VLDB Journal, 2, 361-406 (1993), Fred J. Maryanski, Editor

Performance of B+ Tree Concurrency Control Algorithms

V. Srinivasan and Michael J. Carey

Received April 19, 1991; revised version received, June 13, 1992; accepted January 17, 1993.

Abstract. A number of algorithms have been proposed to access B^+-trees concurrently, but they are not well understood. In this article, we study the performance of various B^+-tree concurrency control algorithms using a detailed simulation model of B^+-tree operations in a centralized DBMS. Our study covers a wide range of data contention situations and resource conditions. In addition, based on the performance of the set of B^+-tree concurrency control algorithms, which includes one new algorithm, we make projections regarding the performance of other algorithms in the literature. Our results indicate that algorithms with updaters that lock-couple using exclusive locks perform poorly as compared to those that permit more optimistic index descents. In particular, the B-link algorithms are seen to provide the most concurrency and the best overall performance. Finally, we demonstrate the need for a highly concurrent long-term lock holding strategy to obtain the full benefits of a highly concurrent algorithm for index operations.

Key Words. Performance, simulation models, B^+-tree structures, resource conditions, workload parameters, lock modes, data contention.

1. Introduction

Database systems frequently use indexes to access data. These systems typically operate at a high level of concurrency and, because any transaction has a high probability of accessing an index, it is necessary to ensure that concurrent access to indexes is not a bottleneck in the system. B^+-trees are the most common dynamic index structures in database systems, thus most earlier work has concentrated on them. (By B^+-tree we mean the variant in which all keys are stored at the leaves, also called B^*-trees [Comer, 1979]). We focus here on B^+-tree concurrency control algorithms. However, many of our results will lend insight into concurrency control for other index structures, also.

V. Srinivasan, Ph.D., is Development Staff Member, IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, CA 95141. Michael J. Carey, Ph.D., is Professor, Computer Sciences Department, University of Wisconsin, Madison, WI 53706.

Under very high transaction rates, contention in heavily used auxiliary data structures like indexes can increase tremendously, necessitating the use of highly concurrent algorithms to manage these structures. Concurrency control techniques that work well for records or data pages, such as two-phase locking (Gray, 1979) are overly restrictive when naively applied to such items as index pages. Special techniques must be employed to prevent indexes and system catalogs from becoming concurrency bottlenecks. A number of algorithms have been proposed to access B^+-trees concurrently (Samadi, 1976; Bayer and Schkolnick, 1977; Miller and Snyder, 1978; Lehman and Yao, 1981; Kwong and Wood, 1982; Kersten and Tebra, 1984; Lausen, 1984; Shasha, 1984, 1985; Goodman and Shasha, 1985; Mond and Raz, 1985; Sagiv, 1985; Biliris, 1987; Mohan and Levine, 1992), but few performance analyses exist that compare these algorithms. Most earlier studies (Bayer and Schkolnick, 1977; Biliris, 1985; Shasha, 1985; Lanin and Shasha, 1986; Johnson and Shasha, 1990) compare only a few algorithms and have been based on simplified assumptions about resource contention and buffer management. Thus, the relative performance of these algorithms in more realistic situations was an open question when work began on this article. An extension of the work by Johnson and Shasha (1990) resulted in a more comprehensive study of B^+-tree concurrency control algorithms (Johnson, 1990). That work was done concurrently with ours, a subset of which appeared in Srinivasan and Carey (1991). We will compare our performance results with those of Johnson (1990) in Section 6.

In this article, we analyze the performance of various B^+-tree concurrency control algorithms using a simulation model of B^+-tree operations in a centralized DBMS. Our study differs from earlier ones in several aspects:

1. We study a representative list of algorithms, including variations of the Bayer-Schkolnick (1977), top-down, and B-link algorithms, as well as a new algorithm that allows deadlock detection at a single node. Based on our analysis of these algorithms, we make further projections about the performance of additional algorithms that have been proposed in the literature.

2. We use a closed queuing model that is quite detailed and consists of a B^+-tree in a centralized DBMS with a buffer manager, lock manager, CPUs and disks. The results presented here should therefore be useful to database system designers for a wide range of systems, including single and multiple processor systems with one or more disks.

3. We consider tree structures with high and low fanouts, a wide range of resource conditions, and workloads which contain various proportions of searches, inserts, deletes, and appends.

4. We measure a wide variety of performance measures like throughput, average response time per operation type, resource utilizations, lock waiting times, buffer hit rates, number of I/Os, frequency of link chases in B-link algorithms,

probability of splitting and merging, frequency of restarts, etc. These measures help us to make precise statements about the performance of searches, inserts, deletes, and appends in the different versions of the algorithms.

Section 2 briefly reviews the set of B^+-tree concurrency algorithms that have been proposed in the literature, focusing on the ones that were chosen for our study. The simulation model and performance metrics that we use in our study are described in Section 3. Section 4 presents details of our experiments and results. In Section 5, we discuss how these results can be used to predict the performance of other protocols. Section 6 compares this study with related work. In Section 7, we discuss the performance effects of locking strategies that are used for serializing transactions with multiple B^+-tree operations. Section 8 presents a discussion of some practical issues that arise regarding the implementation of B-link algorithms in the context of actual database systems. In Section 9, we summarize our key results.

2. B^+-Trees in a Database Environment

An index is a structure that efficiently stores and retrieves information (usually one or more record identifiers) associated with a search key. The index can be either one-to-one (unique) or one-to-many (non-unique). The keys themselves can have fixed or variable lengths. Though we restrict ourselves to unique indexes with fixed length keys, most of our results also apply directly to trees with variable length keys or duplicate keys.

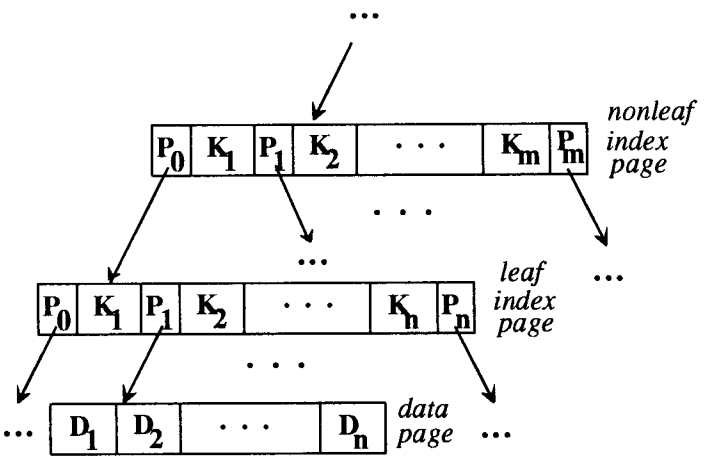
2.1 B^+-Tree Review and Terminology

A B^+-tree index is a page-oriented tree that has the following properties. First, it is a balanced *leaf* search tree (actual keys are present only in the leaf pages, and all paths from the root to a leaf are of the same length). A B^+-tree is said to be of order d if every node has at most $2d$ separators,[1] and every node except for the root has at least d separators. The root has at least two children. The leaves of the tree are at the lowest level of the tree (level 1) and the root is at the highest level. The number of levels in the tree is termed the tree height. A non-leaf node with j separators contains $j + 1$ pointers to children. A <pointer, separator> pair is termed an index entry. Thus, a B^+-tree is a multi-level index with the topmost level being the single root page and the lowest level consisting of the set of leaf pages. Figure 1 summarizes these concepts.

The index is stored on disk, and a search, insert, or delete operation starts by searching the root to find the page at the next lower level that contains the subtree having the search key in its range. The next lower level page is searched, and so

1. A *key* usually implies that associated information for that value exists in the index. A *separator* defines one step in a search path to leaf pages that contain actual keys and associated information.

Figure 1. A B-tree fragment



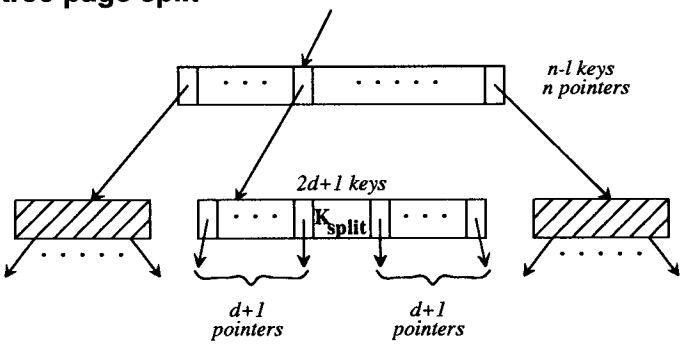
on, until a leaf is reached. The leaf is then searched and the appropriate action is performed. Operations can be unsuccessful; for example, a search may not find the required key.

As keys are inserted or deleted, the tree grows or shrinks in size. When an updater tries to insert into a full leaf page or to delete from a leaf page with d entries, a page split or page merge occurs. A B^+-tree page split is illustrated in Figure 2. B^+-trees in real database systems usually perform page merges only when pages becomes empty; nodes are not actually required to contain at least d entries, because this simplifies implementation and for practical workloads is not found to decrease occupancy by much (Johnson and Shasha, 1989). We employ this approach to B^+-tree merges in this study. A node is considered *safe* for an insert if it is not full, and safe for a delete if it has more than one entry. A split or merge of a leaf node propagates up the tree to the lowest safe node along the path from the root to this leaf. If all nodes from the root to the leaf are unsafe, the tree increases or decreases in height. The set of pages that are modified in an insert or delete operation is called the *scope* of the update.

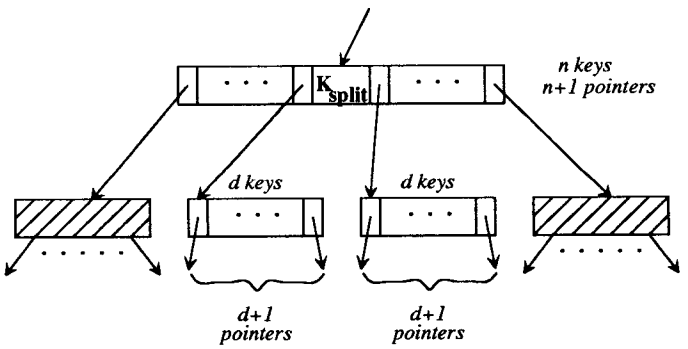
2.2 B^+-Tree Concurrency Control Algorithms

In this article, we use the lock modes IS, IX, SIX, and X. The first three lock types were originally described for use in hierarchical (i.e., multi-granularity) locking protocols (Gray, 1979). Although hierarchical locking is not used by any of the algorithms we discuss, we chose to use the same lock mode names because these lock types and their compatibility relationships are well known. For our purpose, the IS and IX lock modes can both be thought of as shared lock modes; we need two such lock modes because some algorithms distinguish between the shared locks used by readers and writers. The SIX lock mode is used to exclude other writers

Figure 2. B-tree page split



(a) Before split.



(b) After split.

while allowing readers, and the X lock mode is the traditional exclusive lock mode. The lock compatibility relationships are summarized in Table 1.

A "naive" B^+-tree concurrency control algorithm would treat the entire B^+-tree as a single data item and use locks (or latches[2]) on just the root page to prevent conflicts. Readers (searches) would get IS locks on the root, while updaters (inserts or deletes) would get X locks on the root. Locks would be held for the entire duration of an operation. This naive algorithm can be improved by considering every index page as an independently lockable item and making use of the following relationship between a safe node and the scope of an update.

When an updater is at a safe node in the tree, the only pages that can be present in the scope of this update are nodes in the path from this node to the leaf. Any locks held on nodes at higher levels can thus be released. Several algorithms use a technique called *lock-coupling* in their descent from the root to the leaf, releasing

2. Latches (Mohan and Levine, 1992) can be thought of as fast locks. They are also less general than locks (e.g., no deadlock detection is performed for latch waits). In this article, latches can be used wherever locks are used.

Table 1. Lock compatibility

mode	IS	IX	SIX	X
IS	√	√	√	
IX	√	√		
SIX	√			
X				

locks early using the above property. An operation is said to lock-couple when it requests a lock on an index page while already holding a lock on the page's parent, releasing the parent lock only after the child lock is granted.

In a simple algorithm (Samadi, 1976), all operations get an X lock on the root and then lock-couple their way to the leaf using X locks, releasing locks at higher levels whenever a safe node is encountered. This strategy ensures that when an update operation reaches a leaf, it holds X locks on all pages in its scope and no locks on any other index nodes. Updaters and readers whose scopes do not interfere can thus execute concurrently. However, a considerable number of conflicts may be caused at higher level nodes due to the use of X locks. A class of algorithms that improves on the above idea was proposed by Bayer and Schkolnick (1977).

2.2.1 Bayer-Schkolnick Algorithms. In all Bayer-Schkolnick algorithms, searches follow the same locking protocol. In particular, a search gets an IS lock on the root and lock-couples to the leaf using IS locks. The various algorithms differ in the locking strategy used by updaters. We describe three representative algorithms: B-X, B-SIX, and B-OPT.

In the first algorithm, B-X, updaters get an X lock on the root and then lock-couple to the leaf using X locks. With this simple approach, the X locks of updaters on the path from the root to the leaf may temporarily shut off readers from areas of the tree not in the actual scope of an update. This problem can be rectified if updaters lock-couple using SIX locks in their descent to the leaf. This algorithm, called B-SIX, allows readers to proceed faster (because SIX locks are compatible with IS locks), but updaters, on reaching the target leaf, have to convert the SIX locks in their scope to X locks. This top-down conversion drives away any readers in the updater's scope.

In both algorithms above, updaters that do not conflict in their scope may still interfere with each other at higher level nodes. Moreover, in most B^+-trees, especially ones with large page capacities, page splits are rare. The third algorithm, which we call B-OPT, makes use of this fact, letting updaters make an optimistic descent using IX locks. They take an IX lock on the root and then lock-couple their way to the leaf with IX locks, taking an X lock at the leaf. Here, regardless of safety, the lock at each level of the tree is released as soon as the appropriate

child has been locked. If updaters find the leaf to be safe, the operation succeeds. Otherwise, the updater releases its X lock on the unsafe leaf and makes a pessimistic descent using SIX locks, as in the B-SIX algorithm. If very few updaters make a second pass, this algorithm is expected to perform well.

Updaters in the Bayer-Schkolnick algorithms essentially update the entire scope at one time, making it necessary for them to hold several X locks at the same time. Several alternative algorithms have been proposed that instead split the updating of the scope into several smaller, atomic operations. We consider two of these next, the top-down and B-link algorithms.

2.2.2 Top-down Algorithms. In top-down algorithms (Guibas and Sedgewick, 1978; Carey and Thompson, 1984; Mond and Raz, 1985; Lanin and Shasha, 1986) updaters perform what are known as preparatory splits and merges. If an inserter encounters a full node during its descent, it performs a preparatory page split and inserts an appropriate index entry in the parent of the newly split node. Similarly, a deleter merges any node encountered that contains one entry with its sibling during its descent, deleting the appropriate entry from the parent. Leaf level insertion or deletion is similar except that the preparatory operations ensure that a leaf's parent will always be safe. As always, a merge or a split of the root page leads to an increase or decrease in the tree height.

Based on the preparatory operations described above, we consider three top-down algorithms that correspond to the Bayer-Schkolnick algorithms in terms of the type of locking that updaters do. In the first algorithm, TD-X, updaters get an X lock on the root and then lock-couple using X locks to the leaf. At every level, before releasing the lock on the parent, an appropriate merge or split is made. This algorithm can be improved by using SIX locks and converting them to X locks only if a split or merge is actually necessary. This variation is called TD-SIX. In the optimistic top-down algorithm, TD-OPT, updaters make an optimistic first pass, lock-coupling from the root to the leaf using IS locks and then getting an X lock on the leaf. If the leaf is unsafe, the updater releases all locks and then restarts the operation, making a second descent à la TD-SIX (Lanin and Shasha, 1986). Readers use the same locking strategy as in the Bayer-Schkolnick algorithms.

Top-down algorithms break down the updating of a scope into sub-operations that involve nodes at two adjacent levels of the tree. The B-link algorithms go one step further and limit each sub-operation to nodes at a single tree level. They also differ from the top-down algorithms in that they do their updates in a bottom-up manner.

2.2.3 B-link Tree Algorithms. A B-link tree (Lehman and Yao, 1981; Sagiv, 1985) is a modification of the B^+-tree that uses links to chain all nodes at each level together. A page in a B-link tree contains a high key (the highest key of the subtree rooted at this page) and a link to the right sibling. The link enables a page split to

occur in two phases: a *half-split*[3] followed by the insertion of an index entry into the appropriate parent. After a half-split, and before the <pointer, separator> pair corresponding to the new page has been inserted into the parent page, the new page is reachable through the right link of the old page. A B-link tree node and an example page split are illustrated in Figure 3. Operations arriving at a newly split node with a search key greater than the high key use the right link to get to the appropriate page. Such a sideways traversal is termed a *link-chase*. Merges can also be done in two steps (Lanin and Shasha, 1986), via a half-merge followed by an entry deletion at the next higher tree level. The proposed B-link algorithms (Lehman and Yao, 1981; Sagiv, 1985) differ from the Bayer-Schkolnick and top-down algorithms in that neither readers nor updaters lock-couple on their way down to a leaf. We study three variations of the B-link algorithms: LY, LY-LC and LY-ABUF.

In the LY (Lehman and Yao, 1981), a reader descends the tree from the root to a leaf using IS locks. At each page, the next page to be searched can either be a child or the right sibling of the current page. Here, readers release their lock on a page *before* getting a lock on the next page. Updaters behave like readers until they reach the appropriate leaf node. On reaching the appropriate leaf node, updaters release their IS lock on the leaf and then try to get an X lock on the same leaf. After the X lock on the leaf is granted, they may find either that the leaf is the correct one to update or that they have to perform one or more link-chases to get to the correct leaf. Updaters use X locks while performing all further link chases, releasing the X lock on a page before asking for the next. If a page split or merge is necessary, updaters perform a half-split or half-merge. They then release the leaf lock *before* they search for the parent node, starting from the last node (at the next higher level) that they used in their descent. That is, updaters that are propagating splits and merges use X locks at higher levels and do not lock-couple. In the basic LY algorithm, operations therefore lock a maximum of one node at a time.

Due to the early lock releasing strategy in the basic LY algorithm, updaters that propagate index entries after completing half-splits or half-merges can encounter "inconsistent" situations; e.g., a deleter at a higher level may find that the key to be deleted does not exist there (yet), and an inserter at a higher level may find that the key it is trying to insert already (still) exists. One solution is to force updaters to restart repeatedly until they succeed (Lanin and Shasha, 1986). Our solution is different, however, and we modify the LY algorithm to hold locks more conservatively, thus eliminating inconsistent situations altogether. The modified algorithm is called the LY-LC algorithm, in which updaters hold an IS lock

3. A half-split involves the following steps: First, a new node is allocated. Then, a splitting-key is chosen and entries are moved from the old node to the new node. The transferred entries are then deleted from the old node and the new node is included in the chain for that level. Finally, the new entry (that triggered the page split) is added to either the old node or the new node depending on the value of the splitting-key chosen (Lehman and Yao, 1981).

on a newly split or merged node while acquiring an X lock on the appropriate parent node (in essence, lock-coupling on the way up). That is, the LY-LC algorithm differs from the LY algorithm in that updaters release their lock on a node that is half-split or half-merged only *after* getting an X lock on its current parent.

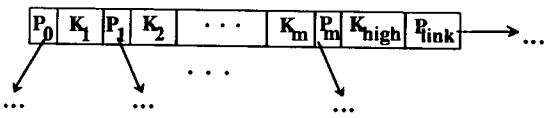
In the B-link algorithm as it was first proposed (Lehman and Yao, 1981), readers did not use locks at all. Instead, they relied on the atomic nature of disk I/Os and used their own consistent copies of pages. To account for the impact of such an approach on buffer hits, we modified this original algorithm to use a buffer manager that supports such an atomic read-write model. Because readers do not lock pages, they instead get a read-only copy of the most recent version of a page. Updaters in their first descent to the leaf behave just like readers, using read-only copies with no locking. However, updaters do have to acquire an X lock on a page before requesting a writable copy of the page. Finally, whenever an updater frees a writable copy, this copy is made the current version of the page and future page requests get a copy of this new version. We implemented an algorithm based on the above atomic buffer model called LY-ABUF.

2.2.4 A New Optimistic Descent Algorithm. Updaters in the optimistic descent algorithms (TD-OPT and B-OPT) restart operations if they encounter a full leaf node, rather than restarting them due to actual conflicts with other updaters. In a new optimistic algorithm that we designed, called OPT-DLOCK, restarts depend solely on deadlock-inducing lock conflicts. OPT-DLOCK detects such conflicts by watching for circular waits of lock upgrade requests for the same index page. A comparison of the performance of this algorithm with that of the other optimistic algorithms provides interesting insights on the efficacy of the two restart strategies under various system and workload conditions.

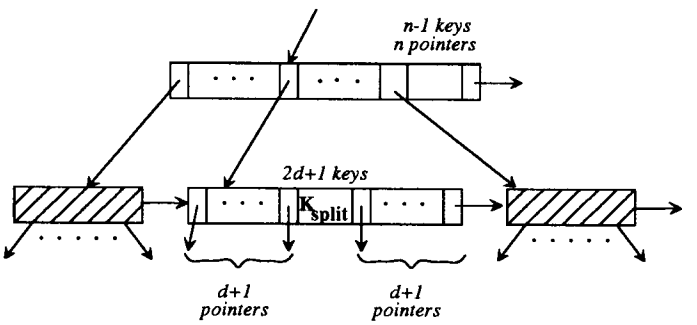
In the OPT-DLOCK algorithm, readers follow the same locking strategy as in the Bayer-Schkolnick and top down algorithms. Updaters descend using IS locks, keeping their scope locked until a safe node is reached; they then take an X lock on the leaf. (Recall that updaters in the algorithms B-X and TD-X execute similarly, but use X locks at all levels.) In OPT-DLOCK, however, a node is considered safe only if it is both insertion safe *and* deletion safe. If the leaf is safe, the update is performed and all locks are released. An updater that reaches an unsafe leaf node will have at least all of the nodes in its scope (and possibly more, due to the new definition of a safe node) locked with IS locks, having the leaf itself locked with an X lock in addition. Updaters reaching an unsafe leaf node release the leaf lock and then try to convert the IS lock on the topmost node of their scope to an X lock. If this lock is granted, updaters proceed to drive away readers by getting X locks on the other nodes in their scope before performing the actual update.

The new definition of safe node used in the OPT-DLOCK algorithm ensures that two updaters whose scopes intersect will always have the same top level safe node. Thus, two updaters with the same top level safe node will both try to convert their IS locks on that node to X locks, creating a local deadlock at that node.

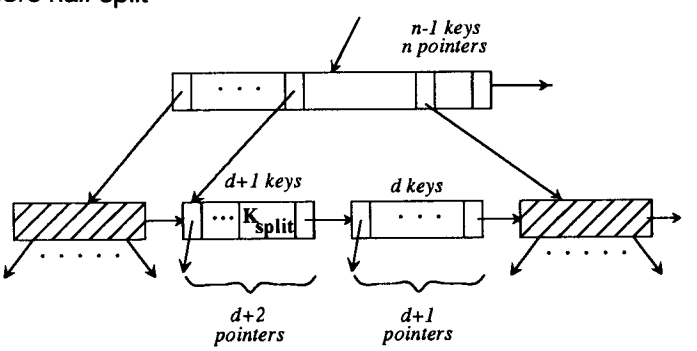
Figure 3. A B-link tree page split



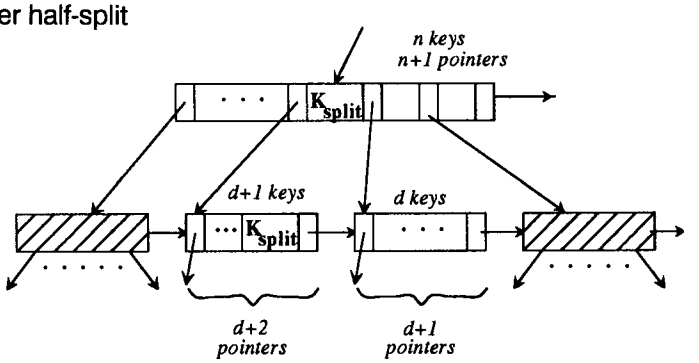
(a) Example B-link tree node



(b) Before half-split



(c) After half-split



(d) After key propagation

Only one of them will succeed, while the other will be restarted after releasing all locks associated with the failed B^+-tree operation. A restarted updater[4] repeatedly tries the protocol until it succeeds; starvation is avoided by assigning priorities to operations based on their first start time. Apart from the algorithms described above, several other B^+-tree concurrency control algorithms have been proposed as well (Kwong and Wood, 1982; Biliris, 1987; Mohan and Levine, 1992). We discuss these other algorithms in Section 5.

3. Simulation Model

Our model is a closed-queuing model with a varying number of terminals and a zero think time between the completion of one operation submitted by a given terminal and the submission of the next one. Operations in this study are "tree operations," each performing a single B^+-tree operation (search, insert, or delete). There are three main components of the simulation model: the system model, which models the behavior and resources of the database system; a workload model, which models the mix of operations in the system's workload; and a B^+-tree model, which characterizes the structure of the B^+-tree. Apart from these, there is the actual concurrency control algorithm that is being executed.

Rather than using a closed-system model, as we have done, it also would have been possible to model the DBMS as an open system. The use of an open system can make mathematical analysis tractable, and some analytical studies have used an open-system model for this purpose (Bayer and Schkolnick, 1977; Johnson, 1990; Johnson and Shasha, 1990). Our choice of a closed-system model enables us to precisely control the multi-programming level in the system, making it easier to control carefully both the resource and data contention levels in the system. Coupled with our detailed simulation model, this enables us to study the causes for the underlying performance effects in greater detail than was possible in earlier studies. However, the choice of an open or a closed system is unlikely to alter the qualitative performance results. This is borne out by the fact that the overall performance results in our study generally agree with the results of studies that have used open-system models (Johnson, 1990; Johnson and Shasha, 1990).

3.1 System Model

The system model is intended to encapsulate the resources present in a database system and the major aspects of the flow of operations in the system. The system model is a shared memory multiprocessor with multiple disks. All commonly accessed data structures like the buffer pool and the lock table are assumed to be in shared memory.

---

4. Note that a restart involves re-trying the B^+-tree operation and is *not* a transaction abort.

The system on which tree operations operate is modeled using the DeNet simulation language (Livny, 1990). The system can have one or more CPUs and disks, modeled using a module called the *resource manager*. A CPU resource can be used when a concurrency control request, a buffer page request, or a B^+-tree page is processed. Requests for the CPUs are scheduled using a first-come first-served (FCFS) discipline with no preemption. A common queue of pending requests is maintained and, when a CPU becomes free, the first request in the queue is assigned to it. The disk resource is used when a B^+-tree page is read into or written out of the buffer pool. Each of the disks has its own disk queue. These queues are also managed in an FCFS fashion.[5] When a new I/O request is made, the disk for servicing the request is chosen randomly from among all disks (i.e., we assume uniform disk use). Each I/O request is modeled as having three components: a seek to a randomly chosen cylinder from the current position of the disk head, a randomly chosen time between the minimum and maximum rotational latencies, and a fixed page transfer time.[6] The random values used to model seek times and rotational latencies are chosen from a uniform distribution.[7]

The physical resource model also includes a buffer pool for holding B^+-tree nodes in main memory. The behavior of the buffer manager is captured via a DeNet module called the *buffer manager*. The buffer pool is managed in a global LRU fashion. Operations *fix* each page in the buffer pool prior to processing it, and they *unfix* each page when they are done processing it and no longer need it to be memory-resident; pages are placed on the LRU stack at the point when they are unfixed. The buffer performs demand-driven writes. (Fixing a page therefore may involve up to two I/Os, one to write out a dirty page and another to read in the requested page.) Apart from the buffer manager, there is a module called the *lock manager* that models the acquisition and releasing of locks.

3.2 Operation Flow

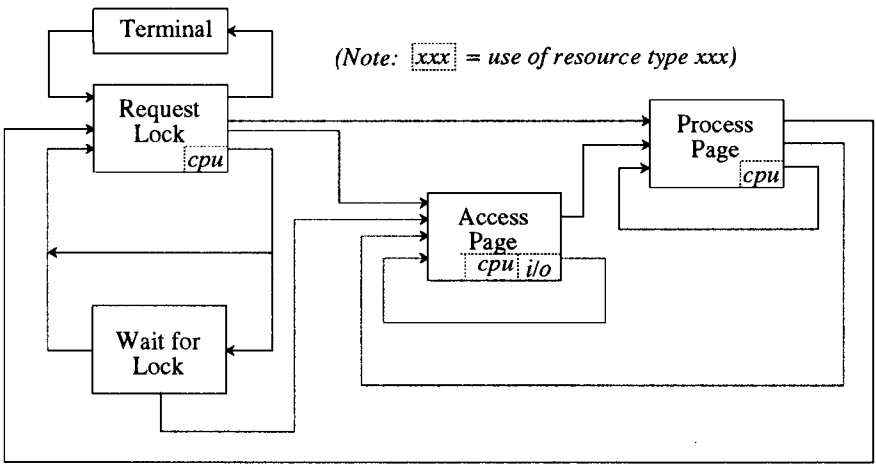
Figure 4 shows the various states of a tree operation in the system. Once a terminal submits an operation, we say that the operation is active. An active operation is

5. We also ran experiments with an elevator disk scheduling algorithm. Results are described later.

6. We have not explicitly modeled the CPU cost for initiating an I/O. While such a cost does exist in real systems, it is important mainly in situations where many I/Os are being initiated. Modeling this cost therefore would not significantly impact the results of our study, as it would amount to a significant cost component only in our disk-bound experiments, where CPU cost is not the performance-limiting factor.

7. Because our model includes a buffer pool that keeps the most frequently used pages (e.g., the higher levels of the B^+-tree) in memory, we did not model the detailed placement of each individual B^+-tree page on disk; accesses to less frequently used (e.g., B^+-tree leaf) pages are essentially random, so buffer misses are adequately modeled by random disk I/Os.

Figure 4. Operation states



always in one of four states. The first state, the "request lock" state, is entered when it needs to set a lock, to convert the mode of an existing lock to a different lock mode, or to release a lock. A concurrency control CPU cost is associated with processing such a request. The second state, "wait for lock," is entered when an operation requests a lock that is already held by another operation in a conflicting mode. Operations waiting for locks on a B^+-tree node are queued in order of arrival, and waiting operations are awakened when an operation holding a conflicting lock releases it. An operation locks and unlocks index pages according to the locking strategy of a particular concurrency control algorithm. The third state, "access page," is entered when an operation wants to fix or unfix a B^+-tree page. In this state, in the event of a buffer pool miss, an operation will either do the I/O itself or wait for an already pending I/O for the page to complete. The time associated with a page access consists of two components: the CPU cost in the buffer manager, and the time for any disk I/O. The fourth possible state for an active operation is the "process page" state shown in Figure 4. This state models the processing of a B^+-tree node (e.g., search, insert, delete, split, merge), and it has a CPU cost associated with it that depends on the type of operation being performed. The particular path that a given operation follows through these four states therefore depends on the type of operation that it performs, the locking protocol employed, the size of the pool of pages for buffering B^+-tree nodes, and the degree of lock conflicts experienced by the operation during its execution.

3.3 Workload Model

One component of the workload is the number of terminals in the system, referred to as the multi-programming level (MPL) of the system. In our experiments, we use the MPL as the parameter to control the actual data and resource contention

levels in the system. A given terminal can submit any one of four types of B^+-tree operations (search, insert, delete, or append). Another component of the workload is therefore a set of probabilities that define the proportion of searches, inserts, deletes, and appends in the workload. A terminal submits operations one at a time. As soon as an operation completes, it returns to the terminal. The terminal immediately generates another operation whose type is randomly determined using the set of probabilities given for the workload.

In generating keys for our workload model, we wanted to achieve the following goals:

1. In our experiments, we wished to keep the composition of the tree in terms of the height of the tree and the occupancy of the nodes relatively unchanged throughout the experiment, so that all of the operations in an experiment see a tree with similar access characteristics.

2. We also wanted all operations (except appends) to uniformly access all portions of the initial tree, so that no "hot spots" are formed that might skew the performance results.

3. We further wished to avoid unsuccessful inserts and deletes whose overhead could vary widely from their successful counterparts (thus causing spurious performance differences).

To satisfy requirement 1, each of our experiments was started with a pre-built initial tree that was later operated on by the various operations in the manner prescribed by the parameters of that experiment. The initial tree was built using a random permutation of all of the odd keys in the key space that consisted of integer values between 1 and 80,000. To satisfy requirements 2 and 3, the keys for individual searches, inserts and deletes were chosen from the key space 1...80,000 as follows: The inserts add randomly chosen even keys from this key space into the initial tree, so every interval between keys of the initial tree is equally likely to be filled. The deletes remove randomly chosen odd keys from the initial tree, and every key from the initial tree has the same probability of being deleted. Searches search for a key value that is uniformly chosen in the range from 1...80,000.

Because inserts, deletes, and searches uniformly access all portions of the initial tree, we avoided any hot spots in accessing B^+-tree nodes. Furthermore, our choice of even keys for inserts and odd keys for deletes ensured that inserts and deletes were always successful. Finally, the keys for appends were chosen sequentially from 80,001 onwards. Because the initial tree contained only keys whose values were less than 80,000, the appends created very high contention for the right-most nodes of the B^+-tree that contained keys from the uppermost portion of the key space.

3.4 B^+-Tree Model

Operations in our simulation model operate concurrently on the same B^+-tree. The B^+-tree model describes the characteristics of this index. An important parameter of

the B^+-tree is the maximum fanout of a B^+-tree page, the page capacity. This gives the maximum number of <pointer, separator> entries in a page. In our model, the physical size of a B^+-tree page is always the same (in bytes), so a variation in fanout should be viewed as being due to different key sizes. (This means that the page processing times like the page insert time, the page delete time, etc., are independent of the B^+-tree fanout.) For simplicity, all keys (and therefore separators) are the same size, and no duplicates are allowed. Another parameter of the B^+-tree model is the particular locking algorithm in use. The B^+-tree and its locking protocols are both modeled by the *B^+-tree manager* module. There are several versions of the B^+-tree manager, each corresponding to a different locking algorithm.

3.5 Performance Metrics

We used the above system, workload, and B^+-tree models as a platform for studying the performance of the B^+-tree concurrency control algorithms (Section 2). The main performance metric used for the study is the throughput rate for tree operations, expressed in units of tree operations per second. We also monitored several other performance measures, including operation-specific measures like tree operation response times, waiting times for locks at various levels, buffer hit rates, I/O service times, frequency of link chases for B-link algorithms, restarts for optimistic protocols, etc. These measures are used to explain the results seen in the throughput curves and also to compare and contrast protocols that perform similarly in terms of throughput.

In addition to concrete performance measures such as operation throughput, it would be advantageous if the *level of concurrency* provided by the protocols could be somehow characterized. For this study, we have adopted the throughput of the protocols under *infinite resources* (Agrawal et al., 1987; Franaszek and Robinson, 1985; Tay, 1984) as a measure of the level of concurrency that they provide. The resource manager simulates such a condition by replacing the CPU and disk scheduling code with pure time delays. Operations then proceed at a rate limited only by their processing demands and locking delays, so protocols that reduce locking delays (i.e., permit higher concurrency) provide significant performance improvements.

3.6 Range of Experiments and Parameters

In our experiments, three factors were varied: the workload (the percentage of searches, inserts, deletes, and appends), the system (the number of CPUs, disks, and buffers), and the structure of the B^+-tree (the fan-out and the initial number of keys). These factors determine the data and resource contention levels of the system. The simulation parameters for our experiments are listed in Table 2.

The B^+-tree can have one of two fanouts, a high fanout (200 entries/page) or a low fanout (8 entries/page). In all of our experiments, we started with a tree containing 40,000 keys. In the high fanout case, the initial tree is a three-level tree

Table 2. Simulation parameters

num-cpus	Number of CPUs (1..∞)
num-disks	Number of disks (1..∞)
disk-seek-time	Min: 0 msec; Max: 27 msec
cpu-speed	20 MIPS
cc-cpu	CPU cost for a lock or unlock request (100 instructions)
buf-cpu	CPU cost for a buffer call (1000 instructions)
page-search-cpu	CPU cost for a page binary search (50 instructions)
page-modify-cpu	CPU cost for a insert/delete (500 instructions)
page-copy-cpu	CPU cost to copy a page (1000 instructions)
num-init-keys	Number of keys present in the initial tree (40,000)
fanout	Number of index entries per page (200/page, 8/page)
cc-alg	Concurrency control protocol (LY, B-X, TD-SIX, etc.)
num-bufs	Size of the buffer pool (see text)
num-operations	Number of operations in the simulation run (10,000)
mpl	Multiprogramming level (1..300)
search-prob	Proportion of searches (0.0 .. 1.0)
delete-prob	Proportion of inserts (0.0 .. 1.0)
insert-prob	Proportion of inserts (0.0 .. 1.0)
append-prob	Proportion of appends (0.0 .. 1.0)

containing 3 non-leaf index pages and 260 leaf pages. The initial tree in the low fanout case has six levels (seven for the top-down algorithms[8]) with around 1500 non-leaf pages and 7000 leaf pages.

For each of the high and low fanout trees, we consider two buffer pool size settings: one where the tree fits entirely in memory, and the other where the number of buffer pages is less than the number of pages in the B^+-tree. Furthermore, the buffer pool sizes for the low and high fanout tree experiments were chosen so that experiments with high multi-programming levels (200-300) could be run without running out of pinnable buffer pages for active processes. For the high fanout case, the two buffer pool sizes are 200 and 600 pages. A 200 page buffer pool results in around 75% of the tree being in memory, while the 600 page setting leads to an in-memory tree (even if the tree grows in size). The corresponding sizes for the

8. Pre-splitting in top-down algorithms leads to early splits for non-leaf pages; this causes the trees built using these algorithms to sometimes have a greater height than those built using bottom-up strategies. This effect is significant only for low fanouts.

buffer pool in the low fanout tree are 600 pages (7% of the tree in memory) and 12,000 pages (memory-resident tree). In cases where the buffer pool size is smaller than the tree, the system will be disk bound due to the large difference between the per-page CPU and disk service times.

In an actual database system, it is difficult to predict exactly what the operation mix will be. Furthermore, any system is bound to undergo changes in workload from time to time. To capture a wide range of operating conditions, we used four different workloads in our experiments: a search dominant workload (80% searches, 10% deletes and 10% inserts), an update dominant workload (40% inserts, 40% deletes and 20% searches), an insert workload (100% inserts), and an append workload (50% each of searches and appends).

Like the workload, the system resources in our experiments also spanned a wide range of conditions. For the in-memory tree case, we studied three different resource settings: one CPU, eight CPUs, and infinite resources. In this case, the number of disks is immaterial since the tree is always in memory and no I/Os are performed. For the case where the buffer pool size is smaller than the size of the B^+-tree, where the system is disk-bound, we again studied three situations: one CPU and one disk, one CPU and eight disks, and infinite resources.

A system configuration consists of a fixed value for each of the following parameters: the workload, the number of CPUs, the number of disks, the B^+-tree fanout, and the buffer pool size. Using the parameter values described above, there were twelve system configurations possible for each workload. In each configuration, we varied the MPL and conducted one experiment for each concurrency control algorithm. At the start of each experiment, the buffer pool wss initialized with as many B^+-tree pages as would fit; higher level pages were given priority in this initialization. The experiment was stopped after 10,000 operations had completed. Batch probes in the DeNet simulation language were used with the response time metric to generate confidence intervals. For all of the data presented here, the 90% confidence interval is within 2.5% (i.e., $\pm 2.5\%$) of the mean.

4. Performance Results

The relative performance of the various B^+-tree algorithms depends on characteristics like the workload composition, the system resources available, the B^+-tree structure, and the multiprogramming level. We divided the algorithms into four classes and analyzed the performance of these classes. The groupings of algorithms into classes are indicated in Table 3. When presenting data on performance measures, we provide the curve for a representative algorithm of a class rather than reproduce all curves for all algorithms. We reproduce the actual curve for an algorithm only when it differs from the others in its class. The classes SIX-LC and X-LC are referred to collectively as *pessimistic algorithms* in the following discussion. We discuss the results of our experiments in a manner organized by workload.

Table 3. Algorithm classification

Class	*Algorithms*	*How Related*
B-LINK	{LY, LY-ABUF, LY-LC}	B-link algorithms
OPT	{OPT-DLOCK, TD-OPT, B-OPT}	Optimistic descent algorithms
SIX-LC	{B-SIX, TD-SIX}	SIX lock coupling algorithms
X-LC	{B-X, TD-X}	X lock coupling algorithms

4.1 Experiment Set 1: Low Data Contention, Steady State Tree

In our first set of experiments, we used a workload that consisted of 80% searches and 10% each of inserts and deletes. The use of an equal proportion of random inserts and deletes, along with the other workload and tree characteristics (Section 3.3) ensured that a negligible number of splits and merges took place. We refer to a tree with a very low number of merges and splits as a *steady state tree*.[9] The presence of relatively few updaters and a small number of splits creates a low data contention situation, and we used this workload as a filter to eliminate bad algorithms. We shall first discuss experiments conducted on the high fanout tree (200 keys/page), followed by the results for the low fanout tree (8 keys/page). For each subset of experiments, we compare the performance of alternative B^+-tree locking algorithms at various resource levels starting with a single CPU and disk.

4.1.1 High Fanout Tree Experiments. The throughput curves for the case with a single CPU and disk are shown in Figure 5. The buffer pool in this experiment had 200 pages, or about 75% of the B^+-tree size. The throughputs for all the algorithms were only slightly greater at higher MPLs than at an MPL of 1. This is because there was only one CPU and disk, and the disk rapidly became a bottleneck. We found that, for all algorithms, the disk is around 90% utilized at an MPL of 1, thus making some I/O and CPU parallelism possible. The left-over bandwidth is used up when the MPL is increased, and the disk becomes fully utilized by an MPL of 4 for all algorithms. After that, no improvement in throughput is possible.

With low data contention, the only performance difference visible in Figure 5 is that LY-ABUF performs worse than the other B-link algorithms at high MPLs. On further investigation, we found that while the number of I/Os performed by the other algorithms does not change much with the MPL, the number of I/Os in LY-ABUF increases. In fact, at an MPL of 200, LY-ABUF performs about 12% more I/Os than the other algorithms. This is due to the fact that the atomic read-write

9. Note that our definition is different from an alternative definition based on unchanging occupancy (Johnson and Shasha, 1989). In our experiments in this section, the occupancy of the B^+-tree nodes does not change significantly throughout the experiment.

Figure 5. High fanout, single disk

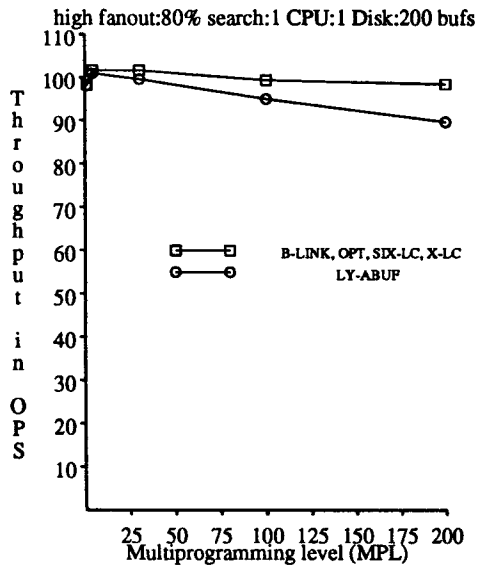
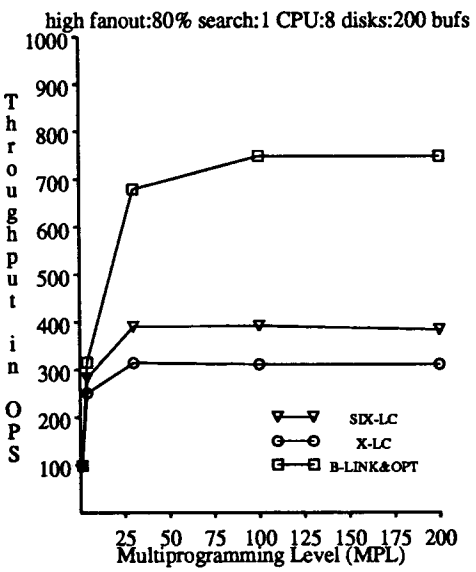


Figure 6. High fanout, 8 disks



model used by LY-ABUF causes multiple copies of pages to exist in the buffer pool, while all other algorithms have just one copy of a page in the buffer pool. These extra copies cause the buffer pool to be used inefficiently in LY-ABUF, especially at higher MPLs, leading to more disk I/Os and extra waiting at the bottlenecked disk. Also, the LY-ABUF algorithm handles the cases when (1) a requested buffer page is not in memory, and (2) the page is in memory but is being modified by another operation, in the same way; it must perform a disk I/O in both cases. In this experiment, case (2) is more likely due to the size of the buffer pool being a sizable fraction of the B^+-tree. Note that this phenomenon should not occur when the buffer pool is much smaller or much larger than the B^+-tree size, and we indeed found that there was essentially no performance difference between LY-ABUF and the other B-link algorithms in those cases. We will drop the LY-ABUF algorithm from future graphs since we found that it never performed better than the other two B-link algorithms.

The throughput curves for the case with 1 CPU and 8 disks is given in Figure 6. Due to the additional system resources, the throughput of all algorithms increases to a higher level before leveling off than in Figure 5. In this case, the SIX-LC and X-LC lock-coupling algorithms only reached a maximum throughput of about half that of the optimistic (OPT) and B-link (B-LINK) algorithms. In trying to explain this, we found that the pessimistic algorithms (SIX-LC and X-LC) have a peak utilization of less than half of the available disk capacity, while the optimistic algorithms use the disk completely at high MPLs. This suggests that the throughputs for the pessimistic algorithms level off due to data contention rather than resource

Figure 7. Insert times 8 disks

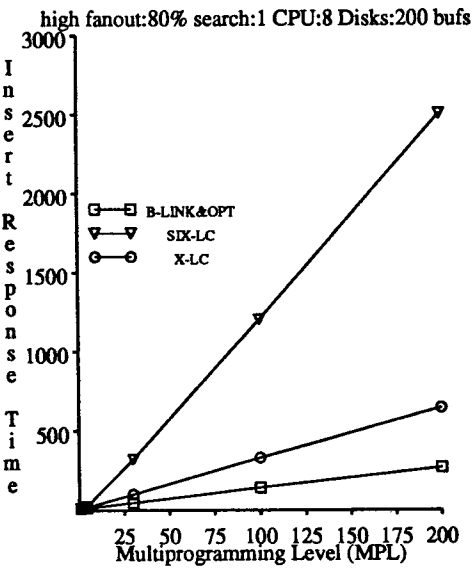
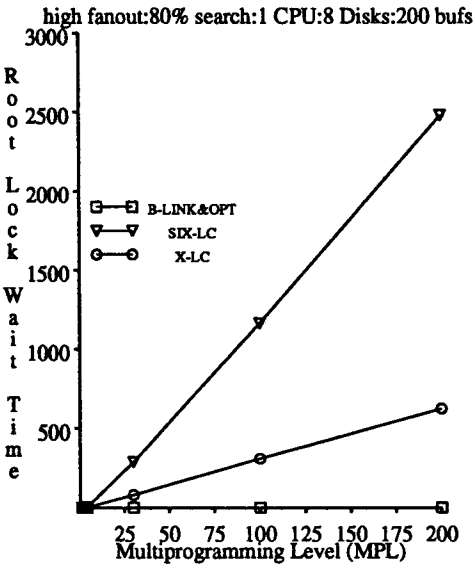


Figure 8. Wait at root (insert), 8 disks



contention. It is evident from the corresponding response time curves (Figure 7) and lock waiting times at the B^+-tree's root page (Figure 8) that the lock waiting time at the root is a significant fraction of the insert operation response time for the X-LC and SIX-LC algorithms; this indicates that locking and searching the root is the bottleneck. In contrast, for the OPT and B-link algorithms, the response time increases at higher MPLs are due only to contention for the disks.

The bottleneck that the X-LC and SIX-LC algorithms form at the root can be understood intuitively by considering a system in which operations have to execute in several stages with the restriction that no two operations can execute the first stage simultaneously (though any number of operations can execute subsequent stages in parallel). Assume that it takes exactly 1 second to run through all stages, and that the first stage takes a fraction k ($0 < k < 1$) of the total time to execute. Now, at an MPL of 1, the throughput will be 1 operation/second. At an MPL of 2, both operations may be phase-shifted and may not collide at the first stage. In that case, they will both have a response time equal to 1. However, the worst case is when both operations arrive at the first stage simultaneously, and one of them has a response time of 1 while the other has $1 + k$. Assuming equal probability for the collision and non-collision cases, we get an average response time of $(1 + k/2)$ and an average throughput of less than 2. At higher and higher MPLs, more and more collisions will occur. In fact, it can be shown that the asymptotic throughput at very high MPLs is $1/k$. Consequently, a bottleneck will form at very high MPLs in front of the first stage.

Searching the root page in the X- and SIX-locking algorithms is analogous to the first stage in the example system, and at high MPLs a bottleneck forms at the root. The bottlenecks are accelerated at higher MPLs due to the lock-coupling overhead of waiting for the lock at the next level. Notice that if k is very small, then bottlenecks will first form at much higher MPLs than if k were large. We indeed noticed that in the memory-resident tree experiments, the bottlenecks formed earlier than in the experiments where the response time includes I/O. This is because the overhead of searching the root (and waiting for a lock at the next level) in the memory-resident experiments is a significant proportion of the actual response time, while in situations that require disk I/Os for non-root nodes, it is a much smaller fraction of the response time.

The bottleneck at the root can affect the response time of operations either symmetrically or asymmetrically. In the X-LC algorithms, the bottleneck affects all types of operations equally; the X-LC search response times in Figure 9 are very close to the corresponding insert response times in Figure 7. On the other hand, in the SIX-LC algorithms, the response time for searches increases only slightly with MPL (Figure 9), while the response time for updaters increases steeply with MPL (Figure 8). This is because, in the SIX-locking algorithms, searches can overtake updaters on their way to a leaf and hence escape the bottleneck at the root. However, the system then gets filled with slower updaters, and the contention levels are much higher than in X-locking (where searches and updaters take approximately equal times to complete). In fact, the increase in response time for updaters is so large that the throughput of the SIX-locking algorithms is only slightly better than that of the X-locking algorithms (Figure 6) in spite of the almost constant search response times of the SIX algorithms. It should be noted that the phenomenon of a bottleneck at the root for pessimistic algorithms has been mentioned in earlier papers (Biliris, 1985; Johnson, 1990; Johnson and Shahsa, 1990); our contribution is to the understanding of how bottlenecks affect the response times of different operation types.

Finally, to get an idea of the extent to which the different algorithms can take advantage of the concurrency available in the workload and the high fanout B^+-tree structure, we present their throughput curves for the case of infinite resources in Figure 10. Note how the pessimistic algorithms level off at around the same maximum throughput as in the 1 CPU and 8 disks case (Figure 6), while the optimistic and B-link algorithms make excellent gains in throughput with increasing MPL. The reason that the optimistic and B-link increases are slightly less than linear is due to contention at the buffer pool (for the buffer pool latch) which results in increased buffer access times at higher MPLs.

In addition to the above experiments, we also performed experiments in which the entire tree is in memory. The only difference between the disk bound experiments described above and the experiments with the memory-resident tree was that the CPU resource became the bottleneck at earlier MPLs than the disks did in the disk-bound experiments, as would be expected. However, the qualitative results

Figure 9. Search times (msec), 8 disks

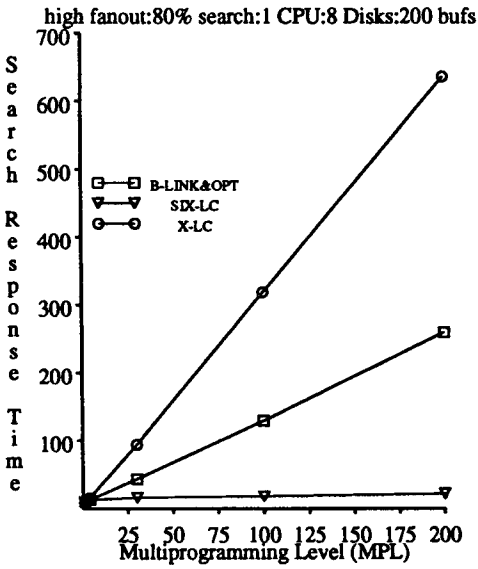
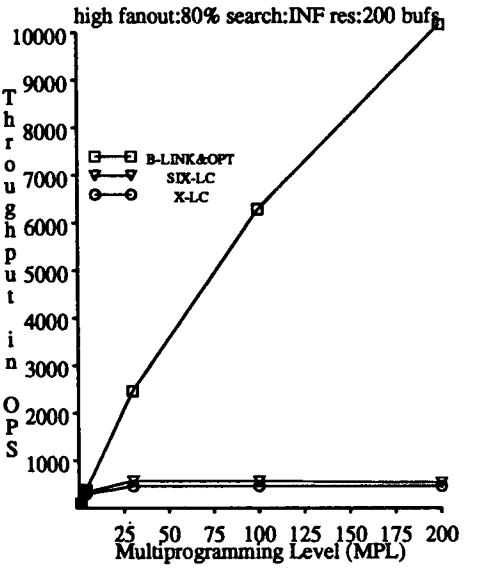


Figure 10. High fanout, ∞ resources



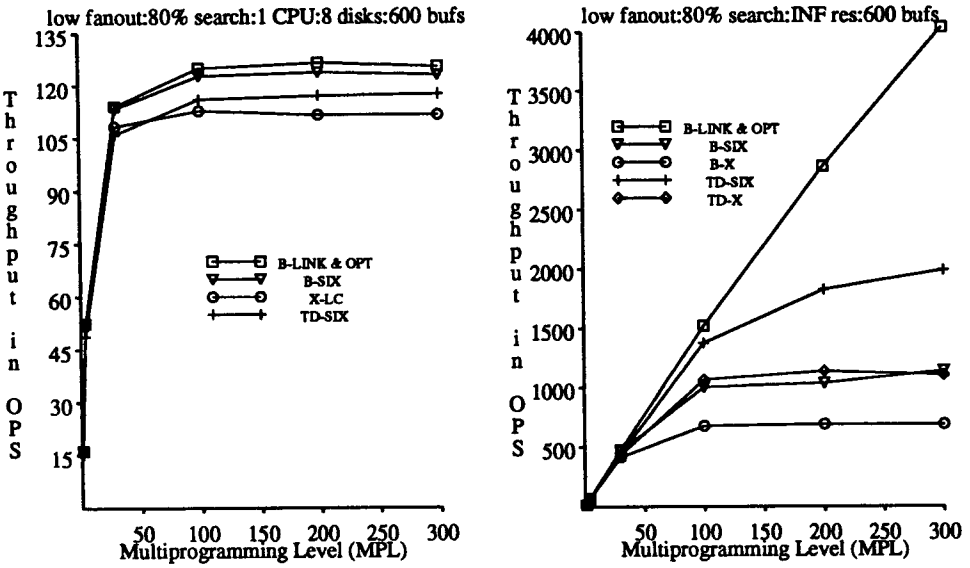
were similar to those for the disk bound case. Thus, we omit these graphs to conserve space.

Notice that in the above experiments, there was no significant performance difference between the top-down algorithms and the corresponding Bayer-Schkolnick algorithms. This is to be expected because the tree has only three levels; the number of exclusive locks held at one time on the scope of an update is hardly different in the two cases (due to the rarity of splits and merges).

4.1.2 Low Fanout Tree Experiments. Recall that the initial B^+-tree index in the low fanout case has 40,000 keys, 7,000 leaf pages and 1,500 non-leaf pages. The tree has six levels (seven in the top-down algorithms, as explained earlier). As in the high fanout experiments described above, the interesting cases are experiments in which the size of the buffer pool is less than the B^+-tree size. Here the buffer pool size is 600 pages, or about 7% of the B^+-tree size. The workload here consists of 80% searches and 10% each of inserts and deletes.

In the single CPU and disk case, there was little difference between the various algorithms, so we omit these results here. The throughput curves for the 1 CPU and 8 disks case is shown in Figure 11, where there is still not much difference in throughput between the various algorithms. In particular, the throughput of the pessimistic algorithms differs little from the throughput of the optimistic and B-link algorithms, unlike in the high fanout tree (Figure 6). This is because the bottleneck

Figure 11. Low fanout, 8 disks **Figure 12. Low fanout, ∞ resources**



at the root forms at higher MPLs here due to the fact that a lesser proportion of the response time is spent searching the root than in the high fanout case described earlier.

An interesting point is that the peak throughput for the top-down algorithms in Figure 11 is somewhat less than that of the corresponding Bayer-Schkolnick algorithms (i.e., the peak throughput of TD-SIX is slightly lower than that of B-SIX). This is because early splitting in the top-down trees results in less occupancy for pages at the non-leaf levels. In the low fanout case, with more than a thousand non-leaf index nodes, this reduced occupancy causes a significant increase in the number of pages in the tree; the result is a reduced hit rate for index pages in the buffer pool. This reduced hit rate translates into more I/Os and, because the disk is a bottleneck in this experiment, the top-down algorithms perform slightly worse. The relatively small difference is due to the fact that the top-down buffer hit rates, though uniformly lower than those for the other algorithms, are still fairly close to the other hit rates because the leaf node hit rate dominates (the number of leaf nodes is approximately four-fifths of the total number of nodes).

Figure 12 shows the throughput curves for the infinite resources, low fanout case with the 600 page buffer pool. Note that the optimistic and B-link algorithms perform much better than the pessimistic lock-coupling algorithms. Among the pessimistic lock-coupling algorithms, the top-down algorithms perform better than the corresponding Bayer-Schkolnick algorithms (i.e., TD-X is better than B-X and TD-SIX is better than B-SIX). These differences are due to the varying amounts of lock waiting at the root, as the lock waiting at all other levels is very small. The

top-down algorithms benefit from having to lock less of the scope in exclusive mode at any one time than the Bayer-Schkolnick algorithms, and therefore perform much better in Figure 12 in spite of a slightly lower hit rate for non-leaf index pages. As before, search operations are favored by the SIX-locking algorithms and, because searches are in the majority, the SIX-locking algorithms outperform the X-locking algorithms.

4.1.3 Summary. In a relatively low data contention situation, except for the single CPU and disk case, the optimistic and B-link algorithms performed much better than the pessimistic lock-coupling algorithms. Even in a system with relatively few resources, the throughputs of the optimistic and B-link algorithms were better than those of the pessimistic ones. This is because the pessimistic algorithms are unable to take advantage of the low data contention of the workload due to their exclusive locking of the root. Using a SIX policy that allows readers to overtake updaters does not alleviate this problem, because even the few updaters that are present take a long time to complete; the overall throughput is therefore not increased much beyond the X-locking situation. The differences between the top-down and Bayer-Schkolnick algorithms were greater in the low fanout case, but the high fanout tree is much more likely to occur in practice. In future graphs, we will represent the performance of the pessimistic algorithms by that of the best lock-coupling algorithm, as they will be seen to consistently perform much worse than the optimistic and B-link algorithms.

To see how a larger percentage of updaters affects the above results, we also experimented with a workload of 20% searches and 40% each of inserts and deletes. Since these results differ only in a few respects from the first set of experiments, we omit the graphs and summarize the key results. The pessimistic algorithms performed even worse for this workload than in the search-dominated workload. For all conditions except the low fanout, infinite resources case, the optimistic algorithms performed quite close to the B-link algorithms. In the low fanout case under infinite resource conditions, the algorithms TD-OPT and B-OPT performed somewhat worse than OPT-DLOCK and the B-link algorithms; this was due to their relatively larger probability of restarts. We characterize this restart behavior in the next set of experiments.

4.2 Experiment Set 2: High Data Contention, Growing Tree

To study further the performance differences between the optimistic algorithms and the B-link algorithms, our next set of experiments uses a workload consisting only of inserts. This 100% insert workload differs from the one used in the first set of experiments in that it creates higher data contention due to the significant number of splits required to accommodate the new keys being inserted. In particular, the nodes at the level above the leaves (i.e., their parent nodes) are modified frequently under this workload.

4.2.1 High Fanout Tree Experiments. The results in this section are based on the subset of the experiments that were performed on high fanout trees. We first consider experiments that were conducted with a buffer pool size of 200 pages, or about 75% of the initial B^+-tree size. We omit the curves for the single CPU and disk case, because there is again not much difference between the algorithms. Figure 13 contains the throughput curves for the 1 CPU and 8 disks case. As in the earlier set of experiments, the pessimistic algorithms (Best LC) again perform much worse than the optimistic and B-link algorithms. In addition, for the first time we see significant differences between the B-link and optimistic algorithms. We comment on three important aspects of these differences:

1. The optimistic algorithms perform worse than the B-link algorithms here because they are only able to use a maximum of 80% of the disk resources due to data contention under this workload. The B-link algorithms, however, are still able to saturate the disks at high MPLs. As with the pessimistic algorithms in the first set of experiments, the algorithms TD-OPT and B-OPT lose their performance here due to lock waiting at the root.

2. The algorithm TD-OPT achieves a peak throughput higher than B-OPT (Figure 13). We found that the lock waiting time for the B-OPT algorithm increases faster than that of TD-OPT, so TD-OPT performs better than B-OPT. Recall that, in both algorithms, inserters make a second pass with SIX locks if they encounter a full node. Because SIX locks are incompatible with each other, two updaters in their second phase interfere with each other if both try to lock the root at the same time. Furthermore, in B-OPT, an inserter in the second pass can also interfere with an inserter in the first pass.[10] This extra interference causes the average waiting time at the root for B-OPT to be greater than that of TD-OPT. Moreover, in the TD-OPT algorithm, inserters in their first pass can overtake those in their second pass. Such overtaking, while leading to less waiting time at the root, could also increase the number of restarts since overtaking will allow more than one operation to reach the same full node. A look at the restart counts for the TD-OPT and B-OPT algorithms (Figure 14) indeed shows that TD-OPT at high MPLs performs about 4 times as many restarts as B-OPT. However, these restarts are inexpensive in this disk bound case, as all pages needed after a restart are most likely in memory.

3. We found that the throughput of OPT-DLOCK increases quickly at low MPLs and then more slowly at high MPLs. Unlike the other optimistic

10. The first pass in TD-OPT is done by lock-coupling with IS locks, while in B-OPT, the initial lock-coupling is done with IX locks. IX locks are compatible with other IX locks but not with SIX locks, while IS locks are compatible with both.

Figure 13. High fanout, 8 disks

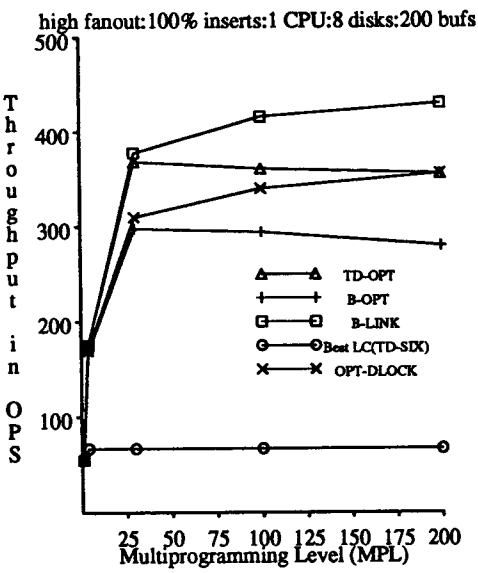
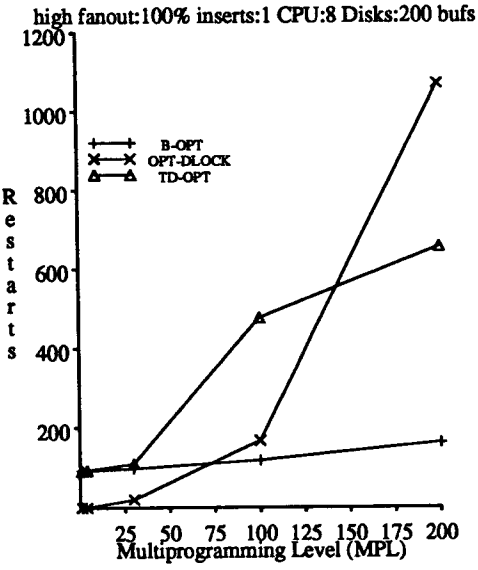


Figure 14. Restarts/10,000 ops



algorithms, however, its throughput does not quickly saturate. The reason for this behavior is the difference in OPT-DLOCK's handling of restarts. Because the conflict level is high, OPT-DLOCK performs many restarts (Figure 14). However, while restarts in TD-OPT and B-OPT result from conflicts at the root node, those in OPT-DLOCK result from conflicts at the level above the leaf. Recall that there are two nodes at this level in the high fanout tree, so conflicts in OPT-DLOCK are divided between the two non-leaf index nodes at this level. Thus, the waiting times for deadlock resolution in OPT-DLOCK increase more slowly than those for B-OPT and TD-OPT, and hence the throughput of OPT-DLOCK increases slightly even at high MPLs. The presence of more nodes at the level of the tree above the leaf would make this algorithm perform even better due to shorter waiting times for deadlock resolution.

In the infinite resources situation (Figure 15), OPT-DLOCK performs better at high MPLs than the other optimistic algorithms because its restarts become more inexpensive due to the increase in the number of CPUs from 1 (Figure 13) to infinity (i.e., due to the lack of resource contention). As before, the B-link algorithms perform much better than all of the other algorithms.

Just to give a flavor of the in-memory tree results for this workload, we reproduce the throughput curves for the in-memory infinite resources case in Figure 16. We find that TD-OPT performs worse than B-OPT here due to the greater number of restarts, because the overhead of a restart is now comparable to the response time

Figure 15. High fanout, ∞ resources

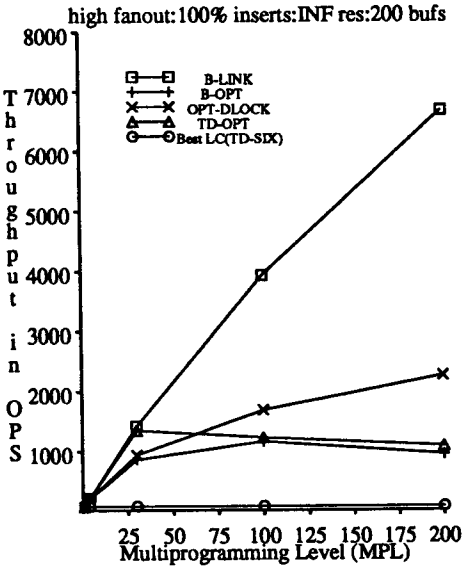
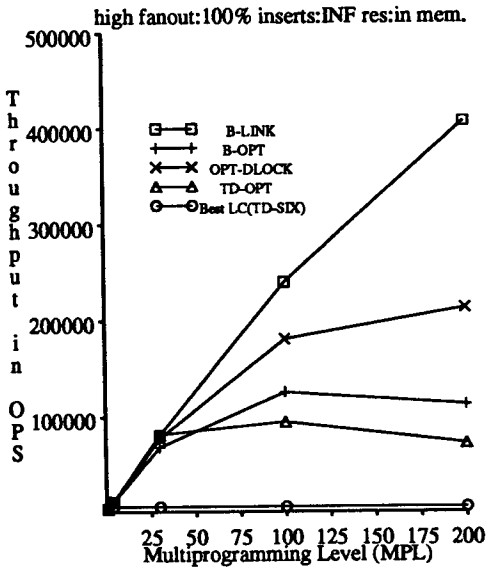


Figure 16. High fanout, in-mem tree



of a successful tree operation itself. The number of restarts here is essentially the same as in the earlier disk bound case (Figure 14).

4.2.2 Low Fanout Tree Experiments. The second subset of experiments using the 100% insert workload was performed on a low fanout tree. As in the low fanout experiments of experiment set 1, we first considered configurations in which the buffer pool contains 600 pages.

The 100% insert throughput curves with 1 CPU and 8 disks are given in Figure 17. As in the high fanout experiments discussed for this workload, TD-OPT and B-OPT perform worse than the B-link algorithms. However, the OPT-DLOCK algorithm performs close to the B-link algorithms at low MPLs and actually does slightly better here at high MPLs. OPT-DLOCK is better than the other optimistic algorithms because the number of restarts in the case of OPT-DLOCK decreases drastically (to just 100 from the maximum of 1200 in Figure 14) due to a reduction in lock conflicts from the earlier high fanout case. On the other hand, due to the increased probability of finding a full leaf page, the restarts for B-OPT and TD-OPT increase to a maximum of around 1600 from the much smaller values (100 and 600, respectively) found in Figure 14. The reason that OPT-DLOCK performs better than the B-link algorithms at high MPLs is that it is able to better utilize the buffer pool than the B-link algorithms. Specifically, the B-link algorithms have to reacquire buffers for propagation of splits, while OPT-DLOCK always keeps them pinned. The B-link algorithms therefore perform more buffer calls, resulting in somewhat more I/Os at high MPLs.

Figure 17. Low fanout, 8 disks

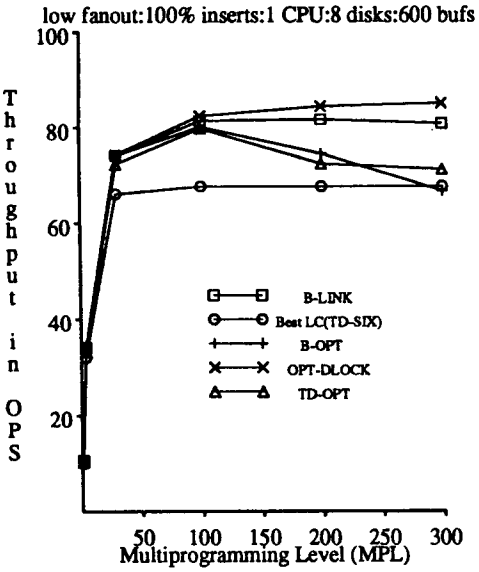
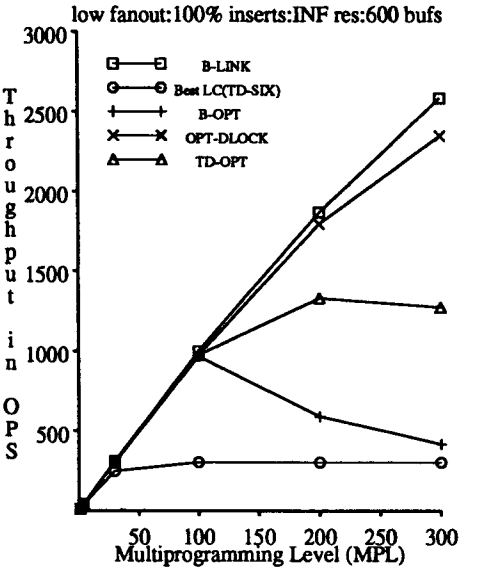


Figure 18: Low fanout, ∞ resources



The results for the infinite resources case (Figure 18) further illustrate the above concepts. The B-link and OPT-DLOCK algorithms keep making gains in throughput, while the throughputs of the B-OPT and TD-OPT algorithms saturate at MPLs of 100 and 200 respectively. At even higher MPLs (not shown in the figure), both the TD-OPT and B-OPT algorithms end up performing close to (but slightly better than) their respective pessimistic versions. This is a surprising result, and the explanation is as follows: The probability that an operation will undergo a restart in the optimistic algorithms is the same at all MPLs, and is equal to the probability of finding a full leaf page. Therefore, at high MPLs, more restarters are active in their second (pessimistic) descent at the same time than at low MPLs. In B-OPT, as discussed earlier, restarts slow down operations in their first descent also, and this causes a bottleneck much like that of the X-LC case (Section 4.1.1).

In the TD-OPT algorithm, the loss in throughput occurs for a slightly different reason. In TD-OPT, only operations in their second pass interfere with each other; operations in the first pass are allowed to overtake those in their second pass. Operations that succeed in their first descent at high MPLs execute much faster than operations that have to restart, much like searches and inserts behaved in the SIX-LC algorithms (Section 4.1.1). As a result, the system eventually becomes filled with updaters in their second pass, which causes a bottleneck so serious that the throughput starts to fall. We verified this by looking at the standard deviation of the insert response times at low and high MPLs for TD-OPT. As expected, we found a significant difference between the standard deviations at low and high MPLs

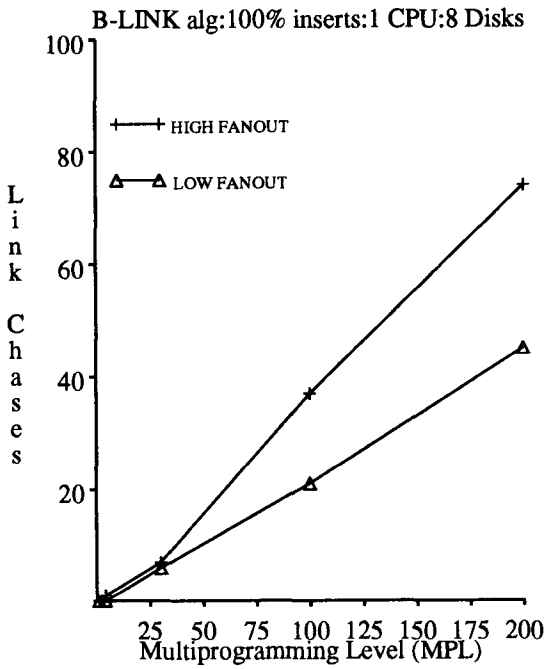
(the standard deviation varied from 50% of the value of the mean at low MPLs up to 120% the value of the mean at high MPLs), indicating that the response times for restarted and non-restarted updaters vary widely at high MPLs. For B-OPT, there was no such variation in standard deviation, as expected. At an MPL of 300, both optimistic algorithms achieve a throughput slightly greater than that of their pessimistic counterparts for exactly the same reason that the SIX-LC algorithms performed slightly better than the X-LC algorithms in the high fanout tree experiment with a search dominant workload (Figure 6); the average of the fast and slow response times for the optimistic algorithms is slightly less than the average for the pessimistic cases.

Just as the optimistic algorithms perform restarts, the B-link algorithms also involve extra overhead in high contention situations due to link-chases. The numbers of link-chases for the B-link algorithms in the 100% insert workload for both high and low fanout trees are presented in Figure 19. (The figures for a particular tree do not differ significantly with the resource level.) Unlike the optimistic algorithms, where the number of restarts varied widely with the fanout of an index node, the number of link-chases is not very different between trees with high and low fanout. In the case of low fanout trees, splits are more frequent, but the probability of an operation visiting a leaf that is being split by an earlier operation is very small due to the presence of thousands of leaf nodes. In trees with high fanouts, even though the probability of an operation visiting a leaf node while it is being split is fairly high, splits are relatively infrequent, thus keeping link-chases down. These two effects seem to balance each other and keep the number of link-chases fairly independent of the fanout. It should also be noted that a link-chase in the B-link algorithms is much less expensive than a restart in the optimistic algorithms.

4.2.3 Summary of Experiment Set 2. In the experiments with a 100% insert workload, the pessimistic algorithms performed worse than the other algorithms, as in Experiment Set 1. The optimistic algorithms TD-OPT and B-OPT performed worse than the B-link algorithms for all system conditions except the single CPU and single disk case. Surprisingly, however, the OPT-DLOCK algorithm performed as well as the B-link algorithms in the low fanout tree, but was worse again for trees with high fanout. Link-chases for the B-link algorithms were too infrequent in both the high and low fanout cases to affect their performance significantly.

So far, we have found the B-link algorithms to be consistently quite superior to the other algorithms in their ability to exploit the concurrency available in the workload and the B^+-tree structure. The only exceptions have been restricted resource situations and situations with a low percentage of updaters, where there is simply very little difference in performance between any of the algorithms. The B-link algorithms therefore appear to be strong candidates for use in a practical system. Because there have been several variations proposed for B-link algorithms, it is interesting to see how these perform among themselves. We already determined that the algorithm LY-ABUF makes inefficient use of the buffer pool, and hence is not a

Figure 19. Link-chases (per 10K ops.)



suitable alternative. Therefore we are interested in performance differences between the algorithms LY and LY-LC, which performed identically in both the first two sets of experiments. Our next set of experiments uses a workload that generates a large amount of localized data contention so as to measure any performance differences between these two B-link algorithms.

4.3 Experiment Set 3: Extremely High Data Contention

In our third and final set of experiments, we used a workload that consisted of 50% appends and 50% searches. Such a workload may arise, for example, given a history index to which appends are being made all the time while searches are being done to find old entries. We used appends to create extremely high contention for the few right-most leaf nodes in the tree. The searches were random, however, and did not interfere with the appends that were taking place. In situations with the buffer space being less than the tree size, the searches serve to keep the system disk bound. In such disk bound situations, only the searches perform I/Os, as the appends always find the pages that they need in the buffer pool. Thus, in this workload we had a situation where two operations with widely varying response times were interacting in the system. In addition, the searches in this experiment simulated the resource contention that is likely to occur between appends and other

types of operations on a B^+-tree.[11]

In our graphs for the set of experiments in this section, we compare LY and LY-LC with the best optimistic algorithm (Best OPT) and the best pessimistic algorithm (Best LC). As usual, we divide the results into the high fanout and low fanout cases.

4.3.1 High Fanout Tree Experiments. The first subset of experiments were for the high fanout tree with a buffer pool size of 200 pages. We found little or no difference in throughput between the algorithms in the single CPU and single disk case, or even in a system with 1 CPU and 8 disks. In both these cases the resource contention at the CPU was the dominating factor, and all the algorithms rapidly attained the same maximum throughput. We therefore omit the graphs for those experiments.

In the infinite resource situation with a 200-page buffer pool, (Figure 20), the B-link algorithms performed better than all of the other algorithms, and their throughputs continued to increase at high MPLs. Notice that there is still not much difference between the LY and LY-LC algorithms, however.

We now switch to experiments where the entire tree is in memory. One of the few cases where a pessimistic algorithm actually performs better than the B-link and optimistic algorithms is the case of a 1 CPU and 1 disk system with the entire tree in memory (Figure 21). The reason is that the very high level of data contention among the appends causes the number of link-chases to increase enormously at high MPLs for the B-link algorithms (Figure 22). Also, notice that the optimistic algorithms perform an increasing number of restarts. Due to the very fast response time of operations on a memory-resident tree, the overheads for a link-chase or a restart are of the same order as the response time of an operation, and the B-link and optimistic algorithms show thrashing behavior at high MPLs. When more CPU resources are available in the system, however, the B-link and optimistic algorithms once again perform better than the pessimistic algorithms.

The B-link algorithm's rate of link-chase *increase* at high MPLs is about half that at lower MPLs (Figure 22). Because searches do not interfere with appends, all link-chases are due to appends. Recall that a link-chase has a high probability of occurring when a split happens. The rate of increase of link-chases reduces at high MPLs due to a randomization of the actual sequence of the appends and a consequent reduction in the number of splits.[12] Split reduction occurs for the optimistic algorithms TD-OPT

11. Unlike appends, which are concentrated in the last few nodes of the tree, the other operations (e.g., insert, delete and search) are likely to access all nodes of the tree uniformly, causing very little data contention between appends and other operations. This is mainly resource contention, which is modeled in the append experiments by the 50% search workload.

12. A strategy that causes appends to be inserted in the increasing order of key values will attain a leaf page occupancy of only 50% for new pages, while a randomization of the appends causes page occupancy to be around 69% (Yao, 1978). Thus, randomization leads to fewer splits.

Figure 20. High fanout, ∞ resources

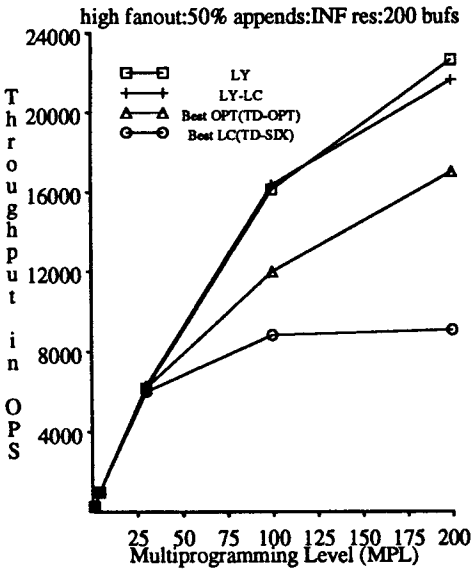
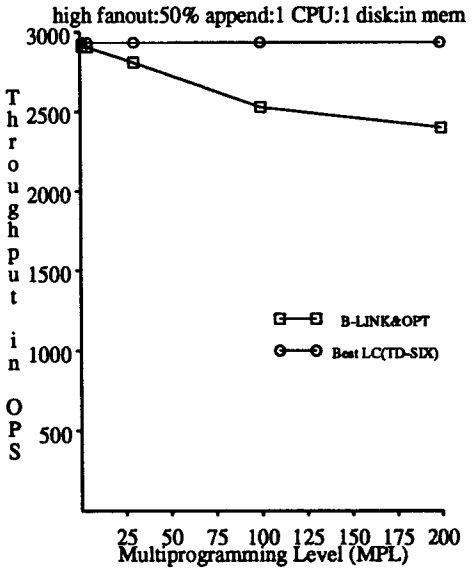


Figure 21. High fanout, in-mem tree



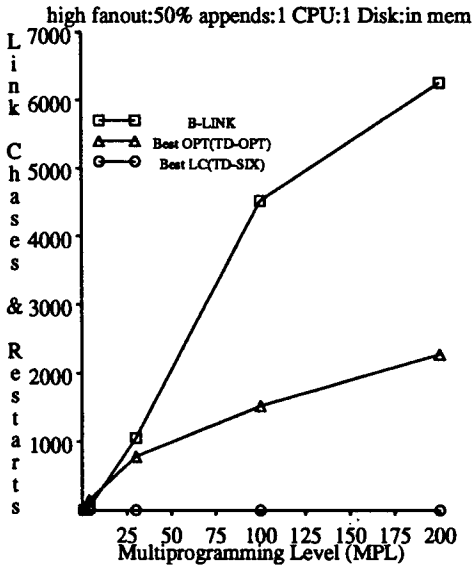
and B-OPT as well, but they are unable to take advantage of the reduction because of the bottleneck that forms at the root. They are also saturated at throughput values close to those of their pessimistic counterparts. OPT-DLOCK exhibited thrashing behavior in the append experiments due to unbounded restarts, but it was still better than TD-OPT and B-OPT throughout.

4.3.2 Low Fanout Tree Experiments. The qualitative results for the low fanout tree are similar to those of the high fanout case, so the graphs are not shown. We found no significant performance differences between the LY and LY-LC algorithms in situations where the buffer pool size was 600 pages (much less than the B^+-tree size). Even in heavily disk bound cases, link-chases did not significantly affect the hit rates of the buffer pool. This suggests that link chasing tends to occur almost immediately after a page is modified and is therefore very inexpensive.

In experiments with the entire tree in memory, we found that the LY-LC algorithm performed slightly worse than the LY algorithm at high MPLs (providing around 20% less throughput). The reason for this is the lock waiting introduced by holding IS locks while searching for the parent of a leaf node. Still, both the LY and LY-LC algorithms performed better than the rest of the algorithms.

4.3.3 Summary of Experiment Set 3. Even in this very high contention workload, the B-link algorithms increased in throughput except in low resource situations. For the B-link and optimistic algorithms, a large MPL randomizes an append workload and reduces the number of splits. The B-link algorithms effectively use the reduced

Figure 22. Overhead (per 10,000 ops)



number of splits to lower the rate of increase in link-chases at high MPLs. Moreover, link-chases are extremely cheap and cause no bottlenecks to form, unlike restarts that cause either a bottleneck at the root (TD-OPT and B-OPT) or thrashing behavior (OPT-DLOCK). There was no significant difference between the LY and LY-LC algorithms across a wide range of resource conditions and tree structures; the exception was a system configuration with a low fanout tree, infinite resources, and a buffer pool as large as the tree, where the LY-LC algorithm performed slightly worse than the LY algorithm at high MPLs. This sort of situation can likely be ruled out in practice, and even if it occurs, the LY-LC algorithm performs reasonably close to the LY algorithm and much better than all non-B-link algorithms.

5. Discussion of Performance Results

We can broadly summarize the results of the previous section into the following points.

1. In a system with a single CPU and disk, there is no significant performance difference between the various algorithms.

2. Lock-coupling with exclusive locks is generally bad for performance. Even for workloads dominated by searches, algorithms in which updaters use such a lock-coupling strategy cannot take full advantage of even the small amount of parallelism available in systems with a few CPUs and disks.

3. Optimistic algorithms that restart on encountering a full leaf node attain only a limited amount of performance improvement over their pessimistic counterparts. In fact, the algorithms TD-OPT and B-OPT perform close to their corresponding pessimistic algorithms at high MPLs. Because the probability of a restart is fixed by the tree structure, the number of restarts interfering with each other increases at higher MPLs–irrespective of any actual data contention. Therefore, these algorithms cannot adapt to dynamically changing workload conditions. A naive algorithm like OPT-DLOCK, in which restarts are based on actual lock conflicts, is much better in most situations than algorithms that restart based on leaf node occupancy.

4. The extent to which an overhead (e.g., a restart or a link-chase) directly affects performance depends more on the number of conflicts that it creates than on the extra resources used. For example, in the append experiments, the overhead for the link-chases in the B-link algorithms was comparable to that of the restarts in the optimistic algorithms. However, the B-link algorithms continued to increase in throughput, given enough resources, while the optimistic algorithms saturated at a throughput level close to that of their pessimistic counterparts. This is because the restarts in the optimistic algorithms caused a concurrency bottleneck at the root (TD-OPT and B-OPT) or at the level above the leaf (OPT-DLOCK), while the link-chases of the B-link algorithms were spread over many different leaf nodes and did not cripple performance.

5. The fanout of index pages can greatly affect performance, as one would expect. For example, the performance of the OPT-DLOCK algorithm relative to the B-link algorithms in high and low fanout situations varies quite widely (Figures 15 and 18).

The only algorithms that performed consistently well are the B-link algorithms. In all of the workloads examined, the B-link algorithms were able to make use of extra resources and increase their throughput. They treat searches and inserts symmetrically, unlike the SIX algorithms, which speed up search response times at a heavy cost to inserts. Moreover, the overhead in the B-link algorithms due to page splits can actually decrease at high MPLs in high contention situations. A final important observation is that the B-link algorithms performed best in both high and low fanout trees. This leads us to conclude that B-link algorithms also will perform well for trees with variable length keys, in which the fanout may vary widely from node to node.

In addition to the above experiments, which used a FCFS scheduling algorithm at the disk, we also performed experiments in which disk requests were scheduled using an elevator algorithm. The elevator algorithm differs from FCFS disk scheduling by significantly reducing the average seek time for disk requests when there are many concurrent requests. We found that this change in seek times affected the results

of the disk bound experiments quantitatively but not qualitatively. The reason is as follows: At higher MPLs, algorithms that do not bottleneck at the root are able to queue multiple requests at the disk at the same time, thus attaining better throughput than in the FCFS case due to the reduction in response time caused by faster seeks. Algorithms that do bottleneck at the root allow much less concurrent operation, and hence their disk seek times do not change much from the FCFS case; they attain the same throughput as before. Thus, the differences in throughput between the bottlenecked algorithms and the other algorithms increases, but the qualitative results are the same as before. For example, in a low contention workload on the high fanout tree with a system configuration of 200 buffers, 1 CPU, and 8 disks using the elevator scheduling algorithm, the optimistic and B-link algorithms reached a peak throughput that was 70% higher than in Figure 6 (while the throughput of the pessimistic algorithms was close to that in the corresponding FCFS case).

Even though our experiments were run for a certain tree size (in terms of the number of keys), we believe that our results will scale up to trees of larger sizes. To understand our results, consider not just the size of the tree (in terms of the number of keys), but also consider the *system configuration* as well as the fanout of the B^+-tree nodes. The system configuration can fall into one of three overall categories: (1) when the entire tree is in memory, (2) when most of the tree is in memory, and (3) when most of the tree is on disk. Using our results for three system configurations for trees with small and large fanout, it should be possible to answer questions about relative algorithm performance given a tree and the system configuration.

In addition, based on the above observations and the results of the previous section, we can also comment on the performance of algorithms that have not been explicitly simulated in our system relative to the performance of the B-link algorithms.

5.1 Side-Branching Technique

The *side-branching* solution proposed in Kwong and Wood (1982) is a modification to the SIX locking Bayer-Schkolnick algorithm, B-SIX. Updaters in this algorithm allocate new pages and copy keys from old pages to new ones while holding SIX locks on the scope of the update. After making changes on the side, the updaters then make a quick pass down the scope with X locks and patch up the nodes. This strategy aims to minimize the time during which X locks are held on nodes in order to speed up searches relative to the B-SIX algorithm. However, in our low data contention experiments we found that there are so few updaters that are not waiting at the root at high MPLs that searches basically have a free run of the B^+-tree. The side-branching technique, which endeavors to minimize interference between updaters and searches, is thus unlikely to have much of an effect on performance because, due to updaters experiencing a bottleneck at the root, there is already little interference (as evidenced by the almost constant search response times in Figure 9 for the SIX-LC algorithm class).

5.2 The mU Protocol

An interesting algorithm called the mU protocol was proposed by Biliris (1985). An important feature of this algorithm is that the compatibility graph for locks on a node is dependent on the occupancy of the node. The algorithm uses special insert and delete lock modes (distinct from IS and X locks) to reserve slots in a node for later insertions or deletions. (The exclusive lock mode used by pessimistic algorithms can be thought of as locking all slots.) The maximum number of insert locks that can be held on a node at any one time is equal to the number of empty <pointer, separator> slots in the node; similarly, the maximum number of delete locks on a node is equal to the occupancy of the node. Insert and delete lock modes are incompatible with each other. This algorithm uses high keys and right links like the B-link algorithms, and in addition it maintains a low key and a left link in all nodes.

There are workloads and system conditions under which this algorithm is likely to perform as well as the B-link algorithms, but there are others where it will surely perform worse. The mU algorithm is sensitive to the occupancy of the index pages–in particular that of the root page–because the number of simultaneous updaters that can be reading a page is limited by the number of empty or full slots in the node. The mU algorithm is therefore likely to perform badly for trees with mostly full or mostly empty nodes as well as for trees with a small number of keys per page. We have seen in our experiments that the probability of finding full pages in a low fanout tree can be quite high, and in such cases the mU protocol will perform worse than the B-link algorithms.

Apart from the above problem with low fanout trees, inserts and deletes interfere with each other at the root in the mU protocol because their respective lock modes are incompatible. In a workload with an equal proportion of inserts and deletes, we can therefore expect this interference to cause a loss of throughput at sufficiently high MPLs due to a bottleneck forming at the root, much like in the TD-OPT and B-OPT algorithms.

5.3 ARIES/IM Algorithm

An algorithm for high-concurrency index management was described by Mohan and Levine (1989, 1992). We will not describe the details of this algorithm, except to state that it does not suffer from any of the performance drawbacks that are present in non-B-link algorithms and discussed at the beginning of this section.

ARIES/IM has both left and right pointers linking nodes at the leaf level, but, unlike the B-link algorithm, the nodes at higher levels do not have right links. Updaters in ARIES/IM make an initial descent to the leaf using IS locks, and at the leaf level they may perform link-chases just as in the B-link algorithms. However, while the B-link algorithms perform link-chases at all levels, updaters in ARIES/IM instead use a complex protocol based on recursive restarts. These restarts do not have the disadvantage of creating any bottlenecks, however, as the operations use

extra information stored in the B^+-tree nodes to ensure consistency rather than using exclusive locks. We observed that most of the link-chases in the B-link algorithms were at the leaf level (100% in the high fanout case and over 90% in the low fanout case). Because the leaf level ARIES/IM algorithm resembles the B-link algorithm in many respects, we can expect the ARIES/IM algorithm to perform close to the LY algorithm for most workloads (including the append workload).

An important difference between the ARIES/IM algorithm and the other algorithms discussed so far is that ARIES/IM allows only one page split (or merge) at a time. To find out exactly what impact this has on performance, we modified the LY algorithm to limit itself to one page split at a time (using a tree latch) and ran experiments with the 100% insert workload in infinite resource conditions. In trees with a high fanout, we found that the modified algorithm performed slightly worse than the B-link algorithms; the throughput of the modified algorithm was in between that of the B-link and OPT-DLOCK algorithms in Figure 15 and, at an MPL of 200, the throughput of the modified algorithm was about 25% less than that of the B-link algorithm. The waiting time for the tree latch for page splits contributed to the increase in response time, leading to a loss in throughput. In trees with a low fanout, however, the modified algorithm performed much worse when compared to the B-link algorithms. In fact, it performed even worse than the optimistic algorithms in Figure 18. The impact of waiting for the tree latch was much higher in the low fanout case due to an increased number of splits.

A modification to the ARIES/IM algorithm to handle more than one page split at a time is suggested by Mohan and Levine (1992) and, if implemented, this should allow ARIES/IM to perform comparably to the B-link algorithm. It should be noted, however, that we have looked at ARIES/IM only from the concurrency perspective of single B^+-tree operations; the ARIES/IM algorithm also describes how to hold extended locks on records (to allow serializability of transactions that perform more than one B^+-tree operation) as well as how to perform recovery using write-ahead logging.

6. Comparison with Related Work

The formulas provided by Bayer and Schkolnick (1977) calculate quantities like the number of locks held by tree operations and the maximum MPL that can be handled without creating a bottleneck at the root. This is a static analysis, so it does not provide insight into the dynamic performance of the various algorithms.

Biliris (1985) described a simulation model for the evaluation of B^+-tree algorithms and presented a set of experiments comparing four algorithms that included the Samadi algorithm (Samadi, 1976), the B-SIX algorithm, the side-branching algorithm (Kwong and Wood, 1982), and the mU algorithm (Biliris, 1987). This study found that the pessimistic algorithms bottleneck at the root and that the mU algorithm performed better in the situations considered. The side-branching technique was found not to give any improvement over B-SIX. The main shortcomings of

this study are that the optimistic and B-link algorithms were not studied, response times for individual operation types were not given, and no detailed analysis of the results was provided.

The most recently published performance analysis of B^+-tree concurrency control algorithms was based on analytic modeling of an open queuing system (Johnson and Shasha, 1990). This study assumed infinite resource conditions and did not model buffer management. The algorithms compared in the study are a naive lock-coupling algorithm (B-X), an optimistic algorithm (B-OPT modified with the second phase using X locks instead of SIX locks), and the LY version of the B-link algorithm. The key results of this study are that the root becomes a bottleneck for the lock-coupling algorithms, and that, in situations where link-chases are rare, the LY B-link algorithm performs much better than the other algorithms. Our simulation model differs from their analytical model in that we take into account resource contention and buffer management. In addition to the low contention situations analyzed in their model, we have studied very high concurrency situations where a large number of link-chases are performed by the B-link algorithms. We have also analyzed differences between several variants of the B-link algorithm. Some of our results differ from theirs; for example, they found that the optimistic descent algorithm always performed much better than the naive lock-coupling algorithm, while we found that the optimistic algorithms sometimes perform close to their corresponding pessimistic versions at high MPLs (e.g., for the 100% insert workload on a low fanout tree), though they indeed provide much better performance at intermediate MPLs. Finally, their model allows only IS and X locks, while we have considered more complicated algorithms that use SIX locks to enable certain tree operations to overtake others on their descent to the leaf.

In parallel with the work reported in this article, an extended study (Johnson, 1990) handled SIX locks and four additional algorithms: TD-X, B-OPT, two-phase locking, and a parameterized optimistic descent algorithm (in which the number of levels X-locked during the first descent is a parameter). The LY algorithm still performed the best among all of the algorithms considered. The response times of the two-phase locking algorithm varied widely due to a large number of deadlocks and long waiting times at the root. While we have not modeled two-phase locking, the OPT-DLOCK algorithm is really an optimized version of the standard two-phase locking algorithm. Because we found that OPT-DLOCK performed very well for low fanout trees, not so well for high fanout trees, and poorly in the append experiments, our results (like Johnson, 1990) predict that two-phase locking (which is necessarily worse than OPT-DLOCK) cannot be better than the B-link algorithms. Furthermore, we have also modeled an additional algorithm, TD-OPT, and found in our experiments that TD-OPT almost always outperforms B-OPT. An important additional result in our study is that bottlenecks at the root were shown to affect different operation types differently. In our low contention experiments (Experiment Set 1), for example, searches in B-SIX were faster even than searches in the B-link algorithms, though the overall throughput of the B-link algorithms

was much higher than that of the B-SIX algorithm (Figures 6 and 9).

The analytical model in Johnson and Shasha (1990) and Johnson (1990) assumes that the proportion of inserts in the workload is always larger than that of deletes (i.e., they model only growing trees). Also, the workload model assumes that operations access leaf nodes uniformly. In contrast, we have studied a wider range of workloads that lead to both steady-state trees as well as growing trees, and hence our results are somewhat more general. In addition, our append experiments model a situation with highly concurrent and skewed access to portions of the B^+-tree.

In addition to extending their work to additional algorithms, Johnson (1990) modeled resource contention using a service time dilation factor and found that (1) under very high resource contention there was no difference between the various algorithms and (2) under moderate to light contention there was a significant difference between the algorithms. These conclusions essentially agree with our results. In addition, we found that in high resource contention situations a highly skewed append workload causes the B-link algorithms to exhibit a thrashing behavior; in fact, in such situations, the B-link algorithms performed worse than the exclusive lock-coupling algorithms at high MPLs (Figure 21). Johnson (1990) also showed that LRU buffering will hold the highest levels of the tree in memory. While our results agree, we have also characterized the extra utilization of the buffer pool by variations of the B-link algorithms. Finally, using the insights generated from our study, we have been able to make predictions regarding the performance of other algorithms in the literature.

7. Effect of Multiple Operation Transactions

So far we have considered only one aspect of concurrency control on B^+-tree indices, namely, transactions that perform single B^+-tree operations. In the more general case, where several B^+-tree operations are part of a larger transaction, DBMSs serialize transactions by retaining some lock(s) from each B^+-tree operation until the end of that transaction.

An obvious long-term locking strategy would be to hold the locks on all modified index pages until end of transaction. Such a strategy can be quite restrictive, however, so an alternative strategy that only holds locks on leaf pages can be much more efficient (Johnson and Shasha, 1990). In some cases, even holding leaf page locks might be too restrictive, and therefore highly concurrent strategies like record locking (Mohan and Levine, 1992) or key-value locking (Mohan, 1990) have been proposed in the literature. In addition to the data contention caused by using B^+-tree concurrency control algorithms, long-term lock holding strategies like those above can cause additional data contention that may have a significant performance impact on the concurrent execution of transactions. It is possible that using a restrictive long-term locking strategy might nullify the effects of using a highly concurrent B^+-tree concurrency control algorithm. In this section, therefore, we investigate how the leaf-page locking and record locking strategies affect the performance of

transactions that contain multiple B^+-tree operations.

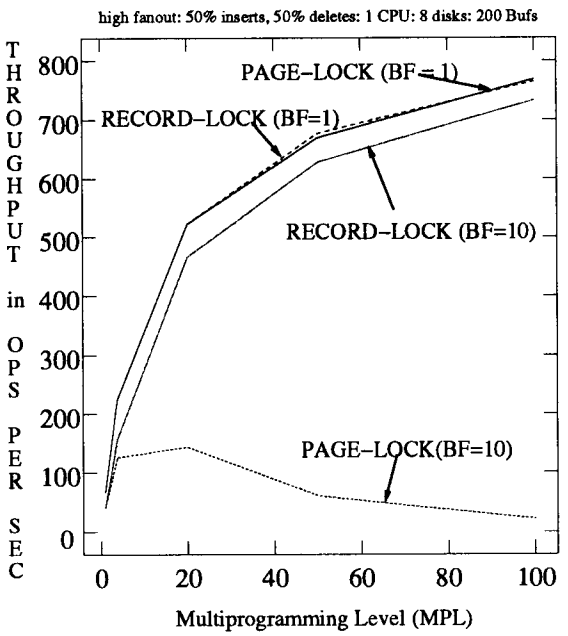
To investigate long-term lock holding strategies, we modified our simulator as follows: Instead of modeling just the individual B^+-tree updates as before, B^+-tree updates are now modeled as random updates to a relation. The relation itself is maintained as a heap file on disk, and a B^+-tree index is associated with one of the relation's columns. Our workload model is now composed of transactions that contain multiple B^+-tree operations. The number of operations per transaction is termed the *batching factor*. The proportion of relation searches, inserts, and deletes is controlled by the workload parameters, as in our earlier experiments. Each transaction executes a stream of relation operations, each of which is immediately followed by a corresponding B^+-tree operation. Leaf-page locks or record locks from each B^+-tree operation, depending on the locking strategy, are held until end of transaction. Deadlocks that arise while holding long-term locks are broken by aborting the most recently started transaction; an aborted transaction first undoes any updates that it has already performed to the relation and the B^+-tree, and it then releases its long-term locks before re-executing.

Figure 23 presents throughput curves for the two long-term locking strategies, leaf-page locking (PAGE-LOCK) and record locking (RECORD-LOCK), for a workload of 50% inserts and 50% deletes; results are shown for batching factors of 1 and 10. The tree here is the high fanout tree discussed earlier in Section 5. The initial tree has 40,000 entries and around 250 leaf pages. The experiments were run on a system configuration of 1 CPU and 8 disks with a buffer pool size of 200. In all cases, the B^+-tree concurrency control algorithm used was LY-LC. The throughput in Figure 23 is given in units of operations/second, where an operation consists of a relation operation and its corresponding B^+-tree operation, to allow all of the results to be compared via a single graph.

In Figure 23, we find virtually no difference between leaf-page locking and record locking for a batching factor of 1. This is expected because, with only one operation per transaction, the long-term locking strategy has no effect on performance; any performance difference would be determined by the B^+-tree concurrency control algorithm, which is the same in both cases (LY-LC). We also find that record locking with a batching factor of 10 performs very close to record locking with a batching factor of 1. On the other hand, leaf-page locking with a batching factor of 10 performs much worse than both leaf-page locking with a batching factor of 1 and record locking with a batching factor of 10. The reduced performance of leaf-page locking at higher batching factors is due to deadlocks that are caused by holding leaf-page locks. These deadlocks cause transaction aborts and re-executions that result in a severe loss of throughput. This does not occur in the case of record locking because there are many more record IDs (around 40,000) than there are leaf pages (around 250), so very few deadlocks occur with record locking.

The above experiments clearly show that using a highly concurrent B^+-tree algorithm is warranted only if highly concurrent long-term locking strategies are used to protect the updated data, as one might well expect. In particular, leaf page

Figure 23. Long-term locking effects



locking is unlikely to be sufficient, and it appears necessary to use record locking (or another highly concurrent strategy like key-value locking [Mohan, 1990]) to make full use of the concurrency achieved by the highly concurrent LY-LC algorithm.

8. Some Practical Issues

The performance experiments presented here have demonstrated the high concurrency allowed by B-link algorithms in a wide range of situations. Before these algorithms can be incorporated into a real database system, however, certain extensions to the basic algorithms are needed; this has led some researchers to question their practicality. In particular, recovery strategies are needed, variable length keys need to be handled, and deleted B^+-tree pages must be re-usable. In this section, we argue that the required extensions either exist or are relatively straightforward.

8.1 Recovery

As initially proposed, little or no consideration was given to the problem of the interaction of recovery with the B-link algorithms. Fortunately, Lomet and Salzberg (1992) have proposed viable recovery strategies for the B-link algorithms. As another alternative, it would also be possible to adapt the techniques used in the ARIES/IM algorithm (Mohan and Levine, 1989, 1992) for this purpose. While a discussion of B-link tree recovery issues is beyond the scope of this article, the point to be noted

is that viable approaches do exist.

8.2 Variable Length Keys

Recall that in a B-link tree, all of the nodes at the same level are linked together, and a high key is maintained in every node at each level of the tree (Figure 3). Mohan and Levine (1989) pointed out that the combination of variable length keys and the use of high keys at every level can lead to certain extra overheads related to the maintenance of high keys in non-leaf pages. For example, deleting a node that is the last child of its parent node requires a new high key to be propagated up to the parent. Because such a high key may sometimes have to be propagated up more than one level, it is possible for such propagations to result in extra disk I/Os for reading in the required parent nodes. Furthermore, the propagation of high keys can actually result in page splits; this occurs when the propagated key is larger than the current high key in a page and the page is already full. For these reasons, the ARIES/IM algorithm does not store high keys in non-leaf nodes; only leaf nodes store high key values. However, there are also disadvantages to not keeping high keys in the non-leaf nodes, such as the need for repeated accesses from the root when certain ambiguities arise due to concurrent structure modifications (Mohan and Levine, 1989). For example, when a key is found not to be bounded in a non-leaf page that recently participated in a page split or a page merge operation, a re-traversal from the root is sometimes required.

Despite the aforementioned potential for extra overhead in B-link deletion due to non-leaf high keys and variable length data, this extra overhead is likely to be minimal if node deletions are rare (as was the case in most of our experiments). Even when node deletions are more frequent, they only lead to high key propagation when the last child of a parent node is deleted. Assuming random access, a B^+-tree with a high occupancy ratio (>70%), and a moderate node capacity (>100 entries/page), such propagations should be relatively infrequent (<2% of all node deletions). Thus, this does not appear to be a serious impediment to the deployment of B-link algorithms in practice.

8.3 Reusing Deleted Pages

Recall that the B-link algorithms do not lock couple on their way down the tree. Since the path from the root to the leaf is remembered in each traversal, and can be later retraced (in reverse) during the upward propagation of structure modification operations, deleted non-leaf nodes cannot be reused immediately. Instead, their removal must be deferred until it can be guaranteed that they will not be reaccessed by a concurrent operation. We will mention two possible ways to handle this.

One (simple) solution is to wait for all of the tree operations that were active at the time of the delete to complete before actually reusing the deleted node. This is called the drain technique (Lanin and Shasha, 1986). An alternative (slightly more complicated) technique would be to store a version number on each B^+-tree

page, incrementing this number each time a page is reused. While reading pages on the way down the tree, a process should remember the version number associated with each accessed page. Later, when reaccessing pages during the propagation of a structure modification operation, the process can compare the current version number of the page to the version number noted earlier. If the two version numbers do not match, then a retraversal from the root is necessary.

9. Conclusion

The most important conclusion of this study is that the B-link algorithms perform the best among all of the algorithms that we studied over a wide range of resource conditions, B^+-tree structures, and workload parameters. Even in a high contention workload of appends, the B-link algorithms show gains in throughput under plentiful resource conditions. The reason for the excellent performance of the B-link algorithms is the absence of any bottleneck formation (except, of course, at the CPUs or disks in resource-constrained situations). In contrast, in all of the other algorithms, locking bottlenecks form at high MPLs if the workload contains a significant percentage of updaters. Moreover, the overhead that the B-link algorithms incur in very high data contention situations are link-chases, which turn out to be inexpensive. We also found interesting differences in the behavior of the optimistic and pessimistic algorithms among themselves.

Among the B-link algorithms, we have shown further that a slightly conservative update algorithm that locks a maximum of three nodes at a time generally performs as well as one that locks a maximum of only one node at a time. The more conservative variation of the B-link algorithm is more suitable for use in practice, as it avoids some rather complex (inconsistent) situations that can arise in the latter. Finally, we demonstrated that using a highly concurrent long-term lock holding strategy is necessary to get the full benefits of using a highly concurrent algorithm for index access.

Our investigation of multiple operation transactions in the last section did not consider certain special types of queries like range scans. Range scans, which typically access a much larger amount of data than normal transactions, may interact differently with long-term lock holding strategies than normal transactions would. In particular, leaf page locking might be better than record locking if the range scan accesses a rather large number of entries. Furthermore, using two-phase locking to serialize large range scans with conventional transactions may lead to an unacceptable reduction in transaction throughput, and techniques like transient versioning (Chan et al., 1982) or compensation-based query processing (Srinivasan and Carey, 1992) might be preferable for such workloads. We are currently studying these and other related issues.

Acknowledgments

This research was partially supported by the National Science Foundation under grant IRI-8657323 and by a University of Wisconsin Vilas Fellowship. We would like to thank Miron Livny for his help in using the DeNet simulation environment; David Dewitt and Miron Livny for suggestions that helped us to improve the presentation of the results; Rajesh Mansharamani, Manolis Tsangaris, and S. Seshadri for their constructive comments on a preliminary version of this article; and the anonymous referees for suggesting changes that have improved the presentation significantly.

References

Agrawal, R., Carey, M., and Livny, M. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems,* 12(4):609-664, 1987.

Bayer, R. and McCreight, E. Organization and maintainance of large ordered indices. *Acta Informatica,* 1(3):173-189, 1972.

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

Biliris, A. A model for the evaluation of concurrency control algorithms on B-trees. *Computer Science Technical Report,* No. 85-015, Boston University, 1985.

Biliris, A. Operation specific locking in B-trees. *Proceedings of the Sixth ACM Symposium on Principles of Database Systems,* San Diego, California, 1987.

Carey, M. and Thompson, C. An efficient implementation of search trees on $\lceil \lg N + 1 \rceil$ processors. *IEEE Transactions on Computer Systems,* C-33(11):1038-1041, 1984.

Chan, A., Fox, S., Lin, W., Nori, A., and Ries, D. The implementation of an integrated concurrency control and recovery scheme. *Proceedings of the ACM SIGMOD Conference,* Orlando, Florida, 1982.

Comer, D. The ubiquitous B-tree. *ACM Computing Surveys,* 11(4):121-137, 1979.

Franaszek, P. and Robinson, J. Limitations of concurrency in transaction processing. *ACM Transactions on Database Systems,* 10(1):1-28, 1985.

Goodman, N. and Shasha, D. Semantically-based concurrency control for search structures. *Proceedings of the Fourth ACM Symposium on Principles of Database Systems,* Portland, Oregon, 1985.

Gray, J. Notes on database operating systems. In: *Operating Systems: An Advanced Course* New York: Springer-Verlag, 1979.

Guibas, L. and Sedgewick, R. A dichromatic framework for balanced trees. *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science,* New York, 1978.

Johnson, T. and Shasha, D. Utilization of B-trees with inserts, deletes and searches. *Proceedings of the Eighth ACM Symposium on Principles of Database Systems,* Philadelphia, Pennsylvania, 1989.

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

Johnson, T. The performance of concurrent data structure algorithms. Ph.D. Thesis, New York University, May 1990.

Kersten, M. and Tebra, H. Application of an optimistic concurrency control method. *Software-Practice and Experience,* 14(2):153-168, 1984.

Kung, H. and Lehman, P. A concurrent database manipulation problem: Binary search trees. *ACM Transactions on Database Systems,* 5(3):339-353, 1980.

Kwong, Y. and Wood, D. A new method for concurrency in B-trees. *IEEE Transactions on Software Engineering,* 8(3):211-223, 1982.

Lanin, V. and Shasha, D. A symmetric concurrent B-tree algorithm. *Proceedings of the Fall Joint Computer Conference,* Dallas, Texas, 1986.

Lausen, G. Integrated concurrency control in shared B-trees. *Computing,* 33(1):13-26, 1984.

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

Livny, M. *DeNet User's Guide,* Version 1.5. Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1990.

Lomet, D. and Salzberg, B. Access method concurrency with recovery. *Proceedings of the ACM SIGMOD Conference,* San Diego, California, 1992.

Miller, R. and Snyder, L. Multiple access to B-trees. *Proceedings of the Conference on Information Science and Systems,* Johns Hopkins University, Baltimore, Maryland, 1978.

Mohan, C. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. *Proceedings of the Sixteenth International Conference on Very Large Data Bases,* Brisbane, Australia, 1990.

Mohan, C. and Levine, F. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. IBM Research Report, RJ6846. IBM Almaden Research Center, August 1989.

Mohan, C. and Levine, F. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. *Proceedings of the ACM SIGMOD Conference,* San Diego, California, 1992.

Mond, Y. and Raz, Y. Concurrency control in B^+-trees databases using preparatory operations. *Proceedings of the Eleventh International Conference on Very Large Data Bases,* Stockholm, Sweden, 1985.

Sagiv, Y. Concurrent operations on B^*-trees with overtaking. *Proceedings of the Fourth ACM Symposium on Principles of Database Systems,* Portland, Oregon, 1985.

Samadi, B. B-trees in a System With Multiple Users. *Information Processing Letters,* 5(4):107-112, 1976.

Shasha, D. Concurrent algorithms for search structures. Ph.D. Thesis, Aiken Computation Laboratory, Harvard University, 1984.

Shasha, D. What good are concurrent search structure algorithms for databases anyway? *Database Engineering,* 8(4):84-90, 1985.

Srinivasan, V. On-line processing in large-scale transaction systems. Ph.D. Thesis, Computer Science Department, University of Wisconsin-Madison, 1992.

Srinivasan, V. and Carey, M. Performance of B-tree concurrency control algorithms. *Proceedings of the ACM SIGMOD Conference,* Denver, Colorado, 1991.

Srinivasan, V. and Carey, M. Compensation-based on-line query processing. *Proceedings of the ACM SIGMOD Conference,* San Diego, California, 1992.

Tay, Y. A mean value performance model for locking in databases. Ph.D. Thesis, Computer Science Department, Harvard University, 1984.

Yao, A. On random 2-3 trees. *Acta Informatica,* 9(2):159-170, 1978.