On-Line Processing
In
Large-Scale Transaction Systems

by

Venkatachary Srinivasan

Computer Sciences Technical Report #1071
January 1992

ON-LINE PROCESSING IN LARGE-SCALE TRANSACTION SYSTEMS

by

Venkatachary Srinivasan

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
University of Wisconsin-Madison
1992

Abstract

In this thesis, we provide techniques to adapt current database technology to account for the following trends that can be observed in database management system (DBMS) usage:

1. DBMSs are being increasingly used in applications, like computerized stock trading, that have very high transaction rates.

2. Database sizes are growing rapidly, and future databases are expected to be several orders of magnitude larger than the largest databases in operation today.

3. Next generation DBMSs are expected to gravitate more and more towards what is referred to as *24(hour)* $\times$ *7(day)* operation.

In order to handle high transaction rates, future DBMSs have to use highly concurrent algorithms for managing often-used auxiliary data structures like indices. To better understand the performance of concurrency control algorithms for index access, we first compare the performance of B-tree concurrency control algorithms using a simulation model of a centralized DBMS. In our performance study, we look at a wide range of B-tree concurrency control algorithms, including several variants of existing algorithms as well as a new algorithm. Based on the performance results, we characterize how specific details of a concurrency control algorithm can enhance or reduce concurrency. In particular, our results show that, over a wide range of resource and data contention situations, the B-link algorithm of Lehman and Yao performs very well; we identify a particular variant of this algorithm that appears especially suitable for use in practice.

On-line DBMS utilities are an important step in reaching the goal of handling large amounts of data and achieving 24×7 operation. This thesis addresses issues involved in executing on-line utilities by developing several new algorithms for on-line index construction. These algorithms each permit an index to be built while the corresponding data is concurrently accessed for reads and writes. The algorithms work incrementally, producing a consistent index in the end. They differ in the data structures used for storing concurrent updates as well as in the degree of concurrency allowed during index construction. A comprehensive performance study of the proposed on-line

index construction algorithms is used to determine the best candidate for use in a DBMS. The performance study also clearly demonstrates the superiority of on-line operation in a DBMS, even for small data sets.

The techniques used in on-line index construction algorithms can be generalized to efficiently execute long-running queries that are currently handled unsatisfactorily in conventional DBMSs. Using conventional concurrency control techniques for obtaining serializable answers to long-running queries leads to an unacceptable drop in system performance. Current DBMSs therefore execute such queries under a reduced degree of consistency, thus obtaining non-serializable answers. Applying the techniques used for on-line index construction to query processing leads to a new, highly concurrent method of query execution called compensation-based query processing. In this new approach to query processing, concurrent updates to any data participating in a query are communicated to the query's on-line query processor, which then compensates for these updates so that the final answer reflects changes caused by the updates. Very high concurrency is achieved by locking data only briefly, at the tuple-level, while still delivering transaction-consistent answers to queries. Compensation-based query processing can co-exist with conventional query processing, and a cost model similar to that used for optimizing conventional queries can be used for optimizing queries in the new model as well. Finally, it appears that compensation-based query processing can be implemented efficiently in a DBMS.

Acknowledgements

Mike Carey introduced me to the field of database management systems. It has been extremely challenging and enjoyable to work with him, and he has truly taught me a great deal. David Dewitt, Yannis Ioannidis, Miron Livny, and Jeff Mogul have taught me much as well. I would also like to specifically thank Richard Lipton of MITL for a valuable discussion I had with him regarding a part of this thesis. In addition, I wish to thank all of my other teachers who have been instrumental to my success.

My colleagues, Dan, Divesh, Kurt, Manish, Manolis, Mike Franklin, Paul, Praveen, Scott, Sesh, Seth, Sudarshan, and Tan sat through many practice talks, acted as a bouncing board for my ideas, and read drafts of papers. I thank Lorene, Marie, Sheryl, and Susan for their valuable help with various administrative matters.

I wish to express the deepest gratitude to my parents for the love and affection that they have bestowed on me throughout. I would also like to thank my friends and relatives who have treated me with kindness and affection. Most importantly, my wife Viji's love and support encouraged me to complete this dissertation.

I also acknowledge the support of the National Science Foundation under grant IRI-8657323, and the Vilas fellowship provided by the University of Wisconsin.

Contents

Chapter 1

Introduction

The field of database management systems (DBMS) has seen phenomenal growth in the last two decades, and DBMSs have become standard software in all sorts of computer systems – from large mainframe computers to smaller, less powerful personal computers. In spite of the widespread use of DBMSs, much work remains to be done in order to adapt current database technology to accommodate the following trends that can be observed in DBMS usage:

1. DBMSs are being increasingly used in applications, like computerized stock trading, that have very high transaction rates.

2. Database sizes are growing rapidly, and future databases are expected to be several orders of magnitude larger than the largest databases in operation today. Databases on the order of terabytes (10^{12} bytes) will soon be in active use [Silb90].

3. Next generation DBMSs are expected to gravitate more and more towards what is referred to as *24(hour) × 7(day)* operation. There exist important DBMS applications that have no significant *off-peak* time, which is time when it becomes acceptable to take the data off-line for maintenance purposes. Examples of such 24×7 systems include database management for multinational companies with a global reach, hospital management systems, round-the-clock shopping services, etc. In order to service such applications, next generation databases will be required to keep their data on-line all of the time [Dewi90, Silb90].

The above trends affect important aspects of database design and implementation, as discussed below.

Under very high transaction rates, contention in heavily used auxiliary data structures like indices can increase tremendously, necessitating the use of highly concurrent algorithms for managing these data structures. Concurrency control techniques that work well for records or data pages,

such as two-phase locking [Gray79], are overly restrictive when naively applied to such items as index pages. Special techniques must be employed to prevent indices and system catalogs from becoming concurrency bottlenecks. A number of algorithms have been proposed for accessing indices concurrently [Sama76, Baye77, Mill78, Lehm81, Kwon82, Shas84, Good85, Mond85, Sagi85, Shas85, Lani86, Bili87, Moha89, Weih90], but no performance analyses existed until recently that compare all of these algorithms. The earlier studies [Baye77, Bili85, Shas85, Lani86, John90a] each compare only a few algorithms and have been based on simplified assumptions about resource contention and buffer management. As a result, the relative performance of these algorithms was still an open question when work began on this thesis[1].

The explosion in database sizes will necessitate the scaling up of all of the algorithms used in a DBMS, including the class of database utilities. Utilities are typically used for re-organization of data and for construction and maintenance of hidden data structures like indices. While DBMS utilities like index construction take a few minutes to execute for relatively small amounts of data (e.g., a gigabyte), they can take days to complete for large amounts of data (e.g., a terabyte) due to the time it takes to scan the data itself. Current DBMSs execute utilities off-line and hence are ill-suited to execute utilities on large amounts of data (since the data would be taken off-line for days). In addition, on-line utilities are essential, even for smaller data sets, in order to achieve 24×7 operation. With high transaction arrival rates, any off-line processing is bound to cost heavily in terms of the number of transactions that are queued up during the associated down-time. Techniques for executing all utilities in an on-line manner are therefore needed in next generation DBMSs.

In addition to the need for on-line utilities in next generation DBMSs, there is also a need for improving the execution of certain queries (used, for example, in decision support) on large amounts of data which are not executed satisfactorily by current conventional DBMSs. We shall explain the sort of queries of interest by using an example.

Q1: Suppose an auditor of a company wants to know the average salary of all of the employees of the company. Assuming the existence of a relation called EMPLOYEE with a SALARY attribute, the SQL form of the auditor's query is given below.

```
SELECT    AVG(SALARY)
FROM      EMPLOYEE
```

[1]It should be noted that the work reported in [John90a] has been extended concurrently with this work, and another comprehensive study of B-tree concurrency control algorithms is now available in [John90b].

Current systems, depending on the details of their implementation, will handle such a query in one of several ways:

1. One way of executing a transaction to compute Q1 using two-phase locking involves locking the EMPLOYEE relation in *Share* mode, reading all the tuples of the relation, and keeping a running sum of the salary values encountered as well as a count of the number of tuples read. On completing the scan of the relation, the average salary is computed by dividing the sum by the count. It is easy to see that the above method of executing Q1 makes it serializable with respect to all other transactions using the EMPLOYEE relation. However, this method of executing Q1 is disastrous for concurrency purposes, as no updates to the EMPLOYEE relation are allowed during the execution of Q1.

2. A slightly more concurrent way to execute Q1 would be to lock the EMPLOYEE relation in Intention-Share mode and then lock individual tuples as they are read in Share mode, with all locks being held until end of transaction. This method allows execution of update transactions on the portion of the EMPLOYEE relation that has not yet been read by Q1, but it still locks out large portions of the EMPLOYEE relation for a significant period of time. This strategy is roughly half as restrictive as the first one in terms of the amount of data locked by Q1 as a function of time.

3. A third way of executing Q1 would be to lock only the SALARY attribute of the EMPLOYEE relation in Share mode, thus allowing updates to other attributes of existing tuples of the relation. Such selective locking of attributes may be possible, for example, in a system that uses key-value locking of the type described in [Moha90] if an index exists on the SALARY attribute. Still, locking the SALARY attribute would rule out inserts and deletes of new EMPLOYEE tuples, and would block updates to the SALARY attribute of the existing tuples.

Since each of these ways of executing Q1 involves significant concurrency restrictions for other transactions, DBMSs currently tend to execute queries like Q1 under a weakened degree of consistency. For example, IBM's System R and DB2 offer the concept of *cursor stability* [Gray79], where a query like Q1 looks only at committed updates of other transactions, but holds locks on tuples only while their values are actually being read. Executing Q1 under a reduced degree of consistency would involve getting an Intention-Share lock on the EMPLOYEE relation and then scanning the relation tuple by tuple, acquiring a Share lock on each tuple before reading it and then releasing the lock immediately after the tuple is read. The advantage of cursor stability is that there is minimal

delay for other transactions in the system due to executing Q1. The disadvantage is that, while the salary values read are individually correct values, they are not from one transaction-consistent state of the EMPLOYEE relation. The answer obtained by Q1 is therefore approximate, and in some cases may bear little resemblance to the correct value. The deviation of the answer from the actual value is determined by the rate and magnitude of changes to the SALARY attribute of the EMPLOYEE relation. In some applications this may be unacceptable to the person running the query, in this case the auditor.

The above discussion indicates that queries like Q1 are not dealt with satisfactorily in conventional DBMSs. Apart from the simple query that we have used as an example, many other types of large queries (used for decision support) on single or multiple base relations suffer from similar concurrency problems. In current DBMSs, the size of the data and the transaction rates might be small enough that the inefficient execution of such queries is only a minor problem. For example, a simple solution that is likely to work satisfactorily with a conventional DBMS is to take a copy of the database periodically (during off-peak time) and to run such large queries on the copy. As transaction rates and database sizes increase, however, such a solution becomes unsuitable due to the following reasons:

1. With large databases, it could take a very long time to create a copy of the database, and the copy would also double the already large storage requirements for the database.

2. With large transaction rates and long copy-times, the copied data might become out of date very soon after a copy is taken. This would necessitate frequent copying for obtaining relatively recent answers to queries.

For these reasons, efficient on-line execution of long-running queries will become essential in next generation DBMSs; any other solution that provides transaction-consistent answers will surely turn out to have too high a concurrency overhead and/or copying overhead.

In this thesis, we provide techniques for solving the above problems, thus helping to pave the way for DBMSs to handle applications with large transaction rates, very large database sizes, and completely on-line (24×7) operation. This thesis is subdivided into three major parts, each of which is previewed briefly below.

1.1 Performance of B-Tree Concurrency Control Algorithms

The performance of concurrency control algorithms for index access had not been satisfactorily studied when we started working on this thesis, as mentioned earlier. Since B-trees[2] are the most common dynamic index structures in DBMSs, most earlier work has concentrated on them, and our focus is also on B-tree concurrency control algorithms. However, many of our results will lend insight into concurrency control for other index structures also. We study the performance of various concurrency control algorithms using a detailed simulation model of B-tree operations in a centralized DBMS. Our study considers a wide range of data contention situations and resource conditions. Based on the performance results, we characterize how specific details of a concurrency control algorithm can enhance or reduce concurrency. An interesting aspect of our study is that, based on the performance of a representative set of B-tree concurrency control algorithms, including one new algorithm, we can make projections on the performance of others in the literature. Finally, our study is more detailed in several ways than other earlier and contemporary studies, thus resulting in a significant set of new results as well as a corroboration of other recent results.

1.2 On-Line Index Construction

On-line utilities are an important step in reaching the goals of handling large amounts of data and achieving 24×7 operation. This thesis addresses issues involved in executing on-line utilities by describing several new algorithms for on-line index construction. All of the algorithms presented build an index while the underlying data is concurrently accessed and updated, and we prove that our on-line index construction algorithms create a consistent index. A comprehensive performance study of the index construction algorithms is used to determine the best among the candidate on-line index construction algorithms for use in a DBMS. The performance study also clearly demonstrates the superiority of on-line operation in a DBMS, even for small data sets.

1.3 Compensation-Based Query Processing

The techniques used in on-line index construction algorithms can be generalized to efficiently execute large queries of the sort discussed earlier. This leads to a new, highly concurrent method of query processing that we refer to as compensation-based query processing. In this new approach to query processing, concurrent updates to any data participating in a query are communicated to the on-line query processor, which then compensates for these updates so that the final answer reflects

[2]By B-tree we mean the variant in which all keys are stored at the leaves, also called B^+-trees and sometimes B^*-trees [Come79].

changes caused by the updates. Compensation-based query processing achieves very high concurrency by locking data only briefly, at the tuple-level, while still delivering transaction-consistent answers to queries. Such a model of query processing makes it possible for long-running queries, which usually run under a reduced degree of concurrency in current DBMSs, to obtain transaction-consistent answers without adversely affecting system performance. Compensation-based query processing can co-exist with conventional query processing, and a cost model similar to that used for optimizing conventional queries can be used for optimizing queries in the new model as well. It also appears that compensation-based query processing can be implemented efficiently in a DBMS.

1.4 Thesis Organization

The remainder of the thesis is organized as follows: In Chapter 2, we present a performance study of B-tree concurrency control algorithms. Chapter 3 describes a set of algorithms for on-line index construction, and Chapter 4 presents a performance study of these algorithms that identifies the best of the candidate algorithms for use in a DBMS. Chapter 5 generalizes our on-line index construction techniques to tackle the problem of efficient and concurrent execution of large queries in a DBMS. Finally, in Chapter 6 we summarize our major conclusions and discuss opportunities for future work. The Appendix contains proofs of correctness for the on-line index construction algorithms described in Chapter 3.

Chapter 2

Performance of B-Tree Concurrency Control Algorithms

2.1 Introduction

Database systems frequently use indices to access data. These systems typically operate at a high level of concurrency, and since any transaction has a high probability of accessing an index, it is necessary to ensure that concurrent access to indices is not a bottleneck in the system. As we mentioned earlier, since B-trees[1] are the most common dynamic index structures in database systems, we will focus on B-tree concurrency control algorithms in this chapter.

A number of algorithms have been proposed for accessing B-trees concurrently [Sama76, Baye77, Mill78, Lehm81, Kwon82, Shas84, Good85, Mond85, Sagi85, Shas85, Lani86, Bili87, Moha89, Weih90], but few performance analyses exist that compare these algorithms. Most earlier studies [Baye77, Bili85, Shas85, Lani86, John90a] each 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 thesis. An extension of the work in [John90a] resulted in a more comprehensive study of B-tree concurrency control algorithms [John90b]. That work was done concurrently with our work, which first appeared as [Srin91], and we will compare our performance results with those of [John90b] at the end of this chapter.

In this chapter, 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,

[1] Again, by B-tree we mean the variant in which all keys are stored at the leaves, also called B^+-trees and sometimes B^*-trees [Come79]

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. In our experiments, 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 2.2 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 2.3. Section 2.4 presents details of our experiments and results. In Section 2.5, we discuss how these results can be used to predict the performance of other protocols. Section 2.6 compares this study with related work. Finally, in Section 2.7, we summarize our key results.

2.2 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 shall restrict ourselves to unique indices with fixed length keys for this study, most of our results also directly apply to trees with variable length keys or duplicate keys.

2.2.1 B-tree Review and Terminology

A B-tree index is a page-oriented tree that has the following properties. Firstly, 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

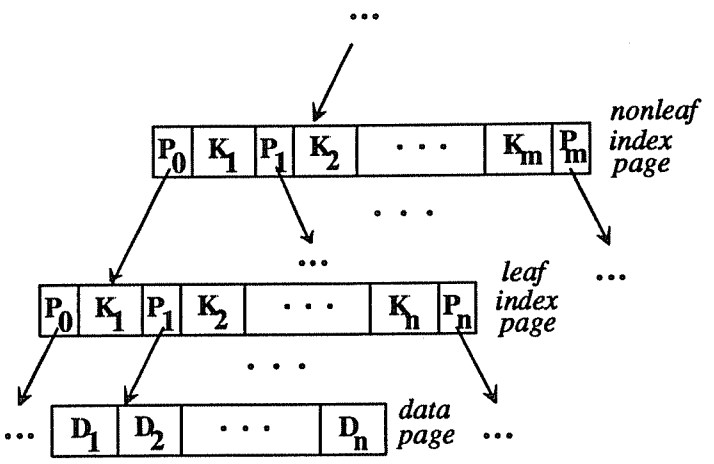


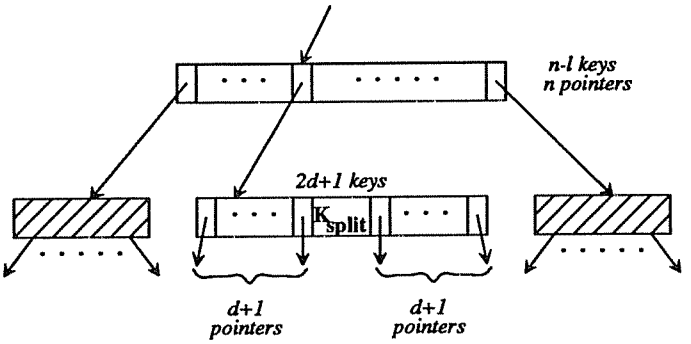
Figure 2.1: A B-Tree Fragment

are of the same length. A B-tree is said to be of order d if every node has at most $2d$ separators[2], 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 nonleaf 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 2.1 summarizes these concepts.

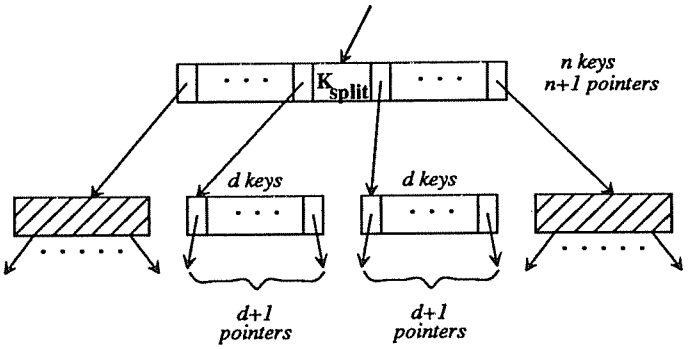
The index is stored on disk, and a search, insert, or delete operation starts by searching the root to find the page at the next lower level that contains the subtree having the search key in its range. The next lower level page is searched, and so on, until a leaf is reached. The leaf is then searched 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 2.2. 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, since this simplifies implementation and for practical workloads is not found to decrease occupancy by much [John89]. 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

[2]A *key* is usually meant to imply 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.



2a) before split



2b) after split

Figure 2.2: A B-Tree Page Split

mode	S	IX	SIX	X
S	√	√	√	
IX	√	√		
SIX	√			
X				

Table 2.1: Lock Compatibility Table

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.

2.2.2 B-tree Concurrency Control Algorithms

In our discussions of this section and the rest of the chapter, we shall use the lock modes S, IX, SIX, and X. Their lock compatibility relationships are given in Table 2.1.

A naive B-tree concurrency control algorithm would treat the entire B-tree as a single data item and use locks (or latches[3]) on just the root page to prevent conflicts. Readers (searches) would

[3]Latches [Moha89] can be thought of as fast locks. They are also less general than locks; eg., no deadlock detection is performed for latch waits. In this chapter, latches can be used wherever locks are used.

get S 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 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 proposed in [Sama76], 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 [Baye77].

Bayer-Schkolnick Algorithms

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

In the first algorithm, called 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 (since SIX locks are compatible with S 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.

Top-down Algorithms

In top-down algorithms [Guib78, Care84b, Mond85, Lani86], 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. The above 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 S 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 [Lani86]. Readers use the same locking strategy as in the Bayer-Schkolnick algorithms.

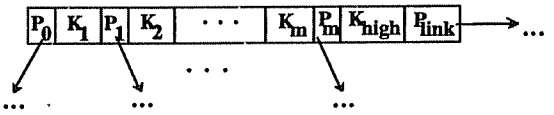
The top-down algorithms break down the updating of a scope into sub-operations that involve nodes at two adjacent levels of the tree. The B-link algorithms go one step further and limit each sub-operation to nodes at a single tree level. They also differ from the top-down algorithms in that they do their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

B-link Tree Algorithms

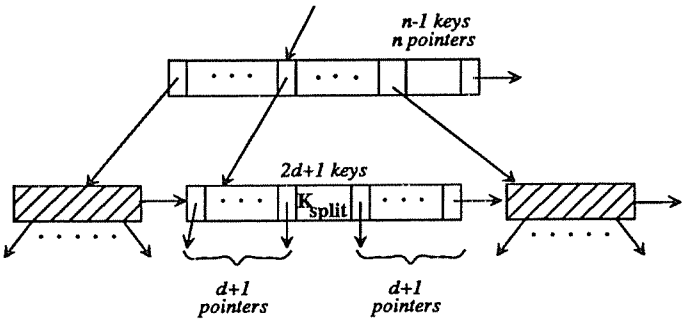
A B-link tree [Lehm81, Sagi85, Lani86] is a modification of the B-tree that uses links to chain all nodes at each level together. A page in a B-link tree contains a high key (the highest key of the subtree rooted at this page) and a link to the right sibling. The link enables a page split to occur in two phases: a half-split, 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 2.3. 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 [Lani86], via a half-merge followed by an entry deletion at the next higher tree level. The B-link algorithms that have been proposed [Lehm81, Sagi85, Lani86] 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 algorithm (LY stands for Lehman-Yao), a reader descends the tree from the root to a leaf using S 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 S 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 either find 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.

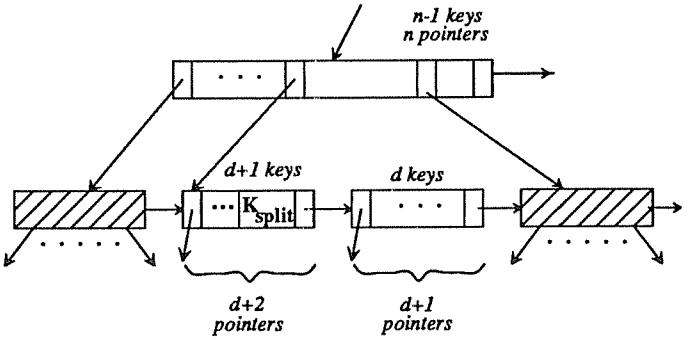
Due to the early lock releasing strategy in the basic LY algorithm, updaters that propagate index entries after completing half-splits or half-merges can encounter "inconsistent" situations; for example, a deleter at a higher level may find that the key to be deleted does not exist there (yet), and an inserter at a higher level may find that the key it is trying to insert already (still) exists. One way to take care of this problem is to force updaters that encounter such inconsistencies to



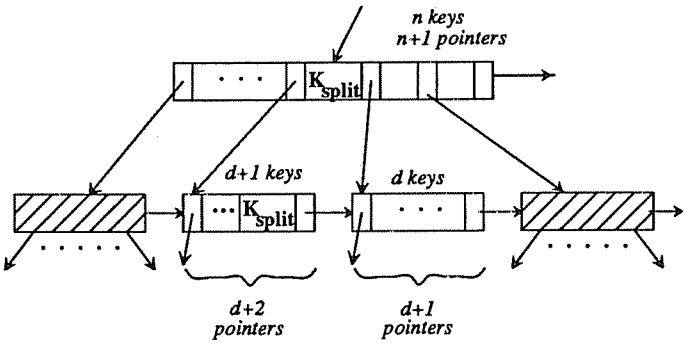
3a) example B-link tree node



3b) before half-split



3c) after half-split



3d) after key propagation

Figure 2.3: A B-Link Tree Page Split

restart repeatedly until they succeed [Lani86]. 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 S lock 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 in [Lehm81], 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 provides support for such an atomic read-write model. Since readers do not lock pages, a reader instead gets 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.

A New Optimistic Descent Algorithm

Updaters in the optimistic descent algorithms described earlier (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 will provide 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 S 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 S 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 S 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 S locks on that node to X locks, creating a local deadlock at that node. Only one of them will succeed, with the other being 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 [Kwon82, Bili87, Moha89]. We shall discuss these other algorithms in Section 5.

2.3 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 transaction submitted by a given terminal and the submission of the next one. Transactions in this study are "tree transactions," 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 transactions 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.

2.3.1 System Model

The system model is intended to encapsulate the resources present in a database system and the major aspects of the flow of transactions in the system.

The system on which tree transactions operate is modeled using the DeNet simulation language [Livn90]. The system can have one or more CPUs and disks and these are modeled using a module called the *resource manager*. A CPU resource can be used in several ways — it is used when a concurrency control request or a buffer page request is processed, or when a B-tree page is processed. Requests for the CPUs are scheduled using a FCFS (first-come first-served) discipline

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

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, and 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 utilization). 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.

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. Transactions *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 may therefore involve up to two I/Os, one to write out a dirty page and another to read the requested page in.) Apart from the buffer manager, there is a module called the *lock manager* that models the acquisition and releasing of locks.

2.3.2 Transaction Flow

Figure 2.4 shows the various states of a tree transaction in the system. Once a terminal submits a transaction, we say that the transaction is active. An active transaction is 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 a transaction requests a lock that is already held by another transaction in a conflicting mode. Transactions waiting for locks on a B-tree node are queued in order of arrival, and waiting transactions are awakened when a transaction holding a conflicting lock releases it. A transaction locks and unlocks index pages according to the locking strategy of a particular concurrency control algorithm. The third state, "access page," is entered when a transaction wants to fix or unfix a B-tree page. In this state, in the event of a buffer pool miss, a transaction 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

[5]We also ran experiments with an elevator disk scheduling algorithm, and the results of these experiments will be briefly described later.

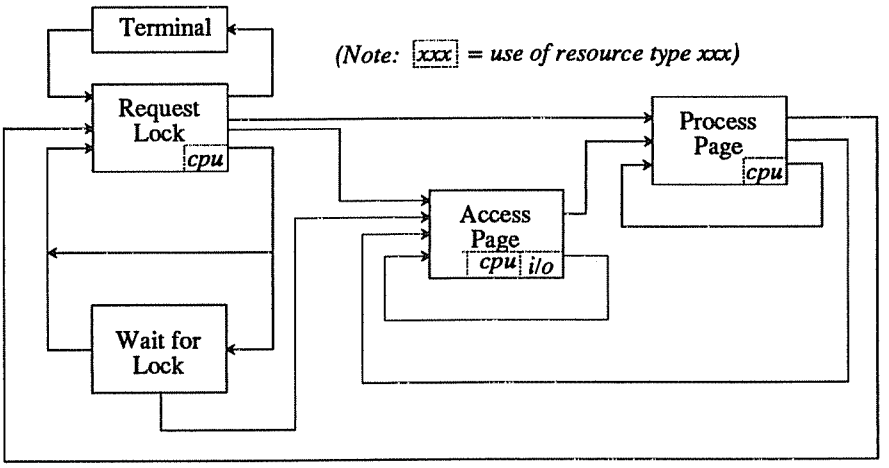


Figure 2.4: Transaction states

I/O. The fourth possible state for an active transaction is the "process page" state shown in Figure 2.4. 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 transaction 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 transaction during its execution.

2.3.3 Workload Model

One component of the workload is the number of terminals in the system, referred to as the multiprogramming level (MPL) of 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 transactions one at a time. As soon as a transaction 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.

Keys for the search, insert, and delete operations are chosen from a key space that consists of integer values between 1 and 80,000. Inserts use even keys from the key space, while deletes use odd keys, thus ensuring that inserts and deletes do not interfere at the level of key values. To ensure that deletes are always successful, an initial tree is built using a random permutation of all of the odd keys in the key space. In contrast to updaters, searches can use both odd and even integers as key values. Finally, the keys for appends are chosen sequentially from 80,001 onwards.

The actual series of keys for inserts and deletes are chosen from random permutations of their respective portions of the key space.

2.3.4 B-Tree Model

Transactions in our simulation model concurrently operate 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. For simplicity, all keys (and therefore separators) are of 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.

2.3.5 Performance Metrics

We use the above system, workload, and B-tree models as a platform for studying the performance of the B-tree concurrency control algorithms described in Section 2. The main performance metric used for the study is the throughput rate for tree operations, expressed in units of tree transactions per second (TPS). 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 will be 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 transaction throughput, it would be nice 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* [Fran85, Tay84, Agra87] 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. Transactions 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.

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)

Table 2.2: Simulation Parameters

2.3.6 Range of Experiments and Parameters

In our experiments, three factors will be 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 2.2.

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 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[6]) 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. For the high fanout case, the two buffer pool sizes are 200 pages 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 buffer pool in the low fanout tree are 600 pages (7% of the tree in memory) and 12,000

[6]Pre-splitting in the top-down algorithms leads to early splits for nonleaf pages, and 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.

pages (memory-resident tree). In cases where the buffer pool size is smaller than the size of 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 is going to be. Furthermore, any system is bound to undergo changes in workload from time to time. In order 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 span a wide range of conditions. For the in-memory tree case, we study 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 study 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 are 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 is initialized with as many B-tree pages as will fit; higher level pages are given priority in this initialization. The experiment is stopped after 10,000 operations have completed. Batch probes in the DeNet simulation language are 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.

2.4 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 divide the algorithms into four classes and analyze the performance of these classes. The groupings of algorithms into classes are indicated in Table 2.3. When presenting data on performance measures, we will 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

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

Table 2.3: Algorithm Classification

pessimistic algorithms in the following discussion. We discuss the results of our experiments in a manner organized by workload.

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

In our first set of experiments, we used a workload that consists of 80% searches and 10% each of inserts and deletes. The use of an equal proportion of random inserts and deletes ensures that a negligible number of splits and merges take place. The presence of relatively few updaters and a small number of splits creates a low data contention situation, and we use 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 shall compare the performance of alternative B-tree locking algorithms at various resource levels starting with a single CPU and disk.

High Fanout Tree Experiments

The throughput curves for the case with a single CPU and disk are shown in Figure 2.5. The buffer pool in this experiment has 200 pages, or about 75% of the B-tree size. It is seen from the figure that the throughputs for all algorithms are only slightly greater at higher MPLs than at an MPL of 1. This is because there is only one CPU and disk, and the disk rapidly becomes 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 2.5 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 model used by LY-ABUF causes multiple copies of pages to exist in the buffer pool,

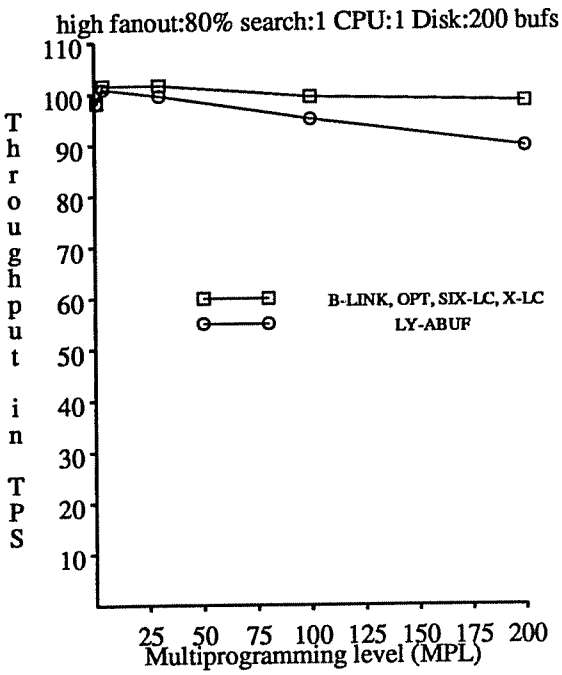


Figure 2.5: High fanout, single disk

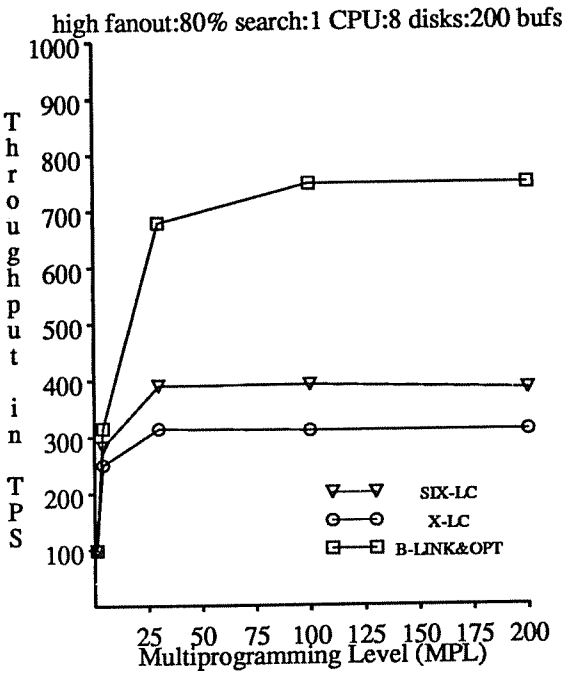


Figure 2.6: High fanout, 8 disks

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 (i) a requested buffer page is not in memory, and (ii) the page is in memory but is being modified by another transaction, in the same way; it must perform a disk I/O in both cases. In this experiment, case (ii) 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 2.6. Due to the additional system resources, the throughput of all algorithms increases to a higher level before leveling off than in Figure 2.5. In this case, the SIX-LC and X-LC lock-coupling algorithms only reach 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 this is because 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 utilize the disk completely at high MPLs. This suggests that the throughputs

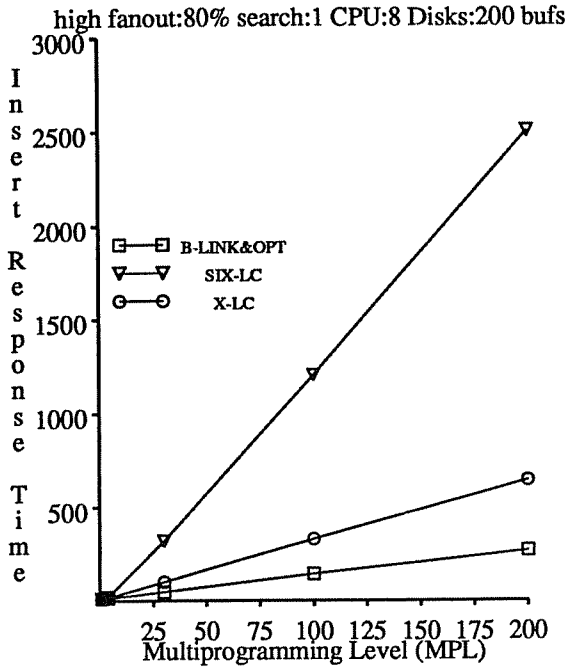


Figure 2.7: Insert times, 8 disks

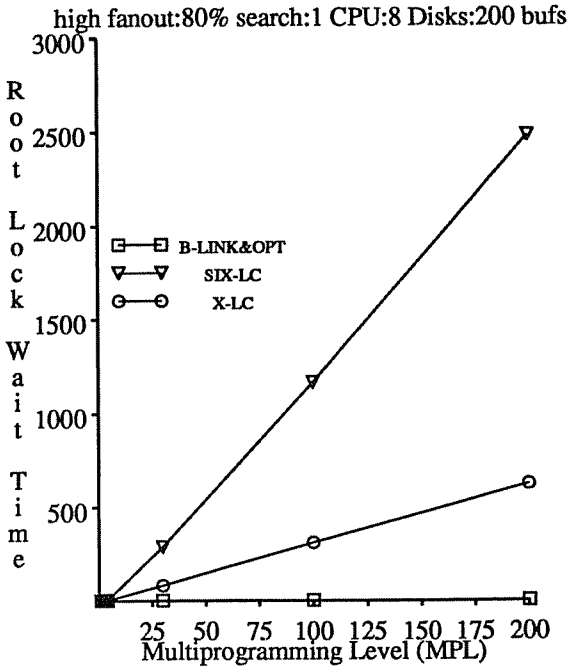


Figure 2.8: Wait at root (insert), 8 disks

for the pessimistic algorithms level off due to data contention rather than resource contention. It is evident from the corresponding response time curves (Figure 2.7) and lock waiting times at the B-tree's root page (Figure 2.8) 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 2.9 are very close to the corresponding insert response times in Figure 2.7. On the other hand, in the SIX-LC algorithms, the response time for searches increases only slightly with MPL (Figure 2.9), while the response time for updaters increases steeply with MPL (Figure 2.8). 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 2.6) 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 [Bili85, John90a]; 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 2.10. Note how the pessimistic algorithms level off at around the same maximum throughput as in the 1 CPU and 8 disks case (Figure 2.6), 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 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

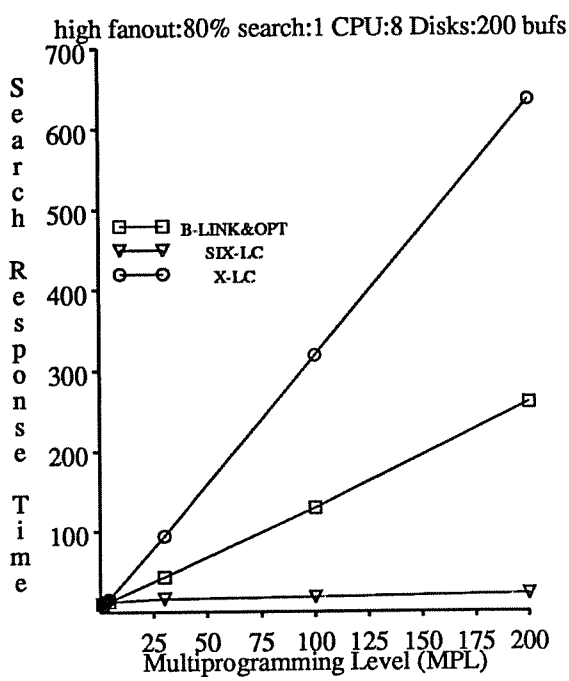


Figure 2.9: Search times (msec), 8 disks

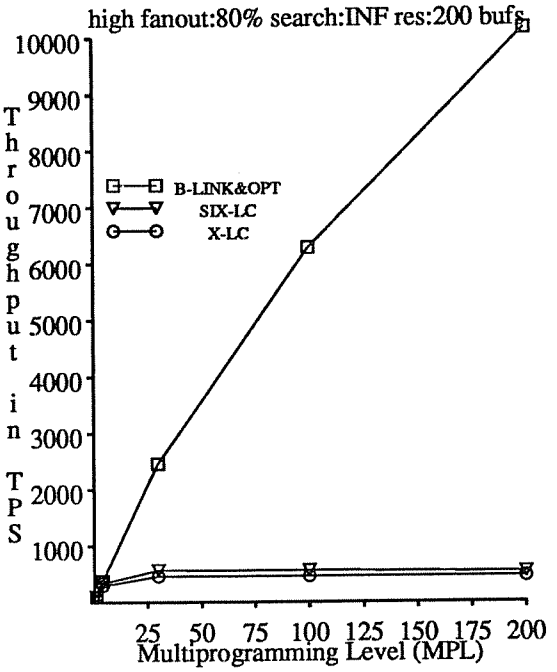


Figure 2.10: High fanout, ∞ resources

earlier MPLs than the disks did in the disk-bound experiments, as would be expected. However, the qualitative results 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 since 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).

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 6 levels (7 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.

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 2.11, 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

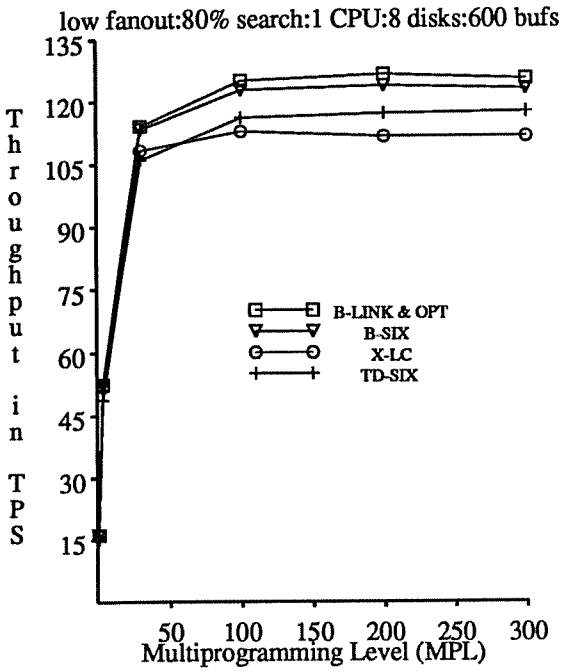


Figure 2.11: Low fanout, 8 disks

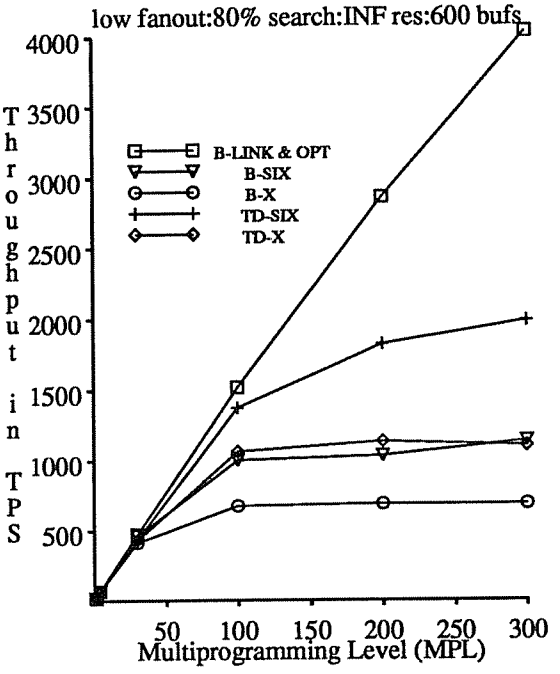


Figure 2.12: Low fanout, ∞ resources

optimistic and B-link algorithms, unlike in the high fanout tree (Figure 2.6). This is because the bottleneck 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 to note is that the peak throughput for the top-down algorithms in Figure 2.11 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 in Figure 2.11. 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, since 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 since the leaf node hit rate dominates (the number of leaf nodes is approximately four-fifths of the total number of nodes).

Figure 2.12 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 2.12 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, since searches are in the majority, the SIX-locking algorithms outperform the X-locking algorithms.

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, as 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 shall characterize this restart behavior in the next set of experiments.

2.4.2 Experiment Set 2: High Data Contention, Growing Tree

To further study 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.

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, since there is again not much difference between the algorithms. Figure 2.13 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 shall comment on three important aspects of these differences.

Firstly, the optimistic algorithms perform worse than the B-link algorithms here. The reason is that the optimistic algorithms are only able to utilize a maximum of 80% of the disk resources due to data contention under this workload, while the B-link algorithms 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.

Secondly, we notice in Figure 2.13 that the algorithm TD-OPT achieves a peak throughput higher than B-OPT. This is because 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. Since 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[7]. 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,

[7]The first pass in TD-OPT is done by lock-coupling with S 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 S locks are compatible with both.

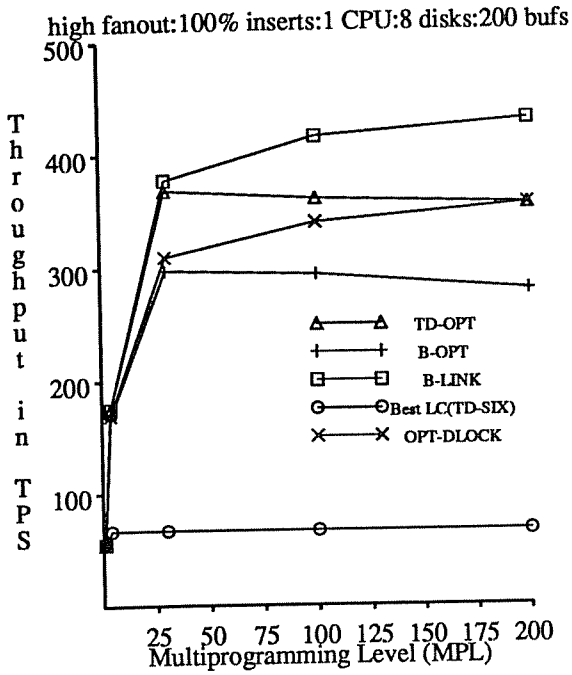


Figure 2.13: High fanout, 8 disks

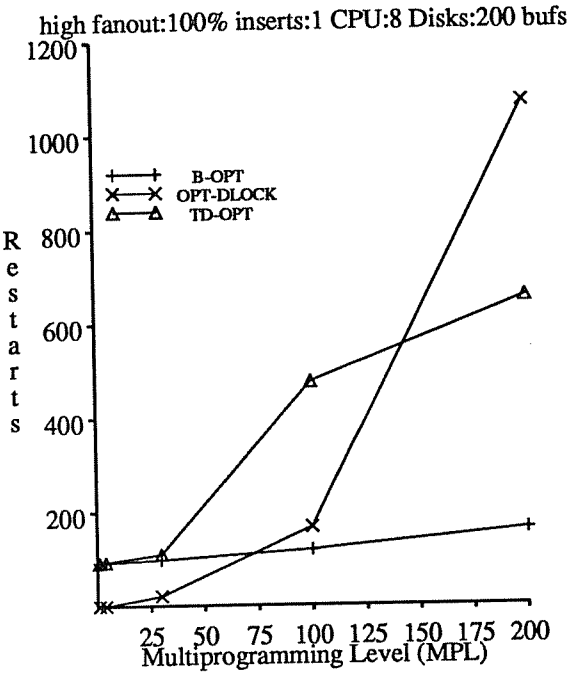


Figure 2.14: Restarts (per 10,000 ops.)

while leading to less waiting time at the root, could also increase the number of restarts since overtaking will allow more than one transaction to reach the same full node. A look at the restart counts for the TD-OPT and B-OPT algorithms (Figure 2.14) indeed shows that TDOPT 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.

Thirdly, we find that the throughput of OPT-DLOCK increases quickly at low MPLs and then more slowly at high MPLs. Unlike the other optimistic algorithms, however, its throughput does not quickly saturate. The reason for this behavior is the difference in OPT-DLOCK's handling of restarts. Since the conflict level is high, OPT-DLOCK performs many restarts (Figure 2.14). 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 2.15), OPT-DLOCK performs better at high MPLs

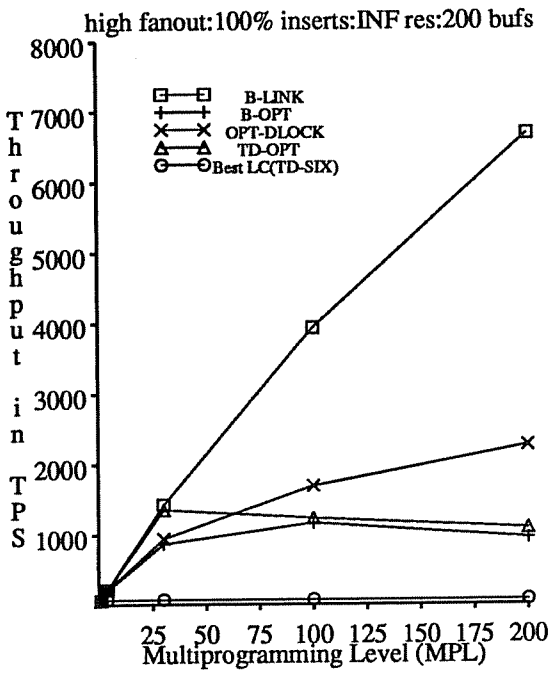


Figure 2.15: High fanout, ∞ resources

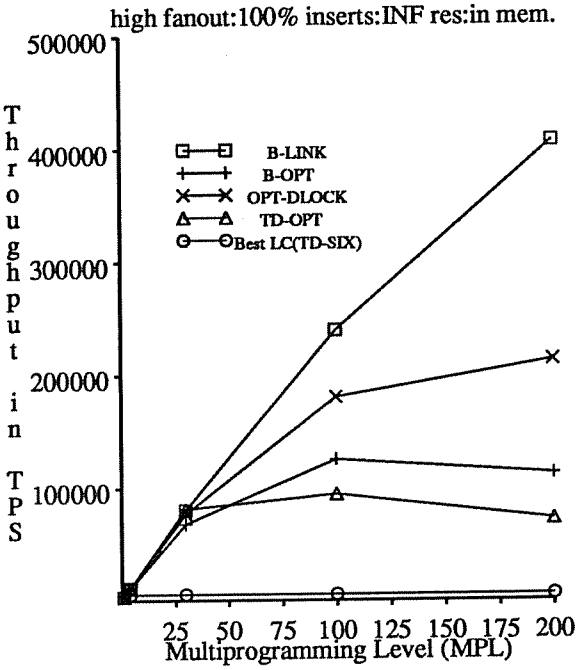


Figure 2.16: High fanout, in-mem. tree

than the other optimistic algorithms since its restarts become more inexpensive due to the increase in the number of CPUs from 1 (in Figure 2.13) 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 2.16. We find that TD-OPT performs worse than B-OPT here due to the greater number of restarts, as the overhead of a restart is now comparable to the response time of a successful tree operation itself. The number of restarts here are essentially the same as in the earlier disk bound case (Figure 2.14).

Low Fanout Tree Experiments

The second subset of experiments using the 100% insert workload is performed on a low fanout tree. Just as in the low fanout experiments of experiment set 1, we first consider configurations in which the buffer pool contains 600 pages,

The 100% insert throughput curves with 1 CPU and 8 disks are given in Figure 2.17. 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

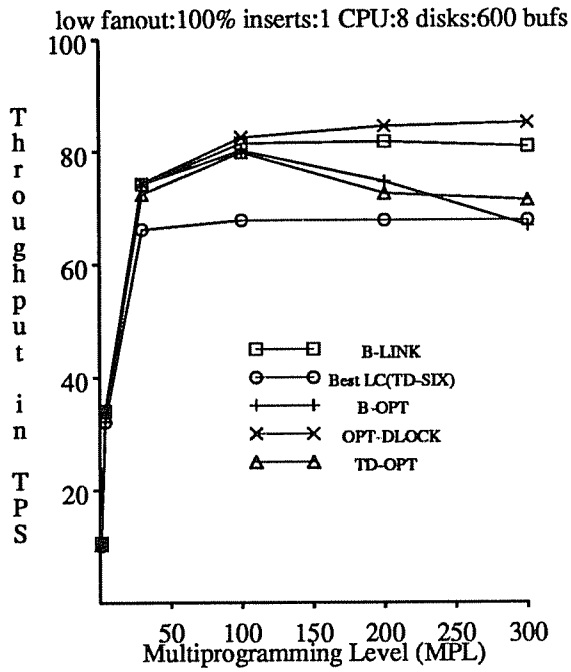


Figure 2.17: Low fanout, 8 disks

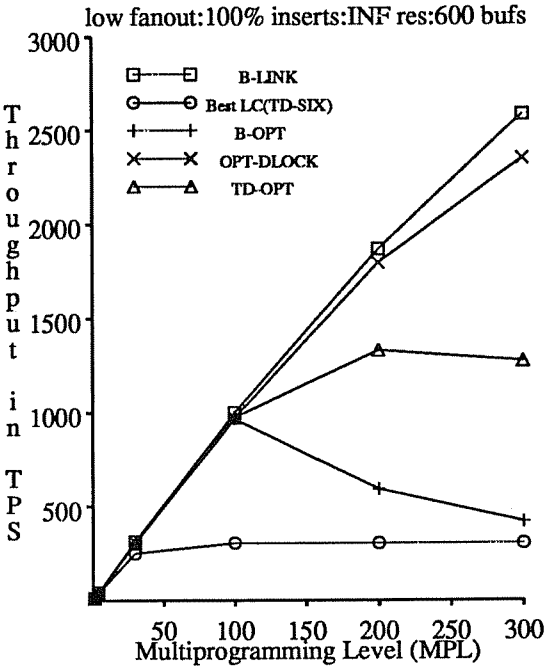


Figure 2.18: Low fanout, ∞ resources

drastically (to just 100 from the maximum of 1200 in Figure 2.14) 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 2.14. 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.

To further illustrate the above concepts, the results for the infinite resources case (Figure 2.18) show that 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 discussed in Section 4.1.1.

In the TD-OPT algorithm the loss in throughput occurs for a slightly different reason. Recall that 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 which 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 that this was indeed the case 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. Both optimistic algorithms achieve a final 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 2.6); 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 2.19. (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.

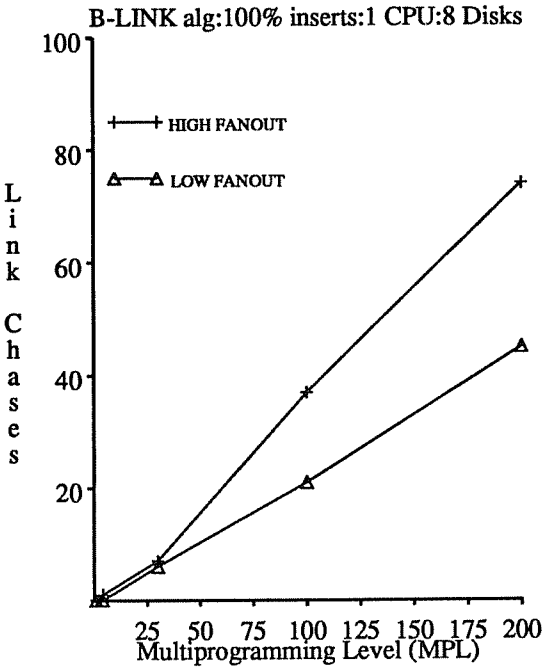


Figure 2.19: Link-chases (per 10K ops.)

Summary

To summarize the results of this section, 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 again worse 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 a bit better than 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. Since 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 suitable alternative. We are therefore 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.

2.4.3 Experiment Set 3: Extremely High Data Contention

In our third and final set of experiments, we use a workload that consists 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. The appends create extremely high contention for the few right-most leaf nodes in the tree. The searches are random, however, and do not interfere with the appends that are 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 have a situation where two operations with widely varying response times are interacting in the system.

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.

High Fanout Tree Experiments

The first subset of experiments are 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 2.20), the B-link algorithms perform better than all of the other algorithms, and their throughputs continue 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 2.21). 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 2.22). Also, notice that the optimistic algorithms perform an increasing number of restarts. Due to the very fast response

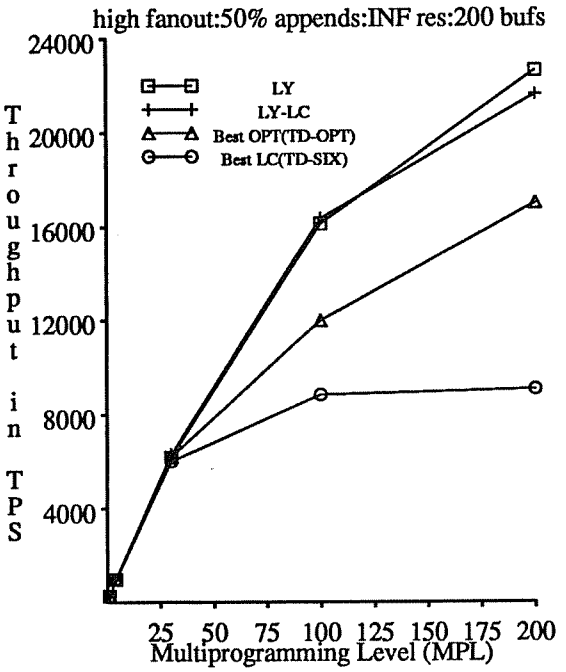


Figure 2.20: High fanout, ∞ resources

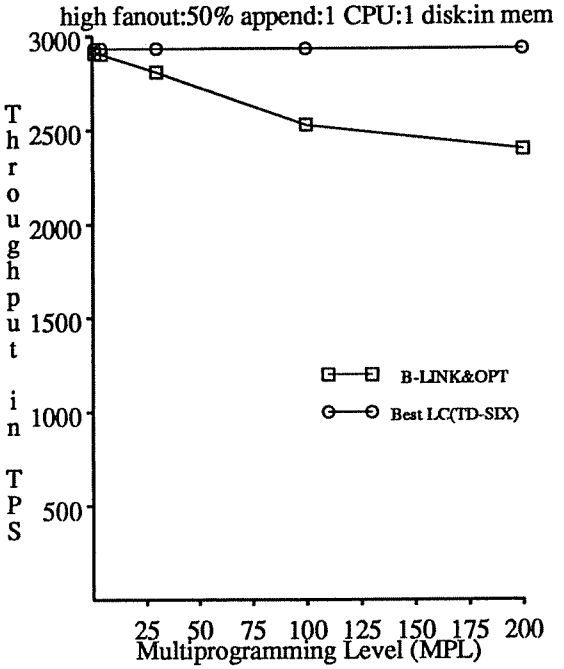


Figure 2.21: High fanout, in-mem. tree

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.

We find from Figure 2.22 that the B-link algorithm's rate of *increase* of link-chases at high MPLs is about half of that at lower MPLs. Since 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, and a reduction in the number of splits will reduce link-chases. 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[8]. The reduction in the number of splits occurs for the optimistic algorithms TD-OPT and B-OPT as well, but these optimistic algorithms are unable to take advantage of it because of the bottleneck that forms at the root. Unlike TD-OPT and B-OPT, which saturated at throughput values close to those of their pessimistic counterparts, we found that OPT-DLOCK exhibited thrashing behavior in the append experiments due to unbounded restarts; however, OPT-DLOCK was still better than TD-OPT and B-OPT throughout.

[8]A strategy that causes the 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 will cause the page occupancy to be around 69% [Yao78]. Thus, randomization leads to fewer splits.

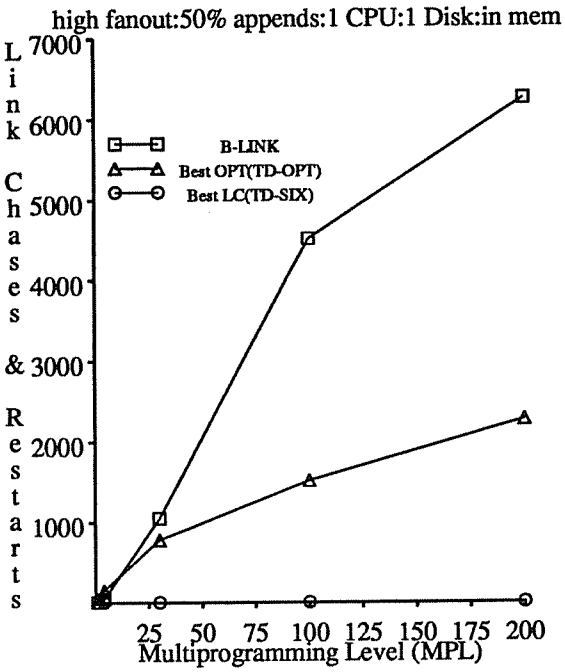


Figure 2.22: Overhead (per 10,000 ops.)

Low Fanout Tree Experiments

The qualitative results for the low fanout tree are mostly 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 S locks while searching for the parent of a leaf node. Still, both the LY and LY-LC algorithms performed an order of magnitude better than the rest of the algorithms.

Summary

To summarize, 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 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 either cause 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 performed reasonably close to the LY algorithm and much better than all non-B-link algorithms.

2.5 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 upon 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. Since 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 whether there is any actual data contention or not. 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 than algorithms that restart based on leaf node occupancy in most situations.

4. The extent to which an overhead like 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 2.15 and 2.18).

The only algorithms that were seen to perform 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 (for example) 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 a high contention situation like our append experiments. A final important observation is that the B-link algorithms performed the best in both high and low fanout trees. This leads us to conclude that B-link algorithms will also perform well for trees that have 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, we found that while the throughput of the pessimistic algorithms was close to that in the corresponding FCFS case (Figure 2.6), the optimistic and B-link algorithms reached a peak throughput that was 70% higher than in Figure 2.6.

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.

2.5.1 Side-Branching Technique

The *side-branching* solution proposed in [Kwon82] is a modification to the SIX locking Bayer-Schkolnick algorithm, B-SIX. Updaters in this algorithm perform the allocation of new pages and the copying of 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 since, 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 2.9 for the SIX-LC algorithm class).

2.5.2 The mU Protocol

An interesting algorithm called the mU protocol was proposed in [Bili87]. 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 S 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 – since 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 since 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.

2.5.3 ARIES/IM Algorithm

An algorithm for high-concurrency index management was described in [Moha89]. 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 S 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. In our experiments, 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). Since 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 2.15 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 2.18. 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 in [Moha89] 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.

2.6 Comparison with Related Work

An approximate analysis of the Bayer-Schkolnick algorithms was included in [Baye77]. The formulas provided there 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 [Bili85] 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 [Sama76], the B-SIX algorithm, the side-branching algorithm [Kwon82], and the mU algorithm [Bili87]. 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 [John90a]. 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 will become 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 (eg., 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 S 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 thesis, the authors of [John90a] have extended their work. In their extended study [John90b], they now handle SIX locks and analyze 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). They found that the LY algorithm still performed the best among all of the algorithms considered. They also found that 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. Since 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 theirs) 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. Another 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 (see Figures 2.6 and 2.9).

The analytical model in [John90a, John90b] 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, [John90b] modeled resource con-

tention using a service time dilation factor and found that (i) under very high resource contention there was no difference between the various algorithms and (ii) 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 2.21). [John90b] also showed that LRU buffering will hold the highest levels of the tree in memory. While our results agree with theirs, 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.

2.7 Conclusions

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 further shown 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. Before using B-link algorithms in real systems, however, recovery strategies have to be designed for them, and a way has to be found to handle variable length keys. Recovery strategies for B-link algorithms have been proposed recently [Lome91].

In this study we have considered only one aspect of concurrency control on B-tree indices, namely, transactions that perform single B-tree operations. When B-tree operations are part of larger transactions, there are several strategies available for holding long term locks for recovery

purposes. For example, a transaction could hold extended locks on index leaf pages, on individual slots in the leaf pages, or on individual record ids or key values. These lock holding strategies are also closely linked to recovery strategies; few comprehensive recovery strategies have been proposed for B-trees. Furthermore, special types of queries such as range scans may interact differently with long term lock holding and recovery strategies than single operation transactions would.

Finally, another important facet of B-tree concurrency control and recovery involves the problem of building a B-tree index on a relation while the relation is being concurrently accessed by other transactions. This problem must be addressed in order to move database management systems toward fully on-line operation. We will turn our attention to the problem of on-line index construction in the next chapter.

Chapter 3

On-Line Index Construction Algorithms

3.1 Introduction

As we mentioned earlier, the coming explosion in database sizes will necessitate the scaling up of all the algorithms used in a DBMS, including the class of database utilities. These utilities are typically used for index construction, database reorganization, and checkpointing. On-line checkpointing has been discussed in [Rose78, Pu85, Pu88], and concurrent database reorganization was addressed in [Salz91, Sock78, Sock79, Sode81a, Sode81b, Ston88]. In this chapter, we address the open problem of on-line index construction.

A need arises for the construction of a new secondary index on an attribute of a relation when it is discovered that a significant proportion of the (current or future) queries on the relation could be executed more efficiently using an index on this attribute. Current DBMSs typically perform index construction in an off-line manner (i.e., by locking the relevant relation). This approach is not suitable for building an index for a very large relation, however, as index construction requires a relation scan and scanning a 1-terabyte table may take days. It is almost sure to be unacceptable to exclude updates for such an extended period of time. In fact, leading researchers have identified the problem of on-line index construction for very large databases as an important open research problem [Dewi90, Silb90].

In this chapter, we present several algorithms for on-line index construction. All of the algorithms presented scan the relation (copying out index entries) concurrently with updaters, while somehow keeping track of the updates that take place during the scan; they then combine these updates with the scanned entries before registering the index in the system catalogs. The algorithms differ in the data structures used for storing the concurrent updates, their strategies for combining these updates with the scanned entries, and finally, in the degree of concurrency allowed

following the scan phase of the index construction process. Since B-trees are the most common dynamic index structure in database systems, we will focus here on the on-line construction of B-tree indices. However, the techniques presented here can be easily extended to design on-line algorithms for other types of index structures as well.

The rest of this chapter is organized as follows. In Section 3.2, we describe the basic steps involved in building a B-tree index. We describe a simple off-line index construction algorithm in Section 3.3. After describing index updates in Section 3.4, we discuss various on-line index construction algorithms in Sections 3.5, 3.6 and 3.7. Section 3.8 describes related work that was done in parallel with the work described here. Finally, Section 3.9 summarizes the key results of this chapter. Correctness proofs for our on-line index construction algorithms are outlined in the Appendix.

3.2 Primitives and Data Structures

In this section, we describe some basic primitives and data structures that will be used later by all of the proposed index construction algorithms. We assume that the tuples contained in a relation are stored as records that each have a unique record identifier (rid).

Constructing a B-tree index from a relation involves three basic steps. The first step involves scanning the relation and collecting the (key, rid) entries that are needed to build the index. In the second step, the entries collected in the first step are sorted to produce a linked-list of leaf pages of the index. The third and final step involves creating the non-leaf pages in a bottom-up fashion from the leaf page list created in the previous step. These three basic steps are implemented by the pseudo-code functions *Extract_keys*, *Sort*, and *Make_index* respectively in Figure 3.1. A brief description of these functions follows.

Extract_keys: This function reads the pages of the relation one after another, acquiring short term Share locks on the relation pages if necessary (i.e., unless the relation itself has been locked in Share mode). The appropriate (key, rid) entries from a page are copied into a heap file which is the output of this function.

Sort: This procedure sorts a heap file of (key, rid) entries into increasing key order using a method like the (two-phase) sort-merge algorithm described in [Shap86]. A heap file is first scanned, producing runs of size $\sqrt{N}$ pages where N is the number of pages in the heap file. Next, these runs are concurrently merged. This causes at most 4N I/Os (2N I/Os more than an in-memory sort), and it works as long as $\sqrt{N}$ pages of memory are available for use by the sort [Shap86]. If the file to be sorted is actually a sequence of updates of the form (key, rid, insert/delete) then only the last

entry in the input determines the state of a particular (key, rid) pair and needs to be retained. One way of eliminating duplicates in this manner is for the *Sort* routine to tag all entries in the input file with a number that indicates their relative order in the input, and use these tags later while eliminating duplicates, retaining only the latest entry for a particular (key, rid) pair. Duplicate elimination is done during step 3 of the figure.

Make_index: This procedure takes as input two lists of (key, rid) pairs that are each sorted in increasing key order; at least one of these lists is required to be non-empty. If one of the input sorted lists is empty, the non-empty sorted list of pages is used directly as the list of leaf pages, and the nonleaf index pages are built on top of them in a bottom-up fashion (ending with the creation of a single root page). If both input lists are non-empty, however, this function merges them in increasing key order to form the leaf pages of the index. During the merge, duplicates are eliminated concurrently, and specially marked deleted entries are matched with their corresponding inserts, if any[1]. Note that the construction is accomplished in one pass from left to right through the newly created leaf pages; as each new leaf page is generated, an index entry at the next level is generated simultaneously.

Apart from the above procedures, we also assume that for every attribute of a relation the system catalog contains the entries 'Phase', 'Index' and 'U' (see Figure 3.2). These catalog entries are used for communication between the updaters and the index construction process. Phase[R, A] stores information about the state of the index on attribute A of relation R, and Index[R, A] points to the corresponding index if it exists. Finally, U[R, A] points to a list storing the concurrent updates, and is used in list-based on-line algorithms.

Having described the basic index utilities and shared data structures, we now describe a simple off-line strategy for index construction that works by locking the entire relation.

3.3 Off-Line Algorithm

The simplest way to construct a new index on a relation would be to lock the relation in Share mode, build the index, and then release the lock. Updaters are assumed to hold an Intention-exclusive lock on the relation while modifying a page of the relation (à la the hierarchical locking scheme of [Gray79]) and would therefore be unable to execute concurrently with the index building process. On the other hand, readers (which only acquire an Intention-share lock on the relation) can access the relation's pages concurrently with the building process.

The pseudo code for this algorithm is given in Figure 3.3. The code is straightforward and builds

[1]The merge and duplicate elimination steps will be used in several of the on-line algorithms described later in the chapter.

```
function Extract_keys(R)
    begin
        R: Input relation
        H: Empty heap file
        step 1: foreach page in R do
                    begin
                        Lock page in share mode, if necessary
                        Append (key, rid) entries from page to H
                        Unlock page
                    end
        step 2: return H
    end

function Sort(H)
    begin
        H: Input heap file with N pages
        S: Final list of pages sorted by (key,rid)
        step 1: Scan H and produce sorted output runs of length √N pages
        step 2: Allocate one block of memory for each sorted run
        step 3: Concurrently merge all runs, eliminating duplicates,
                producing sorted entries, and appending the results to S
                While eliminating duplicates of a (key,rid) pair, the pair
                occurring last in the input file H is retained in the output
        step 4: return S
    end

function Make_index(S, T)
    begin
        S, T: Sorted files
        L: Initially empty linked list of leaf pages
        I: Initially empty B-tree index
        step 1: Merge S and T to produce the list of leaf pages, L. During merge,
                eliminate duplicates and discard matching insert/delete entry pairs
        step 2: Concurrently create the higher levels of the index I based on L
        step 3: return I
    end
```

Figure 3.1: Index Construction Primitives

```
Phase[R, A]
    State of the index on attribute A of relation R
    Possible values are:
        none        No index exists
        scan        Build process is in relation scan phase
        build       Build process is making the index
        catchup     Final clean-up phase of the build
        available   Index exists and can be accessed normally

Index[R, A]
    Pointer to the index that exists or is being built on attribute A of relation R
U[R, A]
    Pointer to the list of concurrent updates,
    while an index is being built on attribute A of relation R
```

Figure 3.2: Shared data structures for Index Construction

the index in three steps, using the utilities described in the previous section. As an optimization, the *Make_Index* step (step 4) could be combined with the *Sort* (step 3) to build the index with one fewer pass through the leaf pages. This optimization is possible since only one of the arguments to *Make_index* is nonempty and the top levels of the index can be built during the merge step of the sort (step 3 of *Sort* in Figure 3.1). We assume that such optimizations would be made in any implementation but we retain the present form of the code for clarity.

In the off-line strategy, the existing concurrency control mechanism will take care of conflicts between updaters and the index building process, so no modifications to the behavior of update transactions are needed. To the best of our knowledge, this is how most commercial database systems operate today. Due to the absence of concurrent updaters in the off-line algorithm, the index building process benefits in the following ways:

1. The building process faces no interference from updaters in terms of waiting for locks for relation pages. It also avoids the overhead of locking each page, but this is likely to be insignificant relative to the actual resources used by the index building process.

2. Since the relation is not modified during the index building time, the index building process does not have to incorporate any concurrent updates at the end to make the index consistent.

This algorithm, therefore, is the fastest way to build the index, but it will provide zero throughput for updaters. As we indicated earlier, the scan phase for a large relation is likely to be quite large (e.g., it may take days for a terabyte of data), and this phase will dominate the index building process. Thus, the off-line algorithm will be unacceptable to use for building indices on very large

```
Build process
    begin
        R: Input relation
        A: Attribute of R on which to build index
        H: Heap file
        L: List of leaf pages
        step 1:  Lock relation R in share mode
        step 2:      H = Extract_keys(R)
        step 3:      L = Sort(H)
        step 4:      Index[R, A] = Make_index(L, empty file)
        step 5:      Phase[R, A] = available
        step 6:  Unlock relation R
    end
```

Figure 3.3: Off-Line Algorithm

relations due to the lack of updater throughput for long periods of time. We plan to use the performance of this algorithm as a baseline for understanding the behavior of alternative on-line algorithms.

3.4 Concurrent Updates

One way to improve on the off-line approach is to allow updaters to proceed during the scan phase, somehow communicating their updates to the building process at the end of the scan phase. The scanned values and this list of updates can then be merged and made consistent. It is possible that the duration of the scan phase will be increased due to the presence of concurrent updates, but permitting updates during the index construction phase makes it attractive to use such on-line strategies. All of the on-line algorithms described in the rest of this chapter allow updaters to concurrently execute during the scan phase. The algorithms will vary in the data structures used to record concurrent updates, in the amount of concurrency allowed after the scan phase, and finally, in the strategies used for applying the concurrent updates to the scanned entries. We will further subdivide the on-line algorithms into list-based algorithms and index-based algorithms, depending on whether they use a list or an index for storing concurrent updates. In this section we describe when and how transactions perform index updates, and in the next two sections we describe the on-line index construction algorithms themselves.

3.4.1 Impact of Updates

If an index exists on an attribute of a relation, updates to the relation (in the form of inserting, deleting or modifying a record) can result in updates to the corresponding index. An index update

consists of three parameters: a key, an associated rid, and a flag depicting whether it is an insert or a delete. It is possible for a particular relation update (involving exactly one record) to result in zero, one or two index updates:

- When a record in a relation is modified without changing the value of the indexed attribute, no index update needs to be done.
- When a new record is inserted into a relation or an existing record is deleted, there is exactly one index operation that has to be done. The key value for the index update is the value of the indexed attribute of the inserted or deleted record, and the rid value is the rid of this record.
- When the value of an indexed attribute in a record is modified, it causes two updates to be done to the corresponding index: an index delete (with the key being the old attribute value) followed by an index insert (with the key being the new attribute value).

Update transactions are assumed to execute "correctly", and obey the following two rules: (i) they hold an Exclusive lock on a relation page while they are making changes to it, and (ii) they do not try to insert the same index entry (key/rid pair) twice successively without deleting it in-between (and vice versa). To ensure correct behavior of our on-line algorithms in a serializability sense, we also require that update transactions hold locks on record ids until they terminate (commit or abort). It should be pointed out that the page lock mentioned in (i) above need not be held until end of transaction; a short-term Exclusive page latch will suffice. All of our algorithms will also work (unchanged) in systems where page locks are held until commit or abort time. Update transactions that encounter an index construction process will perform the index updates corresponding to their relation updates using special code that depends on the type of on-line algorithm being used. We assume for simplicity that this special code will be executed immediately after the corresponding relation update, i.e., part of this code will be executed while still holding the Exclusive lock on the modified relation page. We will indicate later how to relax this restriction[2], though we will also argue that it may not be such a serious restriction.

3.4.2 Impact of Aborts

The above method of transaction execution takes care of situations like transaction abort in a straightforward manner. During a transaction abort (even if index construction is still going on),

[2]Note that this restriction applies only for index updates that occur during the on-line index construction phase. Index updates at other times can be handled either immediately after the corresponding relation update or grouped together at commit time.

undo operations will generate index updates in exactly the same manner as they do during normal operation. During index construction, these updates will be executed using the same special purpose code that was used by the transaction during its forward execution. However, it should be noted that these index updates may not be the exact inverse operations of the index updates performed during the forward execution. For example, it is possible that a transaction undoing a record delete operation may be assigned a different rid for the record, in which case the abort-time index update will be an insert with the same key but a different record id.

3.5 List-Based Algorithms

The list-based algorithms, like all on-line algorithms described in this chapter, allow updaters to concurrently operate on the relation during the scan phase. Updaters that execute concurrently with an index construction process store index updates (corresponding to their relation updates) in a special update-list. There is one such list for every index that is being built, and it is maintained as a heap file. The individual list-based algorithms differ from each other in their method of combining the list of concurrent updates with the scanned list of entries, and also in the amount of concurrency provided after the scan phase. We will begin our discussion of these algorithms by discussing three methods for combining the list of concurrent updates and the scanned entry list in order to produce a consistent index.

A simple way of combining the scanned entries with the update-list is to first build an intermediate index using the scanned entries alone, and then sequentially apply the update-list entries to this index like ordinary index inserts and deletes. A disadvantage of this strategy is that it may result in leaf nodes of the intermediate index being accessed more than once, especially if the number of entries in the update-list is large. If the amount of available buffer space is small enough so that leaf pages are typically paged out before being accessed again, such repeated accesses will result in an increased number of I/Os.

A second method of combining the scanned entries with the update-list is to build an intermediate index using the scanned entries (as above) but to then sort the update-list before sequentially applying its entries to the intermediate index. In such a strategy, the problem of additional I/Os will be solved since the leaf pages of the intermediate index will be scanned once from left to right[3]. Also, sorting the update-list has an added advantage: we can eliminate matching inserts and deletes during the sort, and thus avoid inserting entries into the intermediate index that have a corresponding delete entry later in the unsorted list (and vice versa).

[3]This strategy could be further optimized by grouping together all inserts from the sorted list that can be accommodated in a leaf page before making the next index traversal from the root page.

Name	How Updates Are Applied?	After Scan Phase
List-X-Basic	Sequentially apply from unsorted list	exclude concurrent updaters
List-X-Sort	Sequentially apply from *sorted* list	exclude concurrent updaters
List-X-Merge	*Merge* sorted list and scanned entries	exclude concurrent updaters
List-C-Basic	Sequentially apply from unsorted list	allow concurrent updaters
List-C-Sort	Sequentially apply from *sorted* list	allow concurrent updaters
List-C-Merge	*Merge* sorted list and scanned entries	allow concurrent updaters

Table 3.1: List-Based Algorithms.

A third method of combining the concurrent updates with the scanned entries would be to first sort the scanned entries, as well as the update-list entries, to produce two sorted lists. The index can then be built in a bottom-up manner using the *Make_Index* function (of Figure 3.1) with the two sorted lists as parameters. This merging approach is bound to be better than either of the sequential strategies if the number of operations performed in the sequential strategies is very large. An added advantage of merging is that the utilization of the pages in the final index can be strictly controlled, being kept at any desired (high) level of occupancy. In contrast, this occupancy can be strictly controlled only for the intermediate index in the sequential insert strategies, and the sequential inserts performed later could reduce the occupancy of leaf pages.[4] This effect will be significant when the number of sequential inserts is large.

Apart from the above, list-based algorithms can also vary in the amount of concurrency allowed **after** the scan phase. (Recall that all of our on-line algorithms will allow concurrent updaters during the scan phase.) The simplest strategy involves locking out updaters after the scan phase, and applying the concurrent updates using one of the three methods described above. This way, no concurrent updaters are allowed while the scanned entries are being combined with the updates that took place during the scan phase. If the amount of the time taken to apply these changes is significant compared to the scan time, however, locking out updaters could still lead to a noticeable loss of concurrency. To avoid such a loss, updaters can be allowed to execute concurrently during the scan phase also; this requires an appropriate strategy for merging in this second set of concurrent updates at the end. Such a strategy is quite a bit more complicated than the one that exclusively locks the relation after the scan phase; we will postpone the details of this strategy until later.

Based on the three possible ways of merging the update-list with the scanned entries, and the presence or absence of concurrent updaters after the scan phase, there are six possible variations of the list-based algorithms. Their names and classifications are given in Table 3.1. We also illustrate the three List-X-* algorithms in Figure 3.4, and the three List-C-* algorithms in Figure 3.5. Note that the List-X-* algorithms execute in two phases, the scan phase (in which updaters execute

[4]The expected leaf page occupancy of a B-tree built with random inserts is about 69% [Yao78].

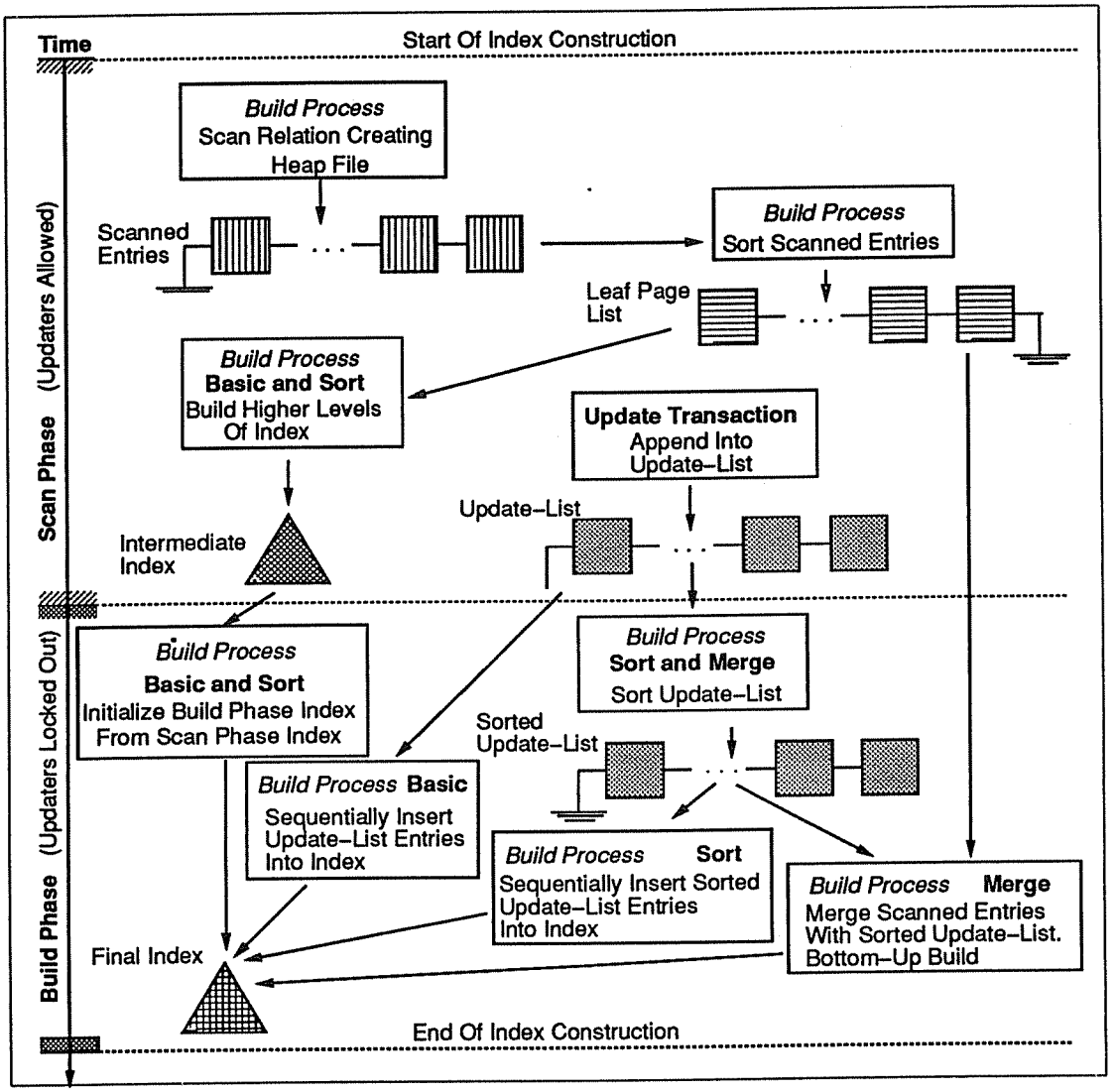


Figure 3.4: The List-X-* Algorithms

concurrently) and the build phase (in which updaters are locked out). The List-C-* algorithms execute in three phases, the scan, build and catchup phases, and updaters are allowed to execute concurrently in all three phases. We now describe each of the list-based algorithms of Table 3.1 in more detail.

3.5.1 The List-X-Basic Algorithm

In this algorithm (detailed in Figure 3.6), the building process executes in two phases, the *scan* and *build* phases, executing all steps except step 6a in the figure. In the scan phase, the building process first creates an empty update-list. It then proceeds to scan the relation, a page at a time, collecting index entries into a heap file. The heap file is sorted, and an intermediate B-tree index is

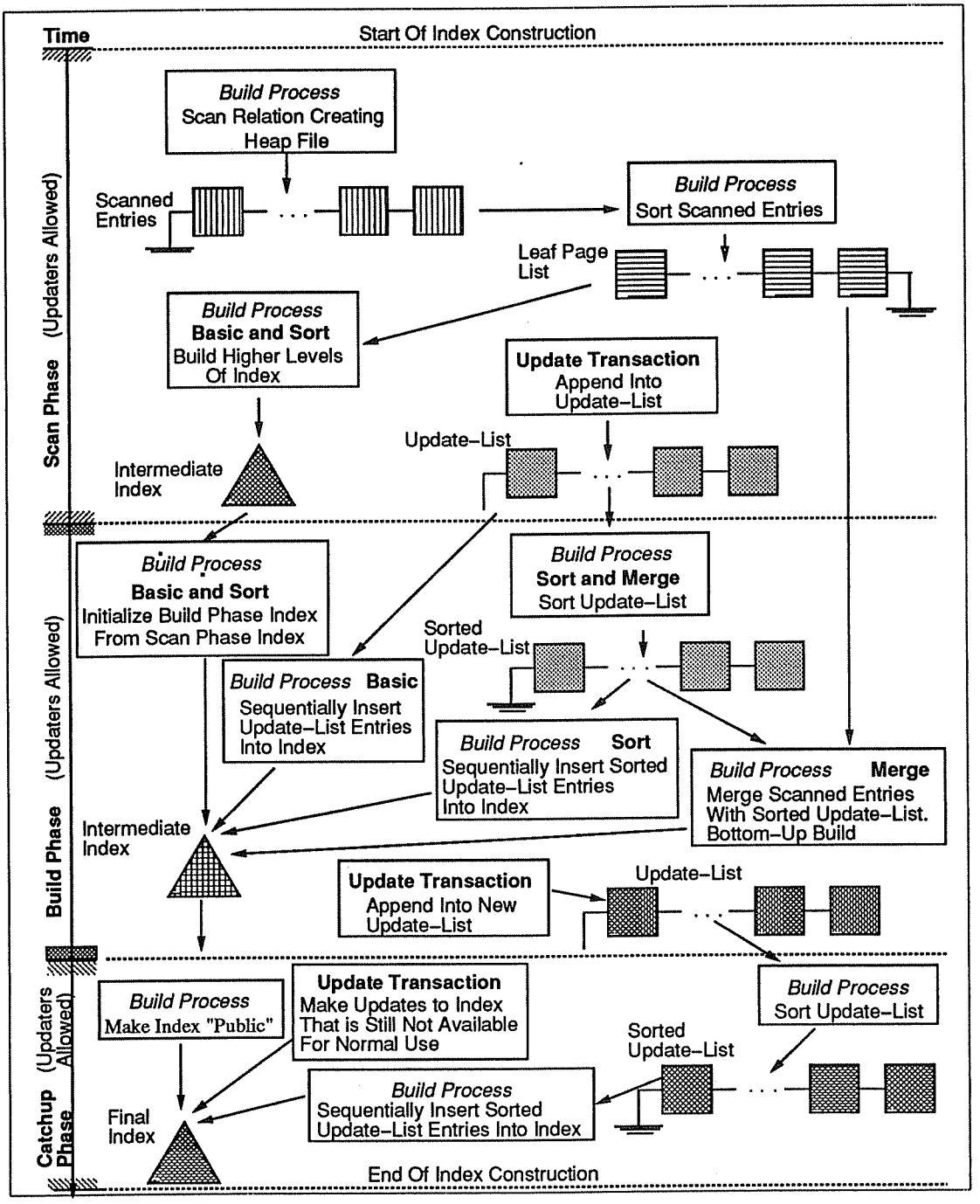


Figure 3.5: The List-C-* Algorithms

```
Build process
    begin
        R: Input relation
        A: Attribute of R on which to build index
        H: Heap file
        step 1:  U[R,A] = empty, Phase[R, A] = scan
        step 2:  H = Extract_keys(R)
        step 3:  H = Sort(H)
        step 4:  Index[R, A] = Make_index(H, empty)
        step 5:  Lock U[R,A] in exclusive mode
        step 6:      Phase[R, A] = build
        step 6a:     if algorithm is List-X-Sort then
                         U[R,A] = Sort(U[R,A])
        step 7:      foreach entry in U[R,A] do
                         case entry type of
                             insert: if entry not in Index[R, A], insert it
                             delete: if entry in Index[R, A], delete it
        step 8:      Phase[R, A] = available
        step 9:  Unlock U[R,A]
    end

Update transaction
    begin
        R: input relation
                . . . Normal processing . . .
        step n:  if Phase[R, A] = scan or build then
                         Lock U[R,A] in exclusive mode
                             if Phase[R, A] = scan then
                                 Append (key, rid, insert/delete) entry to U[R,A]
                             else Unlock U[R,A], goto step n
                         Unlock U[R,A]
    end
```

Figure 3.6: The List-X-Basic and List-X-Sort Algorithms

built on these data items. After this, the index building process locks the update-list in Exclusive mode and enters the build phase.

In the build phase, the building process applies the entries from the update-list one after another to the intermediate index, bringing it up to date. Applying an update-list entry (step 7 in Figure 3.6) to the intermediate index may involve either inserting an entry, deleting an entry, or no action at all. The last case could occur, for example, if the page on which a concurrent insert acts is scanned in the scan phase **after** the insert has already been done. In this case, no change needs to be made to the intermediate index on encountering the corresponding entry in the update-list, and an index traversal will find the appropriate entry to be already present in the intermediate index. An analogous scenario could occur for deletes. After all of the entries in the update-list have been applied to the intermediate index, the lock on the update-list is released and the index is made

available for normal use.

Unlike the off-line case, where the code for update transactions did not change due to the presence of an index building process, updaters behave differently here in the presence of an index building process. In the List-X-Basic algorithm, an updater that finds an index building process in the scan phase will append an index update (corresponding to its relation update) to the update-list. Each entry in the update-list consists of three components: the key, the rid, and the type of operation (insert or delete). Each list append is done while holding an Exclusive lock on the list, so these entries are ordered according to when the list updates are performed. The special code needed for update transactions is illustrated by step n of the update process in Figure 3.6. For every update to a relation, there will be one such step executed for each attribute of the relation on which an index is being currently built[5]. During the time while the index construction process is in the build phase (steps 6 and 7 in Figure 3.6), updaters cannot proceed. However, assuming that the number of entries in the update-list will be much smaller than the size (in pages) of the relation being scanned, this phase should be much shorter than the scan phase.

Due to the concurrent execution of updaters in the scan phase, unlike the off-line algorithm, this algorithm is expected to provide a significant level of throughput for updaters. On the other hand, the time for building the index will be longer than in the off-line algorithm, due to both page-locking interference with updaters during the scan phase and the extra work needed to apply scan phase updates to the intermediate index.

3.5.2 The List-X-Sort Algorithm

The List-X-Sort algorithm is described in Figure 3.6 along with the List-X-Basic algorithm. The code for the update transaction in this algorithm is the same as for the List-X-Basic algorithm. However, the code for the build process includes a step for sorting the update-list entries (step 6a in Figure 3.6). As explained earlier, sorting the update-list before inserting its entries into the intermediate index (created in step 4) avoids repeated accesses to the same leaf page. Another potential advantage of the List-X-Sort algorithm is that its sort step provides an opportunity to match inserts with later deletes, if any, and vice versa.

It is possible that multiple insert and delete entries may be present in the update-list for the same (key, rid) pair. In this case, only the latest update entry needs to be applied to the intermediate index. This is because insert and delete entries for the same (key, rid) pair have to alternate in the

[5]In Figure 3.6, we do not show the logic for index updates for existing indices. In fact, an updater that was waiting for an Exclusive lock in step n of Figure 3.6 may find, after the lock is granted, that the index is indeed available for normal use. In this case, it must perform a normal index update, as indicated by the **goto** statement in the pseudo-code.

update-list, and it is the last entry that determines whether this (key, rid) pair should be present in the final index or not. The duplicate elimination step of the sort routine (step 3 of *Sort* in Figure 3.1) therefore retains only the latest index operation for a (key, rid) pair that occurs more than once in the update-list.

The first two list-based algorithms both use a two-pass strategy to build the index: first, they build an intermediate index from the scanned entries, and then they sequentially insert the entries from the update-list into this index.

3.5.3 The List-X-Merge Algorithm

The List-X-Merge algorithm merges a sorted list of the scanned entries with the sorted update-list to build the index in one pass. The pseudo-code for this algorithm is given in Figure 3.7. The code for the updaters is the same as that for the earlier algorithms (see Figure 3.6). Also, as in the two earlier List-X-* algorithms, the index building process works in two phases, scan and build, with concurrent updaters permitted only in the scan phase.

The key difference between the earlier algorithms and the List-X-Merge algorithm is that the *Make_Index* function, which was executed in the scan phase earlier (step 4 in Figure 3.6), has now been moved to the build phase (step 7 of Figure 3.7). Also, *Make_Index* takes two sorted lists as arguments here. Note the slight difference between the entries of the sorted update-list (U[R,A]) and the sorted scanned list(H). While the former contains information on the latest operation for a (key, rid) pair, the latter contains just (key, rid) index entries. The merge step in the *Make_Index* function (step 1 of *Make_Index* in Figure 3.1) now performs all of the actions that were done via the sequential inserts into the intermediate index in the two earlier algorithms. In particular, a (key, rid) pair in the sorted list of scanned entries will appear in the final index only if the entry for the same pair in the sorted update-list is not a delete entry.

In the three algorithms discussed so far, there is one major critical section (steps 5-9 in Figure 3.6 and steps 4-9 in Figure 3.7). If the list of updates is large, then this critical section may take a significant amount of time. During this time, updaters are locked out. It may be possible to increase updater throughput in these algorithms by replacing the one long critical section with several smaller ones, thus allowing concurrent updates even during the build phase. The three remaining list-based algorithms are modifications of the first three algorithms that allow concurrent updates both during and after the scan phase.

```
Build process
    begin
        R: Input relation
        A: Attribute of R on which to build index
        H: Heap file
        step 1:  U[R,A] = empty, Phase[R, A] = scan
        step 2:  H = Extract_keys(R)
        step 3:  H = Sort(H)
        step 4:  Lock U[R,A] in exclusive mode
        step 5:      Phase[R, A] = build
        step 6:      U[R,A] = Sort(U[R,A])
        step 7:      Index[R, A] = Make_index(H, U[R,A])
        step 8:      Phase[R, A] = available
        step 9:  Unlock U[R,A]
    end
```

Figure 3.7: The List-X-Merge Algorithm

3.5.4 The List-C-Basic Algorithm

In the List-C-Basic algorithm (detailed in Figure 3.8), the index building process executes in three phases, the *scan*, *build*, and *catchup* phases. The scan phase of this algorithm is identical to the scan phase of the List-X-Basic algorithm. As before, the relation is scanned, and an intermediate index is then built with the scanned entries. The build phase here is also similar to the build phase of the earlier algorithm, except that the exclusive lock is released as soon as the phase state has been changed. During the build phase, the updates that occurred during the scan phase are now applied to the intermediate index, just as in the earlier List-X-Merge algorithm.

Note that the behavior of the updaters (given in Figure 3.9) in the scan and build phases in this algorithm is the same as their behavior in the scan phase of the three earlier List-X-* algorithms (Figure 3.6). Also note that, at the end of the build phase in Figure 3.8, the intermediate index (Index[R,A]) is consistent[6] w.r.t. all of the scan phase updates, unlike the intermediate index at the start of the build phase. That is, at the end of the build phase (step 10 in Figure 3.8), Index[R,A] is consistent with respect to the state of the relation as of the start of the build phase (step 6 in Figure 3.8). This is because the inconsistencies introduced during scanning are removed due to the build phase's incorporation of all scan phase updates into the index. This index is still behind by one phase, however, so the build phase updates have to be applied to bring it up to date. This is done in the catchup phase.

One way of implementing the catchup phase would be by using a single critical section. With this approach, the build phase updates would be sequentially applied to the index during the

[6] A consistent index is one that correctly reflects the actual state of the corresponding relation as of some time.

```
Build process
    begin
        R: Input relation
        H: Heap file
        S: Temporary file
        step 1:  Phase[R, A] = scan, U[R,A] = empty list
        step 2:  H = Extract_keys(R)
        step 3:  H = Sort(H)
        step 4:  Index[R,A] = Make_index(H, empty)
        step 5:  Lock U[R,A] in exclusive mode
        step 6:      S = U[R,A], U[R,A] = empty
                     Phase[R, A] = build
        step 7:  Unlock U[R,A]
        step 7a: if algorithm is List-C-Sort then
                     S = Sort(S)
        step 8:  foreach entry in S do
                     case entry type of
                         insert: if entry not in Index[R, A], insert it
                         delete: if entry in Index[R, A], delete it
        step 9:  Lock U[R,A] in exclusive mode
        step 10:     S = U[R,A], U[R,A] = empty
                     Phase[R, A] = catchup
        step 11: Unlock U[R,A]
        step 12: S = NSort(S)
        step 13: foreach entry in S do
                     case entry type of
                         insert:  if marked delete entry present in Index[R,A] then
                                      Delete the marked entry
                                  else Insert this entry normally
                         delete:  if marked insert entry present in Index[R,A] then
                                      Delete the marked entry
                                  else Delete the normal entry that is present
        step 14: Phase[R, A] = available
    end
```

Figure 3.8: Build Process in the List-C-Basic and List-C-Sort Algorithms

```
Update transaction
    begin
        R: input relation
                . . . Normal processing . . .
        step n:  if Phase[R, A] = scan or build then
                            Lock U[R,A] in exclusive mode
                                if Phase[R, A] = scan or build then
                                    Append (key, rid, insert/delete) entry to U[R,A]
                                else Unlock U[R,A], goto step n
                            Unlock U[R,A]
                        else if (Phase[R, A] = catchup) then
                            case updater of
                                insert:
                                    if the entry is present in Index[R,A] then
                                        Add a specially marked insert entry
                                    else Insert (key, rid) into Index[R,A]
                                delete:
                                    if the entry is not present in Index[R,A] then
                                        Add a specially marked delete entry
                                    else Delete (key, rid) from Index[R,A]
    end
```

Figure 3.9: Updater in the List-C-Basic, List-C-Sort and List-C-Merge Algorithms

catchup phase without allowing concurrent updaters. In such an algorithm, the catchup phase would be similar to the exclusive build phase of the earlier List-X-Basic algorithm. It turns out, however, that we can devise a strategy for the catchup phase that allows concurrent execution of updaters along with the build process. In this strategy, the build process (described in Figure 3.8) enters the catchup phase after applying the scan phase updates to the intermediate index. This phase change occurs in a short critical section (steps 9-11 of Figure 3.8). During the catchup phase, the build process actually shares the index with concurrent updaters: i.e., when index building is in the catchup phase, updaters register their index updates directly in the intermediate index.

The process of incorporating the build phase changes is slightly different than incorporating the scan phase changes since (i) the build phase changes are applied starting from a consistent initial index (Index[R,A] at step 10 of Figure 3.8), whereas the scan phase changes are applied starting from a possibly inconsistent index built using scanned entries (Index[R,A] at step 6), and (ii) build phase changes are applied to the index while updaters are also concurrently operating on the same index. Because of these differences, the routine *NSort* (used in step 12 of Figure 3.8) is slightly different from the earlier *Sort* routine (Figure 3.1), in that it eliminates duplicates differently. If an even number of occurrences of a (key, rid) entry are found in the input file, *NSort* eliminates this entry altogether from the output file. If an odd number of entries (for the same key/rid pair) are found in the input file, however, the latest entry is kept in the output file. Recall that the

earlier *Sort* procedure **always** retained the latest entry. The reason that *NSort* removes the entry in the even case here is that, since we start with a consistent index, and since insert and delete entries have to alternate, the state for this entry after an even number of inserts and deletes is the same as in the initial consistent index (Index[R,A] at step 10 of Figure 3.8). Note that the build process code for applying these build phase updates (step 13 of Figure 3.8) differs from the code for applying scan phase updates (step 8).

The actions performed by updaters during the catchup phase are given in Figure 3.9. As mentioned earlier, updaters are allowed to register their updates directly in the intermediate index during the catchup phase (though the index is still not available for normal use). In this phase, special handling is required for certain deletes (those that do not find an entry to delete in the index) and certain inserts (those that find an entry already in the index). This is accomplished via the use of special marked index entries; these are removed when the index update for such a special entry is performed by the build process using the sorted update-list (step 13 of Figure 3.8), and they do not re-appear afterward. When the build process has completed all of its updates, no such special entries will remain in the index, at which point the index is made available for normal use (step 14 of Figure 3.8). We prove the above assertions in the Appendix.

3.5.5 The List-C-Sort Algorithm

The List-C-Sort algorithm differs from the List-C-Basic algorithm in that the build process sorts the update-list before inserting its entries sequentially into the intermediate index. This is indicated by the addition of an extra sort step for this algorithm (step 7a of Figure 3.8). The differences here are identical to the differences between the List-X-Basic and the List-X-Sort algorithms described earlier. The code for the List-C-Sort updaters is identical to that in the List-C-Basic algorithm (Figure 3.9).

3.5.6 The List-C-Merge Algorithm

The last list-based algorithm that we discuss is the List-C-Merge algorithm (Figure 3.10). This algorithm is derived from the List-C-Basic algorithm in the same way that the List-X-Merge algorithm was derived from the List-X-Basic algorithm earlier. For the build process, the key difference from List-C-Basic and List-C-Sort is that the intermediate index is built in one pass using both the scanned entries and the scan phase updates. As in List-C-Basic and List-C-Sort, concurrent updates are allowed in the build phase in List-C-Merge, and these updates are then applied to the intermediate index during the catchup phase. Notice that the code for the catchup phase in

```
Build process
    begin
        R: Input relation
        H: Heap file
        S: Temporary sort file
        step 1:  Phase[R, A] = scan, U[R,A] = empty list
        step 2:  H = Extract_keys(R)
        step 3:  H = Sort(H)
        step 4:  Lock U[R,A] in exclusive mode
        step 5:      S = U[R,A], U[R,A] = empty
                     Phase[R, A] = build
        step 6:  Unlock U[R,A]
        step 7:  S = Sort(S)
        step 8:  Index[R, A] = Make_index(H, S)
        step 9:  Lock U[R,A] in exclusive mode
        step 10:     S = U[R,A], U[R,A] = empty
                     Phase[R, A] = catchup
        step 11: Unlock U[R,A]
        step 12: S = NSort(S)
        step 13: foreach entry in S do
                     case entry type of
                         insert:  if marked delete entry present in Index[R,A] then
                                      Delete the marked entry
                                  else Insert this entry normally
                         delete:  if marked insert entry present in Index[R,A] then
                                      Delete the marked entry
                                  else Delete the normal entry that is present
        step 14: Phase[R, A] = available
    end
```

Figure 3.10: The List-C-Merge Algorithm

Name	How Updates Are Applied?	After Scan Phase
Index-X-Basic	Sequentially apply from leaf-page list	exclude concurrent updaters
Index-X-Merge	*Merge* leaf-page list with scanned entries	exclude concurrent updaters
Index-C-Basic	Sequentially apply from leaf-page list	allow concurrent updaters
Index-C-Merge	*Merge* leaf-page list with scanned entries	allow concurrent updaters

Table 3.2: Index-Based Algorithms.

this algorithm (steps 12-14 in Figure 3.10) is the same as that in the List-C-Basic and List-C-Sort algorithms (steps 12-14, Figure 3.8). The code for the updaters is also the same here (Figure 3.9).

3.5.7 System Log Versus Update-List

Instead of using a special update-list to store concurrent updates, the system log can be used for this purpose. The advantage of using the log would be that no changes are needed to the logic for concurrent updates[7], just as in the off-line algorithm. There are several disadvantages to using a log-based strategy to build an index on a large relation, however. Because the portion of the log to be processed may be large (since scanning may take days), it may be unreasonable to expect all of the log records written during the scan to be kept on-line. Moreover, only a small fraction of the log will be relevant to any particular index construction process, and this fraction will be distributed throughout the overall log. These disadvantages led us to instead use a special update-list in the algorithms presented in this section.

3.6 Index-Based Algorithms

The index-based on-line index construction algorithms use an index to store concurrent updates instead of the update-list used by the list-based algorithms. Like all of our on-line algorithms, the index-based algorithms allow updaters to execute concurrently during the scan phase. Like the list-based algorithms, the various index-based algorithms differ in their method of combining the scanned entries with the concurrent updates as well as in the amount of concurrency allowed after the scan phase. However, there are only four possible index-based algorithms. This is because there are no counterparts to the List-C-Sort and List-X-Sort algorithms in the index case, as the leaf pages of the index that will be created by concurrent updaters will already contain the keys in sorted order. The various possible index-based algorithms are listed in Table 3.2. In addition, we also illustrate the Index-X-* algorithms and the Index-C-* algorithms in Figures 3.11 and 3.12 respectively. As for the corresponding list-based algorithms, the Index-X-* algorithms execute in

[7]If the log of a record update does not always contain the value of all the attributes needed for building a multi-attribute index, some changes are necessary to ensure that sufficient information is always logged.

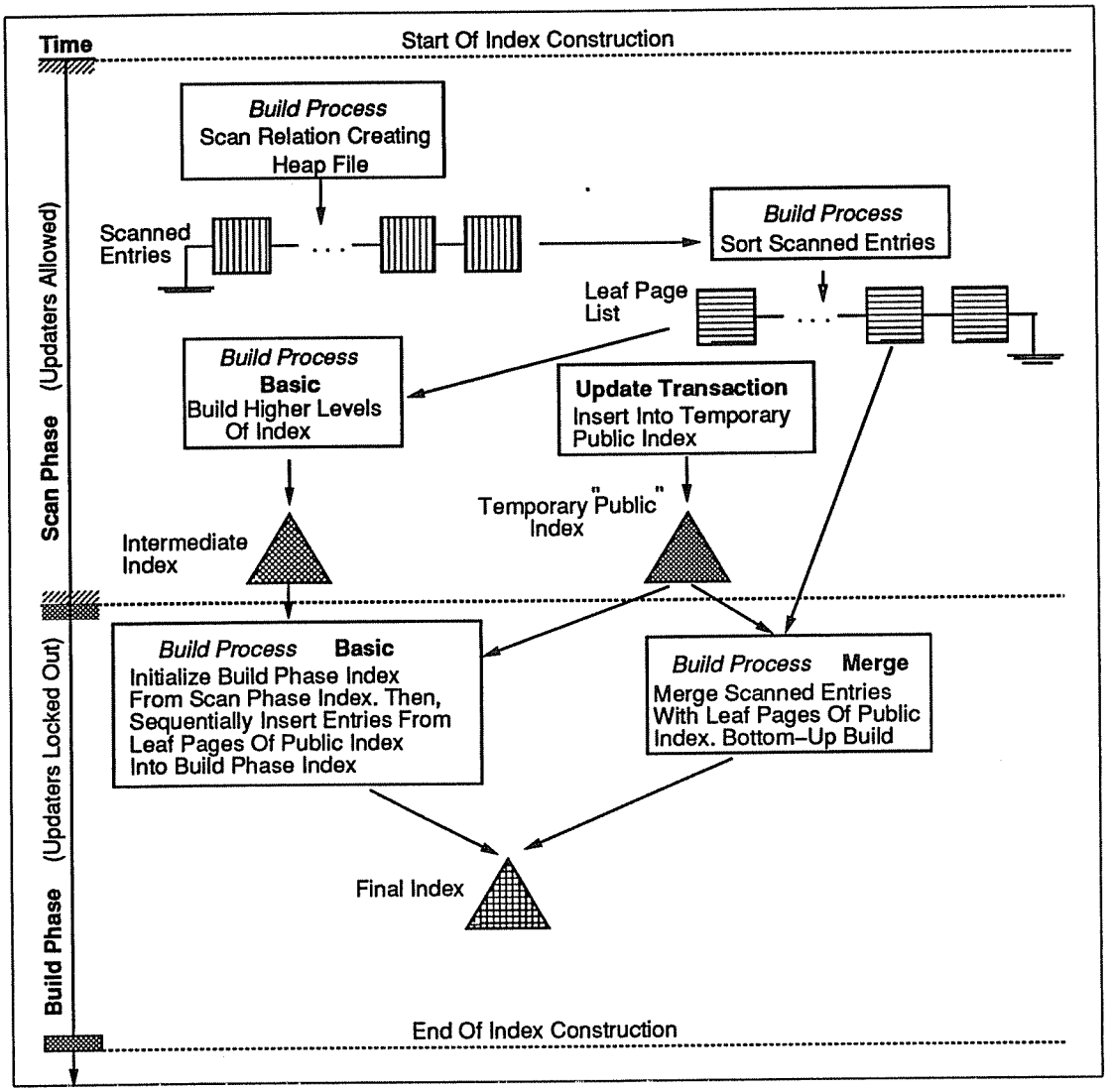


Figure 3.11: The Index-X-* Algorithms

two phases (scan and build) while the Index-C-* algorithms execute in three phases (scan, build, and catchup). We will now describe each of the four index-based algorithms in more detail.

3.6.1 The Index-X-Basic Algorithm

The pseudo-code for the build process and an update transaction in the Index-X-Basic algorithm are given in Figures 3.13 and 3.14 respectively. The build process of the Index-X-Basic algorithm executes in two phases, the scan and build phases, and is very much like the build process in the List-X-Basic algorithm (Figure 3.6). The difference here is the use of a "public" B-tree index (visible to concurrent updaters, Index[R,A] in Figure 3.13) to store the concurrent updates. As before, the build process scans the relation to gather index entries and builds a private intermediate index (T

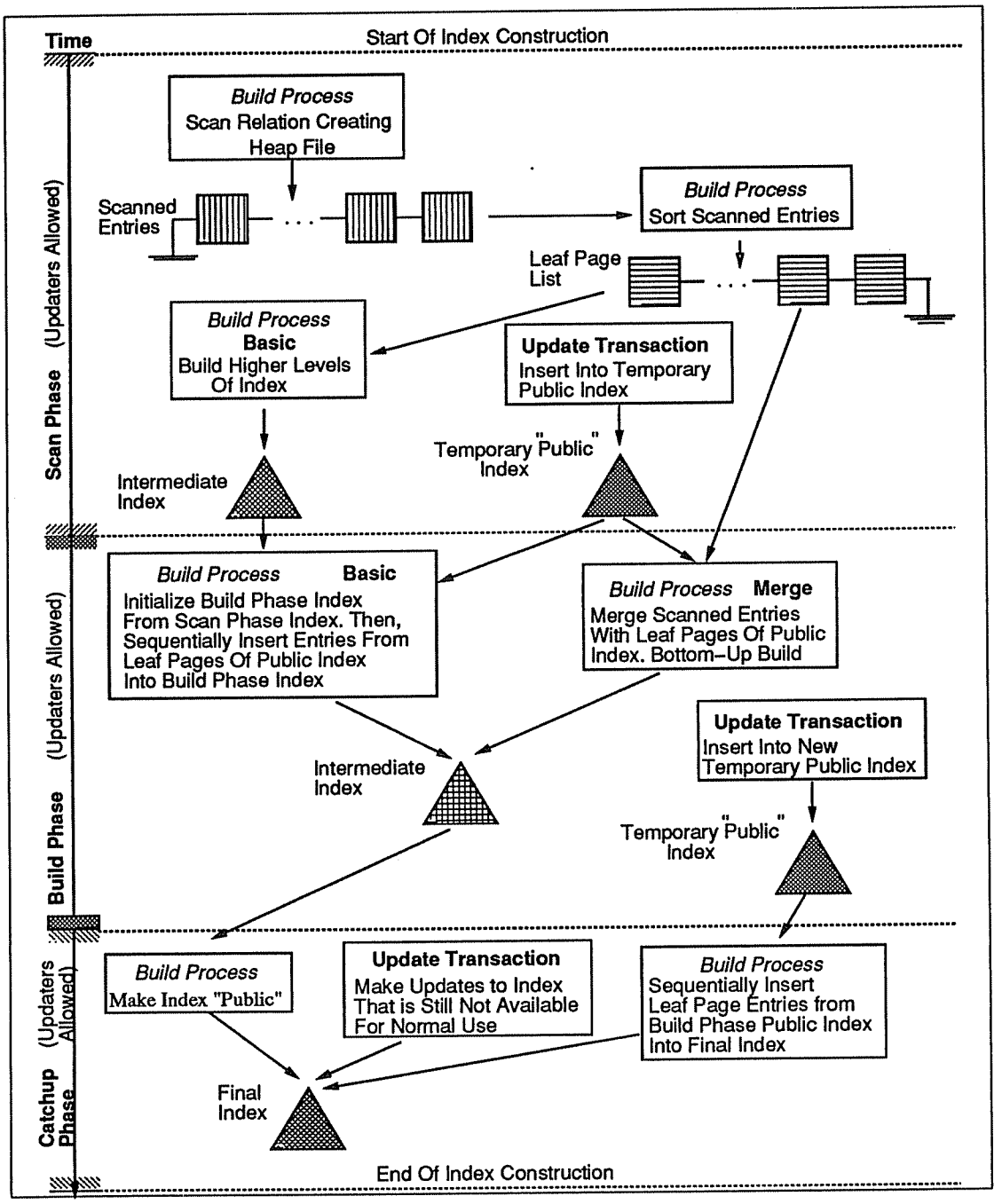


Figure 3.12: The Index-C-* Algorithms

in Figure 3.13) with these scanned entries. It then enters the build phase, locking the public index in Exclusive mode. The entries in the leaf pages of the public index (built by concurrent updaters) are applied to the privately built index, bringing it up to date. The build process here treats the leaf pages of the public index in much the same way that the List-X-Basic build process treated the list of updates. After all of the entries in the leaf pages have been applied to the intermediate index, the build process makes the index available for normal use (step 8), releasing the Exclusive lock.

Updaters finding that index construction is in the scan phase directly update the public index (step n of Figure 3.14). There are several key differences in the way that this public index is used, however, compared to how a normal (available) index is used:

- Updaters take a Share lock on the public index before accessing it in the scan phase, while this is not done in a normal index traversal. This share lock is used by updaters simply to exclude the build process from the index; updaters can still concurrently update the index. Concurrent updaters resolve their conflicts on individual index pages using the B-tree concurrency control algorithm described in Section 2.2.

- The public index cannot be used for searching like a normal index. In Figure 3.13, until the build process makes the final consistent index available in step 8, no searches can take place using the index.

- Updates to this public index add entries that are similar to those appended to the update-list in the list-based algorithms: each entry here has three components, a key, a rid, and the operation type (*insert/delete*); entries in a normal index do not have the operation type component.

- Updates to the public index differ from normal index updates in that they leave entries that keep track of the latest operation, if any, that took place for a particular (key, rid) entry in the index. This is to ensure that enough information is available about the concurrent updates to prevent inconsistent situations[8] from occurring when the scanned entries and the concurrent updates are combined. Any inconsistencies that are present in the private intermediate index

[8] An example inconsistency is as follows: During the scan phase, if a deleter finds a to-be-deleted entry to already be present in the public index, this entry must have been inserted after the start of the scan phase. However, it is possible for the index building process to have read the entry during the relation scan after it was inserted and before the impending delete. If the deleter now completely removes this entry from the public index, it may reappear later as a spurious entry (from the build process's scanned list) when index building completes, thus leaving an inconsistent index. Retaining a delete entry in the public index, and later matching this delete entry with the spurious entry in the scanned list, prevents this inconsistency in the final index.

```
Build process
    begin
        R: Input relation
        T: Temporary B-tree index
        A: Attribute of R on which to build index
        H: Heap file
        L: Head of a list of leaf pages
        step 1:  Phase[R, A] = scan, Index[R, A] = empty index
        step 2:  H = Extract_keys(R)
        step 3:  H = Sort(H)
        step 4:  T = Make_index(H, empty)
        step 5:  Lock Index[R, A] in exclusive mode
        step 6:      Phase[R, A] = build
                     L = List of leaf pages of Index[R, A]
                     Index[R, A] = T
        step 7:      foreach entry in L do
                         case entry type of
                             insert: if entry not in Index[R, A], insert it
                             delete: if entry in Index[R, A], delete it
        step 8:      Phase[R, A] = available
        step 9:  Unlock Index[R,A]
    end
```

Figure 3.13: Build Process in the Index-X-Basic Algorithm

built using the scanned entries alone will be removed by the index building process during the build phase (step 7 of Figure 3.13).

The behavior of updaters here can be best understood by noting that the leaf pages of the public index at the end of the scan phase have the same contents as would be obtained by storing these updates in a list and then sorting them, eliminating any duplicates by keeping only the latest entry. Recall that this is what the build process in the List-X-Sort algorithm accomplished by sorting the scan phase updates (using the *Sort* function of Figure 3.1) in the exclusive build phase. In the Index-X-Basic algorithm, this work is done by updaters using the public index. The build process in the Index-X-Basic algorithm thus performs less work in the critical section than the List-X-Sort algorithm. Updaters, however, perform more work in the Index-X-Basic algorithm, as an index insert is bound to take more time than list append (especially if the leaf pages of the public index do not all fit in memory).

3.6.2 The Index-X-Merge Algorithm

The Index-X-Merge algorithm is illustrated in Figure 3.15. The build process again executes in two phases, and is very much like the corresponding List-X-Merge build process (Figure 3.7). The only differences are (i) the use of a public B-tree index in lieu of the update-list used earlier, and (ii) the

```
Update transaction
    begin
        R: Input relation
                . . . Normal processing . . .
        step n:  if Phase[R, A] = scan or build then
                            Lock Index[R, A] in share mode
                                if Phase[R, A] ≠ scan then
                                    Unlock Index[R, A], goto step n
                                else case updater of
                                    insert:  if delete entry present in Index[R, A] then
                                                 Replace the entry by an insert entry
                                             else Add an insert entry
                                    delete:  if insert entry present in Index[R,A] then
                                                 Replace the entry by a delete entry
                                             else Add a delete entry
                            Unlock Index[R, A]
    end
```

Figure 3.14: Updater in the Index-X-Basic Algorithm

absence of the sort phase (step 6 of Figure 3.7) that was necessary for merging in the list of updates in the earlier algorithm. The reason for difference (ii) is that the leaf pages of the public index already contain the keys in sorted order and, since the leaf pages are linked from left to right, it can be directly used as an argument to the *Make_index* procedure (step 6 of Figure 3.15). The code for update transactions is exactly the same as that for the Index-X-Basic algorithm (see Figure 3.13).

The preceding index-based algorithms do not allow concurrent updaters after the scan phase. We now describe their *-C-* counterparts, which do allow concurrent updaters throughout the entire execution of the index building process. The next index-based algorithm that we discuss is the Index-C-Basic algorithm.

3.6.3 The Index-C-Basic Algorithm

The build process of the Index-C-Basic Algorithm is described in Figure 3.16, and the corresponding update transaction code is given in Figure 3.17. The build process of this algorithm is similar to the build process in the List-C-Basic algorithm (see Figure 3.8). The build process here also executes in three phases, the scan, build and catchup phases. The scan phase here is similar to the scan phase of the earlier Index-X-Basic algorithm. In this phase, the build process first scans the entries from the relation and then builds an intermediate index using the scanned entries. During the scan phase, concurrent updaters operate on the public index in the same manner as in the Index-X-Basic algorithm, i.e., the scan phase code for updaters is identical in Figures 3.13 and 3.17.

After building the intermediate index (after step 4, Figure 3.16), the build process locks the

```
Build process
    begin
        R: Input relation
        A: Attribute of R on which to build index
        H: Heap file
        L: Head of a list of leaf pages
        step 1:  Phase[R, A] = scan, Index[R, A] = empty index
        step 2:  H = Extract_keys(R)
        step 3:  H = Sort(H)
        step 4:  Lock Index[R, A] in exclusive mode
        step 5:      Phase[R, A] = build
                     L = List of leaf pages of Index[R, A]
        step 6:      Index[R, A] = Make_index(H, L)
        step 7:      Phase[R, A] = available
        step 8:  Unlock Index[R, A]
    end
```

Figure 3.15: The Index-X-Merge Algorithm

public index in Exclusive mode, driving away any updaters, and records a pointer to the head of the leaf page list of the public index. It then re-initializes the public index to be empty, changes the state variable to indicate to updaters that the build phase has started, and finally releases the Exclusive lock on the public index. During the build phase, the build process sequentially applies the updates from the leaf page list (saved above) to the intermediate index that it built with the scanned entries. Updaters continue to add their updates to the public index during the build phase, but these build phase updates are stored differently from the scan phase. This is illustrated by the difference in the updater code segments for the scan and build phases in Figure 3.17. The basic idea in the build phase is to retain the latest update for a (key, rid) pair only if an odd number of operations took place after the start of the build phase. (Recall that at the start of the build phase, the public index is empty.)

This difference between the handling of scan phase and build phase updates is similar to the difference between the *Sort* and *NSort* functions described earlier. The explanation is analogous, so we do not repeat it here.

After applying all of the scan phase updates to the intermediate index, the build process enters the catchup phase, where the intermediate index is made visible to concurrent updaters. The build process now applies the build phase updates to this intermediate index. During the catchup phase, updaters apply their updates directly to the public intermediate index, behaving exactly like updaters in the catchup phase of the List-C-Basic algorithm (Figure 3.9). These concurrent updaters leave specially marked entries in the index in case they detect inconsistencies (e.g., an inserter finds the entry to be inserted to already be in the index, or a deleter does not find an entry

```
Build process
    begin
        R: Input relation
        H: Heap file
        T: Temporary B-tree index
        L: Head of a linked list of leaf pages
        step 1:  Phase[R, A] = scan, Index[R, A] = empty
        step 2:  H = Extract_keys(R)
        step 3:  H = Sort(H)
        step 4:  T = Make_index(H, empty)
        step 5:  Lock Index[R, A] in exclusive mode
        step 6:      L = leaf page list of Index[R, A]
                     Index[R, A] = empty
                     Phase[R, A] = build
        step 7:  Unlock Index[R, A]
        step 8:  foreach entry in L do
                     case entry type of
                         insert: if entry not in T, insert it
                         delete: if entry in T, delete it
        step 9:  Lock Index[R, A] in exclusive mode
        step 10:     L = leaf page list of Index[R, A]
                     Index[R, A] = T
                     Phase[R, A] = catchup
        step 11: Unlock Index[R, A]
        step 12: foreach entry in L do
                     case entry type of
                         insert:  if marked delete entry present in Index[R,A] then
                                      Delete the marked entry
                                  else Insert the entry normally
                         delete:  if marked insert entry present in Index[R,A] then
                                      Delete the marked entry
                                  else Delete the normal entry that is present
        step 13: Phase[R, A] = available
    end
```

Figure 3.16: Build Process in the Index-C-Basic Algorithm

```
Update transaction
    begin
        R = input relation
                . . . Normal processing . . .
        step n:  if Phase[R, A] = scan then
                        Lock Index[R, A] in share mode
                            if Phase[R, A] ≠ scan then
                                Unlock Index[R, A], goto step n
                            else case updater of
                                insert:  if delete entry present in Index[R, A] then
                                             replace the entry by an insert entry
                                         else add an insert entry
                                delete:  if insert entry present in Index[R,A] then
                                             replace the entry by a delete entry
                                         else add a delete entry
                        Unlock Index[R, A]
                else if Phase[R, A] = build then
                        Lock Index[R, A] in share mode
                            if Phase[R, A] ≠ build then
                                Unlock Index[R, A], goto step n
                            else case updater of
                                insert:  if delete entry present in Index[R, A] then
                                             remove the entry
                                         else add an insert entry
                                delete:  if insert entry present in Index[R,A] then
                                             remove the entry
                                         else add a delete entry
                        Unlock Index[R, A]
                else if (Phase[R, A] = catchup then
                    case updater of
                        insert:
                            if entry is already present in Index[R,A] then
                                add a specially marked insert entry
                            else insert this (key, rid) in Index[R,A]
                        delete:
                            if entry is not present in Index[R,A] then
                                leave a specially marked delete entry
                            else delete this entry from Index[R,A]
    end
```

Figure 3.17: Updater in the Index-C-Basic and Index-C-Merge Algorithms

```
Build process
    begin
        R: Input relation
        H: Heap file
        T: temporary B-tree index
        L: Head of a linked list of leaf pages
        step 1:  Phase[R, A] = scan, Index[R, A] = empty
        step 2:  H = Extract_keys(R)
        step 3:  H = Sort(H)
        step 4:  Lock Index[R, A] in exclusive mode
        step 5:      L = leaf page list of Index[R, A]
                     Index[R, A] = empty
                     Phase[R, A] = build
        step 6:  Unlock Index[R, A]
        step 7:  T = Make_index(H, L)
        step 8:  Lock Index[R, A] in exclusive mode
        step 9:      L = leaf page list of Index[R, A]
                     Index[R, A] = T
                     Phase[R, A] = catchup
        step 10: Unlock Index[R, A]
        step 11: foreach entry in L do
                     case entry type of
                         insert: if marked delete entry present in Index[R,A] then
                                     delete the marked entry
                                 else insert the entry normally
                         delete: if marked insert entry present in Index[R,A] then
                                     delete this marked entry
                                 else delete the normal entry that is present
        step 12: Phase[R, A] = available
    end
```

Figure 3.18: The Index-C-Merge Algorithm

to delete). These marked entries will be found and removed by the build process as it executes step 12 of Figure 3.16. When the build process is finished applying all of the build phase updates, the index is up to date and is made available for normal use.

3.6.4 The Index-C-Merge Algorithm

The last index-based algorithm that we discuss is the Index-C-Merge Algorithm. The build process in this algorithm (Figure 3.18) differs from the Index-C-Basic build process only in the manner in which the intermediate index is built. This index is built in one pass in step 7 (Figure 3.18) using the leaf page list of the concurrent scan phase updates as well as the sorted list of scanned entries. This is similar to how the intermediate index was built in the List-C-Merge algorithm. Update transactions in the Index-C-Merge algorithm execute just like updaters in the Index-C-Basic algorithm (Figure 3.17).

3.7 More Concurrency for Updaters

In Section 3.4, we stated that concurrent updaters are assumed to register their index updates immediately into the update-list or the temporary index during index construction. This ensures that all updates to a given page that occur before the page is scanned by the build process are registered in the update-list or the temporary index before the scan phase completes. This is necessary to prevent inconsistent situations from being seen by update transactions due to the late application of index updates[9]. We will now explain how immediate index updates affect concurrency for the various algorithms.

For the list-based algorithms, the above requirement means that updaters have to request an Exclusive lock on the list while still holding a short-term Exclusive lock on a newly modified relation page. The relation page lock can be released only after the list lock is granted. If the list becomes a concurrency bottleneck, this strategy may slow down other updaters as well as the scan phase of the build process due to interference at the relation page level. This seems unlikely to happen, however, due to the fact that the list append is likely to be an in-memory operation and therefore very fast (since the last page will always be in memory). In the index-based algorithms, updaters have to get a Share lock on the index before releasing the Exclusive lock on the modified relation page. This Share lock is only used to exclude the build process from the index, so multiple updates can still proceed concurrently on the temporary index. It therefore appears that there is an even smaller chance of a concurrency problem arising here as compared to the list case.

If the requirement of immediate index updates is considered undesirable, it can be relaxed by employing the page coloring technique of [Pu85]. In this strategy, all relation pages are colored "white" when the build process starts. As soon as a relation page is scanned in the scan phase, it is colored "black." Now, instead of **always** registering their index updates to the update-list or the temporary index, updaters will only register their index updates on black pages (pages that have already been scanned by the index construction process). This prevents the inconsistent situations mentioned earlier from occurring even if the index updates are not registered immediately. Also, it reduces the total number of updates registered during the scan phase, reducing the work of the build process in the end. An implementation of this algorithm requires a special "color bit" with every relation page to keep track of the color of the page. Multiple color bits per relation page would be needed to accommodate more than one index construction process at a time.

[9] An example of this type of inconsistency is as follows: Suppose a page was scanned by the build process after a (key, rid) pair was inserted into it, but the index construction process completes before the corresponding index update is performed. When the index update for this (key, rid) pair is finally performed by the update transaction (on the newly built index, which is now available as a normal index), the updater will find that this entry is already in the index. While this is legal during the construction phase, during normal operation it is an error.

3.8 Related Work

In parallel with this work, two other on-line index construction algorithms have also been proposed by Mohan and Narang [Moha91]. Their first algorithm is index-based and allows updaters and the build process to share the same index throughout. In this algorithm, the build process first initializes a public index into which updaters concurrently insert their index updates. It then scans the relation, sorts the scanned entries, and inserts them (not necessarily one at a time) into the public index concurrently with updaters. Updaters sometimes have to leave special *pseudo-deleted* entries in the index that later may or may not be removed later by the build process. A disadvantage of this algorithm is that the index building process cannot build the index bottom-up from the sorted entries, and hence it may take a long time to complete. Also, at the end of the index construction process, the new index may still contain superfluous pseudo-deleted entries.

The second algorithm described in [Moha91] is list-based and does not have the above disadvantages. It uses an update-list, like the List-C-Basic strategy, but the index building process performs catch-up differently. In their algorithm, the build process catches up by copying list entries (except those at the very end of the file, where update transactions may be still actively appending entries) into an intermediate index that has been built in a bottom-up fashion using the sorted scanned entries. Finally, when only a small number of entries remain at the end of the update-list, the build process exclusively locks the list, applies the last few entries to the index, and then makes the index available for normal use and releases the exclusive lock.

In both of the above algorithms, an alternative to the coloring strategy described above is used to limit the updates actually deposited in the index or the update-list. In this strategy, the build process scans the relation in a pre-specified order and concurrent updaters figure out if a page has been scanned yet by looking at the current position of the build process (which is available in a shared variable).

As we mentioned earlier, index construction for a terabyte relation may take days (due to the scan phase itself). This means that the index building process must be able to survive crashes and to complete without having to restart from scratch after every crash. Recovery strategies are presented in [Moha91] for their algorithms, and it should be straightforward to apply those recovery techniques to the algorithms described in this chapter.

3.9 Summary

In this chapter, we have presented a range of solutions to the important problem of on-line index construction. In particular, we have described two families of on-line index construction algorithms.

These on-line algorithms vary in the sort of data structures that they use to store the concurrent updates (list or index), the strategies used to actually build the index from leaf-level entries, and the degree of concurrency allowed for concurrent updates. The algorithms trade off, to varying degrees, increased building time for increased updater throughput. Proofs of correctness of these algorithms can be found in the Appendix.

In our discussions of on-line index construction algorithms, we have identified certain situations that could favor one on-line algorithm over another, but a detailed performance study is needed to clearly understand the tradeoffs among these algorithms under various system resource and workload conditions. An important question to be answered in such a performance study is: How much of an increase in updater throughput warrants allowing a certain increase in build response time? In answering this question, it is necessary to keep in mind that increasing the build time also increases the "lost opportunity" cost for queries that can run only after the index is built (since they may have unacceptably high costs without the index). This suggests that an appropriate cost model needs to be developed to evaluate the performance tradeoffs of these algorithms. A performance study of on-line index construction algorithms is the topic of the next chapter.

Chapter 4

Performance of On-Line Index Construction Algorithms

4.1 Introduction

In this chapter, we evaluate the relative performance of the on-line index construction algorithms proposed in Chapter 3. Using a detailed simulation model of a centralized DBMS, we compare the performance of these on-line algorithms with that of a good off-line algorithm as well as amongst themselves. By running experiments over a wide range of system, workload, and storage conditions, we investigate the performance trade-offs for the proposed algorithms. In particular, to assist in our analysis, we employ a performance metric, "loss," that captures the lost work in terms of update transactions that are unable to execute due to contention caused by conflicts with the index construction process. We also compare algorithms using other relevant metrics, including the "off-line fraction," which characterizes the fraction of time (relative to the response time of an off-line algorithm) during which updaters are unable to proceed.

The rest of the chapter is organized as follows: Section 4.2 discusses the performance trade-offs involved in choosing one on-line algorithm over another. The simulation model used in our study is described in Section 4.3. Section 4.4 describes the performance experiments that we conducted and presents their results. In Section 4.5 we predict the performance of other proposed algorithms based on our performance results. Finally, in Section 4.6, we summarize the results of the performance study.

4.2 Performance Trade-Offs

4.2.1 Hidden Costs

Studying the performance of on-line index construction algorithms is complicated due to the hidden "lost opportunity" costs involved in building a new index. These costs arise because the performance of the database system with the new index may be much different from its performance without the index. Not surprisingly, these hidden costs are closely related to the reason for building the new index. The decision to build a new index may be made for either of the following reasons, each of which relates to a performance-improving opportunity:

1. The new index would significantly speed up a class of queries that are currently running inefficiently (i.e., using sub-optimal access plans) in the system.

2. The new index would enable a new class of queries to be executed that cannot be executed at all (reasonably) given the current system configuration. For example, a credit card company might want to provide a new service that involves accessing its customer records using a currently unindexed attribute. A naive way of executing such queries without building a new index on the relevant attribute might involve a relation scan, which could be prohibitively expensive for large relations (e.g., it could take days for a terabyte relation).

Accounting for either of the aforementioned considerations is difficult, as the interests of queries that will not be speeded up by the new index conflict with those of the queries that will indeed benefit from the new index. In particular, for the queries that will benefit from the new index, the best way to build the index is to build it as soon as possible. On the other hand, for existing queries that will not benefit from the new index, the best way to build the new index is the way that creates the least interference for them during index construction. The question of which class of queries is more important, and thus needs to be given priority in the system, is application-specific and depends on factors like the economic benefit of preferring one class of queries over another. Such factors are hard to quantify and will vary from system to system.

Despite the complexity of the general problem, there is an objective factor that we can indeed quantify: the effect of index construction on the system's already existing on-line workload. Therefore, as a first step towards understanding the relative performance of the various on-line index construction algorithms, we will consider the impact of index construction on concurrent transactions that use other access paths to efficiently access and update the relation on which the index is being built. Furthermore, we assume that the decision to build the index is made off-line. Although we do not explicitly consider the waiting-costs for the class of queries in 1 and 2 above,

such waiting costs can be easily factored into our performance metrics, if such costs are available for a particular system.

4.2.2 Performance Metrics

In order to study the performance impact of index construction on an already existing on-line workload, two important metrics have to be studied: the index build time, and the on-line workload's throughput during the index construction period. As we mentioned earlier, the off-line algorithm provides the fastest way of building an index at the cost of providing no throughput for updaters. On the other hand, the on-line algorithms each allow concurrent updaters during some or all of the index construction process, with the price being an increase in the time required to build the index. For the on-line algorithms then, the following question arises: How much of an increase in updater throughput is needed to compensate for a corresponding increase in the build response time? We shall try to answer this question by quantifying the *loss* to the database system caused by the index construction activity.

Suppose that the best throughput possible in the system *without* the new index is T_{best}. Suppose also that a particular index construction algorithm A has a build response time of R_A seconds and that during its building time it permits a throughput of T_A transactions per second. In an on-line algorithm, update transactions face interference from the index construction process in terms of data and resource contention, and read-only transactions face resource contention from the index construction process, so T_A will be less than T_{best}. Using R_A, T_A, and T_{best}, we can estimate the loss to the system in terms of the number of potential transactions that could not execute due to contention caused by the index construction activity:

$$loss = (T_{best} - T_A) \times R_A \tag{4.1}$$

Interestingly, the formula for loss can also be applied to the off-line algorithm directly. The loss metric thus gives us a way of comparing the performance of an on-line algorithm both with that of other on-line algorithms and with that of the off-line algorithm. From the loss formula, it can be seen that the loss will be high if index construction takes a long time (if R_A is large) or if the throughput during index construction is very low (if T_A is small). The loss metric thus offers a simple way to answer the question posed in the previous paragraph regarding the amount of additional throughput needed to offset an increase in the build response time. In terms of this metric, an algorithm with a lower loss is better than one with a higher loss. Between two algorithms with the same loss, the one with the smaller response time is better since the index is available earlier in that case. Finally, the *normalized loss* for an algorithm can be defined as the loss for that

algorithm divided by the loss for the off-line algorithm.

Though the loss metric provides a clean way to combine the build response time and the throughput into one measure, it alone is not sufficient to characterize the performance of an on-line algorithm completely. In particular, recall that some on-line algorithms (e.g., the List-X-* and Index-X-* algorithms) have exclusive phases during their execution. Since high-performance transaction processing systems may have severe maximum updater response time requirements, the durations of such exclusive phases may be critical to the performance of such systems. Thus, when evaluating an index construction algorithm, we will also consider the *off-line fraction* of the algorithm, which is defined as the ratio of the duration of its exclusive phase (if any) to that of the off-line algorithm (which has a single exclusive phase equal to its entire build response time).

Our above discussion for the loss and off-line fraction metrics is directly applicable to closed systems. In a stable open system, the throughput is equal to the arrival rate. For open systems, therefore, the important performance metric is response time. Here again, we can define a loss metric analogous to the one described above based on the average transaction response time without the index construction process (the best response time) and the average response time obtained when the the index construction process executes in the system. Another alternative for an open system would be to analyze the various algorithms based on a loss metric constructed from the *maximum throughput*, which is defined as the maximum possible arrival rate that can be handled stably by the open system [John90a, John90b]. Thus, we believe that the performance results for on-line index construction algorithms that we obtain here using a closed system model would to a large extent, also hold under an open system model.

4.2.3 Overall Cost

We now indicate how the waiting costs for queries can be combined with the loss metric to determine the overall cost of index construction in a system. Suppose that the index is needed at time t_n. The cost of waiting is proportional to the length of time that the index is unavailable beyond t_n, i.e., to $t_a - t_n$, where t_a is the time when the index actually becomes available. We now combine the waiting cost with the earlier equation for loss and provide a formula for the overall cost for index construction. Assuming that the cost of losing one existing transaction is C_1, and the cost of waiting for one unit of time is C_2, the following equation gives the overall cost of building the index:

$$Cost = C_1 \times loss + C_2 \times (t_a - t_n) \tag{4.2}$$

In the rest of this chapter, we concentrate on estimating the loss, i.e., the impact of index construction activity on the performance of the on-line workload in a system.

4.3 Simulation Model

In this section, we describe the simulation model used to study the performance of the on-line algorithms described in Chapter 3. This model, which is a closed queueing model with a fixed multi-programming level (MPL), is an extension of the model used earlier in Chapter 2 for studying the performance of B-tree concurrency control algorithms. While the focus earlier was a B-tree that was being accessed concurrently, the central focus of the model here is a relation on which a new index is being built.

The model of the system hardware assumes a computer system with one or more CPUs and disks. Requests for the CPUs are scheduled using an FCFS (first-come, first-served) discipline with no preemption. Each of the disks has its own disk queue, and each queue is managed using an elevator disk scheduling algorithm[1]. Apart from the CPUs and disks, the physical resource model also includes a buffer pool for holding disk pages in main memory. The buffer pool is managed in a global LRU fashion for all pages except relation pages. Since relation page accesses are either sequentially (due to the build process) or randomly (due to concurrent update or search activity), relation pages are added to the tail of the LRU list upon being released, effectively giving them lower priority than index pages. The buffer manager performs demand-driven writes when dirty pages are chosen for replacement. The system model also includes a lock manager for setting and releasing locks on pages and records.

The components of the database storage model include the relation (which is a heap file) on which an index is being built, one already existing unclustered B-tree index on this relation, and auxiliary data structures like the temporary lists and indices needed by the various on-line index construction algorithms. Important parameters of the database storage model include the size of the initial relation in tuples, the maximum number of tuples per relation page, and the maximum fanout of a B-tree index page. In our experiments, the physical size of a page is always the same (4K bytes), so a variation in the maximum capacity of a relation page should be viewed as being due to different tuple sizes. For modeling simplicity, all tuples are assumed to be of the same size, all index entries are assumed to be of the same size, and no duplicate keys are allowed in the index. (These simplifications should not impact our qualitative performance results.) The distribution of values for an indexed attribute of the relation are drawn from a random permutation over an

[1] Unlike Chapter 2, the use of the elevator algorithm for disk scheduling is quite important here. This is due to the fact that the workload will now involve both random (updater) and sequential (index builder) I/O patterns.

num-cpus	Number of CPUs (1)
num-disks	Number of disks (1, 8)
disk-seek-time	Min: 0 msec; Max: 27 msec
cpu-speed	20 MIPS
cc-cpu	Cost for lock/unlock (100 inst.)
buf-cpu	Cost for buffer call (1000 inst.)
page-search-cpu	Cost for page search (50 inst.)
page-modify-cpu	Cost for key insert/delete (500 inst.)
page-copy-cpu	Cost for page copy (1000 inst.)
compare-cpu	Cost for comparing keys (2 inst.)
init-rel-keys	Tuples in initial relation (100,000)
max-fanout	Index-entries/page (200/page)
rel-page-capacity	Tuples/page (2, 20, or 200 /page)
page-size	Size of a disk page (4KB)
alg	On-line algorithm (List-X-Basic, Index-C-basic, etc.)
num-bufs	Size of the buffer pool (250)
search-term	Number of search terminals (0 or 50)
update-mpl	Number of insert and delete terminals (0, 2, 10, 20, 40)
insert-prob	Proportion of inserts among updates (50%)

Table 4.1: Simulation Parameters

integer key space of 1..400,000.

The workload model consists of the index build process and a fixed set of user terminals, each of which submits one of three types of relation operations (search, insert, or delete). Since the relation is stored as a heap file, insert operations find a non-full page in the relation using a bit map and then insert a tuple into that page. After their relation insert, they immediately perform an insert into the relation's existing B-tree index. Following this, they take the appropriate action, if any, required by the on-line index building algorithm. Deletes, on the other hand, access a single relation page at random, delete a randomly chosen tuple from that page, and then immediately perform the corresponding B-tree delete to the already available index. Like inserts, they then take the action called for by the relevant on-line index building algorithm. Finally, searches randomly access a single tuple of the relation using the existing B-tree index. Each terminal submits its operations one at a time. As soon as a terminal submits an operation, it becomes active and executes in the system; when the operation completes, it returns to the terminal. In the current study, we use a terminal think time of zero, so the terminal immediately generates another operation of the same type when its current operation completes.

The simulation parameters for our experiments are listed in Table 4.1. In all of the experiments discussed in this chapter, there is exactly one CPU in the system. Apart from the single CPU, a system configuration consists of a fixed number of disks and a fixed capacity for relation pages. In each system configuration, we varied the MPL for updaters from 0 (where only the build process executes) to a maximum of 40, conducting one experiment for each on-line index construction

algorithm as well as for the off-line algorithm. At the start of each experiment, the buffer pool is initialized with randomly chosen pages from the initial relation; the build process is then started along with the specified number of terminals. The terminal types (search, insert, and delete) are initialized according to the workload parameters. The experiment is stopped when the build process terminates. A special additional experiment is run to calculate the best updater throughput in the system without the new index (for calculating the loss using equation 4.1). Batch probes in the DeNet simulation language are used with the operation response time metric to generate confidence intervals. For all of the data presented here, the 90% confidence interval for relation operation response times was within 2.5% (i.e., $\pm$2.5%) of the mean.

4.4 Performance Results

An important factor likely to affect the performance of an on-line index construction algorithm is the relative proportions of time spent in the various phases of index construction. These relative proportions depend on the size of the index relative to the size of the relation itself. We modeled different relative sizes by keeping the size of an index entry constant (20 bytes) and considering three different tuple sizes, small (20 bytes), medium (200 bytes), and large (2000 bytes). We subdivide the performance experiments into three categories based on the tuple size and present the results for each of these categories.

4.4.1 Experiment Set 1: Small Tuple Size (20 Bytes)

In this set of experiments, the size of the index is comparable to the size of the relation since an index entry and a tuple are the same size. The system workload consists of the build process and a set of concurrent updaters. The multiprogramming level (MPL) gives the total number of updaters in the system; half are inserts while the other half are deletes. We will subdivide the small tuple experiments into those involving a system with a single disk and those involving a system with eight disks.

Single Disk Results

The build response times for the various algorithms in the single disk case are given in Figure 4.1, and the updater throughput curves are given in Figure 4.2. As expected, the build response time for all of the on-line algorithms increases with the MPL due to an increase in resource contention at higher MPLs. We also see from Figure 4.1 that the List-C-Basic algorithm's build response time increases much faster than those of all the other algorithms. The reason for this is that the

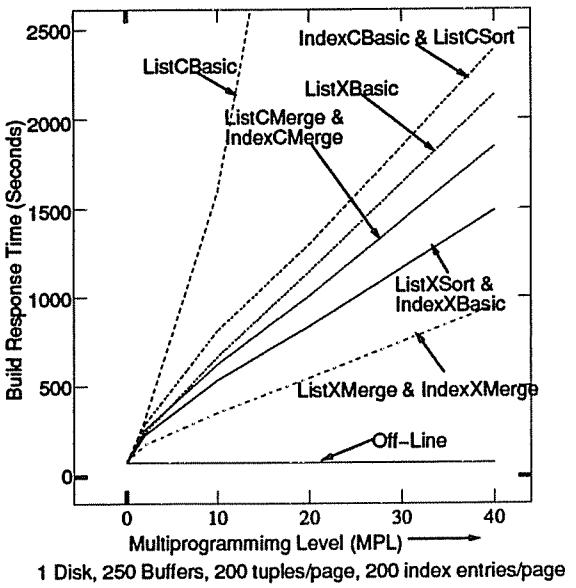


Figure 4.1: Build Times, 1 Disk

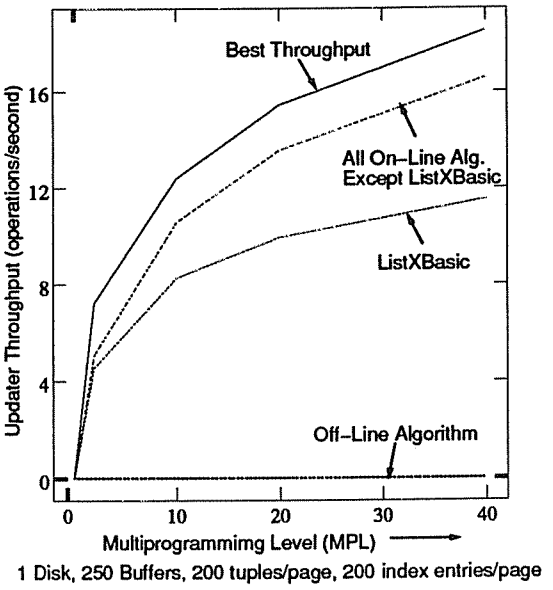


Figure 4.2: Updater Thruput, 1 Disk

build process in List-C-Basic sequentially inserts unsorted entries from the update-list into the intermediate index during the build phase (Figure 3.5). These sequential inserts can cause multiple accesses to a given leaf page which, since the buffer pool can hold only a subset of the leaf pages in memory, results in multiple I/Os for the same page. These extra I/Os cause the build phase to become very long in this extremely disk bound situation, which in turn increases the size of the build phase update-list, thus increasing the duration of the catchup phase as well. Sorting the list before insertion into the intermediate index (as in List-C-Sort) alleviates this problem, so the response time is much lower for List-C-Sort than for List-C-Basic. In contrast to List-C-Basic, List-X-Basic does not suffer as much because its build phase length increases much more slowly due to the absence of buffer and resource contention during this phase.

In Figure 4.1, the build response time ordering of the on-line algorithms other than List-X-Basic and List-C-Basic reflects the increasing amount of work that they have to do for index construction. Among these eight algorithms, each of the concurrent (*-C-*) algorithms has a higher response time than all of the exclusive (*-X-*) algorithms. This is expected since the *-C-* algorithms allow concurrency throughout the index construction period, resulting in increased contention as well as extra work for catching up. Among the four algorithms within each class, the algorithms that use merging are better than those that perform sequential (sorted) inserts. This is because the sequential insert strategies perform slightly more work in the scan phase than the merge-based algorithms, while the build phases and catchup phases (if any) are comparable in length. The extra work results in expensive I/Os that increase the build response times of the sequential-insert

algorithms (List-*-Sort and Index-*-Basic).

While the various on-line algorithms differ widely in their build response times, all except List-X-Basic attain the same updater throughput[2] (Figure 4.2). In particular, this means that all of the *-X-* algorithms except List-X-Basic attain the same throughput as the *-C-* algorithms. This is surprising since we would expect the *-X-* algorithms to attain less throughput than the *-C-* algorithms due to their exclusive build phase. This is due to the extremely high level of resource contention in this experiment. In all of the *-X-* algorithms except List-X-Basic, a bottleneck at the lone disk increases the scan phase duration tremendously at high MPLs, while the (exclusive) build phase duration remains about the same due to lack of contention; the build phase therefore forms a negligible part of the build response time, causing a negligible effect on the throughput. In List-X-Basic, however, the extra I/Os during the build phase cause this phase to become a significant fraction of the build response time at higher MPLs, resulting in a significant drop in throughput.

Having looked at the updater throughput and response times separately, we now look at the normalized loss in Figure 4.3 in order to combine both measures. Recall that the normalized loss for a given algorithm is the ratio of the loss for the algorithm (calculated using Equation 4.1) to the loss for the off-line algorithm. Note from Figure 4.3 that the loss for the List-X-Basic and the List-C-Basic algorithms is very high compared to that for the other algorithms (so much so that their values for high MPLs are omitted from the figure). The large loss in List-C-Basic is due to its very high response time (Figure 4.1), while the loss in List-X-Basic is due to its lower updater throughput (Figure 4.2). The normalized loss metric also gives an idea of the improvement achieved by using an on-line algorithm instead of the off-line algorithm. The loss curves in Figure 4.3 show that, at high MPLs, only two merge-based algorithms (List-X-Merge and Index-X-Merge) manage to achieve better loss than the off-line algorithm. As shown, the loss for the *-C-* algorithms increases with MPL and reaches a maximum value of between 225% to 250% of the loss for the off-line algorithm. In contrast, the losses for the *-X-* algorithms (except List-X-Basic again) track the loss for the off-line algorithm more closely.

Despite the loss results, it is not necessarily the case that the *-X-* algorithms are preferable to the *-C-* algorithms here, as they might have an unacceptably large exclusive build phase. In order to investigate this, we plot the off-line fraction of these algorithms in Figure 4.4. We see that the duration of the exclusive build phase for the List-X-* and the Index-X-* algorithms is a sizable fraction (25% to 50% for all algorithms except List-X-Basic) of the total response time of

[2]There were slight (but insignificant) differences between the various algorithms. We only present significant differences in our graphs.

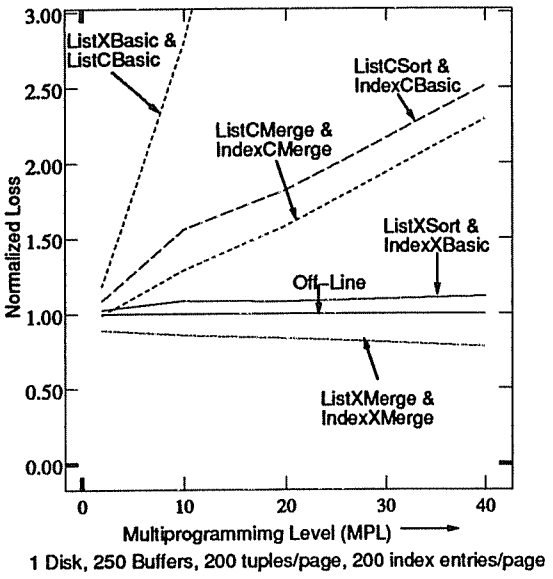


Figure 4.3: Loss, 1 Disk

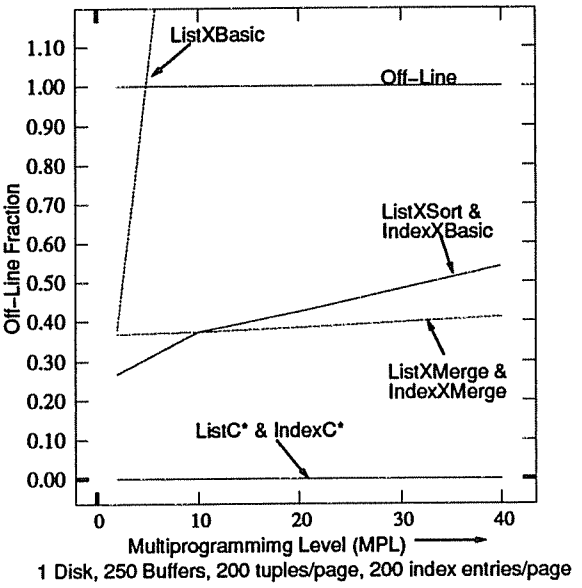


Figure 4.4: Off-Line Frac., 1 Disk

the off-line algorithm. Thus, if it is unacceptable for updaters to wait in the case of the off-line algorithm, it is likely to be unacceptable for them to wait in the *-X-* algorithms as well (since the waiting times are of the same order of magnitude). Using the best among the *-C-* algorithms (List-C-Merge or Index-C-Merge) thus seems to be a better choice here even though they have a higher loss than most of the *-X-* algorithms.

In the single disk experiments, the builder in the on-line algorithms faces very high resource contention. In such a situation, a bottleneck forms at the disk in the on-line index construction algorithms at even small MPLs; this increases the build response time enormously, and concurrent execution of updaters does not provide enough throughput to offset this increase in response time. This experiment therefore represents a worst case for the on-line algorithms. In order to study their performance in a less resource-bound situation, the next set of experiments assumes a system with eight disks. Due to the extremely poor performance of the List-X-Basic and the List-C-Basic algorithms, we will omit these two algorithms from all future graphs.

Eight Disk Results

The build response time and updater throughput curves for the eight disks case are given in Figures 4.5 and 4.6 respectively. It can be seen from the build response time curves that the ordering of response times among the various on-line algorithms is the same as in the one disk case. The key difference here is that the best on-line algorithm now has a maximum response time of only a few times that of the off-line algorithm, while in the one disk case the best on-line algorithm

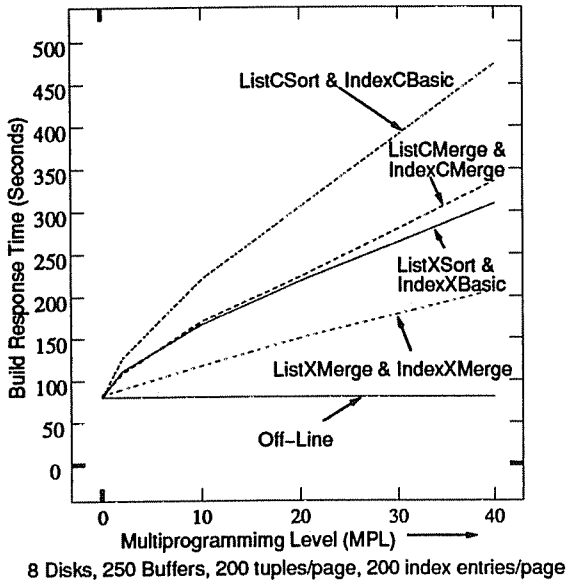


Figure 4.5: Build Times, 8 Disks

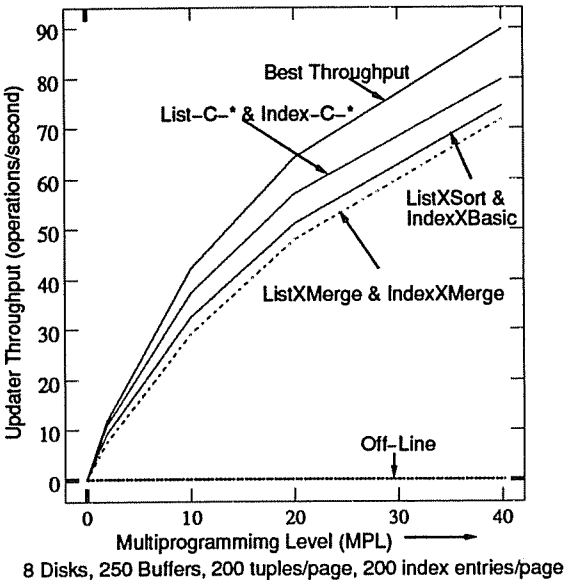


Figure 4.6: Updater Thruput, 8 Disks

was more than ten times slower than the off-line algorithm at an MPL of 40 (Figure 4.1). Adding disks has reduced the level of resource contention considerably for the builder, resulting in a faster index building time for all of the on-line algorithms.

From the updater throughput curves (Figure 4.6), we see that the List-X-* algorithms and the Index-X-* algorithms have a slightly lower throughput than the corresponding *-C-* algorithms. In the *-X-* algorithms, the (exclusive) build phase now forms a significant proportion of the index construction time, thus leading to a noticeable loss in throughput. Another point to be noted from Figure 4.6 is that the List-X-Sort and the Index-X-Basic algorithms attain a slightly higher updater throughput than the List-X-Merge and the Index-X-Merge algorithms. Again, the reason is that the build phase proportion is greater in the two merge-based algorithms than in List-X-Sort and Index-X-Basic at high MPLs. This is because the (non-exclusive) scan phase in List-X-Sort and Index-X-Basic is slightly longer than for the merge-based algorithms, while the build phases are comparable in length. As seen in the single disk case earlier, extra work in the scan phase is expensive at high MPLs due to resource contention; this fact is shown by the higher build response times of List-X-Sort and Index-X-Basic in Figure 4.5.

In Figure 4.7, we present the normalized loss for the various algorithms in the eight disks case. In contrast to the results of the one disk experiments, where the loss for the *-C-* algorithms was larger than that for the other algorithms, the *-C-* algorithms (which have no exclusive phases) perform better than the other algorithms in terms of the loss metric here. The off-line fractions of the various algorithms for this experiment were the same as in the earlier high contention case

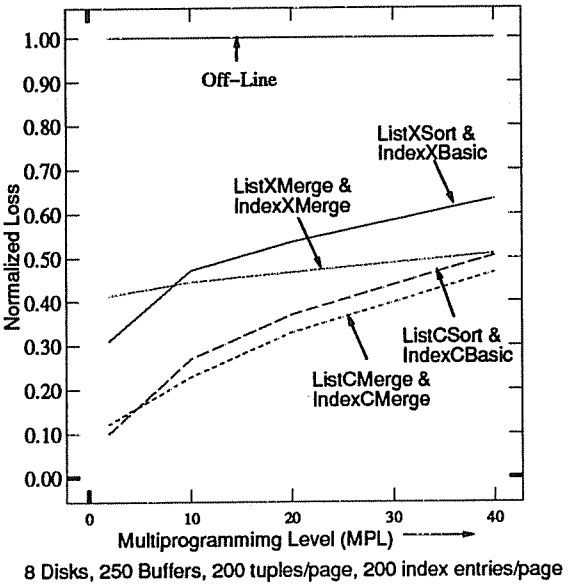


Figure 4.7: Loss, 8 Disks

(Figure 4.4), so we omit those curves here; this is to be expected since the duration of the periods during which updaters are locked out should be the same as in the one disk case (since the build process is the only active process in the system during that time). Since the off-line fraction for the *-C-* algorithms is very close to zero, and they also have the least loss, they are unequivocally better than the *-X-* algorithms in this situation. Among the *-C-* algorithms, List-C-Merge and Index-C-Merge are the best since they involve the least overhead (as demonstrated by their superior build response times).

This set of experiments (both the single and eight disk results) examined the case where a tuple is the same size as an index entry. We also ran experiments in cases where the tuple size is ten and then a hundred times the size of an index entry, respectively. To conserve space, we will only present a complete set of results for the large tuple experiments. Also, since we found above (and in all of our other experiments as well) that the merge method of building the index was better overall than the basic and sort strategies, we will show only the four merge-based algorithms and the off-line algorithm in the remaining graphs.

4.4.2 Experiment Set 2: Large Tuple Size (2000 Bytes)

In this set of experiments, a tuple is a hundred times larger than an index entry, unlike in the earlier set of experiments where they were the same size. This causes an increase in the relation size (200MB here as compared to 2MB in Experiment Set 1), while the size of the index remains the same as before ($\sim$ 2MB). This increase, in turn, causes the scan phase to dominate the process of

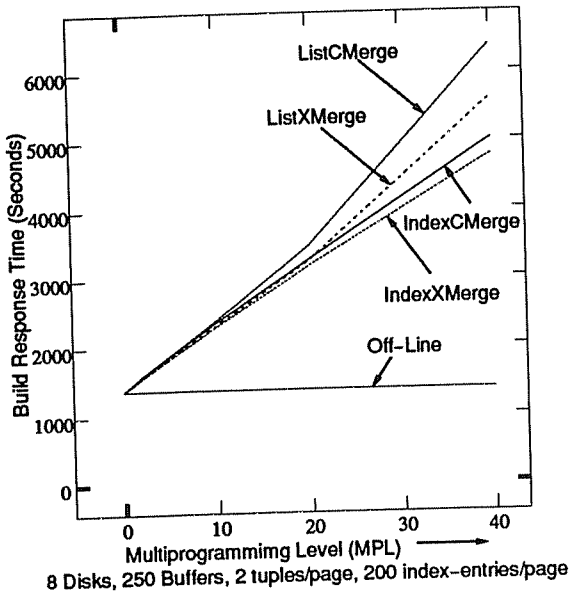


Figure 4.8: Build Times, Large Tuples

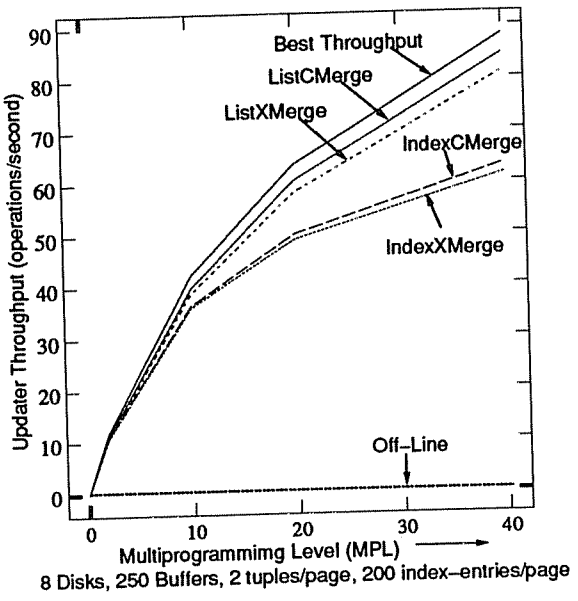


Figure 4.9: Updtr. Tput., Large Tups.

index construction (accounting for more than 95% of the build response time). Since a bottleneck at the disk can be expected to swamp the performance differences between the various on-line algorithms in the one disk case, as we saw earlier in the small tuple experiments, we only conducted experiments on a system with eight disks here.

Eight Disk Results

The build response time curves for the large tuple experiments on a system with eight disks are given in Figure 4.8. Compared to the response time differences seen in the corresponding small tuple experiments (Figure 4.5), the relative response time differences between the various on-line algorithms are smaller here. This is because the scan phase (during which all on-line algorithms perform similarly) is dominant, and the other phases (which were primarily responsible for the build response time differences seen previously) form only a small portion of the overall index construction time. Another thing to note is that all of the list-based algorithms perform slightly worse at high MPLs here than all of the index-based algorithms. The reason is that the update-list becomes large, due to the large scan phase, and the sorting that takes place during the build phase of the list-based algorithms thus contributes a significant overhead which is absent in the index-based algorithms. Also, between the two list-based algorithms, List-X-Merge is faster than List-C-Merge; this is due to the additional catchup phase in List-C-Merge. Similar behavior is seen between the index-based algorithms.

The updater throughput curves for this experiment are given in Figure 4.9. As shown there,

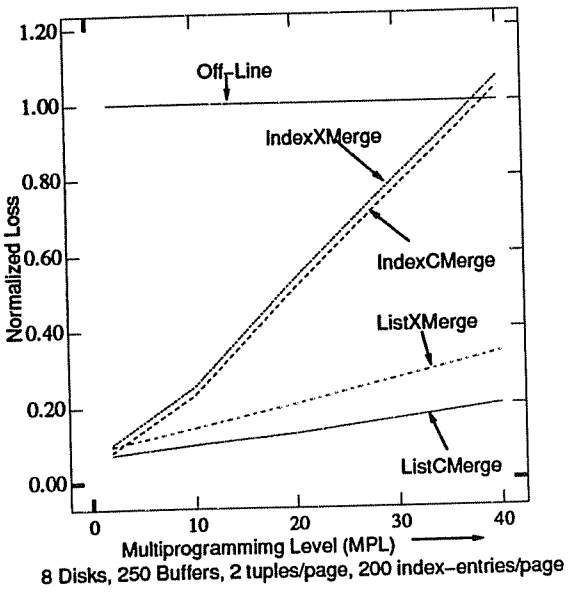


Figure 4.10: Loss, Large Tuples

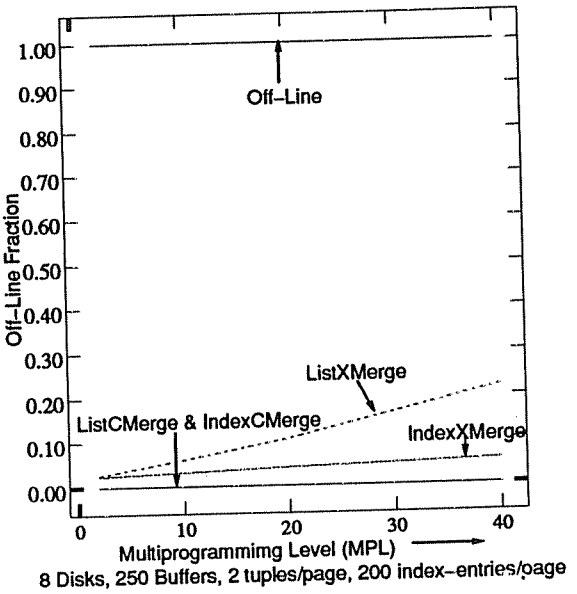


Figure 4.11: Off-Line Fr., Large Tups.

the throughput for the index-based algorithms is significantly less (by about 25%) than that of the list-based algorithms at high MPLs. This is because the public index (into which concurrent updaters insert their updates) in the index-based algorithms becomes large enough at high MPLs in this experiment for every index access to have a high probability of performing a disk I/O for a leaf page. This extra disk access causes a significant increase in the overhead for concurrent updaters in the index-based algorithms, while there is no such overhead in the list-based algorithms (since the append to the update-list almost certainly does not involve a disk access). The reason why this effect was not significant in the small tuple experiments is that the scan phase was much smaller there and, even at high MPLs, the number of updates recorded in the public index did not cause it to become large enough for its leaf pages to be paged out often. Apart from the above differences in throughput between the list-based and the index-based algorithms, we find that among the list-based algorithms, the List-C-Merge algorithm attains a slightly higher throughput than the List-X-Merge algorithm. This is because there is no exclusive phase in List-C-Merge; similar behavior is exhibited by the index-based algorithms.

In order to understand whether or not the increase in throughput achieved here by the list-based algorithms compensates for their increase in response time, we plot the normalized loss for each of these algorithms in Figure 4.10. The loss curves indicate that the index-based algorithms suffer quite a bit due to their reduction in throughput. In fact, at an MPL of 40, Index-X-Merge and Index-C-Merge are even a bit worse than the off-line algorithm. In contrast, List-X-Merge and List-C-Merge perform much better than the off-line algorithm throughout the entire MPL range,

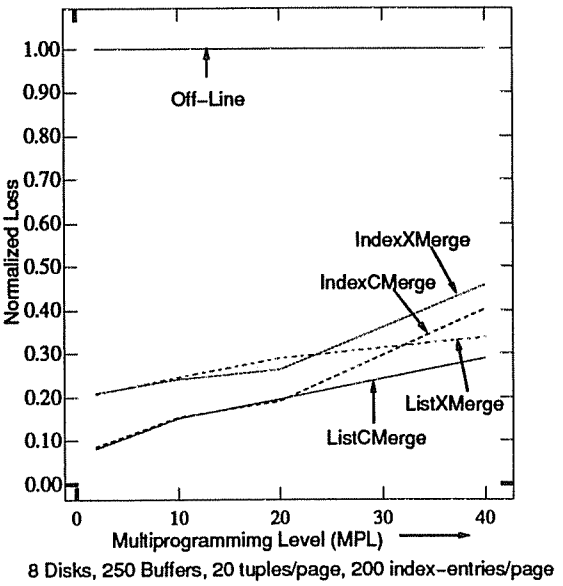


Figure 4.12: Loss, Medium Tuples.

with a maximum normalized loss of 35% for List-X-Merge and only 20% for List-C-Merge. Finally, the off-line fractions for the various on-line algorithms are given in Figure 4.11. As expected, the off-line fractions for the *-X-Merge algorithms are smaller here than in the small tuple experiments (Figure 4.4), but they are still not negligible. In fact, the off-line fraction for the List-X-Merge algorithm increases with MPL from less than 5% to a maximum value of greater than 20% due to the overhead of the sort performed during its exclusive build phase.

4.4.3 Other Experiments

Apart from the large (2000 bytes) and small (20 bytes) tuple experiments, we also performed experiments in which the tuple size was intermediate (200 bytes), as mentioned earlier. In these medium tuple experiments, an index was approximately one-tenth the size of the relation. The results of the medium tuple experiments were essentially a hybrid of the results of the small and large tuple experiments. To illustrate the performance of the algorithms there, we reproduce their loss curves for a system with eight disks in Figure 4.12. From these curves, it is clear that at lower MPLs the trends are like those of the small tuple experiments (Figure 4.7), where the *-C-* algorithms had smaller losses than the *-X-* algorithms, while at higher MPLs the trends are like those observed in the large tuple experiments (Figure 4.10), where the list-based algorithms had smaller losses than the index-based algorithms. This behavior is expected since the scan phase proportion (which affected the relative performance of the on-line algorithms in the small and large tuple experiments) lies between those of the small and large tuple experiments here. A final point

to note from Figure 4.12 is that List-C-Merge again has the least loss throughout the range of updater MPLs considered.

The experiments that we have discussed up to now have only had updaters in the workload. We also conducted a series of experiments where there was a constant background search query load on the relation along with the concurrent updaters. In these experiments, the relative performance of the various algorithms was essentially the same as in the case with no searches, except that due to the resource contention generated by the concurrent searches, all algorithms took much longer to build the index and the on-line algorithms each attained a lower maximum updater throughput.

4.5 Discussion

The performance results of the previous section can be summarized as follows:

- Except in extremely resource bound situations, most of the on-line index construction algorithms clearly outperformed the off-line algorithm. In other words, the throughput that the on-line algorithms achieved for updaters during index construction more than compensated for their increase in build response time.

- Among the on-line algorithms, the best among the algorithms with no exclusive phase (List-C-Merge) outperformed the best among the algorithms with an exclusive phase (List-X-Merge) except in heavily resource bound situations.

- Even in heavily resource bound situations, the best fully concurrent algorithm (List-C-Merge) had a loss of only a few times that of the best partially exclusive algorithm (List-X-Merge). Furthermore, List-X-Merge was found to have an exclusive phase whose length was a non-negligible fraction of the response time of the off-line algorithm, likely making it unacceptable for use in high performance transaction processing systems.

- As should be expected, the relative performance of the various algorithms depended on the proportion of time spent in the initial relation scan phase of index construction. The list-based algorithms performed better than the index-based algorithms when the scan phase was a large proportion ($> 95\%$) of the index building time. When the scan phase was around 50% of the index building time, the fully concurrent (*-C-*) algorithms were found to be superior to the partially exclusive (*-X-*) algorithms.

- The merge strategy for building the index was clearly superior in performance to the basic and sort strategies.

- As a result of the points above, the List-C-Merge algorithm achieved the lowest loss among all of the on-line algorithms over a wide range of tuple sizes, except in heavily resource bound situations.

Even though our simulation results were obtained for relatively small relation sizes (2MB to 200MB), the basic performance conclusions should hold for very large database sizes as well. This is because the relative performance of the various algorithms is affected by the ratio of the size of the index to the size of the relation, which in turn determines the time spent by the on-line algorithms in their different phases of index construction. This ratio depends only on the size of an index entry relative to the size of a tuple (assuming there is enough memory for an efficient sort [Shap86]), and not on the absolute size of the relation itself. Finally, using the above results, we can now make informed projections about the performance of other on-line algorithms that have been proposed in the literature.

Other Candidate Algorithms

As we mentioned earlier in Section 3.8, Mohan and Narang have proposed two algorithms for on-line index construction [Moha91]. One of their algorithms is index-based while the other is list-based. While we have not explicitly simulated the two algorithms from [Moha91] in our experiments, we believe that their performance can be inferred from that of the Index-C-Basic and List-C-Basic algorithms.

First, the performance of our Index-C-Basic algorithm should be comparable to (or better than) the performance of the index-based algorithm from [Moha91]. This is because the set of concurrent updates is inserted into the index built with the scanned entries in the Index-C-Basic algorithm, while in their index-based algorithm the sorted scanned entries are inserted concurrently into an index built with concurrent updates. In realistic situations, the list of concurrent updates is likely to be much smaller than the list of scanned entries, leading Index-C-Basic to perform better than their index-based algorithm.

Turning to the list-based algorithm from [Moha91], the performance of their list-based algorithm should be similar to that of our List-C-Basic algorithm. Their algorithm should perform better than List-C-Basic, in the best case having about half of the response time of List-C-Basic, since updaters selectively (rather than always) record updates to the update-list. Note, however, that our list-based and index-based algorithms can also be modified to make use of a similar optimization (as described in Section 3.7). Since we found the performance of List-C-Basic to be an order of magnitude worse than that of the other on-line algorithms (Figures 4.1 and 4.3), their list-based

algorithm can be expected to perform quite a bit worse than that of the other on-line algorithms (once they too are optimized to selectively record updates). Finally, due to the multi-phase catch-up strategy used in their list-based algorithm, a race condition could occur if the rate of concurrent updates to the update-list somehow happens to be higher than the rate at which the build process can apply these updates to the intermediate index; under such circumstances, their build process may never terminate. This problem cannot occur in the *-C-* algorithms of Chapter 3 due to the fact that the build process and the concurrent updaters share the same index during the catchup phase.

4.6 Conclusions

In this chapter, we have studied the performance of a collection of candidate algorithms for on-line index construction. To aid in our study, we employed a performance metric that measures the loss to the system due to interference between concurrent updaters and the index building process. An important property of the loss metric is that it enables us to directly compare the on-line algorithms with the best off-line algorithm as well as comparing them to one another.

An important conclusion of this study is that in most cases, the fully on-line algorithms (which have no exclusive phase) perform very well and do better than the partially on-line algorithms (which have a concurrent relation scan phase but an exclusive build phase) and the off-line algorithm. In fact, even in a highly resource-bound situation, which is the worst case for the fully on-line algorithms, some of the fully on-line algorithms were only a factor of 2 to 3 worse in terms of loss than the best partially on-line or off-line algorithm. The list-based fully on-line algorithms were found to perform better than the index-based alternatives overall due to the smaller overhead that they impose on concurrent updaters. List-C-Merge, the fully on-line list-based algorithm that uses the merge strategy, appears to be a very good candidate for use in a real system.

An interesting avenue of future work is to extend our on-line index construction strategies to work for indices other than B-tree indices. Most of the work in such extensions will likely involve the development of efficient strategies for combining the scanned entries with the concurrent updates. (We found that the merge strategy was efficient in B-trees for combining the scanned entries with the concurrent updates, but this may not be the case for other indices like a hash index.) The concurrency control techniques employed in our index construction algorithms should be directly applicable for the on-line construction of other types of indices. In fact, the techniques that we have used to design on-line index construction algorithms are sufficiently general that applying them to the problem of executing long-running queries in a DBMS gives rise to a new, highly concurrent

model of query processing. This is the topic of the next chapter.

Chapter 5

Compensation-Based On-Line Query Processing

5.1 Introduction

As we have seen in earlier chapters, one implication of completely on-line operation is that maintenance operations like checkpointing, index management, and storage reorganization have to be performed concurrently with normal database access and updates. Since it is inevitable that commercial database systems will have to implement such on-line utilities in the near future, primitives are bound to appear in these systems to enable such on-line operation. In this chapter, we show how the same primitives that are required for on-line utilities can be used to provide a new, highly concurrent way of executing long-running queries.

We first recall our earlier discussion in Chapter 1 on why long-running decision support queries like the following cannot be executed satisfactorily by current conventional DBMSs.

Q1: Suppose an auditor of a company wants to know the average salary of all of the employees of the company. Assuming the existence of a relation called EMPLOYEE with a SALARY attribute, the SQL form of the query is given below.

```
SELECT    AVG(SALARY)
FROM      EMPLOYEE
```

We explained in Chapter 1 that the only on-line way of executing a query such as Q1 in a typical commercial DBMS is to use a lower degree of consistency (like cursor stability [Gray79]), therefore obtaining only an approximate (non-serializable) answer. Any attempt to obtain a serializable answer for such queries is likely to have an unacceptable impact on system performance.

Apart from the simple query that we have used as an example, many other types of large queries on single or multiple base relations suffer from similar concurrency problems. In order to execute such queries more concurrently yet serializably, it turns out that we can use the techniques developed for the on-line index construction algorithms of Chapter 3. In this chapter, we adapt and extend the techniques used for on-line index construction to process queries like Q1, as well as more complex queries, in a correct (i.e., serializable) and efficient manner.

Our new method of query processing is *on-line*, i.e., it allows updates to be performed on base relations while those same relations are being concurrently accessed by queries. Moreover, our method is *compensation-based*, i.e., concurrent updates to any data participating in a query are communicated to the query's on-line query processor, which then compensates for the updates so that the final answer reflects any changes that they cause. Our compensation-based on-line query processing method requires the addition of only a few new primitives to the system, and the required primitives are similar to those required for supporting on-line index construction and other on-line utilities.

The rest of the chapter is organized as follows: In Section 5.2, we review the basic features of an efficient on-line index construction algorithm and explain which are useful for compensation-based query processing. In Section 5.3, we describe the basic ideas underlying the proposed query processing method. Section 5.4 explains how our compensation-based method can support the execution of a number of single relation query types. In Section 5.5, we describe how the method can be used to process queries on more than one base relation. Section 5.6 deals with implementation issues arising in the design of the proposed approach, and Section 5.7 discusses how compensation-based query processing could be used to compute queries whose answers are transaction-consistent as of some pre-specified time. Section 5.8 compares and contrasts our work with other related work. Finally, in Section 5.9 we summarize the results of this chapter.

5.2 On-Line Index Construction

Before we describe our new query processing method, we first briefly review the features of the best on-line index construction algorithm (List-C-Merge of Chapter 3).

5.2.1 Algorithm Overview

In an on-line index construction algorithm, the index is built by a build process while updaters can concurrently modify the data on which the index is being built. The concurrent execution of the build process and update transactions in the on-line index construction algorithm of interest here

is illustrated in Figure 5.1.

Index construction proceeds in three phases, as shown in the figure. In the first phase, the *scan phase*, the build process scans the relation page by page to collect <key, rid> entries to add to the index. While reading a relation page, the build process holds a short-term exclusive latch on that page. After scanning the entire relation, the build process sorts the index entries that it has collected. Meanwhile, an updater that finds an index building process in the scan phase will append an index update of the form <key, rid, insert/delete> corresponding to its relation update to the update-list (Figure 5.1). Updater appends are synchronized via a short-term exclusive latch on the update-list. The scan phase ends when the build process has finished sorting the scanned entries.

In the second phase of index construction, the *build* phase, the build process combines the entries in the update-list from the end of the scan phase with its initial sorted list of scanned entries. It does so by first sorting the update-list entries and then building an intermediate index (in a bottom-up manner) by merging the sorted scanned entries with the sorted update-list entries. The entries in the update-list have enough information, and the logic involved in sorting and merging is sophisticated enough, to resolve *inconsistencies* that may exist in the scanned entries due to the non-2PL locking strategy used by the build process to scan the relation. The type of inconsistencies and how they are resolved is explained in Section 5.2.2.

As in the earlier scan phase, update transactions append their updates to an update-list during the build phase; this update-list is distinct from the one used in the earlier scan phase and is initialized to empty at the start of the build phase. The build phase ends when the build process completes the construction of the intermediate index.

In the third and final phase of index construction, the *catchup* phase, the updates in the update-list from the end of the build phase are incorporated into the intermediate index from the end of the build phase. Since the details of this phase are not essential for understanding compensation-based query processing, we omit the review of the catchup phase details here.

5.2.2 Inconsistencies and Their Resolution

Since the build process does not use 2PL on the tuples that it scans, the set of entries obtained by the build process at the end of the scan phase is likely to differ from the actual state of the relation at the end of the scan phase. For example, a relation tuple may have been updated *after* its state was copied by the build process in the scan. Also, additional tuples may have appeared (or disappeared) on pages of the relation *behind* the current scan position of the build process. Deviations of the scanned state from the actual state, like the two cases above, are called *inconsistencies* and are

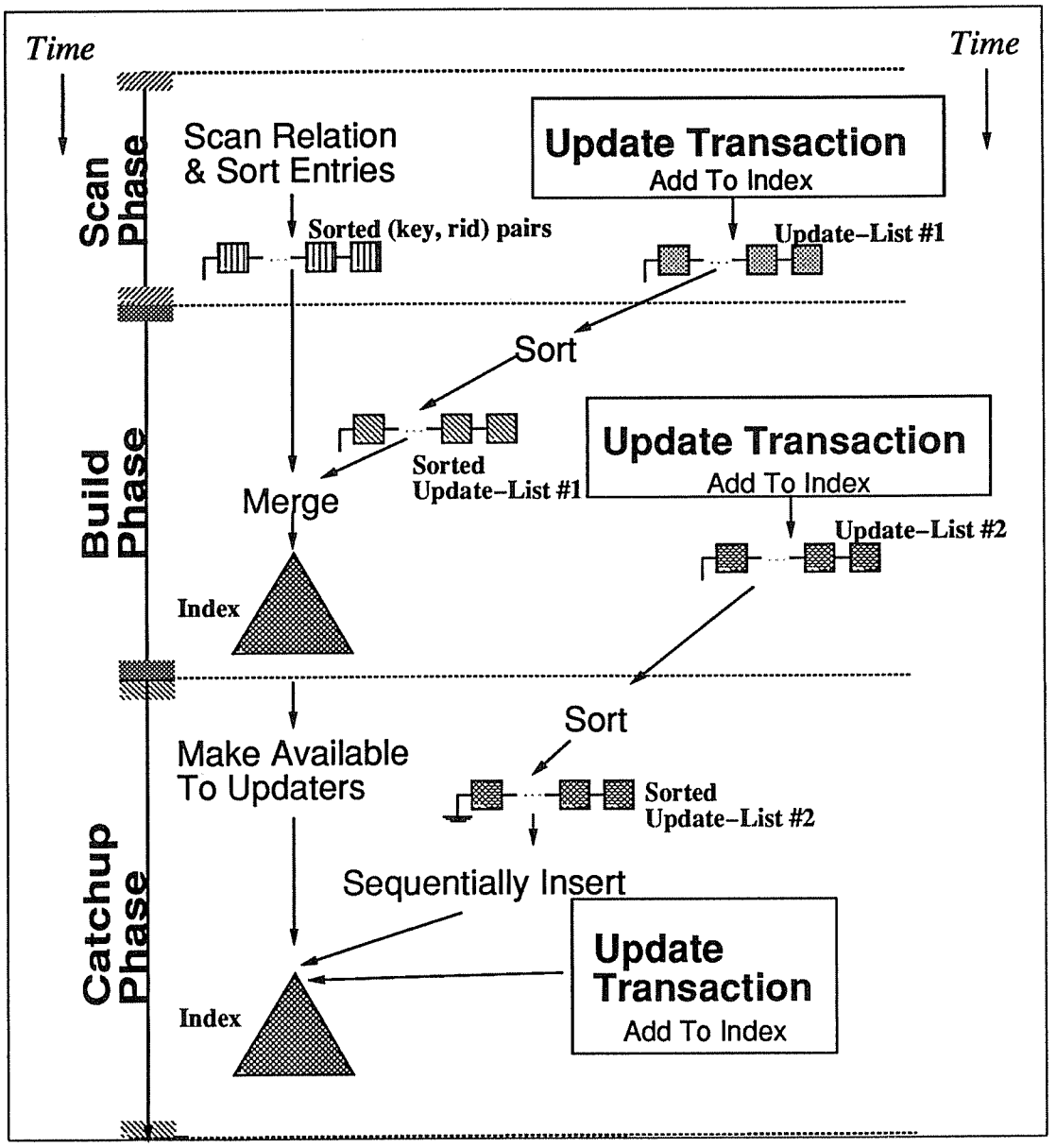


Figure 5.1: On-Line Index Construction Algorithm (List-C-Merge)

corrected by sorting and merging the scanned entries and the update-list.

The logic needed in the sort and merge steps of the build phase to resolve inconsistencies can be understood by noting the following fact. Since the update-list is actually a sequence of updates, only the *last entry* (insert or delete) for a given (key, rid) pair determines if this (key, rid) pair should be present in the merged output. During the sorting of the update-list, all entries except the last one for a particular (key, rid) pair can therefore be discarded. In order to be able to identify the last entry for a (key, rid) pair during sorting, one either has to use a sort that preserves the same input order for duplicates or else tag the update-list entries with a *timestamp* field and sort duplicates based on this field. After sorting, the following actions are taken during the merge in order to remove inconsistencies:

1. If the sorted scanned list contains a particular (key, rid) pair, and the last entry in the update-list is a delete, this (key, rid) pair is omitted from the output of the merge.

2. If the sorted scanned list does not contain a particular (key, rid) pair, and the last entry in the update-list is an insert, this (key, rid) pair is included in the output of the merge.

3. If the sorted scanned list contains a particular (key, rid) pair, and the update-list has no entry for this (key, rid) pair, then the (key, rid) pair is included from the output of the merge.

Performing sorting and merging in the above manner ensures that the intermediate index at the end of the build phase reflects the relation's transaction-consistent state as of the end of the scan phase, as described in Chapter 3 (and proven in the Appendix).

5.2.3 Generalization

Compensation based query processing uses techniques from the on-line index construction algorithm to evaluate queries in a manner that makes them serializable with other transactions in the system. In the case of index construction, it is necessary to fully catch up with the activity that is going on in the database; hence the need for the catchup phase in Figure 5.1. As we shall illustrate in the next section, however, for query processing purposes it is likely to be acceptable to stop with a prior serializable execution state (even though it may not be current at the time the answer is returned to the user).

5.3 Compensation-Based Query Execution

In this section, we shall illustrate the general principles of the compensation-based query processing method. In this approach, a *query* process executes the query on a relation while transactions

updating the same relation are allowed to execute concurrently. The query process here is analogous to the build process that builds the index in an on-line index construction algorithm. The query process executes in two phases, the *scan* phase and the *compensation* phase. The behavior of the query process and update transactions is illustrated in Figure 5.2. The specific actions of the query process and update transactions in Figure 5.2 are for a query that computes a transaction-consistent copy of a full relation. We will later show how these actions can be modified and, more importantly, suitably optimized for the efficient execution of more complicated queries.

During the scan phase, the query process scans the relation's tuples one-by-one to collect the information (from values of the tuples' attributes) necessary for the execution of the query[1]. During the scan, the query process locks a tuple in Share mode only while the tuple is being read, using the same mechanism that is used in cursor stability [Gray79]. For each tuple it encounters in the scan, the query process extracts the data that it needs for query execution. Depending on the type of query being executed, this data is either stored as is for later processing, or it is pre-processed using a function and the result of this function is stored. In addition, for certain queries (including the consistent-copy query illustrated in Figure 5.2), an associated *tuple-id* that uniquely identifies the tuple from which the data was extracted will be stored along with the data. These tuple-ids can be either logical or physical, and they can be re-used; the only requirement is that no two tuples present in the relation at a particular instant can have the same tuple-id, i.e., the tuple-id has to be an *internal* or *external* **key** for the relation. If the attributes needed for query processing already contain a key, we can choose to use that key for identification purposes instead of storing an additional tuple-id field.

During the scan phase, transactions that update the relation being queried perform a special action for every tuple (of this relation) that they update. This action can be as simple as appending the updated tuple to an *update-list* along with an associated tuple-id (in the case of the consistent-copy query), or it can be something query-specific like maintaining aggregate information. For ease of discussion, we assume that transactions collect all of their updates and perform the associated special actions in a critical section at commit time. This assumption is rather restrictive, so we will later show how to achieve the same logical effect while allowing transactions to execute their special actions as their updates occur.

If entries are added to an update-list, the update-list entry for a tuple update contains the type

[1]It should be noted that this scan does not necessarily need to be a simple relation-scan; other efficient access paths may be used as well to perform the scan. For example, if a suitable index exists that *matches* a predicate in the query [Seli79], an index scan can be used to efficiently retrieve only those tuples from the relation that actually satisfy the predicate.

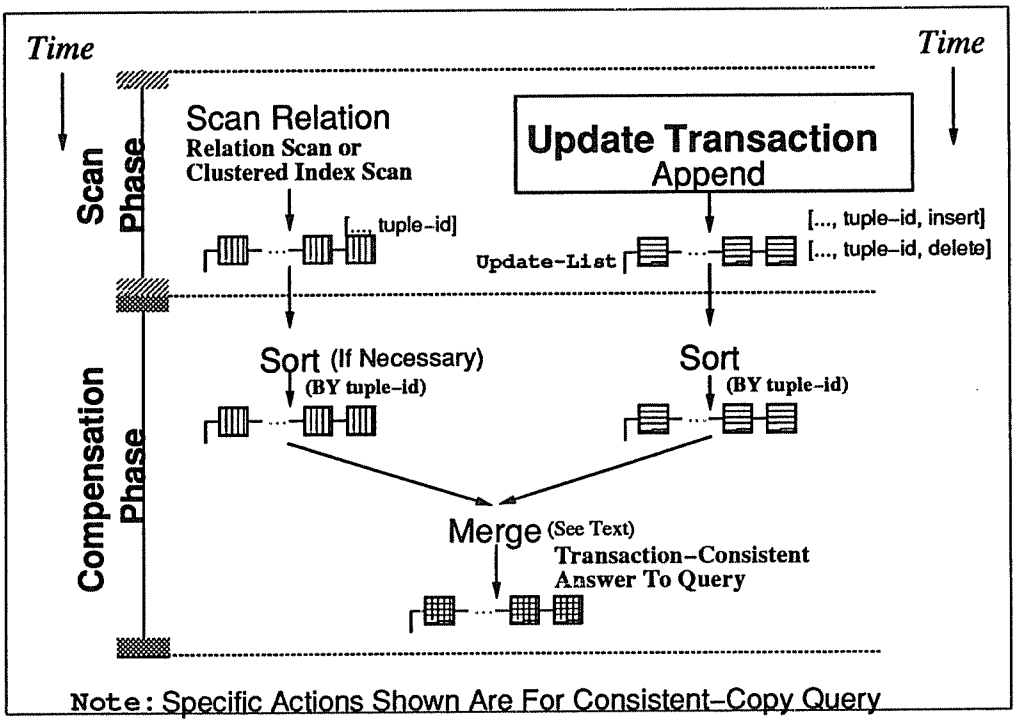


Figure 5.2: Compensation-Based Query Execution

of update (insert or delete[2]), the values of the attributes that are relevant to the query, and an optional tuple-id. At the end of the scan phase, the update-list therefore contains a record of all relevant concurrent updates that occurred during the scan phase. The scan phase ends when the query process finishes scanning all of the relation's tuples.

At the end of the scan phase, the query process enters the compensation phase. In this phase, the query process combines the results from its scan with the changes stored in the update-list as of the end of the scan phase. In order to enter the compensation phase, the query process locks the update-list in read mode, thereby excluding update transactions from the list, and switches the state from *scan* to *compensation*. Unlike the scan phase, update transactions can once again behave normally during the compensation phase of query processing, i.e., they no longer need to perform any special action when they update the relation.

In cases where the scan phase information kept by the query process and/or the update transactions is query specific, then the method of combining information in the compensation phase is also query specific. We shall see examples of such behavior in the next section. In contrast, if both the query process and update transactions store portions of extracted tuples and associated tuple-ids (as in the consistent-copy query of Figure 5.2), then the following strategy is used for merging the scanned entries and the update-list: The query process first sorts the scanned entries

[2]A record modify is deposited in the update-list as a delete followed by an insert.

by tuple-id to create a sorted run. The update-list is then sorted by tuple-id to create another run. The two runs are then merged to resolve inconsistencies like those described in Section 5.2.2[3]. The logic used while merging is similar to that used in the build phase of on-line index construction (Figure 5.1). The merging logic ensures that if an entry for a particular tuple-id occurs in the update-list, then the latest such entry is used to determine the entry that is retained in the merged list. The merged list will thus contain a copy of the relevant information from the relation that is transaction-consistent as of the end of the scan phase. In other words, *all* updates performed by transactions that committed **before** the end of the scan phase are present in the copy, and *no* updates of transactions that committed **after** the end of the scan phase are present. A proof of this assertion follows from the proof of a similar property of on-line index construction algorithms, described earlier.

The above strategy for sorting and merging can be optimized in two ways. First, sorting the scanned entries can be eliminated entirely if the relation was scanned in the order of its tuple-ids. Second, sorting can still be made very efficient if the relation is scanned in the order of *some attribute A* which is not the tuple-id. In the second case, we can first efficiently sort the scanned entries by <A, tuple-id> (which will be efficient due to the order of scanning), also sorting the update-list by <A, tuple-id>. A merge similar to that in Section 5.2.2 is then performed to create a transaction-consistent copy. This strategy avoids a full sort of the scanned entries and automatically makes use of any chosen scanning strategy. We believe that one of these two optimizations should be possible in most (if not all) cases.

Based on the preceding discussion, one way of executing a query on a relation would be to obtain a transaction-consistent copy of the relevant attributes of the relation and then run the query on the copy. However, this may result in a potentially large storage overhead as well as wasting resources for sorting and merging. We shall see in the following section that, in some cases, it is possible to optimize the above naive strategy and execute queries much more efficiently. The optimizations are usually related to the kind of data extracted from tuples by the query process in the scan phase, the information recorded by concurrent update transactions, and the way in which these are combined in the compensation phase to obtain the correct answer to the query.

[3]Typically, one can optimize this strategy by concurrently merging several runs in parallel instead of completely sorting down to two runs before merging [Shap86]. We assume that this optimization will be done in any implementation, but retain the simpler description in our discussion for clarity. Similarly, for all queries, it should be possible to execute the query itself during this final merge step.

5.4 Single Relation Queries

In this section, we demonstrate how SQL queries on a single relation can be efficiently executed in our compensation-based query model. Since the compensation-based model is particularly well suited to executing aggregate queries efficiently, we consider several examples of aggregate queries first. Techniques for processing aggregate queries in relational database systems have been discussed earlier [Epst79]; our work employs some of those techniques as well as extensions needed in the context of our compensation-based model.

Aggregate queries come in two types, *scalar aggregates* and *aggregate functions.* Scalar aggregates compute a single scalar value, like the aggregate **AVG** in query Q1 of Section 5.1. Aggregate functions differ from scalar aggregates in that they return a set of values; the data to be aggregated is logically partitioned by one or more attributes, and aggregation takes place within each partition. Aggregates can also have an optional qualification that restricts the base tuples to which they are applied. Aggregates that commonly occur in database systems include **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**. The **COUNT** function returns the number of tuples or values specified in a query. The functions **SUM**, **MAX**, **MIN**, and **AVG** are applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average of those values.

5.4.1 Scalar Aggregates

A scalar aggregate consists of an aggregate value and an optional qualification. The following steps are needed for processing a general scalar aggregate [Epst79]:

1. Allocate two variables, one for storing the aggregate result and another for storing a count. Initialize both to zero.

2. For each tuple that satisfies the qualification, update the aggregate result and increment the counter.

As an example of computing a scalar aggregate in our model, we shall describe three ways of executing the aggregate query Q1 of Section 5.1. This query computes the average of the salary values over all tuples of the EMPLOYEE relation. In each approach, the scan performed by the query can be either a relation scan or an index-only scan [Epst79].

One way to execute query Q1 is similar to the execution of the consistent-copy query of Figure 5.2, with minor differences. The query process first scans the EMPLOYEE relation collecting [SALARY, tuple-id] pairs. During the scan phase, concurrent update transactions store entries

of the form [SALARY, tuple-id, insert/delete] in the update-list. In the compensation phase, the query process sorts its scanned entries and the update-list, and then merges them to determine the transaction-consistent value of the average salary (without explicitly storing the output of the merge). In spite of the high concurrency achieved this method of computing the average salary may take much more time (due to the overhead of sorting and merging) than a conventional strategy that locks the relation and computes the average during the scan.

The above technique can be improved upon in systems where it is possible to scan the EMPLOYEE relation in a pre-specified order. The order can be, for example, the order in which the relation's records are physically laid out on disk, or it can be ordered on the value of a particular attribute via an index-scan. In the improved technique (given in Figure 5.3), the query process maintains a variable called the *cursor* that gives the position of the last tuple that it has completed reading. At the start of the scan phase, the query process's cursor is positioned at the beginning of the relation (zeroth position). After the first tuple is read, the cursor gets moved to the first position, and so on until the end of the relation is reached. Given the current position of the cursor and a tuple to be updated, it is therefore possible to determine whether the tuple is *behind* the cursor (has already been read) or *ahead* of the cursor (will be read in the future). Update transactions add entries to the update-list *only* for updated tuples that are behind the cursor; for updated tuples that are ahead of the cursor, the update transaction does not add entries to the update-list, as the query will eventually see these tuples. A similar optimization has been suggested for on-line index construction [Moha91] as well as for on-line checkpointing of databases [Pu85]. If updaters follow the approach just outlined, then the query process no longer has to store individual [SALARY, tuple-id] pairs; instead it can directly compute a count and a running average value in the scan phase itself. The update transactions store entries of the form [SALARY, insert/delete] in the update-list, and the query process applies these entries to the average and count variables during the compensation phase. The average value obtained is transaction-consistent as of the end of the scan phase, just like before[4].

The compensation phase in Figure 5.3 can be *eliminated* if update transactions are required to directly update the average and count values maintained by the query process in the scan phase instead of appending their updates to an update-list. Inserts cause values to be added to the average and count variables; deletes cause values to be subtracted.

The last approach to computing the average is certainly the fastest way of executing Q1. In fact, this method has virtually the same overhead as one that executes Q1 using cursor stability,

[4]The proof of this assertion again follows from similar proofs for on-line index construction algorithms that are outlined in the Appendix.

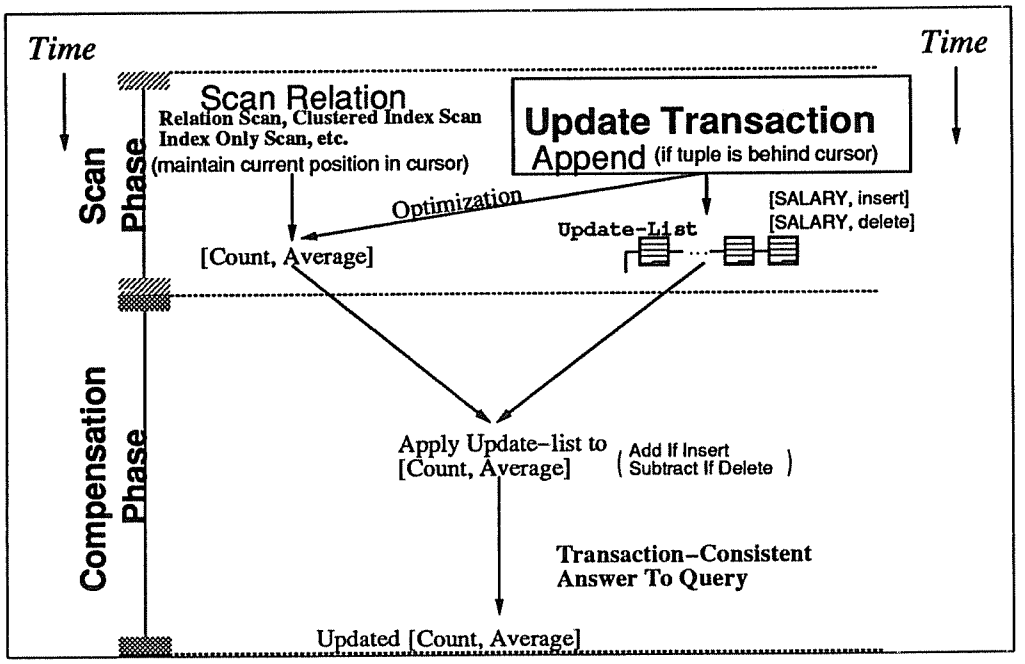


Figure 5.3: Scalar Aggregate: AVG(SALARY) of the EMPLOYEE relation

while providing an answer that is transaction-consistent. The only potential problem with this approach is that update transactions now have to know specific details of the aggregate query that is being executed. It might be more convenient (in terms of system implementation) to have update transactions simply append selected attributes of each tuple (with a tuple-id, if necessary) to the update-list, as in Figure 5.3, as such a facility may already be available in the form of support for executing on-line utilities. Also, an important consideration is that care needs to be taken not to increase the path length of update transactions significantly. The path length for updaters is unlikely to increase significantly in any of the above three cases, though, as the time for appending to an update-list or performing a few arithmetic operations should be small compared to the execution time of a typical transaction. We will see examples later where this path length increase is more significant.

Note that all three of the schemes described above for computing the average can also be used to compute the sum and the count. We will now describe an efficient way of computing the maximum (Figure 5.4). The minimum can be computed similarly. To compute the maximum, the query process scans the relation and collects data of the form [SALARY, tuple-id] that it extracts from the relation's tuples. It also maintains the maximum value of the salary that it encounters during the scan in a variable called MAX. During the scan phase, update transactions append their tuple updates to the update-list, and they also keep track of the largest salary inserted (MAXINS) and deleted (MAXDEL) by any transaction during the scan phase. In the compensation phase,

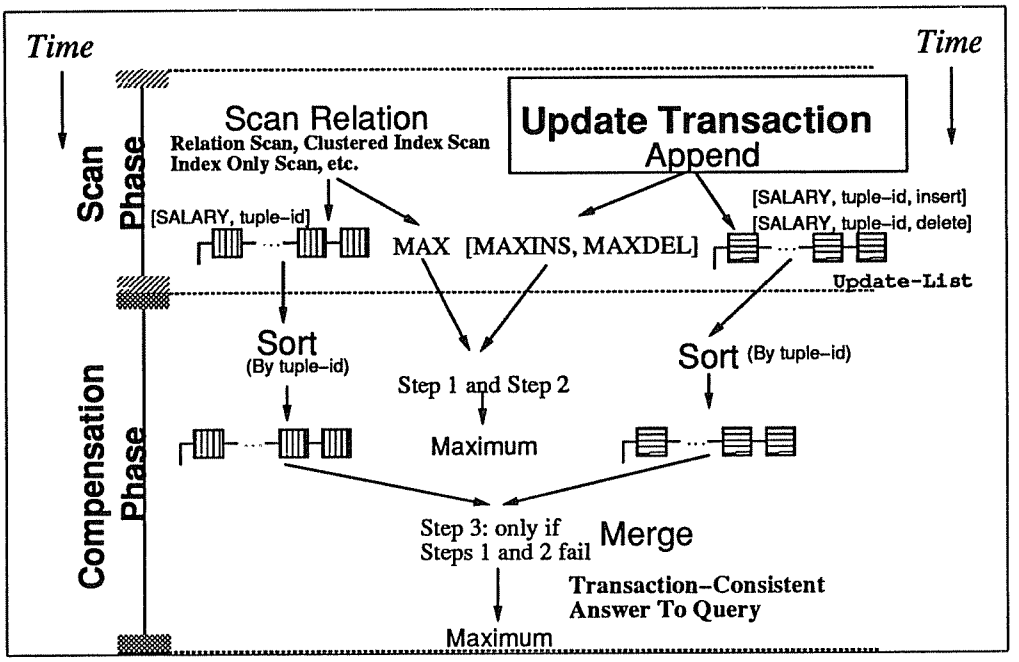


Figure 5.4: Scalar Aggregate: MAX(SALARY) of the EMPLOYEE relation

the query process first performs an optimistic check to see if the correct maximum can be found without actually scanning the update-list. The query process executes the following three steps in order:

1. If MAX is larger than both MAXINS and MAXDEL, then MAX is the desired maximum.
2. If MAXINS is larger than both MAXDEL and MAX, then MAXINS is the desired maximum.
3. If neither of the above conditions hold, then the scanned values and the update-list are sorted and merged to find the correct value for the maximum.

This strategy will perform well if step 3 (which is expensive) is executed rarely compared to steps 1 and 2 (which are inexpensive). This is likely to be the case, as the odds of the maximum value being deleted from a relation during a scan is presumably low. Furthermore, the above strategy is easily modified to further reduce the probability of executing step 3 by maintaining the top few values of the data (along with their tuple-ids) both during the scan and during appends to the update-list. Steps 1 and 2 now become more complicated, but are still quite inexpensive, and the probability that all of the top values will be deleted during the scan (necessitating step 3) becomes extremely low.

5.4.2 Aggregate Functions

Aggregate functions are normally applied to *subgroups* of the tuples in a relation based on certain attribute values, as specified by the query's *BY-list.* SQL has a **GROUP BY**-clause for this purpose. Aggregate functions require the maintenance of an aggregate value, a count field, and the actual BY-list attribute value for each different value of the BY-list attribute. There are two basic algorithms that are commonly used to compute aggregate functions [Epst79]:

1. The first technique scans the base relation, maintaining a temporary relation which has attributes for the count field, the aggregate value, and the BY-list attribute value. When a tuple is scanned from the base relation, the query process updates the aggregate and count values of the tuple in the temporary relation whose BY-list value matches the BY-list value of the scanned tuple in the base relation. If an appropriate tuple does not exist in the temporary relation, a new one is created. When the query process finishes scanning the relation, the temporary relation will contain one tuple for every different BY-list value encountered in the scan.

2. An alternative processing strategy for computing the aggregate function is to first project the base relation on the needed attributes and then sort the result on the BY-list (**not** removing duplicates). The final merge of the sort can easily produce the aggregate value required since all of the entries encountered are clustered on their BY-list values.

Which of the above techniques is better depends on the number of distinct values in the BY-list of the query being executed. For a small number of values in the BY-list, the first method is likely to be more efficient, while the sort-based method is likely to be superior for handling many BY-list values. We now consider an example query that uses aggregate functions and show how such a query can be computed in our compensation-based model.

Q2: For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
SELECT      DEPT_NO, COUNT(*), AVG(SALARY)
FROM        EMPLOYEE
GROUP BY    DEPT_NO
```

The compensation-based execution of query Q2 using method 2 above is a modification of the strategy for the consistent-copy query (Figure 5.2). In the scan phase, the query process stores

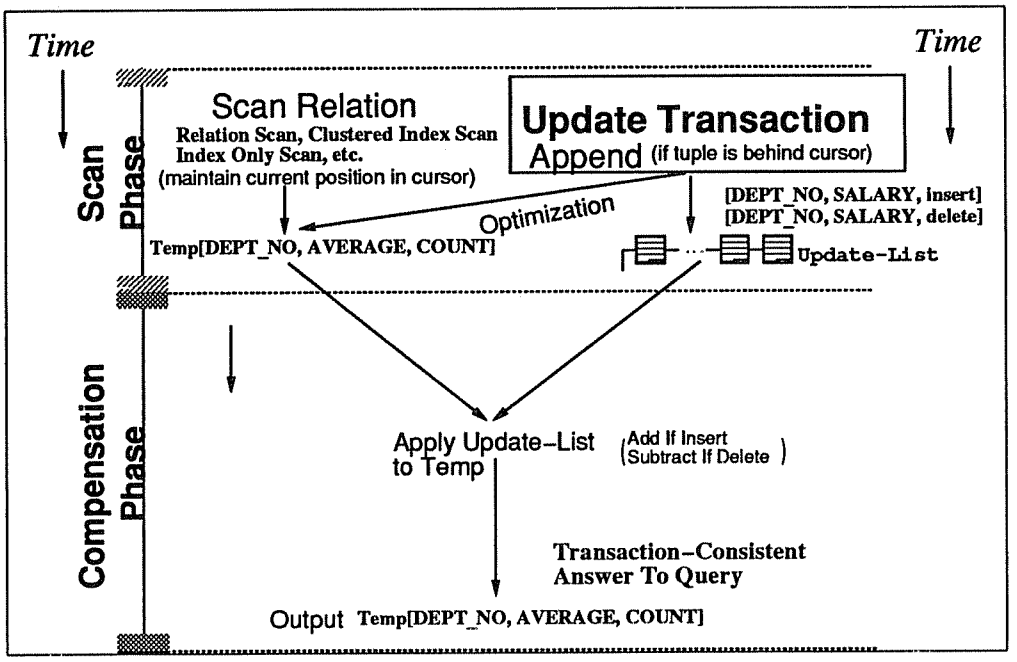


Figure 5.5: Aggregate Function: AVG(SALARY) GROUPED BY DEPT_NO

entries of the form [SALARY, DEPT-NO, tuple-id], while update transactions append similar entries with an additional operation-type field to the update-list. In the compensation phase, the query process sorts both the scanned entries and the update-list by <DEPT_NO, tuple-id> to create two sorted runs. It then merges the runs, removing possible inconsistencies caused during scanning, and computes the average salary and count values grouped by the values occurring in the DEPT_NO field (without actually storing the final merged list). Sorting by DEPT_NO makes it efficient to calculate the answer during the final merge, and merging based on both DEPT_NO and tuple-id, using logic similar to that in Section 5.2.2, still creates a transaction-consistent answer.

We can execute query Q2 more efficiently using method 1 above given the availability of cursor-based scanning (as illustrated in Figure 5.5). The query process now forms a temporary relation whose tuples are of the form [DEPT_NO, Count, Average] by scanning the base relation and operating on the temporary relation in the manner described earlier. When the query finishes scanning the relation, the temporary relation will have one tuple for every different DEPT_NO encountered in the scan. Concurrently, update transactions add entries of the form [DEPT_NO, SALARY, insert/delete] to the update-list. In the compensation phase, the query process applies the entries in the update-list to the temporary relation. Inserts in the update-list cause values to be added to the salary and count attributes of the appropriate tuple in the temporary relation, while deletes cause values to be subtracted. When all values in the update-list have been applied to the temporary relation, it contains the relevant average and count values for the various departments.

In both of the compensation-based methods described above for executing Q2, update transactions append entries to an update-list. There is a third strategy possible in which update transactions would directly operate on the temporary relation used by the query process rather than the update-list. (This is indicated in Figure 5.5 as a possible optimization.) In this strategy, the compensation phase would involve no action at all by the query process. However, a possible drawback of this strategy is that inserting into a temporary relation might increase the path length of update transactions significantly, unlike the fast appends and arithmetic operations that were used in the optimizations for earlier queries. Increasing the overhead for update transactions can cause a significant drop in performance, as illustrated by our study of on-line index construction algorithms, so one has to be very careful in choosing the sort of special actions that can be required of update transactions.

5.4.3 Aggregates with Predicates

Typically, a query in a database system has certain predicates applied to its input and/or output that limit the tuples that are needed to execute the query. Two methods of specifying such predicates in SQL involve the **WHERE** and **HAVING** clauses. Our method of query execution handles such selection predicates by using *filters*. Typically, predicates specified in the **WHERE** clause can be evaluated on individual tuples and are thus implemented by *input filters*, while predicates specified in the **HAVING** clause can only be evaluated after processing the query and are therefore implemented via *output filters*.

To execute a query with a **WHERE**-clause, an input filter is applied to the entries scanned by the query process. This filter only lets through those tuples that satisfy the condition(s) specified in the **WHERE**-clause. Of course, further optimizations can be applied in the initial scan if an index that *matches* a predicate [Seli79] exists on an attribute. For example, an index on the SALARY attribute matches the predicate (SALARY > 40,000), and in the presence of such a predicate in the **WHERE**-clause, an index on SALARY can be used to scan the data efficiently. Furthermore, given that the leaf page of the index stores [SALARY, rid] pairs, the relation does not have to be accessed at all; an *index-only* scan is sufficient [Epst79]. (Note that the rid will be used to lock the tuple while its SALARY attribute is being read from the index in this case.)

Concurrent update transactions can either be required to apply the filter as well, thereby restricting their appends to the update-list, or they can simply append all of their updates to the update-list and have the query process apply the filter to the update-list in the compensation phase. As before, the trade-off here is between increasing the path length of updaters and increasing the overhead for the query process in the compensation phase. Testing a filter with a long string

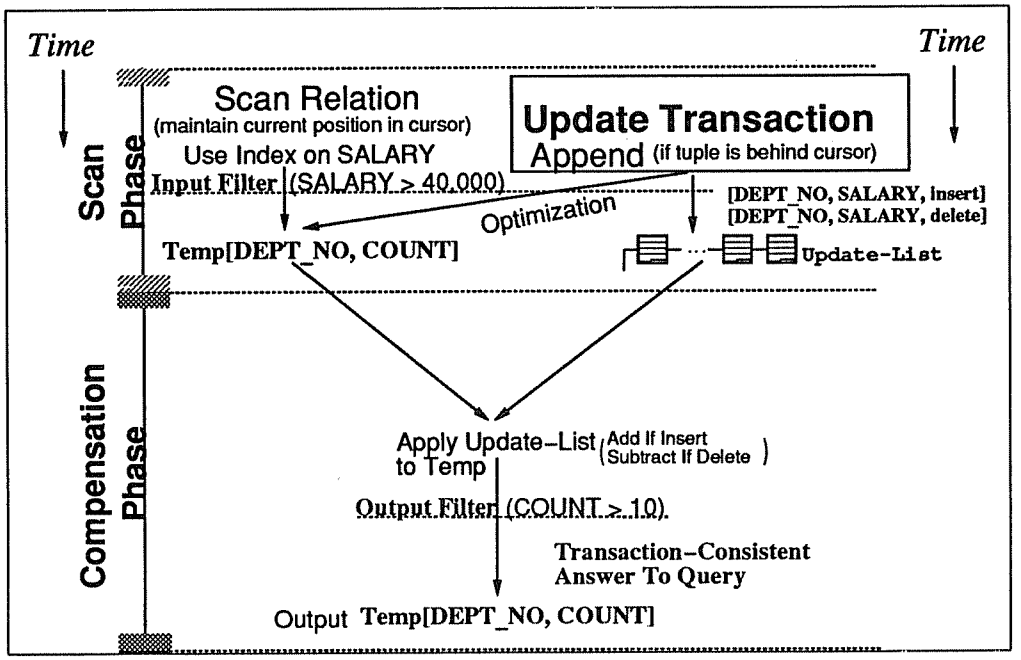


Figure 5.6: Filters: **SELECT** and **HAVING** clauses

of predicates may be expensive, and this may cause an unacceptable increase in path length for updates, in which case it is probably best left for the query process to do.

In order to tie together all of the ideas discussed this far, we will demonstrate how to execute the following query using the techniques discussed in this section.

Q4: What are the department numbers in which there are more than 10 employees, each of whom earns more than 40K, and how many such employees are there in each such department?

```
SELECT      DEPT_NO, COUNT(*)
FROM        EMPLOYEE
WHERE       SALARY > 40,000
GROUP BY    DEPT_NO
HAVING      COUNT(*) > 10
```

The compensation-based execution of this query is shown in Figure 5.6. Note that the index available on the SALARY attribute is used to speed up the scan of the query process. Notice also that the conditions in the **WHERE**-clause are on individual *tuples*, while the conditions in the **HAVING** clause are on *whole groups*. The filter to check the condition in the **HAVING** clause thus comes after the grouping has been done and the aggregates have been computed.

5.4.4 General Single Relation Queries

In our discussion thus far, we have shown how aggregate queries on a single relation can be executed efficiently in the compensation-based model. These are the sorts of queries that often suffer from inefficient execution in current database systems, as described in Section 5.1, and the compensation-based model provides a cost-effective and low-interference alternative for obtaining transaction-consistent answers for these queries. The model itself is more general, however, and can also be used to improve the performance of non-aggregate queries. Consider the following simple query that selects a subset of the tuples in the EMPLOYEE relation:

Q4: Find the names and salaries of all employees who earn more than 40K. The result should be sorted by employee names.

```
SELECT      NAME, SALARY
FROM        EMPLOYEE
WHERE       SALARY > 40,000
ORDER BY    NAME
```

If no index exists on the SALARY attribute, the way that most current database systems would execute this query is to first lock the EMPLOYEE relation in Share mode, then scan every tuple of the EMPLOYEE relation, retaining only tuples that satisfy the condition (SALARY > 40,000) in the answer. After the relation has been scanned, the resulting tuples are sorted, and the Share lock is released before the results are presented to the user. (Note that the Share lock could actually be released before sorting.) Locking the EMPLOYEE relation during the scan causes a long waiting period for update transactions, leading to reduced concurrency, as for the aggregate queries earlier. And, as was the case there, we can instead execute Q4 using the compensation-based model to improve concurrency.

Executing Q4 using the compensation-based approach would work as follows: During the scan phase of query Q4, the query process collects entries of the form [NAME, SALARY, tuple-id] from those EMPLOYEE relation tuples with salary > 40K. Either a relation scan or an index scan (e.g., if a clustered B-tree index exists on the salary attribute) can be used for this purpose. Meanwhile, update transactions add entries of the form [NAME, SALARY, tuple-id, insert/delete] to the update-list. In the compensation phase, the query process first sorts each list on <NAME, tuple-id>[5] to create two sorted runs. Finally, the two runs are merged to generate a transaction-

[5]This sorting strategy is chosen because of the condition in the **ORDER BY**-clause.

consistent answer. The final output of the merge is a set of tuples of the form [NAME, SALARY], ordered by NAME, as required by the query.

The compensation-based method can also be used for computing the projection of a relation on a subset of its attributes, and in fact query Q4 has already illustrated this point. The basic idea is to scan the relation, retaining only the projected attributes along with a tuple-id. Concurrent update transactions deposit their (projected) tuple updates into the update-list along with a tuple-id. In the compensation phase, as usual, the query process sorts the two lists by tuple-id and then merges them to create a transaction-consistent projection.

To summarize, in this section we have shown how the compensation-based model can be used to improve the performance of single-relation queries that would otherwise have to lock massive portions of a relation in order to provide transaction-consistent answers. We now extend our techniques to compute results for more general relational queries, including queries on multiple base relations.

5.5 Join Queries

A join query is a complex query that is (usually) defined on multiple base relations. In addition, it may be nested, and the same relation may occur several times in different roles within the query. A simple but not very efficient way of executing any complex query in the compensation-based model is to first obtain transaction-consistent copies of the various base relations, and then execute the query on these copies. To execute complex queries more efficiently than this, we can integrate the process of producing a transaction-consistent copy of a base relation more tightly with the execution plans generated by the relational query optimizer. This is the strategy that we will use to execute join queries. A third strategy that could be considered for compensation-based query execution is one that actually executes the query on the base relations during the scan phase, obtaining an answer that is not necessarily transaction-consistent. In the compensation phase, one would then try to get a transaction-consistent answer by applying the update-list to the intermediate answer. This third strategy has characteristics similar to keeping materialized views up-to-date when the base relations are updated [Blak86a, Blak86b]; this is a hard problem for arbitrary join queries, even when an initial transaction-consistent copy of the view exists, so we will not consider the third strategy further.

5.5.1 Optimizing Compensation-Based Queries

While executing complex queries in a compensation-based manner, certain query optimization techniques carry through with minor modifications from conventional query processing. One standard query optimization strategy that is virtually always beneficial is *pushing* applicable selects and projects through to base relations so that they can be evaluated before joins. In the scan phase, the query process can easily be made to apply selection conditions on tuples and project out unnecessary attributes as it scans the base relations. Furthermore, applying optimizations like using an index to evaluate a predicate efficiently can be easily incorporated into the scan, as illustrated in Figure 5.6 for query Q4.

Update transaction behavior during complex queries is quite similar to that in the single relation case, except updates to any of several relations must now be recorded. Updaters can either append their tuple updates into one common update-list or into several update-lists (with one for each base relation). Using one list simplifies the special action code in update transactions, slightly increasing the overhead of the query process in the compensation phase (as the list then has to be decomposed into separate lists for the various relations). In our discussion, we will assume that update transactions append to several update-lists, each corresponding to a base relation. Whenever selects and projects are done by the query process during scanning, either the update transactions can also apply the input filters to restrict the tuples that are appended to the update-lists during the scan phase, or they can simply append all base relation tuple-updates to the update-list and let the query process apply the input filters in the compensation phase. Which approach is better is likely to depend mainly on the path length impact for update transactions caused by filtering, rather than on the increase in overhead for the query process, as the update-list for a relation should usually be small compared to the size of the relation and applying filters in the compensation phase should not cause a significant increase in elapsed query time.

Given a join query, the query optimizer generates a plan that specifies the join order, the join method for each individual join, and the selects and projects that have to be done at various stages. It is clear how to perform joins in the order specified in the plan, and it is also clear how to perform selects and projects at various stages of the query. What is not obvious, given a join method, is how to efficiently compute that particular join in a transaction-consistent manner under the compensation-based query execution model. The types of joins that are most commonly used in database systems are the *nested loops*, *sort-merge*, *hash*, and *index* join methods. We shall first consider how to efficiently join two relations R and S (where R is smaller than S) using each of these join methods in the compensation-based model, and we will later describe how a sequence

of joins can be computed through repeated applications of the basic method. In the following discussion of join methods, assume that the query process always stores an associated tuple-id with the data read during its scans.

5.5.2 Nested Loops Join

Nested loops is sometimes used when the smaller of the two relations to be joined, in this case R, fits in memory. In the basic nested loops method, R is read into memory. S is then scanned and, for each tuple in S, all of the tuples that it joins with in R are found (by scanning R in memory) and the result tuples are produced.

In the compensation-based nested loops join method, unlike the basic strategy, the join cannot be performed while scanning S during the scan phase; this is because we need to combine the scanned entries with the corresponding update-list, which can only be done in the compensation phase. Thus, to execute a nested loops join in the compensation-based model, the query process first scans R and S and collects two unsorted lists of R and S tuples, called R_s and S_s respectively. Recall that in order to create transaction-consistent copies, R_s and S_s have to be merged with the corresponding update-list entries for R and S. After accommodating R_s in memory, if there is enough space to construct hash tables in memory for the update-list entries for R and S, we can perform the join with little extra overhead as follows: At the beginning of the compensation phase, two hash tables on tuple-id are created in memory, one for the update-list of R (H_R) and another for the update-list of S (H_S). In these hash tables, only the latest entry for each tuple-id (insert or delete) needs to be retained (for the same reason stated in Section 5.2.2).

After the hash tables H_S and H_R have been populated, R_s is read into memory. As soon as a tuple of R_s is read, H_R is probed to find out if the latest entry for this tuple's tuple-id is a delete. If so, the tuple is discarded; if not, it is retained. The first time a tuple's entry in H_R is accessed, it is marked as having been used. After all of R has been read into memory, there may be entries in H_R that were not used during the scan of R_s. If any of these entries are inserts they should be added to R_s, as such tuples must have been added to the relation behind the query process's scan of R (i.e., they were not seen by the query process). After R_s has been completely loaded into memory, the stored list of S_s tuples is then scanned. A tuple from S_s is joined with R_s only if there is no delete entry for it in H_S. Also, after all tuples of S_s have been processed, the unaccessed insert entries in H_S must be joined with the in-memory relation R_s.

This method of executing a nested loops join requires one extra read and write of R and S (creating R_s and S_s) in the scan phase as compared to the conventional nested loops algorithm. Further optimizations are possible; the extra read and write of R can be avoided if R is scanned

after S, and we only need enough extra memory for the larger of the H_R and H_S hash tables if H_S is constructed after R_s has been completely initialized in memory. The overhead of probing H_R and H_S should be negligible compared to the in-memory join processing overhead, as this probing is done exactly once for every tuple of R_s and S_s. The extra overhead for computing the join in this compensation-based manner might then be justifiable, given the added concurrency obtained by the update transactions. As we will see next, we can do even better in case of sort-merge and hash joins, where the cost for compensation-based execution more closely approaches that of conventional execution.

5.5.3 Sort-Merge Join

In order to execute a sort-merge join, we extend the efficient strategy described in [Shap86]. The first step is to scan R and S, creating sorted runs of size $2 \times M$ where M is the size of memory in pages. Assuming that M is at least $\sqrt{||S||}$, where $||S||$ is the size of the larger relation (in pages), the sorted runs are then merged concurrently by allocating one page of memory to each run of R and S and computing the join during the merge. If $M < \sqrt{||S||}$, some of the R runs are merged among themselves and some of the S runs are merged among themselves until the number of runs is small enough for a final one-pass merge of all remaining runs of R and S to compute the join.

In the compensation-based model, the query process scans R and S and creates a set of sorted runs R_s and S_s, just like the basic sort-merge join strategy. During the sort, however, tuples with the same join attribute value are further sorted by tuple-id. The query process then enters the compensation phase, where it begins by creating similarly sorted runs for the update-lists of R and S. At this point we now have four sets of sorted runs, R_s, S_s, and the runs from the update-lists for R and S. Let us assume that the number of pages in memory is larger than the total number of runs. (This is probably a reasonable assumption given that the update-list sizes are likely to be only a small fraction of the corresponding relations.) In this case, the runs can be merged, as in the basic sort-merge strategy, by allocating one page of memory for each run. However, a tuple from S is joined with a tuple from R only if the latest entries for both the R and S tuple-ids in the corresponding update-lists are not deletes. In addition, tuples from the update-list runs of R and S themselves qualify for the join if they are inserts that were not seen by the query process during its scan. If the size of memory is too small to permit a one-pass merge, then the number of runs of R_s and S_s can each be reduced by merging until a final merge pass is possible.

It should be noted that this strategy for computing the sort-merge join using the compensation-based model is almost as efficient as the basic strategy, as the main extra work required is the creation of sorted runs from the (presumably small) update-lists.

5.5.4 Hash Join

There are three common types of hash join: the simple hash join, the GRACE hash join, and the hybrid hash join [Shap86]. Here we demonstrate how to adapt the GRACE hash join algorithm to the compensation-based model. The GRACE algorithm first scans R and partitions it into roughly equal subsets such that the hash table for each partition of R will fit in memory. The algorithm then scans S and creates partitions of S corresponding to those for R. Finally, each individual partition of R is read into memory, with a hash table being constructed on its join attribute, and the join is computed by probing the in-memory hash table with tuples from the corresponding partition of S.

Implementing the GRACE algorithm in the compensation-based model is straightforward. The query process scans both R and then S during its scan phase, obtaining R_s and S_s, storing the necessary partitions of R_s and S_s during this phase. In the compensation phase, the query process firsts scans the update-lists for R and S and partitions them using the same hash function used for R_s and S_s. Then, for each partition of R_s, the query process starts by reading the corresponding partition of the update-list of R into memory and constructing a hash table using its entries. The query process then scans the partition of R_s itself and continues building the hash table with entries from R_s; tuples of R_s are discarded whenever a delete entry is encountered in the hash table. After completing the hash table on the current partition of R_s, the build-process scans the corresponding partition of the update-list of S and builds a hash table H_S based on its entries. (It is assumed that H_S also fits in memory.) The join is then performed by scanning the relevant partition of S_s; before probing the hash table of R_s with an S_s tuple, however, H_S is probed to see if there is a corresponding delete entry. Finally, after all tuples in the partition of S_s have been exhausted, any remaining unaccessed insert tuples in H_S must be joined with the current partition of R_s. This process is repeated to join each partition of R and S.

As was the case for sort-merge join, the compensation-based GRACE hash join algorithm has nearly the same cost as the basic GRACE algorithm. Creating hash partitions for the update-lists is not likely to add much overhead, as these lists are likely to be small. It should be noted that the techniques used for GRACE can be adapted to implement the simple and hybrid hash join algorithms as well. However, when implemented in the compensation-based model, these later two algorithms will require some extra reading and writing of (at least one bucket of) the base relations as compared to their conventional counterparts. This is due to the same reason as in the nested loops join method, i.e., because no join output can be generated during the scan phase of compensation-based query processing.

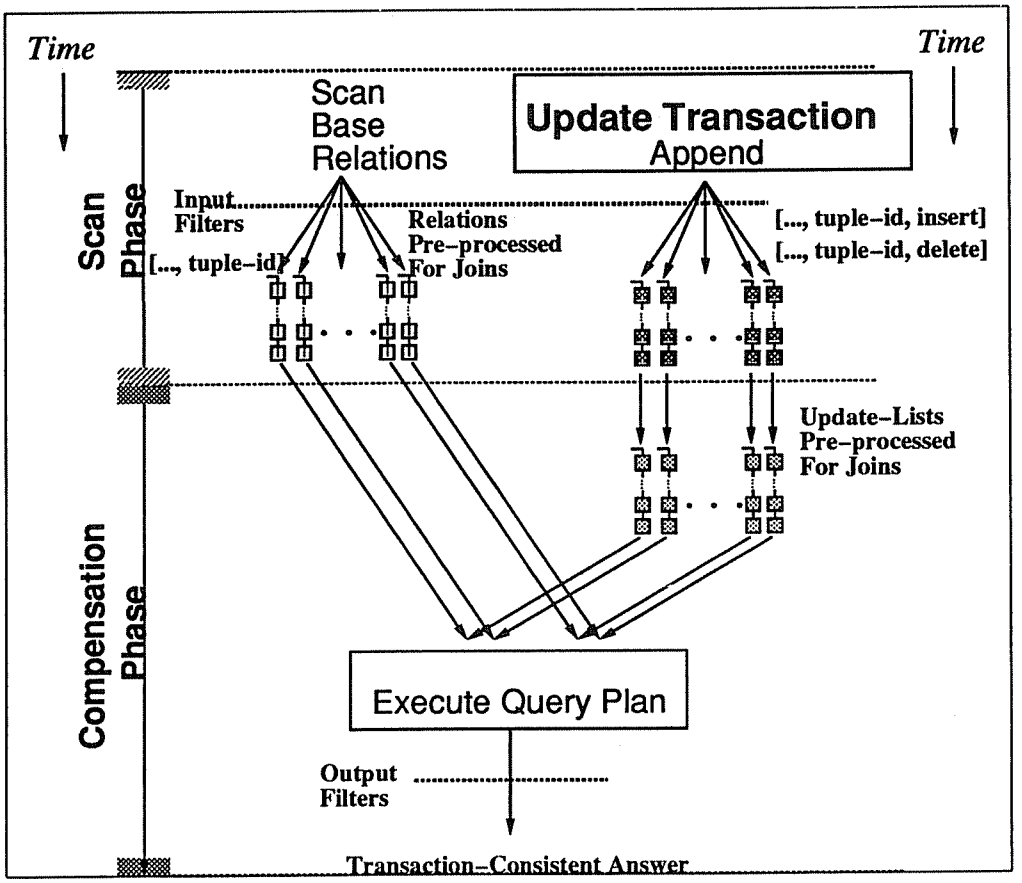


Figure 5.7: Complex Query Execution

5.5.5 Index Join

In a conventional index join, for each tuple in R, an index existing on the join attribute of S is used to efficiently extract tuples from S. This type of join is usually chosen when R is quite small (as otherwise the implied index I/O, especially if the index in unclustered, is too costly). Due to the fact that the scanning of one relation (S) is dependent on data from another relation (R), and that both relations are being updated simultaneously, it appears difficult to efficiently adapt the index join method to the compensation-based query execution paradigm. Fortunately, the lack of index join support is unlikely to matter all that much for the kind of long-running queries that are likely to benefit from our compensation-based model. This is because, as mentioned above, index joins are primarily used when the join selectivity and the number of tuples in R are both small. Such conditions are more likely to arise in conventional short-duration queries than they are in the longer queries that compensation-based query processing is intended to help with.

5.5.6 Multiple Joins

In the discussion so far, we have described how to execute two-way joins using various join methods. The generalization to many-way joins is straightforward. During the scan phase, all participating relations are accessed (Figure 5.7). Depending on the join order and the join methods of the various relations in the query, the outcome of scanning will vary. For example, if a relation is to take part in a sort-merge join, then its sorted runs are created while it is being scanned. If it is to take part in a GRACE hash join, it is partitioned into buckets based on hashing during the scan phase. In the compensation phase, the query process will process each relation's update-list according to the type of join that the corresponding relation is taking part in. Intermediate join results can be handled just as in conventional query processing, as these results are guaranteed to be transaction-consistent.

5.6 Implementation Considerations

As we mentioned in Section 5.1, executing queries in the compensation-based model requires certain special capabilities to be present in the DBMS for handling the query process and any update transactions that affect its data. Update transactions have to be aware of the presence of any compensation-based queries and must take appropriate actions when updating their data. The required state information can be placed by the query process in the system catalogs, where update transactions generally go to find information about auxiliary data structures like indices that they are required to update along with the data. (The update-list can be added as just another auxiliary data structure in the catalog entry corresponding to the relation to be updated.) In some current DBMSs, the system catalogs are not accessed by transactions at run time; in such systems a special data structure will have to be implemented for this purpose.

There is an important performance issue pertaining to the time when an update transaction should execute its special code for each update. We assumed earlier that this is done at commit time, but this would mean that transactions have to save a list of all of their updates until commit time. This is probably an unreasonable requirement and may be unenforceable in practice. Fortunately, equivalent behavior can be obtained by implementing the update-list much like a log. Transactions can append their updates to the update-list, tagged with their transaction id, at the time when they actually update the data. In addition, at commit time they should append a commit or abort record. In the compensation phase, the query process can now analyze the update-list to determine which of the updates are committed and which are aborted (or incomplete); the query process will eliminate the updates of all but the committed transactions from the list and then proceed

as before. This log-like approach works in the case where an update-list is actually maintained, but there is a slight problem if updaters operate on variables instead. For example, consider the optimized version of the query for computing the average (Figure 5.3), where update transactions directly update the average and count variables as they update relation records. In this case, they will have to do the corresponding inverse operations if they abort, and the query process will have to wait in the compensation phase for all running transactions that have updated the average and count values to either commit or abort.

In our discussions thus far, we have assumed that the query process executes under cursor stability (also called level 2 consistency) [Gray79], in which case it reads only committed data from other transactions. This form of execution is sufficient for producing transaction-consistent answers, but is not strictly necessary. Recall that the build process in the on-line index construction algorithms of Chapter 3 performed dirty reads (level 1 consistency) and yet was still able to build a consistent index. A similar execution strategy can be used for the query process in compensation-based query processing. If the query process performs dirty reads, in order to create a transaction-consistent answer, it *must* wait in the compensation phase for all of the running transactions that have updated the update-list (and/or other query-specific information) to either commit or abort. In other words, the query process will serialize itself *after* all transactions whose dirty data it could have read, thus ensuring that its results are transaction-consistent.

Finally, given a query, the compensation-based query processor has to optimize the query and then execute it according to the optimized plan. The optimizer will need to use slightly different cost models for compensation-based plans, as the costs of certain joins (e.g., nested loops) are somewhat higher in this model. Also, some conventional execution plans (e.g., index join) are not available for compensation-based query processing use.

5.7 Pre-Specified Time Queries

In transaction processing applications, there is sometimes a need to run a class of queries whose results are valid as of some pre-specified time. For example, banks periodically mail out statements to their customers that contain records of all transactions that took place on the customer's accounts during the preceding period. Companies take stock of their inventories periodically when they need to know exactly what items they have in stock. All organizations have a yearly audit when a balance-sheet of all of their activities for the year are listed and accounted for. Typically, these activities are done on a certain fixed day of the month or year. With DBMSs being widely used for record-keeping, the above examples translate into queries over on-line databases. An example

of a time-based query is given below.

Q5: Print the names and salaries of all employees of the company (ordered by salary) as of the end of the year, 1991. The required answer can be obtained by running the following SQL query on the system, with the restriction that the answer obtained be transaction-consistent as of midnight on December 31, 1991.

```
SELECT      NAME, SALARY
FROM        EMPLOYEE
ORDER BY    SALARY
```

Interestingly, it turns out that compensation-based query processing can be used to execute such queries efficiently. In order to understand how this is possible, recall that the answer obtained for a query in the compensation-based model (Figure 5.7) is transaction-consistent as of the end of the scan phase. In all of the examples thus far, we have assumed that the scan phase ends as soon as the query process finishes scanning the base relation(s) used in the query. (This is certainly the best way to execute a normal query as soon as possible.) To execute a query whose results are valid as of some pre-specified time, however, we can force the end of the scan phase to occur at the time specified in the query. On completing the scan of the base relation(s) in the scan phase, the query process can then simply wait until the pre-specified time before switching to the compensation phase. In order for this strategy to work, the query process has to ensure that its scan of the relations is completed by the pre-specified time; this can be done by starting the query sufficiently far in advance of that time. The above strategy can also be optimized further: Instead of waiting from the end of the scan phase until the time pre-specified in the query, the query process can also perform some or all of any pre-processing that it needs to do on the scanned data.

5.8 Related Work

The work that is most closely related to compensation-based query processing is the work on *on-line index construction algorithms* in Chapter 3 of this thesis and [Moha91]. In fact, it was our work on techniques for incrementally building an index that led us to discover the techniques described in this chapter. Compared to that work, the method described here is more general and handles arbitrarily complex queries. The process of scanning the base relations in the compensation-based method is also akin to taking a *fuzzy dump* of the relations [Gray79]. After the dump is taken, the log generated during the fuzzy dump is merged with the fuzzy dump to produce a *sharp dump.*

This is quite similar to the way that queries execute in the compensation-based model in order to generate transaction-consistent answers.

Concurrency control algorithms based on *transient versioning* (e.g., [Chan82, Agra89, Bobe92]) are also related to compensation-based query processing. In transient versioning algorithms, prior versions of data are retained to allow queries to see slightly outdated but transaction-consistent database snapshots. A transient versioning mechanism operates uniformly on all data in the database whenever it is updated, keeping copies of each updated record or page. In contrast, in the compensation-based model, a query (together with cooperative updaters) essentially creates its own consistent version of the relevant underlying data while executing. In some sense, our model can be viewed as an approach based on *semantic versioning.*

5.9 Conclusions

In this chapter, we have described a novel and highly concurrent approach to executing queries in a database management system. The proposed approach is called compensation-based query processing. This approach achieves very high concurrency by locking data only briefly, at the tuple-level, while still delivering transaction-consistent answers to queries. We believe that such a model of query processing will make it possible for long-running queries, which usually run under a reduced degree of consistency in current DBMSs, to obtain transaction-consistent answers without adversely affecting system performance. Compensation-based query execution can co-exist with conventional query processing, and a cost model similar to that used for optimizing conventional queries can be used for optimizing queries under the new model as well. Finally, it appears that compensation-based query processing can be implemented without too much extra effort in a conventional DBMS (at least once the conventional DBMS has been modified to support on-line index construction).

Since we have not analyzed the performance of the proposed scheme, one potential area for future work is to characterize the performance of a system when queries are executed in a compensation-based manner. The trade-offs involved in increasing the work of the query process versus increasing the path length of updaters have to be studied in more detail. Furthermore, the performance of compensation-based queries needs to be compared with the performance of the same queries when they are run with conventional (serializable and non-serializable) concurrency control methods or with concurrency control methods based on transient versioning. Another possible topic for future work would be to determine how to implement intermediate recoverable points for particularly long-running compensation-based queries so that they need not re-execute from scratch at every crash. Finally, the ultimate test would be to implement this model of query processing in a real DBMS.

Chapter 6

Conclusion

6.1 Summary of Results

The major results of this thesis are summarized below. The results are subdivided into three categories: performance of B-tree concurrency control algorithms, on-line index construction, and compensation-based query processing.

Performance of B-Tree Concurrency Control Algorithms

The main conclusion of the B-tree concurrency control study of Chapter 2 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 showed gains in throughput under plentiful resource conditions. The reason for the excellent performance of the B-link algorithms was 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 formed at high MPLs when the workload contained a significant percentage of updaters. Moreover, the overhead that the B-link algorithms incurred in very high data contention situations were link-chases, which turned out to be inexpensive. We also found interesting differences in the behavior of the optimistic and pessimistic algorithms among themselves.

The results of the performance study of B-tree concurrency control algorithms indicate that for very high transaction rates, only the most concurrent algorithms are likely to be acceptable. Algorithms which simplify the complexity of coding at the cost of concurrency are bound to be unsatisfactory in some operating regions.

On-Line Index Construction

We presented a range of solutions to the important problem of on-line index construction in Chapter 3. In particular, we described two families of on-line index construction algorithms. These on-line algorithms vary in the sort of data structures that they use to store the concurrent updates (list or index), the strategies used to actually build the index from leaf-level entries, and the degree of concurrency allowed for concurrent updates. The algorithms trade off, to varying degrees, increased building time for increased updater throughput. Proofs of correctness of these algorithms can be found in the Appendix. The prospective ease of implementation of our algorithms is an important reason why we believe that next generation databases will incorporate such algorithms in their repertoire.

In Chapter 4 we presented the results of a comprehensive performance study of the proposed index construction algorithms that was conducted in order to determine the best algorithm for use in a DBMS. To aid in our study, we employed a performance metric that measures the loss to the system due to interference between concurrent updaters and the index building process. An important property of the loss metric is that it enabled us to directly compare the on-line algorithms with the best off-line algorithm as well as amongst themselves. An important conclusion of this study was that in most cases, the fully on-line algorithms (which have no exclusive phase) performed very well and did better than the partially on-line algorithms (which had a concurrent relation scan phase but an exclusive build phase) or the off-line algorithm. The list-based fully on-line algorithms were found to perform better overall than the index-based alternatives due to the smaller overhead that they imposed on concurrent updaters. The fully on-line list-based algorithm that uses the merge strategy (i.e., List-C-Merge) appeared to be a very good candidate for use in a real system.

Compensation-Based Query Processing

Chapter 5 showed how the techniques used in the on-line index construction algorithms can be generalized to efficiently execute long-running queries, which are currently handled unsatisfactorily in conventional DBMSs. This led to a novel and highly concurrent approach to executing queries that we call compensation-based query processing. Compensation-based query processing achieves very high concurrency by locking data only briefly, at the tuple-level, while still delivering transaction-consistent answers to queries. We believe that such a model of query processing will make it possible for long-running queries, which usually run under a reduced degree of concurrency in current DBMSs, to obtain transaction-consistent answers without adversely affecting system performance. Compensation-based query execution can co-exist with conventional query

processing, and a cost model similar to that used for optimizing conventional queries can be used for optimizing queries in the new model as well. Furthermore, it appears that compensation-based query processing can be implemented without too much extra effort in a conventional DBMS (at least once the conventional DBMS has been modified to support on-line index construction). Finally, extrapolating from our performance results for on-line index construction algorithms, where fully on-line algorithms performed the best, it seems likely that compensation-based on-line query processing will be cost-effective to use in a DBMS.

6.2 Future Work

In spite of the steps taken in this thesis to design efficient on-line processing techniques for next generation DBMSs, important opportunities for future work remain.

In our study of B-tree concurrency control, we have considered only one aspect of concurrency control on B-tree indices, namely, transactions that perform single B-tree operations. There are still important open questions related to how B-tree concurrency control algorithms perform when they occur as part of a larger transaction:

1. Several strategies have been suggested for obtaining serializable executions of transactions that perform multiple B-tree operations. For example, a transaction could hold an extended lock on the index leaf page, on a single slot in the leaf page, or on a single record id or a key value. In addition, such a locking strategy can be combined with deferred index updates (where index updates are grouped together and applied at commit time). The relative performance of these alternative strategies is an open question.

2. The aforementioned lock holding strategies are closely linked to recovery strategies. Few comprehensive recovery strategies have been proposed for B-trees [Moha89, Lome91], and the interaction of concurrency control and recovery in B-trees has not yet been satisfactorily studied.

3. Furthermore, special types of queries such as range scans (particularly those that read a large amount of data) may interact differently with long term lock holding strategies and recovery strategies than conventional transactions (which usually access a small amount of data) would.

Apart from using the alternatives mentioned in point 1 for executing range scans, a new alternative that could work very well is to forego long-term locking and turn to compensation-based execution of such queries.

Turning to on-line index construction algorithms, recall that all the algorithms described in Chapter 3 apply to the construction of B-tree indices. An interesting and useful avenue of future work will be to extend our on-line index construction strategies to work for indices other than B-tree indices. Most of the work in such extensions will involve the development of efficient strategies to combine the scanned entries with the concurrent updates. While we found the merge strategy to be efficient in B-trees for combining the scanned entries with the concurrent updates, this may not be the case for other indices like a hash index. Nevertheless, we believe that the concurrency control techniques employed in our index construction algorithms can be directly applied for the on-line construction of other types of indices.

The area of compensation-based query processing has several opportunities for future work, the first being a performance analysis of the new approach. While it is clear that the compensation-based approach to query processing is highly concurrent, the following performance issues need to be studied in order to clearly establish its viability relative to other existing methods.

1. The trade-offs involved in increasing the work of the query process versus increasing the path length of updaters have to be studied in more detail. Such path length increases can have a significant impact on performance, as evidenced by the behavior of the index-based on-line index construction algorithms in Chapter 4. Any implementation of compensation-based query processing therefore has to ensure a minimal increase to the path length of the concurrent transaction workload.

2. The performance of compensation-based query execution needs to be compared with both conventional serializable and non-serializable query execution; this would provide an accurate picture of the costs and benefits of obtaining serializable results via the compensation-based model.

3. The performance of compensation-based query execution needs to be compared with concurrency control strategies that use transient versioning. Such a comparison would examine the tradeoffs of uniformly versioning all the data (as is done in transient versioning strategies) versus selectively versioning only the data that is being concurrently used by active queries (which is the case in compensation-based query processing). In addition, such a study should also compare the increases in update transaction path lengths caused by the two strategies.

Another topic for future work would be to determine how to provide intermediate recoverable points for particularly large compensation-based queries so that they need not re-execute from scratch at every crash. Parallel and distributed execution of queries using the compensation-based method

is another interesting topic for future work. Finally, the ultimate test will be to implement this model of query processing in the context of a real DBMS.

Appendix A

Correctness Proofs for On-Line Index Construction Algorithms

A relation consists of records, each of which is identified by an identifier called its record id (rid) which is unique within all records of the relation. The set of rids in a relation R is given by $S(R)$. The value of the attribute A of the record associated with rid r is denoted by $A(r)$. A relation is assumed to be stored in pages 1..N. We assume that N is a large constant and is the maximum possible size of the relation. The mapping $M(R)$ maps individual elements in $S(R)$ to a value between 1 and N, i.e., $M(R)$ gives the page where a record with a particular rid resides. The mapping $M(R)$ is a many-to-one and into function, i.e., every rid is mapped to exactly one page, and there can be pages that are not mapped from any rid (empty pages of the relation). To keep our discussion simple, we will assume further that, given a set of rids of a relation, the mapping from rids to pages for this set can be determined from fields of the rid itself. Therefore, $S(R)$, the set of rids, also determines $M(R)$ completely. (In other words, we assume physical rids, though it is straightforward to extend the proofs for logical rids as well.) An index on an attribute A of a relation R ($Index[R, A]$) consists of a set of entries, each of the form (k, r), where $k \in Dom(A)$ (where $Dom(A)$ is the set of all possible values for attribute A) and $r \in S(R)$. Finally, the set of rids $S(R)$ can vary with time, and the value of the set at time t is denoted by $S_t(R)$.

Definition A.0.1 A *relation update* consists of the 3-tuple (r, v, o) where r is the record id, v is the value of all of the fields of the inserted or deleted record, and o is one of i (insert) or d (delete). An *index update* is also a 3-tuple (k, r, o) where k is a key value, r is a record id, and o is one of i (insert) or d (delete). Also, every relation update $u = (r, v, o)$, gives rise to a corresponding index update $t(u) = (A(r), r, o)$ for every indexed attribute A. □

From Definition A.0.1, it follows that modifying a field of a record is modeled as two operations, a delete of the old record followed by an insert of a new record with the same record id but a different

value for the modified field. Such a two-operation modification will result in two corresponding index updates. Recall from our discussion in Section 3.4 that an update transaction performing a relation update (r, v, o) holds an Exclusive lock on the page where r resides. This lock is released after the update on the page is completed. The release of this lock is assumed to be done in a critical section in the lock manager.

Definition A.0.2 A relation update is defined to *occur* atomically at the time when the Exclusive lock on the modified relation page is released. Similarly, an index update is defined to *occur* atomically at the time when the Exclusive lock on a modified leaf page is released by the B-tree concurrency control algorithm (Section 2.2). □

Definition A.0.3 If s and t are times such that $s < t$, then $U_{s,t}$ is the sequence of updates to relation R in the time interval between s and t, in increasing order of lock release time. For the sequence $U_{s,t}$ of relation updates of length n, we define a sequence $T_{s,t}$ of index updates (to an index on attribute A) also of length n as follows: if the i^{th} entry of $U_{s,t}$ is u, then the i^{th} entry of $T_{s,t}$ is $t(u)$. □

Definition A.0.4 An index $I[R, A]$ is *consistent* with respect to the relation R at time t if (i) for every rid r in $S_t(R)$, the entry $(A(r), r)$ is present in $I[R, A]$, and (ii) for every entry (k, r) in the index, $r \in S_t(R)$ and $A(r)$ is k. □

A.1 List-X-Basic Algorithm

Let t_s denote the time at the start of the scan phase, step 1 of Figure 3.6. Let t_b denote the start of the build phase, i.e., the Exclusive lock on $U[R, A]$ (step 5 of Figure 3.6) is granted at time t_b. Finally, let t_f be the time when the index construction process terminates (step 9 of Figure 3.6). Recall that, in this algorithm, update transactions that make a relation update u add the corresponding index update $t(u)$ to the update-list $U[R, A]$ using an Exclusive lock (step n, Figure 3.6). Therefore, just like for relation updates, the append to the update-list can be viewed as an atomic occurrence at the time when the Exclusive lock on the list is released. The update-list $U[R, A]$ thus contains a sequence of index updates ordered by this lock release time.

Definition A.1.1 The time when a page p is scanned by the index construction process is denoted by t_p. For all pages p in a relation R, $t_s < t_p < t_b$. □

Lemma A.1.1 *If a relation update u is performed on a page p at any time t, $t_s < t < t_p$, then $t(u)$ will be added to U[R,A] before time t_b.*

Lemma A.1.2 *The sequence of index updates present in $U[R, A]$ at t_b is T_{t_s,t_b}. Recall that T_{t_s,t_b} is the sequence of index updates corresponding to the actual relation updates U_{t_s,t_b} that took place in $[t_s, t_b]$ (see Definition A.0.3).*

Proof: Both of the lemmas above can be proved from the fact that an updater in the List-X-Basic algorithm will release its Exclusive lock on the modified relation page only **after** getting an Exclusive lock on the update-list, U[R,A] (Section 3.7).

Since the build process has not scanned page p before time t, it can scan p only after the updater has acquired the Exclusive lock on U[R,A] and released its page lock (since the builder needs a Share lock on p to read it). So, before the time when the build process gets the Exclusive lock in step 5 of Figure 3.6 (t_b), the index update $t(u)$ will be in the update-list. This proves the first lemma above.

It also follows from the lock-chaining strategy described above that the order of releasing relation page locks (which determines the relative order between relation updates) is the same as the order of releasing update-list locks (which determines the relative order of index updates). This, along with the additional fact that the index update is registered in $U[R, A]$ immediately after the relation update, proves the second lemma. □

Theorem A.1.1 *At time t_f, the index $Index[R, A]$ is consistent with respect to R.*

Proof: We will prove this theorem by proving that, for any page p in the relation, the entries in index $Index[R, A]$ at t_f accurately reflect the state of that page at t_f. Since we assume physical rids, the rid of a record determines uniquely the page of the relation where the record is stored. Therefore, of the sequence of updates in U[R,A] at t_b, it is possible to identify the sub-sequence of index updates that were caused by relation updates to page p. This sub-sequence of updates can be further divided into the sequence of updates B_p that took place before t_p (the time when this page was scanned by the build process), and the sequence A_p of updates that took place after t_p. Consider the execution of step 7 (in Figure 3.6) of the build process. For page p, the sequence of index updates in B_p are first redone on the contents of page p, followed by the (new) index updates in A_p.

For page p, $Index[R, A]$ at t_b is consistent w.r.t. the state of the relation page p at time t_p. Consider a record id r that does not occur in B_p. For such a record, applying B_p to $Index[R.A]$ does not change the entry corresponding to r in $Index[R, A]$. Now consider a record id q that does occur in B_p. In this case, applying B_p leaves the state of this record consistent with the last index update in B_p for q. But, this will be the same as the state of this record id as read from the page itself at t_p because of Lemmas A.1.1 and A.1.2. Therefore, at the time when all updates from B_p

have been applied and no updates from A_p have begun, the state of the page p in the index is consistent with the state of p at time t_p when it was scanned i.e., after applying all updates from B_p, $Index[R, A]$ contains the same entries obtained for page p during the scan. Applying the index updates in A_p thus cannot cause any consistency problems, as these updates happened after page p was scanned and will be applied in the same order as the relation updates themselves.

The same argument can be used for all pages in R. Furthermore, no updates will be lost since it is clear from Lemma A.1.2 that all relation updates that have completed before t_b will be present in U[R,A]. Also, no updates can complete in $[t_b, t_f]$ due to the Exclusive lock. $Index[R, A]$ is therefore consistent w.r.t R at t_f. □

The above proof can be extended to prove Theorem A.1.1 for the List-X-Sort algorithm as follows. Lemmas A.1.1 and A.1.2 are true here also. To prove the theorem, one has to show that the sort (step 6a in Figure 3.6) does not change the proof above. This follows from the fact that for every (key, rid) pair e in the update-list $U[R, A]$, the sort retains the latest entry for e that was appended to $U[R, A]$, thus causing the same effect for e as would be obtained by sequentially inserting the unsorted entries in the same order in which they were appended to $U[R, A]$.

For the List-X-Merge algorithm, the proof of Theorem A.1.1 directly follows from the proof for the List-X-Sort algorithm. The reason is that merging the sorted scanned entries with the sorted update-list in List-X-Merge (step 7, Figure 3.7), is identical to creating an index out of the sorted scanned entries first and then sequentially applying the entries from the sorted update-list to this index (as in List-X-Sort, step 7 in Figure 3.6).

A.2 List-C-Basic Algorithm

In this algorithm, as before, t_s and t_b are times that denote the start of the scan and build phases, respectively (steps 1 and 6 respectively in Figure 3.8). Time t_f is again the time when the index construction process is completed (step 14). In addition to these, we denote as t_c the time of the start of the catchup phase (step 10 in Figure 3.8). These times are illustrated in Figure A.1. We further assume the correctness of the B-tree concurrency control algorithm used for performing concurrent operations on the index. Lemmas A.1.1 and A.1.2 are true here as before. Our goal now is to prove Theorem A.1.1 here as well.

Lemma A.2.1 *At time t_c, the index $Index[R, A]$ is consistent with respect to the state of R at time t_b, $S_{t_b}(R)$.*

Proof: The proof of this lemma follows from the proof of Theorem A.1.1 earlier and the fact that during $[t_b, t_c]$ there is no interference between concurrent updaters and the build process. □

Lemma A.2.2 *The sequence of index updates present in $U[R, A]$ at t_c is T_{t_b,t_c} (see Definition A.0.3).*

Proof: This proof follows the same logic as that for Lemma A.1.2. □

Theorem A.2.1 *At time t_f, the index $Index[R, A]$ is consistent with respect to the state of R at t_f.*

Proof: For a moment, assume that no concurrent updater accesses the index during $[t_c, t_f]$. Under this assumption, we will prove that the above theorem is true. The routine *NSort* (step 12, Figure 3.8) retains the latest entry for only those (key, rid) pairs that occur an odd number of times in T_{t_b,t_c}. It is clear that, since $Index[R, A]$ at t_c is consistent w.r.t. $S_{t_b}(R)$ (Lemma A.2.1), the (key, rid) pairs with an even number of entries in T_{t_b,t_c} can be ignored (since inserts and deletes for a (key, rid) pair have to alternate in T_{t_b,t_c}, see Section 3.4). Also, if a (key, rid) pair has an odd number of index updates in T_{t_b,t_c}, only the latest entry determines the state of this (key, rid) pair in the final index. Recall that T_{t_b,t_c} truly reflects the relation updates U_{t_b,t_c} (by definition). Since in the absence of concurrent updates, the catchup phase merely performs actions that are equivalent to applying T_{t_b,t_c} to an index that is consistent w.r.t. to $S_{t_b}(R)$, at time t_c the index is consistent w.r.t. $S_{t_c}(R)$, the state of the relation at t_c. Since U_{t_c,t_f} is empty due to our assumption of no concurrent updates in the catchup phase, the theorem is therefore true.

Now, we must prove that the theorem is true even in the presence of concurrent updaters in the catchup phase. First, if no (key, rid) pair associated with the index updates performed during $[t_c, t_f]$ is present in T_{t_b,t_c} (the sequence of build phase updates applied to the index in the catchup phase), then the theorem is trivially true.

Now consider a (key, rid) pair e for which one or more concurrent index updates occurs in $[t_c, t_f]$, and where there is at least one entry for e in T_{t_b,t_c}. If there are an even number of entries for e in T_{t_b,t_c}, then $Index[R, A]$ contains the true state for e at t_c, and any updates during the catchup phase $[t_c, t_f]$ will not see any inconsistency. (Note that, the build process will not insert or delete any index entry during $[t_c, t_f]$ for e, since it ignores all rids with even number of entries in T_{t_b,t_c}.)

If there is an odd number of entries for e in T_{t_b,t_c}, the build process applies only the latest entry for e to the index at some time t_e in $[t_c, t_f]$ (see Figure A.1). If the first concurrent update to e after t_c occurs **after** t_e, then no inconsistency is seen by a concurrent update, either then or at any time afterward. If one or more concurrent updates to e occurs **after** t_c but **before** t_e, then the first update **will necessarily** find an inconsistent situation. This follows directly from Lemmas A.2.1 and A.2.2 and the fact that insert and delete entries for e have to alternate. Since the latest update for e in T_{t_b,t_c} is not entered in the index until t_e, the first concurrent index update for e (which

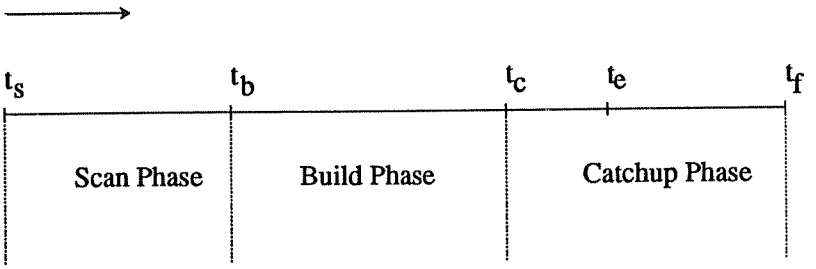


Figure A.1: Phases in the List-C-Basic algorithm

happens before t_e) will detect this inconsistency (i.e., an insert will find an existing entry and a delete will not find the required entry) and will add a special *marked entry* to the index (step n, Figure 3.9). After this first index update for e, the state in the index for e is a true reflection of its state in the relation, ignoring the marked entry for e. Subsequent updaters to e ignore the marked entry for e and will not find any inconsistency. At t_e, the build process will access the index and find the marked entry and delete it, after which no new marked entry for e can be introduced. Since the above is true for every e in T_{t_b,t_c}, $Index[R, A]$ will be consistent w.r.t to the state of the relation at t_f. □

The proof for the List-C-Basic algorithm can be extended to prove the correctness of the List-C-Sort and the List-C-Merge algorithms. The only difference is in the proof of Lemma A.2.1, which is similar to how Theorem A.1.1 was proved for the List-X-Sort and List-X-Merge algorithms. Apart from Lemma A.2.1, the rest of the proofs for List-C-Sort and List-C-Merge are the same as for List-C-Basic, as all three algorithms behave similarly in the catchup phase.

A.3 Index-Based Algorithms

We saw earlier that for all of the list-based algorithms, Lemmas A.1.1 and A.1.2 were true. For all of the index-based algorithms, only Lemma A.1.1 holds. Lemma A.1.2 doesn't hold here because, unlike the list-based algorithms, where an *Exclusive* lock is requested on the update-list before releasing the Exclusive lock on the modified relation page, here only a *Share* lock is requested on the temporary index before releasing the relation page lock. However, it turns out that proofs similar to those earlier can still be obtained by using the following weaker lemma which has to hold (by definition) for serializability of transactions (assuming that a relation update and the corresponding index update occur within the same transaction).

Lemma A.3.1 *If two relation updates u_1 and u_2 occur (Definition A.0.2) in a certain order, both involving the same rid r, then the corresponding index updates $t(u_1)$ and $t(u_2)$ occur in the same*

order. I.e., if u_1 occurs before u_2, then $t(u_1)$ occurs before $t(u_2)$ and vice versa.

Using the above lemma and Lemma A.1.1, it is possible to prove Theorem A.1.1 for the Index-X-Basic algorithm. We will outline the key ideas of the proof here. First, observe that at time t_b, the end of the scan phase (step n, Figure 3.13), the public index ($Index[R, A]$) contains exactly the same entries as would be obtained by storing all index updates during the scan phase in a list and then sorting this list using *Sort* (Figure 3.1), as in the List-X-Sort algorithm (Figure 3.6). The proof for Index-X-Basic is therefore similar to that for List-X-Sort. The key difference is that the consistency of the index is proven on a per-rid basis instead of the per-page basis used in proving Theorem A.1.1.

The proof for the Index-X-Merge algorithm (Figure 3.15) follows from the proof for the Index-X-Basic algorithm in the same way that the List-X-Merge algorithm proof followed from that for the List-X-Sort algorithm. The first part of the proof for the less restrictive Index-C-Basic algorithm involves re-proving Lemma A.2.1 using the proof of the Index-X-Basic algorithm outlined above. The rest of the proof follows from the proof of Theorem A.2.1 for the List-C-Basic algorithm by observing how the state of the temporary index at the start of the catchup phase (t_c, step 10 in Figure 3.16) is similar to the result produced by storing the updates that occurred in $[t_b, t_c]$ in a list and sorting it using *NSort* (Figure 3.8). After proving the Index-C-Basic algorithm, the extension to Index-C-Merge is straightforward, as for the corresponding list algorithms.

A.4 Coloring Algorithms

Finally, we briefly touch upon how to prove algorithms which use the coloring strategy described in Section 3.7. Here, the updaters know whether or not a particular page has been scanned by a build process, and they only communicate to the build process those changes that take place after the page has been scanned. The following lemma can be used along with Lemma A.3.1 to derive proofs for this class of algorithms in a manner similar to that described earlier in this section.

Lemma A.4.1 *Only updates to a relation page p that occur after t_p (the time when p is scanned) will be present in either the update-list or the temporary index.*

Proof: This is ensured directly by the coloring scheme outlined in Section 3.7. □

Bibliography

[**Agra89**] Agrawal, D. and Sengupta, S., "Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control", *Proceedings of the ACM SIGMOD Conference,* June 1989.

[**Agra87**] Agrawal, R., Carey, M., and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications", *ACM Transactions on Database Systems,* **12(4),** December 1987.

[**Baye72**] Bayer, R. and McCreight, E.M., "Organization and Maintainance of Large Ordered Indices", *Acta Informatica,* **1(3),** 173–189, 1972.

[**Baye77**] Bayer, R. and Schkolnick, M., "Concurrency of Operations on B-trees", *Acta Informatica,* **9,** 173–189, 1977.

[**Bern81**] Bernstein, P., and Goodman, N., "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys,* **13(2),** June 1981.

[**Bili85**] Biliris, A., "A Model for the Evaluation of Concurrency Control Algorithms on B-trees", *Computer Science Technical Report,* **No. 85-015,** Boston University, 1985.

[**Bili87**] Biliris, A., "Operation Specific Locking in B-trees", *Proceedings of the Sixth ACM Symposium on Principles of Database Systems,* San Diego, California, 159–169, March 1987.

[**Blak86a**] Blakeley, J., Larson, P. and Tompa, F., "Efficiently Updating Materialized Views", *Proceedings of the ACM SIGMOD Conference,* May 1986.

[**Blak86b**] Blakeley, J., Coburn, N. and Larson, P., "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates", *Proceedings of the Twelfth International Conference on Very Large Data Bases,* Aug. 1986.

[**Bobe92**] Bober, P. and Carey, M., "On Mixing Queries and Transactions Via Multiversion Locking", *Proceedings of the Eighth IEEE Conference on Data Engineering,* February 1992, to appear.

[**Care84a**] Carey, M., and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems", *Proceedings of the Tenth International Conference on Very Large Data Bases,* Singapore, August 1984.

[**Care84b**] Carey, M., and Thompson, C., "An Efficient Implementation of Search Trees on ⌈lg N + 1⌉ Processors", *IEEE Transactions on Computer Systems,* **11(2),** November 1984.

[**Chan82**] Chan, A. et al, "The Implementation of an Integrated Concurrency Control and Recovery Scheme", *Proceedings of the ACM SIGMOD Conference,* June 1982.

[**Chen84**] Cheng, J.M., Loosley, C.R., Shibamiya, A. and Worthington, P.S., "IBM Database 2 Performance: Design, Implementation, and Tuning", *IBM Systems Journal,,* **23(2),** 189–210, 1984.

[**Come79**] Comer, D., "The Ubiquitous B-Tree", *ACM Computing Surveys,* **11(4),** 1979.

[**Dewi90**] DeWitt, D. J. and Gray, J., "Parallel Database Systems: The Future of Database Processing or a Passing Fad?", *SIGMOD Record,* **19(4)** December 1990.

[**Elli80a**] Ellis, C., "Concurrent Search and Insertion in 2-3 Trees", *Acta Informatica,* **14(1),** 1980.

[**Elli80b**] Ellis, C., "Concurrent Search and Insertion in AVL Trees", *IEEE Transactions on Computers,* **C-29(9),** September 1980.

[**Elli83**] Ellis, C., "Extendible Hashing for Concurrent Operations and Distributed Data", *Proceedings of the 2nd ACM Symposium on Principles of Database Systems,* Atlanta, Georgia, March 1983.

[**Epst79**] Epstein, R., "Techniques For Processing of Aggregates in Relational Database Systems", *Memorandum* **No. UCB/ERL M79/8,** Electronics Research Laboratory, University of California-Berkeley, 1979.

[**Fran85**] Franaszek, P., and Robinson, J., "Limitations of Concurrency in Transaction Processing", *ACM Transactions on Database Systems,* **10(1),** 1–28, March 1985.

[**Good85**] Goodman, N., and Shasha, D., "Semantically-based Concurrency Control for Search Structures", *Proceedings of the Fourth ACM Symposium on Principles of Database Systems,* March 1985.

[**Gray79**] Gray, J., "Notes On Database Operating Systems", *in Operating Systems: An Advanced Course,* Springer-Verlag, 1979.

[**Guib78**] Guibas, L., and Sedgewick, R., "A Dichromatic Framework for Balanced Trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computer Science,* 1978.

[**Haer83**] Haerder, T., and Reuter, A., "Principles of Transaction-Oriented Database Recovery", *ACM Computing Surveys,* **15(4),** December 1983.

[**John89**] 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,* 235–246,

1989.

[John90a] 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,* April 1990.

[John90b] Johnson, T., "The Performance of Concurrent Data Structure Algorithms", *Ph.D. Thesis,* New York University, May 1990.

[Kell88] Keller, A. and Wiederhold, G., "Concurrent Use of B-trees with Variable-Length Entries", *SIGMOD Record,* **17(2),** June 1988.

[Kung80] Kung, H., and Lehman, P., "A Concurrent Database Manipulation Problem: Binary Search Trees", *ACM Transactions on Database Systems,* **5(3),** September 1980.

[Kwon82] Kwong, Y., and Wood, D., "A New Method for Concurrency in B-trees", *IEEE Transactions on Software Engineering,* **SE-8(3),** May 1982.

[Lani86] Lanin, V. and Shasha, D., "A Symmetric Concurrent B-tree Algorithm", *Proceedings of the Fall Joint Computer Conference,* 380–389, 1986.

[Lehm81] Lehman, P., and Yao, S., "Efficient Locking for Concurrent Operations on B-trees", *ACM Transactions on Database Systems,* **6(4),** December 1981.

[Livn90] Livny, M., "DeNet User's Guide", version **1.5,** 1990.

[Lome91] Lomet, D. and Salzberg, B., "Concurrency and Recovery for Index Trees", *Technical Report,* **No. CRL 91/8,** Cambridge Research Laboratory, Digital Equipment Corporation, August 1991.

[Mill78] Miller, R., and Snyder, L., "Multiple Access to B-trees", *Proceedings of the Conference on Information Science and Systems,* Johns Hopkins University, Baltimore, MD, March 1978.

[Moha89] Mohan, C. and Levine, F., "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging", *IBM Research Report,* **RJ 6846,** 1989.

[Moha90] 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,* 392–405, September 1990.

[Moha91] Mohan, C. and Narang, I., "Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates", *IBM Research Report,* **RJ 8016,** March 1991.

[Mond85] 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,* 331–334, 1985.

[**Omie89**] Omiecinski, E., "Concurrent File Conversion Between B+ Tree and Linear Hash Files", *Information Systems,* **14(5),** 1989.

[**Omie92**] Omiecinski, E., Lee, L. and Scheuermann, P., "Performance Analysis of a Concurrent File Reorganization Algorithm for Record Clustering", *Proceedings of the Eighth IEEE Conference on Data Engineering,* February 1992, to appear.

[**Pu85**] Pu, C., "On-the-Fly, Incremental, Consistent Reading of Entire Databases", *Proceedings of the Eleventh International Conference on Very Large Data Bases,* 369–375, 1985.

[**Pu88**] Pu, C., Hong, C. H. and Wha, J.M., "Performance Evaluation of Global Reading of Entire Databases", *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems,* 167–176, 1988.

[**Rose78**] Rosenkrantz, D., "Dynamic Database Dumping", *Proceedings of the ACM SIGMOD Conference,* May 1978.

[**Sagi85**] Sagiv, Y., "Concurrent Operations on B*-trees with Overtaking", *Proceedings of the Fourth ACM Symposium on Principles of Database Systems,* 28–37, 1985.

[**Salz91**] Salzberg, B. and Dimock, A., "Record Level Concurrent Reorganization", *Technical Report,* **No. NU-CCS-91-6,** College of Computer Science, Northeastern University, May 1991.

[**Sama76**] Samadi, B., "B-trees in a System With Multiple Users", *Information Processing Letters,* **5(4),** 1976.

[**Seli79**] Selinger, P., et al, "Access Path Selection in a Relational Database Management System", *Proceedings of the ACM SIGMOD Conference,* June 1979.

[**Shap86**] Shapiro, L., "Join Processing in Database Systems with Large Main Memories", *ACM Transactions on Database Systems,* **11(3),** September 1986.

[**Shas84**] Shasha, D., "Concurrent Algorithms for Search Structures", *Ph.D. Thesis,* Aiken Computation Laboratory, Harvard University, June 1984.

[**Shas85**] Shasha, D., "What Good are Concurrent Search Structure Algorithms for Databases Anyway?", *Database Engineering,* **8(2),** June 1985.

[**Silb90**] Silberschatz, A., Stonebraker, M. and Ullman, J. D., "Database Systems: Achievements and Opportunities", *SIGMOD Record,* **19(4)** December 1990.

[**Sock78**] Sockut, G. H., "A Performance Model for Computer Data-Base Reorganization Performed Concurrently with Usage", *Operations Research,* September–October 1978.

[**Sock79**] Sockut, G. and Goldberg, R., "Database Reorganization – Principles and Practice", *ACM Computing Surveys,* **11(4),** December 1979.

[**Sode81a**] Soderlund, L., "Concurrent Database Reorganization – Assessment of a Powerful Technique through Modeling", *Proceedings of the Seventh Conference on Very Large Data Bases,* September 1981.

[**Sode81b**] Soderlund, L., "Evaluation of Concurrent Physical Database Reorganization through Simulation Modeling", *Proceedings of the ACM SIGMETRICS Conference,* Sept. 1981.

[**Srin91**] Srinivasan, V. and Carey, M. J., "Performance of B-tree Concurrency Control Algorithms", *Technical Report,* **TR 999,** Computer Sciences Department, University of Wisconsin, February 1991.

[**Ston88**] Stonebraker, M., Katz, R., Patterson, D. and Ousterhout, J., "The Design of XPRS", *Proceedings of the Fourteenth Conference on Very Large Data Bases,* Los Angeles, CA, August 1988.

[**Tay84**] Tay, Y., "A Mean Value Performance Model For Locking in Databases", *Ph.D. Thesis,* Computer Science Department, Harvard University, February 1984.

[**Verh78**] Verhofstad, J., "Recovery Techniques for Database Systems", *ACM Computing Surveys,* **10(2),** June 1978.

[**Weih90**] Weihl, W. E., Wang, Paul., "Multi-Version Memory: Software Cache Management for Concurrent B-trees", *MIT Laboratory of Computer Science Manuscript,* (1990).

[**Yao78**] Yao, A. C., " On Random 2–3 Trees", *Acta Informatica,* **9,** 159–170, 1978.