Concurrency in B-Trees and Tries: Search and Insert

Ian Spencer Garton
School of Computer Science
McGill University, Montreal
September 2000

A Thesis Submitted to the Faculty of Graduate Studies and
Research in partial fulfillment of the requirements of the degree of
Master of Science in Computer Science

Abstract

Multiuser database systems require concurrency control in order to perform correctly. B-trees have become the standard data structure for storing indices that aid in data retrieval and there have been many algorithms published to enable concurrent operations for B-trees. Tries are another data structure useful for storing index data, particularly for text and spatial databases. Significant data compression can be achieved by using a trie to store index values. However, there have been no algorithms published to support concurrent trie operations.

We present algorithms that enable concurrent searches and inserts for tries with pointerless representation. We also measure the performance of our algorithms and compare with that of the best B-tree algorithms. In order to measure trie concurrency, we survey a number of studies that have been made for B-tree concurrency. Using these published studies, we build a simulation model to measure the concurrency of our algorithms.

Résumé

Pour fonctionner correctement, la concurrence doit être contrôlée dans les systèmes de gestion de base de données multi-utilisateur. Les B-arbres sont devenus la structure de données standard pour sauvegarder les indexes qui assistent dans la récupération des données. Par conséquent, beaucoup d'algorithmes publiés traitent des opérations concurrentes sur des B-arbres. Dans le même temps, les *tries* sont une autre structure de données particulièrement utiles pour la sauvegarde d'indexes, et cela dans le contexte des base de données textuelles et spatiales. Un taux de compression significatif peut être obtenu en utilisant un trie pour stocker des indexes. Cependant, aucun algorithme traitant des opérations concurrentes sur des tries n'a été publié jusqu'à présent.

Nous présentons un algorithme qui permet les insertions et les recherches concurrentes sur des tries dont la reprsentation n'utilise pas de pointeurs. De plus, nous mesurons les performances de notre algorithme et les comparons avec les meilleurs algorithmes traitant des B-arbres. Pour mesurer la concurrence sur les tries, on a examiné un certain nombre d'études qui portent sur la concurrence des B-arbres. En se basant sur ces publications, on a construit un modèle de simulation pour mesurer la concurrence de nos algorithmes.

Acknowledgements

First and foremost, I wish to express my gratitude to my supervisor, Professor Tim H. Merrett, for providing a tremendous wealth of knowledge, experience, and encouragement. He contributed a lot of time and effort to help me and I really appreciate that. In addition, his generous offer to employ me as lab manager for the ALDAT Lab was instrumental in providing financial support and some insight into lab operations.

Many thanks go to a great friend, Chrislain Razafimahefa, for translating the abstract into French. I am grateful to friend and officemate, Sergei Savchenko, for his helpful advice and extremely interesting conversations. I am also grateful to Danielle Azar for her feedback and to Steve Robbins for his assistance. The secretaries at the School of Computer Science were also very helpful, especially Franca Cianci, Lise Minogue, and Lucy St. James.

I thank friends and family members for their support and encouragement. Tallman Nkgau and Chrislain Razafimahefa are great friends whose company made for many happy memories. My grandparents, Ralph and Barbara Garton and Herbert and Irene Franson, were sorely missed during my stay in Montreal. I appreciate tremendously the love, support, and encouragement they have provided throughout my life.

Finally, I express my profound gratitute to my parents, Robert and Janice Garton, and my sister, Angela Garton, for the endless love, support, and encouragement they have provided throughout my entire life. Without their guidance and teachings, this thesis would not have been possible. I dedicate this thesis to them.

To Bob, Janice, and Angie

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Concurrency control is the act of ensuring that concurrent operations do not interfere with one another and cause incorrect results. Concurrent data structures are used most often in database systems. Such data structures are useful in multiuser applications; such as, banking, ticket reservation, point of sale, inventory management, billing, and communications. This thesis studies concurrency control techniques for two popular data structures: B-trees and tries.

1.1 B-Tree Preliminaries

The B-tree was introduced in 1972 by Bayer and McCreight [BM72] and has since become the standard data structure for implementing indices in a database management system. Comer [Com79] has written a survey about B-trees and their variations. The variation known as the B^+-tree by Wedekind [Wed74] is popular because it is easier to implement and likely to be smaller than the B-tree. The main difference between the B^+-tree and the B-tree is that all records in the B^+-tree are stored at the leaf level. Another popular variation is the B*-tree by Knuth [Knu73] in which all nodes are at least 2/3 full and all records are stored at the leaf level. When using the term "B-tree" in this chapter and Chapter 2, we are referring to the B^+-tree.

1.1.1 B-Tree Description

A B-tree is a balanced search tree in which each path from root to leaf has the same height h, where height is measured in terms of node levels. A B-tree consists of nodes and links (or pointers) from one node to another. A parent node n located on node level ℓ_n contains pointers to children that are on node level $\ell_n + 1$. The root of the tree is the node that has no parents and is located on level 1. The leaves are the nodes that have no children and are located on level h.

In addition to pointers, each node contains either keys or separators. Separators are stored in non-leaf nodes and define a search path from root to leaf for a given key value. Keys are stored in the leaf nodes and imply that the associated information for the key value exists in the index. For simplicity, we will refer to separators as keys. The keys and pointers within a node are arranged in the following sequence: $\langle P_0, K_1, P_1, K_2, \ldots, K_x, P_x \rangle$ where P_i is a pointer and K_i is a key.

Within each node, keys are stored in ascending order. Keys create a search path from root to leaf by indicating the correct pointer and child to select in order to reach the correct information associated with the key value. For any non-leaf node n containing the sequence $\langle \ldots K_i, P_i, K_{i+1} \ldots \rangle$, the child n' pointed to by P_i contains only key values v such that $K_i < v \leq K_{i+1}$. Thus, the subtree rooted at n' contains only key values v such that $K_i < v \leq K_{i+1}$.

Each (key, pointer) pair in a node is called an entry. A tree parameter k controls the size of the tree nodes. Each node in the B-tree has has at most $2k$ entries. Every node, except the root, has at least k entries. The root has at least 1 entry (i.e. 2 children). Such a B-tree is said to be of order k. B-tree node size is also specified by *fanout*, which is the maximum number of entries each node may contain. A marker "M" may be stored in a node in place of P_0 to indicate that it is a leaf. In such a case, the information for key value K_i is located by following pointer P_i instead of P_{i-1}. Figure 1.1 shows a portion of a B-tree.

The B-tree index is stored on disks which are partitioned into pages of fixed size. Each node is stored on its own page on disk and pages are the smallest unit in which processes read and write information. Hence, later in the thesis, we will refer to pages instead of the logical nodes of the tree.

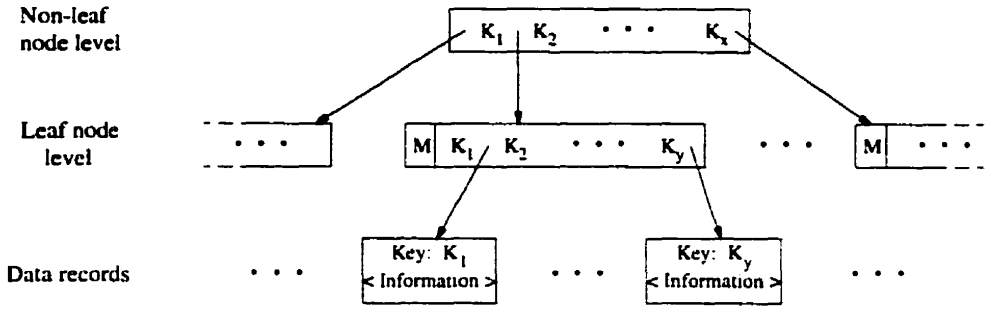


Figure 1.1: B-tree nodes

1.1.2 B-Tree Operations

All basic B-tree operations (search, insert or append, and delete) start at the root and traverse the tree down to the leaves. At each node, operations search for the minimal key that is greater than or equal to the operation key. They then follow the appropriate pointer and fetch the child node. Operations repeat this process until they reach the leaves, at which point, they search the keys and perform whatever action is appropriate. For search operations, if the operation key is in the leaf, the search is successful and the process retrieves the information related to the key. For update operations, the leaf will be modified by inserting or deleting a key value. If the operation is an insert and the key value is already in the leaf, the insert fails. If the operation is a delete and the key value is not in the leaf, the delete fails.

Figure 1.2 shows the nodes that are encountered in the traversal from root to leaf for, in this case, a search for key 45. In this case, the search is successful.

Performing an update may, however, result in a restructuring of the tree. If an insert operation attempts to insert an entry into a node with $2k$ entries (i.e. a full node), it will have to split the node. And, if a delete operation attempts to delete an entry from a node with only k entries, it will have to merge or redistribute its entries with another node such that all remaining nodes have at least k entries in them. It has been shown that, for trees that change quite frequently due to numerous insertions, it is better to allow nodes to contain less than k entries [JS89, JS93a]. In such a scheme, restructuring occurs less frequently. In fact, real database systems often perform merges only when nodes become empty.

A node is defined as being *safe* for an insert if it is not full and safe for a delete if an entry can

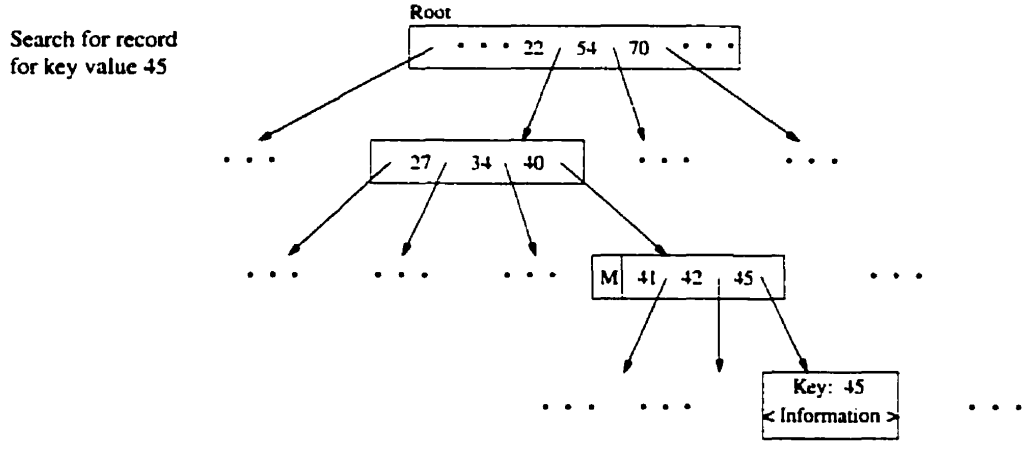


Figure 1.2: B-tree traversal

be deleted from the node without requiring a merge or rotation. A rotation occurs when entries are moved from a node to its sibling and the parent's key that separates the pointers to the pair of children is updated. Figure 1.3 shows a node split for a B-tree with $k = 2$ that occurs when key value 9 is inserted. Note that a new key must then be inserted into the parent for the new pointer. Also, if the split node had been a leaf node, the key value of 13 would still have been placed into the parent, but not removed from the node.

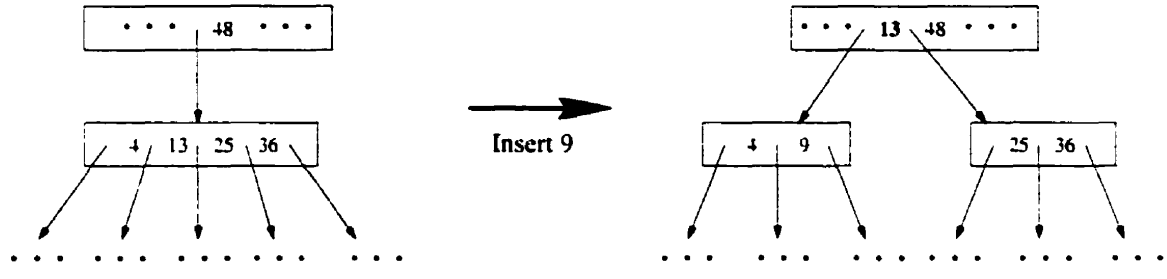


Figure 1.3: B-tree node split

It is important to note that the restructuring of the B-tree may propagate upward towards the root. If an unsafe child is split and the parent is not safe for an insert, the parent will need to be split and the grandparent modified. When the root splits, a new root is created and the B-tree increases in height by 1 level. Conversely, a merge may propogate towards the root too and possibly cause a

reduction in tree height by 1 level. The scope of an update is the set of nodes that are modified by the update operation.

1.2 B-Tree Concurrency Control Algorithms

Restructuring of the B-tree may cause problems when operations operate concurrently. The following example illustrates how concurrency of operations may lead to incorrect results.

Example Consider the B-tree split in Figure 1.4. There are two transactions that operate concurrently on the B-tree:

- Transaction 1: read 46
- Transaction 2: write 9

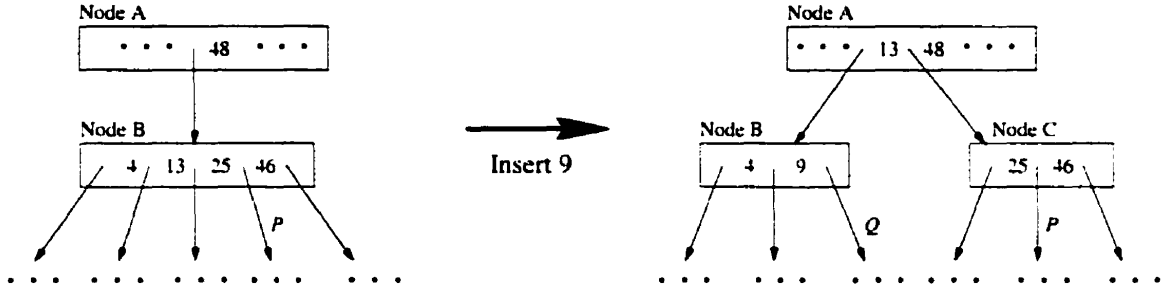


Figure 1.4: B-tree for concurrency example

These transactions are executed such that the sequence of events given by Figure 1.5 takes place.

Transaction 1 fails to traverse the B-tree in a correct manner because Node B changes after Transaction 1 determines that Node B is the next node that must be read. When Transaction 1 finally reads Node B, the correct pointer P is no longer located in Node B, so Transaction 1 follows the incorrect pointer Q. Concurrency control algorithms are required so that concurrent B-tree operations can operate correctly.

1.2.1 Early Algorithms

In this thesis, we use the following locks. S-locks are "shared" locks, meaning that multiple terminals can hold an S-lock on the same item simultaneously. IX-locks are "intention exclusive" locks

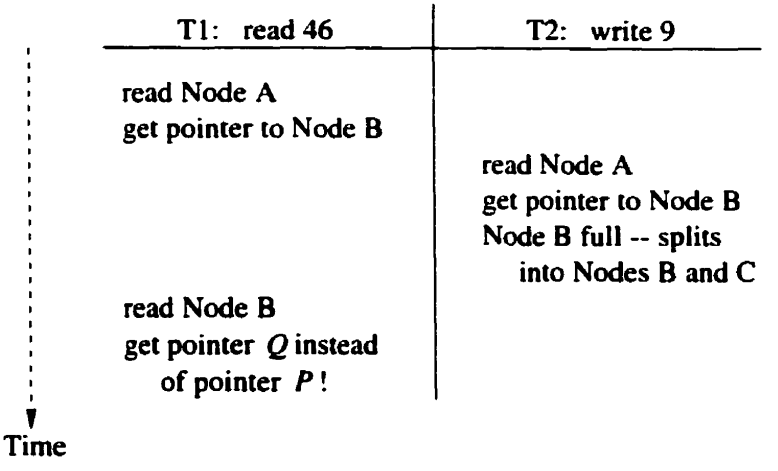


Figure 1.5: Sequence of events for B-tree concurrency example

and SIX-locks are "shared intention exclusive" locks. IX-locks and SIX-locks are typically used to indicate a lock that may be upgraded to an X-lock. X-locks are "exclusive" locks. When a terminal holds an X-lock on an item, no other terminal may hold a lock on the locked item. We assume the lock mode compatibilities that are given in Table 1.1, where a check indicates that the requested lock mode is granted.

Requested Mode	Current Lock Mode				
	Free	S	IX	SIX	X
S	✓	✓	✓	✓	
IX	✓	✓	✓		
SIX	✓	✓			
X	✓				

Table 1.1: Lock compatibility table

The simplest concurrency control algorithm would be to treat the entire B-tree as a single data record. In such a case, there would be only 1 lock. Search operations would hold an S-lock on the tree during their entire search of the B-tree and update operations would hold an X-lock on the tree during their entire update. Thus, searches would be allowed to perform concurrently, but not updates. Such an algorithm is naive and provides very little concurrency of operations. Improvement is made by treating each node in the B-tree as an individual data record that can be locked.

Typical concurrency control techniques for data records, such as two-phase locking [Gra78]

where operations perform no further locking once they perform an unlock, reduce concurrency because operations are needlessly blocked out of certain areas of the data structure. Several algorithms specifically for B-tree concurrency control have been proposed over the years. We now briefly describe various algorithms for B-tree concurrency control.

Metzger, Samadi, and Parr

Metzger [Met75], Samadi [Sam76], and Parr [Par77] proposed the first solution for the B-tree concurrency control problem. With only X-locks, this simple algorithm uses the *lock-coupling* technique; that is, operations unlock a node only after locking its appropriate child. Update operations unlock a node only if the child is found to be safe and release all its ancestor locks after locking a safe node. Thus, if a leaf is unsafe, all ancestors that will be modified remain locked. In other words, the scope of the update is locked. Due to the exclusive use of X-locks, lock conflicts occur high in the tree, often at the root.

Bayer and Schkolnick

Bayer and Schkolnick [BS77] proposed a class of four algorithms to improve on Samadi's approach. In all four algorithms, search operations lock-couple from root to leaf with S-locks. The update operations for each algorithm differ. Bayer and Schkolnick's algorithms are as follows:

Algorithm 1: Updates lock-couple from root to leaf with X-locks, releasing all their locks on a child's ancestors if the child is found to be safe. As with Samadi's algorithm, updates X-lock the root, even if it is not in their scope. Hence, the term "naive lock-coupling" is often used to describe these algorithms.

Algorithm 2: This algorithm performs what is known as "optimistic descent." Updates lock-couple from root to leaf, placing S-locks on all non-leaf nodes and an X-lock on the leaf. The parent of the locked child is always unlocked, even if the child is an unsafe node. If the leaf is unsafe, the leaf and its parent are unlocked and Algorithm 1 is performed. If very few updates are retried, this algorithm is expected to perform well; otherwise, Algorithm 3 performs better.

Algorithm 3: Updates lock-couple from root to leaf with SIX-locks (which are compatible with

S-locks), releasing all locks on ancestors if a child is found to be safe. After locking the leaf, all currently held locks are converted, top-down, into X-locks. The top-down lock conversion drives search operations out of the scope before modification occurs. This algorithm's advantage is that searches and updates can concurrently access the same nodes. The disadvantage is that updates cannot concurrently access the same nodes.

Algorithm 4: This is a generalized algorithm that combines the other three algorithms. Two parameters determine which algorithms to use for which parts of the B-tree. These parameters are P and Ξ, which specifiy the maximum number of levels on which updates can place S-locks and X-locks respectively.

Updates in the Bayer-Schkolnick algorithms hold numerous X-locks at the same time because they update the entire scope at one time.

Miller and Snyder

Miller and Snyder [MS78] proposed an algorithm that differs from Bayer and Schkolnick's in that updates X-lock only the nodes that are going to be modified. All operations lock from root to leaf with S-locks. There is no lock-coupling since nodes are unlocked prior to locking the child. X-locks are made only when updates reach the leaf. For an unsafe node, inserts X-lock up to three ancestor nodes, as well as its parent's adjacent siblings. Deletes X-lock in the same manner, except that they also X-lock the children of the parent's siblings. As needed, the block of locked nodes ascends up the tree.

Kwong and Wood

Kwong and Wood [KW80b, KW80a, KW82] proposed a solution designed to improve on the Bayer-Schkolnick algorithms and Ellis's solution for 2-3 trees [Ell80] by minimizing the time that X-locks are held. On the descent down the tree, operations perform as in Bayer and Schkolnick's Algorithm 3. If the leaf is unsafe, "side-branching" occurs; that is, for inserts, half of the entries from the unsafe node (as well as the new entry) are copied into a new node. For deletes, if the leaf is unsafe, rotation with a safe sibling and parent occurs and results in both siblings being safe. Side-branching

for deletes occurs only if both siblings are unsafe, in which case, entries from the node containing the entry to be deleted are copied (minus the deleted entry) into the adjacent sibling. Modification occurs up the tree until a safe ancestor is modified. Then, the update resumes its descent down the tree to remove the copied entries from the unsafe nodes (in the case of an insert) or remove the redundant nodes (in the case of a delete). On this second descent, X-locks are used and the parent is unlocked prior to X-locking the child.

1.2.2 Top-down Algorithms

Algorithms are considered to be top-down if they perform preparatory node splits or merges. B-tree restructuring by an operation occurs only from root to leaf and in sub-operations that involve only two node levels at a time.

Mond and Raz

Mond and Raz [MR85] proposed a top-down algorithm based on an algorithm by Guibas and Sedgewick [GS78] that introduced preparatory node splits for 2-3 and 2-3-4 trees and the idea by Keshet [Kes81] of immediately splitting or merging unsafe nodes to avoid long chains of locks. During the descent from root to leaf, inserts perform a node split on any unsafe node they encounter and deletes perform a node merge or entry redistribution if they encounter a node unsafe for deletion. Hence, whenever a node is restructured, the parent is safe. Searches use S-locks and updates use X-locks. Each operation holds only a pair of locks at any one time — the current node and its parent. The locking technique is slightly different than lock-coupling in that, before locking any node, its grandparent is unlocked.

Lanin and Shasha [LS86] note that Mond-Raz algorithm can be improved by using optimistic descents as in Bayer and Schkolnick's Algorithm 2. In such a scheme, updates would use S-locks on their descent and X-lock the leaf. If the leaf is unsafe, the update releases all its locks and restarts, using all X-locks. Srinivasan and Carey [SC91a, SC91b] note that the algorithm can also be improved if updates use SIX-locks on descent and convert them to X-locks only if a split or merge is necessary.

Keller and Wiederhold

Keller and Wiederhold [KW88] determined that Mond and Raz's use of preparatory splitting cannot be used for trees with variable-length key values because of its inability to determine with absolute certainty that a new entry will fit into the newly split node. They introduced the sibling promotion technique for the case when there is insufficient room to add the entry in the parent of a new node. In this case, at least half the entries in the node to be split n are put in a new sibling node n'. A pointer from n to n' is created and the parent p_n is marked to indicate that it must be split. The next update operation to reach p_n splits p_n (which creates p'_n) and moves the pointer that leads to n' out of n and into p'_n. If the parent of p_n needs to be split, it is marked and the process repeats for the next update operation.

Setzer and Zisman

Setzer and Zisman [SZ94] propose a technique based on [MR85] and [GS78] in which tree nodes are maintained such that they are safe for inserts and deletes. Searches, inserts, and deletes use only X-locks. A tree compression process operates concurrently and uses a new lock type, which they call a c-lock. Search, insert, and delete operations lock pairwise, as in Mond and Raz's algorithm. As in Keller and Wiederhold's algorithm, there may be variable-length keys. Also, leaves contain a pointer to their right neighbour.

The algorithm uses load factors to determine which method of tree restructuring is best. A load factor of a node $F_n = I_n/2k$, where I_n is the number of entries in node n, determines what action to take if a node is unsafe for insertion. A predetermined split factor limit f_s determines when it is better to redistribute entries among 2 nodes and change the separator into the parent instead of splitting a node and inserting a new separator into the parent. All operations restructure the tree when they encounter an unsafe node. Nodes are considered unsafe for deletes if they contain k entries. A cycle (which may also occur in Mond and Raz's algorithm) may be caused by merging 2 nodes to form a node unsafe for insertion. To prevent such a cycle, merging or redistribution of entries occurs only if 2 adjacent nodes contain fewer than k entries or if 2 nodes with fewer than k entries are separated by a node with k or more entries. Thus, it is possible for nodes to remain with less than k entries. If the total load factor of the tree becomes too small, the compression process

will be triggered.

The compression process starts by c-locking the leftmost leaf node. The other operations may read nodes that are c-locked and read and update nodes to the right of the c-locked node. If an update process reaches a c-locked leaf node, it is interrupted and restarted at the new B-tree root once the compression process finishes. The compression process builds a new, compressed B-tree in another area of the disk and then copies the new tree over the old tree once compression is complete. The process copies the entries of the c-locked leaf into a new node n_1. Then, the process follows the pointer to the right neighbour and c-locks the right neighbour. The entries of the right neighbour are copied into n_1 if there is enough room in n_1; otherwise, the entries are copied into a new node n_2. This process continues until the rightmost leaf is copied into new node n_m. Nodes $n_1, \ldots, n_m$ are the leaves of the new tree. The process then creates new parents for the new leaves and works its way up the new tree until it creates a new root. While the compression process is taking place, no other restructurings of the tree, except for node splits, occur.

1.2.3 B^{link}-Tree Algorithms

The B^{link}-tree (pronounced "B-link-tree") was proposed by Lehman and Yao in 1981 [LY81]. It is based on the idea of using link pointers in concurrent data structures by Kung and Lehman [KL80], who used link pointers in a concurrent binary tree. B^{link}-tree algorithms differ from top-down algorithms because restructuring occurs in sub-operations that involve only one node level and ascend the tree in a bottom-up manner.

The B^{link}-tree is a B-tree with the addition of a high key and link pointer in each node. The high key in a node n specifies the highest key value for the subtree that is rooted at node n. The link pointer goes from a node to the node immediately to the right on the same node level. Figure 1.6 shows a fragment of a B^{link}-tree.

The B^{link}-tree provides the ability to recover when operations read an incorrect node. Node splits occur in two stages: the half-split, then the add-link[1]. In the half-split, the node n is split into nodes n and n' and the link from n to n' is added. The link in n' points to the node that the link in n pointed to prior to the split. In the add-link stage, the pointer from the parent to the new node n'

[1]These terms were introduced by Lanin and Shasha [LS86]. Their B^{link}-tree algorithms will be discussed shortly.

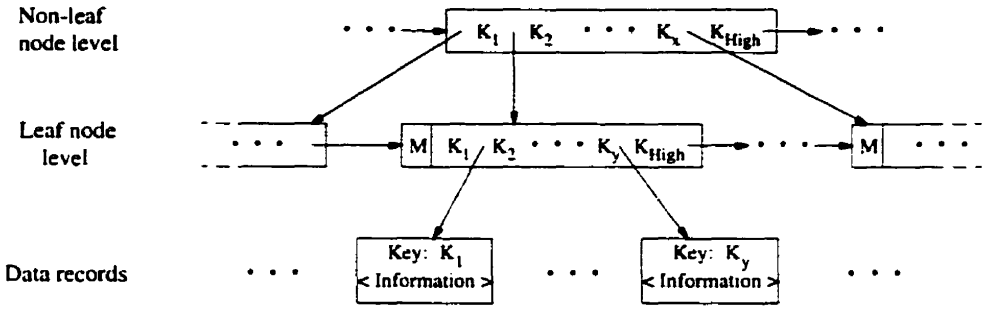


Figure 1.6: B^{link}-tree nodes

is created. Consider the example at the beginning of this section on page 5. The same example is presented for the B^{link}-tree below.

Example Consider the B^{link}-tree split in Figure 1.7. The two transactions that operate concurrently on the B^{link}-tree are:

- Transaction 1: read 46
- Transaction 2: write 9

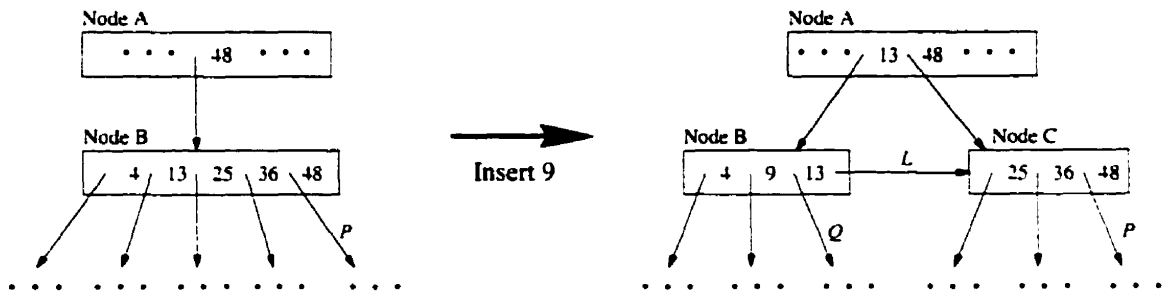


Figure 1.7: B^{link}-tree for concurrency example

These transactions are executed such that the sequence of events in Figure 1.8 takes place.

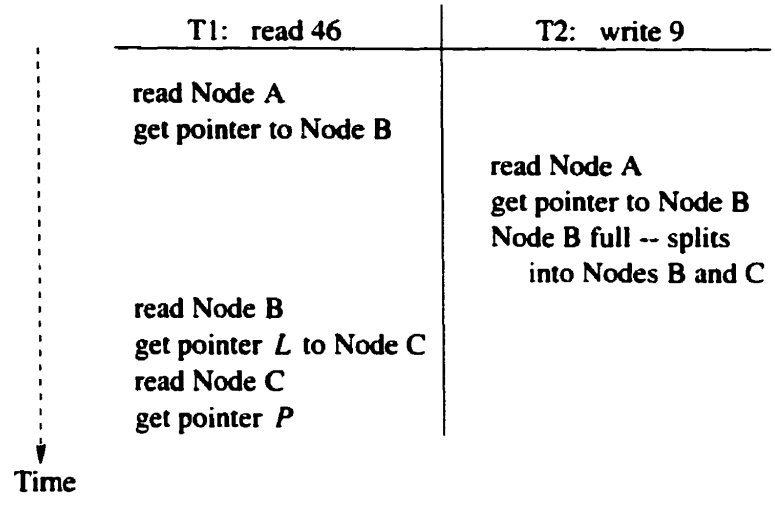


Figure 1.8: Sequence of events for B^{link}-tree concurrency example

Transaction 1 succeeds in traversing the tree because the link pointer L allows the operation to advance to the right when it is found that the operation key is bigger than any of the keys in the current node. Following a link pointer to a neighbour node is called a *link chase*. Transaction 1 thus advances to the correct node and is able to locate the correct pointer P.

Lehman and Yao

In the Lehman-Yao algorithm, searches do not do any locking. Updates do not lock on their initial descent from root to leaf and use X-locks once they reach the leaf level. Once the update X-locks the leaf, any required link chases are performed by lock-coupling with X-locks. If a leaf needs to be split, lock-coupling up the tree occurs until no more ancestors need to be split. In such a scheme, at most 3 nodes are locked by any operation. If a link chase needs to occur while selecting the correct parent, lock-coupling will occur and 3 nodes will be locked. There is no algorithm for performing any concurrent restructuring due to deletion. Instead, Lehman and Yao suggest that if nodes become excessively underutilized, a batch restructuring can lock the entire tree and take place while all other operations wait.

Lehman and Yao do not use any S-locking because they assume atomic disk I/O of nodes for each operation. A modification of their algorithm is to have operations S-lock from root to leaf

and have updates release their S-lock on the leaf before X-locking the leaf and performing any link chases.

Sagiv

Sagiv [Sag85, Sag86] improves on Lehman and Yao's algorithm by creating a B^{link}-tree compression procedure that can run concurrently with other operations and by reducing the number of locks an insert operation holds at any given time. There can be either 1 compression process that periodically restructures the entire tree or multiple processes that modify only 2 adjacent nodes each. Restructuring consists of merging adjacent nodes if they contain $2k$ or fewer entries in total, or redistributing entries among adjacent nodes if they contain more than $2k$ entries in total. Compression processes hold locks on 3 nodes at a time: a parent and 2 children. If there is 1 compression process, the process traverses each tree level and examines pairs of adjacent nodes. This process is similar to the idea that was proposed by Salzberg [Sal85]. If there are multiple compression processes, each process examines only 1 pair of nodes.

Sagiv also modifies the insert algorithm so that it has at most 1 node locked at any one time. There is no reason why update operations should not be allowed to overtake one another on the ascent up the B^{link}-tree. Plus, holding only 1 lock prevents the possibility of deadlock with concurrent compression processes.

Because compression of the B^{link}-tree occurs concurrently with other operations, it is possible that an operation may find that the node it is to operate on is no longer the correct node or no longer exists. Sagiv's solution is to simply restart the operation that fails to traverse the tree correctly.

Lanin and Shasha

Lanin and Shasha [LS86] perform B^{link}-tree compression by using deletes that occur, similarly to insertions, in two stages: the half-merge stage and the delete-link stage. In the half-merge stage, the node to be deleted n' gets all its entries moved to its left neighbour n and its link pointer set to point to n. Thus, any operation that encounters n' (since the pointer to it is still in its parent) will be able to get to n and traverse the tree correctly. The delete-link stage is the removal of the entry that points to n' from the parent.

In the Lanin-Shasha algorithm, S-locks are always used without lock-coupling in the initial descent down the B^{link}-tree. Once an update reaches the leaf, it releases its S-lock and X-locks the leaf. Operations perform any required link chases without lock-coupling. On the ascent up the tree, inserts hold no more than 1 lock at any time and deletes hold no more than 2 locks at any time.

There are possible inconsistent situations that arise due to early unlocking. It is possible that an insert operation may find that the key to add to the parent of a split node already exists in the parent. Figure 1.9 shows such a case. In a similar manner, a delete operation may find that the key to delete in the parent that separates newly merged nodes does not yet exist in the parent. A simple solution to this problem, noted by Srinivasan and Carey [SC91a, SC91b], is to lock-couple on the ascent up the B^{link}-tree. Operations hold an S-lock on their split or merged node until after they've acquired an X-lock on the parent. By doing this, an update will not encounter a parent that has yet to be modified by another update.

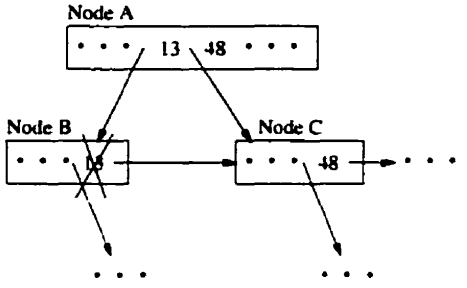
1.2.4 Other Algorithms

Biliris

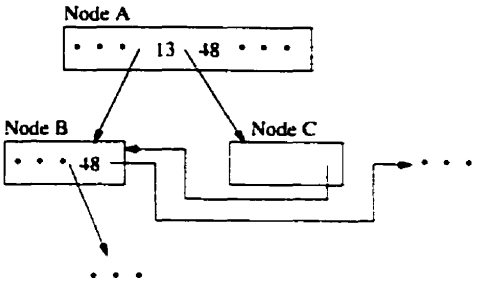
The algorithm by Biliris [Bil87] is called the mU protocol. Two different types of SIX-locks, which are incompatible with each other, exist for inserts and deletes. In addition to the high keys and right links for each node as in the B^{link}-tree, each node contains a low key and left link. The maximum number of insert SIX-locks on a node at any given time is equal to the number of insertions that can be performed on the node without causing a split. Conversely, the maximum number of delete SIX-locks on a node at any given time is equal to the number of deletions that can be performed on the node without requiring a merge.

Mohan and Levine

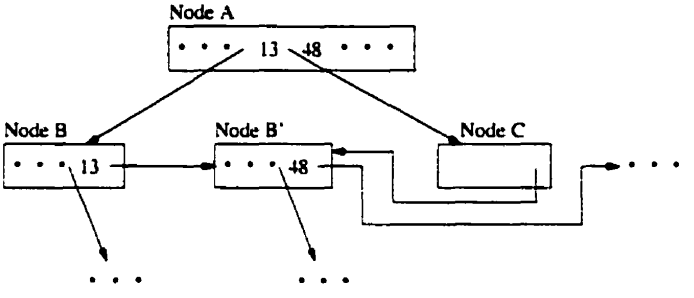
The ARIES/IM algorithm by Mohan and Levine [ML89, ML92] considers transactions that may contain multiple operations on B-trees. The nodes of the tree are such that leaves contain left and right links and non-leaves do not contain any links. Update operations lock-couple from root to leaf, placing S-locks on all non-leaf nodes and an X-lock on the leaf. Searches use only S-locks. Link chases may be performed at the leaf level. For the non-leaf levels, instead of link



(a) Delete operation deletes entry for key value 13



(b) Delete has performed half-merge and unlocked Nodes B and C



(c) Insert operation has inserted key value 13 back into tree, performed half-split of Node B into Nodes B and B', and unlocked Node B. When the insert operation goes to insert key value 13 into Node A due to the node split, it will find that key value 13 is already there!

Figure 1.9: B^{link}-tree inconsistency encountered by insert operation

chases, operations use a complex method based on recursive retries. An important property of the ARIES/IM algorithm that distinguishes it from the other algorithms is that only 1 restructuring operation (either a node split or merge) is allowed to occur at a time. Mohan and Levine do suggest, however, that multiple restructuring at the leaf level can occur by locking a tree lock for restructuring operations in IX-mode for leaf level splits or merges, locking the tree lock in X-mode for non-leaf level splits or merges, and performing deadlock detection to avoid deadlock caused by multiple operations attempting to upgrade their IX-lock to an X-lock.

1.3 Trie Preliminaries

The trie was developed by de la Briandais in 1959 [dlB59] and Fredkin in 1960 [Fre60]. The trie (pronounced "try" even though it is derived from "information retrieval" [Fre60]) is also known as a digital tree [Knu73]. A trie stores data along the paths from root to leaf, unlike a B-tree which stores data at the nodes. Multiple key values share paths near the root; thus, tries achieve significant data compression rates of 90% or higher for large files [Mer98]. As well as its use for large amounts of general and spatial data, tries have other applications. Tries were used in the first sublinear-time algorithm for retrieval of substrings from large texts [Mor68]. In addition, tries are particularly useful for variable-resolution queries since they store the most significant digits or characters near the root [MS94, Sha95].

1.3.1 Trie Description

We will consider only binary tries for fixed-length keys. In a binary trie, the maximum fanout is 2. Each edge between a parent and a child represents a data bit. A left edge represents a "0" bit and a right edge represents a "1" bit. The first bit of any key is stored at an edge from the root to its child. Since the key values are of a fixed length, all paths from root to leaf are of the same length. Figure 1.10 shows a trie for 9 key values: 00000011, 00101100, 00101111, 10000000, 10000101, 10001000, 10100000, 10101100, and 11010000. In the trie, the key value of 00000011 is represented by the leftmost path and the key value 11010000 is represented by the rightmost path. The leaves of the trie are empty and not shown.

Orenstein developed a method of storing tries such that pointers are not used [Ore82, Ore83]. Instead of pointers, a pair of bits is used to represent the edges from a node to its children. Each node is represented by a *bit pair* that consists of 2 bits such that the left bit indicates the existence of a left edge or "0" bit and the right bit indicates the existence of a right edge or "1" bit. A "1" in the bit pair indicates that the corresponding edge exists and a "0" in the bit pair indicates that the edge does not exist. For example, the root node of the trie in Figure 1.10 is represented by the bit pair "11" because it has 2 children and the root's left child is represented by the bit pair "10" because it has only a left child. Figure 1.11 shows the pointerless representation of the entire trie in Figure 1.10.

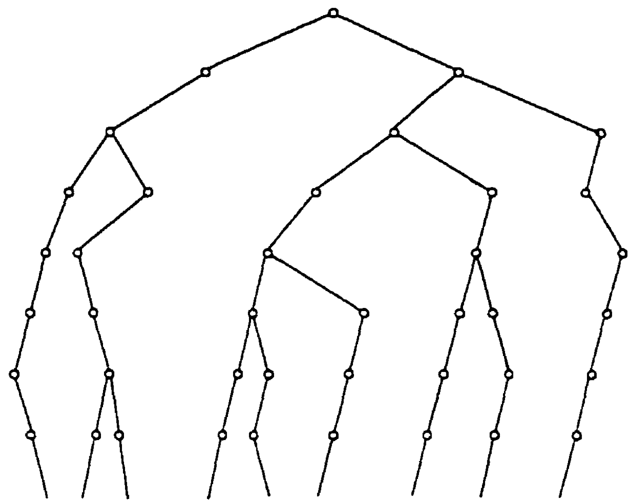


Figure 1.10: Trie

Root ⟶							11						
		10									11		
		11							11				10
	10		10				10				10		01
	10		01				11				11		10
	10		01			11		10		10		01	10
	01		11		10		10	10		10		10	10
	01	10		01	10		01	10		10		10	10

Figure 1.11: Pointerless representation of trie

Without pointers, operations need to traverse the trie by counting the "1" bits in the bit pairs. The operation compares the current input bit of the operation key with the node. If there is a "1" in the bit pair that corresponds to the input bit, the operation advances to the next node level; otherwise, the operation key is not in the trie. To determine which bit pair to examine in the next node level, the operation counts the number of "1" bits it encounters in the bit pairs of its current level as it advances from left to right.

For example, say that an operation is searching for key value 11010000 in the trie in Figures 1.10 and 1.11. The input bit is "1" (the first bit of the operation key) and the bit pair for the root node is "11". The right (second) bit of the bit pair indicates that an edge corresponding to "1"

exists. The operation has encountered 2 "1" bits: the left (first) "1" bit and the right (second) "1" bit. Since the second bit matches the current input bit, the operation determines that it must next examine the second node of the next node level.[2]

At the next node level, the operation checks if the second bit of the operation key, which is "1", exists in the trie. The operation advances from left to right, taking note of the "1" bit in the leftmost node and then examining the second node. The second node is "11" and the second bit of the node matches the current input bit. By encountering this second bit of the node, the operation has encountered 3 "1" bits in the level: 1 "1" bit in the leftmost node and both "1" bits in the second node. Thus, the operation will examine the third node of the next node level. Traversal of the trie continues in this manner until the operation is finished.

Since each "1" bit indicates a child in the next node level, an operation can determine how many nodes are in the next level by simply counting all the "1" bits in its current level. Thus, the bit pairs can be stored simply as a sequence of bit pairs. For the trie in Figure 1.10, this sequence is:

⟨ 11 10 11 11 11 10 10 10 10 10 01 10 01 11 11 10 10 ... ⟩

Rather than traverse the entire trie sequentially, we divide the trie into levels of pages that can be traversed [Ore83]. By using *page counts*, an operation can determine which page contains the next node to examine without traversing the entire node level of the trie from left to right. With these page counts, operations need only traverse the node levels within specific pages. Each page has two counts: a *T-count* and a *B-count*. The T-count for a page n specifies the number of edges that enter the top of all pages to the left of n on page level ℓ_n. The B-count for a page n specifies the number of edges that exit the bottom of all pages to the left of n on page level ℓ_n. In addition, each page level has a T-count and B-count that specifies, respectively, the total number of edges entering the tops and exiting the bottoms of all pages in the level. To ensure that these counts are effective, edges are allowed to enter and exit pages only at the tops and bottoms, not the sides. Figure 1.12 shows a paged trie for the trie in Figure 1.10.

To determine which page to traverse next, operations first add the value of the B-count for their current page to the value they have calculated as the next node they must examine. Consider our earlier example of a search for key value 11010000. Once the last node level of the root page is

[2] If the current input bit had instead been a "0", only the first "1" bit of the root would have been encountered and the operation would be examining the first node of the next level.

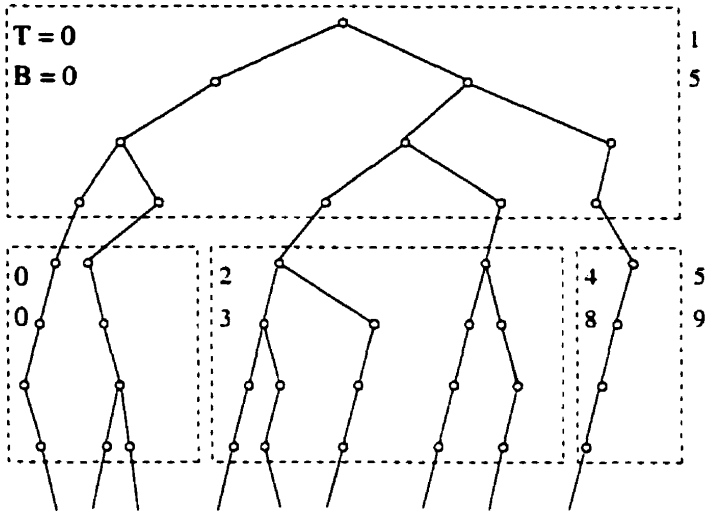


Figure 1.12: Paged trie

examined, the operation has counted the "1" bits and concluded that it must next examine the fifth node on the next node level. However, since there may be pages to the left of its current page, the operation must add all the edges descending from the pages to the left in order to truly determine which node on the next node level to examine. In this case, the current page is the root page, so there is no page to the left and the B-count is 0. Thus, the operation determines that it must examine the fifth node on the next node level since $5 + 0 = 5$.

To determine which page the next node to examine is in, the operation uses the T-counts for the next page level. The operation selects the page to the left of the page with the minimum T-count that is greater than or equal to its calculation of $5 + 0 = 5$. The T-counts for the pages are 0, 2, 4, and 5. For this search, the page level count 5 is the minimum T-count that is greater than or equal to 5. Hence, the operation chooses the page to the left; that is, the page with $T = 4$. Instead of traversing the entire node level of the trie, only the page with $T = 4$ will be traversed.

1.3.2 Trie Operations

We now discuss in detail how operations search and insert key values in a paged trie with pointerless representation. Since the focus of this thesis is trie concurrency for search and insert operations, we do not discuss deletions in the paged trie.

Searching a Trie Page

This section on searching a trie page is from [Mer98] except the discussion on finding the next page to search and the page search algorithm presented in Figure 1.15. The algorithm used for searching a trie page that is represented as a sequence of bit pairs is from [Ore83]. Consider the paged trie in Figure 1.12 on page 20 .

The sequence of bit pairs for the root page is:

```
( 11  10  11  11  11  10  10  10  10  10  01 )
```

To navigate the bit pair sequence, a counter and a cursor are used. The counter, $size$, stores the number of bit pairs on each node level. Using $size$, the cursor, $last$, stores the location of the last bit pair on the current node level. Initially for the root page: $size = 1$ and $last = 0$. As we traverse the bit pair sequence, $size$ and $last$ are modified as follows:

- $size$ is incremented by 1 each time the bit pair "11" is encountered
- $last$ is incremented by $size$ each time the last bit pair of the current node level is encountered

For the trie in Figure 1.12, $size$ and $last$ are modified for the root page as shown in Figure 1.13.

position	bit pair	$size$	$last$	comments
		1	0	initial values for root page
0	11	2	2	"11" read, $last = last + size$
1	10			
2	11	3	5	"11" read, $last = last + size$
3	11	4		"11" read
4	11	5		"11" read
5	10		10	$last = last + size$
6	10			
7	10			
8	10			
9	10			
10	01		15	$last = last + size$

Figure 1.13: Modification of $size$ and $last$ during page traversal

To actually search the page, a bit pair in the page must be compared with a current input bit. As soon as the current input bit is not found in the bit pair being compared, the search terminates. The

bit pair used for comparison is in the position identified by $next$. To find the value of $next$ for the next node level, a counter, $srch$ is used. Initially for the root page: $next = 0$ and $srch = 0$. As we traverse the bit pair sequence, $next$ and $srch$ are modified as follows:

- Before the bit pair at $next$ is reached, $srch$ is incremented by 1 for each "1" bit in the bit pair.
- At the bit pair at $next$, increment $srch$ for each "1" bit in the bit pair, including the bit that matches the current input bit, but not past it.
- At the bit pair at $last$, $next = last + srch$ and then $srch = 0$.

For the trie in Figure 1.12, the search for the search key 10101100 proceeds for the root page as shown in Figure 1.14.

position	bit pair	input bit	*srch*	*next*	*size*	*last*
		1	0	0	1	0
0	11	0	2, then 0	2	2	2
1	10		1			
2	11	1	2, then 0	4	3	5
3	11		2		4	
4	11	0	4		5	
5	10		4, then 0	9		10
6	10		1			
7	10		2			
8	10		3			
9	10	1	4			
10	01		4, then 0	14		15

Figure 1.14: Modification of counters during page search

After searching the root page, the search may need to continue and search other pages of the trie. Rather than traverse all the bit pairs of the trie, we use T and B to determine which trie page to search next.

The $srch$ counter, by counting the "1" bits in the current node level, counts the descendent nodes or subtries from the current node level. Before being reset to 0 in our example, $srch = 4$ after the

last node level of the root page is searched. This means that we must go to the fourth subtrie from the beginning of the next page level. Using the T-counts for the pages on the next page level, the correct page to use for continuing the search is determined. With the current page having a B-count of B', we scan the next page level for $T \geq B' + srch$ and choose the page immediately to the left.

We must also reset $next$, $last$, $size$, and $srch$ in order to correctly traverse the next page. They are modified, in order, as follows:

- $next = B' + srch - T - 1$, where T is the T-count for the new page to be searched and B' is the B-count for the page that was just searched
- $srch = 0$
- $size = T'' - T$, where T'' is the T-count for the page immediately to the right of the new page to be searched
- $last = size - 1$

To summarize, the algorithm for searching a trie page is a loop over all node levels within the page. The loop contains a loop until $next$, code for $next$, a loop until $last$, and code for $last$. This algorithm is presented in Figure 1.15. on page 24.

Inserting Data into a Trie Page

When inserting data into a trie page, the page is traversed in the same manner as a page search. The values of $size$, $last$, $srch$, and $next$ are used and modified in the same fashion. There are two phases for inserting data into a trie page:

- Phase 1: Search page and change bit pair that does not match current input bit
- Phase 2: Search page and insert bit pairs that represent subsequent current input bits

When the current input bit is not found in the bit pair at $next$, the current input bit is inserted and the bit pair at $next$ is changed. For example, if the current input bit is "1", the bit pair at $next$

```
SEARCH-TRIE-PAGE(key, page)
    /* counters size, srch, next, and last have been initialized */
 1  for node-level ← 1 to t        /* t is the number of node levels in a page */
 2      while bit-pair is before next do
 3          if bit-pair = "11" then
 4              size ← size + 1
 5              srch ← srch + 2
 6          else if bit-pair = "10" or bit-pair = "01" then
 7              srch ← srch + 1
 8      if bit-pair does not match input bit then
 9          return "search failed"
10      if bit-pair = "11" and input bit = "0" then
11          size ← size + 1
12          srch ← srch + 1
13      else if bit-pair = "11" and input bit = "1" then
14          size ← size + 1
15          srch ← srch + 2
16      else
17          srch ← srch + 1
18      if input bit was last input bit then
19          return "search succeeded"
20      input bit ← next input bit
21      while bit-pair is before last do
22          if bit-pair = "11" then
23              size ← size + 1
24      if node_level < t then
25          next ← last + srch
26          srch ← 0
27          last ← last + size
```

Figure 1.15: Trie Page Search Algorithm

will change from either "00" to "01" or from "10" to "11." After the bit pair is changed, $size$ and $srch$ may need to be incremented by 1 to reflect the modified bit pair.

After the bit pair is changed and $last$ is reached, the insert procedure then proceeds to insert bit pairs into the page at the location specified by $next$. Each time a bit pair is inserted, the position of all subsequent bit pairs is incremented by 1. Also, $srch$ is incremented by 1 since the inserted bit pair is either "10" or "01". The B-counts of all pages to the right on the page level will also be

incremented by 1.

After reaching the last node level of a page, the insertion may need to continue and insert the remaining portion of the key value into other pages. The method of determining which page to insert the remaining portion of the value into is the same as that used to determine which page to search next.

Once a bit pair has been changed in a page, other pages selected by the insert operation have bit pairs inserted, but no bit pairs changed. In other words, they have subtries inserted into them. The page traversal is the same, except that now, before starting the insertion, $size$ and $last$ must be incremented by 1 due to the addition of the new subtrie. Also, due to the addition of the subtrie, all pages to the right on the page level will have their T-counts and B-counts incremented by 1.

Consider again the trie from Figure 1.12 on page 20. Inserting key value `01001010` into the trie will yield the trie in Figure 1.16. The new key value is indicated by the bold path and the modified T-counts and B-counts are indicated by the bold italics.

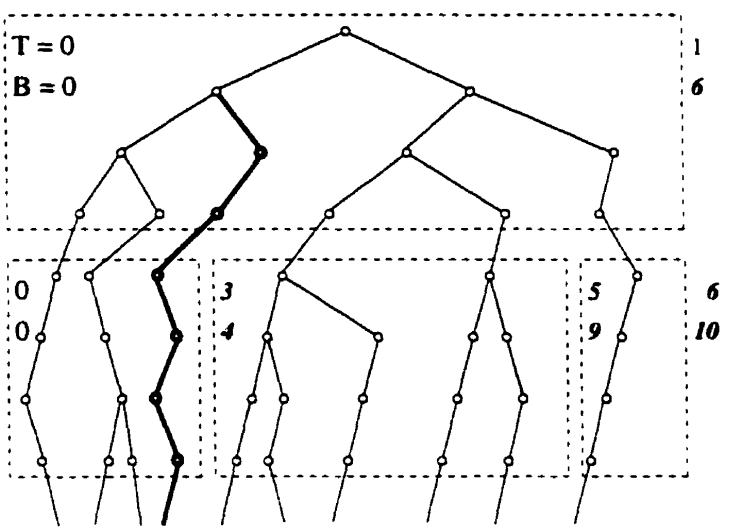


Figure 1.16: Paged trie after insertion

The insertion of key value `01001010` proceeds as shown in Figure 1.17 for the root page and in Figure 1.18 for the descendent page.

position	bit pair	input bit	*srch*	*next*	*size*	*last*	comments
		0	0	0	1	0	
0	11	1	1, then 0	1	2	2	0 in bit pair
1	10		1				no match
1	11	0	2		3		bit pair changed
2	11		2, then 0	4	4	6	
3	11		2		5		
4	01	0	3				0 inserted
5	11				6		
6	10		3, then 0	9		12	
7	10		1				
8	10		2				
9	10	1	3				0 inserted
10	10						
11	10						
12	01						end of page

Figure 1.17: Modification of counters during page insert for root page

position	bit pair	input bit	*srch*	*next*	*size*	*last*	comments
		1	0	2	3	2	*size* + 1, *last* + 1
0	10		1				
1	01		2				
2	01	0	3, then 0	5		5	1 inserted
3	10		1				
4	01		2				
5	10	1	3, then 0	8		8	0 inserted
6	01		1				
7	11		3		4		
8	01	0	4, then 0	12		12	1 inserted
9	01		1				
10	10		2				
11	01		3				
12	10	-	4				0 inserted, end

Figure 1.18: Modification of counters during page insert for descendent page

The algorithms for both phases of insertion are in Figures 1.19 and 1.20 on pages 27 and 28 respectively.

```
MAKE-INITIAL-TRIE-INSERT(key, page)
    /* counters size, srch, next, and last have been initialized */
 1  bit-pair-changed ← FALSE
 2  for node-level ← 1 to t          /* t is the number of node levels in a page */
 3      while bit-pair is before next do
 4          if bit-pair = "11" then
 5              size ← size + 1
 6              srch ← srch + 2
 7          else if bit-pair = "10" or bit-pair = "01" then
 8              srch ← srch + 1
 9      if bit-pair does not match input bit then
10          Change bit-pair to "11" so that it does match input bit
11          bit-pair-changed ← TRUE
12      if bit-pair = "11" and input bit = "0" then
13          size ← size + 1
14          srch ← srch + 1
15      else if bit-pair = "11" and input bit = "1" then
16          size ← size + 1
17          srch ← srch + 2
18      else
19          srch ← srch + 1
20      if input bit was last input bit then
21          return "done"
22      input bit ← next input bit
23      while bit-pair is before last do
24          if bit-pair = "11" then
25              size ← size + 1
26      if node_level < t then
27          next ← last + srch
28          srch ← 0
29          last ← last + size
30      if bit-pair-changed = TRUE then
31          old-node-level ← node-level
32          node-level ← t          /* to break out of this for-loop */
33  for node-level ← old-node-level +1 to t
        /* see for-loop in INSERT-SUBTRIE algorithm in Figure 1.20 */
```

Figure 1.19: Trie Page Insert Algorithm (Initial Insertion)

```
INSERT-SUBTRIE(key, page)
    /* counters size, srch, next, and last have been initialized */
 1  for node-level ← 1 to t        /* t is the number of node levels in a page */
 2      while bit-pair is before next do
 3          if bit-pair = "11" then
 4              size ← size + 1
 5              srch ← srch + 2
 6          else if bit-pair = "10" or bit-pair = "01" then
 7              srch ← srch + 1
 8      Insert bit pair that corresponds to input bit
 9      if node-level = 1 then
10          size ← size + 1
11          last ← last + 1
12      srch ← srch + 1
13      if input bit was last input bit then
14          return "done"
15      input bit ← next input bit
16      while bit-pair is before last do
17          if bit-pair = "11" then
18              size ← size + 1
19      if node_level < t then
20          next ← last + srch
21          srch ← 0
22          last ← last + size
```

Figure 1.20: Trie Page Insert Algorithm (Subtrie Insertion)

When the node capacity of a page is exceeded, the page must be split into two pages. Since no trie edges can cross the side boundaries of a page, a page must have a capacity large enough to store a full subtrie. So, for t, where t is the number of node levels per page, the minimum node capacity must be $2^t - 1$. For example, the insertion of value `10001010` in the trie of Figure 1.16 results in a page split if the node capacity per page is 16 nodes. In this trie, $t = 4$, so $16 \geq 2^t - 1$. Figure 1.21 on page 29 shows the page split that results.

Before a page split occurs, there are at least two subtries in the page. When there are more than two subtries, a split can be made such that the most even distribution of nodes possible between the two pages is achieved. The worst node distribution that can be made is one where a new page

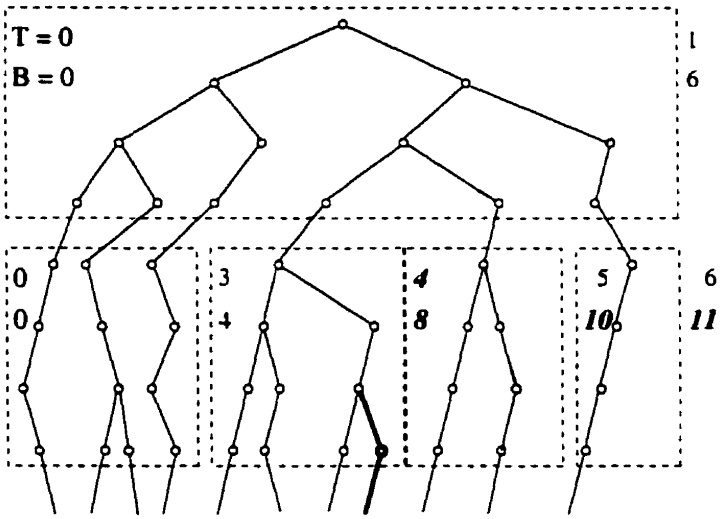


Figure 1.21: Paged trie after split

contains only t nodes; yet, this choice may very well be necessary.

To determine an optimal node distribution between the two pages, the number of nodes in each subtrie of the split page must be known. We will call $count_i$ the number of nodes for subtrie i, where the n subtries in the page are numbered left to right from 0 to $n-1$. To calculate $count_i$, the location of the last node for subtrie i on the current node level must be stored in $last_i$. We know we are in subtrie i if the current node position is $> last_{i-1}$ and $\leq last_i$. For the next node level, the new value for $last_i$ is calculated by using $size_i$ and $last$. Initially, $count_i = 0$, $last_i = i$, and $size_i = 1$. As the page is traversed, these cursors are modified as follows:

- $count_i$ is incremented by 1 each time a bit pair in subtrie i is encountered.
- $size_i$ is incremented by 1 each time the bit pair "11" is encountered in subtrie i

When the bit pair at $last$ is reached, $last_i$ is modified for each subtrie i as follows before incrementing $last$ by $size$:

- $last_i = last + \sum_{j=0}^{i} size_j$

To reduce the number of page traversals that are required, the calculation of $count_i$, $size_i$, and $last_i$ can be done while the insertion procedure is traversing the page.

Once the node counts for the subtries are known, we take the absolute value of all $n-1$ differences $(count_0 + \cdots + count_x) - (count_{x+1} + \cdots + count_{n-1})$ and select x from the miminum difference. The page is then traversed again, keeping all nodes at positions $\leq last_x$ in the page that is being split, and placing all nodes at positions $> last_x$ into the new page that is created.

Consider again the trie from Figure 1.16 on page 25. If the value `10010000` is inserted instead of `10001010`, the page that will be split contains three subtries and we must decide where to split the page. The trie in Figure 1.22 shows the trie after the insertion, but before the page is split.

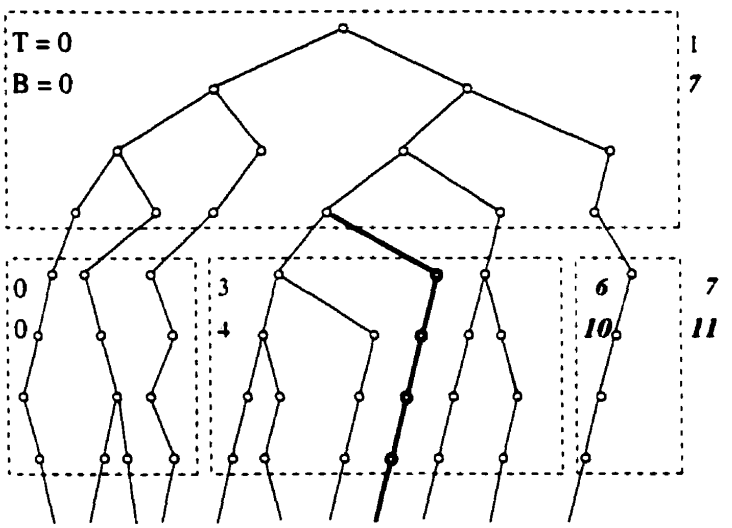


Figure 1.22: Paged trie before optimal split

From the calculation of $count_i$, $size_i$, and $last_i$ in Figure 1.23, we see that for the three subtries $count_0 = 9$, $count_1 = 4$, and $count_2 = 7$. The two differences that must be calculated are $|9 - (4 + 7)| = 2$ and $|(9 + 4) - 7| = 6$. The minimum difference is 2 and the value for x is 0, so for each node level, all nodes at positions $\leq last_0$ remain in the page and all nodes at positions $> last_0$ get placed in the new page. The resulting trie is in Figure 1.24.

position	bit pair	$count_0$	$count_1$	$count_2$	$size_0$	$size_1$	$size_2$	$last_0$	$last_1$	$last_2$
		0	0	0	1	1	1	0	1	2
0	11	1			2					
1	10		1							
2	11			1			2	4	5	7
3	11	2			3					
4	10	3								
5	10		2							
6	10			2						
7	01			3				10	11	13
8	10	4								
9	10	5								
10	10	6								
11	10		3							
12	10			4						
13	10			5				16	17	19
14	10	7								
15	01	8								
16	10	9								
17	10		4							
18	10			6						
19	10			7						

Figure 1.23: Modification of counters prior to optimal page split

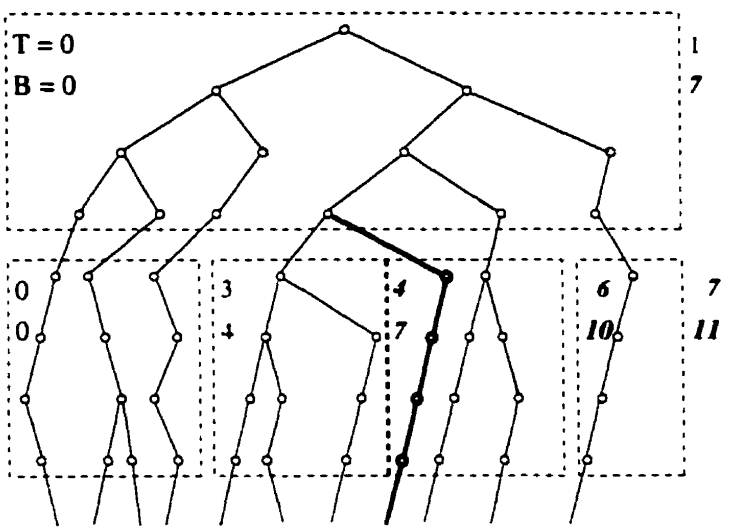


Figure 1.24: Paged trie after determining optimal split

1.4 Thesis Overview

We have introduced B-trees and tries, as well as various algorithms for concurrent B-tree operations. In Chapter 2, we summarize various performance studies that have been published for B-tree concurrency algorithms and describe our model for simulating B-tree and trie concurrency. Chapter 3 describes the simulation experiments that we performed to measure B-tree concurrency. We present our performance results for B-tree concurrency and compare and contrast them with published results. Our goal is to be able to scale our results for trie concurrency onto the published graphs for B-tree concurrency. In Chapter 4, we present our algorithms for concurrent search and insert operations in a paged trie with pointerless representation. We then present our performance results for these algorithms and compare them with the B-tree results. We conclude in Chapter 5 by summarizing the thesis and proposing future work related to trie concurrency.

Chapter 2

Concurrency Simulation

In this chapter, we first describe various concurrency studies that have been published for B-trees. We selected Srinivasan and Carey's work [SC91b] as a basis for and evaluating our own B-tree results because they perform a detailed simulation study by using a large variety of experiments for B-tree concurrency in a centralized DBMS. Thus, we describe the simulation model we used to simulate B-tree and trie concurrency involving the various system resources specified by Srinivasan and Carey.

2.1 Related Work

When proposing a new concurrency control algorithm, authors usually attempt to estimate its performance by analytical or simulation methods. We now briefly discuss work that has been done to measure the performance of various B-tree concurrency control algorithms.

Samadi

Samadi [Sam76] presents simulation results for his algorithm. He measures performance by measuring access and waiting times for each terminal. The interarrival time between requests decreases until requests arrive fast enough that the system can no longer handle them.

Bayer and Schkolnick

Bayer and Schkolnick [BS77] use formulas to approximate various performance measurements. They approximate the number of waiting operations, the number of nodes encountered during retries, and the number of lock conversions. Using these approximations, they determine what values to use for the parameters in their Algorithm 4. Their model is static and they assume that all operations descend the tree simulaneously.

Biliris

Biliris [Bil85] uses simulation to compare four algorithms: naive lock-coupling, optimistic descent with retries using SIX-locks instead of X-locks, side-branching, and his mU protocol. He finds that naive lock-coupling produces the worst performance and that the mU protocol produces the best performance. Unfortunately Biliris does not study the B^{link}-tree algorithms nor provide response times for individual operations or a detailed analysis of the results.

Lanin and Shasha

Lanin and Shasha [LS86] use simulation to compare five algorithms: their modification of Lehman and Yao's B^{link} algorithm, Bayer and Schkolnick's Alglrithms 1 and 2, the Mond-Raz algorithm, and the Mond-Raz algorithm modified so that optimistic descents are made as in Bayer and Schkolnick's Algorithm 2 and retries are made with the Mond-Raz algorithm. They compare speedup, which is the ratio of time it takes 1 terminal to do a unit of work to the time it takes n terminals to do the same unit of work. They do not simulate very many concurrently operating terminals.

Their results show that with low fanout, their modified B^{link} algorithm performs much better than the other algorithms. They found that the B^{link} algorithm runs 26.5 times faster with 40 terminals than with 1 terminal and that the next best algorithm, the Mond-Raz algorithm with optimistic descent, runs only about 10 times faster. With higher fanout, the B^{link} algorithm and the Mond-Raz algorithm with optimistic descent perform almost equally well.

Johnson and Shasha

Johnson and Shasha [JS90] use an analytic model with an open queue to analyze three algorithms: Bayer and Schkolnick's Algorithms 1 and 2, and Lehman and Yao's B^{link} algorithm. The version of the Lehman-Yao algorithm they use has no lock-coupling on ascent and no node merging since merges are rare if inserts outnumber deletes [JS89]. In their experiments, there are few link chases in the B^{link} algorithms, as well as no buffers or resource contention.

They find that the lock-coupling algorithms create a bottleneck at the root and that the B^{link} algorithm provides much better performance. Maximum throughput for Algorithm 1 occurs for an arrival rate of about 0.6 operations at the root per unit of time. Algorithm 2 is better since its maximum throughput occurs at an arrival rate of about 2.7. The B^{link} algorithm, however, reaches no maximum throughput. Even at an arrival rate of 14, response times for B^{link} operations remain almost constant.

Srinivasan and Carey

Srinivasan and Carey [SC91a, SC91b] use simulation with a closed queue to test various algorithms in a system with buffers and contention for disks and CPUs. They study numerous algorithms for four classes of algorithms: B^{link} algorithms, optimistic descent, lock-coupling with SIX-locks, and lock-coupling with X-locks. They also use their results to predict the performance of the side-branching technique, the mU protocol, and the ARIES/IM algorithm. They simulate a large variety of situations with varying workloads and resource contention. They also study situations where the number of concurrent operations is high enough to cause many link chases.

Their results are similar with those of Johnson and Shasha. They find that the B^{link} algorithms perform the best and allow the most concurrency. In fact, they find that the B^{link} algorithm with lock-coupling on ascent, which is more practical, is generally as good as the Lanin-Shasha version. Unlike Johnson and Shasha, they find that Bayer and Schkolnick's Algorithm 1 performs similarly to Algorithm 2 when there are many concurrent operations. They predict that the side-branching algorithm will not perform as well as the B^{link} algorithms and that the mU protocol may perform as well as the B^{link} algorithms only in certain situations. With the modification that allows the ARIES/IM algorithm to perform multiple tree restructuring simultaneously, they suggest that the

ARIES/IM algorithm may perform similarly to the B^{link} algorithms.

Johnson and Shasha

Johnson and Shasha [JS93b] publish an elaboration of [JS90]. They analyze seven algorithms: Bayer and Schkolnick's Algorithms 1 and 2, the Mond-Raz algorithm, optimistic descent using Algorithm 3 and SIX-locks instead of X-locks for retries, optimistic descent where retries occur when all of a certain number of bottom tree levels need to be restructured, the Lehman-Yao B^{link} algorithm, and two-phase locking. In addition, they include contention for disks and buffers. Using simulation, they confirm their analysis.

They again find that the Lehman-Yao algorithm provides the most concurrency. At an arrival rate of 160 with no resource contention, the B^{link} algorithm still does not reach any maximum throughput. The next best algorithm, which is the optimistic descent with SIX-locks, reaches a maximum throughput at an arrival rate of about 10. With resource contention, the still find that the B^{link} algorithm performs best even though it approaches the same performance as the optimistic descent algorithm when resource contention is high. They recommend using either the ARIES/IM or optimistic descent algorithms when the B-tree is shrinking because the Lehman-Yao algorithm provides no node merging. Since their model is analytic, it may be applied to future concurrency control algorithms and even future data structures.

2.2 Simulation Overview

Discrete-event simulation, which emerged as an established discipline with the publication of [Toc63], was originally designed to solve complex queueing theory problems. In a discrete-event simulation, the system state changes at distinct points in time, in contrast to a continuous simulation where the system state changes continuously as a function of time. We have implemented an asynchronous discrete-event simulation where events (which change the system state) may occur at any time, instead of a synchronous discrete-event simulation where events occur at fixed time intervals.

There is an event for each terminal in the system. Each event contains the next action to be performed by the terminal, the time when the event will occur, and other variables needed to store

the present state of the terminal (such as operation type, key, operation start time, current page, etc.). The simulation program is a loop that fetches the next event, calls a subroutine which performs the state change specified by the event, and creates a new event which will occur at a future time until the simulation is finished.

Future events with known activation time are in a (binary) heap [Wil64] and future events with unknown activation time are on a wait queue. We use a heap to store future events whose future activation time is known because insertion and extraction of events in a heap require only $O(\log_2 n)$ time, where n is the number of events in the heap. Events are sorted on the heap such that events are extracted in increasing order of activation time.

We store future events whose future activation time is unknown on the wait queue of the lock or other system resource for which they are waiting. Once the grant time for these events is known, the event's activation time is set and the event is placed in the heap. We discuss these wait queues in more detail in the next section.

We use CPU instructions as the unit of time for scheduling events. Each action that an event specifies requires a number of CPU instructions to perform. After the action is performed for the terminal and the system state changed, the next event for that terminal will occur at a time equal to the current time plus the time required to perform the action. We convert the time required for disk usage, traditionally measured in milliseconds (ms), into CPU instructions by using the CPU speed, which is measured in millions of instructions per second (MIPS).

Furthermore, our simulation model is a closed queueing model. Rather than set an arrival rate for transactions into the system as in an open queueing model, each terminal submits a transaction and, upon completion of the transaction, immediately submits another transaction. There is zero think time between completion of one transaction and submission of the next one. By varying the number of terminals in the system (the multiprogramming level or MPL), we vary the number of concurrent transactions in the system.

We implement our simulation using the Java programming language [AG96, GJS96] because Java is a simple general-purpose object-oriented platform-independent programming language. Srinivasan and Carey, however, implement their simulations using the DeNet simulation language [Liv90].

2.3 System Resources

Srinivasan and Carey simulate B-tree concurrency in a system containing various types of available resources. We now explain how we simulate these resources.

2.3.1 Locks

There is a lock for each page in the B-tree and trie. Each lock consists of a current lock mode, a grant count, and a wait queue. The current lock mode indicates whether the lock is free, S-locked, X-locked, or in wait mode. The grant count contains the number of terminals currently granted the lock. The grant count is 0 when the lock is free, 1 when the lock is X-locked, and 1 or more when the lock is S-locked or in wait mode. If a terminal attempts to lock a page in a mode incompatible with the current lock mode, the lock mode is set to indicate a wait and the future event for the terminal is placed on the wait queue. Locks are granted FCFS (first-come, first served), so any subsequent requests for a lock in wait mode are placed on the wait queue. When a terminal waiting on the queue is granted the lock, the activation time for that terminal's event is set to the current time and the event is moved from the wait queue to the heap. Since the activation time for the granted terminal's future event is set to the current time, the next event to occur will be the granted terminal's future event – acquiring the lock.

2.3.2 Buffers

The buffer pool exists to store currently and recently used pages in main memory and reduce disk usage. The buffer manager consists of the pages that are presently in the buffer, a fix count for each page, and an "LRU (least recently used) stack." The LRU stack orders pages that are in the buffer and not currently in use such that the page at the top of the stack, which is least recently used, is the page to be replaced when another page needs to be inserted into the buffer.

Before starting a simulation, we initialize the buffer from root to leaf and from left to right so that the most frequently used pages (those closest to the root page) are in the buffer. Initially, all pages in the buffer are unfixed (with fix count of 0) and are therefore in the LRU stack.

When a terminal requests a page for processing, it fixes the page in the buffer. When a page is

fixed, the fix count for the page is incremented by 1. If the requested page is in the LRU stack at the time of fixing, the page is removed from the LRU stack so that it is not written out of the buffer while in use by the terminal. If the requested page is not in the buffer at the time of fixing, a buffer miss occurs. When a buffer miss occurs, the terminal must perform disk I/O twice: once to write out the LRU page from the buffer and once to read the requested page into the buffer. The LRU page written out of the buffer is also removed from the LRU stack. Once the terminal is finished processing the page, the fix count for the page is decremented by 1. If the fix count becomes 0, the page is then added to the LRU stack.

We simulate an infinite buffer pool by simply assuming that all pages (except newly created pages which result from a page split) are always in the buffer. Newly created pages are immediately placed in the buffer after their creation. Therefore, with an infinite buffer pool, there is no disk I/O.

2.3.3 Disks

Each disk has a flag indicating whether it is in use or not, as well as its own wait queue for pending I/O requests. Requests for a disk are serviced in a FCFS manner. A disk is used to write out a page from the buffer or to read a page into the buffer. Disk I/O occurs when a buffer miss occurs or when a page must be written out of the buffer to make space for a newly created page.

When disk I/O is required, the disk chosen to perform the I/O is selected at random from all the existing disks. If the requested disk is free, the disk is assigned to the terminal and the flag is set to indicate the disk is in use. If the requested disk is already in use, the terminal must wait for the disk to become free. When the terminal must wait for a disk, its future event is placed on the disk's wait queue. Once the grant time for the disk is known, the waiting terminal's future event activation time is set and the future event is moved from the wait queue to the heap. The time required for a disk I/O is calculated at random between 0 and 27 ms. This I/O time includes seek time, rotational latency, and transfer time.

The idea of selecting a disk at random for reading a page into the buffer is rather unrealistic since the required page may be stored on a specific disk. However, since pages are uniformly distributed among all the disks, selecting a disk at random for input produces the same result as using the requested page to determine which disk to use. Therefore, for simplicity, we select a disk at random

for page input.

We simulate infinite disks by creating only one disk and ignoring its status flag. Thus, with infinite disks, terminals never wait to use a disk.

2.3.4 CPUs

Terminals use a CPU to perform various tasks. These tasks and their cost in terms of CPU instructions are given in Table 2.1. These costs, except for the T-count access cost for tries, are the same as those used by Srinivasan and Carey.

Parameter	Description	Cost (in CPU instructions)
CC_CPU	CPU cost for a lock or unlock request	100
BUF_CPU	CPU cost for a buffer call	1000
PAGE_SEARCH_CPU	CPU cost for a page search	50
PAGE_MODIFY_CPU	CPU cost for a page modification	500
PAGE_COPY_CPU	CPU cost to copy a page between buffer and disk	1000
PAGE_COUNT_CPU	CPU cost to access T-count from memory (for trie only)	50

Table 2.1: CPU costs

To simulate a number of CPUs, we use a counter to indicate the number of CPUs currently in use and a wait queue. CPUs are granted to terminals on a FCFS basis. If the number of CPUs currently in use is less than the number of existing CPUs, the terminal is granted use of a CPU. If all the CPUs are in use at the time a terminal requests a CPU, the terminal's future event goes on the wait queue. As soon as a CPU becomes available, the waiting terminal's future event activation time is set and the future event is moved from the queue to the heap.

To simulate infinite CPUs, we simply allow an arbitrary number of CPUs to be in use at any given moment. Thus, with infinite CPUs, terminals never wait to use a CPU.

To summarize, the future event for a terminal indicates that terminal's current state and specifies what action will be taken next by the terminal. Future events for a terminal are either in the heap with a known activation time or on a wait queue with an unknown activation time. Once the grant time for the resource is known, the grant time is assigned to the future event and the event is moved from

the wait queue to the heap. Figure 2.1 indicates the location of future events during the concurrency simulation.

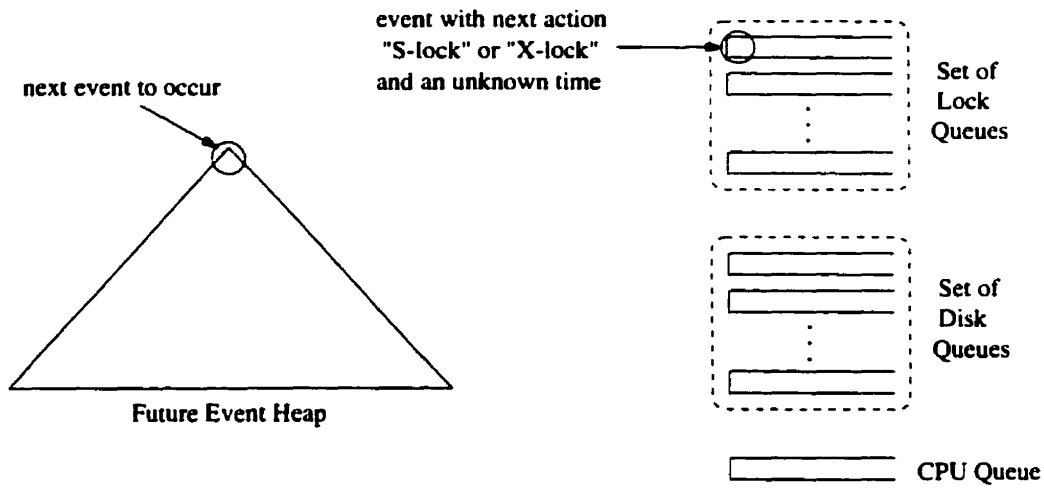


Figure 2.1: Location of future events during simulation

Chapter 3

B-Tree Concurrency Implementation

In this chapter, we describe the experiments used to evaluate the accuracy of our concurrency simulation. We then present and discuss the experimental results we obtained and compare and contrast them with published results.

3.1 Experimental Procedure

The B^{link} algorithms usually provide the best B-tree concurrency. Therefore, we use the throughput results obtained by Srinivasan and Carey [SC91b] for the B^{link} algorithm with lock-coupling on ascent as a basis for evaluating our own B-tree experiments. Unless otherwise noted, our method is the same as that performed by Srinivasan and Carey. In this chapter and Chapter 4, we use the term "B-tree" to refer to the B^{link}-tree. We also now refer to nodes as pages to simplify comparison with tries.

3.1.1 B-Tree Properties

We build a B-tree with a random permutation of all 40,000 odd integers valued between 0 and 80 000. A B-tree can have either a high fanout of 200 entries per page or a low fanout of 8 entries per page. The high-fanout B-tree has 3 levels and contains initially 3 non-leaf pages and 264 leaf pages. The low-fanout B-tree has 6 levels and contains initially 1464 non-leaf pages and 6999 leaf pages. Srinivasan and Carey's initial high-fanout B-tree contains 3 non-leaf pages and 260 leaf

pages, and their initial low-fanout B-tree contains around 1500 non-leaf pages and 7000 leaf pages.

3.1.2 B-Tree Operations

Searches use any key value between 0 and 80 000, inserts use only even key values between 0 and 80 000 (so that they are always successful), and appends use keys sequentially from 80 001 onwards. Unlike Srinivasan and Carey, we do not implement concurrent deletions for B-trees because our research focuses on concurrent trie search and insert operations.

Our experiments simulate either high data contention with 100% inserts or extremely high data contention with roughly 50% searches and 50% appends. We do not simulate any low data contention because Srinivasan and Carey use deletes along with inserts and searches in their low data contention simulations to achieve a steady state B-tree.

3.1.3 System Properties

We vary the system properties to simulate various situations. The buffer for the high-fanout B-tree can be either 200 pages, which puts 75% of the initial B-tree in memory, or an infinite number of pages, which creates an in-memory B-tree. We simulate an infinitely large buffer by simply assuming every page is in the buffer (except for new pages which must be placed in the buffer). Srinivasan and Carey instead simulate an in-memory B-tree by using a 600-page buffer. For the low-fanout B-tree, the buffer is 600 pages, which puts only 7% of the B-tree in memory. Srinivasan and Carey also implement an in-memory low-fanout B-tree, but we do not since Srinivasan and Carey do not present graphical results for this case.

To simulate various environments, we also vary the level of available resources. For B-trees that are not memory-resident, the use of disk I/O is necessary. In such a case, there are either 8 disks or an infinite number of disks. With infinitely many disks, no terminal ever waits for an available disk.

The other resource that we vary in number is the CPU. There can either be only 1 CPU or an infinite number of CPUs. As with the disk resource, with infinitely many CPUs, no terminal ever waits to use a CPU.

To sumarize, Table 3.1 contains the values we may use for the various parameters. Disk time is measured in terms of the number of CPU instructions that can be performed while the disk is in use.

The maximum disk time is 27 ms, which is the equivalent to performing 540,000 CPU instructions with a 20 MIPS CPU.

Parameter	Description	Values (in CPU instructions unless otherwise noted)
NUM_CPUS	Number of CPUs	1, ∞
NUM_DISKS	Number of disks	8, ∞
CPU_SPEED	in MIPS (millions of instructions per second)	20
DISK_TIME	Includes seek, latency, and transfer time (max 27 ms)	0..540000
CC_CPU	CPU cost for a lock or unlock request	100
BUF_CPU	CPU cost for a buffer call	1000
PAGE_SEARCH_CPU	CPU cost for a page search	50
PAGE_MODIFY_CPU	CPU cost for a page modification	500
PAGE_COPY_CPU	CPU cost to copy a page between buffer and disk	1000
FANOUT	Number of entries per B-tree page	8, 200
NUM_PAGE_LEVELS	Number of page levels	6, 3
INITIAL_NUM_KEYS	Number of keys in initial B-tree	40000
NUM_BUFFERS	Number of buffers	200, 600, ∞
MPL	Multiprogramming level (number of terminals)	1..300
NUM_OPERATIONS	Number of operations performed in each simulation	10000
SEARCH_PROB	Probability of search operation	0.0, 0.5
INSERT_PROB	Probability of insert operation	0.0, 1.0
APPEND_PROB	Probability of append operation	0.0, 0.5

Table 3.1: Parameters for B-tree simulations

3.1.4 Experiments

Srinivasan and Carey provide throughput curves for 7 of their experiments involving high and extremely high data contention. These 7 experiments are:

1. High-fanout, 100% inserts, infinite resources, in-memory
2. High-fanout, 100% inserts, infinite resources, 200 buffer pages
3. High-fanout, 100% inserts, 1 CPU, 8 disks, 200 buffer pages
4. Low-fanout, 100% inserts, infinite resources, 600 buffer pages

5. Low-fanout, 100% inserts, 1 CPU, 8 disks, 600 buffer pages

6. High-fanout, 50% appends, 50% searches, infinite resources, 200 buffer pages

7. High-fanout, 50% appends, 50% searches, 1 CPU, infinite disks, in-memory

Using the parameters in Table 3.1, we perform these 7 experiments as a measure of the correctness of our simulation.

3.2 Experimental Results

We now present and discuss our results for the B-tree concurrency experiments. We also compare and contrast our results with those obtained by Srinivasan and Carey. Each B-tree throughput curve we generate shows the mean throughput for 100 simulations. The error bars show standard deviation. To better understand our results, we break down the average number of CPU instructions for each operation during only 1 simulation into its various components.

3.2.1 B-Tree Experiment 1: High Fanout, 100% Inserts, Infinite Resources, and In Memory

The throughput curves we and Srinivasan and Carey obtained are in Figure 3.1. The breakdown of the average number of CPU instructions required by each operation is in Table 3.2.

Performing 10,000 insertions causes about 102 page splits. However, only about 40% to 45% of the page splits occur in the first 5,000 operations. So, since the B-tree initially has 267 pages, we will assume an average B-tree size of $(0.425 \times 102) + 267 \approx 310$ pages. There are not very many link chases. There is a maximum of only about 54 link chases per 10,000 operations. Therefore, for brevity, we omit them from our calculations below and only briefely note when they have a slight effect.

CC Requests

In the 3-level B-tree, insert operations usually S-lock and unlock 3 pages. After releasing its S-lock on a leaf page, the insert operation X-locks and unlocks the leaf. With the cost per concurrency

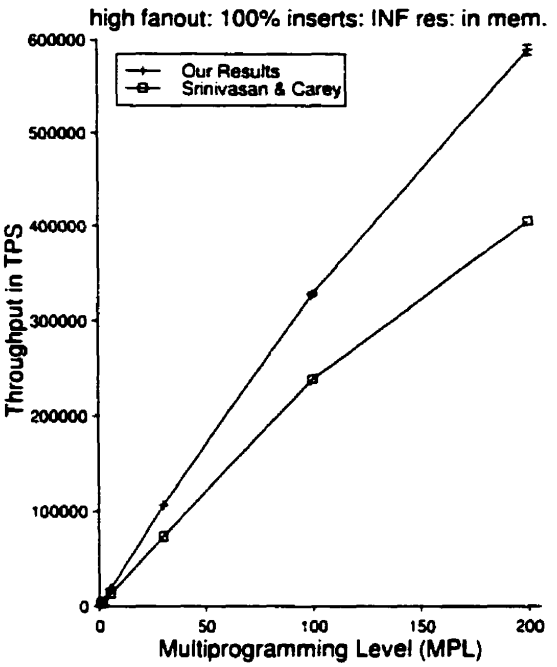


Figure 3.1: B-Tree Experiment 1 throughput

MPL	Request Type	Time (CPU Instructions)
1	CC	803
	BUF	4020
	PAGE SEARCH	151
	PAGE MODIFY	510
	Lock Wait	0
	Total	5484
5	CC	803
	BUF	4021
	PAGE SEARCH	151
	PAGE MODIFY	511
	Lock Wait	18
	Total	5503
30	CC	803
	BUF	4021
	PAGE SEARCH	151
	PAGE MODIFY	510
	Lock Wait	154
	Total	5639
100	CC	804
	BUF	4023
	PAGE SEARCH	151
	PAGE MODIFY	510
	Lock Wait	624
	Total	6111
200	CC	804
	BUF	4026
	PAGE SEARCH	151
	PAGE MODIFY	510
	Lock Wait	1280
	Total	6770

Table 3.2: B-tree Experiment 1 CPU usage per operation

control request being 100 instructions, we expect 800 instructions for these CC requests. The additional 3 instructions per operation arise from page splitting. Each page split yields 3 more CC requests: converting the X-lock on the leaf to an S-lock, and X-locking and unlocking the parent page. Averaged over 10,000 operations, 102 page splits yields about $102/10000 \times 3 \times 100 \approx 3$ more instructions per operation. Any Additional CC requests arise from link chases. So, as shown for CC requests in Table 3.2, there are about 803 instructions per operation used for concurrency control.

BUF Requests

With the B-tree having 3 page levels, operations usually access 3 pages from the buffer. An extra buffer access occurs after X-locking a leaf to ensure that the leaf is still the correct page to modify. Each page split yields 2 extra buffer accesses because the terminal places the new page in the buffer and accesses the parent page again from the buffer. So, for a cost of 1000 instructions per buffer request and about 102 page splits over 10,000 operations, we expect about $(2 \times 102/10000 + 4) \times 1000 \approx 4020$ instructions devoted to buffer requests. In Table 3.2, the instructions for buffer requests are as we expect, and increase slightly at higher MPL due to link chases.

PAGE_SEARCH Requests

Each insert operation searches 3 pages in the B-tree. After a page split, the operation must search the parent page. So, at a cost of 50 instructions per search and 102 page splits for 10,000 insertions, we expect the $(3 + 102/10000) \times 50 \approx 151$ instructions per operation for searching that Table 3.2 indicates. Each link chase causes an additional page to be searched but, with the very small number of link chases, the effects are minute.

PAGE_MODIFY Requests

Usually, operations modify only the leaf page. For each page split, 2 additional pages (the new page and the parent) are modified. Each modification costs 500 instructions, so for 102 page splits and 10,000 insertions, we expect $(2 \times 102/10000 + 1) \times 500 \approx 510$ instructions per operation for page

modification as shown in Table 3.2.

Terminals do not experience many conflicting lock modes in the B-tree. According to Table 3.2, the instructions each operation requires increases by a significantly lower factor than MPL. As a result, throughput increases greatly as MPL increases. For example, operations at an MPL of 200 require about 6770 instructions each, which is a factor of only about 1.1 more than the 6111 instructions required per operation at an MPL of 100. Comparing throughput curves in Figure 3.1, we are satisfied with our results for this experiment.

3.2.2 B-Tree Experiment 2: High Fanout, 100% Inserts, Infinite Resources, and 200 Buffers

The throughput curves we and Srinivasan and Carey obtained are in Figure 3.2. The breakdown of CPU instructions required for each operation is in Table 3.3. We do not discuss the results that are unaffected by the limitation of buffer size since they are explained in Section 3.2.1.

BUF Requests

The limitation of buffer size affects the number of buffer requests because the LRU page may need to be removed so that the requested page can be placed in the buffer. To determine how much of an effect occurs, we calculate the probability that a buffer miss occurs. The top 2 page levels have only 3 pages, so we assume that those 3 pages are always in the buffer. Therefore, for the average B-tree size of 310 pages and a buffer size of 200 pages, we estimate that the probability that a specific leaf page is in the buffer is $(200-3)/(310-3) \approx 0.64$ and that the probability of a buffer miss is about $1-0.64 \approx 0.36$. Each buffer miss yields 2 buffer calls due to writing-out and reading-in, so for each insert operation, we expect $0.36 \times 2 \approx 0.72$ buffer requests (requiring about 720 instructions) due to buffer misses. Adding that to the 4020 instructions each operation needs for page navigation and splitting (which we calculated in Section 3.2.1), we expect about 4740 instructions per operation for buffer requests, which is close to the buffer request measurement in Table 3.3.

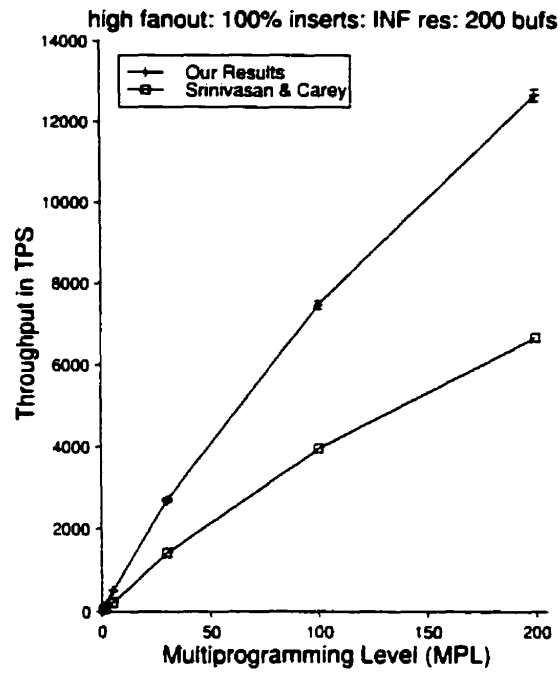


Figure 3.2: B-tree Experiment 2 throughput

PAGE_COPY Requests

Each time the terminal encounters a buffer miss, it accesses a disk twice: once to write-out the LRU page and once to read-in the requested page. Each disk access generates a PAGE_COPY request, so we expect 0.72 PAGE_COPY requests based on our calculation for buffer requests. In addition, each page split yields a PAGE_COPY request because the LRU page must be copied to disk to make room for the new page. At about 102 splits over 10,000 operations, that amounts to an additional 0.01 PAGE_COPY requests per operation. Each PAGE_COPY request costs 1000 instructions, so we expect an average of $(0.72 + 0.01) \times 100 \approx 730$ instructions per operation, which is close to the PAGE_COPY request measurements in Table 3.3.

Disk Time

The disk time varies from 0 to 27 ms, which is the equivalent to performing 0 to 540000 CPU instructions with a CPU rated at 20 MIPS. As explained above, there are 2 disk accesses for each

MPL	Request Type	Time (CPU Instructions)
1	CC	803
	BUF	4747
	PAGE SEARCH	151
	PAGE MODIFY	511
	PAGE COPY	726
	Disk Time	196894
	Lock Wait	0
	Total	203831
5	CC	803
	BUF	4705
	PAGE SEARCH	151
	PAGE MODIFY	510
	PAGE COPY	685
	Disk Time	184865
	Lock Wait	3893
	Total	195610
30	CC	803
	BUF	4722
	PAGE SEARCH	151
	PAGE MODIFY	510
	PAGE COPY	701
	Disk Time	188689
	Lock Wait	25829
	Total	221404

MPL	Request Type	Time (CPU Instructions)
100	CC	804
	BUF	4715
	PAGE SEARCH	151
	PAGE MODIFY	510
	PAGE COPY	691
	Disk Time	187969
	Lock Wait	72970
	Total	267809
200	CC	804
	BUF	4709
	PAGE SEARCH	151
	PAGE MODIFY	510
	PAGE COPY	682
	Disk Time	187366
	Lock Wait	123151
	Total	317372

Table 3.3: B-tree Experiment 2 CPU usage per operation

buffer miss and 1 disk access for each page split; hence, we expect 0.73 disk accesses per operation. At an average disk time equivalent to performing $540000/2 \approx 270000$ instructions, each operation devotes a time equal to performing about $0.73 \times 270000 \approx 197100$ instructions for disk I/O as shown in Table 3.3.

Even with the addition of disk I/O, terminals still do not experience many conflicting lock modes. Therefore, throughput continues to rise as MPL increases. At its lowest rate of throughput increase, which occurs when MPL increases from 100 to 200, the number of instructions required by each operation increases by a factor of only about 1.2 from 267809 to 317372. Comparing the throughput curves in Figure 3.2, we are satisfied that our simulation performs correctly for this experiment.

3.2.3 B-Tree Experiment 3: High Fanout, 100% Inserts, 1 CPU, 8 Disks, and 200 Buffers

Our throughput curves and those of Srinivasan and Carey are in Figure 3.3. We break down the number of CPU instructions required for each operaration into the various components in Table 3.4. We do not discuss the results that are unaffected by the limitation on the number of CPUs and disks. The CC, PAGE_SEARCH, and PAGE_MODIFY results the same as those in Section 3.2.1 and the BUF results are the same as those in Section 3.2.2.

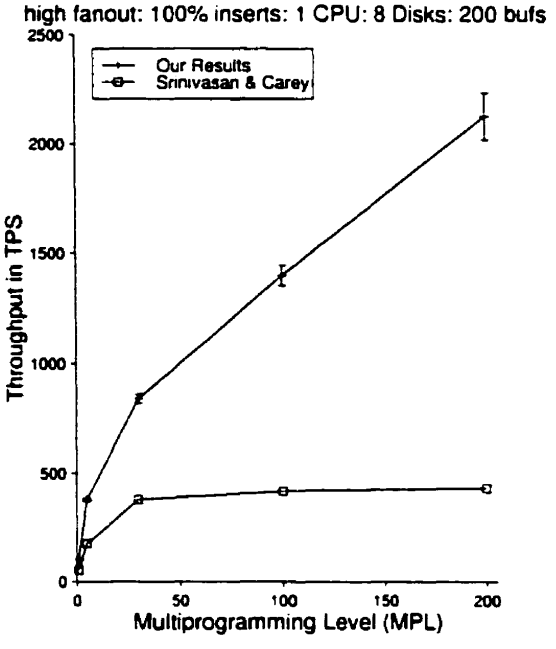


Figure 3.3: B-tree Experiment 3 throughput

PAGE_COPY Requests

Initially, the disk accesses and PAGE_COPY requests are as calculated in Section 3.2.2. However, as MPL increases, the number of disk accesses and PAGE_COPY requests decreases. Pending I/O for any given page may be for a longer period of time now because terminals may need to wait to use a disk. As MPL increases, the probability that there is already pending I/O for the page not

MPL	Request Type	Time (CPU Instructions)
1	CC	803
	BUF	4712
	PAGE SEARCH	151
	PAGE MODIFY	510
	PAGE COPY	692
	Disk Time	184436
	Disk Wait	0
	CPU Wait	0
	Lock Wait	0
	Total	191304
5	CC	803
	BUF	4704
	PAGE SEARCH	151
	PAGE MODIFY	510
	PAGE COPY	684
	Disk Time	185261
	Disk Wait	63005
	CPU Wait	899
	Lock Wait	3216
	Total	259232
30	CC	803
	BUF	4623
	PAGE SEARCH	151
	PAGE MODIFY	510
	PAGE COPY	602
	Disk Time	272098
	Disk Wait	434024
	CPU Wait	5315
	Lock Wait	14395
	Total	732521

MPL	Request Type	Time (CPU Instructions)
100	CC	804
	BUF	4419
	PAGE SEARCH	151
	PAGE MODIFY	510
	PAGE COPY	393
	Disk Time	600467
	Disk Wait	711876
	CPU Wait	30664
	Lock Wait	50890
	Total	1400173
200	CC	806
	BUF	4306
	PAGE SEARCH	151
	PAGE MODIFY	511
	PAGE COPY	269
	Disk Time	813785
	Disk Wait	653324
	CPU Wait	145219
	Lock Wait	147281
	Total	1765653

Table 3.4: B-tree Experiment 3 CPU usage per operation

found in the buffer by a terminal increases. In such a case, disk I/O for the page occurs only once, even though multiple terminals encountered a buffer miss for the page.

Disk Time and Disk Wait

With the limitation on the number of disks, the wait time for a disk increases as MPL increases. The time spent waiting for pending I/O is added to disk time because, for infinite disks, a terminal may be waiting for pending I/O, but not for a disk. Since pending I/O times increase as mentioned above for PAGE_COPY requests, disk time increases.

CPU Wait and Lock Wait

From Table 3.4, we see that the CPU and lock wait times increase expontially due to the limitation on CPU resources and the fact that locks are held longer since they're held while waiting for disk resources.

Throughput for our simulation continues to rise as MPL increases because the number of instructions required for each operation does not increase by a very large factor. For example, even though MPL increases by a factor of 2 between 100 and 200 terminals, the number of instructions per operation increases from 1400173 to 1765653, which is a factor of only about 1.3.

Our results in Figure 3.3 differ from those of Srinivasan and Carey. Srinivasan and Carey state in [SC91b] that B-link algorithms saturate the disks at high MPL. Since the disk times and disk wait times for our results in Table 3.4 increase only logarithmically as MPL increases, our disk usage is likely somehow different from that which they simulated.

3.2.4 B-Tree Experiment 4: Low Fanout, 100% Inserts, Infinite Resources, and 600 Buffers

Our throughput curves and those of Srinivasan and Carey are in Figure 3.4. The CPU usage for each operation is in Table 3.5.

Performing 10,000 insertions in the low-fanout B-tree causes about 2100 page splits. Half of the page splits occur in the first 5,000 insertions, so, since the initial B-tree has 8463 pages, we assume an average B-tree size of $2100/2 + 8563 \approx 9513$ pages. As with the high-fanout B-tree simulations, there are not very many link chases. There is only a maximum of about 65 link chases per 10,000 operations. For brevity, we omit them from our calculations and only briefly note when they have a slight effect.

CC Requests

The B-tree has 6 page levels, so insert operations S-lock and unlock 6 pages on their way from root to leaf. After releasing the S-lock on the leaf, the terminal will X-lock and unlock the leaf.

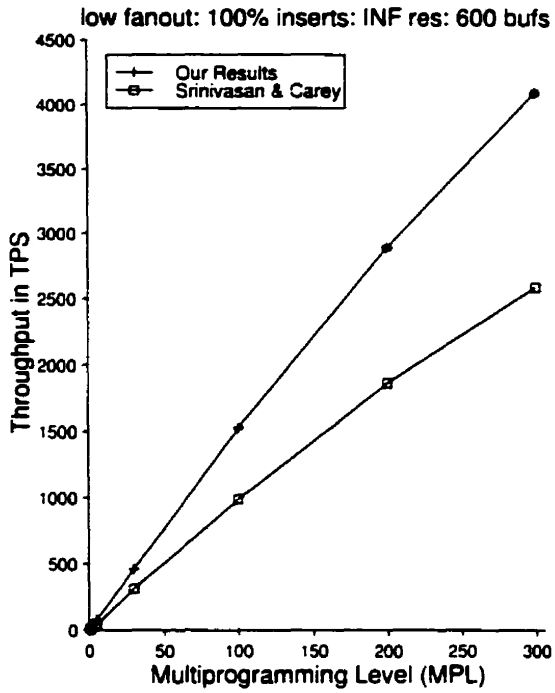


Figure 3.4: B-tree Experiment 4 throughput

Also, we expect $2100/10000 \approx 0.21$ page splits on average per operation and, since each page split yields 3 more CC requests, a total of 0.63 CC requests per operation due to page splitting. At 100 instructions per CC request, we then expect $(14 + 0.63) \times 100 \approx 1463$ instructions per operation for CC requests, as indicated in Table 3.5.

BUF Requests

Operations access 6 pages from the buffer since the B-tree has 6 page levels. Each page split generates 2 buffer calls, so for 0.21 page splits on average per operation, we expect an additional 0.42 buffer calls due to page splitting. We estimate that, due to the buffer size of 600 pages and average B-tree size of 9513 pages, $600/9513 \approx 0.063$ is the probability that a specific page is in the buffer. We will use this probability even when assuming upper pages of the B-tree are always in the buffer since this assumption makes very little difference in the probability.

With the 6-level B-tree, it is more difficult to be certain which pages are always in the buffer. We

MPL	Request Type	Time (CPU Instructions)
1	CC	1462
	BUF	12124
	PAGE SEARCH	310
	PAGE MODIFY	707
	PAGE COPY	4711
	Disk Time	1274000
	Lock Wait	0
	Total	1293314
5	CC	1463
	BUF	12118
	PAGE SEARCH	310
	PAGE MODIFY	709
	PAGE COPY	4700
	Disk Time	1267638
	Lock Wait	51
	Total	1286989
30	CC	1463
	BUF	12153
	PAGE SEARCH	311
	PAGE MODIFY	710
	PAGE COPY	4732
	Disk Time	1277944
	Lock Wait	1260
	Total	1298572

MPL	Request Type	Time (CPU Instructions)
100	CC	1462
	BUF	12157
	PAGE SEARCH	310
	PAGE MODIFY	705
	PAGE COPY	4743
	Disk Time	1286619
	Lock Wait	5455
	Total	1311452
200	CC	1464
	BUF	12419
	PAGE SEARCH	311
	PAGE MODIFY	708
	PAGE COPY	4997
	Disk Time	1354797
	Lock Wait	13731
	Total	1388426
300	CC	1466
	BUF	12664
	PAGE SEARCH	311
	PAGE MODIFY	712
	PAGE COPY	5229
	Disk Time	1416964
	Lock Wait	28025
	Total	1465372

Table 3.5: B-tree Experiment 4 CPU usage per operation

estimate that between 3 and 4 of the upper B-tree page levels are always in the buffer. If 3 of the upper levels are always in the buffer, we expect the number of buffer misses per operation to be about $(0.063^2 \times 0.937 \times 3 \times 1) + (0.063 \times 0.937^2 \times 3 \times 2) + (0.937^3 \times 1 \times 3) \approx 2.81$. If 4 of the upper levels are always in the buffer, we expect there to be about $(0.063 \times 0.937 \times 2 \times 1) + (0.937^2 \times 1 \times 2) \approx 1.87$ buffer misses per operation. Averaging the 2 probabilities, we then expect about 2.34 buffer misses per operation. Each buffer miss generates 2 buffer calls, so we expect about $2.34 \times 2 \approx 4.68$ additional buffer calls per operation. So, in total, we expect about $6 + 1 + 0.42 + 4.68 \approx 12.1$ buffer calls per operation. At 1000 instructions per call, we arrive at about 12100 instructions required for buffer requests, which is very close to the BUF request costs in Table 3.5.

PAGE_SEARCH Requests

Each operation searches 6 pages plus, with an expected 0.21 page splits per operation, an additional 0.21 pages. At 500 instructions per search, we expect $6.21 \times 500 \approx 310.5$ instructions devoted to page searching as indicated in Table 3.5.

PAGE_MODIFY Requests

Each operation modifies 2 pages every time a page split occurs in addition to modifying the leaf page. Hence, for an expected 0.21 page splits per operation, we expect $(2 \times 0.21) + 1 \approx 1.42$ modifications per operation. At 500 instructions per modification, we expect 710 instructios to be devoted to page modification as indicated in Table 3.5.

PAGE_COPY Requests

We estimated about 2.34 buffer misses per operation above when calculating buffer usage. Each buffer miss generates 2 disk accesses, so we expect about 4.68 disk accesses per operation. Adding 0.21 page splits per operation yields $4.68 + 0.21 \approx 4.89$ disk accesses per operation. Each disk access requires a PAGE_COPY request so, at 1000 instructions per PAGE_COPY request, we expect about 4890 instructions total per operation due to PAGE_COPY requests.

Disk Time

At about 4.89 disk access per operation and an average disk access time equivalent to performing 270000 instructions, we expect a time equivalent to performing about 1320300 instructions to be used for disk I/O.

As with the 3-level B-tree, terminals do not experience many conflicting lock modes. As a result, throughput continues to increase greatly as MPL increases. At the lowest rate of throughput increase, which occurs when MPL increases from 100 to 200, the number of instructions each operation requires increases by only a factor of about 1.1 from 1388426 to 1465372. Based on the throughput curves in Figure 3.4, we are satisfied with our results for this experiment.

3.2.5 B-Tree Experiment 5: Low Fanout, 100% Inserts, 1 CPU, 8 Disks, and 600 Buffers

Our throughput curves and those of Srinivasan and Carey are in Figure 3.5. We also present the link chase data for this and the equivalent high-fanout B-tree simulation of Section 3.2.3 in Figure 3.6. We break down the CPU usage for an operation into its various components in Table 3.6. We do not discuss the results that are unaffected by the limits imposed on the number of CPUs and disks since they are explained in Section 3.2.4.

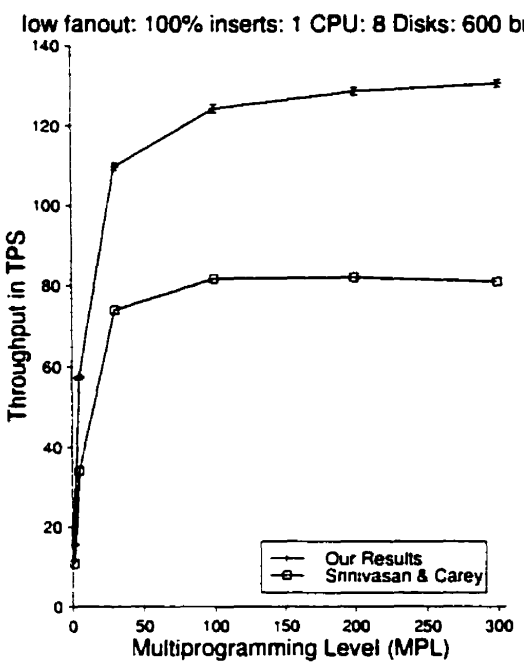


Figure 3.5: B-Tree Experiment 5 throughput

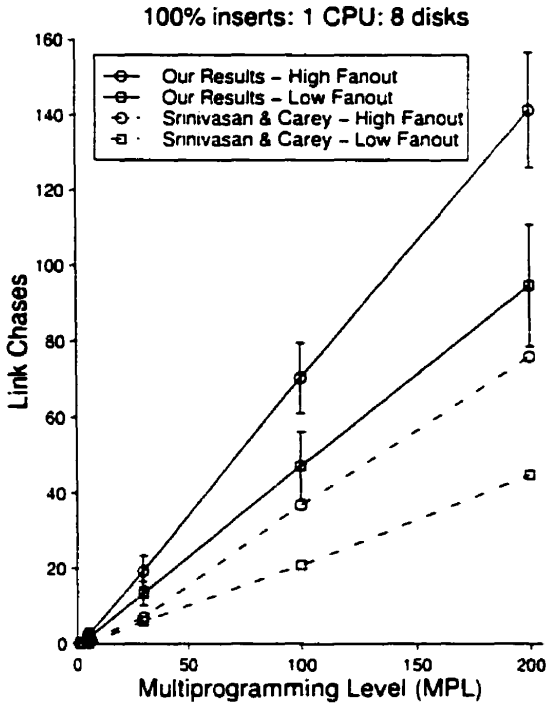


Figure 3.6: B-tree Link Chases

PAGE_COPY Requests

The disk accesses and PAGE_COPY requests are as calculated in Section 3.2.4 but, as with the 3-level B-tree, disk accesses and PAGE_COPY requests decrease in number as MPL increases. As explained in Section 3.2.3, this decrease is as a result of more terminals that encounter a buffer miss waiting for pending disk I/O and not performing any disk I/O themselves. The decrease in disk I/O

MPL	Request Type	Time (CPU Instructions)
1	CC	1463
	BUF	12147
	PAGE SEARCH	310
	PAGE MODIFY	710
	PAGE COPY	4728
	Disk Time	1277722
	Disk Wait	0
	CPU Wait	0
	Lock Wait	0
	Total	1297081
5	CC	1461
	BUF	12106
	PAGE SEARCH	310
	PAGE MODIFY	707
	PAGE COPY	4692
	Disk Time	1265706
	Disk Wait	434376
	CPU Wait	590
	Lock Wait	400
	Total	1720350
30	CC	1463
	BUF	12068
	PAGE SEARCH	310
	PAGE MODIFY	708
	PAGE COPY	4650
	Disk Time	1325745
	Disk Wait	4085154
	CPU Wait	2186
	Lock Wait	4274
	Total	5436559

MPL	Request Type	Time (CPU Instructions)
100	CC	1464
	BUF	11986
	PAGE SEARCH	311
	PAGE MODIFY	711
	PAGE COPY	4561
	Disk Time	1953133
	Disk Wait	14107909
	CPU Wait	10085
	Lock Wait	57221
	Total	16147379
200	CC	1463
	BUF	11916
	PAGE SEARCH	311
	PAGE MODIFY	704
	PAGE COPY	4500
	Disk Time	3471840
	Disk Wait	27328633
	CPU Wait	35547
	Lock Wait	263450
	Total	31118362
300	CC	1465
	BUF	11954
	PAGE SEARCH	311
	PAGE MODIFY	708
	PAGE COPY	4522
	Disk Time	5231190
	Disk Wait	39346353
	CPU Wait	75443
	Lock Wait	731293
	Total	45403241

Table 3.6: B-tree Experiment 5 CPU usage per operation

is smaller than that for the 3-level B-tree though because there are many more pages in the 6-level B-tree and the probability of 2 terminals encountering the same page is lower.

Disk Time and Disk Wait

Table 3.6 indicates that disk times and waits increase dramatically as MPL increases. There are many more buffer misses than with the equivalent high-fanout B-tree simulation in Section 3.2.3, so the terminals compete much more for disk resources.

CPU Wait and Lock Wait

As with the high-fanout B-tree in Section 3.2.3, Table 3.6 shows that CPU and lock waiting times increase expontially as MPL increases.

Throughput begins to maximize at an MPL of about 30, unlike our similar experiment with the high-fanout B-tree in Section 3.2.3. The low-fanout B-tree requires much more disk usage than the high-fanout B-tree because many more pages are not in the buffer. For the low-fanout B-tree, primarily the disk usage causes operation times to increase greatly as MPL increases. So, as with Srinivasan and Carey, we are able to saturate the disks at high MPL with the low-fanout B-tree.

In addition to the throughput curves, we also presented link chase data in Figure 3.6 for both the high and low-fanout B-trees. Our simulations perform more link chases per 10,000 operations, but, as with Srinivasan and Carey's simulations, the high-fanout B-tree performs more link chases than the low-fanout B-tree. So, even though our high-fanout B-tree simulation with limited resources in Section 3.2.3 did not produce the same throughput curve as Srinivasan and Carey's simulation, we are satisfied that our B-tree operations perform correctly and that it is the system that is modeled slightly differently.

3.2.6 B-Tree Experiment 6: High Fanout, 50% Appends, 50% Searches, Infinite Resources, and 200 Buffers

Our throughput curves and those of Srinivasan and Carey are in Figure 3.7. Table 3.7 contains the components of the CPU usage for each operation.

Performing roughly 5,000 appends and 5,000 searches yields about 49 page splits. Half of the page splits occur in the first 5,000 operations, so, since the initial B-tree has 267 pages, we assume an average B-tree size of $49/2 + 267 \approx 292$ pages.

CC Requests

Searches always S-lock and unlock 3 pages in the 3-level B-tree for a total of 6 CC requests. Appends S-lock and unlock 3 pages, as well as X-lock and unlock the leaf for a total of 8 CC requests.

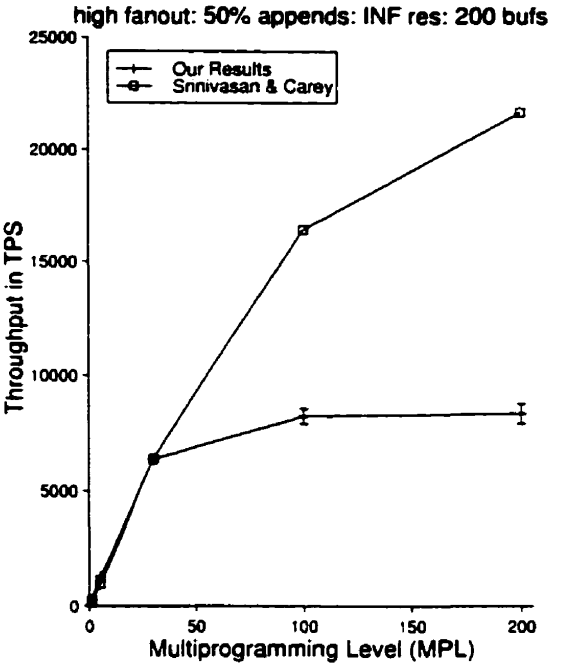


Figure 3.7: B-tree Experiment 6 throughput

Each page split yields 3 CC requests, so for 49 page splits over 10,000 operations, we expect a total of about $49/10000 \times 3 \approx 0.015$ CC requests per operation due to page splits. Since there are about 50% of each operation, we expect $(6 + 8)/2 + 0.015 \approx 7.015$ CC requests on average per operation. At 100 instructions per CC request, we expect about 701.5 instructions per operation for CC requests.

Link chases cause more CC requests to occur. As MPL increases, so do link chases. At an MPL of 200 for example, there are about 8370 link chases. Each link chase causes 2 additional CC requests so, over the 10,000 operations, we expect $(8370 \times 2)/10000 + 7.015 \approx 8.689$ CC requests on average per operation. At 100 instructions per CC request, this equals about 869 instructions required for CC requests as shown in Table 3.7 for the MPL of 200.

BUF Requests

Since appends create heavy traffic on the rightmost pages of the B-tree, we will assume that those pages are always in the buffer. Hence, appends use the disk only when a page splits. Since appends

MPL	Request Type	Time (CPU Instructions)
1	CC	703
	BUF	3770
	PAGE SEARCH	150
	PAGE MODIFY	258
	PAGE COPY	255
	Disk Time	67561
	Lock Wait	0
	Total	72696
5	CC	703
	BUF	3776
	PAGE SEARCH	151
	PAGE MODIFY	256
	PAGE COPY	257
	Disk Time	68461
	Lock Wait	1368
	Total	74972
30	CC	714
	BUF	3846
	PAGE SEARCH	153
	PAGE MODIFY	255
	PAGE COPY	275
	Disk Time	77135
	Lock Wait	19901
	Total	102279

MPL	Request Type	Time (CPU Instructions)
100	CC	772
	BUF	4130
	PAGE SEARCH	168
	PAGE MODIFY	251
	PAGE COPY	269
	Disk Time	74124
	Lock Wait	166581
	Total	246294
200	CC	869
	BUF	4613
	PAGE SEARCH	192
	PAGE MODIFY	256
	PAGE COPY	264
	Disk Time	72153
	Lock Wait	393795
	Total	472142

Table 3.7: B-tree Experiment 6 CPU usage per operation

are working with values outside the range of searches and operate on sequentially higher key values, we will also assume that once traffic on a page modified by appends ceases, it quickly becomes LRU and written out of the buffer. Therefore, we will assume that only the original 267 pages plus the 1 page currently being modified by the appends have a chance of being in the buffer. As a result, the probability that a search operation encounters a buffer miss is about $1 - 200/268 \approx 0.254$. Each buffer miss yields 2 additional buffer calls, so we expect searches to make $(0.254 \times 2) + 3 \approx 3.508$ buffer calls per search. Appends perform 4 buffer calls when there are no page splits. Each page split causes 2 buffer calls so, for the 49 splits over 10,000 operations, we add $49/10000 \times 2 \approx 0.01$ buffer calls per operation. So in total, we expect $(4 + 3.508)/2 + 0.01 \approx 3.76$ buffer calls per operation.

However, each link chase causes 1 more buffer request. For example, at an MPL of 200 with 8370 link chases, we expect an additional $8370/10000 \approx 0.837$ buffer calls due to link chases. At 1000 instructions per call, each operation at an MPL of 200 requires $(3.76+0.837) \times 1000 \approx 4597$,

which is a close estimate of the time needed for buffer requests in Table 3.7 for 200 MPL.

PAGE_SEARCH Requests

Aside from link chases, both searches and appends search 3 pages which, at 50 instructions per page search, amounts to 150 instructions per operation for page searching. Each link chase generates another page search. So, at an MPL of 200 for example, with about 8370 link chases over the 10,000 operations, we expect another $8370/10000 \times 50 \approx 42$ additional instructions per operation for searches resulting from link chases. Adding 42 to 150 yields the 192 instructions for searching for an MPL of 200 in Table 3.7.

PAGE_MODIFY Requests

The appends modify only 1 page unless there is a page split. Page splits generate 2 more page modifications. Since only about half of the operations result in a page modification and there are about 49 page splits, we expect $0.5 + (2 \times 49/10000) \approx 0.51$ page modifications per operation. At 500 instructions per page modification, we expect about 255 instructions per operation for page modification, which is very close to the page modification measurements in Table 3.7.

PAGE_COPY Requests

We estimated in the calculation of buffer requests that the probability of a search operation encountering a buffer miss is 0.254. Since only 50% of the operations are searches, we expect only $0.5 \times 0.254 \approx 0.127$ buffer misses per operation. Each buffer miss yields 2 PAGE_COPY requests and each page split yields 1 PAGE_COPY request. Therefore, we expect $(0.127 \times 2) + 49/10000 \approx 0.259$ PAGE_COPY requests per operation. At 1000 instructions per request, this amounts to 259 instructions per operation for PAGE_COPY requests as in Table 3.7.

Disk Time

Each time terminal makes a PAGE_COPY request, it performs disk I/O. Since we expect about 0.259 PAGE_COPY requests and average disk time is equivalent to performing 270000 instructions, we

expect disk time to be equivalent to performing about $0.259 \times 270000 \approx 69930$ instructions.

Our throughput curve is somewhat different than that of Srinivasan and Carey. Our simulation performs very many link chases – about 8370 at a 200 MPL. Srinivasan and Carey do not publish link chase data for their experiment, so it is difficult to compare in this regard. Since our link chases are more numerous for 100% insert simulations, as shown in Figure 3.6 on page 57, perhaps our simulation performs many more link chases than theirs. With the extremely high data contention, an increase in link chases may cause the significant reduction in throughput.

3.2.7 B-Tree Experiment 7: High Fanout, 50% Appends, 50% Searches, 1 CPU, and in memory

Our throughput curves and those of Srinivasan and Carey are in Figure 3.8. We also present the link chase data in Figure 3.9. The breakdown of CPU usage for each operation is in Table 3.8. We do not discuss results that are unchanged from those explained in Section 3.2.6. as a result of using 1 CPU and infinite buffer space.

CC Requests

Concurrency control requests are calculated as in Section 3.2.6, except that link chases are fewer. For example, with about 5500 link chases at an MPL of 200, we expect $(5500 \times 2)/10000 + 7.015 \approx 8.115$ CC requests on average per operation. At 100 instructions per request, this equals about 811 instructions per operation for CC requests as indicated in Table 3.8.

BUF Requests

The B-tree is in memory, so terminals perform no disk I/O. Therefore searches always make 3 buffer calls. Appends make at least 4 buffer calls. If a page splits, the terminal will make an additional buffer call to write-out the LRU page from the buffer. Also, each time an operation performs a link chase, it makes an additional buffer call. So with about 49 page splits, we expect an average of $(3 + 4)/2 + 49/10000 \approx 3.5$ buffer calls or, at a cost of 1000 instructions per buffer call, 3500

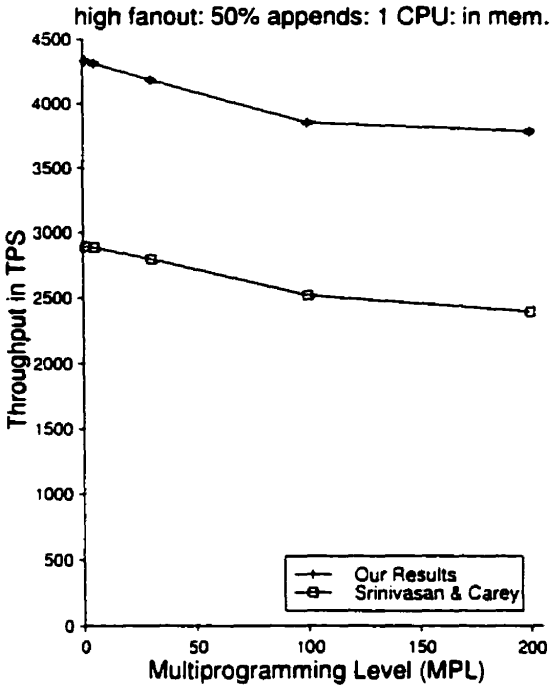


Figure 3.8: B-tree Experiment 7 throughput

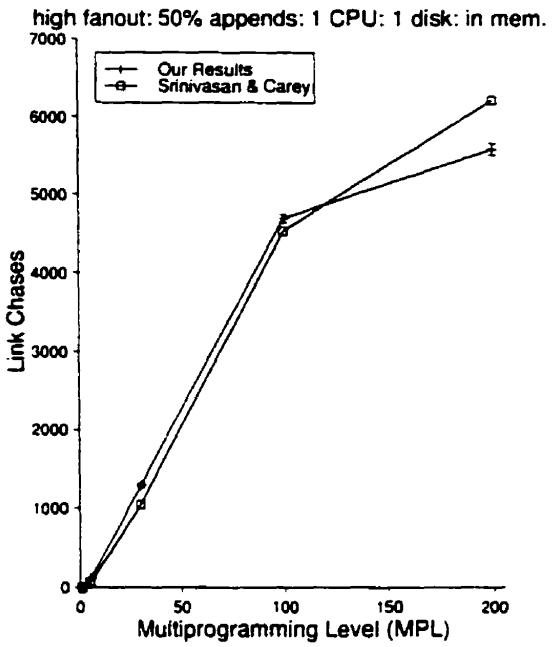


Figure 3.9: B-tree Experiment 7 link chases

instructions needed for buffer calls if there are no link chases. At an MPL of 200, there are about 5500 link chases. In this case, we expect $3.5 + 5500/10000 \approx 4.05$ buffer calls, which requires about 4050 instructions as shown in Table 3.8.

PAGE_SEARCH Requests

The number of pages searches is calculated as in Section 3.2.6, except that link chases are fewer. For example, with about 5500 link chases at an MPL of 200, we expect $5500/10000 \times 50 \approx 27.5$ instructions for page searches due to link chases. Adding this to the 150 instructions for regular page searches yields about 177.5 instructions required for page searching at a 200 MPL as indicated in Table 3.8.

As MPL increases, the number of instructions required by each operation increases by a greater factor than MPL. For example, when MPL increases from 100 to 200, the number of instructions each operation requires increases from 518576 to 1054889, which is a factor of about 2.03. Thus,

MPL	Request Type	Time (CPU Instructions)
1	CC	700
	BUF	3504
	PAGE SEARCH	150
	PAGE MODIFY	252
	CPU Wait	0
	Lock Wait	0
	Total	4607
5	CC	703
	BUF	3519
	PAGE SEARCH	151
	PAGE MODIFY	255
	CPU Wait	16103
	Lock Wait	2456
	Total	23186
30	CC	726
	BUF	3631
	PAGE SEARCH	157
	PAGE MODIFY	252
	CPU Wait	29138
	Lock Wait	109223
	Total	143127

MPL	Request Type	Time (CPU Instructions)
100	CC	793
	BUF	3970
	PAGE SEARCH	173
	PAGE MODIFY	253
	CPU Wait	38285
	Lock Wait	475101
	Total	518576
200	CC	811
	BUF	4058
	PAGE SEARCH	178
	PAGE MODIFY	254
	CPU Wait	65266
	Lock Wait	984323
	Total	1054889

Table 3.8: B-tree Experiment 7 CPU usage per operation

throughput decreases as MPL increases.

According to Figures 3.8 and 3.9, our simulation results resemble those of Srinivasan and Carey very closely, so we are satisfied that our simulation operates correctly for this experiment.

3.3 Summary of Results

Our goal was to achieve a single constant ratio between our results and those of Srinivasan and Carey. By doing so, we could scale our trie concurrency results to fit onto the graphs published by Srinivasan and Carey. Our results show that:

1. There are 2 experiments where the ratio is not constant and varies as MPL increases.

2. There are different constant ratios.

For Experiments 3 and 6, our throughput behaviour differs significantly from Srinivasan and Carey's. The ratio of our throughput to theirs for these 2 experiments varies as shown in Figures 3.10 and 3.11.

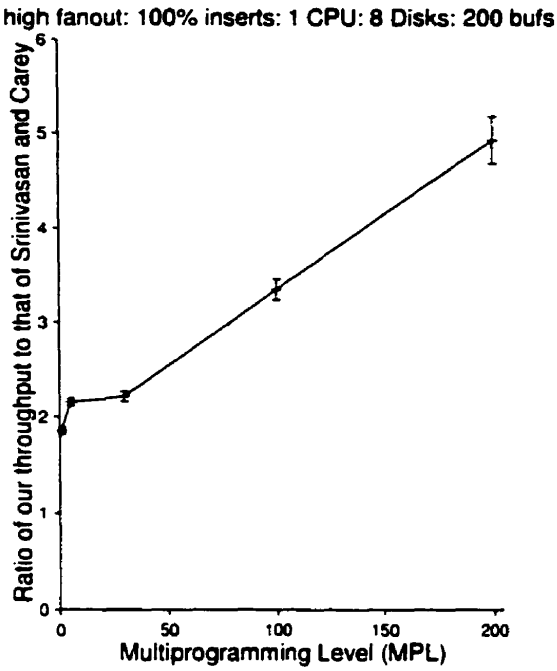


Figure 3.10: Experiment 3 throughput ratio

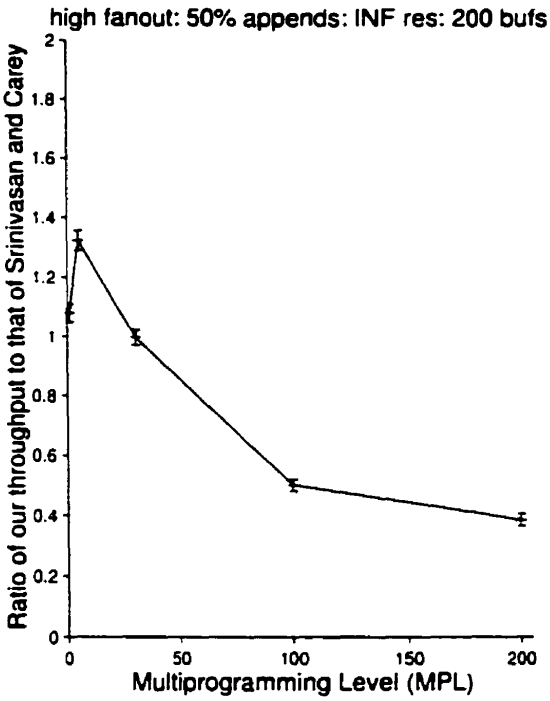


Figure 3.11: Experiment 6 throughput ratio

For each of the other experiments, the ratio between our throughput and Srinivasan and Carey's is fairly constant. However, these constant ratios vary from one experiment to another. The minimum constant ratio we generate is about 1.4, which is for Experiment 1; whereas, the maximum constant ratio we generate is about 1.9, which is for Experiment 2.

Nevertheless, most of our throughput curves behave similarly to those that Srinivasan and Carey have published. We are satisfied that slight differences in the our implementation of system resources from the implementation of Srinivasan and Carey account for the differences in throughput. We therefore use our simulation to study the behaviour of trie concurrency.

Chapter 4

Trie Concurrency Implementation

In this chapter, we present algorithms for concurrent trie searches and insertions. We describe the tries and parameters used for concurrency experimentation. We also present and discuss experimental results that we obtained and compare them to the B-tree results presented in Chapter 3.

4.1 Trie Concurrency Control Algorithms

Unlike B-trees, where recovery during concurrent operations merely requires advancing to the right to reach the correct page, tries seemingly require multiple recovery methods. Concurrent trie operations may need to recover by advancing to pages either to the left or to the right. In addition, situations may arise when, even if an operation reaches a correct page, the page will not be navigated correctly. Three examples follow to illustrate the various recoveries that may be necessary when using tries.

Example 1

There are two transactions:

- Transaction 1: read `10101100`
- Transaction 2: write `01001010`

These transactions are executed in the schedule given by Figure 4.1.

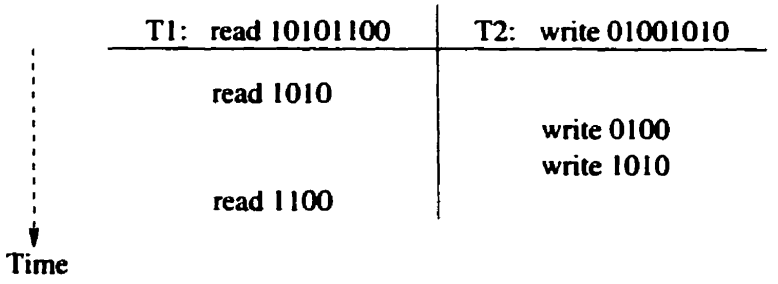


Figure 4.1: Transaction schedule for Example 1

Figures 4.2 to 4.4 show the execution of the transaction schedule.

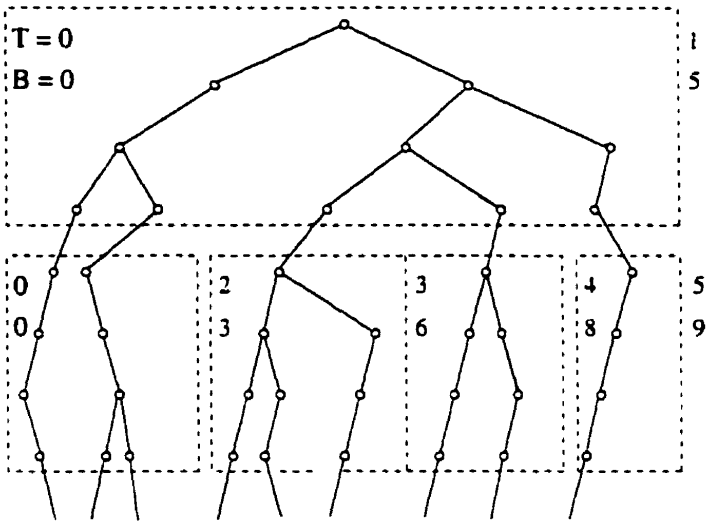


Figure 4.2: Initial trie for Examples 1 and 2

In Figure 4.2, Transaction 1 reads 1010 in the root page. To find the remainder of its key, it needs to read the page to the left of the page with minimum T such that $T \geq B_{root} + srch$ or $T \geq 0 + 4$ on page level 2, as explained earlier in Section 1.3.2.

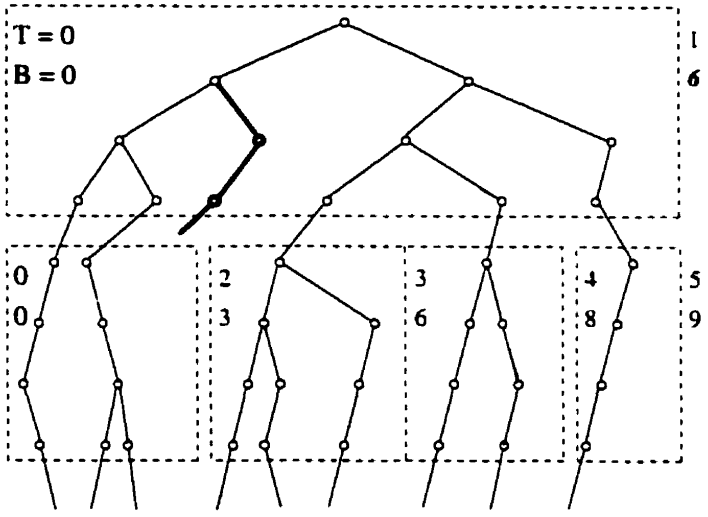


Figure 4.3: Partially modified trie for Example 1

In Figure 4.3, Transaction 2 has inserted 0100 into the root page and modified the B-count for the root page level.

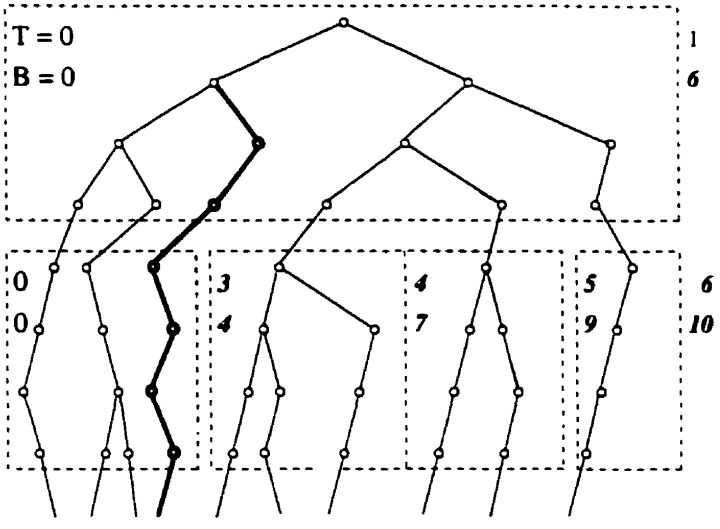


Figure 4.4: Fully modified trie for Example 1

In Figure 4.4, Transaction 2 has inserted the remainder of its key, 1010, and modified the counts for the second page level. Transaction 1 finds the page with minimum $T \geq 4$ and searches the page immediately to the left with $T = 3$. This is the incorrect page and Transaction 1 must recover

by moving to the right. The failure occurs because Transaction 1 uses T-counts that result from Transaction 2's execution, but not a *srch* count that results from Transaction 2's execution.

Example 2

Consider again the trie in Figure 4.2. The same two transactions are executed, but in the schedule given by Figure 4.5:

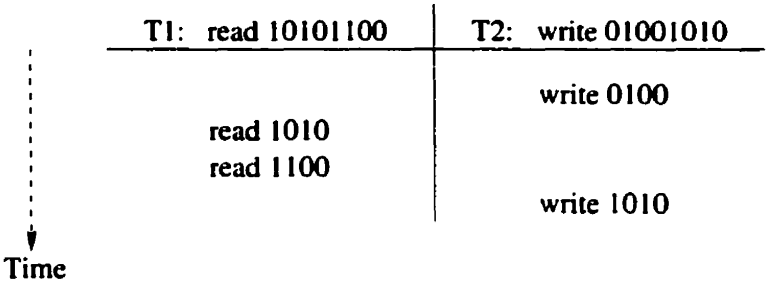


Figure 4.5: Transaction schedule for Example 2

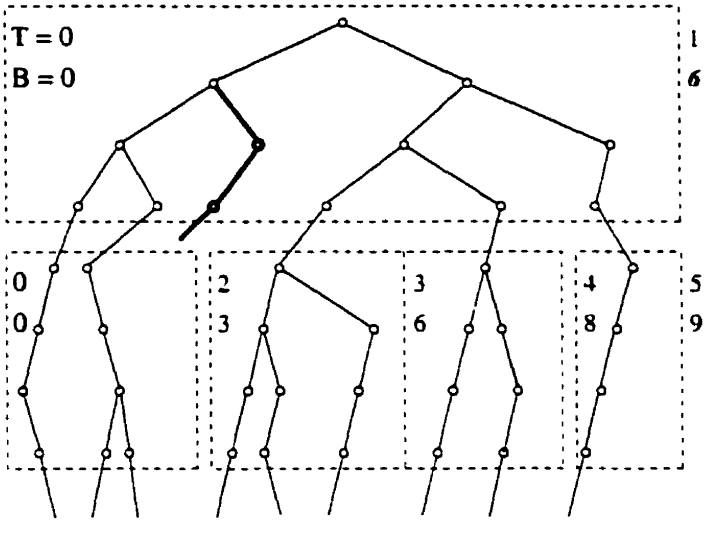


Figure 4.6: Partially modified trie for Example 2

In Figure 4.6, Transaction 2 has inserted `0100` into the root page and modified the B-count for the root page level. Transaction 1 then successfully reads `1010` and calculates $srch = 5$. To find the remainder of its key, it needs to read the page to the left of the page with minimum T such that $T \geq 5$ on page level 2.

Transaction 1 searches the page with $T = 4$, which is incorrect. To recover, transaction 1 must move to the left. This failure occurs because Transaction 1 uses a $srch$ count that results from Transaction 2's execution, but not T-counts that result from Transaction 2's execution.

Example 3

Even if a transaction successfully chooses the next page, it may not be able to navigate the page correctly. Consider a new trie, shown in Figure 4.7 and the transaction schedule for Example 2 in Figure 4.5.

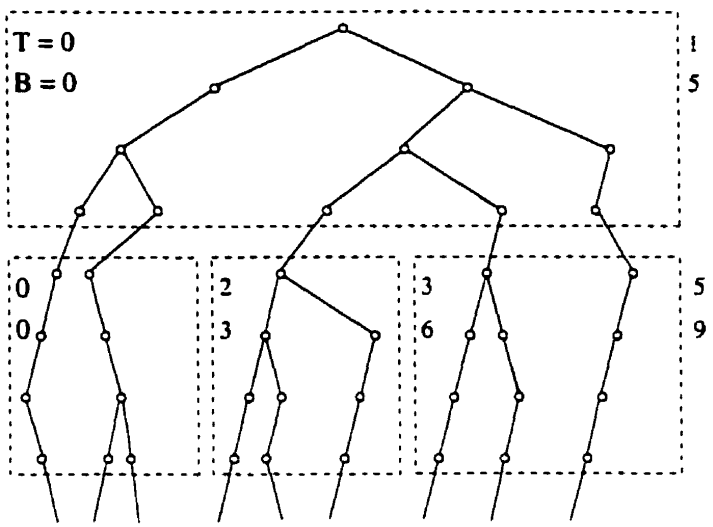


Figure 4.7: Initial trie for Example 3

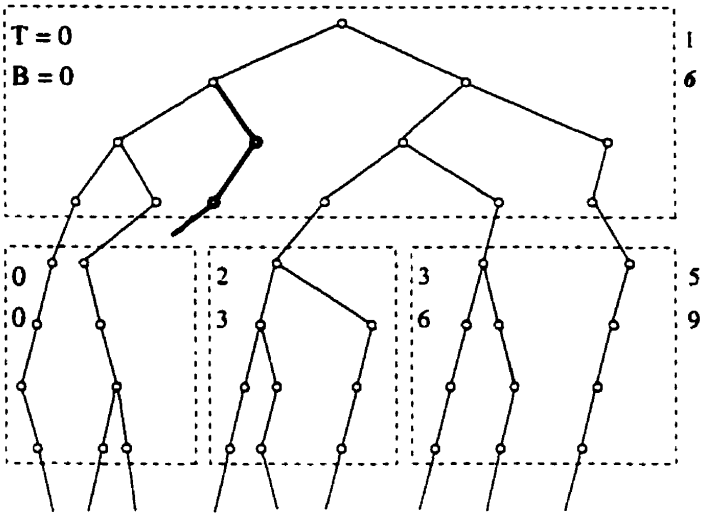


Figure 4.8: Partially modified trie for Example 3

In Figure 4.8, Transaction 2, just as in Example 2, has inserted 0100 into the root page and modified the B-count for the root page level. Again, Transaction 1 reads 1010 in the root page and determines that it will next read the page to the left of the page with mimimum T such that $T \geq 5$ on page level 2. Transaction 1 correctly chooses the page on page level 2 with $T = 3$. However, to navigate the page, it calculates the value for $next = B_{root} + srch - T - 1 = 1$, which is incorrect. To search the correct subtrie in the page, $next$ must equal 0.

One proposed solution for these problems is the addition of redundancy so that an operation is able to determine whether it is operating on the correct page and the correct subtrie. Just as for B-trees, the operation would be able to correct an error caused by a concurrent modification on the trie.

This redundancy could be in the form of a prefix range created for each trie page as in Figure 4.9. With this scheme, we record the minimum and maximum prefix leading into each page. If the operation's key is not in the prefix range for the selected page, the operation can recover to the left or right until the correct page is selected. Note that the entire prefixes must be stored, not just the prefix leading into the page from the previous page level.

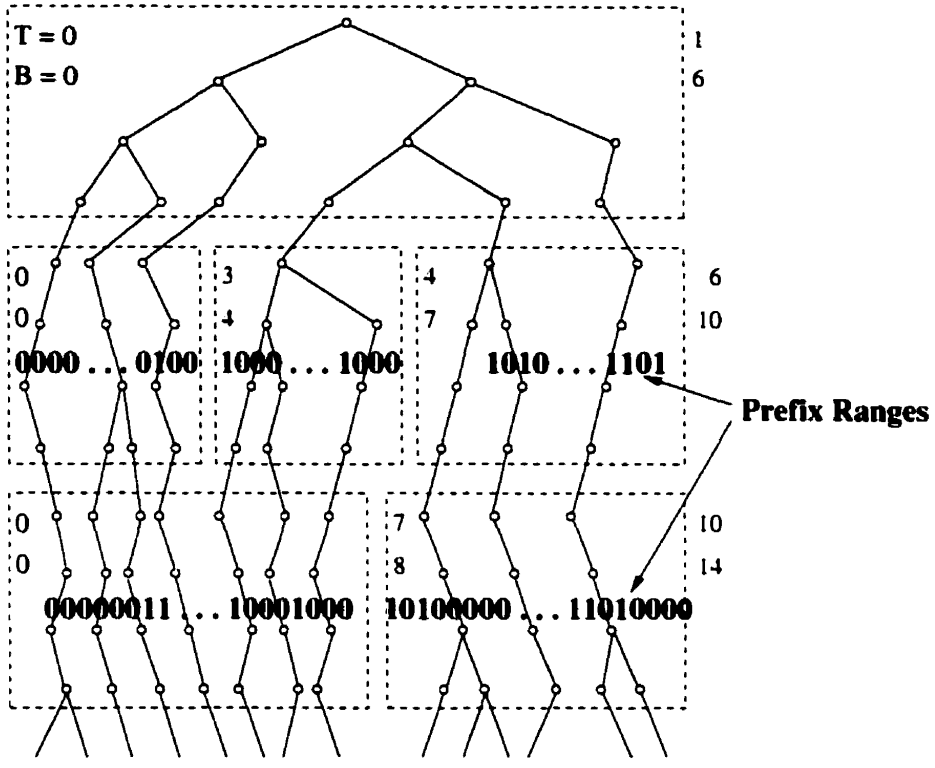


Figure 4.9: Trie with prefix ranges

However, an operation may still navigate a trie page incorrectly even though it has selected the correct page to navigate, as in Example 3. Plus, splitting a full trie page becomes much more difficult because new prefix ranges for the old and new pages need to be determined. To solve these problems, we could store a prefix for each subtrie in a page. This would ensure that the correct trie page can be navigated correctly and that page splits remain easy and efficient to perform. However, this adds an unacceptable amount of redundancy to the trie because the amount of storage required to simply record all the prefixes will easily surpass the amount of storage required to store all the trie pages.

So, to prevent these problems from occurring, we first decide that concurrent trie operations must use consistent pages and counts when progressing from one page level to the next. More specifically, we note that:

1. If an operation does not encounter any modification made by an insertion I on page level ℓ, it will operate correctly on page level $\ell + 1$ if it does not encounter any modifications made by

I on page level $\ell + 1$.

2. If an operation encounters any modifications made by an insertion I on page level ℓ, it will operate correctly on page level $\ell + 1$ if it encounters a point in the trie already encountered by I on page level $\ell + 1$.

So, we create a mechanism that controls the progression of operations from one page level to the next relative to a modifying operation. If an operation is encountering pages (or counts) before a modification to them occurs on one page level, then we ensure that it encounters pages (or counts) before they are modified on the next page level. Also, if an operation encounters any changes on one page level, we ensure that it does not encounter any pages (or counts) on the next page level that are not yet modified.

They key components of this mechanism are the use of a sequence in which pages (and counts) are encountered and the use of lock-coupling. Basically, operations first lock the leftmost page on a page level and lock-couple to the right as they examine the T-counts and find their correct page. We now describe how concurrent trie search and insertion operations provide this mechanism.

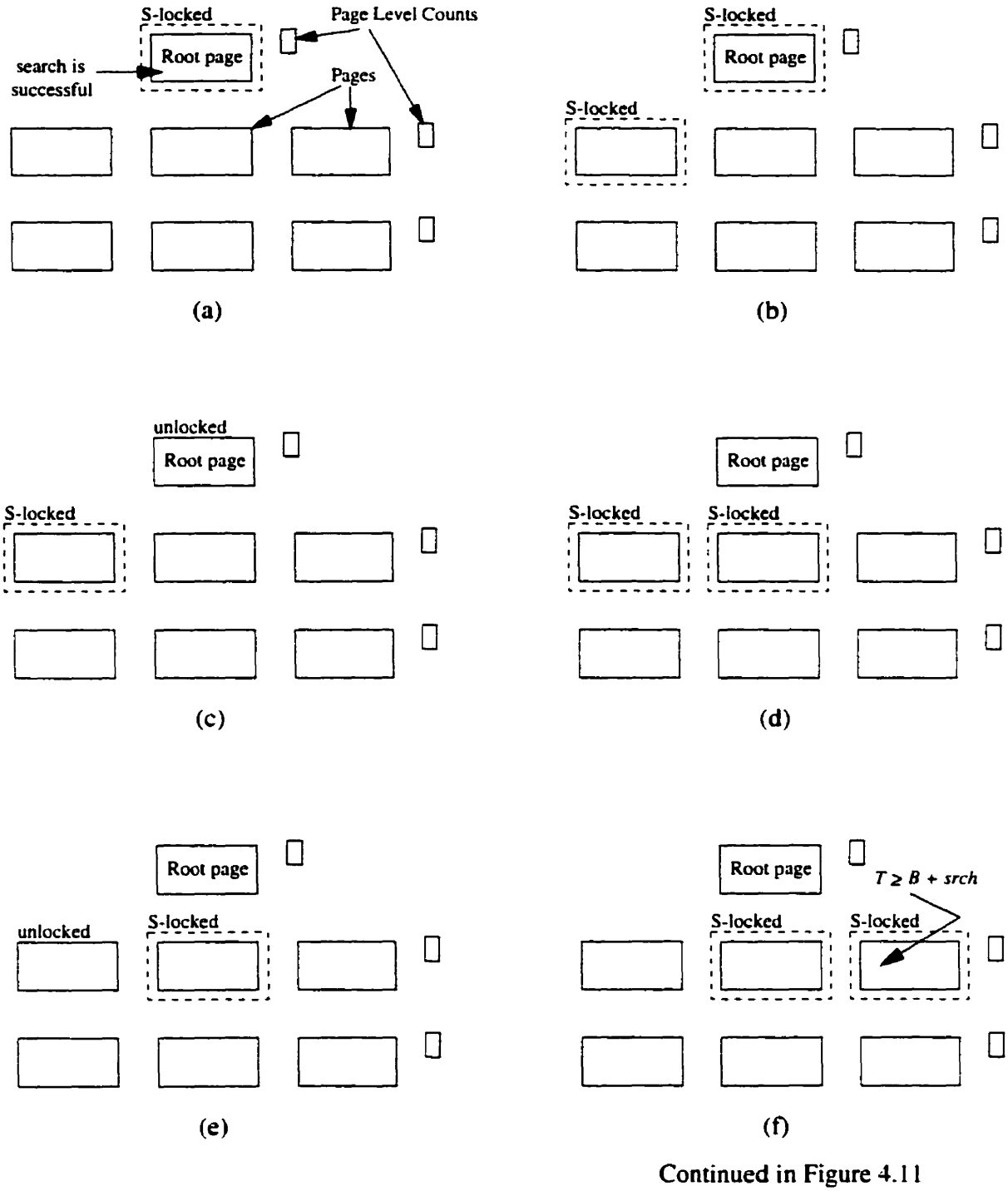
4.1.1 The Trie Search Algorithm

To simplify our description of the locking performed by this and the concurrent trie insertion algorithm, we will assume that the counts for the trie page are locked when the trie page is locked, even though the counts are stored separately from the pages. The only exceptions to this are the counts to the right of all pages on each page level. Hence, the concurrent trie operations lock only trie pages and the page level counts.

Prior to reading any page or count, searches S-lock the page (or page level count). Figures 4.10 and 4.11 on pages 76 and 77 show the step-by-step progression of a typical search operation.

Initially, the search operation S-locks and searches the root page (Figure 4.10a). If the search fails, the root page is unlocked and the operation terminates. If the search is successful, the search operation S-locks the leftmost page on the next page level *before* unlocking the searched page (Figures 4.10b and 4.10c). This lock-coupling is essential because it prevents an insertion operation from modifying a page on a previous level, advancing ahead of the search operation, and modifying a page or count that the search operation does not expect to be changed.

Once the searched page is unlocked, the lock-coupling continues from left to right until a T-count $\geq B' + srch$, where B' is the B-count for the page navigated on the previous page level, is reached (Figures 4.10d– 4.10f). At this point, both the next page to search and page (or level counts) to the right are S-locked. The search operation unlocks the page (or level counts) to the right and searches the page that is still S-locked (Figure 4.11g). If the search fails or the searched page is a leaf page, the searched page is unlocked and the operation terminates. Otherwise, the operation S-locks the leftmost page on the next page level and repeats until the search is unsuccessful or the searched page is a leaf page (Figures 4.11h– 4.11l).



Continued in Figure 4.11

Figure 4.10: Concurrent Search Operation

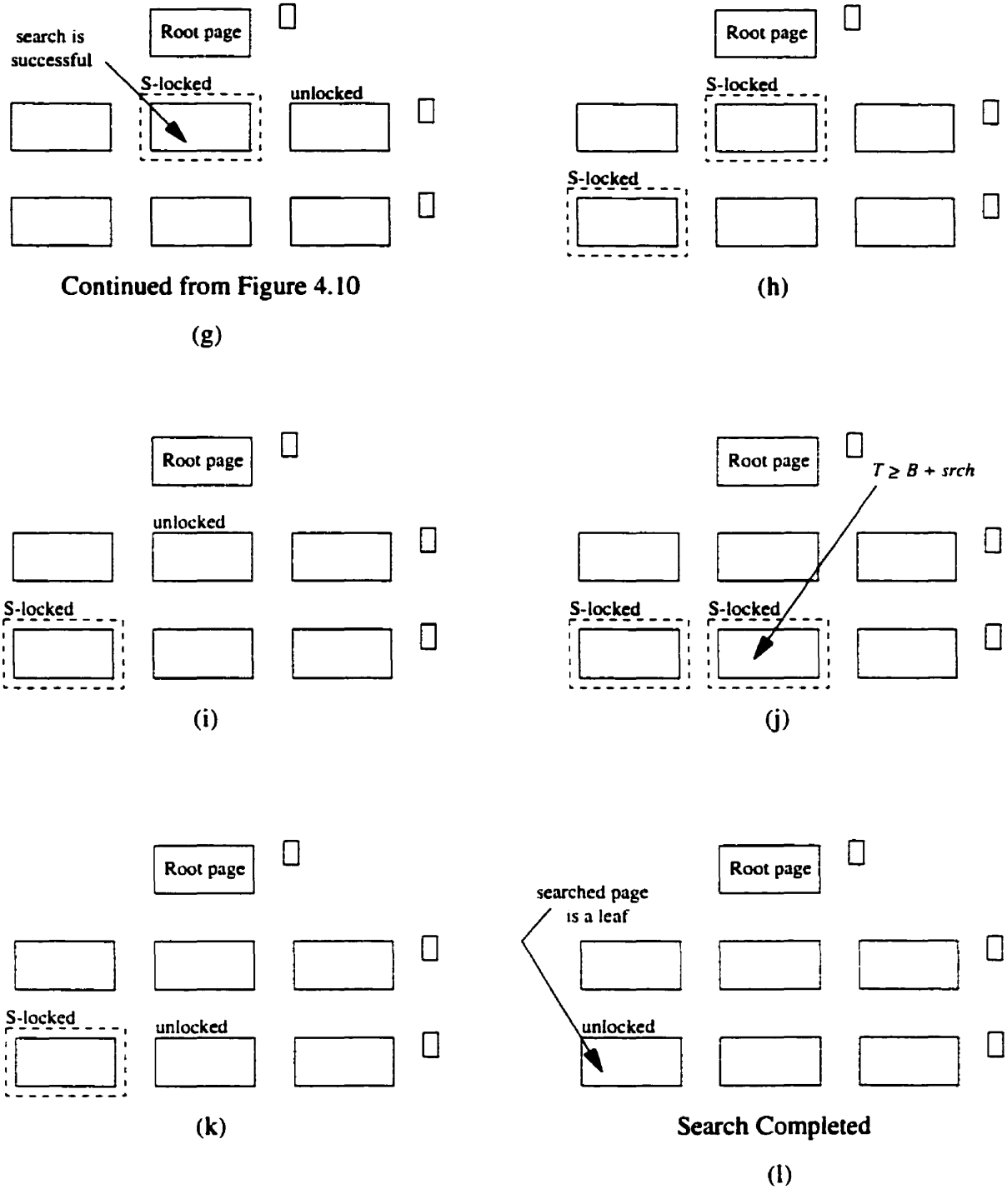


Figure 4.11: Concurrent Search Operation (continued)

The algorithm for concurrent trie searching is presented in Figure 4.12. Note that if *right-page-id* = NIL; that is, *current-page-id* is the rightmost page, the counts for the current page level are locked.

```
TRIE-SEARCH(key)
 1  current-page-id ← root-id
 2  S-LOCK(current-page-id)
 3  current-page ← READ-PAGE(current-page-id)
 4  SEARCH-TRIE-PAGE(key, current-page)
 5  while search successful and current-page is not a leaf page do
 6      B_last-page-searched ← B_current-page-id
 7      leftmost-page-id ← leftmost page on next page level
 8      S-LOCK(leftmost-page-id)
 9      UNLOCK(current-page-id)
10      current-page-id ← leftmost-page-id
11      right-page-id ← page to right of current-page-id
12      S-LOCK(right-page-id)
13      while T_right-page-id < B_last-page-searched + srch do
14          UNLOCK(current-page-id)
15          current-page-id ← right-page-id
16          right-page-id ← page to right of current-page-id
17          S-LOCK(right-page-id)
18      UNLOCK(right-page-id)
19      current-page ← READ-PAGE(current-page-id)
20      SEARCH-TRIE-PAGE(key, current-page)
21  UNLOCK(current-page-id)
```

Figure 4.12: Concurrent Trie Search Algorithm

4.1.2 The Trie Insertion Algorithm

There are three phases of a trie insertion:

- **Phase 1:** Search for first page to modify
- **Phase 2:** Modify first page and B-counts to the right
- **Phase 3:** Modify remaining pages and T-counts and B-counts to the right

Phase 1: Search for first page to modify

Inserts behave exactly like searches until they reach a page where their search fails. This is the page that will have a bit pair changed and, probably, bit pairs inserted.

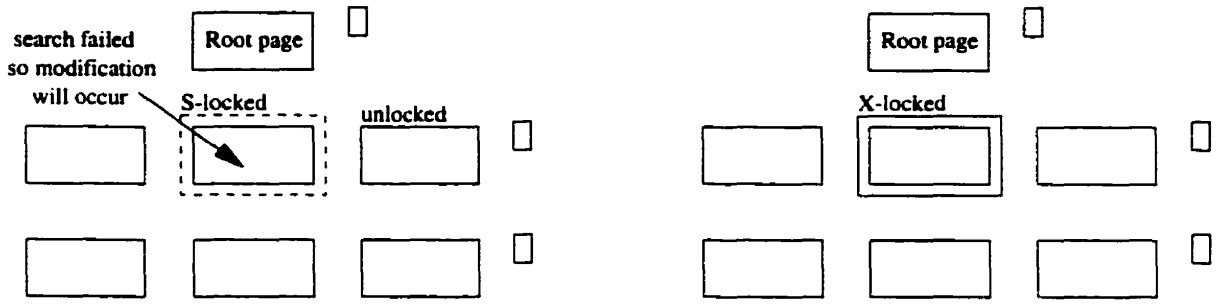
Phase 2: Modify first page and B-counts to the right

Figure 4.13 on page 80 shows the step-by-step actions of Phase 2. Assume that Phase 1 progressed as for the search example in Figures 4.10a–4.10f on page 76.

Before inserting, the operation upgrades the S-lock on the page to an X-lock (Figure 4.13b). If other operations currently hold an S-lock on the page, the operation releases its S-lock, puts its X-lock request on the *head* of the wait queue, and sets the lock mode of the page to "wait" so that subsequent requests for locks on the page go on the tail of the wait queue. Upgrading the S-lock to an X-lock in this manner avoids the possibility of deadlock due to multiple operations simultaneously attempting to upgrade their S-lock. No operations will starve since the number of operations holding an S-lock on the page is finite and any lock request by an operation not already holding a lock on the page goes on the tail of the wait queue. The X-lock request goes on the head of the wait queue so that any operations in Phase 3 do not modify the T-count for the page. Any change in T would disrupt navigation of the page since $next = B' + srch - T - 1$.

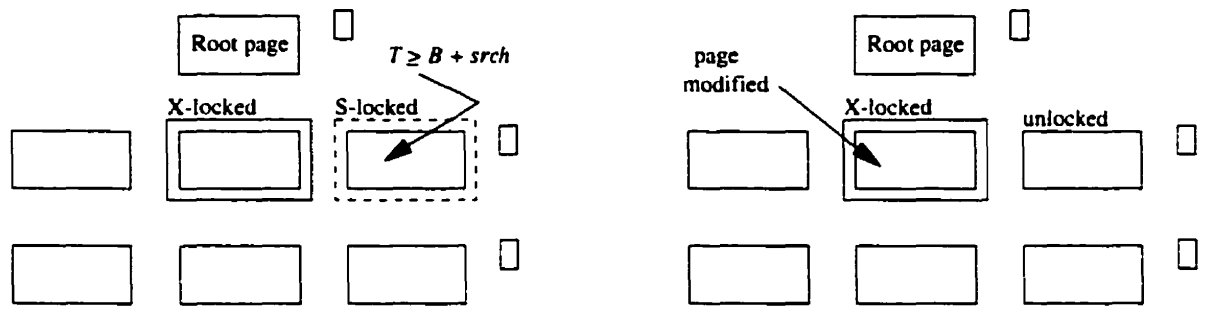
However, while waiting for the X-lock, it is possible that another insert operation has inserted the same bit sequence as the waiting operation intended to or has split the page. This may occur if one of the other insert operations holding an S-lock on the page moved into Phase 2 and upgraded its S-lock to an X-lock. This other insert operation would either go on the head of the wait queue, ahead of the operation already waiting, or acquire an X-lock on the page since it was the last operation holding an S-lock.

Therefore, the page (or level counts) to the right must be checked to ensure that the insertion will be in the X-locked page; that is, the page did not split (Figure 4.13c). If $T_{right} < B' + srch$, then the operation must X-lock the page to the right and unlock the current X-locked page. Lock-coupling to the right continues until the operation reaches the page with $T_{right} \geq B' + srch$. We call this action a *link chase* since it is similar to the recovery performed by the B^{link} algorithms. If the correct page does not need to be modified anymore due to the other operation, the operation S-locks the leftmost

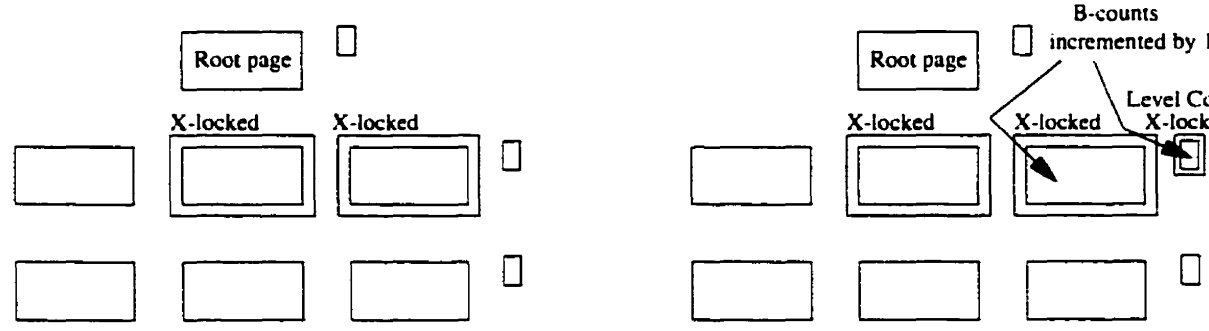


Continued from Phase 1 in Figure 4.10

(a) (b)



(c) (d)



(e) (f)

Continued in Phase 3 in Figure 4.14

Figure 4.13: Concurrent Insert Operation (Phase 2: Modify First Page and B-counts to the Right)

page on the next page level, unlocks the current page, and proceeds as in Phase 1.

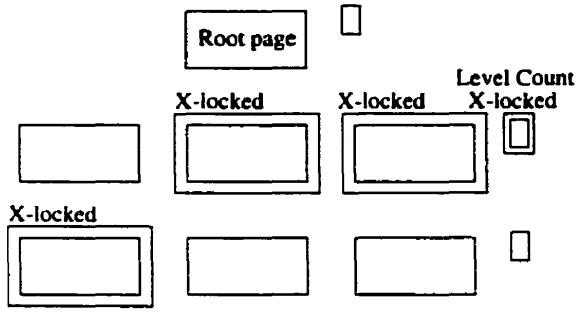
After the operation modifies the page, all the B-counts to the right need to be incremented by 1. The operation X-locks all pages to the right of the modified page, as well as the counts for the page level from left to right (Figures 4.13e and 4.13f) and increments the B-counts by 1 (Figure 4.13f). Remember, even though the counts are not actually on the pages, they are locked when the pages to which they belong are locked. The operation then proceeds to Phase 3 of the insertion.

Phase 3: Modify remaining pages and T-counts and B-counts to the right

Figures 4.14– 4.16 on pages 82– 84 shows the step-by-step actions of Phase 3. These figures are a continuation of Figure 4.13 on page 80.

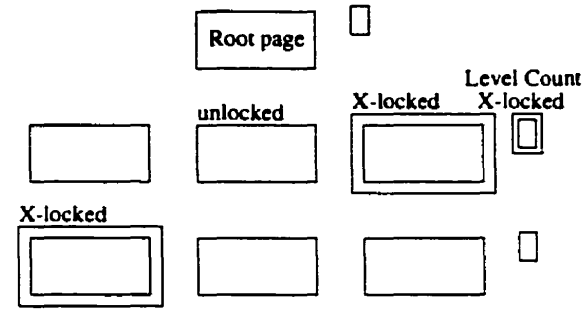
At the start of Phase 3, the insert operation has the modified page, all pages to the right of the modified page, and the counts for the page level X-locked. The insert operation now uses only X-locks until it finishes its insertion.

First, the operation X-locks the leftmost page on the page level immediately below the page level where the modification in Phase 2 took place (Figure 4.14a). The operation then unlocks the X-locks remaining from Phase 2 on the previous page level from left to right (Figures 4.14b– 4.14d). The insert operation then lock-couples from left to right with X-locks until it reaches a T-count $\geq B' + srch$ (Figures 4.14e– 4.15g). The operation unlocks the page (or level count) with $T \geq B' + srch$ and modifies the page that is still X-locked (Figure 4.15h). Then, as in Phase 2, the operation X-locks all the pages and the level counts to the right (Figures 4.15i and 4.15j). Both the T and B-counts must be incremented by 1 since the operation added an incoming trie edge to the page level. Phase 3 then repeats until the modified page is a leaf page, at which point it unlocks the leaf page level from left to right after incrementing the counts (Figures 4.15k– 4.16m) and terminates.

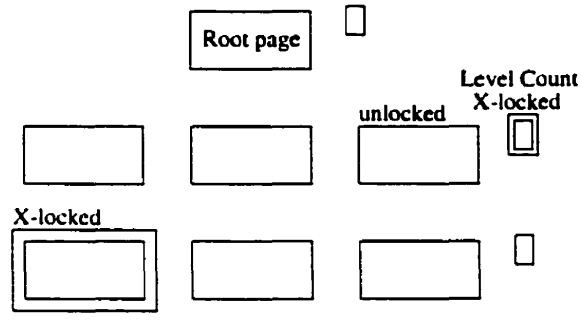


Continued from Phase 2 in Figure 4.13

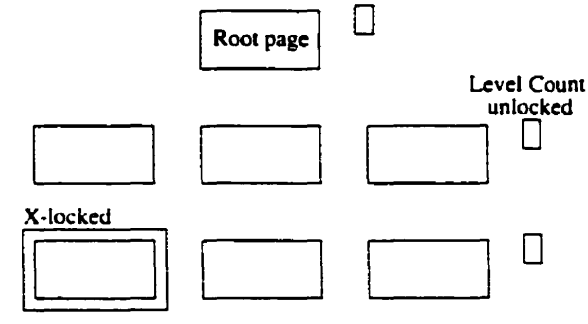
(a)



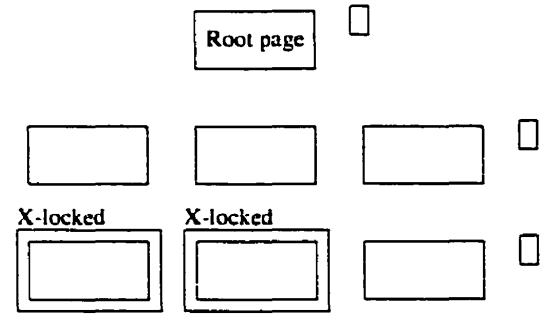
(b)



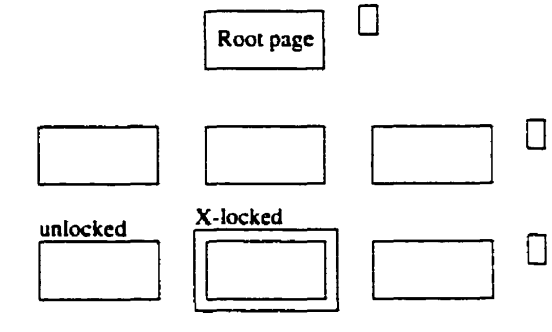
(c)



(d)



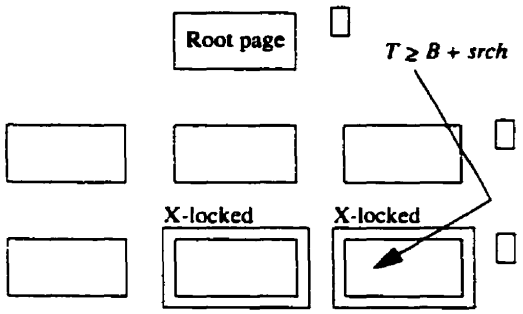
(e)



(f)

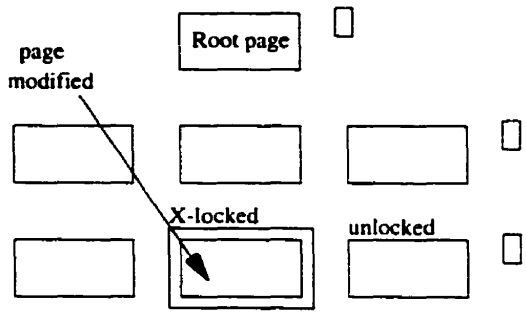
Continued in Figure 4.15

Figure 4.14: Concurrent Insert Operation (Phase 3: Modify Remaining Pages and All Counts to the Right)

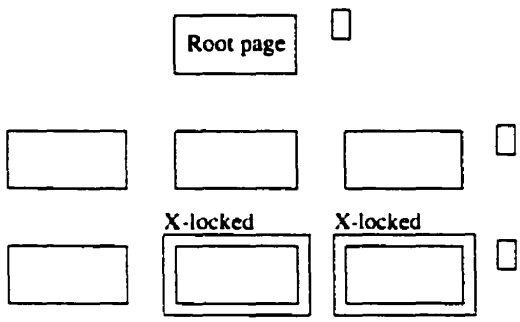


Continued from Figure 4.14

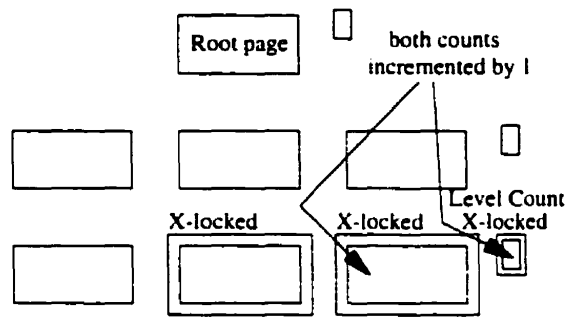
(g)



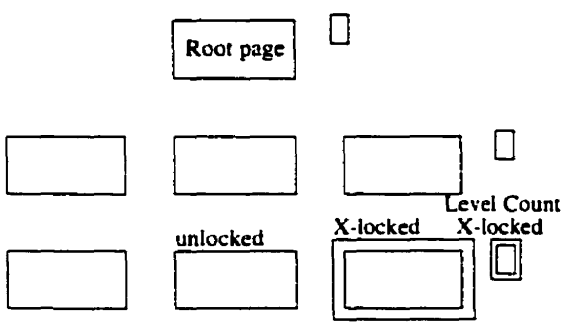
(h)



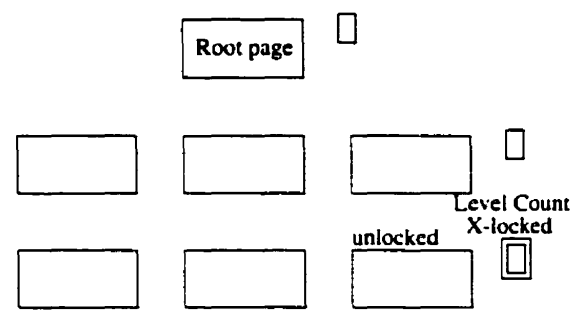
(i)



(j)



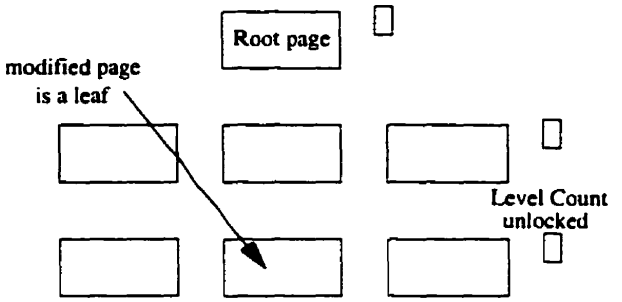
(k)



(l)

Continued in Figure 4.16

Figure 4.15: Concurrent Insert Operation (Phase 3: Modify Remaining Pages and All Counts to the Right) (continued)



Continued from Figure 4.15

Insert Completed

(m)

Figure 4.16: Concurrent Insert Operation (Phase 3: Modify Remaining Pages and All Counts to the Right) (continued)

The algorithm for concurrent trie insertions is presented in Figures 4.17– 4.19. The algorithm has been divided into the three insert phases. Note that if *right-page-id* = NIL; that is, *current-page-id* is the rightmost page, the counts for the current page level are locked.

```
TRIE-INSERT(key)
 1  current-page-id ← root-id
 2  S-LOCK(current-page-id)
 3  current-page ← READ-PAGE(current-page-id)
 4  SEARCH-TRIE-PAGE(key, current-page)
 5  while search successful and current-page is not a leaf page do
 6       B_last-page-searched ← B_current-page-id
 7       leftmost-page-id ← leftmost page on next page level
 8       S-LOCK(leftmost-page-id)
 9       UNLOCK(current-page-id)
10       current-page-id ← leftmost-page-id
11       right-page-id ← page to right of current-page-id
12       S-LOCK(right-page-id)
13       while T_right-page-id < B_last-page-searched + srch do
14            UNLOCK(current-page-id)
15            current-page-id ← right-page-id
16            right-page-id ← page to right of current-page-id
17            S-LOCK(right-page-id)
18       UNLOCK(right-page-id)
19       current-page ← READ-PAGE(current-page-id)
20       SEARCH-TRIE-PAGE(key, current-page)
21  if search successful then
22       UNLOCK(current-page-id)
23       return "key already in trie"
24  else
25       continue on to Phase 2 in Figure 4.18
```

Figure 4.17: Concurrent Trie Insertion Algorithm (Phase 1: Search for First Page to Modify)

```
    /* continued from Phase 1 in Figure 4.17 /*
26  UPGRADE-LOCK(current-page-id)
27  S-LOCK(right-page-id)
28  while T_right-page-id < B_last-page-searched + srch do
29      UPGRADE-LOCK(right-page-id)
30      UNLOCK(current-page-id)
31      current-page-id ← right-page-id
32      right-page-id ← page to right of current-page-id
33      S-LOCK(right-page-id)
34  size ← T_right-page-id − B_current-page-id
35  next ← B_last-page-searched + srch − T_right-page-id − 1
36  UNLOCK(right-page-id)
37  current-page ← READ-PAGE(current-page-id)
38  MAKE-INITIAL-TRIE-INSERT(key, current-page)
39  if current-page was not changed by MAKE-INITIAL-TRIE-INSERT then
40      search is successful      /* for while-loop at Step 5 of Phase 1 */
41      goto Step 5 of Phase 1
42  B_last-page-modified ← B_current-page-id
43  temp-page-id ← current-page-id
44  current-page-id ← right-page-id
45  while current-page-id ≠ NIL do
46      X-LOCK(current-page-id)
47      increment B-count for current-page-id by 1
48      current-page-id ← page to right of current-page-id
49  X-LOCK(level counts)
50  increment B-count for current page level by 1
51  continue on to Phase 3 in Figure 4.19
```

Figure 4.18: Concurrent Trie Insertion Algorithm (Phase 2: Modify First Page and B-counts to the Right)

```
    /* continued from Phase 2 in Figure 4.18 */
52  while current-page-id is not a leaf page do
53      leftmost-page-id ← leftmost page on next page level
54      X-LOCK(leftmost-page-id)
55      while current-page-id ≠ NIL do
56          UNLOCK(current-page-id)
57          current-page-id ← page to right of current-page-id
58      UNLOCK(level counts)
59      current-page-id ← leftmost-page-id
60      right-page-id ← page to right of current-page-id
61      X-LOCK(right-page-id)
62      while T_right-page-id < B_last-page-modified + srch do
63          UNLOCK(current-page-id)
64          current-page-id ← right-page-id
65          right-page-id ← page to right of current-page-id
66          X-LOCK(right-page-id)
67      UNLOCK(right-page-id)
68      current-page ← READ-PAGE(current-page-id)
69      INSERT-SUBTRIE(key, current-page)
70      B_last-page-modified ← B_current-page-id
71      temp-page-id ← current-page-id
72      current-page-id ← right-page-id
73      while current-page-id ≠ NIL do
74          X-LOCK(current-page-id)
75          increment T-count and B-count for current-page-id by 1
76          current-page-id ← page to right of current-page-id
77      X-LOCK(level counts)
78      increment T-count and B-count for current page level by 1
79  current-page-id ← temp-page-id
80  while current-page-id ≠ NIL do
81      UNLOCK(current-page-id)
82      current-page-id ← page to right of current-page-id
83  UNLOCK(level counts)
```

Figure 4.19: Concurrent Trie Insertion Algorithm (Phase 3: Modify Remaining Pages and All Counts to the Right)

4.1.3 Proof of Correctness

There are two properties of these operations that need to be demonstrated:

1. They do not form a deadlock. (Theorem 1)
2. Their correctness is not impaired by other concurrent operations. (Theorem 2)

Freedom from Deadlock

Theorem 1 *The algorithms for concurrent trie search and insertion never form a deadlock.*

Proof We show that any wait-for graph [Hol71, Hol72] generated for the trie where nodes in the wait-for graph are trie pages never contains a cycle because locks are made following a well-ordering of the pages. Since page level counts can also be locked by operations, consider them to simply be an additional page at the end of a each page level. This ordering is as follows:

- If two pages of the trie, x and y, are not on the same page level, then $x < y$ if x is closer to the root page level than y.
- If two pages of the trie, x and y, are on the same page level, then $x < y$ if x is to the left of y.

When pages are created, the ordering remains intact because page creation is done by splitting an existing page. For example, three pages exist such that $x < y < z$. When a new page, y'' is created by splitting y into y' and y'', the ordering remains intact and is $x < y' < y'' < z$.

Locks are made from left to right and from root to leaf. After placing a lock on a page, no page to the left on the same page level and no page on a page level closer to the root is ever locked. Therefore, the concurrent trie operations lock all pages in a well-ordered manner. In addition, upgrading from an S-lock to an X-lock is made by first releasing the S-lock; hence, no deadlocks form due to multiple operations attempting to upgrade their S-lock for the same page. ∎

Correctness of Concurrent Operations

The correctness of concurrent operations is a concern only when an insert operation conflicts with another operation. For this, we assume that insert operation I is in Phase 2 or 3 of the insert

algorithm, since Phase 1 is merely a search. To prove that any modification to the trie does not impair the correctness of another concurrent trie operation, we show that our previous observations are always true:

1. If an operation O does not encounter any modification made by an insertion I on page level ℓ, it will operate correctly on page level $\ell + 1$ if it does not encounter any modifications made by I on page level $\ell + 1$.

2. If an operation O encounters any modifications made by an insertion I on page level ℓ, it will operate correctly on page level $\ell + 1$ if it encounters a point in the trie already encountered by I on page level $\ell + 1$.

We break down these observations into three lemmas as shown in Figure 4.20. We assume for Lemmas 1–3 that I is using only X-locks. The situation when I upgrades from an S-lock to an X-lock is discussed in Lemma 4. With these lemmas, we prove Theorem 2, which states that the correctness of trie operations is not impaired by other concurrent operations.

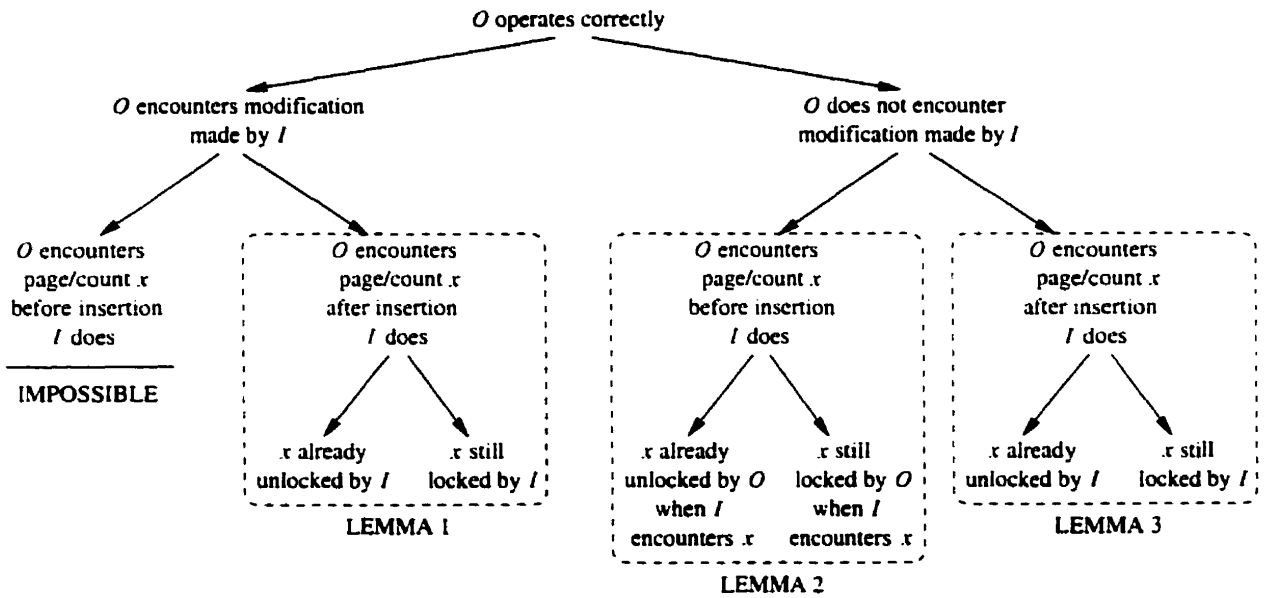


Figure 4.20: Breakdown of observations into lemmas

Lemma 1 *Any operation O that navigates the correct page x on page level ℓ_x where x has been*

modified by insertion I will successfully locate and navigate the correct page y on page level $\ell_y = \ell_x + 1$.

Proof There are two cases for the scenario where O navigates x after I modified x:

Case 1: Operation O encounters x after x is unlocked by insertion I

Once O locks x, it must also lock the page (or level counts) to the right of x to determine $size$ for the purpose of navigating x. Since I unlocks page (or level counts) after unlocking x, the counts to the right of x will be consistent with x and O will navigate x correctly. Because I lock-couples with X-locks when advancing from page level ℓ_x to page level ℓ_y and advancing along ℓ_y, any page y on page level ℓ_y that O encounters has already been encountered by I. So, any modification that O expects I to have made on page level ℓ_y will exist and O will operate correctly on page level ℓ_y.

Case 2: Operation O encounters x while x is locked by insertion I

Since x has been modified by I, I is holding an X-lock on x; otherwise, I would be holding an S-lock and x would not yet be modified. So, O must wait on the queue until I releases its X-lock. Now, when O navigates x, x has been modified and unlocked by I, which is equivalent to Case 1 above.

So, any operation O that navigates the correct page x on page level ℓ_x where x has been modified by insertion I will successfully operate on page level $\ell_y = \ell_x + 1$. ■

Lemma 2 *Any operation O that navigates the correct page x on page level ℓ_x prior to insertion I modifying x will successfully locate and navigate the correct page y on page level $\ell_y = \ell_x + 1$.*

Proof There are two cases for the scenario where O navigates x prior to I modifying x.

Case 1: Insertion I encounters x after x is unlocked by operation O

The counts to the right of x will not yet be modified by I because operations lock from left to right, so O will set $size$ correctly and navigate x correctly. In general, any page/count y encountered by operation O after navigating x will be either to the right of page/count x on page level ℓ_x, or on a page level $\ell_y > \ell_x$. Since all concurrent trie operations lock-couple

from left to right and from root to leaf and insertion I is using X-locks after modifying x, O locks y before I does. Therefore, y is unmodified by insertion I when O encounters y as O expects it to be.

Case 2: Insertion I encounters x while x is locked by operation O

Since operation O currently holds a lock on page/count x and I is attempting to X-lock x, I must wait for O to unlock x before it can lock and modify x. As stated for Case 1, lock-coupling by operation O ensures it never encounters a page/count already modified by insertion I once O encounters a page/count that is not yet modified by I.

So, any operation O that navigates the correct page x on page level ℓ_x prior to insertion I modifying x will successfully operate on page level $\ell_y = \ell_x + 1$. ■

Lemma 3 *Any operation O that navigates the correct page x on page level ℓ_x where x has been encountered, but not modified by insertion I will successfully locate and navigate the correct page y on page level $\ell_y = \ell_x + 1$.*

Proof There are two cases for the scenario where O navigates x after I has encountered, but not modified x:

Case 1: Operation O encounters x after x is unlocked by I

Operation O will operate correctly on page level ℓ_y no matter if it advances to page level ℓ_y before or after I does; therefore, no race condition can occur.

If the page to the right of x is not locked by I and I has not yet locked any page on level ℓ_y, O may be able to search x and S-lock the leftmost page on level ℓ_y before I finishes modifying pages/counts to the right of x on level ℓ_x (Figure 4.21a). In this situation, O will operate correctly on page level ℓ_y because it has not been affected by I on page level ℓ_x and won't be on page level ℓ_y. If O modifies x or I modifies the page to the right of x, then O will not advance to page level ℓ_y before I. Also, if I modifies the page to the right of x, then O remains unaffected because the counts to the right of x will be unchanged by I.

If I advances to page level ℓ_y before O does, O will still operate correctly on page level ℓ_y. Even though O is unaffected by I on page level ℓ_x, any change by I on level ℓ_y can be handled

by O. If O encounters no change by I on level ℓ_y, it will succeed because it encountered no change on level ℓ_x. If O encounters a change by I on level ℓ_y, it will still succeed on level ℓ_y. Since x was not modified by I, the $srch$ count for O was not affected by I when navigating x. Since trie edges do not cross, the $srch$ count will be correct for choosing the next page on level ℓ_y, which is the page to the left of the page with $T \geq B_x + srch$. The only affect a change by I has as far as O is concerned is that the T-count to the right of y is incremented by 1 (Figure 4.21b), which is fine since T is still $\geq B_x + srch$. Even if y is split by I, the T-count to the right of y will still lead O to the correct page (Figure 4.21c).

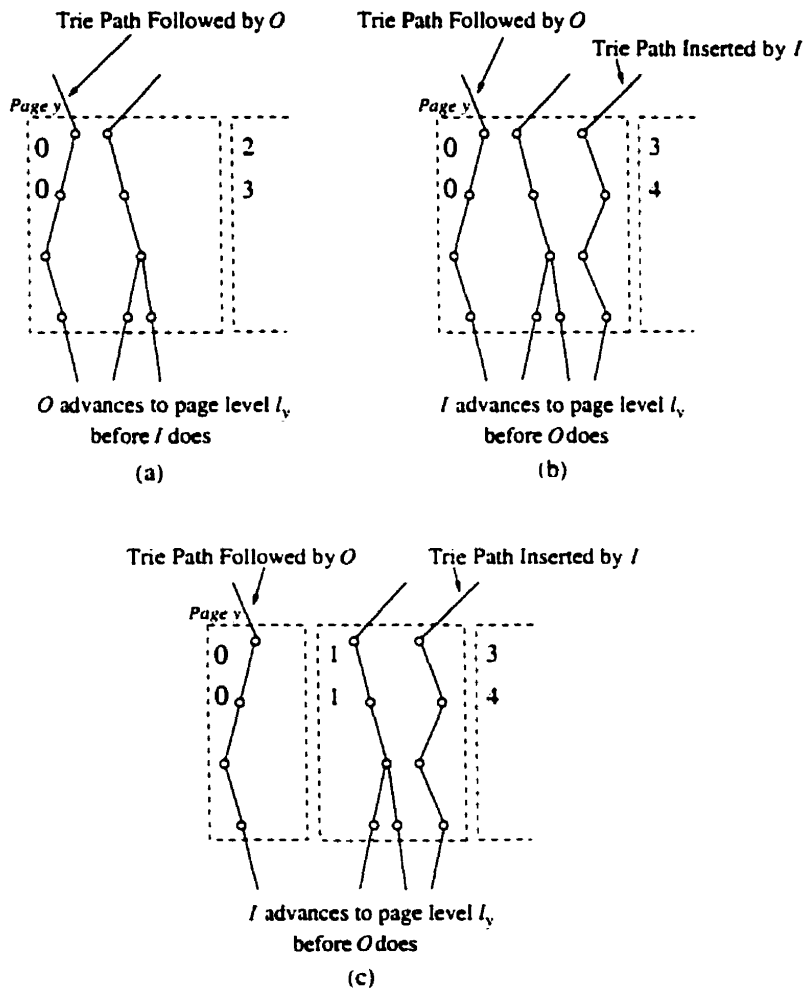


Figure 4.21: Effect of insertion I on operation O's navigation of level ℓ_y

Case 2: Operation O encounters x while x is locked by I

The same occurs as in Case 1 above, except that O does not navigate x until I has released its X-lock.

So, any operation O that navigates the correct page x on page level ℓ_x that has been encountered, but not modified by insertion I will successfully operate on page level $\ell_y = \ell_x + 1$ ∎

The above lemmas discuss the interaction between any operation O and an insertion I that is using X-locks, meaning that both O and I cannot have any page (or level count) locked at the same time. Now, we discuss the case when I is upgrading from an S-lock to an X-lock.

Lemma 4 *Any insert I that upgrades its lock on a page x from an S-lock to an X-lock will perform correctly on page level ℓ_x and not impair the correctness of any other operation O.*

Proof We only discuss the situations where O and I lock the same page since the situations where they lock different pages are equivalent to those discussed in Lemmas 1–3. Upgrading from an S-lock to an X-lock on a page not encountered by another operation does not create any problems. So, when I upgrades its lock on page x from an S-lock to an X-lock, we consider two scenarios that may exist:

Case 1: Page x is S-locked by operation O and there is no wait queue

Both insertion I and operation O currently hold an S-lock on page x. When I determines that it must modify page x, it releases its S-lock and puts its X-lock request on the head of the queue. Now, O may either release its S-lock and allow the I to acquire its X-lock and modify x, or decide to upgrade its S-lock to an X-lock. If O releases its S-lock, there is no conflict because neither O nor I have modified x. Both operations perform correctly. However, if O upgrades its lock on x to an X-lock and modifies x and the B-counts to the right of x, I may encounter an inconsistent page. Operation O may either split x or insert the same bit sequence that I intended to. Insertion I will detect this though since it checks the T-count to the right after acquiring its X-lock to ensure x was not split and then lock-couples to the right if necessary. Also, I will revert back to Phase 1 (searching for the first page to modify) if it discovers that O inserted the bit sequence that I intended to. So, in this case, O will perform correctly and I will recover if need be and perform correctly.

Case 2: Page x has operation O waiting for x in the wait queue

The insertion I currently holds an S-lock on page x. Therefore, operation O waiting in the queue for x must be requesting an X-lock for x. Since I locked x prior to operation O (which is an insertion), I did not encounter any pages/counts on page level $\ell_x - 1$ that were modified by O, so I does not expect any changes to be made to x by O. Therefore, to operate correctly, I must operate on x before any modification to x by O. Since I goes on the *head* of the wait queue, I does indeed operate on x prior to any modification by O. Any modification by I on x does not impair the correctness of O since O can only be affected by a page split of x and O checks the T-count to the right of x to confirm that x is indeed the correct page prior to any modification. If need be, O will lock-couple with X-locks to the right and encounter the correct page. So, in this case, insertions I and O will both perform correctly.

We have shown that the correctness of an operation O is not impaired by an insertion I that upgrades its S-lock to an X-lock and that I will perform correctly after acquiring its X-lock. ■

Theorem 2 *Correctness of the trie operations is not impaired by the concurrent execution of other trie operations.*

Proof Initially, the root page is always navigated. We know that any operation O will navigate the root page correctly because insert operations acquire an X-lock prior to modifying a page and release their X-lock only after their modification is complete; hence, the root page is always consistent.

From the above lemmas, any operation O will successfully locate and navigate their next page on the next page level whether or not an insert operation I modified the root page. By induction, O will successfully operate on each subsequent page level until it terminates. ■

4.2 Experimental Procedure

Our experiments for measuring trie concurrency mirror those performed for B-trees. Most simulation parameters used for the B-tree experiments are not modified for the trie experiments so that we may compare results with those obtained for B-trees. Parameters relating to the structure of the trie; namely, key selection and fanout, are adjusted to create an acceptable amount of similarity

between the trie and the B-tree. Also, parameters affected by the differences between the trie and B-tree structures; namely, the buffer pool size, are adjusted to equalize the portion of the structures in memory. In addition, we add a new parameter to account for the CPU usage required to access T-counts. We now explain the modification and addition of these parameters in detail.

4.2.1 Modification of B-Tree Parameters For Use in Trie Experiments

Modification of Key Selection

Because the structure of tries is based on the binary representation of the data, we use diffeerent keys to construct them. We again use Java to perform our experiments. Since Java uses the first bit in its binary representation of integers as a sign bit, we use both positive and negative keys to construct a trie. This way, the first bit is not always a `0`. Also, when constructing a trie, we use a much larger key space than that used to construct a B-tree. If we were to build a trie with all the odd keys in the key space, as we do to build a B-tree, we would create a trie that is mostly a full binary tree except for the last node level, as shown in Figure 4.22.

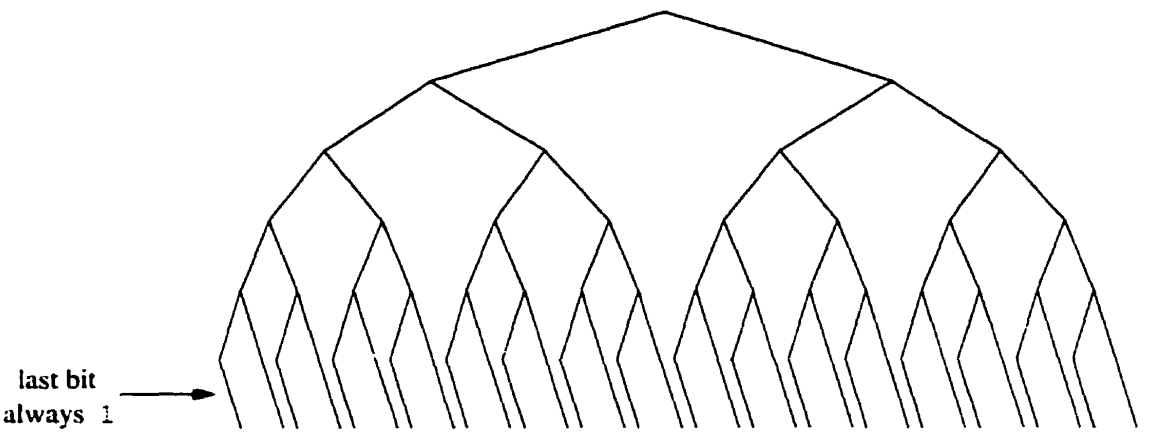


Figure 4.22: Trie constructed with all the odd keys in a small key space

With a trie constructed in this manner, any insert operation will modify only the leaf page level of the trie. So, to generate a greater probability of modifying pages above the leaf page level, we construct a sparser trie. As with B-trees, we use 40,000 32-bit keys to build a trie. However, these initial keys are from a much larger range; specifically, from a random permutation of the following

40,000 values: -2 000 000 000, -1 999 900 000, ..., 1 999 900 000.

Due to the different keys used to construct the trie, all operations use different key ranges than they do for the B-tree experiments. In the B-tree experiments, searches have roughly a 50% chance of succeeding. To achieve this for the trie experiments, we fill half our search key space with the 40,000 keys already in the trie and the other half with random numbers. We then create a random permutation of the search key space. Inserts, as with the B-tree experiments, must always succeed. We fill the insert key space with random numbers and ensure that every key in the insert key space is unique and not already in the trie. Since Java uses the two's complement format to represent negative numbers, care is taken to select proper keys for appends. Keys for appends increase sequentially from -99 999 onwards. This way, all appends operate on the rightmost leaf pages of the trie, as they do with a B-tree.

Modification of Fanout

Since multiple trie edges often enter the same page from the above page level, we choose not to use different fanouts to vary the trie structures. Rather, we modify t, the number of node levels per page. We select t such that the trie has the same number of page levels as a B-tree that we constructed. A trie can have either $t = 11$ or $t = 6$. A trie with $t = 11$ has 3 page levels, just like a B-tree with a fanout of 200 and a trie with $t = 6$ has 6 page levels, just like a B-tree with a fanout of 8. We will refer to tries with $t = 11$ as having high fanout and tries with $t = 6$ as having low fanout. The capacity of each trie page is 2^t nodes, so the page capacities for $t = 11$ and $t = 6$ are 2048 and 64 nodes respectively. There are initially 486 pages in the three-level trie, compared to initially 267 pages in the three-level B-tree. There are initially 16980 pages in the six-level trie, compared to initially 8463 pages in the six-level B-tree.

Modification of Buffer Pool Size

The final parameter that we modify is the buffer pool size. Because the tries have more pages than the B-trees, less of the trie is in memory if we use the same buffer pool size. If a smaller percentage of the trie is in memory, more disk I/O will need to be performed. For the B-tree experiments, 75% of the initial high-fanout B-tree is in memory and 7% of the initial low-fanout B-tree is in memory.

For the trie experiments, we increase the buffer pool size for the trie with $t = 11$ to 365 pages so that 75% of the pages are in memory and, for the trie with $t = 6$, to 1200 pages so that 7% of the pages are in memory.

4.2.2 Addition of a New Parameter For Use in Trie Experiments

For the trie, only the nodes (which are bit pairs) are stored in each trie page. All other information; such as, T-counts, B-counts, and node counts are stored separately from the pages. A trie therefore consists of two files: the page file and the information file. The structures of the files are in Figures 4.23 and 4.24.

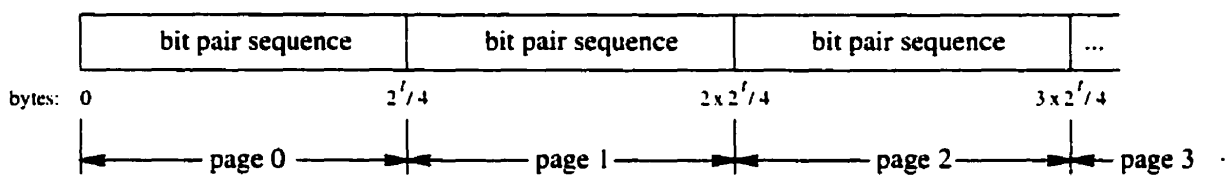


Figure 4.23: Trie page file format

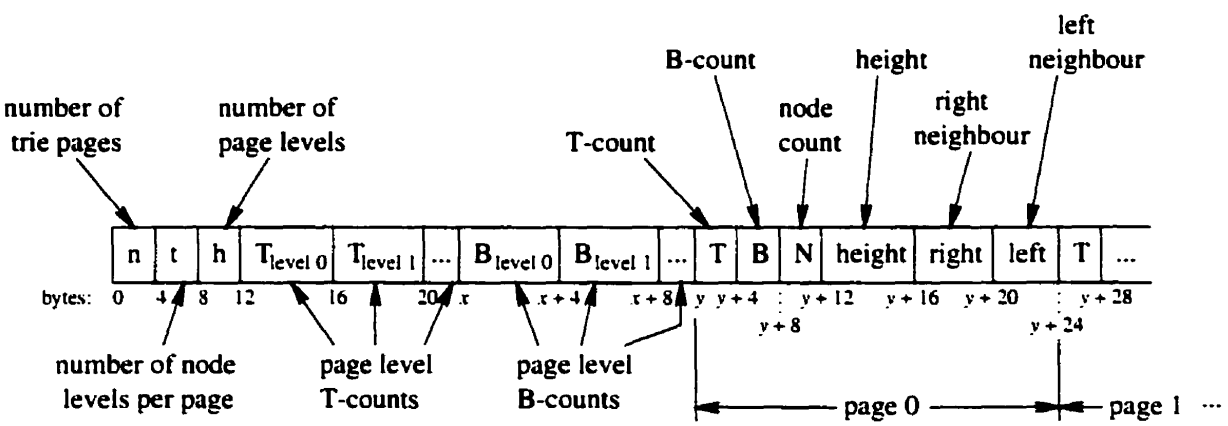


Figure 4.24: Trie information file format

The page file contains the bit pair sequences for every page. The bit pair sequence for any page is easy to locate because the bit pair sequences are sorted by page ID. The information file contains 32-bit integer data. General information about the trie is stored at the beginning of the file: the number

of trie pages, the number of node levels per page (t), and the number of page levels. Following that are the T-counts for each page level and the B-counts for each page level, with page level 0 being the root level. Even though the root page level is level 0 in our simulation program, we will continue to refer to the root page as being on level 1 and the leaf pages as being on level h. We will Finally, the information file contains the T-count, B-count, node count, height, right neighbour page ID, and left neighbour page ID of each page (within the page levels) sorted by page ID. The left neighbour is not necessary and is present only for debugging purposes.

Each experiment we perform is a simulation. Before starting an experiment, the trie information file is loaded into memory. A trie operation needs to access this information, specifically the T-count and right neighbour, in order to navigate the trie. To account for the CPU cost of accessing this information from memory, we add a new parameter called PAGE_COUNT_CPU. It is important to note that the T-count for a page is accessed from memory without accessing the actual page from the buffer or from disk. The simulation parameters for the B-tree and trie experiments are in Table 4.1 on page 99.

Table 4.1 contains the values we may use for the various parameters. With these values, we perform the 7 experiments that were performed with B-trees. As with the B-tree experiments, the disk time is measured in terms of the number of CPU instructions that can be performed while the disk is in use. The maximum disk time is 27 ms, which is the equivalent to performing 540,000 CPU instructions with a 20 MIPS CPU.

4.3 Experimental Results

We perform experiments that correspond to the B-tree experiments described in Chapter 3. We now present and discuss the results for the trie concurrency experiments. We also compare and contrast the results with those obtained for the B-trees under similar conditions. Due to the great amount of time required for each trie simulation, the trie throughput curves show the mean throughput for only 10 simulations. The error bars show the standard deviation. As with the B-tree experiments, we break down the average number of CPU instructions required for each operation during only 1 simulation to better understand our results.

Parameter	Description	Values (in CPU instructions unless otherwise noted)	
		B-Tree	Trie
NUM_CPUS	Number of CPUs	1, ∞	1, ∞
NUM_DISKS	Number of disks	8, ∞	8, ∞
CPU_SPEED	in MIPS (millions of instructions per second)	20	20
DISK_TIME	Includes seek, latency, and transfer time (max 27 ms)	0..540000	0..540000
CC_CPU	CPU cost for a lock, upgrade lock, or unlock request	100	100
BUF_CPU	CPU cost for a buffer call	1000	1000
PAGE_SEARCH_CPU	CPU cost for a page search	50	50
PAGE_MODIFY_CPU	CPU cost for a page modification	500	500
PAGE_COPY_CPU	CPU cost to copy a page between buffer and disk	1000	1000
PAGE_COUNT_CPU	CPU cost to access T-count from memory	-	50
FANOUT	Number of entries per B-tree page	8, 200	-
t	Number of node levels per trie page	-	6, 11
NUM_PAGE_LEVELS	Number of page levels	6, 3	6, 3
INITIAL_NUM_KEYS	Number of keys in initial B-tree or trie	40000	40000
NUM_BUFFERS	Number of buffers	200, 600, ∞	365, 1200, ∞
MPL	Multiprogramming level (number of terminals)	1..300	1..300
NUM_OPERATIONS	Number of operations performed in each simulation	10000	10000
SEARCH_PROB	Probability of search operation	0.0, 0.5	0.0, 0.5
INSERT_PROB	Probability of insert operation	0.0, 1.0	0.0, 1.0
APPEND_PROB	Probability of append operation	0.0, 0.5	0.0, 0.5

Table 4.1: Parameters for B-tree and trie simulations

4.3.1 Trie Experiment 1: High Fanout, 100% Inserts, Infinite Resources, and In Memory

The throughput curve for the trie with $t = 11$ and 3 page levels is in Figure 4.25. We compare the trie performance with that of a B-tree with 3 page levels, whose throughput curve we presented in Chapter 3 and now present again in Figure 4.26. The breakdown of the average number of CPU instructions for each operation is in Table 4.2.

From the two curves, we see that the throughput for the B-tree continues to increase as the MPL increases, but the throughput for the trie reaches a maximum at an MPL of 100 terminals. Since there are infinite resources, the only factor that changes as the MPL increases is the time spent waiting for locks. The reason for this is that operations traverse the trie sequentially and the B-tree logarithmically and that lock-coupling in the trie causes increased lock waiting times. In fact, from

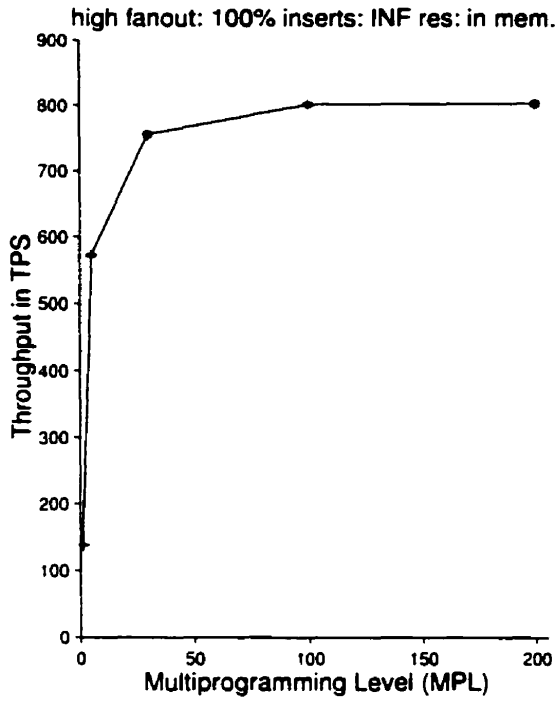


Figure 4.25: Trie Experiment 1 throughput

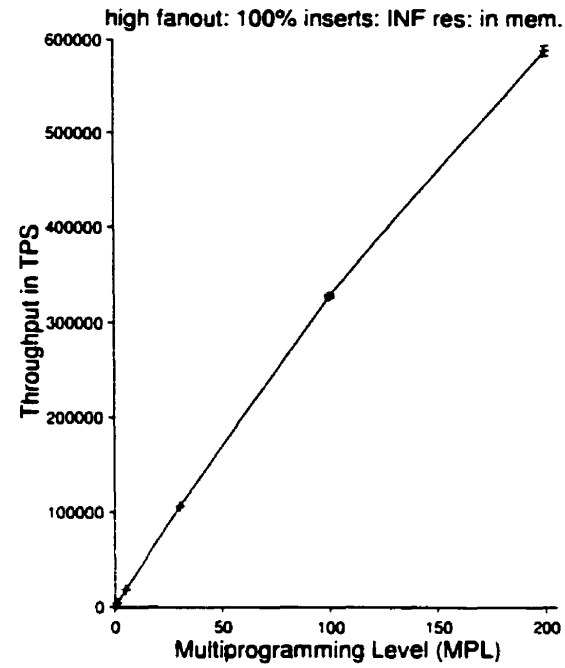


Figure 4.26: B-tree Experiment 1 throughput

analyzing the number of CPU instructions needed for CC and PAGE_COUNT requests during a trie insert operation in Table 4.2, we will see that a terminal usually locks every single page in the trie.

In the following discussion, the costs cited are from Table 4.1 and are in units of CPU instructions for a 20 MIPS CPU. This discussion is an explanation; thus, it is an approximation.

The 10,000 insertions that we perform cause about 115 page splits in the trie. However, about 58% of the page splits occur in the first 5,000 operations. As stated previously, there are initially 486 pages in the trie. Therefore, we will assume an average trie size of $(0.58 \times 115) + 486 \approx 553$ pages. There are very few link chases. The maximum number of link chases over 10,000 operations is only about 11, which occurs at an MPL of 200. Therefore, for brevity, we will ignore them in our explanatory calculations below.

CC Requests

Each concurrency control request costs 100 instructions, so, if each page is locked and unlocked, we estimate that there should be $553 \times 2 \times 100 \approx 110600$ CPU instructions required for CC

MPL	Request Type	Time (CPU Instructions) B-Tree	Trie
1	CC	803	111385
	BUF	4020	4011
	PAGE COUNT	-	27821
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1001
	Lock Wait	0	0
	Total	5484	144268
5	CC	803	111681
	BUF	4021	4012
	PAGE COUNT	-	27895
	PAGE SEARCH	151	51
	PAGE MODIFY	511	1000
	Lock Wait	18	29442
	Total	5503	174079
30	CC	803	111493
	BUF	4021	4012
	PAGE COUNT	-	27848
	PAGE SEARCH	151	51
	PAGE MODIFY	510	999
	Lock Wait	154	648606
	Total	5639	793009

MPL	Request Type	Time (CPU Instructions) B-Tree	Trie
100	CC	804	111640
	BUF	4023	4012
	PAGE COUNT	-	27884
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1000
	Lock Wait	624	2341717
	Total	6111	2486304
200	CC	804	111528
	BUF	4026	4012
	PAGE COUNT	-	27856
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1000
	Lock Wait	1280	4822741
	Total	6770	4967188

Table 4.2: Trie Experiment 1 CPU usage per operation

requests. Considering the page level counts for the 3 page levels and the upgrade lock request yields an expected 111300 CPU instructions required for concurrency control requests if every page is locked. From Table 4.2, we see that the number of CPU instructions required for CC requests nears what we would expect if every page in is locked and unlocked during an insertion.

BUF Requests

Since the trie is 3 levels in height, 3 pages are accessed from the buffer. The fourth buffer call occurs when the terminal, after upgrading its S-lock to an X-lock, checks the page again to confirm that it will still perform its modification of the page. So, at a cost of 1000 instructions per buffer call, we so far have 4000 instructions required for buffer requests. Additional buffer cost occurs because of the buffer call needed to put a new page into the buffer each time a page splits. Since we average about 115 page splits over 10,000 insertions, the added buffer cost per insertion due to page splits is about $115/10000 \times 1000 = 11.5$ instructions. Hence, the average cost for buffer calls is about 4011.5 instructions, as shown in Table 4.2 for BUF requests.

PAGE_COUNT Requests

The number of CPU instructions required to access the T-counts from memory also indicates that most pages are locked and unlocked since terminals always lock pages before accessing the corresponding T-counts. Since each T-count access costs 50 instructions, our average trie is 553 pages, and there are 3 page level counts, we expect $(553 + 3) \times 50 = 27800$ instructions to be required to access all the T-counts as indicated for PAGE_COUNT requests in Table 4.2.

PAGE_SEARCH Requests

Since a page search costs 50 instructions, Table 4.2 indicates that a terminal usually only searches 1 page, the root page, during an insertion.

PAGE_MODIFY Requests

Since the root page is usually only searched and the trie has 3 page levels, we expect the terminal to modify 2 pages. Table 4.2 shows a cost of slightly more than 1000 instructions used for page modification. Since the cost of modifying a page is 500 instructions, indeed, usually the terminal modifies 2 pages.

The main factor affecting the shape of the throughput curves is the time spent waiting for locks. According to Table 4.2, the buffer access cost dominates the total instructions needed per B-tree insertion. However, for an MPL of 30 or greater, the time spent waiting for locks dominates the total instructions needed per trie insertion. The lock-coupling performed in the trie is very restrictive. Once the lock waiting time dominates the total cost of an insertion, the throughput for the trie no longer increases since the total cost per insertion increases by the same factor as the MPL.

The difference in actual throughput values in the graphs between the B-tree and the trie are directly related to the ratio between the cost of a B-tree insertion and a trie insertion. For example, from Table 4.2, at an MPL of 200, the cost per B-tree insertion is 6770 instructions and the cost per trie insertion is 4967188 instructions. Since the trie insertion takes about 734 times longer than a B-tree insertion, the B-tree throughput is 734 times greater than that of the trie. Hence, we see a

throughput of 590800 TPS for the B-tree and a throughput of 805 TPS for the trie at an MPL of 200 in Figures 4.25 and 4.26 on page 100.

4.3.2 Trie Experiment 2: High Fanout, 100% Inserts, Infinite Resources, and 365 Buffers

The throughput curve for the trie with $t = 11$ and 3 page levels is in Figure 4.27. We compare the trie performance with that of a B-tree with 3 page levels. The B-tree throughput curve presented in Chapter 3 is now in Figure 4.28. We break down the average CPU usage for each operation into its various components in Table 4.3. We do not discuss results that are unaffected by the limitation of buffer size since they are explained in Section 4.3.1.

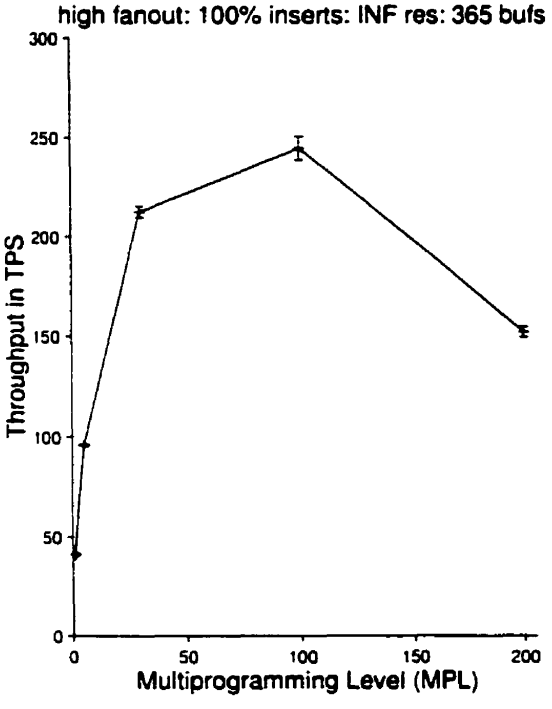


Figure 4.27: Trie Experiment 2 throughput

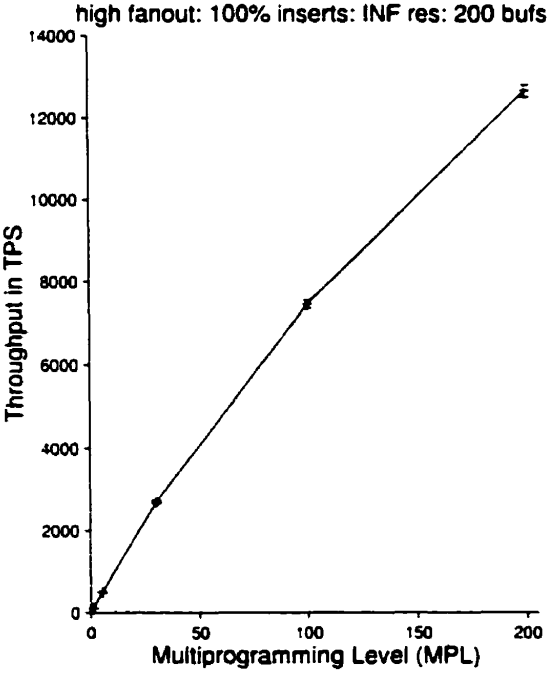


Figure 4.28: B-tree Experiment 2 throughput

From the two curves, again we see that the throughput for the B-tree continues to increase as the MPL increases, but the throughput for the trie reaches a maximum at an MPL of 100 terminals. In fact, throughput decreases for the trie such that there is less throughput with 200 terminals than with

MPL	Request Type	Time (CPU Instructions) B-Tree	Trie
1	CC	803	111707
	BUF	4747	5273
	PAGE COUNT	-	27901
	PAGE SEARCH	151	51
	PAGE MODIFY	511	1000
	PAGE COPY	726	1262
	Disk Time	196894	341324
	Lock Wait	0	0
	Total	203831	488517
5	CC	803	111432
	BUF	4705	5270
	PAGE COUNT	-	27832
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1000
	PAGE COPY	685	1258
	Disk Time	184865	339616
	Lock Wait	3893	556989
	Total	195610	1043447
30	CC	803	111611
	BUF	4722	5262
	PAGE COUNT	-	27877
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1000
	PAGE COPY	701	1250
	Disk Time	188689	341304
	Lock Wait	25829	2282671
	Total	221404	2771025

MPL	Request Type	Time (CPU Instructions) B-Tree	Trie
100	CC	804	111642
	BUF	4715	5269
	PAGE COUNT	-	27885
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1001
	PAGE COPY	691	1258
	Disk Time	187969	360476
	Lock Wait	72970	7697591
	Total	267809	8205173
200	CC	804	111289
	BUF	4709	5261
	PAGE COUNT	-	27797
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1000
	PAGE COPY	682	1249
	Disk Time	187366	396424
	Lock Wait	123151	26373921
	Total	317372	26916990

Table 4.3: Trie Experiment 2 CPU usage per operation

30 terminals. The additional time required to access a disk affects the results. Even though the number of disks is infinite, the operation holds its lock on the page (often an X-lock) for a longer time. Throughput is reduced greatly since the trie is traversed sequentially instead of logarithmically.

BUF Requests

The use of a disk affects the number of buffer calls because we must write out the LRU buffer page before reading in the new page from disk. So, we now calculate the number of times we expect to use a disk. We assume that the root page is always in the buffer because it is navigated most often and that the remaining 2 page levels are distributed evenly in the buffer. Since there are 365 buffers and an average of 553 trie pages, we calculate about a $1 - (365 - 1)/(553 - 1) \approx 0.34$ probability that a specific page other than the root is not in the buffer. However, since there are 2

page levels where the page may not be in the buffer, we expect the number of buffer misses per operation to be $(0.34 \times 0.66 \times 2 \times 1) + (0.34 \times 0.34 \times 1 \times 2) \approx 0.68$. Each buffer miss generates 2 additional buffer calls (write-out and read-in), so we expect $0.68 \times 2 \approx 1.36$ buffer calls per insert operation due to disk I/O. Also, each page split generates 2 buffer requests because the LRU buffer page must be written to disk to allow the new page to go into the buffer. So, with about 115 page splits over 10,000 operations, we expect $2 \times 115/10000 \approx 0.02$ buffer requests due to page splits per operation. Adding these figures to the 4 buffer calls needed to access the trie pages, we expect about 5.38 buffer calls per insertion. At a cost of 1000 instructions per buffer call, this amounts to about 5380 instructions. Examining the instructions used for BUF requests with tries in Table 4.3, we see that the results support this finding.

PAGE_COPY Requests

Each buffer miss generates 2 requests for disk I/O due to the write-out fo the LRU buffer page and read-in for the read page. Additionally, each page split generates 1 disk I/O request because the LRU buffer page is written to disk to allow the new page to go into the buffer. So, with 0.68 buffer misses per operation and about 0.01 page splits per operation, we expect $1.36 + 0.01 \approx 1.37$ occurrences of disk I/O per operation. Each time a terminal performs disk I/O, it makes a PAGE_COPY request. Each PAGE_COPY request costs 1000 instructions, so we expect about 1370 instructions on average needed for copying pages into or out of the buffer as shown in Table 4.3.

Disk Time

The disk time needed per read or write varies from 0 to 27 ms, which is the equivalent to performing 0 to 540000 CPU instructions with a CPU of speed 20 MIPS. Since we write out and read in a page whenever a buffer miss occurs, we expect that each time a buffer miss occurs, we are delayed by an average time equal to the CPU performing $2 \times 540000/2 = 540000$ instructions. Since the number of buffer misses is expected to be about 0.68, we expect a time equivalent to performing about $0.68 \times 540000 \approx 367200$ CPU instructions to be devoted to disk I/O. The measurements for

disk time in Table 4.3 for tries support this finding.

The main factor affecting the shape of the throughput curves is the time spent waiting for locks. All measurements remain about the same except for the lock waiting times, which grow as MPL increases. The rate at which lock wait times for the B-tree grow is relatively unchanged and less than the rate at which the MPL increases. The disk time for the B-tree is always greater than the lock wait time, which limits the effects of increased lock waiting time. For the tries however, lock wait times jump by over a factor of 3 when going to an MPL of 200 from an MPL of 100. Because of this dramatic increase in lock wait times and the fact that the lock wait time accounts for most of the operation time, the throughput for the trie decreases when the MPL is greater than 200.

The ratio of the throughput values between the B-tree and trie is equal to the ratio of the operation times between the B-tree and trie. For example, at an MPL of 100, which is the point of maximum throughput for the trie, the B-tree has a throughput of 7468 TPS and the trie has a throughput of 244 TPS. The throughput of the B-tree is 30.6 times more than that of the trie because the total of 8205173 instructions required to do a typical trie insertion is 30.6 times more than the total of 267809 instructions required to do a typical B-tree insertion.

4.3.3 Trie Experiment 3: High Fanout, 100% Inserts, 1 CPU, 8 Disks, and 365 Buffers

The throughput curve for the trie with $t = 11$ and 3 page levels is in Figure 4.29. We compare the trie performance with that of a 3-level B-tree. The B-tree throughput curve that was presented in Chapter 3 is now in Figure 4.30. Table 4.4 contains the components of the average CPU usage required by each operation.

The throughput curves are similar as those we get when we have infinite resources and limited buffer space. Again the trie throughput reaches a maximum at 100 MPL and then quickly decreases as MPL increases. The B-tree throughput continues to increase as MPL increases. We now use the measurements in Table 4.4 to explain the throughput results.

The trie results in Table 4.4 are the same as those obtained for the 3-level trie with infinite CPUs and disks in Section 4.3.2, except for the wait times for a disk, CPU, and lock. With a single CPU

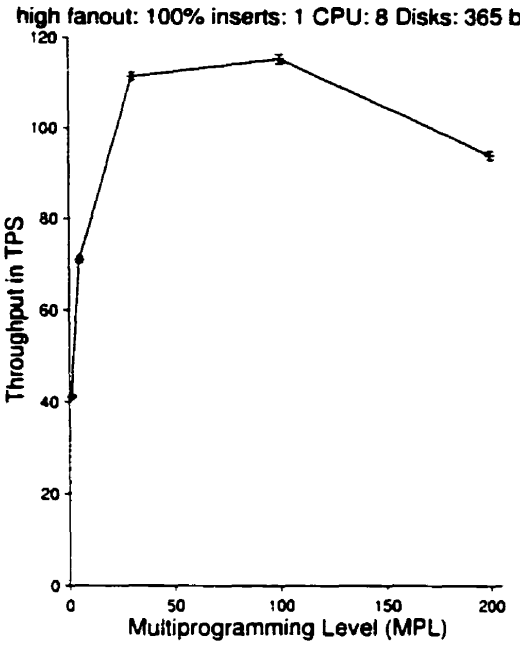


Figure 4.29: Trie Experiment 3 throughput

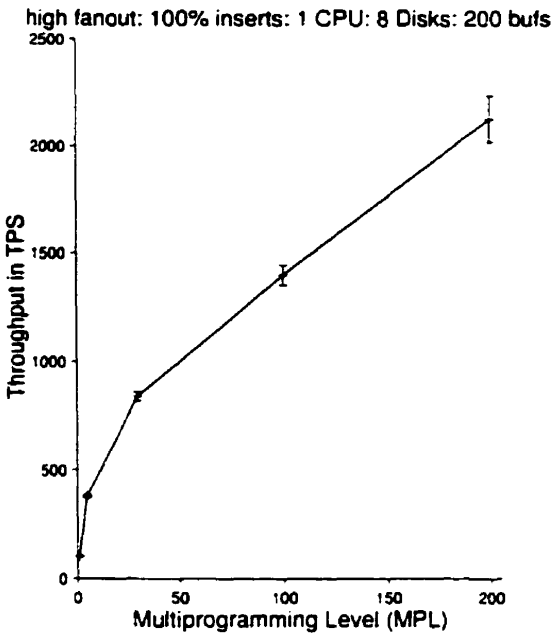


Figure 4.30: B-tree Experiment 3 throughput

and only 8 disks, operations now must now sometimes wait to use the CPU or a disk. Operations hold locks on pages longer because of the limited resources, causing longer wait times for locks.

The waiting time for locks and the CPU dominate the time required for each trie operation; whereas, the disk time and wait dominates the time required for each B-tree operation. Due to the sequential locking that trie operations perform and the large number of T-counts accessed, trie operations use the CPU much more than B-tree operations. Hence the CPU becomes the bottleneck for the trie. Due to the logarithmic locking that B-tree operations perform, disk times and waits are longer for these operations.

Once the MPL reaches 100 terminals, the increase in trie operation time is at a higher rate than the increase in MPL, so throughput decreases. From Table 4.4, we see that the total instructions required for each operation increases from 17682796 to 42458293 when increasing MPL from 100 to 200. The time for each operation increases by a factor of about 2.4, but the MPL increases only by a factor of 2; hence, throughput decreases for the trie.

The ratio of throughput values for the B-tree and trie is the same as the ratio of operation time for the trie and B-tree. For example, at the maximum throughput for the trie, which occurs at an

MPL	Request Type	Time (CPU Instructions) B-Tree	Trie
1	CC	803	111503
	BUF	4712	5284
	PAGE COUNT	-	27850
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1000
	PAGE COPY	692	1272
	Disk Time	184436	344062
	Disk Wait	0	0
	CPU Wait	0	0
	Lock Wait	0	0
	Total	191304	491023
5	CC	803	111712
	BUF	4704	5254
	PAGE COUNT	-	27902
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1000
	PAGE COPY	684	1243
	Disk Time	185261	333065
	Disk Wait	63005	28344
	CPU Wait	899	258495
	Lock Wait	3216	633793
	Total	259232	1400859
30	CC	803	111494
	BUF	4623	5275
	PAGE COUNT	-	27848
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1000
	PAGE COPY	602	1264
	Disk Time	272098	343334
	Disk Wait	434024	80814
	CPU Wait	5315	785033
	Lock Wait	14395	4041029
	Total	732521	5397142

MPL	Request Type	Time (CPU Instructions) B-Tree	Trie
100	CC	804	111640
	BUF	4419	5273
	PAGE COUNT	-	27885
	PAGE SEARCH	151	51
	PAGE MODIFY	510	1001
	PAGE COPY	393	1261
	Disk Time	600467	356382
	Disk Wait	711876	104725
	CPU Wait	30664	2189629
	Lock Wait	50890	14884950
	Total	1400173	17682796
200	CC	806	111512
	BUF	4306	5251
	PAGE COUNT	-	27852
	PAGE SEARCH	151	51
	PAGE MODIFY	511	1000
	PAGE COPY	269	1240
	Disk Time	813785	386278
	Disk Wait	653324	109336
	CPU Wait	145219	4395455
	Lock Wait	147281	37420318
	Total	1765653	42458293

Table 4.4: Trie Experiment 3 CPU usage per operation

MPL of 100, the throughput of the B-tree is 1428 TPS, which is about 12.6 times greater than the throughput of 113 TPS for the trie. This is because the average operation, according to Table 4.4, for trie operations at this MPL requires 17682796 instructions, which is about 12.6 times greater than the 1400173 instructions required by the average B-tree operation at this MPL.

4.3.4 Trie Experiment 4: Low Fanout, 100% Inserts, Infinite Resources, and 1200 Buffers

The throughput curve for the trie with $t = 6$ and 6 page levels is in Figure 4.31. We compare the trie performance with that of a B-tree with 6 page levels, whose throughput curve is in Figure 4.32. We break down the average number of CPU instructions per operation into the various components in Table 4.5.

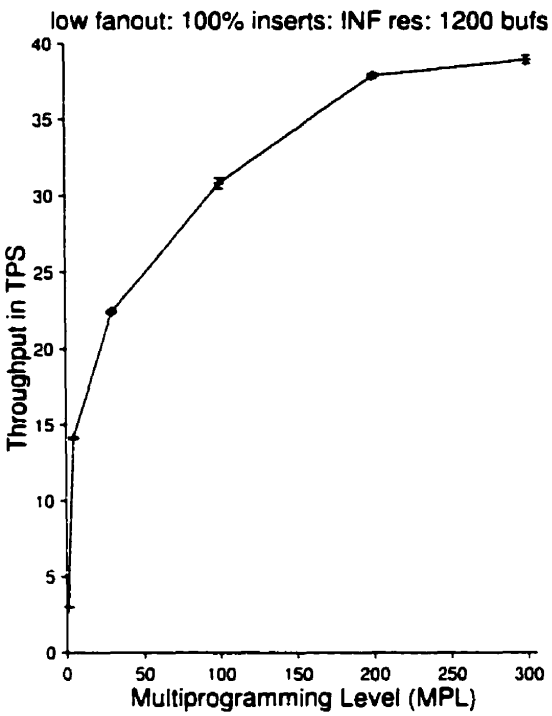


Figure 4.31: Trie Experiment 4 throughput

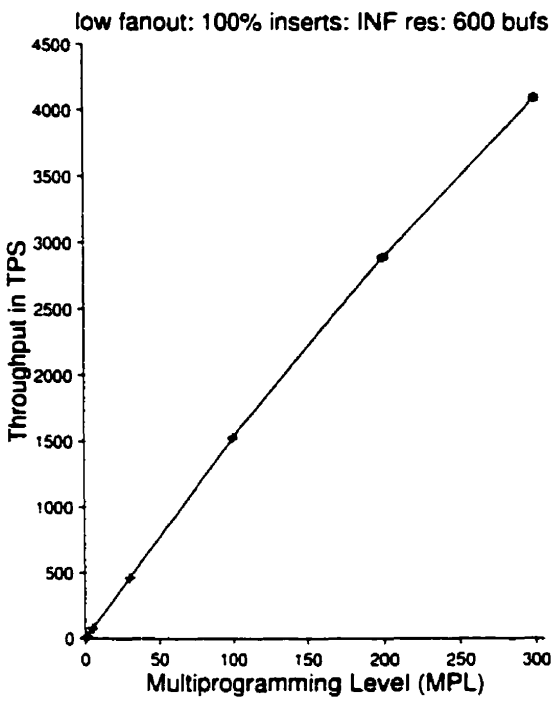


Figure 4.32: B-tree Experiment 4 throughput

From the two curves, we see that the throughput for both the trie and the B-tree continue to increase as the MPL increases, but that the throughput for the trie is nearing a maximum before the throughput for the B-tree does.

Performing 10,000 insertions in the trie with $t = 6$ yields about 3040 page splits. Half the page splits occur in the first 5,000 operations. Initially there are 16980 pages in the trie, so we will assume an average trie size of $3040/2 + 16980 \approx 18500$ pages. Link chases are extremely rare and do not affect the results.

MPL	Request Type	Time (CPU Instructions) B-Tree	Trie
1	CC	1462	3627825
	BUF	12124	15041
	PAGE COUNT	-	906916
	PAGE SEARCH	310	109
	PAGE MODIFY	707	2061
	PAGE COPY	4711	7737
	Disk Time	1274000	2092468
	Lock Wait	0	0
	Total	1293314	6652157
5	CC	1463	3628089
	BUF	12118	15018
	PAGE COUNT	-	906982
	PAGE SEARCH	310	110
	PAGE MODIFY	709	2058
	PAGE COPY	4700	7711
	Disk Time	1267638	2083672
	Lock Wait	51	428508
	Total	1286989	7072147
30	CC	1463	3621156
	BUF	12153	15025
	PAGE COUNT	-	905249
	PAGE SEARCH	311	109
	PAGE MODIFY	710	2057
	PAGE COPY	4732	7722
	Disk Time	1277944	2078374
	Lock Wait	1260	20135100
	Total	1298572	26764791

MPL	Request Type	Time (CPU Instructions) B-Tree	Trie
100	CC	1462	3631474
	BUF	12157	15030
	PAGE COUNT	-	907828
	PAGE SEARCH	310	109
	PAGE MODIFY	705	2064
	PAGE COPY	4743	7718
	Disk Time	1286619	2092439
	Lock Wait	5455	58189056
	Total	1311452	64845719
200	CC	1464	3622391
	BUF	12419	15010
	PAGE COUNT	-	905558
	PAGE SEARCH	311	109
	PAGE MODIFY	708	2059
	PAGE COPY	4997	7711
	Disk Time	1354797	2092743
	Lock Wait	13731	99294980
	Total	1388426	105940561
300	CC	1466	3622701
	BUF	12664	15038
	PAGE COUNT	-	905635
	PAGE SEARCH	311	109
	PAGE MODIFY	712	2060
	PAGE COPY	5229	7736
	Disk Time	1416964	2104459
	Lock Wait	28025	148984209
	Total	1465372	155641946

Table 4.5: Trie Experiment 4 CPU usage per operation

CC Requests

If a terminal locks and unlocks every page, there should be about $18500 \times 2 \times 100 \approx 3700000$ CPU instructions required CC requests. The instructions required for CC requests for the trie in Table 4.5 are slightly below our estimate, indicating that the terminals lock most, but not all, trie pages.

BUF Requests

Since the trie has 6 page levels, the terminal accesses a minimum of 6 pages from the buffer. After upgrading its lock to an X-lock, the terminal accesses the page again to check that modification will still occur. The additional buffer accesses are due to disk I/O and page splitting.

The expected number of disk accesses per operation is calculated as follows. Say that most page

splits are on the bottom 4 page levels and that the top 2 page levels are always in the buffer. There are 1200 buffers and 61 pages in the top 2 page levels, so for the bottom 4 page levels, there is a $1-(1200-61)/(18500-61) \approx 0.938$ probability that a specific page is not in the buffer. However, since there are four page levels where the page may not be in the buffer, we expect the number of buffer misses per operation to be $(0.062^3 \times 0.938 \times 4 \times 1) + (0.062^2 \times 0.938^2 \times 6 \times 2) + (0.062 \times 0.938^3 \times 4 \times 3) + (0.938^4 \times 1 \times 4) \approx 3.75$. Since there are 2 buffer calls for each buffer miss (due to writing out and reading in), we expect that there will be $3.75 \times 2 = 7.5$ buffer accesses per operation for disk I/O. In addition, there are 2 buffer calls made for each page split to write out the LRU page and read in the new page. With about 3040 page splits over 10,000 operations, we expect $2 \times 3040/10000 \approx 0.608$ buffer calls per operation due to page splitting. Adding the buffer calls, we arrive at $7 + 7.5 + 0.608 \approx 15.1$ buffer calls for each operation. At 1000 instructions per call, we estimate about 15100 instructions needed for buffer requests, which is close to the instructions needed for buffer requests for tries in Table 4.5.

PAGE_COUNT Requests

Since terminals lock most trie pages, they access most page counts. With an average trie size of 18500 pages plus the 6 page level counts, accessing all counts at a cost of 50 instructions per access yields $18506 \times 50 \approx 925300$ instructions total for page count accessing. Results for trie page count accesses in Table 4.5 are what we expect if most counts are accessed.

PAGE_SEARCH Requests

Looking at the number of CPU instructions required for a typical insert operation to search a page in Table 4.5, we see that the operation usually searches 2 pages. These searched pages are the root page and the root page's child. At a cost of 50 instructions per search, a total of 100 instructions is used for page searching. Since the measurements for page searches in Table 4.5 are slightly higher, the operation will search 3 or more pages more often than only the 1 root page.

PAGE_MODIFY Requests

Operations usually search 2 pages in the trie, so that leaves 4 pages to modify. In addition, page splits require an additional page modification. With about 0.304 page splits per operation and a cost of 500 instructions per modification, we expect $4.304 \times 500 \approx 2152$ instructions devoted to page modification. The measurements for page modification in Table 4.5 are slightly lower because pages will occasionally search 3 pages and modify only 3 pages.

PAGE_COPY Requests

With an expected 3.75 buffer misses and 2 disk I/O requests per miss, there are about 7.5 requests to copy a page into or out of the buffer. Also, each page split causes a page copy request. So we expect $7.5 + 0.304 \approx 7.8$ page copy requests per operation. At 1000 instructions per request, this requires about 7800 instructions, which is close to the PAGE_COPY measurements for tries in Table 4.5.

Disk Time

We perform disk I/O every time we perform a page copy, so, for an average disk time equivalent to 270000 instructions, we expect a time equivalent to performing $7.8 \times 270000 \approx 2106000$ instructions spent for disk I/O. The disk time measurements for tries in Table 4.5 are very close to our estimate.

As with the 3-level trie with infinite CPUs and disks but limited buffers, the main factor affecting the shape of the throughput curves is the lock wait time. The lock wait time increases for the B-tree, but its effects are not very strong because the disk time is much greater. For the trie, however, the lock wait time quickly dominates the cost of each operation. The increase in lock wait time is not as rapid as it is for the 3-level trie; hence, the curve reaches a gradual maximum. When the MPL doubles from 100 to 200, the lock wait time increases by only a factor of 1.7. When the MPL increases by a factor of 1.5 from 200 to 300, the lock wait time increases by a factor of about 1.5; therefore, reaching a plateau in terms of throughput.

The ratio between the throughput for the B-tree and trie equals the ratio between the operation

costs for the B-tree and trie. At an MPL of 300, the throughput of the B-tree is 4095 TPS and the throughput of the trie is 39 TPS. The B-tree throughput is 105 times greater than that of the trie because the average B-tree operation requires 105 times fewer instructions to perform than the trie. According to Table 4.5, the average instructions required per operation is 1465372 for the B-tree and 155641946 for the trie.

4.3.5 Trie Experiment 5: Low Fanout, 100% Inserts, 1 CPU, 8 Disks, and 1200 Buffers

The throughput curve for the trie with $t = 6$ and 6 page levels is in Figure 4.33. We compare the trie performance with the performance of a 6-level B-tree. The B-tree throughput curve is in Figure 4.34. The breakdown of the average CPU usage for each operation into the various components is in Table 4.6.

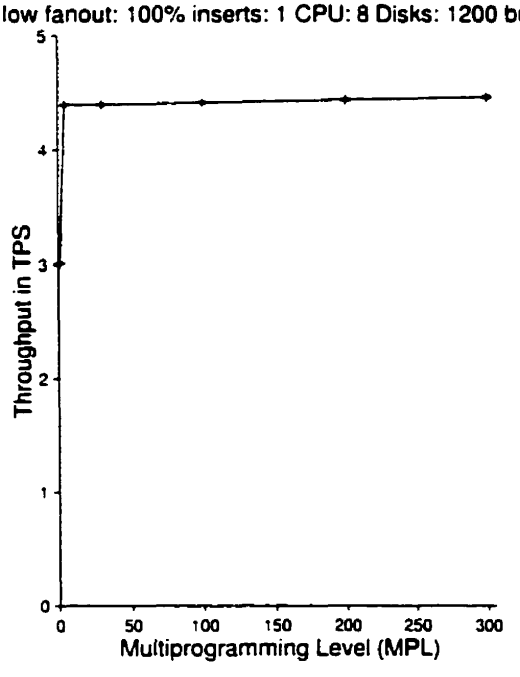


Figure 4.33: Trie Experiment 5 throughput

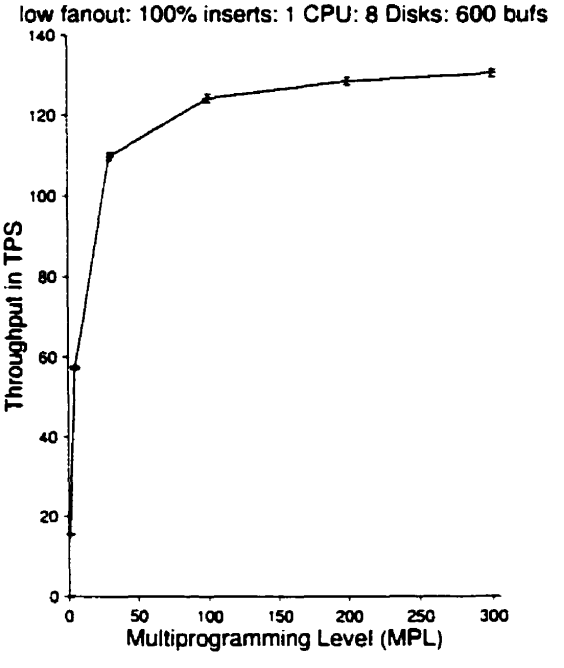


Figure 4.34: B-tree Experiment 5 throughput

From the two curves, we see that the throughput for the trie is very low and almost constant. The throughput for the B-tree under similar circumstances grows and is nearing a maximum throughput.

MPL	Request Type	Time (CPU Instructions)	
		B-Tree	Trie
1	CC	1463	3626013
	BUF	12147	14998
	PAGE COUNT	-	906463
	PAGE SEARCH	310	109
	PAGE MODIFY	710	2062
	PAGE COPY	4728	7692
	Disk Time	1277722	2073223
	Disk Wait	0	0
	CPU Wait	0	0
	Lock Wait	0	0
	Total	1297081	6630559
5	CC	1461	3626459
	BUF	12106	15026
	PAGE COUNT	-	906575
	PAGE SEARCH	310	109
	PAGE MODIFY	707	2060
	PAGE COPY	4692	7725
	Disk Time	1265706	2080955
	Disk Wait	434376	72494
	CPU Wait	590	15320956
	Lock Wait	400	762577
	Total	1720350	22794976
30	CC	1463	3617560
	BUF	12068	15021
	PAGE COUNT	-	904350
	PAGE SEARCH	310	109
	PAGE MODIFY	708	2058
	PAGE COPY	4650	7719
	Disk Time	1325745	2085550
	Disk Wait	4085154	77724
	CPU Wait	2186	28827956
	Lock Wait	4274	100822592
	Total	5436559	136360639

MPL	Request Type	Time (CPU Instructions)	
		B-Tree	Trie
100	CC	1464	3623213
	BUF	11986	15030
	PAGE COUNT	-	905763
	PAGE SEARCH	311	109
	PAGE MODIFY	711	2061
	PAGE COPY	4561	7723
	Disk Time	1953133	2088655
	Disk Wait	14107909	114423
	CPU Wait	10085	54722414
	Lock Wait	57221	392266305
	Total	16147379	453745695
200	CC	1463	3620631
	BUF	11916	14989
	PAGE COUNT	-	905118
	PAGE SEARCH	311	109
	PAGE MODIFY	704	2056
	PAGE COPY	4500	7689
	Disk Time	3471840	2080415
	Disk Wait	27328633	161503
	CPU Wait	35547	106381448
	Lock Wait	263450	789210397
	Total	31118362	902384355
300	CC	1465	3618501
	BUF	11954	15012
	PAGE COUNT	-	904585
	PAGE SEARCH	311	109
	PAGE MODIFY	708	2058
	PAGE COPY	4522	7710
	Disk Time	5231190	2083200
	Disk Wait	39346353	182522
	CPU Wait	75443	140551238
	Lock Wait	731293	1199166425
	Total	45403241	1346531361

Table 4.6: Trie Experiment 5 CPU usage per operation

We now analyze the data in Table 4.6 to better understand why the trie throughput is almost constant.

From Table 4.6, we see that the trie results are the same as those obtained for the 6-level trie with infinite CPUs and disks in Section 4.3.4, except for the time spent waiting for a disk, waiting for a CPU, and waiting for a lock. Because of the wait for limited CPUs and disks, operations hold locks on pages for a longer period of time, causing other operations to spend more time waiting for the locks.

For the B-tree, the time spent waiting for a disk dominates the time required for each operation; whereas, for the trie, lock and CPU waiting times dominate the time each operation requires. We

expect this because the trie operations use the CPU much more than the B-tree operations. Since trie operations lock many more pages and the CPU must be used to lock and unlock pages, as well as to access the T-counts, the CPU becomes the bottleneck for the trie. Since the total time for each trie operation increases by the same factor as MPL, throughput for the trie is at a maximum. The total time for each B-tree operation increases by a factor slightly less than the factor at which MPL increases once the MPL reaches 100 terminals.

The ratio of the B-tree throughput to the trie throughput is equal to the ratio of time required to perform a trie operation to the time required to perform a B-tree operation. At an MPL of 300, for example, a typical trie operation requires 1346531361 instructions compared with the total of 45403241 instructions required for a typical B-tree operation. The trie operations take about 29.7 times longer; hence, the throughput for the B-tree is about 29.7 times higher. At an MPL of 300, the throughput of the B-tree is about 132 TPS compared to about 4.5 TPS for the trie.

4.3.6 Trie Experiment 6: High Fanout, 50% Appends, 50% Searches, Infinite Resources, and 365 Buffers

The throughput curve for the trie with $t = 11$ and 3 page levels is in Figure 4.35. We compare the trie performance with that of a B-tree with 3 page levels, whose throughput curve is in Figure 4.36. We break down the average number of CPU instructions required per operation into the various components in Table 4.7.

With a workload of 50% appends and 50% searches, there are 5 page splits in the trie. So, since the initial trie has 486 pages, the average size of the trie is $486 + 5/2 \approx 489$ pages. Link chases are still rare, with a maximum of only about 34 occurring at an MPL of 200; therefore, we will not include them in our calculations below. We now analyze the measurements in Table 4.7 and discuss the throughput curves that result from them.

CC Requests

Since traversal of the trie is sequential from left to right and appends modify the rightmost leaf pages, appends lock and unlock every page in the trie. So, with a cost of 100 instructions per concurrency control request, we expect that tries will require $489 \times 2 \times 100 = 97800$ CPU instructions for

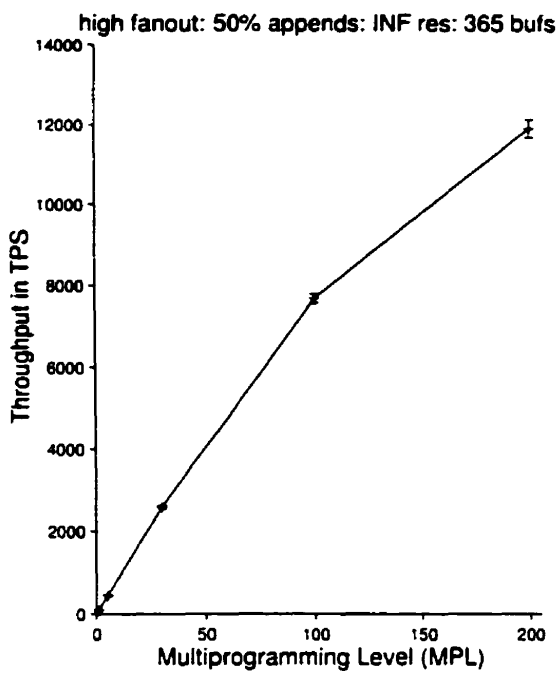


Figure 4.35: Trie Experiment 6 throughput

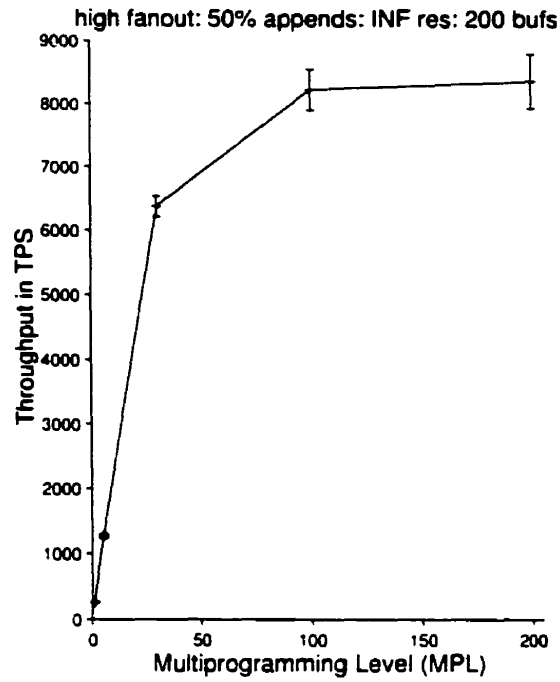


Figure 4.36: B-tree Experiment 6 throughput

locking and unlocking pages, omitting the page level counts and lock upgrade. Searches will on average lock and unlock half the trie pages, for a total of $489 \times \frac{1}{2} \times 2 \times 100 = 48900$ instructions. Since there are 50% appends and 50% searches, we expect an average of $(97800 + 48900)/2 = 73350$ CPU instructions devoted to lock and unlock requests. The CC request measurement for tries in Table 4.7 is very close to our estimate.

BUF Requests

Rarely do appends use a disk because they navigate the same pages much of the time; hence, the pages do not become LRU and written out of the buffer. The only time an append uses a disk is when writing out the LRU page after a page split to make room for the new page. Since there are only 5 page splits, we will say that only searches use a disk. For a trie of 489 pages and a buffer of 365 pages, the probability that a page is not in the buffer is about $1 - (365/489) \approx 0.254$. Since we may encounter a buffer miss twice per operation, the expected number of buffer misses per search is $(0.254 \times 0.746 \times 2 \times 1) + (0.254 \times 0.254 \times 1 \times 2) \approx 0.508$. Each time an operation

MPL	Request Type	Time (CPU Instructions) B-Tree	Trie
1	CC	703	74025
	BUF	3770	3976
	PAGE COUNT	-	18469
	PAGE SEARCH	150	125
	PAGE MODIFY	258	252
	PAGE COPY	255	474
	Disk Time	67561	128314
	Lock Wait	0	0
	Total	72696	225634
5	CC	703	73183
	BUF	3776	3976
	PAGE COUNT	-	18258
	PAGE SEARCH	151	125
	PAGE MODIFY	256	247
	PAGE COPY	257	482
	Disk Time	68461	130491
	Lock Wait	1368	708
	Total	74972	227470
30	CC	714	74193
	BUF	3846	3968
	PAGE COUNT	-	18511
	PAGE SEARCH	153	125
	PAGE MODIFY	255	252
	PAGE COPY	275	464
	Disk Time	77135	126202
	Lock Wait	19901	8939
	Total	102279	232653
100	CC	772	73755
	BUF	4130	3964
	PAGE COUNT	-	18401
	PAGE SEARCH	168	125
	PAGE MODIFY	251	250
	PAGE COPY	269	465
	Disk Time	74124	130510
	Lock Wait	166581	33865
	Total	246294	261336
200	CC	869	73932
	BUF	4613	3933
	PAGE COUNT	-	18445
	PAGE SEARCH	192	125
	PAGE MODIFY	256	253
	PAGE COPY	264	427
	Disk Time	72153	120784
	Lock Wait	393795	112936
	Total	472142	330835

Table 4.7: Trie Experiment 6 CPU usage per operation

encounters a buffer miss, the operation makes 2 buffer requests. We expect searches to call the buffer $(0.508 \times 2) + 3 \approx 4.02$ times and appends to call the buffer 4 times (as insert operations with the trie in memory do). So, the expected number of buffer accesses is $(4.02 + 4)/2 \approx 4.01$ which, at 1000 instructions each, accounts for a total of 4010 CPU instructions. Results in Table 4.7 for trie BUF requests are very close to this estimate.

PAGE_COUNT Requests

Appends access every page count, searches access half the page counts, and page count accesses use 50 instructions each. So, we expect $489 \times 50 = 24450$ instructions to be required for appends to access the page counts and $489 \times \frac{1}{2} \times 50 = 12225$ instructions to be required for searches to access the page counts. With our workload, that averages to 18338 CPU instructions required per

operation to access the page counts. The PAGE_COUNT request results for tries in Table 4.7 are very similar to our expectation.

PAGE_SEARCH Requests

Appends most often modify only a leaf page in the trie; therefore, for the 3-level trie, appends usually search 2 pages. Search operations search 3 pages. With our workload, we average 2.5 searches per operation and, since each search uses 50 instructions, we expect about 125 instructions to be devoted to page search per operation as shown in Table 4.7 for trie PAGE_SEARCH requests.

PAGE_MODIFY Requests

Since appends most often modify only a leaf page in the trie, a typical append requires 500 CPU instructions for page modification. Since only 50% of the operations are appends, we expect about 250 instructions per operation needed for modifying pages as indicated by Table 4.7 for trie PAGE_MODIFY requests.

PAGE_COPY Requests

Since there are so few page splits, we say that only searches use disk accesses. Each disk access is accompanied by a PAGE_COPY request. Since we expect 0.508 buffer misses per search and each buffer miss requires 2 disk accesses, we estimate that there exists about 1.02 disk accesses per search. Since searches account for 50% of the workload, we expect 0.508 disk accesses and PAGE_COPY requests. Each PAGE_COPY request uses 1000 instructions, so we expect 5080 instructions to be used for copying pages into and out of the buffer, which is close to the results in Table 4.7 for trie PAGE_COPY requests.

Disk Time

As stated above, we expect 0.508 disk accesses per operation. Since the average time for each disk access is equivalent to doing 270000 CPU instructions, we estimate a disk time equivalent to about

137160 instructions per operation. The disk times in Table 4.7 are close to this estimate.

The main factor affecting the shape of the throughput curves is the lock waiting time. Appends modify the same areas of the data structures; namely, the rightmost leaf pages. However, there are about 10 times more page splits in the B-tree than in the trie, which causes many more link chases to occur in the B-tree than in the the trie. Therefore, since operations must wait again after a link chase for the correct page, lock waits are greater for the B-tree operations. As a result, the trie throughput continues to grow at a higher rate than the B-tree throughput.

The ratio of the trie throughput to the B-tree throughput is equal to the ratio of the number of instructions required for B-tree operations to the number of instructions required for trie operations. At an MPL of 200 for example, the trie throughput is 12090 TPS, which is about 1.43 times greater than the B-tree throughput of 8470 TPS. We see in Table 4.7 that the total number of instructions required for a B-tree operation is on average 472142, which is about 1.43 times greater than the average number of instructions required for a trie operation, which is 330835.

4.3.7 Trie Experiment 7: High Fanout, 50% Appends, 50% Searches, 1 CPU, and in memory

The throughput curve for the trie with $t = 11$ and 3 page levels is in Figure 4.37. We compare the trie performance to the throughput in the corresponding B-tree experiment, whose curve is in Figure 4.38. The various components of the average CPU usage required per operation are in Table 4.8.

Both throughput curves indicate that no increase in throughput is made by increasing the MPL. We now analyze the measurements in Table 4.8 to determine the reasons. We do not discuss results that are unaffected by the use of only 1 CPU and an infinite buffer pool because they are explained in Section 4.3.6.

BUF Requests

No operations access the disk, so appends usually make 4 buffer calls and searches make 3 buffer calls. Since there are 50% of each operation type, we expect about 3.5 buffer calls per operation or,

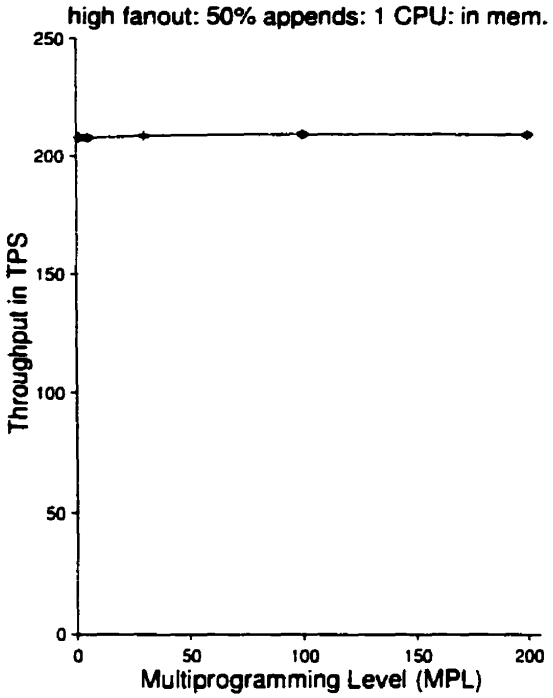


Figure 4.37: Trie Experiment 7 throughput

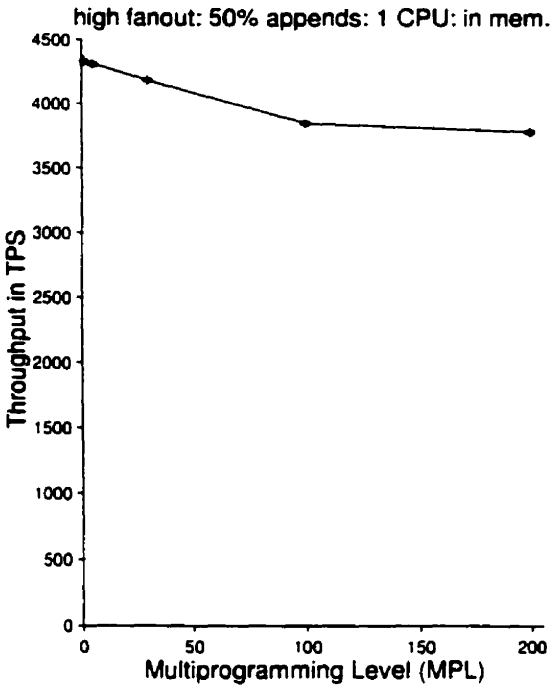


Figure 4.38: B-tree Experiment 7 throughput

at 1000 instructions per buffer call, 3500 instructions per operation devoted to buffer calls. The trie results in Table 4.8 for BUF requests are very close to our estimate.

The major factors affecting the shape of the throughput curves are the wait times for the CPU and for locks. Since trie operations use the CPU considerably more than B-tree operations because of all the locking and unlocking that they do, the CPU wait times are much longer for the trie than for the B-tree. Since B-tree operations are not idle as often, waiting for the CPU, they traverse the B-tree quicker than trie operations traverse the trie and create more lock conflicts at the rightmost leaf pages.

We compare the ratio of the B-tree throughput to the trie throughput with the ratio of the trie operation time to the typical B-tree operation time. For example, at an MPL of 200, the B-tree throughput is 3792 TPS, which is about 18.1 times greater than the trie throughput of 210 TPS. This is so because the typical trie operation requires 19073135 instructions, which is about 18.1 times greater than the number of instructions required by a typical B-tree operation, which is 1054889.

MPL	Request Type	Time (CPU Instructions)	
		B-Tree	Trie
1	CC	700	73974
	BUF	3504	3504
	PAGE COUNT	-	18456
	PAGE SEARCH	150	125
	PAGE MODIFY	252	252
	CPU Wait	0	0
	Lock Wait	0	0
	Total	4607	96311
5	CC	703	74275
	BUF	3519	3509
	PAGE COUNT	-	18531
	PAGE SEARCH	151	125
	PAGE MODIFY	255	255
	CPU Wait	16103	386655
	Lock Wait	2456	64
	Total	23186	483413
30	CC	726	73743
	BUF	3631	3498
	PAGE COUNT	-	18398
	PAGE SEARCH	157	125
	PAGE MODIFY	252	249
	CPU Wait	29138	2780029
	Lock Wait	109223	2534
	Total	143127	2878576

MPL	Request Type	Time (CPU Instructions)	
		B-Tree	Trie
100	CC	793	74011
	BUF	3970	3502
	PAGE COUNT	-	18465
	PAGE SEARCH	173	125
	PAGE MODIFY	253	251
	CPU Wait	38285	9473135
	Lock Wait	475101	45812
	Total	518576	9615302
200	CC	811	73549
	BUF	4058	3496
	PAGE COUNT	-	18349
	PAGE SEARCH	178	125
	PAGE MODIFY	254	248
	CPU Wait	65266	18507398
	Lock Wait	984323	469970
	Total	1054889	19073135

Table 4.8: Trie Experiment 7 CPU usage per operation

4.4 Summary of Results

The differences in throughput between the B-tree and trie occur primarily due to the concurrency control algorithm used for each data structure. For a data structure of n pages, the algorithm for B-trees typically locks $O(\log n)$ pages; whereas, the algorithm we have presented for tries typically locks $O(n)$ pages. Also, since trie traversal lock-couples from left to right and root to leaf, X-locks in the trie often prevent other operations from accessing any pages to the right or below the X-locked page. We now describe this difference between the algorithms in further detail, as well as the effects caused by the various restrictions of system resources.

4.4.1 Experiments 1–3: High Fanout, 100% Inserts

The B-tree operations typically lock only 3 pages; however, the trie operations lock about 556 pages (including level counts). In addition, trie operations usually place X-locks on the page level

immediately below the root; whereas, B-tree operations usually X-lock only the leaf pages. Since the trie operations lock many more pages and lock out more of the data structure than the B-tree operations, lock wait times for the trie are greatly larger than those for the B-tree.

With the addition of buffer constraints, disk usage occurs. There are longer wait times for locks because operations hold locks while performing disk I/O. Since trie operations usually start X-locking before reaching the leaf level, they usually perform any disk I/O for leaf pages while holding an X-lock. In contrast, B-tree operations perform disk I/O while holding an S-lock. Due to the sequential traversal of the trie, operations which encounter the X-locked leaf page while it is being accessed from disk are blocked from all leaf pages to the right by a considerably greater amount of time.

Limiting the number of disks and CPUs creates more bottlenecks. For the B-tree, disk contention becomes a bottleneck. For the trie, however, disk waits do not increase very much, but wait times for the CPU do increase significantly. Trie operations use the CPU much more than B-tree operations because there are many more locks placed in the trie. With both algorithms, lock waits increase due to locks being held while waiting for a resource.

4.4.2 Experiments 4–5: Low Fanout, 100% Inserts

For the low-fanout data structures, the B-tree operations usually lock only 6 pages; whereas, the trie operations usually lock most of the trie — which averages about 18500 pages. Also, for the trie, most modifications begin on the third level of the 6-level trie, so operations typically X-lock every page on the bottom 3 levels of the trie as well as every page to the right of the initially X-locked page. With a limited buffer pool, we estimate that there are about 3.75 buffer misses for each trie operation. Since these buffer misses occur among the bottom 4 levels of the trie, operations perform most disk I/O while holding X-locks. As a result, trie inserts block off much of the trie while performing disk I/O and cause lock wait times to be immensely greater than those for the B-tree.

Limiting the number of disks and CPUs creates the same bottlenecks as the high-fanout experiments. Disks become the main bottleneck for the B-tree operations and the CPU becomes a major bottleneck for the trie operations. Again, lock waits for both algorithms increase due to waiting for resources while holding locks.

4.4.3 Experiments 6–7: High Fanout, 50% Appends, 50% Searches

For the high-fanout data structures and a workload of 50% appends and 50% searches, trie throughput is greater than B-tree throughput when the buffer size is limited. Both algorithms are very similar in that the X-locks are usually placed only on the rightmost leaf page. The trie experiences only 5 page splits, compared to about 49 for the B-tree. Link chases for the B-tree are far more numerous than for the trie. At an MPL of 200, there are about 8370 link chases in the B-tree and only about 34 in the trie. With the added wait times for locks after a link chase, B-tree throughput is slightly lower than trie throughput.

Limiting the system to 1 CPU greatly affects trie throughput. Due to the intense CPU usage for trie operations, the CPU becomes the bottleneck and trie throughput remains constant. B-tree throughput is greater than trie throughput, but decreases slightly as MPL increases due to the large increase in link chases that accompanies the increase in MPL.

Chapter 5

Conclusion

This thesis presents algorithms for concurrent search and insert operations in a pointerless trie. To the best of our knowledge, these are the first algorithms for concurrent trie operations. Using simulation, we studied the performance of our trie concurrency control algorithms for a variety of situations with varying trie structure, resource contention, and workload. We also compared our algorithms' performance with that of the B^{link} algorithms. We now present a more detailed summary of our work and suggestions for future work in the study of trie concurrency.

5.1 Summary

Many database systems are used in a multiuser environment; thus, require concurrency control in order to operate correctly. B-trees have become the standard data structure for storing indices in a database system and many different algorithms have been designed to enable concurrent B-tree operations. Tries, which generate significant data compression, are useful, not only for storing indices for general databases, but also for text and spatial data. Tries have not yet, however, been applied to databases requiring concurrent operations.

There have been several performance studies for various B-tree concurrency control algorithms. We selected Srinivasan and Carey's work [SC91b] as a basis for evaluating our simulation and algorithm performance. They specify many situations that cover a variety of tree properties, resource contention, and workloads. Using an asynchronous discrete-event simulation with closed queueing,

we performed many of the experiments for which they provide throughput results. Our simulation involves events with activation times and movement of these events between various queues. Events that change the state of the system are stored in sorted order on a heap. Events that are waiting for a lock, disk, or CPU are stored on a wait queue. Once a terminal is granted the resource, its event is moved from the wait queue to the heap.

Our goal was to use our simulation to produce trie concurrency results that could be scaled onto the throughput graphs of Srinivasan and Carey. Using the results that Srinivasan and Carey present for the B^{link} algorithm with lock-coupling on ascent, we attempted to obtain a constant ratio between our B-tree results and theirs. With this constant ratio, we would be able to scale the trie throughput results obtained by our system onto the graphs.

While most of the experiments we performed generated B-tree results that were within a similar factor of their results, there were differences between our results and theirs. For the 5 results that are similar, our results are consistent with theirs by a factor of between 1.4 and 1.9. However, for 2 of the experiments, the throughput we obtained behaves differently than that obtained by Srinivasan and Carey. Satisfied with our simulation model, we used it to measure the concurrency performance of our trie algorithms.

We presented algorithms for concurrent searches and inserts in a pointerless trie. Our algorithms are relatively simple and use only S-locks and X-locks. They are also deadlock free. We attempted to use prefixes to aid in recovering from interference caused by other concurrent operations, but were unsuccessful. As a result, our algorithms require lock-coupling sequentially along each trie page level.

With tries and our trie concurrency control algorithms, we performed the experiments that were previously conducted for B-trees. While the experiments are identical in terms of workload, some modification had to be made due to differences between the trie and B-tree structures. Each trie node has at most 2 children and multiple trie edges often enter the top of a trie page from the page level above. As a result, setting a fanout in the trie similar to the fanouts used for the B-tree experiments produces a structure very different from the B-tree. Rather than modify fanout, we modified the number of node levels in each page so that each trie has the same number of page levels as its corresponding B-tree. In addition, a trie has more pages than the corresponding B-tree,

so we increased buffer sizes to maintain the same percentage of data structure initially present in the buffer.

As a result of the lock-coupling along page levels, our trie algorithms do not allow as much throughput and concurrency as the B^{link} algorithms. Where n is the number of pages in the data structure, the trie concurrency control algorithms lock $O(n)$ pages and the B-tree algorithms lock only $O(\log n)$ pages. There is also added restriction because trie operations lock-couple and the B-tree operations do not. In addition, it is not unusual for a trie insertion to lock every page in the trie, possibly with X-locks. As a result, other concurrent operations may not be able to overtake the insert operation.

The presence of resource contention also affects the trie algorithms more than the B-tree algorithms. Since CPU usage is required to perform any locking request and trie operations lock many more pages than the B-tree operations, limiting the number of CPUs limits trie throughput dramatically. For tries, the CPU becomes the bottleneck, but for B-trees, the disks become the bottleneck. When the number of CPUs is infinite, the disk wait affects the trie operations greatly because operations may often be holding an X-lock on a page while waiting for a disk. Other operations cannot overtake an operation that is waiting for the disk while holding an X-lock.

Trie performance is better than the B-tree performance for a situation with infinite CPUs and a workload of half searches and half appends. In this situation, both algorithms are very similar because they tend to place X-locks only on the leftmost leaves of the data structure. However, since the data capacity for a trie page is greater due to data compression, fewer new pages are created. As a result, far fewer link chases occur for the trie operations and throughput is greater. However, when we impose the limitation of using only 1 CPU, trie throughput becomes substantially less than the B-tree throughput again due to the extensive CPU usage by trie operations.

5.2 Future Work

We identify three potential areas for future work in the study of trie concurrency: new algorithms that improve on the concurrency provided by our algorithms, modification of the pointerless trie data structure to enable better throughput, and creation of an algorithm for concurrent deletions of keys from a trie.

New Algorithms for Concurrent Trie Searching and Inserting

The performance of concurrent searches and inserts in a pointerless trie must be improved. Our initial algorithms are very restrictive because there are few provisions for any recovery due to interference from other concurrent operations. As a result, these algorithms use lock-coupling, which is a very limiting in terms of throughput performance. There may be modifications to these algorithms that allow for less locking.

Modification of the Pointerless Trie Representation

Modification of our algorithms may not, however, provide a great improvement in concurrency for trie operations. There may be some modification to the pointerless trie structure required before any significant improvements in throughput can be made. Such a modification may be simply to include a pointer from a trie page to its leftmost child page. Perhaps such a modification could prevent the need for operations to lock-couple across the entire page level by allowing them to safely bypass all pages that are definitely not going to be navigated. For this modification, page splits would not cause updates to propagate up the trie; however, splitting a parent would introduce new solutions needed to maintain correctness of the trie.

The link pointer created an enormous improvement in B-tree throughput. Perhaps a similar modification to the trie can be made to allow for a less restrictive locking technique. We attempted to use a prefix for each page that would allow an operation to recover from interference caused by other operations. While we were not successful in implementing this idea, there may be a similar approach to increasing trie concurrency.

Concurrent Deletions for Tries

We have not implemented deletions for a trie with concurrent operations. Deletions could be implemented to perform while all other operations wait or concurrently with searches and inserts. Care must be taken, however, to ensure that deletion of key values does not cause concurrent operations to perform incorrectly.

Bibliography

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[Bil85] A. Biliris. A model for the evaluation of concurrency control algorithms on B-trees: Experimental comparisons of four locking protocols. Technical Report 85-015, Computer Science Department, Boston University, 1985.

[Bil87] A. Biliris. Operation specific locking in B-trees. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 159–169, March 1987.

[BM72] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[BS77] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9(1):1–21, 1977.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[dlB59] R. de la Briandais. File searching using variable length keys. In *Proceedings of the Western Joint Computer Conference*, volume 15, pages 295–298, New York, 1959. Spartan Books.

[Ell80] C. S. Ellis. Concurrent search and insertion in 2-3 trees. *Acta Informatica*, 14(1):63–86, 1980.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[Gra78] J. N. Gray. Notes on database operating systems. In *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pages 393–481. Springer-Verlag, 1978.

[GS78] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21. IEEE Computer Society, October 1978.

[Hol71] R. C. Holt. Comments on prevention of system deadlocks. *Communications of the ACM*, 14(1):36–38, January 1971.

[Hol72] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, September 1972.

[JS89] T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 235–246, March 1989.

[JS90] T. Johnson and D. Shasha. A framework for the performance analysis of concurrent B-tree algorithms. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 273–287, April 1990.

[JS93a] T. Johnson and D. Shasha. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *Journal of Computer and System Sciences*, 47(1):45–76, August 1993.

[JS93b] T. Johnson and D. Shasha. The performance of concurrent B-tree algorithms. *ACM Transactions on Database Systems*, 18(1):51–101, March 1993.

[Kes81] Y. Keshet. Concurrency control problems in B^+-tree databases. Master's thesis, The Technion - Israel Institute of Technology, 1981.

[KL80] H. T. Kung and P. L. Lehman. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems*, 5(3):354–382, September 1980.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[KW80a] Y. S. Kwong and D. Wood. Approaches to concurrency in *B*-trees. In *Proceedings of the 9th Symposium on Mathematical Foundations of Computer Science*, volume 88 of *Lecture Notes in Computer Science*, pages 402–413. Springer-Verlag, September 1980.

[KW80b] Y. S. Kwong and D. Wood. Concurrent operations in large ordered indexes. In *Proceedings of the 4th International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, pages 207–222. Springer-Verlag, April 1980.

[KW82] Y. S. Kwong and D. Wood. A new method for concurrency in B-trees. *IEEE Transactions on Software Engineering*, SE-8(3):211–222, May 1982.

[KW88] A. M. Keller and G. Wiederhold. Concurrent use of B-trees with variable-length entries. *SIGMOD Record*, 17(2):89–90, June 1988.

[Liv90] M. Livny. *DeNet User's Guide*. version 1.5, 1990.

[LS86] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *Proceedings of the Fall Joint Computer Conference*, pages 380–389, 1986.

[LY81] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[Mer98] T. H. Merrett. Overview and simple coding guide. Available at URL: <http://www.cs.mcgill.ca/~tim/tries/tries.html>, September 1998.

[Met75] J. K. Metzger. Managing simultaneous operations in large ordered indexes. Report, Technische Universität München, Institut für Informatik, TUM-Math, 1975.

[ML89] C. Mohan and F. Levine. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. Research Report RJ 6846, IBM Almaden Research Center, August 1989.

[ML92] C. Mohan and F. Levine. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, volume 21 of *SIGMOD Record*, pages 371–380, June 1992.

[Mor68] D. R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[MR85] Y. Mond and Y. Raz. Concurrency control in B^+-trees databases using preparatory operations. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 331–334, Stockholm, Sweden, August 1985.

[MS78] R. Miller and L. Snyder. Multiple access to B-trees. In *Proceedings of the Conference on Information Science and Systems*, Johns Hopkins University, Baltimore, MD, March 1978.

[MS94] T. H. Merrett and H. Shang. Zoom tries: A file structure to support spatial zooming. In *Sixth International Symposium on Spatial Data Handling*, volume 2, pages 792–804, 1994.

[Ore82] J. A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4):150–157, June 1982.

[Ore83] J. A. Orenstein. Blocking mechanism used by multidimensional tries. Unpublished Letter, February 1983.

[Par77] J. R. Parr. An access method for concurrently sharing a B-tree index. Technical Report 36, University of Western Ontario, Department of Computer Science, April 1977.

[Sag85] Y. Sagiv. Concurrent operations on B-trees with overtaking. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 28–37, March 1985.

[Sag86] Y. Sagiv. Concurrent operations on B^*-trees with overtaking. *Journal of Computer and System Sciences*, 33(2):275–296, October 1986.

[Sal85] B. Salzberg. Restructuring the Lehman-Yao tree. Technical Report BS-85-21, College of Computer Science, Northeastern University, Boston, Mass., January 1985.

[Sam76] B. Samadi. B-trees in a system with multiple users. *Information Processing Letters*, 5(4):107–112, October 1976.

[SC91a] V. Srinivasan and M. J. Carey. Performance of B-tree concurrency control algorithms. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, volume 20 of *SIGMOD Record*, pages 416–425, May 1991.

[SC91b] V. Srinivasan and M. J. Carey. Performance of B-tree concurrency control algorithms. Technical Report CS-TR-91-999, University of Wisconsin, Madison, February 1991.

[Sha95] H. Shang. *Trie methods for text and spacial data on secondary storage*. PhD thesis, School of Computer Science, McGill University, 1995.

[SZ94] V. W. Setzer and A. Zisman. New concurrency control algorithms for accessing and compacting B-trees. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 238–248, September 1994.

[Toc63] K. D. Tocher. *The Art of Simulation*. English Universities Press, London, 1963.

[Wed74] H. Wedekind. On the selection of access paths in a data base system. In *Proceedings of the IFIP Working Conference on Data Base Management*, pages 385–398, April 1974.

[Wil64] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, June 1964.