ON-LINE INDEX CONSTRUCTION ALGORITHMS

by

V. Srinivasan and Michael J. Carey

Computer Sciences Technical Report #1008

March 1991

ON-LINE INDEX CONSTRUCTION ALGORITHMS

V. Srinivasan
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706 USA

This research was partially supported by the National Science Foundation under grant IRI-8657323 and by a University of Wisconsin Vilas Fellowship.

On-Line Index Construction Algorithms

V. Srinivasan *Michael J. Carey*
Department of Computer Sciences
University of Wisconsin
Madison, WI 53706
srinivas@cs.wisc.edu

Abstract

In this paper, we present several 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. We provide proofs that the algorithms presented here indeed create a consistent index in the presence of concurrent updates.

1 Introduction

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]. This 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 re-organization of data and for construction and maintenance of hidden data structures like indices.

Indices are one of the most important hidden data structures that are created and maintained in a DBMS. A need can arise for the construction of a new secondary index on an attribute of a relation if it is discovered that a significant proportion of the queries on the relation could be executed more efficiently using an index on this attribute. Currently, database systems 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

This research was partially supported by the National Science Foundation under grant IRI-8657323 and by a University of Wisconsin Vilas Fellowship.

problem [Dewi90, Silb90].

In this paper, 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. Since B-trees[1] 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 are also likely to be useful in designing on-line algorithms for other types of index structures.

The rest of the paper is organized as follows. Section 2 briefly reviews the B-tree index structure and describes the B-link concurrency control algorithm [Lehm81] for supporting concurrent B-tree operations. Earlier studies [John90, Srin91] have shown that the B-link algorithm provides the most concurrency among B-tree concurrency control algorithms proposed in the literature (see [Srin91] for details). In Section 3, we describe the basic steps involved in building a B-tree index. We describe a simple off-line index construction algorithm in Section 4. After describing index updates in Section 5, we discuss various on-line index construction algorithms in Sections 6, 7 and 8. Section 9 concludes the paper and discusses our plans for future work. Correctness proofs for our on-line index construction algorithms are outlined in the Appendix.

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.

2.1 B-tree Review and Terminology

The following description assumes fixed length keys, but the extension to variable length keys is straightforward. A B-tree index is a page-oriented search structure that has the following properties. First, it is a balanced *leaf* search tree — actual keys are present only in the leaf pages, and all paths from the root to a leaf are of the same length. A B-tree is said to be of order d if every node has at most $2d$ separators[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 (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, and 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

[1] By B-tree we mean the variant in which all keys are stored at the leaves, often called B^+-trees [Come79].

[2] A *key* is usually meant to imply that associated information for that value exists in the index. A *separator* value defines a search path to leaf pages that contain the actual keys and associated information.

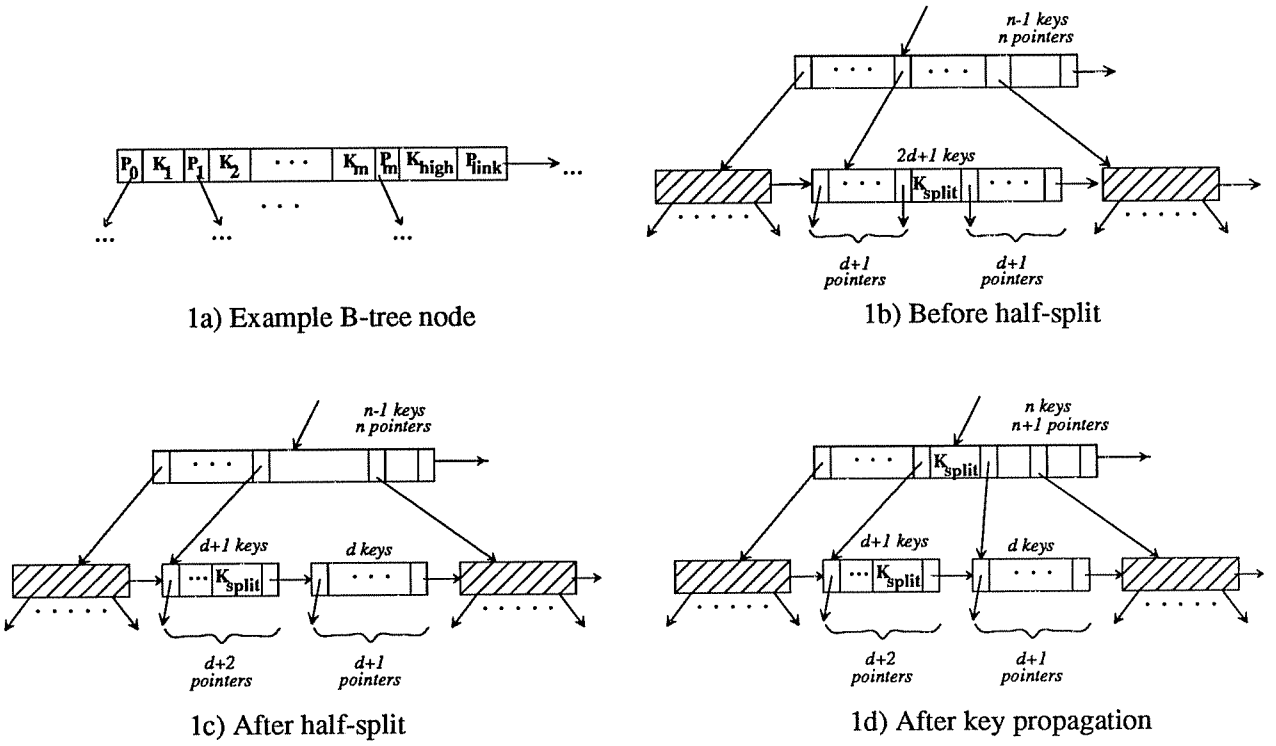

Figure 1: Example B-link tree page and a page split.

and the lowest level consisting of the set of leaf pages. All nodes at a level are linked from left to right; that is, we assume the B-link tree structure described in [Lehm81]. Each page in a B-link tree contains a high key (the highest key in the subtree rooted at the page) and a link to the right sibling of the page. A B-link tree node is illustrated in Figure 1a.

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 only d entries, a page split or page merge occurs. The right link enables a page split to occur in two phases [Lehm81]: a half-split followed by the insertion of an index entry into the appropriate parent. After a half-split, and before the <key, pointer> 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. Merges can also be done in two steps [Lani86], via a half-merge followed by entry deletion at the next higher level. A B-link tree page split is illustrated in Figure 1. B-trees in real database systems usually perform merges only when pages becomes empty; nodes are not required to contain at least d entries, since this does not decrease occupancy by much in practical workloads [John89]. We therefore assume this approach to B-tree merges for this study.

2.2 Supporting Concurrent Tree Operations

Concurrent operations on a B-tree can be supported using the following variation of the B-link algorithm. This variation was shown to perform very well in a recent comparative performance study [Srin91].

A reader descends the tree from the root to a leaf using Share locks. At each page, the next page to be searched can either be a child or the right sibling of the current page. (Following a right link is termed a *link-chase*.) 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. Upon reaching the appropriate leaf, an updater releases its Share lock on the leaf and requests an Exclusive lock on the same leaf. Once the Exclusive lock on the leaf is granted, the updater may either find that the leaf is the correct one to update or that one or more link-chases must be performed to reach the correct leaf. Updaters use Exclusive locks while performing all further link chases, releasing the Exclusive lock on each current page before asking for the next one. If a page split or merge is necessary, an updater will perform a half-split or half-merge; the Exclusive lock on a newly split or newly merged node is then converted to a Share lock, and this lock is held until an Exclusive lock has been acquired on the appropriate parent node, i.e., the Share lock on a node that has been half-split or half-merged is released by an updater only **after** the Exclusive lock has been acquired on its current parent.

3 Index Construction 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 2. 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 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 that for an in-memory sort), and it works as long as $\sqrt{N}$ pages of memory are available for use by the sort [Shap86]. If entries with the same (key, rid) value occur many times in the input, only the last entry in the input file for this (key,rid) value is retained in the output. (Since the file to be sorted is actually a sequence of updates, only the last one determines the state at the end and has to be retained). This is done by duplicate elimination during step 3 of the figure. To achieve this, the *Sort* routine

```
/* Index Construction Primitives */

    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 sorted list of pages sorted by (key,rid)
            step 1:  Scan H in batches of √N pages and produce sorted output runs
            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:  Create the higher levels of the index I based on L
            step 3:  return I
        end
```

Figure 2: Index Construction Primitives

can tag all entries in the input file with a number that indicates their relative order in the input file, and use these tags later while eliminating duplicates.

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[3]. 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

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

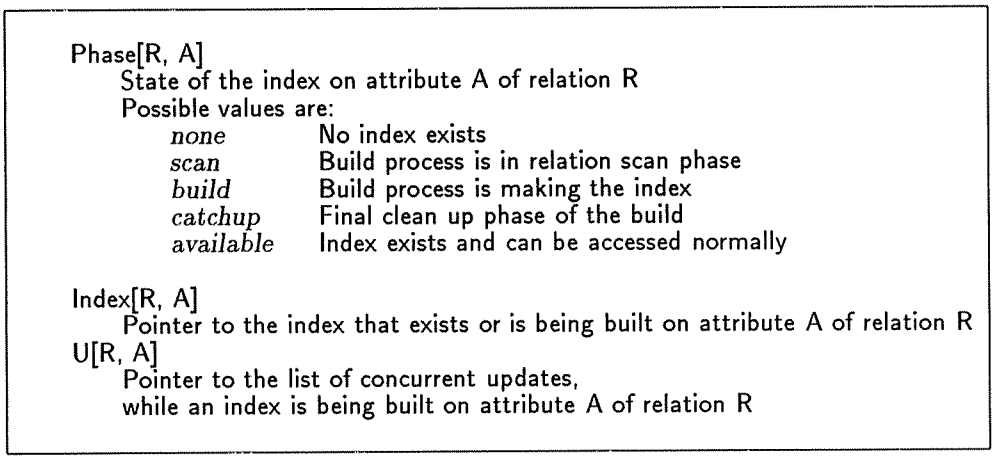

Figure 3: Shared data structures for Index Construction

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). 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.

4 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 4. The code is straightforward and builds 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 2). We assume that such an optimization 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

```
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 4: Off-Line Algorithm

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. We know of only one commercial DBMS (from Synapse) that used an on-line algorithm for index construction [Wood91]. The full details of this algorithm have not been published; we only know that it used a strategy similar to an incremental dump [Pu85] to collect the index entries from the relation, after which it built an index using these entries and then applied the concurrent updates recorded in the system log to the new index, to make it consistent.

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 very 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 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.

5 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 paper 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 algorithms themselves.

5.1 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. However, all of our algorithms will 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[4], though we will also argue that this may not be such a bad idea.

5.2 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), undo operations will generate index updates in exactly the same manner as 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 obtain a different record id, in which case the abort-time index update will be an insert with the same key but a different record id [Wood91].

6 List-Based Algorithms

The list-based algorithms, like all on-line algorithms described in this paper, 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 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 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 then additionally sort the update-list before

[4]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.

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[5]. 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).

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 2) 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.[6] This effect will be significant when the number of sequential inserts is large.

Name	How Updates Are Applied?	After Scan Phase
List-X-Basic	Sequentially apply from unsorted list	X lock list, no concurrent updaters
List-X-Sort	Sequentially apply from *sorted* list	X lock list, no concurrent updaters
List-X-Merge	*Merge* sorted list and scanned entries	X lock list, no concurrent updaters
List-C-Basic	Sequentially apply from unsorted list	Concurrent updaters allowed
List-C-Sort	Sequentially apply from *sorted* list	Concurrent updaters allowed
List-C-Merge	*Merge* sorted list and scanned entries	Concurrent updaters allowed

Table 1: List-Based Algorithms.

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 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 these 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 1. We will now describe each of these list-based algorithms in more detail.

[5]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.

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

```
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
                         end
        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
                     begin
                         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
    end
```

Figure 5: The List-X-Basic and List-X-Sort Algorithms

6.1 The List-X-Basic Algorithm

In this algorithm (described in Figure 5), 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 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 5) 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 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 5. 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.[7] During the time while the index construction process is in the build phase (steps 6 and 7 in Figure 5), 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.

6.2 The List-X-Sort Algorithm

The List-X-Sort algorithm is described in Figure 5 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 5). 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 update-list, and the last entry 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 2)

[7] In Figure 5, 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 5 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.

```
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 6: The List-X-Merge Algorithm

therefore, retains only the latest index operation for a (key, rid) pair that occurs many times 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.

6.3 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 6. The code for the updaters is the same as that for the earlier algorithms (see Figure 5). 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 5), has now been moved to the build phase (step 7 of Figure 6). 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 2) now performs all of the actions that were done by 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 5 and steps 4-9 in Figure 6). 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, respectively, that allow concurrent updates both during and after the scan phase.

6.4 The List-C-Basic Algorithm

In the List-C-Basic algorithm (described in Figure 7), 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. Therefore, after step 8 of Figure 7 has been completed, the intermediate index (Index[R,A]) contains an index that would be up to date with respect to all of the concurrent updates that occurred during the scan phase. However, since updaters are allowed to execute during the build phase also, there will probably be additional entries to apply to the index at the end of the build phase. All that remains to be done is to incorporate, into the intermediate index, concurrent updates that took place since the beginning of the build phase. This is done in the catchup phase.

Before we describe the detailed behavior of the build process in the catchup phase, note that the behavior of the updaters (given in Figure 8) 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 5). Also note that, at the end of the build phase in Figure 7, the intermediate index (Index[R,A]) is consistent[8] 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 7), 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 7). This is because the inconsistencies introduced during scanning are removed due to the build phase 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.

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 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 7) 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 7). 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.

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

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 7), 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 7) is slightly different

```
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-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
                     end
        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
                     end
        step 14: Phase[R, A] = available
    end
```

Figure 7: Build Process in the List-C-Basic and List-C-Sort Algorithms

from the earlier *Sort* routine (Figure 2), in that it eliminates duplicates differently. If an even number of occurrences of a (key, rid) entry is 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)is found in the input file, however, the latest entry is kept in the output file. Recall that the earlier procedure *Sort* **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 7). Note that the build process code for applying these build phase updates (step 13 of Figure 7) differs from the code for applying scan

phase updates (step 8).

```
Update transaction
    begin
        R: input relation
                Normal processing
                    .
                    .
                    .
        step n:  if Phase[R, A] = scan or build then
                    begin
                        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]
                    end
                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
    end
```

Figure 8: Updater in the List-C-Basic, List-C-Sort and List-C-Merge Algorithms

The actions performed by updaters during the catchup phase are given in Figure 8. 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 of 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) is required. This is accomplished via the use of special marked index entries, which 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 7) and 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 7). We prove the above assertions in the appendix.

6.5 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 7). 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 updaters here is identical to that in the List-C-Basic algorithm (Figure 8).

```
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
                     end
        step 14: Phase[R, A] = available
    end
```

Figure 9: The List-C-Merge Algorithm

6.6 The List-C-Merge Algorithm

The last list-based algorithm that we discuss is the List-C-Merge algorithm (Figure 9). 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. Just like in List-C-Basic and List-C-Sort, concurrent updates are allowed in the build phase in List-C-Merge also, and these updates are then applied to the intermediate index during the catchup phase. Notice that the code for the catchup phase in this algorithm (steps 12-14 in Figure 9) is the same as that in the List-C-Basic and List-C-Sort algorithms (steps 12-14, Figure 7). The code for the updaters is also the same here (Figure 8).

6.7 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 (as was done in a commercial DBMS from Synapse [Wood91]). The advantage of using the log would be that no changes are needed to the logic for concurrent updates, 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 on-line. Moreover, only a small fraction of the log will be relevant to a particular index construction process, and this fraction will be distributed throughout the overall log. These disadvantages led us to use a special update-list in the algorithms presented in this section.

7 Index-Based Algorithms

Index-based algorithms use an index to store concurrent updates instead of the update-list used by the list-based algorithms. All of the index-based algorithms, like all of our on-line algorithms, allow updaters to execute concurrently during the scan phase. Like the list-based algorithms, the different 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 public index (created by concurrent updates) already contain the keys in sorted order. The various possible index-based algorithms are listed in Table 2.

Name	How Updates Are Applied?	After Scan Phase
Index-X-Basic	Sequentially apply from leaf-page list	X lock index, no concurrent updaters
Index-X-Merge	*Merge* leaf-page list with scanned entries	X lock index, no concurrent updaters
Index-C-Basic	Sequentially apply from leaf-page list	Concurrent updaters allowed
Index-C-Merge	*Merge* leaf-page list with scanned entries	Concurrent updaters allowed

Table 2: Index-Based Algorithms.

7.1 The Index-X-Basic Algorithm

The pseudo-code for the Index-X-Basic algorithm is given in Figure 10. The build process executes in two phases, the scan and build phases, and is very much like the build process in the List-X-Basic algorithm (Figure 5). The difference here is the use of a "public" B-tree index (visible to concurrent updaters, Index[R,A] in Figure 10) to store the concurrent updates. As before, the build process scans the relation to gather index entries and builds a private intermediate index (T in Figure 10) 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 10). 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 step is not used for a normal index traversal. This share lock is used by updaters simply to exclude the build process from the index; concurrent updaters can still concurrently update this 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 normally available index. In Figure 10, 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, an rid, and the operation type (*insert/delete*). (Entries in a normal index do not have the third, operation type, component.)

- Updates to the public index differ from normal index updates in that they leave special 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[9] from occurring when the scanned entries and the concurrent updates are combined. Any inconsistencies that are present in the private intermediate index built using the scanned entries alone will be removed by the index building process during the build phase (step 7 of Figure 10).

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 2) 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; since 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.

7.2 The Index-X-Merge Algorithm

The Index-X-Merge algorithm is illustrated in Figure 11. The build process again executes in two phases, and is very much like the List-X-Merge build process (Figure 6). The only differences are

[9] 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. Thus, 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 scanned list) when the 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
                         end
        step 8:      Phase[R, A] = available
        step 9:  Unlock Index[R,A]
    end

Update transaction
    begin
        R: Input relation
                 Normal processing
                     .
                     .
                     .
        step n:  if Phase[R, A] = scan or build then
                     begin
                         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
                             end
                         Unlock Index[R, A]
                     end
    end
```

Figure 10: The Index-X-Basic algorithm

```
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 11: The Index-X-Merge Algorithm

(i) the use of a public B-tree index in lieu of the update-list used earlier, and (ii) the absence of the sort phase (step 6 of Figure 6) 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 11). The code for update transactions is exactly the same as that for the Index-X-Basic algorithm (see Figure 10).

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

7.3 The Index-C-Basic Algorithm

The build process of the Index-C-Basic Algorithm is described in Figure 12, and the corresponding update transaction code is given in Figure 13. The build process of this algorithm is similar to the build process in the List-C-Basic algorithm (see Figure 7). 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 10 and 13.

After building the intermediate index (after step 4, Figure 12), the build process locks the public index in Exclusive mode, driving away any updaters, and copies 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.

```
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
                     end
        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
                     end
        step 13: Phase[R, A] = available
    end
```

Figure 12: Build Process in the Index-C-Basic Algorithm

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 code segments for the scan and build phases in Figure 13. 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 8). These concurrent updaters

```
Update transaction
    begin
        R = input relation
                Normal processing
                .
                .
                .
        step n:  if Phase[R, A] = scan then
                    begin
                        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
                            end
                        Unlock Index[R, A]
                    end
                else if Phase[R, A] = build then
                    begin
                        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
                            end
                        Unlock Index[R, A]
                    end
                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
    end
```

Figure 13: Update Transaction 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
                     end
        step 12: Phase[R, A] = available
    end
```

Figure 14: The Index-C-Merge Algorithm

leave specially marked entries in the index in case they detect inconsistencies (e.g., an inserter finds the entry to be inserted already in the index, or a deleter does not find an entry to delete). These marked entries will be found and removed by the build process as it executes step 12 of Figure 12. 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.

7.4 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 14) 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 14) 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 13).

8 More Concurrency for Updaters

In Section 5, we stated that concurrent updaters register their index updates immediately into the updates-list or the temporary index during index construction. This ensures that all updates to a page that occur after index construction starts but by the time when this particular page is scanned by the build process are registered in the updates-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[10]. 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 a 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" in 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 Summary and Future Work

In this paper, 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.

[10] An example 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.

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 the proposed 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 of 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 cost without the index). This suggests that a new cost model needs to be developed to evaluate the performance tradeoffs of these algorithms. In addition, as 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. Appropriate recovery strategies have to be designed for this purpose. Finally, since a terabyte relation is likely to be declustered across several disks, it is necessary to parallelize the various on-line algorithms for use in such parallel database systems. These issues are all good candidates for future work.[11]

References

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

[Come79] Comer, D. "The Ubiquitous B-Tree", *ACM Computing Surveys,* **11(4)** 412 (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.

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

[John89] Johnson, T. and Shasha, D. "Utilization of B-trees with Inserts, Deletes and Searches", *ACM Symposium on Principles of Database Systems,* 235–246, 1989.

[John90] Johnson, T. and Shasha, D. "A Framework for the Performance Analysis of Concurrent B-Tree Algorithms", *Proceedings of the 9th Symposium on Principles of Database Systems,* (1990).

[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.

[11] We recently learned of a newly-available technical report, which appeared while this paper was being finished up, that also proposes solutions to the problem of on-line index construction [Moha91]. A comparison of these two independently-developed approaches is another important topic for future work.

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

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

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

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

[**Srin91**] Srinivasan, V. and Carey, M. J. "Performance of B-tree Concurrency Control Algorithms", *To Appear in Proceedings of the SIGMOD Conference,* May 1991.

[**Wood91**] Wood, D. "Personal Communication",, February, 1991.

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

A Correctness Proofs

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 also. 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.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.1, it follows that modifying a field of a record is considered 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 5 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.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.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.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 Proof for the List-X-Basic Algorithm

Let t_s denote the time at the start of the scan phase, step 1 of Figure 5. Let t_b denote the start of the build phase, i.e., the Exclusive lock on $U[R, A]$ (step 5 of Figure 5) is granted at time t_b. Finally let t_f be the time when the index construction terminates (step 9 of Figure 5). 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 5). 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 updates list $U[R, A]$ contains a sequence of index updates ordered by this lock release time.

Definition A.5 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 *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.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.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 8).

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 5 (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 *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 5) of the build process. For page p, the sequence of index updates in B_p are first redone on the 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. In this case, applying B_p to $Index[R.A]$ does not change the entry corresponding to r in $Index[R, A]$. Consider a record id q that occurs 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 and A.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., only valid entries, one for each record in p, will exist in $Index[R, A]$. Applying the index updates in A_p 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.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 for the List-X-Sort algorithm as follows. Lemmas A.1 and A.2 are true here also. To prove the theorem, one has to show that the sort (step 6a in Figure 5) 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 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 6), is identical to creating an index out of the sorted scanned entries first and then sequentially applying the entries from the sorted updates list to this index (as in List-X-Sort, step 7 in Figure 5).

A.2 The 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 7). 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 7). These times are illustrated in Figure 15. We further assume the correctness of the B-tree concurrency control algorithm used for performing concurrent operations on the index. Lemmas A.1 and A.2 are true here as before. Our goal now is to prove Theorem A.1 here as well.

Lemma A.3 *At time t_c, the index $Index[R, A]$ is consistent with respect to the state of R at t_b, $S_{t_b}(R)$.*

Proof: The proof of this lemma follows from the proof of Theorem A.1 earlier and the fact that during $[t_b, t_c]$ there is no interference between concurrent updaters and the build process. □

Lemma A.4 *The sequence of index updates present in $U[R, A]$ at t_c is T_{t_b,t_c} (see Definition A.3).*

Proof: This proof follows the same logic as that for Lemma A.2. □

Theorem A.2 *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 7) 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.3), 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 5). 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 index updates performed during $[t_c, t_f]$ is present in T_{t_b,t_c} (the 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

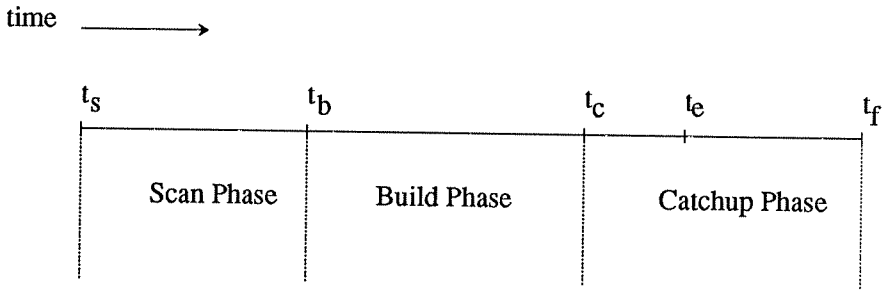

Figure 15: Phases in the List-C-Basic algorithm

phase $[t_c, t_f]$ will not see any consistency. Also, 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 15). 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.3 and A.4 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 happens before t_e) will detect this inconsistency (i.e., an insert will find an entry and a delete will not find the required entry) and add a special *marked entry* to the index (step n, Figure 8). 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 consistency. At t_e, the build process will access the index and find the marked entry and delete it, after which no marked entry can be introduced. Since the above is true for every e in T_{t_b,t_c}, at t_f, $Index[R, A]$ 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.3, which is similar to how Theorem A.1 was proved for the List-X-Sort and List-X-Merge algorithms. Apart from Lemma A.3, 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 Proofs for Index-Based Algorithms

We saw earlier that for all of the list-based algorithms, Lemmas A.1 and A.2 were true. For all of the index-based algorithms, only Lemma A.1 holds. Lemma A.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.5 *If relation updates u_1 and u_2 occur (Definition A.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, it is possible to prove Theorem A.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 10), 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 2), as in the List-X-Sort algorithm (Figure 5). 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.

The proof for the Index-X-Merge algorithm (Figure 11) follows from the proof for the Index-X-Basic algorithm in the same way that the 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.3 using the proof of the Index-X-Basic algorithm outlined above. The rest of the proof follows from the proof of Theorem A.2 for the 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 12) 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 7). After proving the Index-C-Basic algorithm, the extension to Index-C-Merge is straightforward, as for the corresponding list algorithms.

A.4 Proofs for Coloring Algorithms

Finally, we briefly touch upon how to prove algorithms which use the coloring strategy described in Section 8. 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.5 to derive proofs for this class of algorithms in a manner similar to that described earlier in this section.

Lemma A.6 *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 8. □