EFFICIENT CONCURRENCY CONTROL PROTOCOLS FOR B-TREE INDEXES

Ekow J. Otoo

SCS-TR-166, JANUARY 1990

School of Computer Science, Carleton University
Ottawa, Canada, KIS 5B6

Efficient Concurrency Control Protocols for B-tree Indexes

Ekow J. Otoo
Centre for Parallel and Distributed Computing
School of Computer Science
Carleton University
Ottawa, Canada

Abstract

We address the problem of maintaining the consistency and integrity of a B-tree index under a number of independent asynchronous concurrently executing processes. Each process issues an insert, delete or look-up operation. We present a survey of some of the previous works in this area and propose two simple but efficient locking protocol for B-tree indexes. Update processes in our scheme make use of lock conversions. The protocols are deadlock and livelock free and each process ensures that the B-tree characteristics are maintained at all times.

Keywords and Phrases: algorithms, B-tree, concurrency control protocol, data structures.
CR Categories and Subject Description: E.2. Data Storage Representation, H.2.2. Physical Design, H.3.2. Information Storage, H.3.3. Information Search and Retrieval.

1 Introduction

In providing an index into a large database, one typically would like a scheme that is fast for both random and sequential accesses, allows for insertions and deletions without performance deterioration and has sufficiently high storage utilization. The B-tree index [2,7,13] and its variants such as B^*-tree, B^+-tree [13] and Prefix B-tree [4] satisfy these requirements and have become generally, the indexing scheme of choice in most current database systems. The B^+-tree, to be defined subsequently, appears to be the most widely used. In the sequel, we will use the term B-tree as a generic term for all the variants and in particular for B^+-tree. The term B^+-tree will be used explicitly sometimes, when we discuss issues specific to B^+-tree.

Originally proposed for a single user environment, the use of B-trees as an index to very large databases accessible to multiple users has raised issues of concurrency control for this structure. Consequently, the basic algorithms for insertions, deletions and look-up in B-trees have had to be modified to accommodate concurrent operations of search, insert and delete.

The earliest works reported on concurrent operations on B-trees were by Samedi [21] and Bayer and Schkolnick [3]. Later, other works by Kwong and Wood [16], Lehman and Yao [18] and Ellis [9] were presented. Unlike the situation of a single user access to the B-tree, concurrent accesses require explicit specification of rules embedded in the algorithms so that conflicting operations on the same nodes of the tree can be resolved while still maintaining the consistency of the indexes. Essentially the rules require that:

1. each user process that accesses a consistent B-tree, on completion, should leave the tree in a consistent state as if the process had exclusive access to it;

2. as many processes as possible must be allowed to execute without being blocked;

3. each process must complete in a finite time;

The exact rules to be followed by each process are known as the concurrency control protocols. The requirement (1) is generally referred to the condition for *serializability* [5,10,24,25]. The requirement (2) ensures that the protocol has a high *degree of concurrency*, and the requirement (3) ensures that the protocol is *deadlock free* and *live-lock free*. The early works on concurrency control on B-tree ensured that the protocols satisfied the above requirement. Each method had a different specification of the rules for execution, [3,6,9,14,16,18,19,22,21]. Some discussions on the distinctive differences on the various schemes proposed are presented in [16] and [6]. We present a

brief summary of the various schemes in this paper.

Let us first clarify the scope of concurrency control on the B-tree in the context of concurrency control of transactions accessing data items indexed by a B-tree. From a users view point (or external transaction's view), a transaction T that is initiated, accesses a set$S = \{r_1, r_2, \ldots, r_N\}$ of data items. Each $r_i = \langle k_i, info_i \rangle$, where k_i denotes the key and $info_i$ denotes the rest of the information in the record. The key k_i are organized into a B-tree index. The index only provides an access path to the data items so that they can be located efficiently in both random and sequential manner. Consequently each access to a data item by a transaction proceeds by first traversing the nodes of the tree until a leaf node is reached. Ignoring the traversal of the B-tree index, each transaction may be seen as consisting of a sequence of reads and writes of the data items r_i. The execution of multiple user processes may be synchronized with respect to the data items r_i, that are accessed. The database then should maintain a consistent state before and after each execution of a transaction. The sequence of actions on the data items by each transaction must be seen as either executing to completion or not at all. This notion is the *atomicity* property required of a transaction. It should be irrelevant, from the view point of an external transaction, how the data items were located.

One mechanism by which a set of transactions are able to derive serializable schedules is the use of locking protocols. Before a transaction accesses a database item, it locks that item and at some point in the future, it unlocks it. A protocol proposed by Eswaren et al [10], called the two phase locking protocol ensures serializability of a set of external transactions. Equivalently, if the degree of conflicting access on the database items is low, an optimistic concurrency protocol [15,19] may be instituted in place of a locking protocol. We emphasize that the manner of achieving concurrency control for the transactions, in so far as their accesses to the data items r_i is concerned, is independent of the concurrency control required to manipulate the nodes of the index tree to arrive at the data items.

Since each access of a database item involves a traversal of the B-tree index, multiple processes can manipulate the index nodes. We need to enforce concurrency control for the simultaneous access of the nodes of the B-tree. In this regard, we can isolate, the operations on the B-tree nodes of the tree by a transaction T_i as a set of independent sub-transactions $T_i = \{\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_n}\}$, where each sub-transaction τ_{i_j} executes a look-up(), delete() or insert() operation of one key value. We can perceive the concurrency control problem on the B-tree index as that of ensuring the serializability of the sub-transactions τ_{i_j}.

For each external transaction T_i, the sub-transactions $\{\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_n}\}$ may be executed serially or in two phases. The first phase of parallel reads of the read-set, followed by a second phase of validation and parallel writes of the write-set. Whenever the external transactions require access to a number of distinct data items, divorcing the concurrency control mechanism for manipulating the B-tree index from the concurrency control mechanism for the external transactions is advantageous for recovery purposes. We do not address the problem of recovery in this paper. In some specific instances where each external transaction executes either a search, insert or a delete of a record at a time, i.e., $T_i = \tau_i$, the concurrency control for the external transactions may be achieved by a simple extension of the concurrency control protocol on B-tree index.

Given a protocol for concurrent access of the B-tree index, the external transactions may be synchronized in general by either one of the two methods: i) two phase locking, and ii) optimistic concurrency control methods. Note that it is possible to have a deadlock free protocol for the B-tree while the external transactions may eventually suffer from deadlocks depending on the choice of protocol for synchronizing the external transactions. Our focus in this paper is on protocols for concurrent access of the B-tree index.

There are two models of computation by which processes can manipulate the same node of the B-tree. A process modifying a node may do this concurrently with some other process performing a look-up on the same node. The node is assumed to be held in a shared memory. Alternatively, a process modifying a node would have exclusive access to the node either in a shared memory or by making a copy of it in its independent workspace. The former model is implied in the paper by [9] and [16] where inter-nodal concurrency in B-trees is discussed. An update process modifies the node from right to left while a look-up process reads the content of the node from left to right. For our purpose, the model of computation is immaterial. We assume the existence of a lock manager who grants or denies requests for locks.

The various methods for concurrency control in B-trees have been compared under different criteria such as degree of concurrency, number of different types of locks used during a look-up or an update, the number of traversal of the tree to perform an update and the number of simultaneous look-up and update processes allowed in the same sub-tree.

We add to the works on concurrency control on B-trees two new methods: the *Bottom-Up Exclusive Locking (BUX)* and *Top-Down Predictive locking (TDP)*. The bottom-up exclusive method improves upon the approach of Bayer and Schkolnick [3] by allowing look-up processes to be in the scope of update processes during the restructuring phase. The top-down predictive restructuring

method is a realization of a method first proposed by Guibas and Sedgewick [11] for balancing binary trees. Essentially they proposed that in inserting into (or deleting from) a balanced tree, anticipated rotations may be carried out before the actual insertion or deletions of a key is effected at the leaf node.

As we shall show later, the two techniques we propose are basically the same. They differ only with respect to the times at which restructuring of the tree is carried out; either before or after an update to the leaf page is made. Both methods are based on node locking and in each case, the number of exclusive locks held at any one time is at most three.

The rest of the paper is organized as follows. In the next section, we define the B-tree and the variant called the B^+-tree. Some of the terms relevant to the discussions in the subsequent sections are also defined. In section 3, we review some of the related works that have been presented. Details of the algorithms for the two new protocols are presented in sections 4 and 5 respectively. We make some comparisons of our work with the earlier proposed methods in section 6 and we conclude in section 7.

2 Preliminaries

We present first the definition of the basic B-tree structure. The B-tree has been defined by Bayer and McCreight [2], Commer [7] and Knuth [13] and in each case the definition is sightly different. We follow the definition in [13].

Definition 2.1 *A B-tree of order m is a balanced multi-way tree with the following properties:*

1. *A node has at most m children;*
2. *Every node except the root has at least $\lceil m/2 \rceil$ children;*
3. *The root may have at least two children except when it is a leaf node in which case it could be empty;*
4. *A non-terminal node with s children contains s-1 keys for $\lceil m/2 \rceil \leq s \leq m$;*
5. *All terminal nodes appear at the same level, i.e., the path length from the root to any terminal node is the same.*

The structure of a node in a B-tree is shown in Figure 2.1. We ignore the record pointers associated with the keys and show only the pointers to other index nodes. Let the keys and page-pointer pairs in a node Q be labelled as shown below.

$$[(p_0, K_0); (p_1, K_1); \ldots (p_{s-2}, K_{s-2}); (p_{s-1}, K_Q)] .$$

Figure 2.1: Schematic structure of a node of the B-tree.

The key K_Q designates the highest key value in the node Q. Suppose during the search for a record whose key is k_t, we arrive at the node Q. Then if the key occurs in the node, we use the record pointer (not shown in the above diagram) to locate the record. Otherwise we follow the pointer p_i according to the following rule:

$$\begin{cases} i = 0 & \text{if } k_t < K_0; \\ i = j & \text{if } K_{j-1} < k_t < K_j; 1 \leq j \leq s-1; \\ i = s-1 & \text{if } k_t > K_{s-1}; \end{cases}$$

An example of a B-tree of order $m = 3$ is shown in Figure 2.2. A B-tree of order 3 is also called a 2-3-tree [1,9]. Details of insertion and deletion algorithms for a B-tree may be found in [2,7,13].

To search for a target record whose key is k_t, the record is looked-up in the root node. If it is found the algorithm reports success otherwise the pointer determined in accordance with the rule stated above is followed until either the terminal node is reached or the key is found in some node. Some of the details of the search algorithm are embedded in the concurrency control algorithms to be presented later. The cost of searching an N record file indexed by a B-tree structure of order m is given by the height h, of the tree where $h \leq \log_{\lceil m/2 \rceil}(\frac{N+1}{2})$. In the standard B-tree, the cost of retrieving the next record in sequence is also $O(\log_{\lceil m/2 \rceil}(\frac{N+1}{2}))$.

To improve upon the cost of retrieving records in sequence order, the B^+–tree was proposed [13]. This scheme achieves O(1) disk access to retrieve the next record in sequence without compromising the random access retrieval cost. The B^+-tree may be defined basically as a standard B-tree except that the records (or keys and associated pointers to the actual records) reside in the leaf nodes. The leaf nodes are linked in key sequence order and are called the *sequence set.* The internal nodes contain values that serve as separators of the leaf nodes and are duplicates of the highest key values in the leaf nodes. The internal nodes of the B^+-tree are structured as the standard B-tree index. This is termed the *index set.* The entries in the index set only serve to direct the traversal along the correct path to the leaf node where the target record of the search key should reside.

To delete a record, the record is simply removed from the leaf node. However if the key of the deleted record is a separator key, a copy of it is still retained in the index set. This is not immediately deleted until the nodes for which it serves as the separator value, are merged. Figure 2.3 illustrates

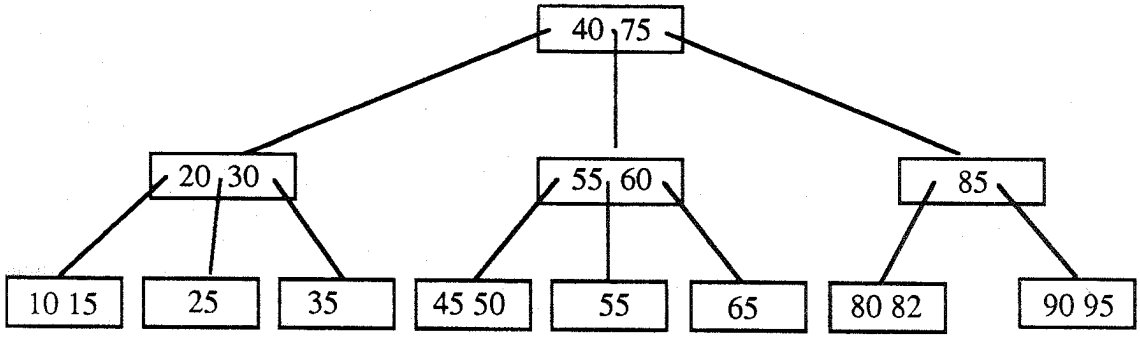

Figure 2.2: An example of a B-tree of order m = 3.

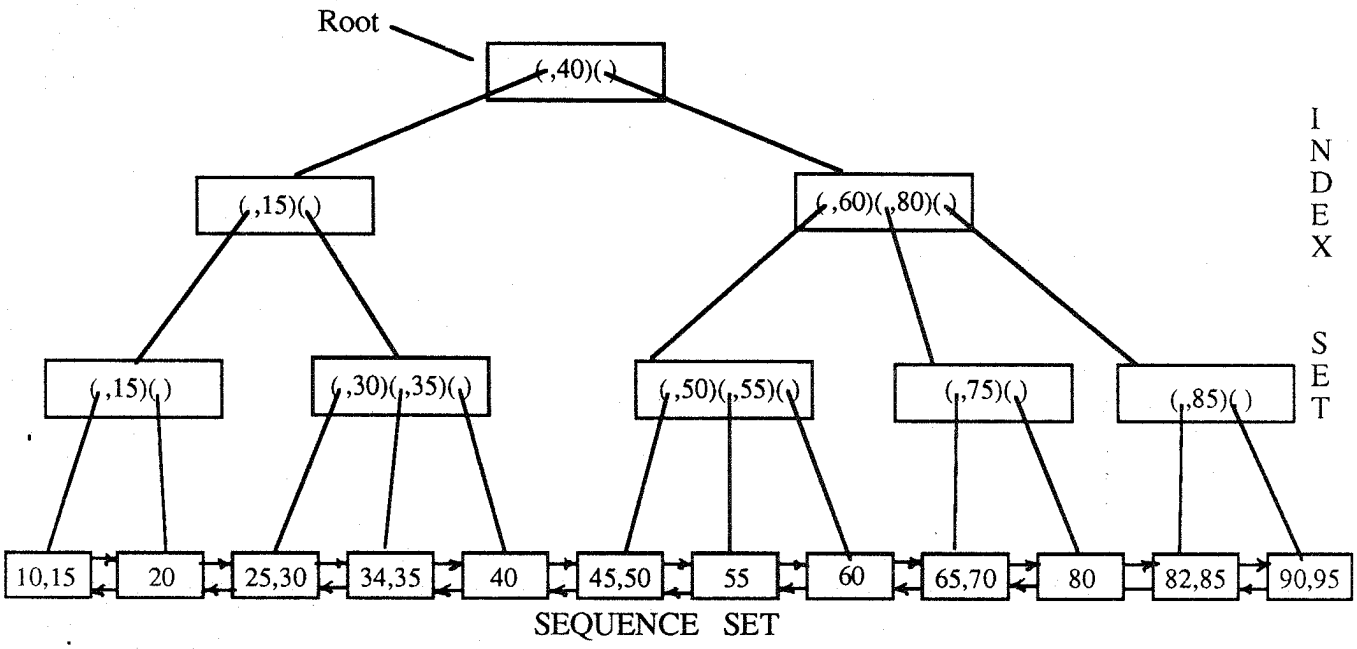

Figure 2.3 : An example of B^+-tree of order m = 3

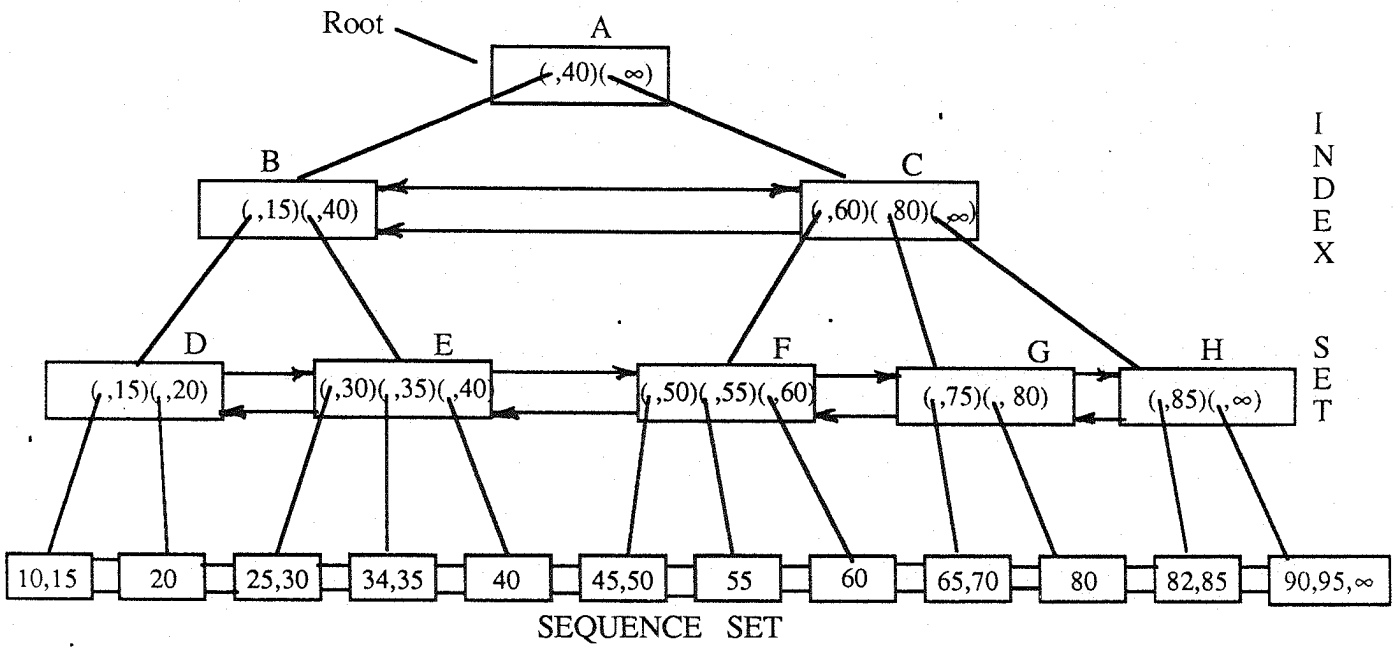

Figure 2.4: An illustration of a modified B -tree of order m = 3.
(This may be percived as a dynamic ISAM organization)

the structure of a $B^+ - tree$. Other variants of the basic B-tree e.g., *Prefix B-tree* are discussed in [4,7,13].

In fact the internal nodes need not be organized as the standard B-tree. Rather the second level of index could be organized as the first level of index by duplicating the highest key values in the first level index nodes. We therefore have a structure very much like a dynamic ISAM organization. Figure 2.4 shows an example of such a B^+-tree with this modification for the same set of keys used to illustrate the basic B-tree of Figure 2.2. A sentinel key K_∞ larger than any key in the file has to be explicitly encoded in the structure.

The use of B^+-tree in a user multi-user environment may require some modification to the basic structure depending on the proposed concurrency control protocol. The $B^{link} - tree$ of Lehman and Yao is one such example. In fact we have made a similar modification in illustrating the B^+-tree of Figure 2.4. In the sequel our use of the term B-tree will mean the structure shown in Figure 2.4. The horizontal links is not necessary for the Top Down Predictive locking protocol to work correctly. Let us first define some general terms applied to tree structures.

Definition 2.2 *The depth of node p, denoted by $\delta(p)$ is the number of edges in the path from the root R, to p.*

The root R, is at depth $\delta(R) = 0$. In so far as concurrent access to the B^+-tree is concerned, we will focus on sub-transactions each of which consists of only one of the following operations.

Look-up(k,R): Determines if the key k is in the tree of root R. If it is, it is made available in the workspace of the transaction. Otherwise it reports failure.

Insert(k, R): Inserts the key k in the B-tree of root R.

Delete(k, R): Deletes the key k from the B-tree of root R.

Given the properties of the B-tree, it is essential that the concurrency control protocol preserves the properties before and after each of the operations of Look-up(), Insert() and Delete(). In the execution of a look-up, no modification to the structure is made. Consequently, the properties of the B-tree are preserved before and after a look-up operation. An insert or a delete operation may restructure the tree. Consider an insert process. The restructuring may be carried out before or after a key is inserted. The latter is the more popular approach and has been discussed extensively in the literature. The tree is reorganized from the leaf towards the root if necessary after the key is inserted at the leaf node. The former method due to Guibas and Sedgewick [11] is less popular and forms the basis of the *top-down-predictive method* to be discussed later. In this approach, the basic idea is to check every node encountered during the traversal from the root to the leaf to see if it is already full. If it is, the node is immediately split.

The crucial problem in concurrency control on B-trees is to ensure the consistency of the tree during the restructuring phase. The restructuring phase is triggered by an insert or delete operation. We will refer to an Insert() or a Delete() process simply as an Update() process when we do not wish to make a distinction between the two.

Definition 2.3 (i-safe, d-safe) *A node p in the access path of an update process U is said to be unsafe for insertion if U = Insert() and p contains m-1 keys otherwise it is i-safe. It is unsafe for deletion if U = Delete() and p contains $\lceil m/2 \rceil - 1$ keys otherwise it is d-safe.*

Definition 2.4 *An i-safe or d-safe node on which an insert (delete) process terminates during its bottom-up restructuring, is called the deepest i-safe (d-safe) node. The subtree rooted at the deepest i-safe (d-safe) node is termed the update scope (u-scope) of the insert or delete process.*

		process Y		
		$\rho - lock$	$\omega - lock$	$\xi - lock$
process X	$\rho - lock$	1	1	0
	$\omega - lock$	1	0	0
	$\xi - lock$	0	0	0

Table 2.1: Compatibility Matrix C of lock types.

As an illustration, consider then execution of a process $X = Insert(32, R)$, where R is the root of the B-tree in Figure 2.4. The deepest i-safe node is node B, the u-scope of X is the subtree rooted at B. Similarly for a delete process $Y = Delete(80, R)$ the d-safe node is C and the u-scope of Y is the subtree rooted at C.

Most of the concurrency control protocols discussed in the literature and in this paper rely on the explicit locking of nodes of the tree. As in a number of other schemes on concurrency control, we identify three lock types: a *read-lock* or shared-lock, a *warning-lock* and an *exclusive locks.* These are denoted by $\rho - lock$, $\omega - lock$, and $\xi - lock$ respectively. The compatibility matrix of the lock types is given below.

A process can hold any one of the lock types on a node p by issuing one of the following atomic operations: $\rho - lock(p)$, $\omega - lock(p)$, or $\xi - lock(p)$. The compatibility matrix shows what lock types can be held simultaneously on a node by two distinct processes. An entry C[x-type, y-type] = 1 specifies that the lock types x-type and y-type, are compatible. This implies that a process X can hold a lock of x-type while process Y holds a lock of y-type simultaneously on the same node. An entry C[x-type, y-type] = 0 signifies that the two locks x-type held by X and y-type held by Y are incompatible. Consequently, only one process can hold its corresponding lock type on the node while the other is blocked until the lock is released. Table 2 illustrates the following facts.

- Two or more processes can simultaneously hold $\rho - locks$ on a node. $\rho - locks$ have been called read or shared locks in the literature.
- A process can hold a $\rho - lock$ on a node on which another process holds an $\omega - lock$. However two $\omega - locks$ can not be held simultaneously on a node.

- A node locked with an $\xi - lock$ can not be locked by any other process with any lock type. This is referred to as an exclusive lock.

When a process requests a lock on a node, it is immediately granted the lock and continues to execute as long as granting the lock request will not violate the compatibility relationship. Otherwise it is suspended and made to wait. Any process that locks a node p in any lock mode is understood to release the lock eventually by issuing an *Unlock(p)* operation. Beside lock and unlock operations, a process can convert an $\omega - lock$ to a $\xi - lock$ and vice versa with the understanding that such conversion is made if the compatibility condition is not violated. We designate such operations as $Lock - Convert(p, \alpha - lock, \beta - lock)$ where $\alpha, \beta \in \{\omega, \xi\}$. The operation $Lock - Convert(p, \alpha, \beta)$ may be perceived as an atomic execution in sequence of two operations, the unlock of the $\alpha - lock$ on p followed immediately by a request for $\beta - lock$ on p. The process executing the lock conversion operation being given the highest priority of all the waiting processes with conflicting lock request.

Most of the lock based concurrency control techniques presented make use of the concept of *lock-coupling*. This is defined in a generalized form as follows.

Definition 2.5 *Let P be a process that traverses a graph G(V,E), where V is a set of nodes and E is a set of edges. Suppose P traverses a connected path* $\langle v_{j_1}, v_{j_2}, v_{j_3}, \ldots, v_{j_n} \rangle$, *by locking and unlocking nodes and suppose the lock requests are made on* $v_{j_1}, v_{j_2} \ldots v_{j_n}$ *in that order. Call a node useless if after it is locked, the process never uses the information at the node or never returns to it. Then the process is said to traverse the path using* lock-coupling *technique if after a node is locked all locks in the preceding useless nodes are immediately released.*

In the methods based on locking the lock types discussed earlier are used except for optimistic methods [15]. A method that combines locking and optimistic concurrency, termed a hybrid method, has been proposed by Lausen [19]. We survey some of the earlier methods of concurrency control on $B - trees$ in the next section. A similar but limited comparative discussion on concurrency methods on $B - trees$ has been presented in [16,6].

3 Related Earlier Works

Kwong and Wood [16] classified the various proposed locking methods of concurrency control on B-trees into two general categories which they called Type 1 and Type 2 solutions. In Type 1, the scope of an update process is prohibited from other update processes during the restructuring phase. However, a look-up process is allowed within the scope of the update process. In a Type 2 solution,

the scope of an update process may contain other update processes. The former is illustrated by the method proposed by Samedi [21] and also by Bayer and Schkolnick [3]. The latter is exemplified by the methods of Lehman and Yao [18] and by Segiv [22]. Biliris similarly, gave a number of properties by which concurrency control methods may be compared. These include the number of lock-types used by the protocol, the number of traversal of the B-tree by each process type etc.

The earliest solution to concurrency control on B-trees was presented in [21]. The solution used exclusive locks only i.e., $\xi - locks$ which may be implemented using P and V semaphores. A Look-up() process as it traverses the tree from the root to the leaf node, first locks the root. It subsequently locks the next node on the path and releases the lock on the parent. At most two locks are held simultaneously by the look-up process. An update process uses $\xi - locks$ as before locking each node on its path from the root to the leaf. However, whenever a node that is safe for update is encountered, the locks held on all the ancestor nodes are immediately released i.e., the update processes use the lock-coupling techniques.

An observation in the method by Samedi is that processes never overtake each other. If one considers each look-up and update process as comprising a single transaction, then this protocol satisfies all the conditions of the protocol for traversing hierarchical organized data items given by Silberschatz and Kedem [24]. Consequently the proof of correctness of the protocol by Samedi follows from the correctness proof of the protocol in [24] for hierarchically organized items. The main advantage of the solution by Samedi is its simplicity. Only one lock type is used. The main disadvantage is that it blocks more transaction from executing than is necessary, particularly where there are predominantly more look-up than update processes.

Bayer and Schkolnick proposed three methods for concurrency control on B-trees. Their schemes basically use the three lock types; $\rho-$, $\alpha-$, and $\xi - locks$. The $\alpha - lock$ in [3] corresponds to the $\omega - lock$ in the Table 2.1 of the previous section.

Solution 1:

Both a look-up process p_l, and an update process p_u apply lock-coupling technique using $\rho - locks$ and $\xi - locks$ respectively as they traverse the tree from the root to a leaf node. An update in a leaf node may be propagated up the tree but only to the deepest-safe node for which it holds a $\xi - lock$. This could be the root. The method is an improvement over the one prosed by Samedi in that multiple look-up processes can access the same node and are allowed to overtake one another. However look-up or other update processes are never allowed in the scope of an update process.

Solution 2:

The second solution of Bayer and Schkolnick relies on the fact that modifications to the upper level nodes of the tree are rare in comparison with look-ups. As such, both look-up and update processes apply lock-coupling with $\rho - locks$ during the top down traversal phase. An update-process then ξ-locks the leaf node to be modified. However, if an updater finds that the node is not safe the lock is released and the update is repeated as in Solution 1. In this solution look-ups and updaters in their read phase may exist in the scopes of each other but not when the scope of an update process is ξ-locked. The price paid in this approach is that an update process may traverses the tree thrice instead of twice.

Solution 3

The third solution makes use of three lock types, $\rho-, \alpha-$ and $\xi - locks$ with compatibility matrix as defined in Table 1 by replacing ω with α. A look-up process and an updater process both use lock-coupling during their top-down traversal with ρ-locks and α-locks respectively. Before restructuring its scope, an update process converts the α-locks to ξ-locks. As in solution 2, an update process may have to traverse the tree thrice.

Each of the above three solutions have certain advantages and disadvantages with respect to the number of traversal of the tree and the number of different lock types used depending on the mix of look-up and update processes. A generalized solution that combines all the three techniques was subsequently proposed. The interested reader may refer to [3] for details.

Using a 2-3-tree [1] as a special case of a B-tree, Ellis [9] proposed two methods for the concurrent operations on 2-3-trees. The first method of Ellis is similar to the third solution of Bayer and Schkolnick except that now look-up processes are allowed in the scope of an update process during the bottom-up restructuring phase. Look-up and updater processes traverse down the tree using lock-coupling techniques with ρ-locks and α-locks) respectively.

An update-process during the bottom-up restructuring phase does not exclusively lock its scope to prevent further look-ups from entering. Rather, only the nodes of the tree that are being modified are ξ-locked. These are the node itself, its left or right brother and its parent. Another idea incorporated in this scheme is that of inter-nodal concurrency whereby look-up and insert processes simultaneously access the same node. A look-up process reads the content of a node from left to right while an insert-process modifies the content from right to left. Such an inter-nodal concurrency was proposed by Lamport [17].

A second solution of Ellis called the pipeline algorithm permits multiple inserters in the scope of another inserter. Basically look-up and insert processes search down the 2-3-tree to the correct leaf node where the key must reside using $\rho - lock$. An insert process stores the key in the leaf node after excluding other insert processes from the node. Any restructuring is done bottom up in a pipeline manner using $\alpha - locks$ to enforce an inserter to proceed up the tree in a *follow-your-leader* fashion. In this manner, multiple updaters can search and restructure the tree along the same path.

A solution to the concurrency control problem on B-trees that does not require look-up processes to lock any node was proposed by Lehman and Yao [18]. An insert process uses $\xi - locks$ during the bottom up restructuring phase and holds at most three locks only. The protocol requires some modification to the basic B-tree to a structure called the $B^{link} - tree$. In this scheme all the nodes at each level are linked from left to right. Each node except the left most node and the root are pointed to by two link pointers. A pointer to it from a node to its immediate left on the same level, and another pointer from its parent node.

A look-up process is slightly modified from the usual search in a B-tree. During its traversal from the root to a leaf node, not only is the correct path to be followed determined by key comparisons in a single node but also in the right brother nodes that are linked to it. An insert-process first proceeds as look-up process without holding locks. The address of the first node arrived at on each level is pushed onto a stack. On encountering the leaf node, it is exclusively locked. If after storing the key the node is i-safe, the lock is immediately released. Otherwise the node is split by moving the right most half of its content to a new node. The new node is inserted as the immediate right brother in the horizontal chain. The process then pops a node from the stack and determines the correct parent node and position in the higher level node where the pointer to the newly created node must be inserted. The horizontal search for the correct position to store the key-pointer pair uses the lock-coupling technique. If the node updated, after insertion, is not safe, the splitting and linking process is repeat up the tree.

The process appears to admit more concurrent execution of look-up and insert process than any of the methods discussed previously. In addition it makes use of only one type of lock. The technique, however has a number of drawbacks.

1. It does not easily admit delete-processes. This is not a major problem if one considers that in a B^{link}-tree, as in B^{+}-tree, record deletions occur at the leaf node and the keys in the internal nodes only serve as separators and may be retained until two adjacent leaf nodes are merged. In the B^{link}-tree, if the utilization falls below an acceptable level, the whole tree is

locked and reorganized.

2. Searching to determine the correct pointer to follow may traverse very long chains of nodes at the same level.

3. The $B^{link} - tree$ actually violates the property of a $B - tree$ in that in an m-order tree, the number of keys in a node is allowed to fall below $\lceil m/2 \rceil - 1$.

4. Nodes that overflow cannot be rotated to spill over keys into left or right brother nodes that are not full in order to improve the storage utilization.

To avoid violating the properties of the $B - tree$ after deletion, Segiv [22] proposed running a separate compression process that may be invoked to run concurrently with other look-up, insert and delete processes. This solution is not an entirely satisfactory one. The compression process may be live-locked and since each delete process invokes a new compression process, the number of concurrently executing processes increase. This extra load could degrade the overall response times of a look-up or an update operation.

An alternative to the locking techniques in B-tree concurrency control is the optimistic concurrency method first presented by Kung and Robinson [15]. The assumption here is that in a sufficiently large B-tree, conflicts between look-up and update processes are rare, This is because processes will tend to access independent portions of the tree. In some cases the activities on the tree are predominantly simple search operations.

In the optimistic approach, each process carries out three phases of access to the data at the leaf nodes. The first phase involves reading the data items into its local workspace. This first phase is done without node locking. For a look-up process, this step completes its operations on the tree. For an update-process all updates are carried out on the local copies in the workspace. Before the update-process writes into the database, a validation step is conducted. If the validation succeeds, the updates are consolidated If the validation fails, the process is aborted and re-executed. The validation and write phases of process are carried out in mutually exclusive manner. The tree phases of a execution may be depicted as shown in Figure 3.1.

Read Phase: All reads are done into local copies of variables. Modification and writes are done on the local copies. The set of items read is called the *read-set.* The set of items to be written is called the *write-set.*

Validation Phase: This determines whether the transaction will cause a loss of integrity and the

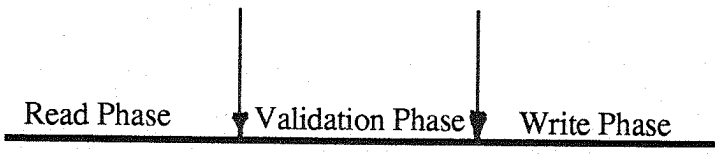

Figure 3.1: Three phases of an optimistic concurency control protocol

result of the updates in the data structure will still maintain the consistency of the data structure.

Write Phase: The updates are done on the actual data structure or database with the items in the write-set.

The validation phase of a transaction Q is evaluated as follows. Let Q' be a transaction that executed its global phase concurrently with the local phase of Q. The validation of Q succeeds if

$$\forall Q, readset(Q) \cap writeset(Q') = \phi.$$

Kung and Robinson applied the optimistic concurrency control method to B-trees and showed that the probability of conflicts, P_c satisfies

$$P_c < \frac{1}{n}(1 - \frac{2}{m+1}) \sum_{1 \leq i \leq h} (\frac{2m}{m+1})^{i-1}.$$

where m is the order of the B-tree, n is the number of leaf pages, i is the size of write-set of a transaction and h is the height of the tree ($h \leq 5$).

The optimistic concurrency control on B-trees is also applied in the prototype implementation of multi-user version of the PLAIN database where the underlying physical structure is a B-tree [12]. A preliminary comparative study made in the project between optimistic approach and locking schemes showed that the former is a good candidate for replacing the locking scheme in situations where the number of updates to the tree is relatively small compared to overall accesses made to the tree.

In the optimistic concurrency approach, conflicting operations cause aborts and restarts. When this is significantly high, starvation of a process may occur and as such locking is preferable.

This suggests that the rate of conflicting operations must be known before deciding on whether optimistic or locking method, should be used for concurrency control on a B-tree index in a multi user environment. Lausen [19] proposed an *integrated concurrency control* method to resolve the issue. In this integrated scheme, a transaction is identified as either an *O-type* (to use optimistic method) or an *l-type* (to use locking method) and executed accordingly. Determining whether a transaction should run either as an O-type or an l-type may be done according to its function, the range of key values or by an adaptive method.

Another approach to concurrency control in B-trees is the *side-branching* method of Kwong and Wood [16]. Like the third solution of Bayer and Schkolnick, the method uses three lock types, ρ-lock, ω-lock and ξ-lock with the same compatibility matrix as in Table 2.1. A look-up process applies lock-coupling technique using ρ-locks to get from the root to a leaf node.

An update-process in its search phase applies lock coupling with ω-lock. A the end of the search phase the scope of the update process is $\omega - locked$. Other look-up process can still be admitted into its scope. Suppose an update process is to execute insertion. During the restructuring phase, no exclusive locking of the nodes within the scope is done. Rather, the contents of the right half of a node to be split is copied into a newly created node. The set of neww nodes form a side branch of the nodes in the scope. The construction of the side branch of nodes proceeds bottom up until the deepest safe node is reached. This node is locked exclusively and modified. Beginning from the deepest safe node down to the leaf node, each of the nodes is cleared of the contents of the right half after being exclusively locked. During this phase, only one node is exclusively locked at each level. The major steps of the insert process may be summarized as follows:

- $\omega - lock$ the scope of the insert-process;
- construct the side branch of the $\omega - locked$ nodes;
- add the side-branch to the deepest safe node after converting its $\omega - lock$ to a $\xi - lock$.
- remove the appropriate half of the each node in the scope after driving off any look-up processes.

A delete-process follows a similar sequence of operations except that instead of splitting nodes in the scope, nodes are merged. The nodes whose content have been added to their left siblings are subsequently removed. In the restructuring phase of a delete process, restructuring may be terminated if rotation restores the balance properties of the B-tree. A similar rotation and key

redistribution may be incorporated in the insert algorithm. This is not explicitly stated in the algorithm given in [16]

A node is i-safe (d-safe) if an insert (delete) process determines that the number of keys $j < m-1$ ($j > \lceil m/2 \rceil - 1$). Consider the insert process. A node containing j keys can still be defined as being insertion safe for s insert processes if $j + s < m - 1$. This is the basic idea behind the sU-protocol of Biliris [6]. The protocol, in many respects, resembles the side-branching scheme of Kwong and Wood except that at most s update processes can be tolerated in the subtree rooted at p. The node p is considered update safe as long as it can be ascertained that the net effect of propagating updates at the lower level of p will still render it update safe.

In the method of Biliris, the compatibility matrix shown in Table 1 no longer applies. Rather the $\omega - lock$ is replaced by two types of locks called i-lock and d-lock. The i-locks are compatible with ρ-locks and other i-locks as long as $j + s < m - 1$. Similarly, d-locks are compatible with ρ-locks and other d-locks as long as $j - s > \lceil m/2 \rceil - 1$. A comparative discussion of concurrency control methods on B-trees is given in [6] as well.

4 Bottom-Up Exclusive Locking Method

The method may be considered as a variation of the third solution of Bayer and Schklonick combined with the B^{link} ideas of Lehman and Yao. Lookup processes can be admitted into the scope of an update process during the bottom-up restructuring phase. The main properties of this method are that:

1. the properties of the B-tree are maintained before and after each process manipulates the tree.

2. No independent compression process is invoked to restore the properties of the tree.

3. It is never necessary to lock the whole tree for the purpose of restoring the characteristics of the tree.

4. Inter-nodal concurrency is not required.

5. A look-up process holds at most two ρ-locks only.

6. At most h ω-locks and no more than three ξ-lock may be held by an update-process at any one time.

7. An update-process may traverse the tree at most twice.

8. Node rotations may be performed by update-processes in other to achieve very high load factors.

The model of computation, whether processes manipulate nodes of the tree directly in shared memory or on separate copies in a process work-space, is immaterial. We shall assume that the nodes of the tree are cached in a shared memory page frame and a Lock-Manager supervises the granting and denying of lock requests.

4.1 A Look-up Process

A look-up process applies lock-coupling technique using ρ-locks as it traverses the B-tree from the root to a leaf node. The following routines and auxiliary structures are assumed in the algorithm of this scheme. We shall use the terms *page* and *node* interchangeably.

BTreeHandle: A pointer to the memory resident file header of the B-tree structure. There are two classes of nodes. The leaf nodes that form the *sequence set* and the internal nodes that form the *index set.*

GetPage(pageAddr): A routine that returns a pointer to the page area of a page frame, where the node given by *pageAddr* resides. If the node is not in memory it is read into memory.

Probe(K, pagePtr): A routine that returns the address of the next lower level node to proceed to during the search for the target leaf page of key K. The node given by the pointer PagePtr is the current page being searched. PagePtr is assumed not to be NULL.

NULLPAGE: A designation for a null page address;

Algorithm for a Lookup Process

```
Look-up(K, BTreeHandle)
{
  CurrPgAddr ← BTreeHandle− >Root ;
  ρ-lock(CurrPgAddr);
  CurrPage ← GetPage(CurrPgAddr);
  while (CurrPage− >nodeType ≠ SeqSet) {
        NextPgAddr ← Probe(K, CurrPage);
        ρ−Lock(NextPgAddr);
        Unlock(CurrPage− >PageNo);
        CurrPage ← GetPage(NextPgAddr);
  }
```

```
  if (K ∈ CurrPage)
    response ← true;
  else
    response ← false;
  Unlock(CurrPage− >PageNo) ;
  return response;
}
```

Algorithm to Probe a Page

```
Probe (K, currNode);
{
  i ← 0; /* Number of children pointers stored in a node is S */
  pageAddr ← NULLPAGE;
  do {
     if (K ≤ currNode− > K_i) {
       pageAddr ← currNode− > p_i;
       break;
     }
     else {
          if (++i > S)
            if (currNode− >RLink ≠ NULLPAGE) {
              nextNodeAddr ← currNode− >RLink ;
              ρ-lock(nextNodeAddr);
              Unlock(CurrNode− >PageNo);
              currNode ← GetPage(nextNodeAddr);
              i ← 0;
            }
     }
  } while (pageAddr = NULLPAGE) ;
  return pageAddr;
} /* end Probe */
```

4.2 The Insert-Process

The insert process in the Bottom-Up Exclusive Locking method consists of two phases. A first phase which involves determining the leaf node into which the insertion must be made. This is carried out essentially as in the look-up procedure, except that ω-locks are used instead of ρ-locks. The second phase involves restructuring of the tree if necessary. This is carried out only if inserting into the leaf node causes an overflow.

The insert-process in this scheme combines the third solution of Bayer and Schklonick with some ideas of the $B^{link} - tree$ of Lehman and Yao. The main difference in our scheme is that the scope of the insert-process is $\omega - locked$ and look-up processes only are still allowed to enter the

scope of an insert process. The nodes at each level of the tree are double linked in both directions instead just one. Unlike the B^{link}-tree, a look-up process never probes more than two nodes on the same level. The restructuring is recursively done from the bottom up, one level at a time, along the path that is ω-locked.

Consider the insert process immediately after inserting into a leaf node. Let this node be denoted by ν. At this instant, the scope of the insert process, i.e., all the nodes along the path from the deepest safe node to node ν, is $\omega - locked$. Let the father of node ν be $\phi(\nu)$ and let the left and right brother nodes be denoted by $\alpha(\nu)$ and $\beta(\nu)$ respectively. If ν will overflow after an insertion, node rotation is first attempted to create room in ν. To do this we select either the left or the right brother node which ever contains a lesser number of keys below a pre-defined threshold H_α. Let this be denoted by $\beta(\nu)$. The nodes $\phi(\nu)$, ν and $\beta(\nu)$ are exclusively locked in that order Rotation involving these nodes and the key K, is then carried out. If this succeeds, all locks held by the insert process are immediately released.

If rotation is projected to be futile, it is not done. Rather, the node ν is split. For simplicity we assume that ν is split into two nodes with half of its content moved into a newly created node ν'. Before splitting ν, the ω-lock on ν is promoted to an exclusive lock and the node $\beta(\nu)$, if it exists is locked as well. It is not necessary to consider splitting two nodes into three here since rotations are allowed and this consequently increases the storage utilization considerably.

The separator key value of nodes ν and ν' is subsequently inserted into $\phi(\nu)$. Let this be denoted by k_ν. The lock on the node ν is released. On the next iteration the lock on $\phi(\nu)$ will be upgraded from an $\omega - lock$ to $\xi - lock$. At the instance of releasing the lock on ν any blocked look-up process prevented from accessing ν and has requested a ρ-lock eventually secures its lock on ν and continues. When a lock conversion from ω to ξ request is made, all pending ρ-locks are granted but subsequent ρ-lock requests are blocked.

When the insert process is granted a ξ-lock on $\phi(\nu)$, the pair of values k_ν and ν', are inserted into $\phi(\nu)$. The insertion into $\phi(\nu)$ is carried out as in the case of ν. If $\phi(\nu)$ is safe the insertion is done and the process terminates. Otherwise, the process determines if a rotation can be performed. The rotation is done by exclusively locking $\phi(\phi(\nu))$, $\phi(\nu)$ and one of $\alpha(\phi(\nu))$ or $\beta(\phi(\nu))$. Failing that, the node $\phi(\nu)$ is split.

This sequence of restructuring the tree from the bottom up is repeated until the deepest safe node is reached. The algorithm for the insert-process is given below. Some supporting data structures and routines utilized in the algorithm are given below.

Stack; A stack into which page addresses are pushed and popped out. The normal operations of Pop() and Push() are assumed.

i_safe(pagePtr): A boolean function that determines whether a memory resident page given by pagePtr is insertion safe.

minCapacity(LPage, RPage, H_α): A function that takes two page pointers, LPage and RPage, and returns the one which contains the lesser number of key and is no more than H_α of the page capacity.

Algorithm for an Insert-Process

```
Insert(K, RecPagePtr, BTreeHandle)
{
  CurrPgAddr ← BtreeHandle− >Root;
  ω-Lock(CurrPgAddr);
  CurrPage ← GetPage(CurrPgAddr);
  while (CurrPage− >nodeType ≠ SeqSet) {
        if i_safe(CurrPage) {
          while (!empty(Stack));
                Unlock(Pop(Stack));
        }
        Push(CurrPgAddr, Stack);
        NextPgAddr ← Probe(key, CurrPage);
        ω-Lock(NextPgAddr);
        CurrPgAddr ← NextPgAddr;
        CurrPage ← GetPage(CurrPgAddr);
  }
  /* Check for duplicate key insertion */
  if (K ∉ CurrPage) { /* Insert key */
    quit ← false;
    while (! quit) {
          if (! i_safe(CurrPage)) {
            if (CurrPage− >LLink ≠ NULLPAGE)
              LeftPage ← GetPage(CurrPage− >LLink);
            else
                LeftPage ← NULL;
            if (CurrPage− >RLink ≠ NULLPAGE)
              RightPage ← GetPage(CurrPage− >RLink);
            else
                RightPage ← NULL;
            BPage ← minCapacity(LeftPage, RightPage, Hα) ;
            LockConvert(ω, ξ, CurrPgAddr);
            if (BPage != NULL) { /* Rotation Possible */
              FcurrPgAddr ← Pop(Stack);
              LockConvert(ω, ξ, FcurrPgAddr);
```

```
            FcurrPage ← GetPage(FcurrPgAddr);
            ξ-Lock(Bpage− >PageNo);
            RotatePages(FcurrPage, CurrPage, BPage, K, RecPagePtr);
            Unlock(FcurrPgAddr);
            Unlock(BPage− >PageNo);
            quit ← true;
          }
          else {
                BPgAddr ← newPage();
                BPage ← GetPage(BPgAddr);
                SplitPage(CurrPage, BPage, K, RecPagePtr);
                FcurrPgAddr ← Pop(Stack) ;
                if (FcurrPgAddr ≠ NULLPAGE) {
                  Unlock(CurrPgAddr);
                  CurrPgAddr ← FcurrPgAddr;
                  CurrPage ← FcurrPage ;
                }
                else { /* Current Page must be a Root */
                     FcurrPgAddr ← newPage();
                     FcurrPage ← GetPage();
                     CopyClear(CurrPage,FcurrPage);
                     StoreRoot(CurrPage, FcurrPgAddr, K, RecPagePtr);
                     RecPagePtr− >LLink ← FcurrPgAddr;
                     quit ← true;
                }
          }
        } /* if ! i_safe */
        else {
              LockConvert(ω, ξ, CurrPgAddr);
              Store(CurrPage, K, RecPagePtr);
              quit ← true;
        } /* end i_safe */
    } /* endwhile */
  } /* end K ∉ CurrPage */
/* K ∈ CurrPage or tidy insertion process */
  Unlock(CurrPgAddr);
  while (!empty(Stack))
    Unlock(Pop(Stack));
} /* end insert */
```

4.3 Illustration of the Insert-Process

Consider two processes T_p and T_q that are attempting to insert keys into the B-tree. Assuming that T_p locks its scope which begins at node μ. Suppose process T_p enters a restructuring phase while other look-up processes T_k and T_j are accessing the tree. The situation may be depicted as

in Figure 4.1 where the nodes ν, $\beta(\nu)$ and $\phi(\nu)$ have just been locked by T_p. The process T_q has been blocked at node μ since the two $\omega - locks$, one by T_p and the other by T_q are incompatible. The look-up processes T_k and T_j applying lock-coupling with $\rho - locks$ are also blocked as they try to access node $\phi(\nu)$. We designate the pairs of locks of process T_j by ρ_j and ρ'_j. A lock not granted is shown waiting in a queue.

If we succeed in rotating keys from node ν into $\beta(\nu)$, the key to be inserted will end up in either ν or $\beta(\nu)$. No further restructuring is necessary and all the locks held by T_p are released. Assuming this was the case then the look-up processes and the insert process T_q proceed downward towards their target leaf nodes. For T_q, it places ω-locks on nodes in its scope. The state of affairs immediately following the release of locks by T_p may be shown as in Figure 4.2.

On the other hand if rotation could not be performed, the node ν would then be split. This causes a new node ν', to be created and double linked to ν. Half the keys in ν are moved into ν'. A copy of the highest key value in ν forms the separator key K_ν between ν and ν'. The values K_ν and ν' form the pair of values to be inserted in $\phi(\nu)$ in the next iteration.

The restructuring is recursively propagated up the tree. To do this, the ξ-lock on ν is released. Assuming that $\phi(\nu)$ is not the deepest safe node of T_p, then a request to convert the $\omega - lock$ on $\phi(\nu)$ is made. In the meantime, the look-up processes T_k and T_j succeed in securing their $\rho - locks$ on $\phi(\nu)$. The situation after the restructuring of the tree by process T_p is propagated one level up the tree is shown in Figure 4.3.

Assuming that the insert-process T_p succeeds in converting the $\omega - lock$ on $\phi(\nu)$ to $\xi - locks$. Rotation is first explored if it can be done. Failing that, splitting is done. These operations of rotation and splitting is continued until the deepest safe node is reached and process T_p unlocks all its locked nodes and terminates.

Note that all locks are requested in top down fashion. This order of lock acquisition guarantees deadlock free locking mechanism. To also allow concurrent sequential scan of the leaf nodes, we require transactions to seek locks at the leaf nodes in a left to right order. All transactions conducting sequential scan of the leaf nodes apply lock-coupling technique using $\rho - locks$ from left to right.

4.4 The Delete Process

A delete process in the BUX method behaves exactly as an insert-process except that keys are removed from nodes instead of being inserted. A node that violates the minimum required number

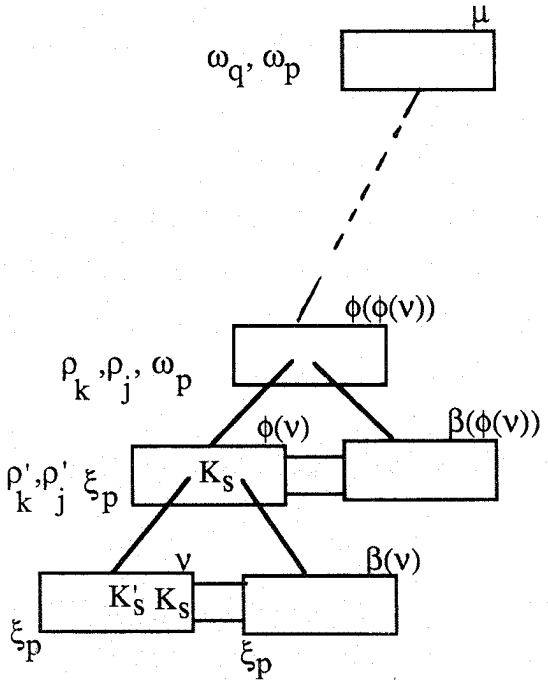

Figure 4.1: State of locks held by look-up transactions T_j and T_k and update transactions T_p and T_q.

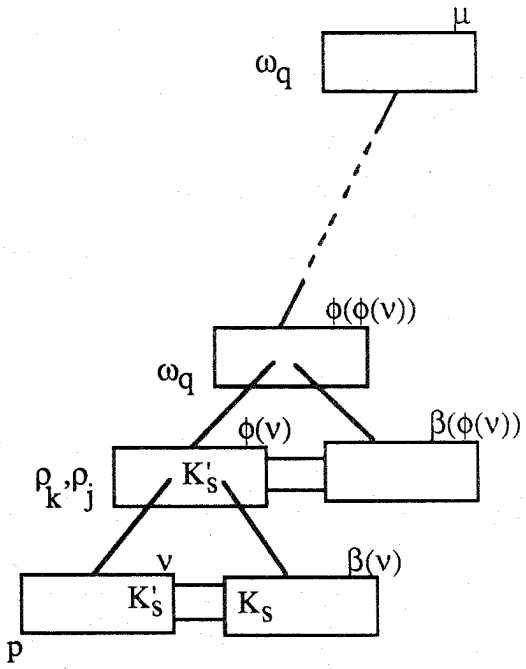

Figure 4.2: State locks after node rotations succeeds.

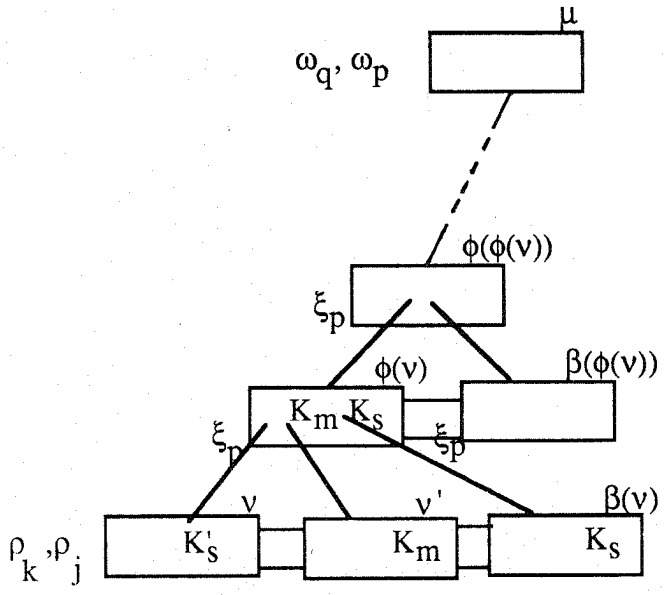

Figure 4.1:

State of locks held by look-up transactions T_j and T_k and update transactions T_p and T_q after node split

of key-pointer pair entries is reorganized by allowing its adjacent brother nodes that is more than half full to spill into it or be merged it.

If a key that acts as a separator is deleted from a leaf node, a copy of it in an internal node need not be immediately removed. A separator key is removed only when the nodes that it separates merge. Once we define the condition under which a node may be considered as delete safe (i.e., d_safe) an algorithm for a delete process, very much like that presented for insertion, can be formulated.

Definition 4.1 *In a $B-$tree of order m, a node is said to be delete safe if it contains s keys for $s \geq \lceil m/2 \rceil$ except if it is both a root and a leaf node.*

With this definition, the algorithm for a delete process may be stated as below. Some minor details can easily be inferred from the names of the routines that implement them. Note that the root node is never deleted as in the case of insertion. The initial condition consists of a root node which is set empty and is also leaf.

Algorithm for a Delete Process

```
Delete(K, BTreeHandle)
{
  CurrPgAddr ← BtreeHandle− >Root;
  ω-Lock(CurrPgAddr);
  CurrPage ← GetPage(CurrPgAddr);
  while (CurrPage− >nodeType ≠ SeqSet) {
        if d_safe(CurrPage) {
          while (!empty(Stack));
                Unlock(Pop(Stack));
        }
        Push(CurrPgAddr, Stack);
        NextPgAddr ← Probe(key, CurrPage);
        ω-Lock(NextPgAddr);
        CurrPgAddr ← NextPgAddr;
        CurrPage ← GetPage(CurrPgAddr);
  }
  /* Check for nonexistent key deletion */
  if (K ∈ CurrPage) { /* Delete key */
    quit ← false;
    RecPagePtr ← DeleteRec(K);
    while (! quit)) {
          if (! d_safe(CurrPage)) {
            if (CurrPage− >LLink ≠ NULLPAGE)
              LeftPage ← GetPage(CurrPage− >LLink);
            else
```

```
            LeftPage ← NULL;
          if (CurrPage− >RLink ≠ NULLPAGE)
            RightPage ← GetPage(CurrPage− >RLink);
          else
            RightPage ← NULL;
          BPage ← maxCapacity(LeftPage, RightPage, L_α) ;
          LockConvert(ω, ξ, CurrPgAddr);
          FcurrPgAddr ← Pop(Stack);
          if (FcurrPgAddr ≠ NULLPAGE) {
            LockConvert(ω, ξ, FcurrPgAddr);
            FcurrPage ← GetPage(FcurrPgAddr);
          }
          if (BPage ≠ NULL) { /* Rotation Possible */
            ξ-Lock(Bpage− >PageNo);
            RotatePages(FcurrPage, CurrPage, BPage, K, RecPagePtr);
            Unlock(FcurrPgAddr);
            Unlock(BPage− >PageNo);
            quit ← true;
          }
          else {
                BPage ← minCapacity(LeftPage, RightPage, L_α);
                if (BPage ≠ NULL) {
                  MergePages(FcurrPage, CurrPage, BPage, K, RecPagePtr);
                  LockConvert(ξ, ω, FcurrPgAddr) ;
                  Unlock(CurrPgAddr);
                  CurrPgAddr ← FcurrPgAddr;
                  CurrPage ← FcurrPage ;
                }
                else { /* Current Page must be a Root */
                    if (empty(CurrPage))
                      if (CurrPage− >nodeType ≠ SeqSet) {
                        RecPagePtr ← CurrPage− > p_0 ;
                        CopyIntoRoot(RecPagePtr− > p_0,CurrPgAddr);
                        DeletePage(RecPagePtr);
                      }
                    quit ← true;
                }
            }
          } /* end if ! d_safe */
          else {
            LockConvert(ω, ξ, CurrPgAddr);
            Remove(CurrPage, RecPagePtr, K);
            quit ← true;
          }
      } /* endwhile */
    } /* end Delete key K */
  /* key ∉ CurrPage tidy deletion process */
    Unlock(CurrPgAddr);
```

```
  while (! empty(Stack)
    Unlock(Pop(Stack));
}
```

4.5 Correctness Proof of the Bottom-Up ξ-locking Protocol

The main proof of correctness of the locking protocol is embodied in the proof of Theorem 4.1. Before expressing the main statement of the theorem, we have the following observations which follow intuitively from the protocols of the look-up and update processes.

Observation 4.1 *Lookup and update processes accessing the same path can overtake one another but an update process can never overtake another update process along the same path.*

This follows from the fact that the look-up and update processes use lock-coupling in their passage from the root to the leaf pages and the ω-lock and ρ-lock are compatible with one another while two ω-locks of update processes are incompatible.

Observation 4.2 *A look-up process can exist within the scope of another update process but an update process can never exist within the scope of another update process.*

Observation 4.3 *The manner of acquiring the locks in a top down and left to right order imposes a total ordering of the conflicting lock requests on the nodes of the tree.*

Theorem 4.1 *The bottom-up exclusive locking protocol on a B-tree is both deadlock and livelock free under fairness of lock acquisition.*

Proof

This is as a consequence of observation 4.3. A deadlock occurs if two processes each holds partial locks and each waits for the other to release its locks. Two update processes can never be deadlocked since the first process to lock a node that serves as the deepest safe node of its scope blocks all other updates processes at that node. Two look-up processes can never be deadlocked since their ρ-locks are compatible. We need then to consider the conflicts between look-up and update processes. Since all locks are acquired in a top-down or in left to right order, there is an strict linear ordering imposed in the manner of lock acquisition. Consequently deadlocks never occur. Under fairness in the granting of locks, all requests are eventually honoured and no process is ever locked out.

Theorem 4.2 *Any concurrently executing process that accesses a $B-tree$ in a consistent state using the bottom-up exclusive locking protocol, will leave it in a consistent state after terminating gracefully.*

Proof

We need to show that any schedule of concurrent execution of look-up and update processes is serializable. Note that with respect to traversal of the $B-tree$ index, a look-up or an update process may be perceived as a single distinct transaction destined to search, insert or delete only one key value from the leaf page.

Consider first the situation where all processes are look-ups. No modification is made to the tree. Hence every look-up transaction that reads its data from a leaf page, after traversing a consistent $B-tree$ and terminates gracefully, leaves the tree in a consistent state.

Similarly if we consider that all transactions are update transactions, only one transaction has exclusive access of the path when two or more transactions have conflicting access requests. Any restructuring of this path is carried out exclusive of all others. As a result, any two or more transactions that have conflicting accesses to a node in their common path, will execute serially in the order in which they first lock the common node.

The more interesting situation is the case in which we have a mix of look-up and update processes. Each access of the B-tree can be perceived as sub-transaction that accesses only one key value at the leaf node. By Theorem 4.1 we know that no deadlock or livelock occurs if each sub-transaction obeys the protocol. A schedule *of conflicting sub-transactions accessing the same leaf node* has an equivalent serial schedule given by the order in which locks on the node are acquired. We need only to show then that the path of a look-up or update sub-transaction in its progress from the root to its target leaf node is not modified by any update sub-transaction that is restructuring the tree bottom up.

Consider the subtree shown in Figure 4.4. Suppose a look-up sub-transaction T_l reads a node $\phi(\nu)$ and determines the next node to go to. The node to arrive at next is determined after T_l has secured a ρ-lock on $\phi(\nu)$. Let the next node be ν. Suppose an update transaction T_u, in its restructuring phase is intending to modify the node ν. The node ν can only be modified if T_u obtains a ξ-lock on ν. Suppose this is the case. After T_u modifies ν and releases the locks, T_l either correctly arrives at ν or follows the right link in ν to get to the correct node. The horizontal chase never goes beyond one extra node. If T_l gets a ρ-lock on ν before T_u locks it then the next node determined by T_l at node $\phi(\nu)$ is node ν which is correct.

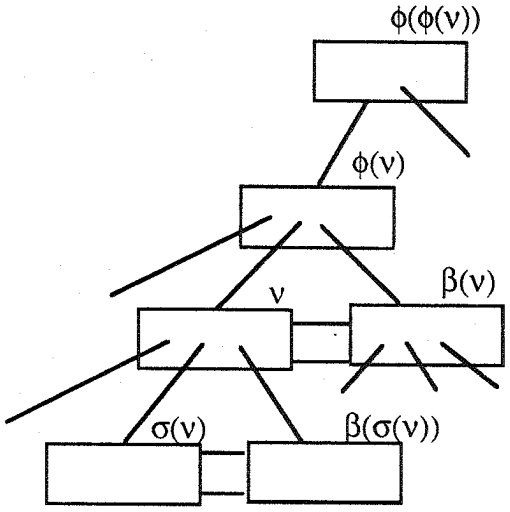

Figure 4.4:
Partial structure of a B-tree to illustrate the correct traversal of a look-up while node restructuring is in progress.

Suppose T_u determines that a rotation has to be done. This implies that T_u must acquire locks on $\phi(\nu)$, ν and $\beta(\nu)$ in that order. If T_l secures its lock on $\phi(\nu)$ before T_u does then the path read by T_l will be correct. On the other hand if T_u obtains a lock on $\phi(\nu)$ before T_l does, then T_l is forced to wait while still holding a lock on $\phi(\phi(\nu))$. After T_u completes modifying the contents of $\phi(\nu)$, ν and $\beta(\nu)$, it releases all its locks, and T_l can secure its lock on $\phi(\nu)$. The correct path then, is always determined and in this manner, every sub-transaction in its top-down traversal correctly arrives at its target leaf node. Consequently, every transaction that gracefully terminates leaves the tree with its proper characteristics. □

5 Top-Down Predictive Restructuring Method

5.1 Basic Ideas

In a paper *A Dichromatic Framework for Balanced Trees*, Guibas and Sedgewick [11] presented a uniform framework for the implementation and study of balanced tree algorithms. In particular they show how look-ups, insertions and deletions in 2-3-trees, 2-3-4-trees and AVL-trees, can be performed in a single top-down pass of the tree. For instance, in inserting into the tree, a full node encountered along the path during the top-down traversal may be split in anticipation that this could happen after an inserting into a leaf node. This avoids any subsequent bottom-up restructuring of the nodes. This idea may be carried over to B-trees. This is the basis of our second

approach to concurrent manipulation of $B-tree$ which we refer to as the *Top-Down Predictive Restructuring Method*. Three lock types are used as before: ρ-lock, ω-lock and a ξ-lock. These have the same compatibility matrix as shown in Table 2.1

5.2 Look-up, Insert and Delete Algorithms

A look-up process applies the lock-coupling technique with ρ-locks in exactly the same manner as in the bottom-up exclusive locking method discussed in the preceding section. An insert process uses lock-coupling with $\omega - locks$. However, in its traversal from the root to a destination node at the leaf, if a node $\phi(\nu)$ say, is $\omega - lockedr$, the lock on $\phi(\nu)$ is released only after the child node to be visited next, say ν, is $\omega - locked$. If the node ν is not insertion safe, (i.e., i_safe), an adjacent brother node $\beta(\nu)$ that is not full is determined if one exists. If such a node is found, the locks on $\phi(\nu)$ is converted to a $\xi - lock$, and node-rotation is performed after driving off all look-up processes in ν and $\beta(\nu)$. This ensures that the node ν is i_safe. If it is determined that rotation cannot be performed, the lock on $\phi(\nu)$ is converted to a $\xi - lock$ and the node ν is split after all readers are driven off it. In the case where horizontal scans of nodes are the same level is allowed, the nodes ν and $\beta(\nu)$ are ξ-locked during node rotation or node splitting.

After the successful completion of the operation that transforms node ν from an unsafe to insertion safe node, the node ν is $\omega - locked$ and the lock on $\phi(\nu)$ is released. This process continues until a leaf node is reached. The insertion is done at the leaf node after it is $\xi - locked$. The idea then is that an insert process in its top-down traversal of the B-tree, on encountering an unsafe node, predicts that a node could possibly split after inserting in the leaf node. Hence it takes the necessary precaution to create room in the node before continuing on.

The delete process behaves exactly as the insert process except that instead of node splitting mode merging is performed. A node that is predicted to be unsafe for deletion, i.e., one that would violate the condition of minimum content of a node, is merged with a similar brother node. Before node merging is performed, the process checks to ensure that rotation could not restore it to a d_safe node. The algorithms for the insert and delete processes are given below. The algorithm for the look-up process is the same as in the BUX method. The Probe() routine in the TDP method does not examine more than one node at a time. We still maintain left and right links in each node to double chain nodes at each level. This is done for consistency with the structure in the BUX method. Double chaining the nodes other than the leaf nodes is not necessary in the TDP method.

A number of auxiliary routines are assumed some of which require clarifications.

Probe(): The probe routine is as before except that in TDP, horizontal pointers are not traversed.

RotatePages(father, firstSon, secSon): This routine spills the content of a node (either firstSon or secSon) that has more keys into its adjacent brother node that has less number of keys. Before this is done, all look-up processes in the nodes firstSon,secSon are driven off.

SplitRoot(root,leftSon,rightSon): This routines splits the root node without changing the pointer to it. Essentially two new nodes are created as the sons of the root node. The left son gets the content of the left half and the right son gets the content of the right half. The root node retains the separator key.

MergePages(father, firstSon, secSon): This merges the contents of secSon into firstSon and deletes the other. The separator key is deleted from the father node. To do this, look-up processes in firstSon and secSon nodes are first driven off. if the father node becomes empty and it is not a leaf node then it must contain a pointer to only one child node. In this case its content is replaced by the content of the child node and the child node is deleted. If it is a leaf node it is retained.

```
TDP_Insert(K, BTreeHandle)
{
  CurrPgAddr ← BTreeHandle− >Root;
  ω − lock(CurrPgAddr);
  CurrPage ← GetPage(CurrPgAddr);
  if (! i_safe(CurrPage)) {
    LockConvert(ω, ξ, CurrPgAddr);
    LeftPage ← newPage();
    RightPage ← newPage();
    SplitRoot(CurrPage, LeftPage, RightPage);
    LockConvert(ξ, ω, CurrPgAddr);
  }
  while (CurrPage− >nodeType ≠ SeqSet) {
        LeftPage ← RightPage ← NULL ;
        FcurrPage ← CurrPage;
        CurrPgAddr ← Probe(K, CurrPage);
        ω − lock(CurrPgAddr);
        CurrPage ← Getpage(CurrPgAddr);
        if (! i_safe(CurrPage)) {
          LeftPgAddr ← CurrPage− >LLink);
          if (LeftPgAddr ≠ NULLPAGE) {
            ω-lock(LeftPgAddr);
            LeftPage ← GetPage(LeftPgAddr);
          }
          RightPgAddr ← CurrPage− >RLink;
```

```
        if (RightPgAddr ≠ NULLPAGE) {
          omega-lock(RightPgAddr);
          RightPage ← GetPage(RightPgAddr);
        }
        LockConvert(ω, ξ, FcurrPage− >PageNo);
        if (! RotatePages(FcurrPage, CurrPage, LeftPage, RightPage, H_α)
          SplitPage(FcurrPage, CurrPage, LeftPage, RightPage);
      }
      Unlock(FcurrPage− >PageNo);
      if (LeftPage ≠ NULL)
        Unlock(LeftPgAddr) ;
      if (RightPage ≠ NULL)
        Unlock(RightPgAddr) ;
  }
  LockConvert(ω, ξ, CurrPage− >PageNo);
  Store(K, CurrPage);
  UnLock(CurrPage− >PageNo);
}
```

```
TDP_Delete(K, BTreeHandle)
{
  CurrPgAddr ← BTreeHandle− >Root;
  ω-lock(CurrPgAddr);
  CurrPage ← GetPage(CurrPgAddr);
  while (CurrPage− >nodeType ≠ SeqSet) {
      LeftPgAddr ← RightPgAddr ← NULL;
      FcurrPage ← CurrPage;
      CurrPgAddr ← Probe(K, CurrPage);
      ω-lock (CurrPgAddr);
      CurrPage ← Getpage(CurrPgAddr);
      if (! d_safe(CurrPage)) {
        LeftPgAddr ← CurrPage− >LLink;
        if (LeftPgAddr ≠ NULLPAGE) {
          ω-lock(LeftPgAddr);
          LeftPage ← GetPage(LeftPgAddr);
        }
        RightPgAddr ← CurrPage− >RLink;
        if (RightPgAddr ≠ NULLPAGE) {
          ω-lock(RightPgAddr);
          RightPage ← GetPage(RightPgAddr);
        }
          LockConvert(ω, ξ, FcurrPage− >PageNo);
        if (! RotatePages(FcurrPage, CurrPage, LeftPage, Rightpage, L_α)
          MergePages(FcurrPage, CurrPage, LeftPage, RightPage, L_α);
      }
      Unlock(FcurrPage− >PageNo);
```

```
        if (LeftPage ≠ NULL)
          Unlock(LeftPgAddr);
        if (RightPage ≠ NULL)
          Unlock(RightPgAddr);
    } /* endwhile */
    LockConvert(ω, ξ, CurrPage− >PageNo);
    Remove(K, CurrPage);
    Unlock(CurrPage− >PageNo);
} /* end TDP_Delete */
```

5.3 Proof of Correctness of the TDP Protocol

It is essential to note that locks are acquired strictly in a top down fashion. If a sequential scan of the leaf nodes should be carried out, we require that the processes lock nodes at the same level in a left to right order. The following observations are based on the algorithms of the look-up, insert and delete processes.

Observation 5.1 *A look-up process and an update process accessing the same path may overtake one another but an update process may never overtake another update process.*

The fact that a $\rho - lock$ is compatible with both a $\rho - lock$ and an $\omega - lock$ allows a look process to share and possibly overtake another look-up or update process progressing along the same path. For the same reason an update process can overtake a look-up process. Since $\omega - locks$ are incompatible with one another, update processes sharing a common path are scheduled serially according to the order in which they are granted $\omega - locks$ on the first node of their common path.

Observation 5.2 *The insertion (deletion) of keys in a leaf node by an insert-process (delete-process) never requires that node restructuring operation be propagated up the tree.*

Consider the algorithm of the update process. Before a key is inserted into or deleted from a leaf node, the process ensures that the node is i_safe or d_safe as appropriate. Consequently, no restructuring of the tree is required after the key is inserted or deleted in a leaf node.

Theorem 5.1 *Under fairness of granting lock requests, a schedule of concurrently executing look-up and update processes accessing a $B-tree$ using the top-down predictive protocol is both deadlock and livelock free.*

Proof

Assuming that using the protocol a deadlock can occur. Then this implies that some process T_i

holds a lock on a node, say γ, and waits on the release of a lock on another node μ being held by some process T_j. At the same time process T_j waits for the release of the lock on node γ. By the protocol, all lock requests are made in a top-dowm fashion. In the case where processes scan nodes on the same level of the tree, lock requests are made in left to right order.

Without loss of generality let γ be an ancestor of node μ. If the nodes are at the same leaf level γ is assumed to be to the left of μ. From the order of lock acquisition as defined by the protocol, no lock can be held on node μ unless a lock was first held on γ except where ν is either the root node or the left most leaf node. Therefore process T_j could not have been holding a lock on μ and subsequently wait for the release of a lock on γ. This contradicts our initial assumption that deadlock does occur. Consequently a system of concurrent processes accessing the $B-tree$ using the top-down predictive protocol is deadlock free. □

Theorem 5.2 *If all processes accessing the $B-tree$ observe the Top-Down Predictive Locking protocol, then any process that accesses a a B-tree in a consistent state and terminates gracefully, leaves the tree in a consistent state.*

Proof

Consider a $B-tree$ in a consistent state in which all processes accessing the tree terminate gracefully. If all processes are look-up processes, then the theorem is trivially true since no modification is made to the tree.

If all processes are update-processes then the protocol enforces a serializable schedule. The equivalent serial schedule is given simply by the order in which the processes first lock a node in their common path, i.e., the root node.

The interesting case is when we have a mix of look-up and update processes. Suppose we focus our attention on one particular leaf node. The processes that arrive at that node define a strictly serial schedule. We need then to show that a path followed by a process T_r to an intended target leaf node is never corrupted by any update process T_u, ahead of it. Without loss of generality, let T_r be a look-up process.

Consider a subtree consisting of a node γ and its child node ν. Suppose T_r accesses the nodes γ and then ν but encounters an update process T_u about to split node ν. By the protocol, T_r requests a $\rho-lock$ on γ while T_u requests a $\xi-lock$. If T_r is granted its lock request on γ first then T_u will be blocked. The look-up process scans the node γ and determines the next node μ to proceed to. T_r secures a $\rho-lock$ on ν before releasing the lock on γ. Since $T-u$ modifies the node

ν only after all look-up processes admitted into node ν are driven off by allowing them to continue along their valid paths, T_r definitely continues along its correct path.

On the other hand if $T-u$ secures its $\xi - lock$ on γ first before T_r does then T_r will be blocked at node γ while the nodes γ and ν are restructured. The $\xi - lock$ on γ is released only after the node ν is $\omega - locked$ by T_u. When T_r obtains its lock on γ, the pointers to the lower level nodes would have been set correctly and T_r will determine the correct child node to proceed to next. Consequently, in either situations, i.e., whether T_r gets its lock on γ before T_u does or vice versa, the correct path of T_r is never corrupted. Hence the statement of the theorem. □

Given a schedule of execution of a set of transactions observing the top-down predictive protocol and accessing singly records from a file indexed by a $B - tree$. An equivalent serial schedule can be derived from topological sort of a graph constructed as follows. For each distinct transection, in the schedule, construct a distinct node. For any leaf node of the $B - tree$ at which two transaction T_i and T_j, arrive and issue conflicting locks, construct a directed edge according to the following simple rule. If T_i locks the node before T_j does place a directed edge from node T_i to node T_j otherwise place the edge from T_j to T_i. The graph obtained in this construction is acyclic and a topological sort generates an equivalent serial schedule.

6 Comparison of the Two Methods

At first glance it may not be immediately obvious that the two methods for concurrency control on B-trees are equivalent. They differ only in the manner in which the update processes restructure the tree. In the Bottom-Up Exclusive Locking protocol, an insert-process that inserted a key into the leaf node and caused the node to overflow, does the restructuring of the tree. In the Top-Down Predictive method, the insert-process after generating unsafe nodes, leaves the responsibility of restructuring the tree to the update-process that comes after it.

There are some minor differences between the two in the implementation details. The BUX method requires that we link all the nodes at the same level. This is necessary to allow look-up processes to proceed along a correct path as they bypass update processes in their bottom-up restructuring phase. The horizontal link may be discarded at the cost of a less dense tree if we redefine the conditions for a node to be i_safe and d_safe in the BUX method.

In both methods however, update processes avoid blocking look-up processes as much as possible. Update processes block other update-processes sharing the same path. Concurrent updates can still be conducted in the tree as long as they affect different leaf nodes in the. In this respect,

the TDP method tolerates more concurrent activity than the BUX method since the TDP method allows other processes to be in the scope of another update process while in the BUX method only look-up processes can be in the scope of another update process.

In comparison to earlier proposed methods for concurrency control on B-trees, the two methods enjoy most of the characteristics required to tolerate high concurrency. The BUX method makes use of strategies proposed in three earlier methods: the third solution of Bayer and Schkolnick [3], the linked B-tree of Lehman and Yao [18].

Comparing the BUX method to the third solution of Bayer and Schkolnick, our solutions avoid the problem of traversing the tree thrice. The method of Lehman and Yao is elegant in that look-up processes do not lock nodes. The price paid for this is in maintaining the B-tree property. This is occasionally violated and the system must then be put in a quiescent state from time to time to allow for total reorganization of the B-tree. Our methods ensure that the properties of the B-tree are maintained at all times. In the BUX method, nodes at the same level of the tree are linked in order that look-up processes follow the correct path. However no more than two nodes at the same level will be traversed by a look-up process. This is not the same in the B^{link} tree. In a B^{link}-tree, a look-up process can follow long chains of nodes before locating the node at the next level to proceed to.

One concern of our schemes is the operation of driving off look-up processes from a node that is about to be modified. This may be implemented by an exclusive lock that is followed immediately by an Unlock. This implies that a node may be $\xi - locked$ twice. The overhead incurred in driving off look-up processes this way is more than compensated for by the simplicity of the algorithms particularly in the case of the TDP method. On the other hand the nodes may be exclusively locked for the entire duration of node rotation, splitting or merging.

It is important to note that we make a clear distinction between the external view of a transaction the transactions operations on the B-tree. The external transaction may be viewed as being composed of a number of internal sub-transactions. Each sub-transaction involves only one operation of look-up, insert or delete of a single key in the B-tree. To distinguish these, from the external view of a transaction, we have used the term *process* when referring to the operation on a B-tree. The external transaction only sees the accesses to the target records required and not the operation on the B-tree index by which the target records were located. For all we know the B-tree could have been replaced by an oracle that magical locates the records. The external view of the transactions is relevant to the overall concurrency control on the target records. Consequently if a

transaction is to update a number of records, then from the view of an external transaction either all the records are successfully updated or none at all. How the records were accessed is immaterial.

Having accessed the records, either locking or optimistic concurrency control method may be applied for consistent update of the records. It is conceivable then to apply a locking mechanism for the B-tree index while optimistic concurrency control is used for the external transaction. It is necessary to coordinate the commit or abort actions of the external transactions with updates made to the B-tree nodes. We do not address this problem and other related problems of resiliency and recovery of B-trees. Our concern for now is in showing a simple but efficient means of concurrent manipulation of a B-tree index.

The methods proposed respond appropriately to the mix of look-up and update processes. If the internal transactions are predominantly look-ups, then the schemes hardly block any transactions. If they are predominantly update operations, then the transaction accessing the same paths are executed serially. Since in a B-tree, the nodes of intense activities are at the leaf nodes, most of the upper level nodes would be free from being exclusively locked for most of the time.

A detailed discussion of a number of concurrency control methods have been presented in section 3. Tables 6.1 and 6.2 below summarize some of the properties of the locking schemes for concurrency control on B-tree indexes.

7 Conclusion

We have presented a survey of some earlier concurrency control protocols on B-trees. Three general classes of protocols exist: locking schemes, optimistic schemes and hybrid schemes. The extent to which these methods tolerate simultaneous accesses to the tree vary considerable depending on the mix of user transactions. Of the schemes so far proposed, a considerable proportion of them are based on locking. Our schemes follow the same trend. A summary of the characteristics of the different locking protocols is given in the preceding section.

Two new techniques on concurrent manipulation of B-trees have been presented. The bottom-up exclusive locking method (BUX), and the top-down predictive method (TDP). The two methods are based essentially on the same principles but differ with respect to the time for restructuring the tree after an insertion or deletion made. In the BUX method, restructuring is done after the leaf node is updated while in the TDP, the restructuring is done before the update is made to the leaf node. The main virtues of the new schemes are that:

- the algorithms are simple to implement;

Proposed Method	No Lock-types	Max. #Nodes Locked by LP	Max #Nodes locked by UP $\rho-, \omega-, \xi-$	Max. #Traversal. of Index Nodes
Samedi, 1976	1	2	0, 0, 1	2
Bayer and Schkolnick, 1977	2	2	0, 0, h	2
	2	2	0, 0, h	3
	3	2	0, h, h	3
Ellis, 1980	3	2	0, h, 1	2
	4	2	2, 2, 2	2
Lehman and Yao, 1981	1	0	0, 0, 3	2
Kwong and Wood, 1982	3	2	0, h, 3	3
Segiv, 1985	1	0	0, 0, 1	3
Biliris, 1986	4	2	0, h, 1	3
Otoo, 1989	3	2	0, h, 3	2
	3	2	0, 2, 3	1

Table 6.1: Summary characteristics of locking protocols for B-trees.

Proposed Method	Overtaking LP/LP, LP/UP, UP/UP	LP's in Scope of UP?	UP's in Scope of UP?	Node Rotation Possible?
Samedi, 1976	N, N, N	N	N	Y
Bayer and Schkolnick, 1977	Y, N, N	Y	N	Y
	Y, Y, Y	Y	N	Y
	Y, Y, N	Y	N	Y
Ellis, 1980	Y, Y, Y	Y	Y	N
	Y, Y, Y	Y	Y	N
Lehman and Yao, 1981	Y, Y, Y	Y	Y	N
Kwong and Wood, 1982	Y, Y, N	Y	N	Y
Segiv, 1985	Y, Y, Y	Y	Y	Y
Biliris, 1986	Y, Y, Y	Y	Y	Y
Otoo, 1989	Y, Y, N	Y	N	Y
	Y, Y, N	Y	Y	Y

Table 6.2: Summary characteristics of locking protocols for B-trees. *Cont.*

- the tree properties are maintained before and after each look-up, insert or delete operation.
- they are free from deadlocks and livelocks.

We have not addressed the problem of B-tree recovery in this paper. We assume that each operation on the B-tree terminates gracefully. However, if the external transaction fails or aborts after the tree is updated, some precaution must be taken to ensure that such failures or aborts do not affect the consistency of the B-tree. An approach to resolve this problem involves maintaining transaction logs and applying *careful pages replacement* policy during writing of the pages of the B-tree onto disk.

Suppose we project an application environment in which an external transaction accesses a number of records indexed by a B-tree. A substantial number of these accesses are look-ups. We could use optimistic concurrency control for the external transactions and use TDP method for concurrency control on the B-tree index. Each external transaction T_i, executes in three phases: a read phase, a validation phase and a write phase. The reads of most transactions may be executed in parallel. After reading the records, each transaction T_i, generates its write-set and then enters the validation phase. Only after the transaction passes the validation test will the update to the B-tree index be applied. Other combinations such as using locking mechanisms for both the external transactions and for the B-tree nodes are also possible. Recovery after failure can be easily done with the aid of the transaction logs. Details of the recovery process is the subject of further work.

Acknowledgement

This work is supported in part by the Natural Science and Engineering Research Council of Canada under grant No 8102-02 and GR-5 grant from Carleton University.

References

[1] Aho, A., Hopcroft, J. and Ullman, J. The design and analysis of algorithms. *Addison-Wesley Publ. Reading Mass. (1974).*

[2] Bayer, R. and McCreight, E. Organization and maintenance of large ordered indexes *Acta Informatica 1, (1972), 173 - 189.*

[3] Bayer, R. and Schkolnick, M. Concurrency of operations in B-trees. *Acta Informatica, 9, 1 (1977), 1 - 21.*

[4] Bayer, R. and Unterauer, K. Prefix B-Tree. *ACM Trans. on Database Syst. 2, 1 (Mar. 1977), 11 - 26.*

[5] Bernstein, P. Hadzilacos, V. and Goodman, N. Concurrency control and recovery in database systems. *Addison-Wesley Publ., Reading, Mass, (1987)*

[6] Biliris, A. A Comparative study of concurrency control methods in B-trees. *Lecture Notes in Computer Science - VLSI Algorithms and Architectures, Proc. Aegean Workshop on Computing, Loutraki, Greece, (Jul. 1986), 305 - 316.*

[7] Comer, D. The ubiquitous B-Tree. *ACM Comput.Surveys, 11, 2 (Jun. 1979), 121 - 137.*

[8] Ellis, C. S. Concurrent search and insertion in AVL trees. *I.E.E.E. Trans. Comput., C-29 (1980), 811 - 817.*

[9] Ellis, C. S. Concurrent search and insertion in 2-3 trees. *Acta Infromatica, 14, 1 (1980), 63 - 86.*

[10] Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L. The notion of consistency and predicate locks in a database system. *Comm. of the ACM 19, 11 (Dec. 1976), 624 - 633.*

[11] Guibas, L. J. and Sedgewick, R. A dichromatic framework for balanced trees. *Proc. 19th Annual Symp. on Foundations of Computer Sc. (1978), 8 - 21.*

[12] Kersten, M. L. and Tebra, H. Application of an optimistic cocurrency control. *Software- Practice and Experience, 14, 2 (Feb. 1984), 153 - 168.*

[13] Knuth, D. E. The art of computer programming, Vol 3: Sorting and Searching. *Addison-Wesley Publ., Reading, Mass. (1973).*

[14] Kung, H. T. and Lehman, P.L. Concurrent manipulation of binary search trees. *ACM Trans. on Database Syst. 5, (1980), 354 - 382.*

[15] Kung, H. T. and Robinson, J. T. On optimistic concurrency control. *ACM Trans. on Database Syst. 6, 2 (Jun. 1981), 213 - 226.*

[16] Kwong, Y. S. and Wood, D. New method for concurrency in B-tree. *I.E.E.E Trans. on Software Eng. SE-8, 3 (1982), 211 - 222.*

[17] Lamport, L. Concurrent reading and writing. *Comm. ACM 22, 11 (1977), 806 - 811.*

[18] Lehman, P. and Yao, S. B. Efficient locking for concurrent operations on B-trees *ACM Trans. on Database Syst. 6, 4 (1981), 650 - 670.*

[19] Lausen, G. Integrated Concurrency Control in Shared B-trees. *Computing, 33, 13 (1984), 13 - 26.*

[20] Manber, U. and Ladner, R. E. Concurrency control in a dynamic search structure. *ACM Trans. Database Syst. 9 (1984), 439 - 455.*

[21] Samedi, B. B-Trees in systems with multiple users. *Inform. Process. Lett. 5, 4 (1976), 107 - 112.*

[22] Segiv, Y. Concurrent operations on B^*-tree with overtaking. *Proc. 4th ACM SIGACT/SIGMOD Symp. on Principles of Database Syst. Portland, Oregon (1985), 28 - 37.*

[23] Shasha,D. and Goodman, N. Concurrent search structure algorithm. *ACM Trans. on Datababase Syst., 13, 1 9 (Mar. 1988), 53 - 90.*

[24] Silberschatz, A. and Kedem, Z. Consistency in hierarchical database systems *J. ACM, 27, 1 (Jan. 1980), 72 - 80.*

[25] Silberschatz, A. and Kedem, Z. A family of locking protocols for distributed systems that are modeled by directed graphs. *IEEE Trans. on Soft. Eng. SE-8, 6 (Nov. 1982), 558 - 562.*

School of Computer Science, Carleton University
Bibliography of Technical Reports

SCS-TR-127 **On the Packet Complexity of Distributed Selection**
A. Negro, N. Santoro and J. Urrutia, November 1987.

SCS-TR-128 **Efficient Support for Object Mutation and Transparent Forwarding**
D.A. Thomas, W.R. LaLonde and J. Duimovich, November 1987.

SCS-TR-129 **Eva: An Event Driven Framework for Building User Interfaces in Smalltalk**
Jeff McAffer and Dave Thomas, November 1987.

SCS-TR-130 **Application Frameworks: Experience with MacApp**
Out of print
John R. Pugh and Cefee Leung, December 1987.
Available in an abridged version in the Proceedings of the Nineteenth ACM SIGSCE Technical Symposium, February 1988, Atlanta, Georgia.

SCS-TR-131 **An Efficient Window Based System Based on Constraints**
Danny Epstein and Wilf R. LaLonde, March 1988.

SCS-TR-132 **Building a Backtracking Facility in Smalltalk Without Kernel Support**
See Third International Conference on OOPSLA, San Diego, Calif., Sept. '88.
Wilf R. LaLonde and Mark Van Gulik, March 1988.

SCS-TR-133 **NARM: The Design of a Neural Robot Arm Controller**
Daryl H. Graf and Wilf R. LaLonde , April 1988.

SCS-TR-134 **Separating a Polyhedron by One Translation from a Set of Obstacles**
Otto Nurmi and Jörg-R. Sack, December 1987.

SCS-TR-135 **An Optimal VLSI Dictionary Machine for Hypercube Architectures**
Frank Dehne and Nicola Santoro, April 1988.

SCS-TR-136 **Optimal Visibility Algorithms for Binary Images on the Hypercube**
Frank Dehne, Quoc T. Pham and Ivan Stojmenovic, April 1988.

SCS-TR-137 **An Efficient Computational Geometry Method for Detecting Dotted Lines in Noisy Images**
F. Dehne and L. Ficocelli, May 1988.

SCS-TR-138 **On Generating Random Permutations with Arbitrary Distributions**
B. J. Oommen and D.T.H. Ng, June 1988.

SCS-TR-139 **The Theory and Application of Uni-Dimensional Random Races With Probabilistic Handicaps**
D.T.H. Ng, B.J. Oommen and E.R. Hansen, June 1988.

SCS-TR-140 **Computing the Configuration Space of a Robot on a Mesh-of-Processors**
F. Dehne, A.-L. Hassenklover and J.-R. Sack, June 1988.

SCS-TR-141 **Graphically Defining Simulation Models of Concurrent Systems**
H. Glenn Brauen and John Neilson, September 1988

SCS-TR-142 **An Algorithm for Distributed Mutual Exclusion on Arbitrary Networks**
H. Glenn Brauen and John E. Neilson, September 1988

SCS-TR-143 to 146 are unavailable.

SCS-TR-147 **On Transparently Modifying Users' Query Distributions**
B.J. Oommen and D.T.H. Ng, November 1988

SCS-TR-148 **An O(N Log N) Algorithm for Computing a Link Center in a Simple Polygon**
H.N. Djidjev, A. Lingas and J.-R. Sack, July 1988
Available in STACS 89, 6th Annual Symposium on Theoretical Aspects of Computer Science, Paderborn, FRG, February 16-18, 1989, Lecture Notes in Computer Science, Springer-Verlag No. 349

SCS-TR-149 **Smallscript: A User Programmable Framework Based on Smalltalk and Postscript**
Kevin Haaland and Dave Thomas, November 1988

SCS-TR-150 **A General Design Methodology for Dictionary Machines**
Frank Dehne and Nicola Santoro, February 1989

SCS-TR-151 **On Doubly Linked List ReOrganizing Heuristics**
D.T.H. Ng and B. John Oommen, February 1989

SCS-TR-152 **Implementing Data Structures on a Hypercube Multiprocessor, and Applications in Parallel Computational Geometry**
Frank Dehne and Andrew Rau-Chaplin, March 1989

SCS-TR-153 **The Use of Chi-Squared Statistics in Determining Dependence Trees**
R.S. Valiveti and B.J. Oommen, March 1989

SCS-TR-154 **Ideal List Organization for Stationary Environments**
B. John Oommen and David T.H. Ng, March 1989

SCS-TR-155 **Hot-Spot Contention in Binary Hypercube Networks**
Sivarama P. Dandamudi and Derek L. Eager, April 89

SCS-TR-156 **Some Issues in Hierarchical Interconnection Network Design**
Sivarama P. Dandamudi and Derek L. Eager, April 1989

SCS-TR-157 **Discretized Pursuit Linear Reward-Inaction Automata**
B.J. Oommen and Joseph K. Lanctot, April 1989

SCS-TR-158 **Parallel Fractional Cascading on a Hypercube Multiprocessor**
Frank Dehne, Afonso Ferreira and Andrew Rau-Chaplin, May 1989

SCS-TR-159 **Epsilon-Optimal Stubborn Learning Mechanisms**
J.P.R. Christensen and B.J. Oommen, June 1989

SCS-TR-160 **Disassembling Two-Dimensional Composite Parts Via Translations**
Doron Nussbaum and Jörg-R. Sack, June 1989

SCR-TR-161 (revised) **Recognizing Sources of Random Strings**
R.S. Valiveti and B.J. Oommen, January 1990
Revised version of SCS-TR-161 "On the Data Analysis of Random Permutations and its Application to Source Recognition", published June 1989

SCS-TR-162 **An Adaptive Learning Solution to the Keyboard Optimization Problem**
B.J. Oommen, R.S. Valiveti and J. Zgierski, October 1989

SCS-TR-163 **Finding a Central Link Segment of a Simple Polygon in O(N Log N) Time**
L.G. Alexandrov, H.N. Djidjev, J.-R. Sack, October 1989

SCS-TR-164 **A Survey of Algorithms for Handling Permutation Groups**
M.D. Atkinson, January 1990

SCS-TR-165 **Key Exchange Using Chebychev Polynomials**
M.D. Atkinson and Vincenzo Acciaro, January 1990

SCS-TR-166 **Efficient Concurrency Control Protocols for B-tree Indexes**
Ekow J. Otoo, January 1990

SCS-TR-167 **A Hierarchical Stochastic Automaton Solution to the Object Partitioning Problem**
B.J. Oommen, January 1990

SCS-TR-168 **Adaptive List Organizing for Non-stationary Query Distributions. Part I: The Move-to-Front Rule**
R.S. Valiveti and B.J. Oommen, January 1990