Query*(*key_set*, $node_j$, *deadline*) in the *Work-Queue* satisfies the above conditions, then the system will replace it with *Batch-Query*(*key_set*, $node_i$, *deadline*). In addition, the system needs to read-lock $node_i$ and unlock $node_j$ on behalf of transaction τ' and without notifying

τ'. Note that the transaction does not know if the system has locked any node for it. When transaction τ' later requests a read-lock on $node_i$, the system grants it immediately.

The reason that the system must read-lock the starting search node for a transaction is to protect the transaction from any structure or node-content modification done by future rebalance or delete transactions. However, such read-locks may increase the number of priority inversion for rebalance or delete transactions. One novel solution is to reset the starting search node of a transaction back to the root if any higher-priority transaction requests a conflicting lock, i.e., write-lock, in the future, on the starting search node of the former transaction.

We should point out that the most urgent transactions should be either query, insert, or delete transactions instead of rebalance transactions, where rebalance transactions are for restructuring the B^+-tree. No transactions except rebalance transactions will write-lock any nodes at a level lower than the highest two levels, including the leaf nodes, of the B^+-tree. We surmise that the modifications of the starting search node for a transaction may improve the system performance significantly without increasing the number of priority inversion.

4.3.2. Node Sharing

This section is meant to further generalize the idea of batch access of B^+-tree-indexed data. We shall provide a mechanism for transactions to share nodes retrieved by each another from the disks such that the decrease in transaction execution duration (because of less disk I/O) may improve the response time of transactions.

Let a transaction τ lock a (leaf or internal) node and read it from the disk at some time point. If the set of keys accessed by some other transaction τ' overlaps with the set of keys contained in the node, and τ and τ' will not access the node in any conflicting mode, then the system may lock the node and cache the contents of the node on behalf of transaction τ'. Note that transaction τ' does not know that the system has locked any node for it. When transaction τ' requests a lock on $node_i$ and tries to read it from the disk, the system grants the lock and returns the node immediately. The compatibility of locks issued or will be issued by any two transactions can be determined easily on the type of transactions and the level of a shared node.

Note that pre-locking of nodes by the system may also increase the number of priority inversion for urgent transactions, e.g., their deadlines are approaching. One novel solution is that the system unlocks and removes the cache for a node if some higher-priority transactions want to lock the node in any conflicting mode. Depending on the level of cached nodes and the ratio of query transactions in the system, the performance of this approach varies. In Section 6, we shall demonstrate the benefits of this approach in reducing the number of deadline violations in the system.

4.3.3. Properties

THEOREM 5 *The system with node sharing and starting-search-node transformations is deadlock-free.*

Proof: Since the read-locks of nodes for node sharing and starting-search-node transformations will be released by the system, when conflicting lock requests occur, the correctness of this theorem follows directly from Theorem 1. ■

THEOREM 6 *The system with node sharing and starting-search-node transformations will not increase the number of priority inversion for any transaction.*

Proof: The correctness of this theorem follows directly from the fact that the read-locks of nodes for node sharing starting-search-node transformations will be released by the system, when conflicting lock requests occur. ■

5. Data Reservation

5.1. Overview

Although the previous sections propose general algorithms for batch insertion, query, deletion, and rebalancing of B^+-tree-indexed data, the worst-case number of priority inversions for a real-time transaction can be significant. Such a large number of priority inversions is, in general, not acceptable to real-time transactions with highly critical timing constraints. Note that the maximum number of priority inversions reflects the worst-case blocking time and the response time for a transaction. The goal of this section is to further explore the access control of B^+-tree-indexed data to significantly reduce the maximum number of priority inversions for highly critical real-time transactions.

Although highly critical real-time transactions may not be necessary hard real-time transactions, it is usually assumed that their arrival patterns and data requirements are known in a priori. We shall explore the idea of data reservation to strictly bound the maximum number of priority inversion for highly critical real-time transactions to guarantee their schedulability if necessary.

Let $T = \{\tau_1, \ldots, \tau_n\}$ be a set of highly critical real-time transactions (or simply called highly critical transaction) in the system, and the set of keys which may be accessed by each of the transactions in T and their timing constraints be known in a priori. (Engineers may need the worst-case assumption on the key set of each transaction.) Regardless of whether a transaction τ_i in T is a periodic or sporadic transaction, the slack of τ_i is defined as the relative deadline of τ_i, i.e., the difference of the absolute deadline and the request time. We assume that the maximum slack of highly critical transactions in T is less than the slack of any other transactions in the system which are not in T. The transactions which are not belonging to T are comparatively less critical, called less critical real-time transactions (or simply called less critical transactions), and their arrival patterns, timing constraints, and data set are unknown before their arrivals.

The basic mechanism in controlling the number of priority inversions for highly critical transactions is to use transaction slacks to manage the locks of B^+-tree nodes. The ideas of preemption level and the stack resource policy (SRP) (Baker, 1990) are adopted to define the minimum slack threshold for a transaction to lock a node (the minimum slack threshold is defined in terms of preemption level in the next section). As required by the basic proto-

cols defined in Section 4, transactions are scheduled using the EDF scheduling algorithm. However, before a transaction starts execution, the slack of the transaction must be higher than the minimum slack threshold, i.e., the preemption level defined in the next section, of any nodes locked by other transactions. Note that transactions are basically scheduled according to the EDF scheduling algorithm, and preemption levels are used to manage preemption to bound the number of priority inversions for highly critical transactions.

5.2. Terminology

Before proceeding with further discussion, we shall first define the important terminology based on definitions in Baker (1990).

Definition 6 (Preemption Level). The *preemption level* $\pi(\tau)$ of a transaction τ equals to a positive real number $1/d_i$, where d_i is the relative deadline of τ.

Since only non-overflowed nodes are locked in the algorithms, the node ceiling of a non-overflowed node is defined as follows. Note that, when a non-overflowed node is locked, every overflow node in the control set of the non-overflowed node, if it exists, is implicitly locked.

Definition 7 (Node Ceiling). The *node ceiling* $\pi(node_i)$ of a (non-overflowed) B^+-tree node $node_i$ with respect to a set T of highly critical transactions equals to the maximum of the preemption levels of transactions in T whose key set overlaps with the key set of the control set of $node_i$.

If the key set of the control set of a non-overflowed node does not overlap with the key set of any transaction in T, then the node ceiling of the node is an arbitrary negative real number. Note that the node ceiling of a node changes dynamically as the structure of the B^+-tree changes.

5.3. Batch Protocols with Transaction Preemption Levels

The scheduling of transactions (which execute batch-insertion, batch-deletion, batch-query, batch-update, and rebalance algorithms in Section 4) is according to the stack resource policy (SRP) (Baker, 1990), where EDF scheduling algorithm is adopted. The only difference between the assumptions of the original stack resource policy and the stack resource policy adopted in this paper is that the set of B^+-tree nodes, i.e., resources in Baker (1990), and their ceilings may change dynamically. The SRP (Baker, 1990) is re-phrased as follows:

Definition 8 (The Data-Reservation Scheduling Mechanism). The scheduler requires that a transaction τ be blocked from starting execution until τ is the highest-priority ready transaction in the system and the preemption level of τ is higher than the maximum node ceiling of the nodes locked by other transactions.

Obviously, once a transaction τ in T has started execution, all subsequent lock requests on nodes by τ will be granted immediately, without blocking from lower-priority (or equal-priority) transactions. Note that it is not possible for a higher-priority transaction which is

not belonging to T arrives later than τ because the assumption of this paper requires that the maximum slack of the highly critical transactions in T is less than the slack of any other transactions in the system which are not in T. However, a higher-priority transaction in T may arrive later (than τ does) and preempt the execution of τ because the scheduling mechanism does not allocate nodes to a transaction before the lock requests of the transaction.

5.4. Properties

THEOREM 7 *The system with the data reservation mechanism is deadlock-free.*

Proof: There should be no deadlock involving both highly critical transactions and less critical transactions. It is because once a highly critical transaction starts execution, it will not be blocked by any less critical transaction, regardless of whether the less critical transaction has a priority higher or lower than the priority of the highly critical transaction. It is because the assumption of the scheduling mechanism with data reservation requires that the maximum slack (which is inversely proportional to the preemption level) of the highly critical transactions is less than the slack of any other less critical transactions in the system.

Theorems 1 and 5 show that there is no deadlock involving only less critical transactions when no data reservation is adopted. Since data reservation only restricts transactions from starting execution, it will not create any situation in which a transaction locks some node and waits to lock some other nodes. In other words, even if the data reservation mechanism is adopted, no deadlock exists among less critical transactions.

There does not exist any deadlock among highly critical transactions because, once a highly critical transaction starts execution, it will not be blocked by any lower-priority (or equal-priority) real-time transactions. ■

The schedulability of highly critical transactions can be enforced by bounding the maximum number of priority inversion (as shown by the following theorem), where the slack of a highly critical transaction must not be less than the execution time of a less critical transaction.

THEOREM 8 *The maximum number of priority inversion for a highly critical transaction in T under the data reservation mechanism is one.*

Proof: Let a less critical transaction τ' block the execution of some highly critical transaction τ. In order for τ' to block τ, τ' must lock some node N_i whose node ceiling is not less than the preemption level of τ. Obviously, after τ' has locked N_i, no less critical transactions can start execution (It is trivial to prove here that there is only one less critical transaction which can block τ). As a result, the maximum blocking time of a highly critical real-time transaction τ is the execution time of the longest critical section of any transaction whose slack is not less than the slack of τ, as defined in Baker (1990), since, once a highly critical real-time transaction starts execution, it will not be blocked by any lower-priority critical or any less critical transactions. ■

6. Performance Evaluation

6.1. Performance Model and Performance Metrics

The experiments described in this section are meant to assess the performance of the proposed algorithms in inserting, deleting, and querying B^+-tree-indexed data. The simulation experiments compare the performance of the proposed algorithms, B-Link algorithms (Lehman and Yao, 1981), and batch update algorithms (Kerttu, Eljas, and Tatu, 1996) for databases using the EDF scheduling algorithm (Liu and Layland, 1973), where the B-Link algorithms had been shown to give the best performance among many B-tree algorithms, e.g., (Bayer and Schkolnick, 1977; Lehman and Yao, 1981; Mond and Raz, 1985), in reducing the number of deadline violations (Goyal et al., 1995). When non-real-time algorithms were simulated, transactions were scheduled with the EDF priority assignment method (Liu and Layland, 1973). All weaker locks in the simulated algorithms, such as intention shared (IS) in the B-Link algorithms, were implemented as defined in the algorithms. The experiments also assess the performance improvement of the node-sharing, starting-node-transformation, and data reservation techniques.

The simulation model consists of five components: the transaction generator, the CPU, the disk, the B^+-tree index, and the database. The transaction generator generates firm real-time transactions with an arrival rate following a Poisson distribution with a mean ranging from 2 arrivals per second to 11 arrivals per second. The transaction might be a query, an insert, or a delete transaction. When a transaction misses its deadline in a simulation, it will be killed.

The number of keys to be accessed by a transaction was randomly chosen between 1 and 10, and the keys were chosen following a normal distribution function with a variance equal to 100 and a mean randomly selected in the range between 1 and 10,000,000. When a value less than 1 was generated, it was set as 1. Similarly, when a value over 10,000,000 was generated, it was set as 10,000,000. 10 transaction traces per arrival distribution were tested, and their results were averaged. The deadline, D_i, of any generated transaction, τ_i, was derived as follow:

$$D_i = ReadyTime + SF * TranSize * Height * AccessTime$$

where *ReadyTime*, *SF*, *TranSize*, *Height*, and *AccessTime* were the arrival time of τ_i, the slack factor of τ_i, the number of keys accessed by τ_i, the tree height, and the disk access time, respectively. The slack factor of τ_i was randomly chosen between 2 and 8. The ratio of query, insert, and delete transactions was 8 : 1 : 1.

The database was disk-resident with a physical block size equal to 1KB. Initially, the database consisted of 10,000,000 data items with different keys in the B^+-tree index structure. Let the key bit-size and the bit number of a physical block address both be 32 bits, and the fanout of the B^+-tree be 128. The access time of a disk block was estimated to be $13ms$, where the access time consisted of seek time, latency delay, and transfer time. The parameters of the experiments and their baseline settings are summarized in Table 1.

The primary performance metric used is the miss ratio of transactions, referred to as *Miss Ratio*. The *Miss Ratio* of a collection of transactions is the percentage of transactions that

Table 1. Simulation parameters.

Parameter	Explanation	Setting
ArrRate	transaction arrival rate (arrivals/sec)	Poisson(mean in (2, 11))
TransSize	the number of accessed keys	(1, 10)
SF	slack factor	(2, 8)
Fanout	fanout of B-tree nodes	128
PageDisk	access time of a physical block	13ms

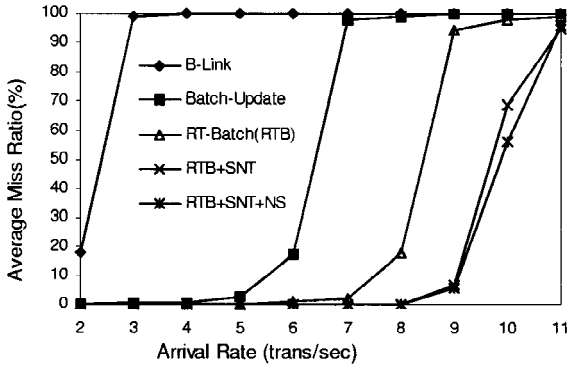
miss their deadlines. Another metric used is the averaged normalized response time of transactions, referred to as *Normalized Response Time*. The *Normalized Response Time* of a transaction is calculated as follows:

$$\textit{Normalized Response Time} = \frac{\textit{CompletionTime} - \textit{ReadyTime}}{\textit{RelativeDeadline}}$$

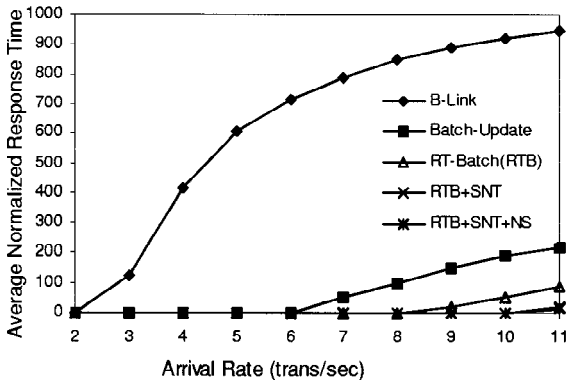
where *CompletionTime*, *ReadyTime*, and *RelativeDeadline* are the completion time, request time, and relative deadline of the transaction, respectively. "Normalized Response Time" is good in measuring the response time of a transaction, which is an indicator regarding how well a real-time application responds to its user requests. The normalization process is to reflect the length of a transaction to make the experiment results more reasonable. (Otherwise, a system should always greedily run short transactions to maximize its performance). Each transaction was considered with the same weight/value in measuring the miss ratios. When the criticality or value of transactions had to be emphasized, the data reservation mechanism was adopted in the simulation, and their miss ratio and response time were reported (Please see Section 6.3)

6.2. Performance Improvement for Batch Protocols

In this set of experiments, we study the performance of the proposed algorithms in inserting, deleting, and querying B^+-tree-indexed data. Figure 4.a shows the miss ratio of all transactions running the B-Link algorithms (B-Link) (Lehman and Yao, 1981), conventional batch update algorithms (Kerttu, Eljas, and Tatu, 1996) with the EDF priority assignment (Batch-Update), the proposed real-time batch algorithms (RT-Batch), RT-Batch with starting-node-transformation (RTB-SNT), and RTB-SNT with node-sharing (RTB-SNT-NS). Obviously, batch algorithms greatly outperformed single-operation algorithms, i.e., the B-Link algorithms. RT-Batch largely reduced the miss ratio of transactions. The miss ratio of RT-Batch started increasing significantly when the arrival rate was higher than 8 arrivals per second, where Batch-Update could not successfully satisfied the deadline requirements when the arrival rate was higher than 6 (even 5) arrivals per second. The starting-node-transformation and node-sharing techniques further improved the miss ratio of transactions. Figure 4.b shows the normalized response time of transactions running B-Link, Batch-Update, RT-Batch, RTB-SNT, and RTB-SNT-NS. RT-Batch, RTB-SNT, and RTB-SNT-NS also greatly outperformed other algorithms.



(a) Average Miss Ratio

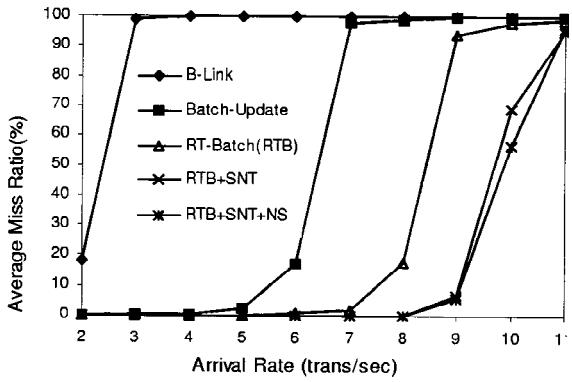


(b) Average Normalized Response Time

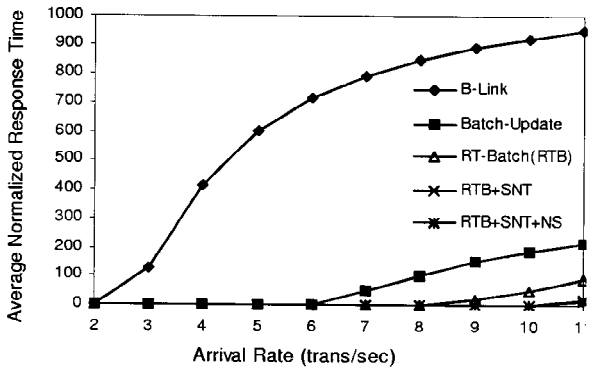
Figure 4. The miss ratio and normalized response time of all transactions, where the ratio of query, insert, and delete transactions were 8 : 1 : 1.

Figures 5, 6, and 7 show the miss ratio and normalized response time of query, insert, and delete transactions running B-Link, Batch-Update, RT-Batch, RTB-SNT, and RTB-SNT-NS, respectively. RT-Batch, RTB-SNT, and RTB-SNT-NS all greatly outperformed other algorithms, and the starting-node-transformation and node-sharing techniques significantly improved the miss ratio of transactions. Note that although the starting-node-transformation and node-sharing techniques can also be used with conventional batch algorithms such as (Kerttu, Eljas, and Tatu, 1996) and have similar performance improvement, the proposed algorithms RT-Batch did have better performance because of the adopted real-time techniques.

When the system consisted of merely query, insert, and delete transactions, Figures 8, 9, and 10 show the miss ratio and normalized response time of all transactions, respectively.



(a) Average Miss Ratio

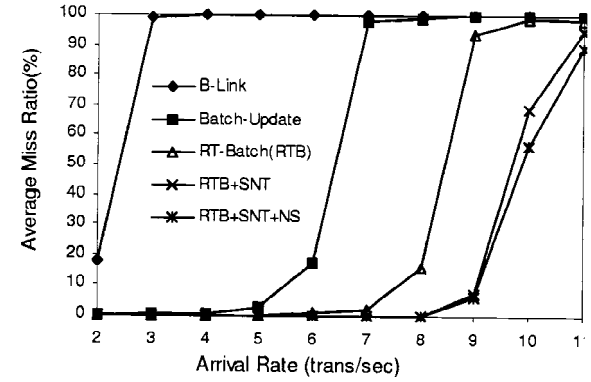


(b) Average Normalized Response Time

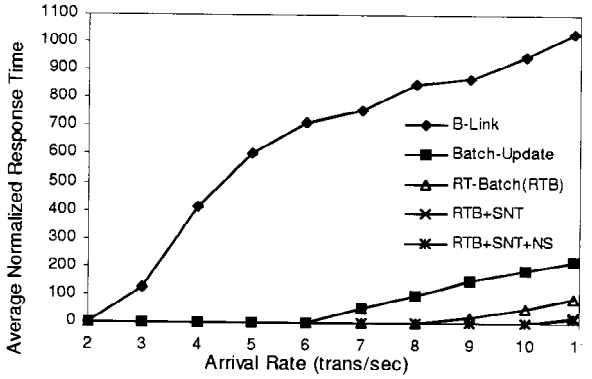
Figure 5. The miss ratio and normalized response time of query transactions, where the ratio of query, insert, and delete transactions were 8 : 1 : 1.

RT-Batch, RTB-SNT, and RTB-SNT-NS greatly outperformed other algorithms when the system consisted of query transactions. However, when the system consisted of only insert or delete transactions, RT-Batch, RTB-SNT, RTB-SNT-NS, and Batch-Update had similar performance because the benefits of adopting the real-time techniques were offset by the longer locking time of intensive exclusive-lock requests by RT-Batch, RTB-SNT, and RTB-SNT-NS. Note that it was pointed out that the relaxed B^+-tree structure adopted in this paper was not good for a system consisting of only insert or delete transactions (Kerttu, Eljas, and Tatu, 1996).

Figure 11 shows that the performance of RT-Batch, RTB-SNT, and RTB-SNT-NS remained similar when the relative deadline of a rebalance transaction was set as one, two, or four times of the relative deadline of its corresponding insert or delete transaction. It was



(a) Average Response Time



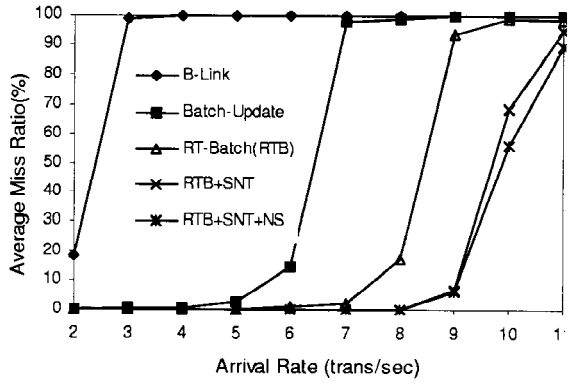
(b) Average Normalized Response Time

Figure 6. The miss ratio and normalized response time of insert transactions, where the ratio of query, insert, and delete transactions were 8 : 1 : 1.

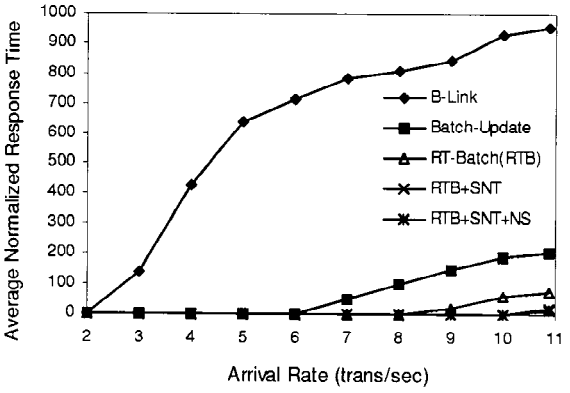
mainly because rebalance transactions only locked one single node (and its child nodes) and did not generate much blocking time to other urgent transactions.

6.3. Data Reservation

The experiments described in this section are meant to assess the capability of the data reservation technique in reducing the miss ratio of highly critical transactions[1]. Let 1% of total transactions be highly critical transactions; a half of them was insert and delete transactions, and the other half was query transactions. The number of keys accessed by a critical transaction was randomly chosen between 1 and 10, and the keys were chosen by a normal distribution function with a variance equals to 100 and a mean in the range



(a) Average Response Time

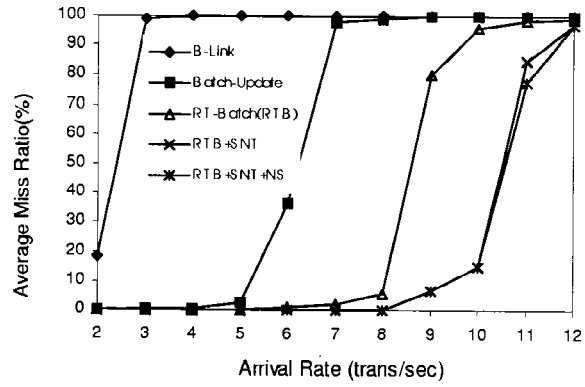


(b) Average Normalized Response Time

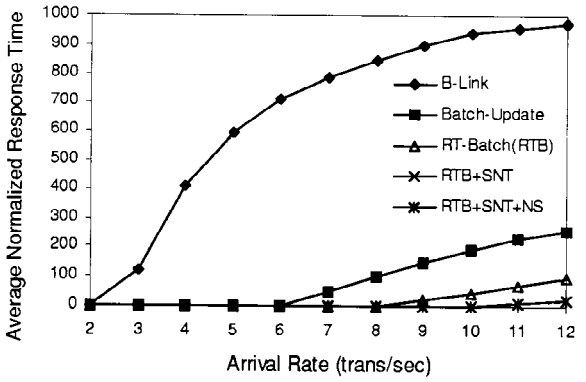
Figure 7. The miss ratio and normalized response time of delete transactions, where the ratio of query, insert, and delete transactions were 8 : 1 : 1.

between 500 and 3,500. The small mean range was to create more conflicts among the (highly critical) transactions.

Figure 12 shows that the miss ratio and normalized response time of highly critical transactions were greatly improved when the priority inheritance mechanism (RTB-PI), data reservation technique (RTB-DR), or both (RTB-PI-DR) were adopted. RTB-DR and RTB-PI-DR had the best performance in the simulation experiments. The priority inheritance mechanism did not improve the system performance when the data reservation technique was already adopted. It was because the data reservation technique already embedded the idea of priority inheritance into its design. Note that when a higher-priority transaction started execution, it will not be blocked by any lower-priority transaction. On the other hand, if a higher-priority transaction was blocked from starting execution, no intermediate-priority transaction can preempt the execution of a lower-priority transaction which currently blocks



(a) Average Response Time

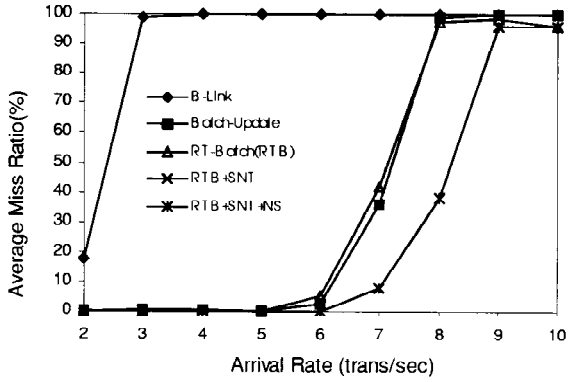


(b) Average Normalized Response Time

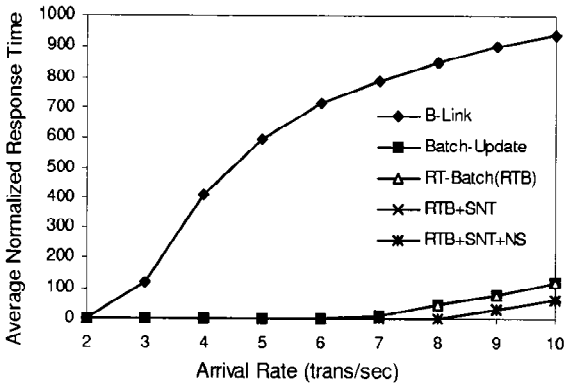
Figure 8. The miss ratio and normalized response time of transactions in a system consisting of merely query transactions.

the higher-priority transaction. In other word, the executing lower-priority transaction already "implicitly" inherited the priority of the higher-priority transaction under the data reservation technique.

Figure 13 shows the miss ratio and normalized response time of all transactions. The system performance was slightly improved when the priority inheritance mechanism (RTB-PI), data reservation technique (RTB-DR), or both (RTB-PI-DR) were adopted. It was because the ratio of critical transactions in the system was very small.



(a) Average Response Time

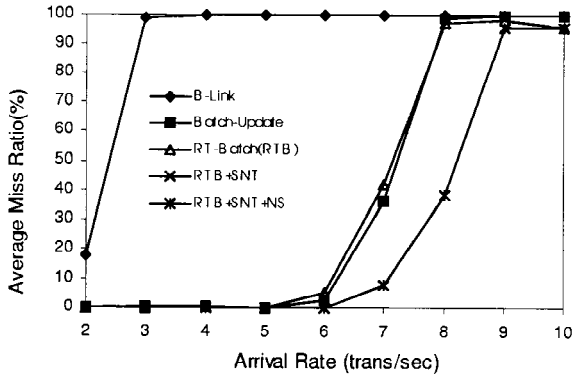


(b) Average Normalized Response Time

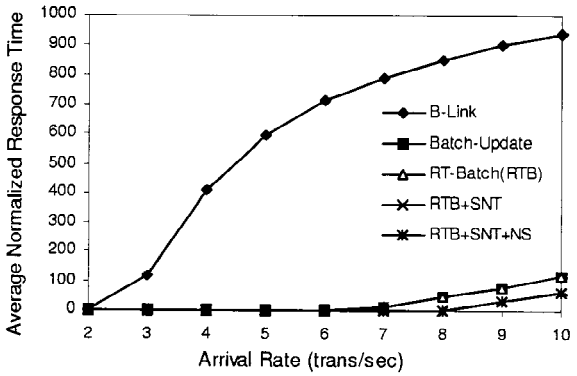
Figure 9. The miss ratio and normalized response time of all transactions in a system consisting of merely insert transactions.

7. Conclusion

Index-based concurrency control is of paramount importance to the performance of many large-scaled real-time databases. Most of the past research studies in data-item-oriented real-time concurrency control cannot be directly applied to a database with index structures. This paper proposes methodologies to control the access of disk-resident B^+-tree-indexed data in a batch and firm real-time fashion. We propose batch algorithms to insert, query, delete, and rebalance B^+-tree-indexed data and to demonstrate the performance of the algorithms by a series of simulation experiments. A relaxed and less blocking B^+-tree index structure is explored for firm real-time and batch processing of transactions with multiple requests. We propose methodologies to share nodes among batch transactions to



(a) Average Response Time

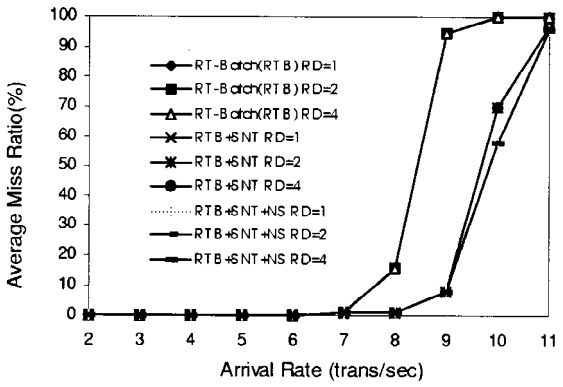


(b) Average Normalized Response Time

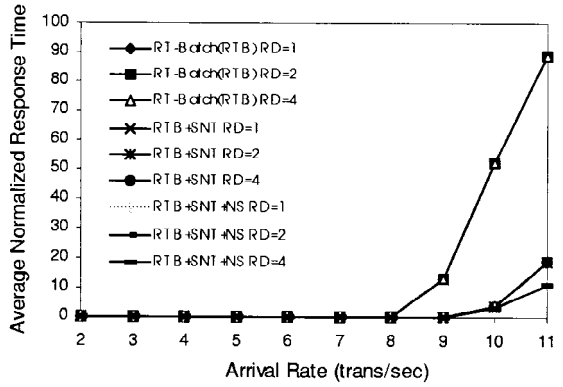
Figure 10. The miss ratio and normalized response time of all transactions in a system consisting of merely delete transactions.

further reduce the number of disk I/O without introducing more priority inversion. When the schedulability of transactions with highly critical timing constraints is considered, we propose a data reservation mechanism based on the idea of preemption level (Baker, 1990). The performance of our methodologies has been evaluated by a series of experiments, for which we have obtained some encouraging results.

The algorithms presented in this paper address applications in which index-based insertions, deletions, and range queries, form a significant portion of the workload. However, there do exist RTDBS applications, where the set of data items is fairly static and little need on range queries exists. For these applications, hash-based techniques may be more appropriate. We intend to investigate this issue in the future research.



(a) Average Response Time

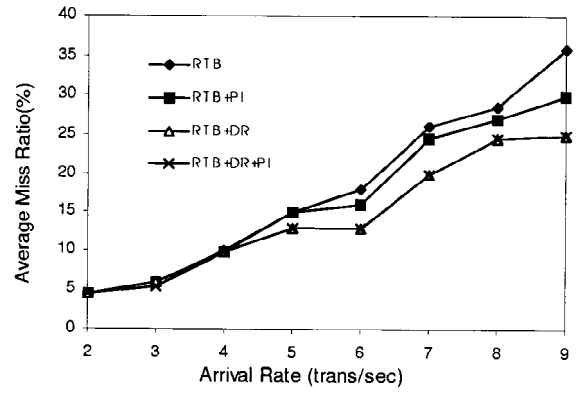


(b) Average Normalized Response Time

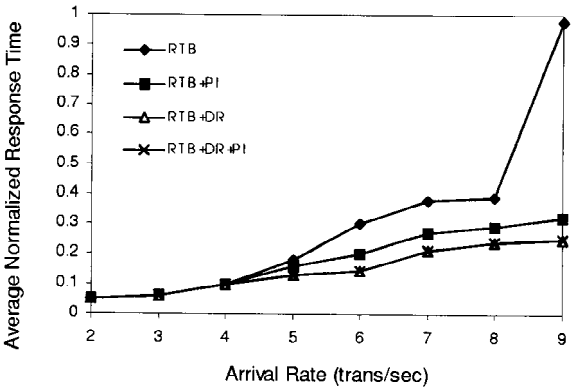
Figure 11. The miss ratio and normalized response time of all transactions, where the ratio of query, insert, and delete transactions were 8 : 1 : 1. ($RD = i$ means that the relative deadline of a rebalance transaction was set as i times of the relative deadline of its corresponding insert or delete transaction.)

Appendix A

Procedure Rebalance(*parent_node*, *child_node*, deadline)
  Input: *parent_node*, *child_node*: nodes which need rebalancing.
    /* *parent_node* is the parent node of *child_node* */
  Begin
    W-Lock(*parent_node*);
    W-Lock(*child_node*);
    Read_Node(*parent_node*);
    Read_Node(*child_node*);



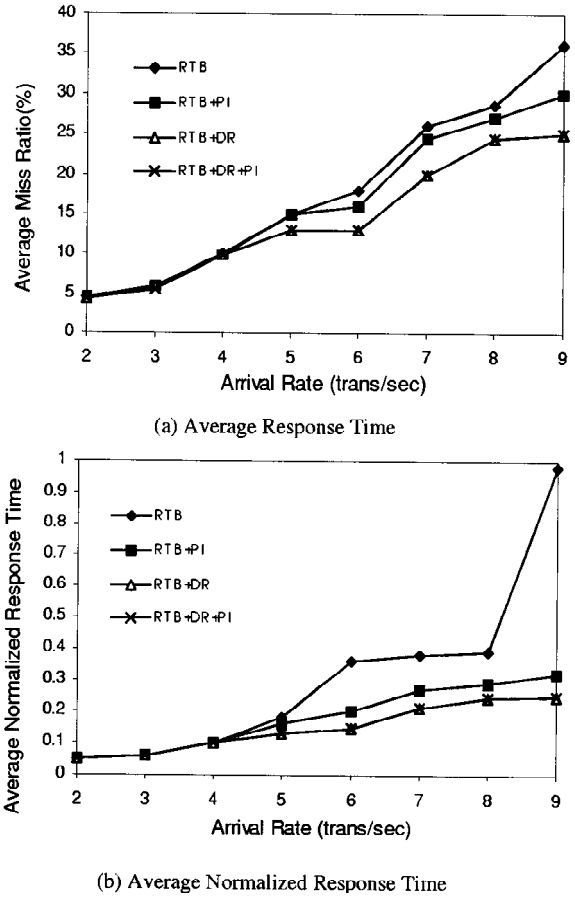
(a) Average Response Time



(b) Average Normalized Response Time

Figure 12. The miss ratio and normalized response time of highly critical transactions, where the ratio of query, insert, and delete transactions were 8 : 1 : 1.

If *parent_node* is no longer the parent node of *child_node*
  Then /* The B^+-tree Index structure is changed. */
    Unlock(*parent_node*);
    Unlock(*child_node*);
    Return;
  EndIf;
For each $node_i$ in the control set of *child_node* from left to right Do
  If $node_i$ is empty
    Then
      If $node_i$ has a right sibling node in the control set
        Then



(a) Average Response Time

(b) Average Normalized Response Time

Figure 13. The miss ratio and normalized response time of all transactions, where the ratio of query, insert, and delete transactions were 8 : 1 : 1.

```
                Merge node_i with its right sibling node;
            Else If node_i has a left sibling node in the control set
                Merge node_i with its left sibling node;
            Else
                Leave node_i as an empty node;
            EndIf;
        EndIf;
            /* All data are sorted, and all nodes are linked
            according to the definitions of B+-tree control set */
        EndIf
EndDo;
```

```
        If the size of the control set is one
          Then /* child_node is the only node in the control set */
              If child_node is empty
                  If parent_node is null
                    Then /* there is no data in the tree. */
                        Set Root of the tree as null;
                    Else
                        Update the keys and links of parent_node;
                        If parent_node is empty
                          Then
                              grand_parent_node = the parent node of parent_node;
                              Add-Work-Queue(Job(Rebalance, grand_parent_node, parent_node, deadline));
                          EndIF;
                    EndIf;
          Else
              If parent_node is null
                  Then /* a new root is needed */
                    Create a new node new_root as the parent node of nodes in the control set
                    Set Root of the tree as new_root;
                    If new_root is overflow
                        Then
                          Split new_root and link them together;
                          Add-Work-Queue(Job(Rebalance, null, new_root, deadline));
                        EndIf;
                  Else
                    Update the keys and links of parent_node;
                    If parent_node is overflowed
                        Then
                          Split parent_node and link them together;
                          grand_parent_node = the parent node of parent_node;
                          Add-Work-Queue(Job(Rebalance, grand_parent_node, parent_node, deadline));
                  EndIF;
          EndIf;
        Unlock(parent_node);
        Unlock(child_node);
    End
```

Appendix B

```
Procedure Batch-Delete(key_set, node, deadline)
    Input: key_set = (key_1, key_2, ..., key_{n_i})
        node: starting node in the tree.
    Begin
```

```
If node is a leaf node
  Then
      W-Lock(node);
      Delete-Data(key_set, node, deadline);
      Return;
  Else IF node is the parent node of a leaf node
      W-Lock(node);
  Else
      R-Lock(node);
  EndIF;
node_list = φ;
EnQueue(node_list, (key_set, node));
While node_list is not empty
  request = DeQueue(node_list); /* request = (key_set, node) */
  Read_Node(request.node); /* disk I/O for a tree node */
  For each c-node_j in the control set of request.node Do
      Child_Set = {node_i | node_i is a child node of c-node_j
          which includes some keys in request.key_set};
      Let key_set_i be a subset of request.key_set
          whose keys are in node_i;
  EndDo;
  If request.node is the parent node of a leaf node
      Then
          For each child_node_i in Child_Set (from left to right) Do
              W-Lock(child_node_i); /* child_node_i is a leaf node. */
              Delete-Data(key_set_i, child_node_i, deadline);
              Unlock(child_node_i);
          EndFor;
      Else
          If request.node is the grand parent node of a leaf node
              Then
                  For each child_node_i in Child_Set (from left to right) Do
                      W-Lock(child_node_i);
                      EnQueue(node_list, (key_set_i, child_node_i));
                  EndFor;
              Else
                  For each child_node_i in Child_Set (from left to right) Do
                      R-Lock(child_node_i);
                      EnQueue(node_list, (key_set_i, child_node_i));
                  EndFor;
              EndIf;
      EndIf;
  Unlock(request.node);
EndWhile;
```

```
    End

Procedure Delete-Data(key_set, leaf_node, deadline)
    Input: key_set = (key_1, key_2, ..., key_{n_i})
        leaf_node: a w-locked leaf node.
    Begin
        Delete keys and data in the proper node in the control set of leaf_node;
            /* empty nodes may exist. */
        If any node in the control set of leaf_node becomes empty
            Then
                parent_node = the parent node of leaf_node;
                Add-Work-Queue(Job(Rebalance, parent_node, leaf_node, deadline));
                    /* A new request is inserted! */
            Else
                Unlock(leaf_node);
            EndIf;
    End
```

Acknowledgments

Supported in part by a research grant from the National Science Council under Grants NSC88-2213-E-194-034, the Hong Kong RGC CERG Grant 9040353 and City University of Hong Kong under Strategic Grant 7000849.

Notes

1. The schedulability of highly critical transactions can be enforced by bounding the maximum number of priority inversion, as shown by Theorem 8.

References

Abbott, R., and Garcia-Molina, H. 1988. Scheduling real-time transactions: a performance evaluation. *Proceeding of the 14th VLDB Conference*, Los Angeles, CA, pp. 1–12.

Andresen, D., Yang, T., Egecioglu, O., Ibarra, O. H., and Smith, T. R. 1996. Scalability issues for high performance digital libraries on the world wide web. *The 3rd Forum on Research and Technology Advances in Digital Library*, pp. 139–148.

Baker, T. P. 1990. A stack-based resource allocation policy for real time processes. *IEEE 11st Real-Time Systems Symposium*, December 4–7.

Bayer, R., and Schkolnick, M. 1977. Concurrency of operations on B-trees. *Acta Informatica* 9.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. 1987. *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley.

Bestavros, A. 1994. Timeliness via speculation for real-time databases. *IEEE 15th Real-Time Systems Symposium*.

Bestavros, A. 1996. Advances in real-time database systems research. *Special Section on RTDB of ACM SIGMOD Record* 25(1).

Bestavros, A., Lin, K. J., and Son, S. H. 1997. *Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers.

Choy, D. M., and Morris, R. J. T. 1996. Services and architectures for electronic publishing. *IEEE COMPCON*, pp. 291–297.

Cutting, D., and Pedersen, J. 1990. Optimization for dynamic inverted index maintenance. *ACM SIGIR 1990, International Conference of Information Retrieval*, pp. 405–411.

Dipippo, L. B. C., and Wolfe, V. F. 1993. Object-based semantic real-time concurrency control. *IEEE Real-Time Systems Symposium*, December.

Elmasri, R., and Navathe, S. B. 1994. *Fundamentals of Database Systems. Second Edition*. Addison-Wesley Publishing Company.

Goyal, B., Haritsa, J. R., Seshadri, S., and Srinivasan, V. 1995. Index concurrency control in firm real-time DBMS. *Proceedings of the 21th VLDB Conference*, pp. 146–157.

Grossman, R., Qin, X., Xu, W., Hulen, H., and Tyler, T. 1995. An architecture for a scalable, high performance digital library. *IEEE 14th Symposium on Mass Storage Systems*, Sept. pp. 89–98,

Haritsa, J. R., Carey, M. J., Livny, M. 1990. On being optimistic about real-time constraints. *Proceeding of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Princples of Database Systems*, April pp. 331–343.

Haritsa, J., and Seshadri, S. 2000. Real-time index concurrency control. *IEEE Transactions on Knowledge and Data Engineering*.

Johnson, T., and Shasha, D. 1990. A framework for the performance analysis of concurrent B-tree algorithms. *ACM Symposium on Principles of Database Systems*, April.

Kamath, M. U., and Ramamritham, K. 1993. Performance characteristics of epsilon serializability with hierarchical inconsistency bounds. *International Conference on Data Engineering*, April, pp. 587–594.

Kerttu, P. M., Eljas, S. S., and Tatu, Y. 1996. Concurrency control in B-trees with batch updates. *IEEE Transaction on Knowledge and Data Engineering* 8(6): 975–984.

Kuo, T.-W. 1994. *Real-Time Database—Semantics and Resource Scheduling*. Ph.D. Thesis, Department of Computer Sciences, The University of Texas at Austin.

Kuo, T.-W., and Mok, A. K. 1992. Application semantics and concurrency control of real-time data-intensive applications. *IEEE 13th Real-Time Systems Symposium*.

Kuo, T.-W., and Mok, A. K. 1993. SSP: a semantics-based protocol for real-time data access. *IEEE 14th Real-Time Systems Symposium*, December.

Kuo, T.-W., Wei, C.-H., and Lam, K. Y. 1999. Real-time data access control on B-tree index structures. *IEEE 15th International Conference on Data Engineering*, April.

Lam, K.-Y., Kuo, T.-W., and Shu, L.-C. 1998. On using similarity to process transactions in stock trading systems. *IEEE Workshop on Dependable and Real-Time E-Commerce Systems*, June.

Lang, S. D., Driscoll, J. R., and Hou, J. H. 1986. Batch insertion for tree structure file organizations—improving differential database represeantation. *Information Systems* 11(2): 167–175.

Lehman, P., and Yao, S. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems* 6(4).

Liang, M.-C., Kuo, T.-W., and Shu, L.-C. 1997. A quantification of aborting effect for real-time data accesses. *IEEE Transactions on Computers*. To appear.

Lin, Y., and Son, S. H. 1990. Concurrency control in real-time databases by dynamic adjustment of serialization order. *IEEE 11th Real-Time Systems Symposium*, December 4–7.

Liu, C. L., and Layland, J. W. 1973. Scheduling algorithms for multiprogramming in a hard realtime environment. *JACM* 20(1): 46–61.

Mohan, C., and Levine, F. 1992. ARIES/IM: An effecient and high concurrency index management method using write-ahead logging. *ACM SIGMOD Conference*, June.

Mond, Y., and Raz, Y. 1985. Concurrency control in B^+-trees databases using preparatory operations. *VLDB*.

Nurmi, O., Soisalon-Soininen, E., and Wood, D. 1987. Concurrency control in database structures with relaxed balance. *Proceedings of Sixth ACM Symp. Principles of Database Systems*, pp. 170–176.

Peng, C.-S., and Lin, K.-J. 1996. A semantic-based concurrency control protocol for real-time transactions. *IEEE 1996 Real-Time Technology and Applications Symposium*.

Pu, C., and Leff, A. 1991. Epsilon-serializability. *Technical Report CUCS-054-90*. Dept. of Computer Science, Columbia University, January.

Ramamritham, K., and Pu, C. 1995. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering* 7(6): 997–1007.

Sha, L., Rajkumar, R., and Lehoczky, J. P. 1990. Priority inheritance protocols: an approach to real-time synchronization. *Technical Report CMU-CS-87-181*. Dept. of Computer Science, CMU, November 1987. *IEEE Transactions on Computers* 39(9) September.

Sha, L., Rajkumer, R., Son, S. H., and Chang, C.-H. 1991. A real-time locking protocol. *IEEE Transaction on Computers* 40(7): 793–800.

Son, S. H., editor. 1988. *ACM SIGMOD Record: Special Issue on Real-Time Databases*, March.

Srinvasan, V., and Carey, M. 1991. Performance of B-tree concurrency control algorithms. *ACM SIGMOD Conference*. May.

Srivastava, J., and Ramamoorthy, C. V. 1988. Efficient algorithms for maintenance of large database indexes. *Proc. Fourth International Conference of Data Engineering*, pp. 402–409.

Xiong, M., Ramamritham, K., Sivasankaran, R., Stankovic, J. A., and Towsley, D. 1996. Scheduling Transactions with temporal constraints: exploiting data semantics. *IEEE Real-Time Systems Symposium* December: 240–251.