Performance of On-Line Index Construction Algorithms

by

V. Srinivasan
Michael J. Carey

Computer Sciences Technical Report #1047
September 1991

PERFORMANCE OF ON-LINE INDEX CONSTRUCTION ALGORITHMS

V. Srinivasan
Michael J. Carey

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706
USA

To appear in the
Proceedings of the International Conference on Extending Database Technology
Vienna, Austria, March 1992

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

Performance of On-Line Index Construction Algorithms

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

Abstract

In this paper, we study the performance of several on-line index construction algorithms that have recently been proposed. These algorithms each permit an index to be built while the corresponding data is concurrently accessed and updated. We use a detailed simulation model of a centralized DBMS to quantify the performance impact of various factors, including the amount of update activity, resource contention, background load, and the size of a record compared to the size of an index entry. The performance comparison makes use of two new metrics, one of which is a "loss" metric that reflects the amount of on-line work lost due to interference with the index construction activity. In our analysis, we find that there is an important trade-off between the time required to build the index and the throughput achieved by update transactions during the index construction period. An important conclusion of our study is that certain on-line algorithms perform very well in all but extremely resource-bound situations.

1 Introduction

Future databases are expected to be several orders of magnitude larger than the largest databases in operation today. In particular, databases on the order of terabytes (10^{12} bytes) are soon expected to be in active use [Silb90]. In such databases, the utilities for index construction, database reorganization, and checkpointing will take enormous amounts of time to run due to the time it takes to scan the data itself (since scanning a 1-terabyte table may take days). Thus, there is a need for these utilities to operate in an on-line fashion [Dewi90]. In terms of related work, algorithms for on-line checkpointing of a global database state have been discussed [Pu85] and their performance has been studied [Pu88]. Also, the problem of on-line index reorganization has been discussed briefly in [Ston88]. Only recently, however, have algorithms been proposed to tackle the problem

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

of on-line index construction [Srin91b, Moha91]. Since B-trees[1] are the most common dynamic index structure in database systems, the existing work has concentrated on algorithms for on-line construction of B-tree indices.

On-line index construction algorithms typically work as follows: A build process scans the data, copying out information for index entries while updaters concurrently modify the same data. The system keeps track of the updates that take place during the scan; the builder then combines these updates with the index entries created during the scan before registering the index in the system catalogs. The proposed on-line index construction algorithms differ in the data structures used for recording the concurrent updates, their strategies for combining these updates with the newly created entries, and finally, in the degree of concurrency supported following the scan phase.

A comprehensive set of on-line index construction algorithms were described in an earlier paper [Srin91b]. In this paper, we evaluate the relative performance of these on-line index construction algorithms. Using a detailed simulation model of a centralized DBMS, we compare the performance of these on-line index construction algorithms with that of a good off-line algorithm as well as amongst themselves. By running experiments over a wide range of system, workload, and storage conditions, we investigate the performance trade-offs for the proposed algorithms. In particular, to assist in our analysis, we define a new performance metric, "loss," that captures the lost work in terms of update transactions that are unable to execute due to contention caused by conflicts with the index construction process. We also compare algorithms using other relevant metrics, including the "off-line fraction," which characterizes the fraction of time (relative to the response time of an off-line algorithm) during which updaters are unable to proceed.

The rest of the paper is organized as follows. In Section 2, we summarize the proposed on-line index construction algorithms. Section 3 discusses the performance trade-offs involved in choosing one on-line algorithm over another. The simulation model used in our study is described in Section 4. Section 5 describes the performance experiments that we conducted and presents their results. In Section 6 we predict the performance of other proposed algorithms based on our performance results. Finally, in Section 7, we present our conclusions and plans for future work.

2 Index Construction Algorithms

Constructing a B-tree index from a relation usually 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 for the index. The third and final step involves creating the non-leaf pages of the index in a bottom-up fashion from the leaf page list created in the previous step.

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

2.1 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 using the three basic steps outlined above, 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 only acquire an Intention-share lock on the relation, and can access the relation's pages concurrently with the building process. Due to the absence of concurrent updaters, this *off-line* strategy is the fastest way to build the index. While the unavailability of the relation for updaters makes the off-line algorithm unacceptable for building indices on large relations, we will use its performance as a baseline for understanding the behavior of alternative on-line algorithms. One way to improve on the off-line approach is to allow updaters to proceed during index construction, somehow communicating their updates to the building process. It is possible that the duration of index construction will be increased due to the presence of concurrent updates, but permitting updates during the index construction phase makes such on-line strategies attractive. Before we describe the on-line algorithms, however, we must first describe the behavior of update transactions.

2.2 Concurrent Updates

Any update to a record in a relation that changes the value of an indexed attribute results in updates to the corresponding index. Update transactions are assumed to obey the following two rules: (i) they hold a short-term 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 long-term Exclusive locks on updated record ids until they terminate (commit or abort). Update transactions that encounter an active index construction process will record the necessary index updates in a manner that depends on the type of on-line algorithm being used. We assume for simplicity that this will occur immediately after the corresponding relation update, i.e., the new index entry will be recorded while the updater still holds its Exclusive lock on the modified relation page[2]. It should be mentioned that this approach to transaction execution handles situations like transaction aborts in a straightforward manner (see [Srin91b] for details).

Given that concurrent update transactions behave as described above, we now describe the on-line index construction algorithms of interest here. We will subdivide the on-line algorithms into two classes, list-based algorithms and index-based algorithms, depending on whether they use a list or an index for storing concurrent updates. More details about the on-line algorithms discussed here can be found in an earlier paper [Srin91b].

[2] A way to relax this restriction is described in [Srin91b].

2.3 List-Based Algorithms

The list-based algorithms for on-line index construction use a list, called the *update-list*, to record concurrent updates. The individual list-based algorithms differ from each other in their method of combining the update-list with the list of index entries obtained by scanning the relation; they also differ in the amount of concurrency provided after the initial scan phase. The various possible list-based algorithms are given in Table 1.

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. We will call this the *basic* strategy for building the index. Note that the basic strategy may result in several I/Os for any given leaf node of the intermediate index, especially if the number of entries in the update-list is large. A second method of combining the scanned entries with the update-list is to build an intermediate index using the scanned entries, as above, but to sort the update-list before applying its entries to the intermediate index. This method, called the *sort* strategy, ensures that a maximum of one disk I/O is incurred for each leaf page of the intermediate index (since leaf nodes are accessed in sorted order of the keys that they contain). A third method of combining the concurrent updates with the scanned entries is to first sort both the scanned entries and the update-list entries, producing two sorted lists; the index is then built in a bottom-up manner by merging these two sorted lists together, thus producing the leaf pages of the final index in one pass (see [Srin91b] for details). In situations where the update-list is large, this *merge* strategy has some advantages over the sort strategy, as we shall see later.

As mentioned earlier and indicated in Table 1, the list-based algorithms also vary in the amount of concurrency allowed **after** 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. List-based algorithms that use this strategy are called the List-X-* algorithms because they use a list for storing concurrent updates and the build process acquires an X lock on the update-list at the end of the scan phase to lock-out updaters.

2.3.1 The List-X-* Algorithms

Based on the three possible methods for combining concurrent updates with the scanned index entries, there are three possible List-X algorithms, List-X-Basic, List-X-Sort, and List-X-Merge. The execution of the build process and update transactions in the various List-X-* algorithms is illustrated in Figure 1. The exclusive phase in which the update-list entries and the scanned entries are combined is called the *build* phase.

In the List-X-* algorithms, 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 (Figure 1).

Updater appends are synchronized via short-term exclusive locking of the update-list. During the build phase, however, updaters cannot execute concurrently with the build process. If the duration of the build phase is significant compared to the duration of the scan phase, there could be a noticeable loss of updater concurrency. To avoid such a loss, we can design list-based algorithms in which updaters can execute concurrently throughout the index construction period; these are called the List-C-* algorithms.

Name	How Updates Are Applied	Concurrency 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.

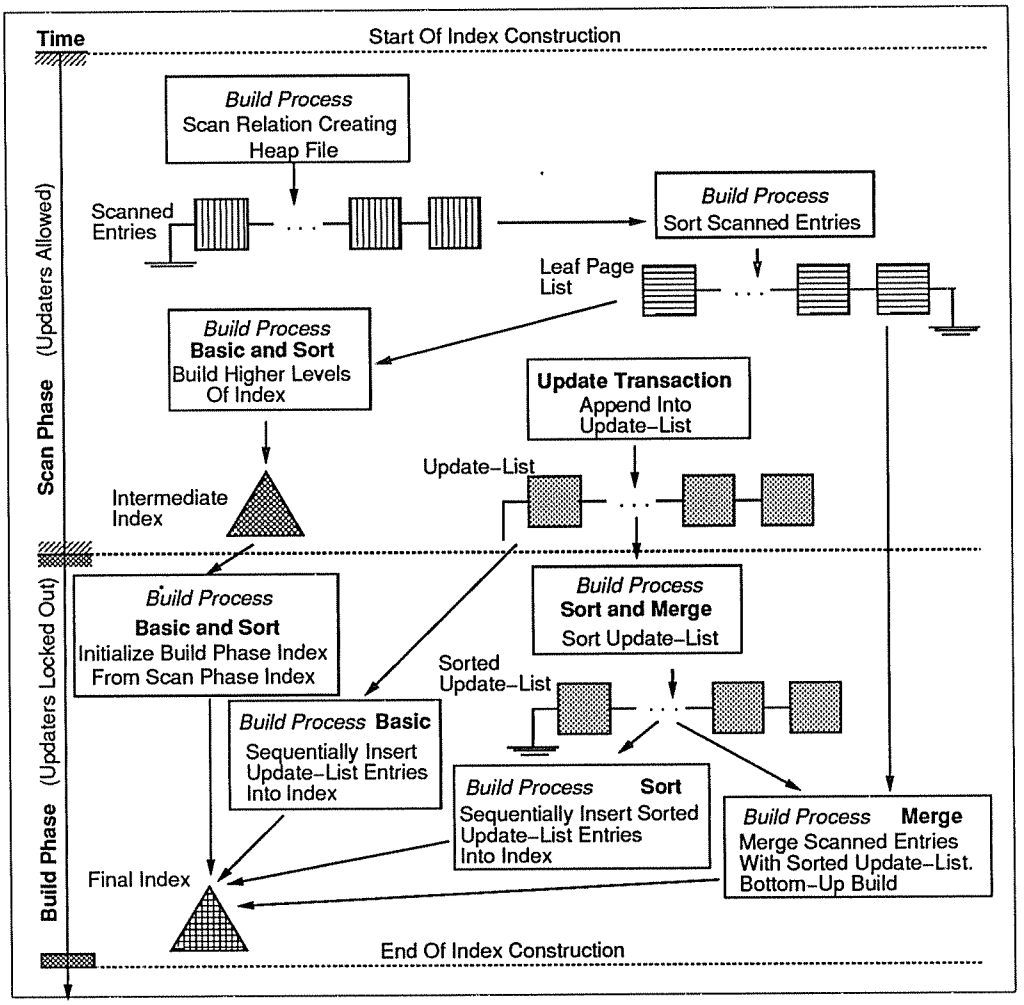

Figure 1: The List-X-* Algorithms

2.3.2 The List-C-* Algorithms

As for the List-X-* algorithms, there are three possible List-C-* algorithms: List-C-Basic, List-C-Sort, and List-C-Merge. The List-C-* algorithms are illustrated in Figure 2. Notice that the behavior of the build process in the scan and build phases of a particular List-C-* algorithm is the same as that of the corresponding List-X-* algorithm. Since the List-C-* algorithms have a build phase in which updaters can execute concurrently, however, they need an appropriate strategy for merging in a second set of concurrent updates at the end. This is done in a third phase called the *catchup* phase. Each of the three List-C-* algorithms uses the same strategy in the catchup phase to combine the build phase updates with the intermediate index that exists at the end of the build phase. In particular, the intermediate index is made public to updaters (even though it is not yet available for normal use) at the start of the catchup phase. The build process then sorts the list of build phase updates and applies them to the intermediate index concurrently with other updaters. A normal B-tree concurrency control algorithm is used to resolve conflicting accesses to the index by the build process and update transactions during the catchup phase [Srin91b]. After the build process completes processing all of the build phase updates, it makes the index available for normal use.

Updaters in the List-C-* algorithms behave like updaters in the List-X-* algorithms in the scan and build phases; i.e., they append their updates to the update-list. (The update-list is initialized to empty at the start of both the scan and the build phases.) During the catchup phase, however, the concurrent updaters directly update the intermediate index to which the build process is applying build phase updates. To ensure that the final index does not contain any inconsistencies, specially marked entries may need to be entered into this index by updaters; such marked entries are later removed by the build process (see [Srin91b] for details).

2.4 Index-Based Algorithms

The second class of on-line index construction algorithms is the class of index-based algorithms. In these algorithms, updaters use an index to store concurrent updates instead of the update-list used by the list-based algorithms. There are four possible index-based algorithms, as indicated in Table 2; since the leaf pages of this index (which are created by concurrent updates) contain the keys in sorted order, the sort method of building the index is inapplicable here and thus there are no index-based counterparts to the List-*-Sort algorithms.

2.4.1 The Index-X-* Algorithms

The Index-X-Basic and Index-X-Merge algorithms are illustrated in Figure 3. In the scan phase, updaters in both of the Index-X-* algorithms register their updates in a temporary public B-tree index. Concurrent updater access to this index is regulated using a B-tree concurrency control algorithm. A difference that arises from using an index, as opposed to an update-list, is that updaters need to leave behind special index entries in certain situations; this avoids any inconsistencies that

might otherwise be caused due to repeated inserts and deletes of the same index entry (see [Srin91b] for details). The build process in each Index-X-* algorithm is similar to that of the corresponding List-X-* algorithm; the key difference is that the (sorted) list of concurrent updates is obtained at the end of the scan phase from the leaf pages of the temporary public index rather than from an update-list.

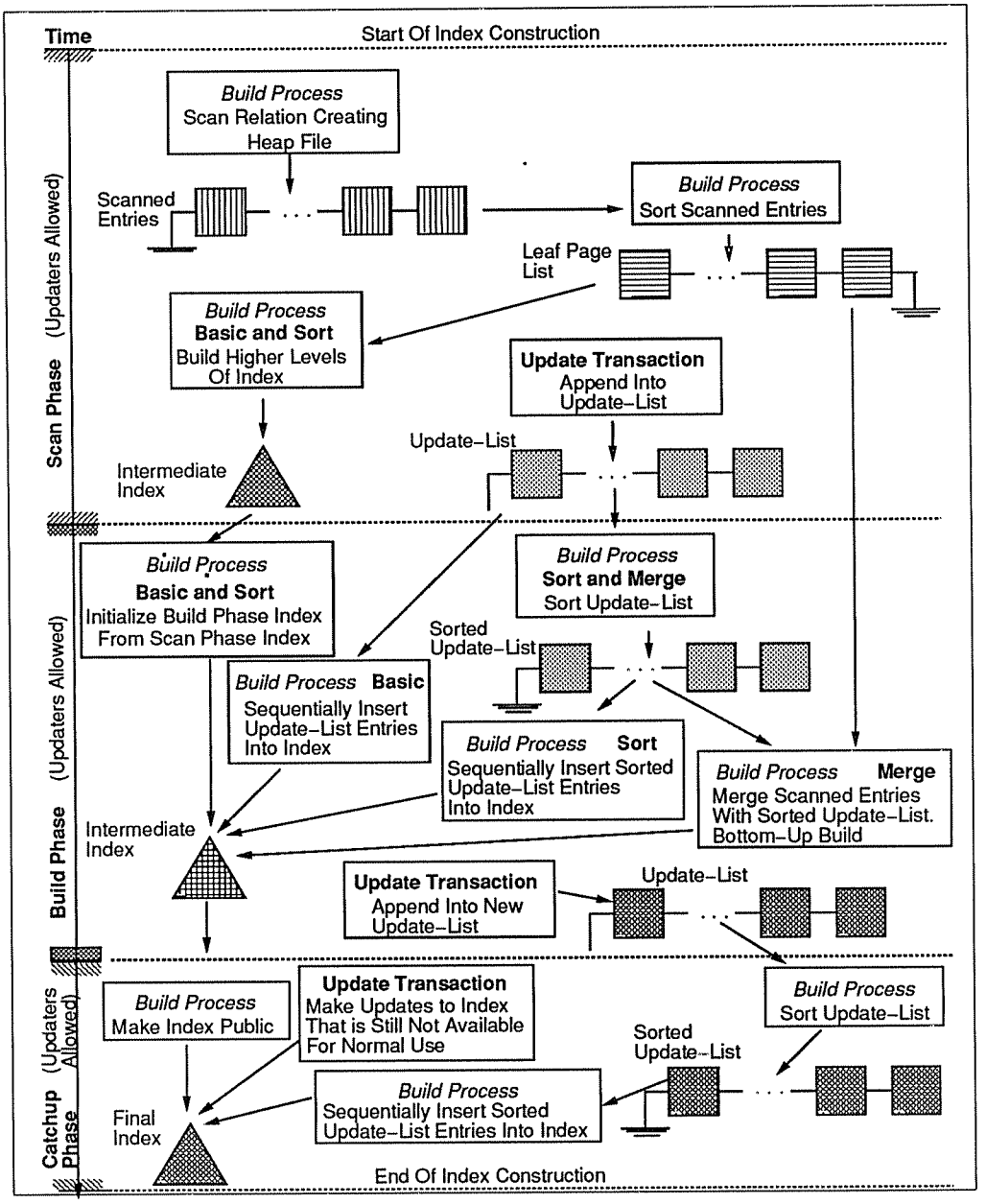

Figure 2: The List-C-* Algorithms

2.4.2 The Index-C-* Algorithms

The Index-C-Basic and Index-C-Merge algorithms are illustrated in Figure 4. In the scan and build phases, updaters in the Index-C-* algorithms register their updates to a temporary public index. This index, just like the update-list in the earlier List-C-* algorithms, is initialized to empty at the

start of the scan and build phases. Updaters in the Index-C-* algorithms behave slightly differently (in terms of leaving marked entries in the public index) during the build phase than they do in the scan phase [Srin91b]. In the catchup phase, where updaters share an index with the build process, updaters in the Index-C-* algorithms behave just like updaters in the catchup phase of the List-C-* algorithms.

Name	How Updates Are Applied	Concurrency 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.

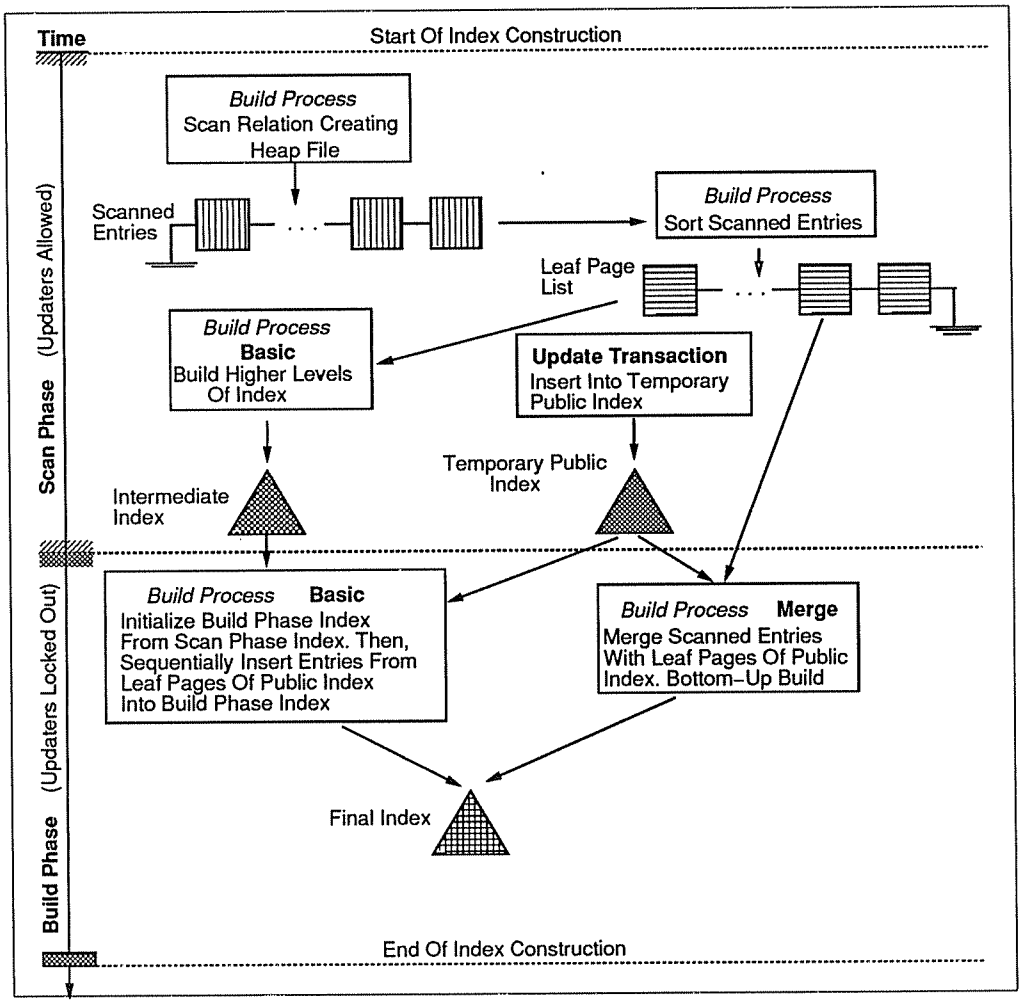

Figure 3: The Index-X-* Algorithms

During the scan and build phases, the build process in the Index-C-* algorithms behaves identically to the build process in the corresponding Index-X-* algorithms. In the catchup phase,

incorporating build phase updates into the intermediate index is done essentially like it is in the List-C-* algorithms. The main difference here is the absence of the sort step that was needed earlier for sorting update-list entries.

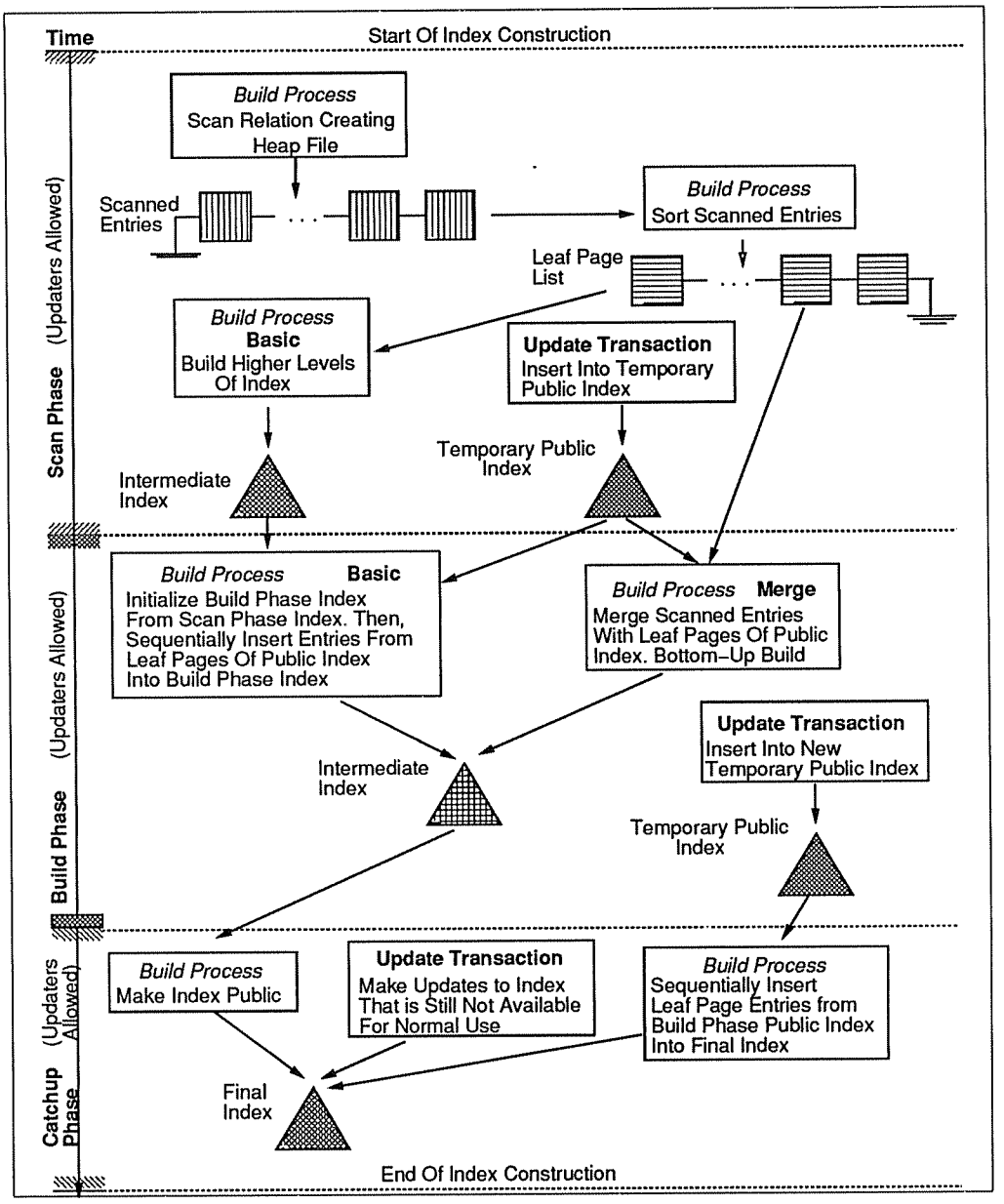

Figure 4: The Index-C-* Algorithms

Apart from the above algorithms, two other on-line algorithms have also recently been proposed in the literature [Moha91]. We will describe these algorithms later, in Section 6.1, and explain how their performance can be predicted from the performance of the algorithms already discussed above.

3 Performance Trade-Offs

As we mentioned earlier, the off-line algorithm provides the fastest way of building an index at the cost of providing no updater throughput. On the other hand, the on-line algorithms allow concurrent updaters during index construction, with the price being an increase in the time required to build the index. For the on-line algorithms then, the following question arises: How much of an increase in updater throughput is needed to compensate for a corresponding increase in the build response time? We shall try to answer this question by quantifying the *loss* to the database system caused by the index construction activity.

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

$$loss = (T_{best} - T_A) \times R_A \quad (1)$$

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

Though the loss metric provides a clean way to combine the build response time and the updater throughput into one measure, it alone is not sufficient to characterize the performance of an on-line algorithm completely. In particular, recall that some on-line algorithms (e.g., the List-X-* and Index-X-* algorithms) have exclusive phases during their execution. Since high performance transaction processing systems may have severe maximum updater response time requirements, the durations of such exclusive phases may be critical to the performance of such systems. Thus,

when evaluating an index construction algorithm, we will also consider the *off-line fraction* of the algorithm, which is defined as the ratio of the duration of its exclusive phase (if any) to that of the off-line algorithm (which has a single exclusive phase equal to its entire build response time).

In our discussion of performance trade-offs thus far, we have ignored certain hidden "lost-opportunity" costs involved in building a new index. These costs arise because the performance of the database system with the new index may be much different from its performance without the index. Not surprisingly, these hidden costs are closely related to the reason for building the new index. The decision to build a new index may be made for either of the following reasons, each of which relates to a performance-improving opportunity:

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

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

Accounting for either of these considerations above is difficult, as the interests of queries that will not be speeded up by the new index conflict with those of the queries that will indeed benefit from the new index (cases 1 and 2 above). In particular, for the queries that will benefit from the new index, the best way to build the index is to build it as soon as possible. On the other hand, for existing queries that will not benefit from the new index, the best way to build the new index is the one that creates the least interference for them during index construction. The question of which class of queries is more important and thus needs to be given priority in the system is application-specific and depends on factors like the economic benefit of preferring one class of queries over another. Such factors are hard to quantify and will vary from system to system. In our discussion of the performance of on-line index construction algorithms, therefore, we assume that the decision to build the index is made off-line, we ignore the hidden performance tradeoffs involved in index construction, and we only consider the impact of index construction on concurrent transactions that use other access paths to efficiently access and/or update the relation on which the index is being built.

4 Simulation Model

In this section, we describe the simulation model used to study the performance of the on-line algorithms described in Section 2. This model, which is a closed queueing model with a fixed

multi-programming level (MPL), was implemented using the DeNet simulation language [Livn90]. The central focus of the model is the relation on which a new index is being built.

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

The components of the database storage model include the relation on which an index is being built, an already existing B-tree index on this relation, and auxiliary data structures like the temporary lists and indices needed by the various on-line index construction algorithms. Important parameters of the database storage model include the size of the initial relation in tuples, the maximum number of tuples per relation page, and the maximum fanout of a B-tree index page. In our experiments, the physical size of a page is always the same (4K bytes), so a variation in the maximum capacity of a relation page should be viewed as being due to different tuple sizes. For modeling simplicity, all tuples are assumed to be of the same size, all index entries are assumed to be of the same size, and no duplicate keys are allowed in the index. (These simplifications should not impact our qualitative performance results.) The distribution of values for an indexed attribute of the relation are drawn from a random permutation over an integer key space of 1..400,000.

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

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

Table 3: Simulation parameters.

study, we use a terminal think time of zero, so the terminal immediately generates another operation of the same type when its current operation completes.

The simulation parameters for our experiments are listed in Table 3. In all of the experiments discussed in this paper, there is exactly one CPU in the system. Apart from the single CPU, a system configuration consists of a fixed number of disks and a fixed capacity for relation pages. In each system configuration, we varied the MPL for updaters from 0 (where only the build process executes) to a maximum of 40, conducting one experiment for each on-line index construction algorithm as well as for the off-line algorithm. At the start of each experiment, the buffer pool is initialized with randomly chosen pages from the initial relation; the build process is then started along with a number of updaters equal to the MPL. The terminal types (search, insert, and delete) are initialized according to the workload parameters. The experiment is stopped when the build process terminates. A special additional experiment is run to calculate the best updater throughput in the system without the new index (for calculating the loss using equation 1). Batch probes in the DeNet simulation language are used with the operation response time metric to generate confidence intervals. For all of the data presented here, the 90% confidence interval for relation

operation response times was within 2.5% (i.e., ±2.5%) of the mean.

5 Performance Results

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

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

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

5.1.1 Single Disk Results

The build response times for the various algorithms in the single disk case are given in Figure 5, and the updater throughput curves are given in Figure 6. As expected, the build response time for all of the on-line algorithms increases with the MPL due to an increase in resource contention at higher MPLs. We also see from Figure 5 that the List-C-Basic algorithm's build response time increases much faster than those of all the other algorithms. The reason for this is that the build process in List-C-Basic sequentially inserts unsorted entries from the update-list into the intermediate index during the build phase (Figure 2). These sequential inserts can cause multiple accesses to a given leaf page which, since the buffer pool can hold only a subset of the leaf pages in memory, results in multiple I/Os for the same page. These extra I/Os cause the build phase to become very long in this extremely disk bound situation, which in turn increases the size of the build phase update-list, thus increasing the duration of the catchup phase as well. Sorting the list before insertion into the intermediate index (as in List-C-Sort) alleviates this problem, so the response time is much lower for List-C-Sort than for List-C-Basic. In contrast to List-C-Basic, List-X-Basic does not suffer as much because its build phase length increases much more slowly due to the absence of buffer and resource contention during this phase.

In Figure 5, the build response time ordering of the on-line algorithms other than List-X-Basic and List-C-Basic reflects the increasing amount of work that they have to do for index construction. Among these eight algorithms, each of the concurrent (*-C-*) algorithms has a higher response

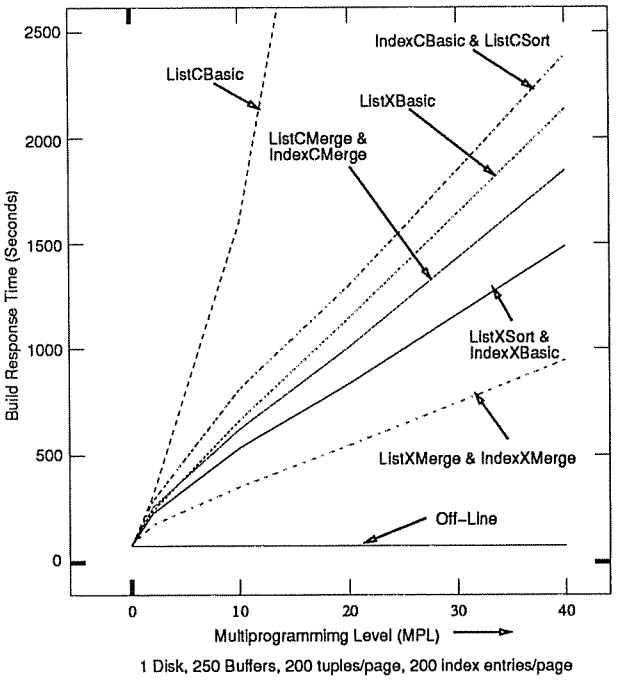

Figure 5: Build Response Time: 1 Disk.

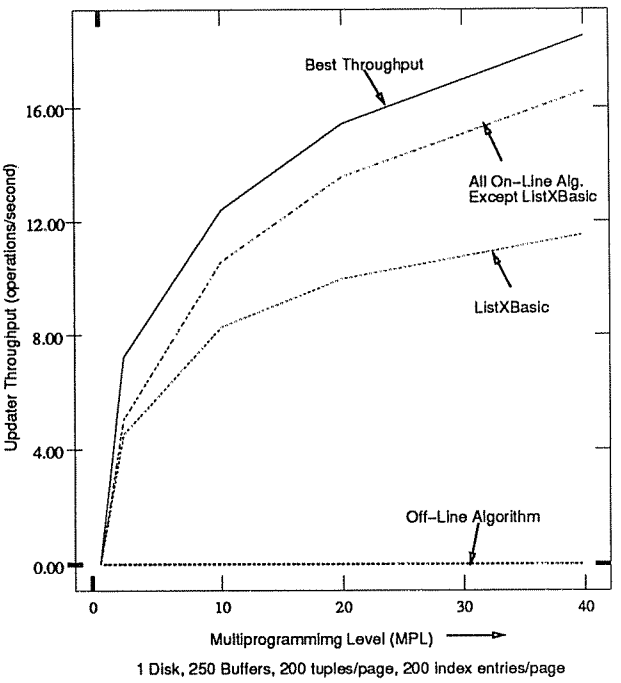

Figure 6: Updater Throughput: 1 Disk.

time than all of the exclusive (*-X-*) algorithms. This is expected since the *-C-* algorithms allow concurrency throughout the index construction period, resulting in increased contention as well as extra work for catching up. Among the four algorithms within each class, the algorithms that use merging are better than those that perform sequential inserts. This is because the sequential insert strategies perform slightly more work in the scan phase than the merge-based algorithms, while the build phases and catchup phases (if any) are comparable in length. The extra work results in expensive I/Os that increase the build response times of the sequential-insert algorithms (List-*-Sort and Index-*-Basic).

While the various on-line algorithms differ widely in their build response times, all except List-X-Basic attain the same updater throughput[3] (Figure 6). In particular, this means that all of the *-X-* algorithms except List-X-Basic attain the same throughput as the *-C-* algorithms. This is surprising since we would expect the *-X-* algorithms to attain less throughput than the *-C-* algorithms due to their exclusive build phase. The reason this is not so is due to the extremely high level of resource contention in this experiment. In all of the *-X-* algorithms except List-X-Basic, a bottleneck at the lone disk increases the scan phase duration tremendously at high MPLs, while the (exclusive) build phase duration remains about the same due to lack of contention; the build phase therefore forms a negligible part of the build response time, causing a negligible effect on the

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

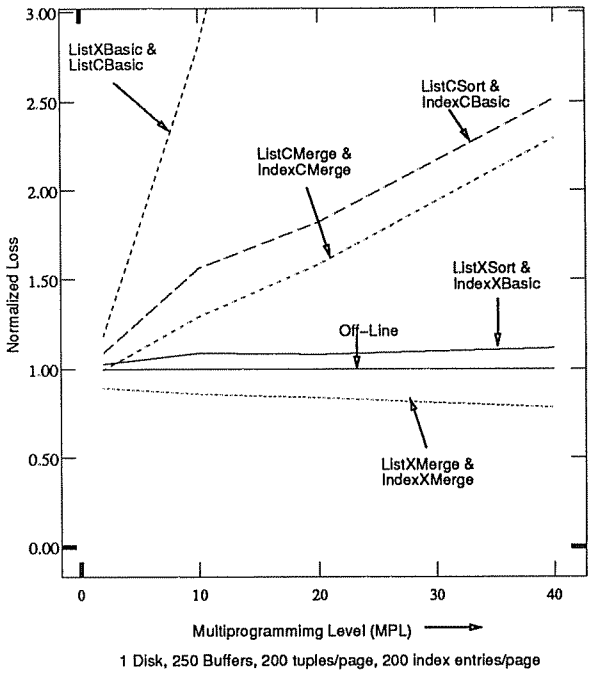

Figure 7: Loss: 1 Disk.

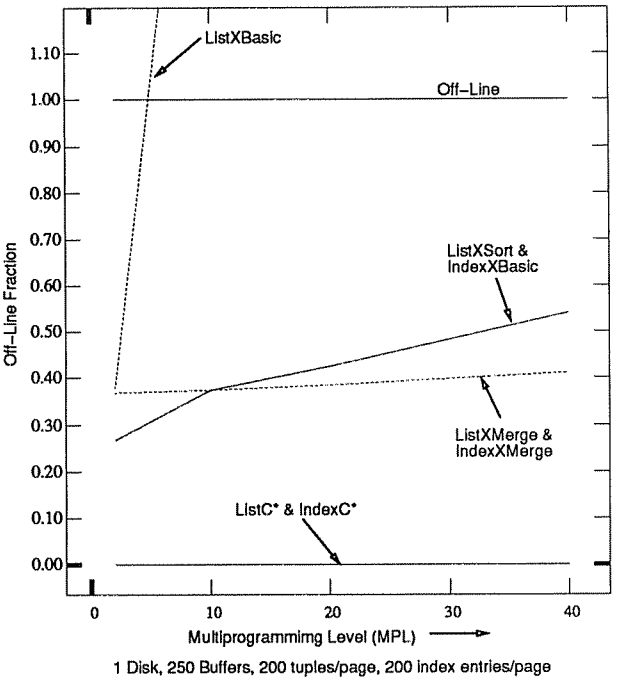

Figure 8: Off-Line Fraction: 1 Disk.

throughput. In List-X-Basic, however, the extra I/Os during the build phase cause this phase to become a significant fraction of the build response time at higher MPLs, resulting in a significant drop in maximum throughput.

Having looked at the updater throughput and response times separately, we now look at the normalized loss in Figure 7 in order to combine both measures. Recall that the normalized loss for a given algorithm is the ratio of the loss for the algorithm (calculated using Equation 1) to the loss for the off-line algorithm. Note from Figure 7 that the loss for the List-X-Basic and the List-C-Basic algorithms is very high compared to that for the other algorithms (so much so that their values for high MPLs are omitted from the figure). The large loss in List-C-Basic is due to its very high response time (Figure 5), while the loss in List-X-Basic is due to its low updater throughput (Figure 6). The normalized loss metric also gives an idea of the improvement achieved by using an on-line algorithm instead of the off-line algorithm. The loss curves in Figure 7 show that, at high MPLs, only two merge-based algorithms (List-X-Merge and Index-X-Merge) manage to achieve better loss than the off-line algorithm. As shown, the loss for the *-C-* algorithms increases with MPL and reaches a maximum value of between 225% to 250% of the loss for the off-line algorithm. In contrast, the losses for the *-X-* algorithms (except List-X-Basic again) track the loss for the off-line algorithm more closely.

Despite the loss results, it is not necessarily the case that the *-X-* algorithms are preferable to the *-C-* algorithms here, as they might have an unacceptably large exclusive build phase. In

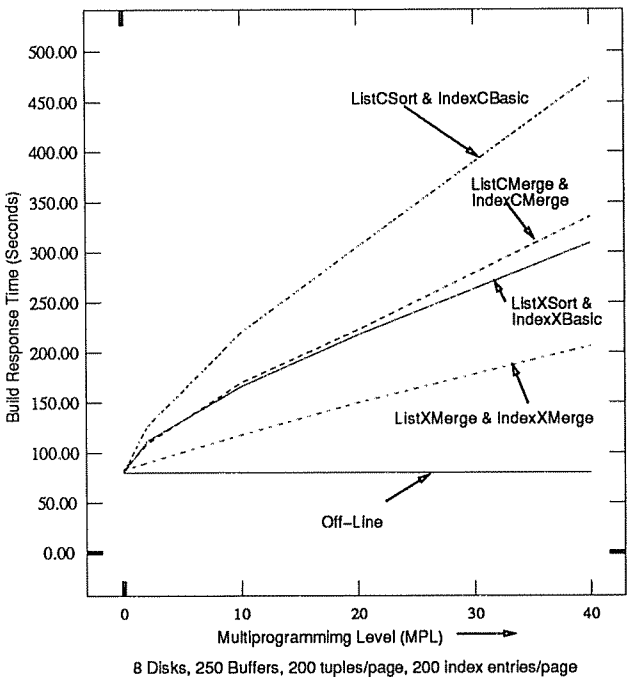

Figure 9: Build Response Times, 8 Disks.

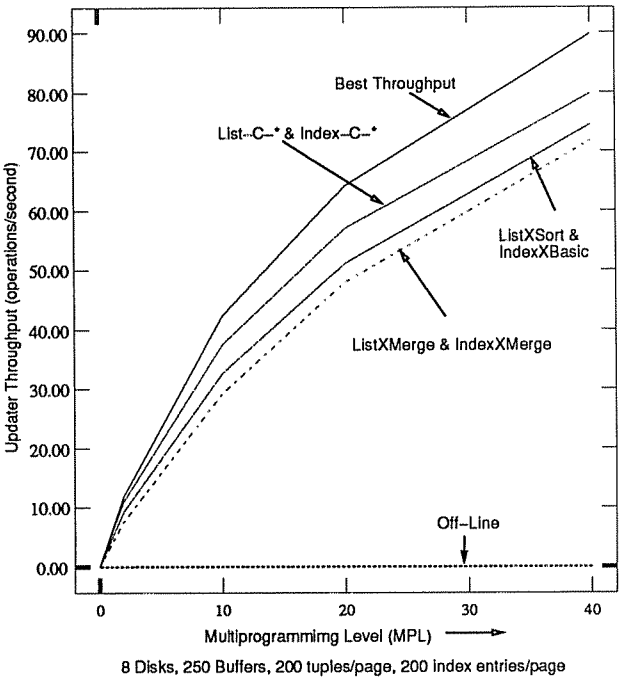

Figure 10: Updater Throughput, 8 Disks.

order to investigate this, we plot the off-line fraction of these algorithms in Figure 8. We see that the duration of the exclusive build phase for the List-X-* and the Index-X-* algorithms is a sizable fraction (25% to 50% for all algorithms except List-X-Basic) of the total response time of the off-line algorithm. Thus, if it is unacceptable for updaters to wait in the case of the off-line algorithm, it is likely to be unacceptable for them to wait in the *-X-* algorithms as well (since the waiting times are of the same order of magnitude). Using the best among the *-C-* algorithms (List-C-Merge or Index-C-Merge) thus seems to be a better choice here even though they have a higher loss than most of the *-X-* algorithms.

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

5.1.2 Eight Disk Results

The build response time and updater throughput curves for the eight disks case are given in Figures 9 and 10 respectively. It can be seen from the build response time curves that the ordering

of response times among the various on-line algorithms is the same as in the one disk case. The key difference here is that the best on-line algorithm now has a maximum response time of only a few times the response time of the off-line algorithm, while in the one disk case the best on-line algorithm was more than ten times slower than the off-line algorithm at an MPL of 40 (Figure 5). Adding disks has reduced the level of resource contention considerably for the builder, resulting in a faster index building time for all of the on-line algorithms.

From the updater throughput curves (Figure 10), we see that the List-X-* algorithms and the Index-X-* algorithms have a slightly lower throughput than the corresponding *-C-* algorithms. In the *-X-* algorithms, the (exclusive) build phase now forms a significant proportion of the index construction time, thus leading to a noticeable loss in throughput. Another point to be noted from Figure 10 is that the List-X-Sort and the Index-X-Basic algorithms attain a slightly higher updater throughput than the List-X-Merge and the Index-X-Merge algorithms. Again, the reason is that the build phase proportion is greater in the two merge-based algorithms than in List-X-Sort and Index-X-Basic at high MPLs. This is because the (non-exclusive) scan phase in List-X-Sort and Index-X-Basic is slightly longer than for the merge-based algorithms, while the build phases are comparable in length. As seen in the single disk case earlier, extra work in the scan phase is expensive at high MPLs due to resource contention; this fact is shown by the higher build response times of List-X-Sort and Index-X-Basic in Figure 9.

In Figure 11, we present the normalized loss for the various algorithms in the eight disks case. In contrast to the results of the one disk experiments, where the loss for the *-C-* algorithms was larger than that for the other algorithms, the *-C-* algorithms (which have no exclusive phases) perform better than the other algorithms in terms of the loss metric here. The off-line fractions of the various algorithms for this experiment were the same as in the earlier high contention case (Figure 8), so we omit those curves here; this is to be expected since the duration of the periods during which updaters are locked out should be the same as in the one disk case (since the build process is the only active process in the system during that time). Since the off-line fraction for the *-C-* algorithms is almost zero, and they also have the least loss, they are unequivocally better than the *-X-* algorithms in this situation. Among the *-C-* algorithms, List-C-Merge and Index-C-Merge are the best since they involve the least overhead (as demonstrated by their superior build response times).

This set of experiments (both the single and eight disk results) examined the case where an index entry is the same size as a tuple. We also ran experiments in cases where the tuple size is ten and then a hundred times the size of an index entry, respectively. Due to space limitations, we will present only the results for the large tuple experiments. Since we found above (and in all our other experiments as well) that the merge method of building the index was better overall than

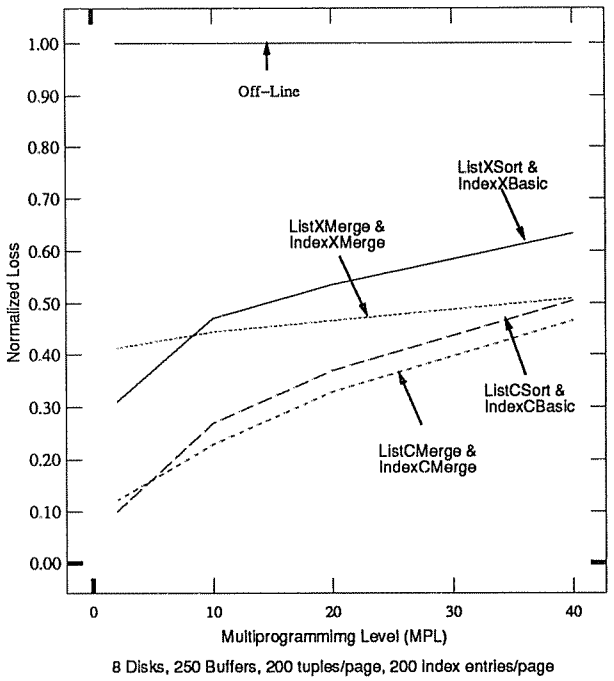

Figure 11: Normalized Loss, 8 Disks.

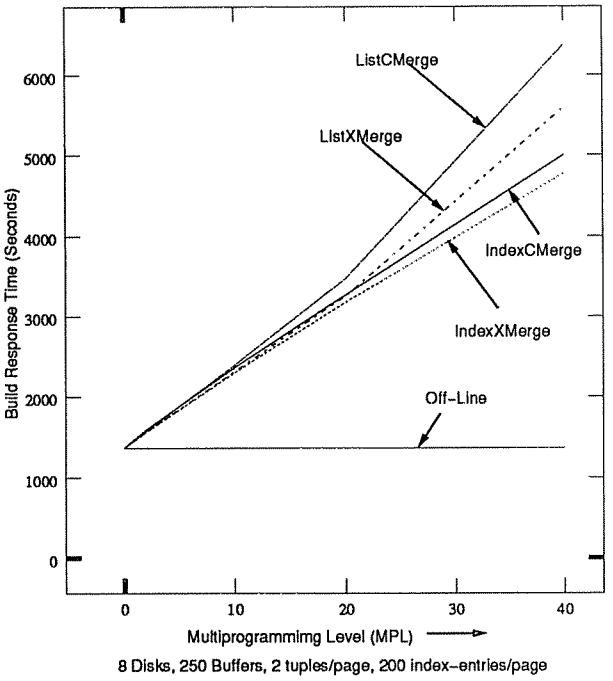

Figure 12: Response Times, Large Tuples.

the basic and sort strategies, we will show only the four merge-based algorithms and the off-line algorithm in the remaining graphs.

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

In this set of experiments, a tuple is a hundred times larger than an index entry, unlike in the earlier set of experiments where they were the same size. This causes an increase in the relation size (200MB here as compared to 2MB in Experiment Set 1) while the size of the index remains the same as before ($\sim$ 2MB). This increase, in turn, causes the scan phase to dominate the process of index construction (accounting for more than 95% of the build response time). Since a bottleneck at the disk can be expected to swamp the performance differences between the various on-line algorithms in the one disk case, as we saw earlier in the small tuple experiments, we only conducted experiments on a system with eight disks here.

5.2.1 Eight Disk Results

The build response time curves for the large tuple experiments on a system with eight disks are given in Figure 12. Compared to the response time differences seen in the corresponding small tuple experiments (Figure 9), the relative response time differences between the various on-line algorithms are smaller here. This is because the scan phase (during which all on-line algorithms perform similarly) is dominant, and the other phases (which are primarily responsible for build response time differences) form only a small portion of the overall index construction time. Another thing to note is that all of the list-based algorithms perform slightly worse at high MPLs here than all of the index-based algorithms. The reason is that the update-list becomes large, due to the large

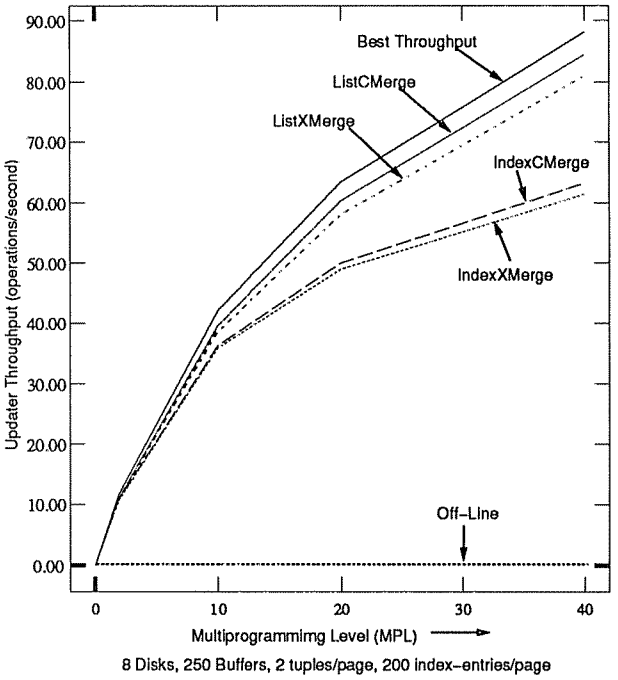

Figure 13: Throughput, Large Tuples.

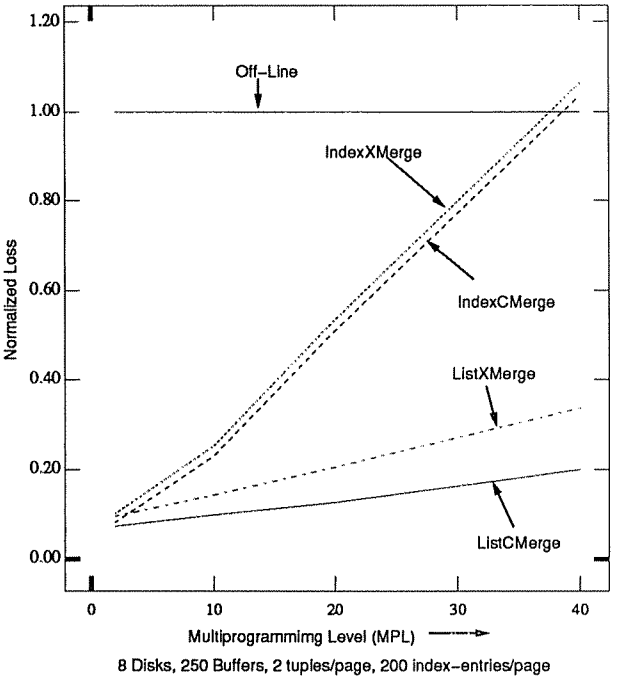

Figure 14: Normalized Loss, Large Tuples.

scan phase, and the sorting that takes place during the build phase of the list-based algorithms thus contributes a significant overhead which is absent in the index-based algorithms. Also, between the two list-based algorithms, List-X-Merge is better than List-C-Merge; this is due to the additional catchup phase in List-C-Merge. Similar behavior is seen between the index-based algorithms.

The updater throughput curves for this experiment are given in Figure 13. As indicated, the throughput for the index-based algorithms is significantly less (by about 25%) than that of the list-based algorithms at high MPLs. This is because the public index (into which concurrent updaters insert their updates) in the index-based algorithms becomes large enough at high MPLs in this experiment for every index access to have a high probability of performing a disk I/O for a leaf page. This extra disk access causes a significant increase in the overhead of concurrent updaters in the index-based algorithms, while there is no such overhead in the list-based algorithms (since the append to the list almost certainly does not involve a disk access). The reason why this effect was not significant in the small tuple experiments is that the scan phase was much smaller there and, even at high MPLs, the number of updates recorded in the public index did not cause it to become large enough for its leaf pages to be paged out often. Apart from the above differences in throughput between the list-based and the index-based algorithms, we find that between the list-based algorithms, the List-C-Merge algorithm attains a slightly higher throughput than the List-X-Merge algorithm. This is because there is no exclusive phase in List-C-Merge; similar behavior is exhibited by the index-based algorithms.

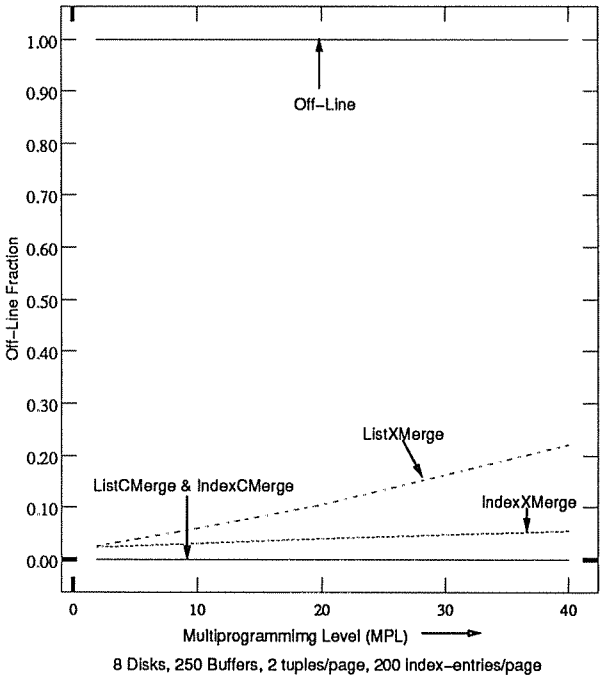

Figure 15: Off-Line Fraction, Large Tuples.

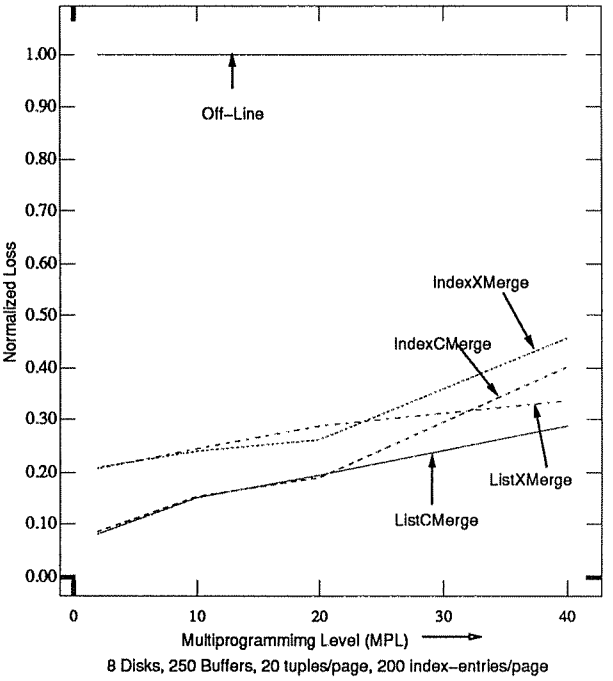

Figure 16: Normalized Loss, Medium Tuples.

In order to understand whether or not the increase in throughput achieved here by the list-based algorithms compensates for their increase in response time, we plot the normalized loss for each of these algorithms in Figure 14. The loss curves indicate that the index-based algorithms suffer quite a bit due to their reduction in throughput. In fact, at an MPL of 40, Index-X-Merge and Index-C-Merge are even a bit worse than the off-line algorithm. In contrast, List-X-Merge and List-C-Merge perform much better than the off-line algorithm throughout the entire MPL range, with a maximum normalized loss of 35% for List-X-Merge and only 20% for List-C-Merge. Finally, the off-line fractions for the various on-line algorithms are given in Figure 15. As expected, the off-line fractions for the *-X-Merge algorithms are smaller here than in the small tuple experiments (Figure 8), but they are still not negligible. In fact, the off-line fraction for the List-X-Merge algorithm increases with MPL from less than 5% to a maximum value of greater than 20% due to the overhead of the sort performed during its exclusive build phase.

5.3 Other Experiments

Apart from the large (2000 bytes) and small (20 bytes) tuple experiments, we also performed experiments in which the tuple size was intermediate (200 bytes) as mentioned earlier. In these medium tuple experiments, an index was approximately one-tenth the size of the relation. The results of the medium tuple experiments were essentially a hybrid of the results of the small and large tuple experiments. To illustrate the performance of the algorithms there, we reproduce their loss curves for a system with eight disks in Figure 16. From these curves, it is clear that at lower MPLs the trends are like those of the small tuple experiments (Figure 11), where the *-C-*

algorithms had smaller losses than the *-X-* algorithms, while at higher MPLs the trends are like those observed in the large tuple experiments (Figure 14), where the list-based algorithms had smaller losses than the index-based algorithms. This behavior is expected since the scan phase proportion (which affected the relative performance of on-line algorithms in the small and large tuple experiments) here lies between those of the small and large tuple experiments. A final point to note from Figure 16 is that List-C-Merge again has the least loss throughout the range of updater MPLs considered.

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

6 Discussion

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

- Except in extremely resource bound situations, most of the on-line index construction algorithms clearly outperformed the off-line algorithm. In other words, the throughput that the on-line algorithms achieved for updaters during index construction more than compensated for their increase in build response time.
- Among the on-line algorithms, the best among the algorithms with no exclusive phase (List-C-Merge) outperformed the best among the algorithms with an exclusive phase (List-X-Merge) except in heavily resource bound situations.
- Even in heavily resource bound situations, the best fully concurrent algorithm (List-C-Merge) had a loss of only a few times that of the best partially exclusive algorithm (List-X-Merge). Furthermore, List-X-Merge was found to have an exclusive phase whose length was a non-negligible fraction of the response time of the off-line algorithm, likely making it unacceptable for use in high performance transaction processing systems.
- As should be expected, the relative performance of the various algorithms depended on the proportion of time spent in the initial relation scan phase of index construction. The list-based algorithms performed better than the index-based algorithms when the scan phase was a large proportion ($> 95\%$) of the index building time. When the scan phase was around 50% of the index building time, the fully concurrent (*-C-*) algorithms were found to be superior to the partially exclusive (*-X-*) algorithms.
- The merge strategy for building the index was clearly superior in performance to the basic and sort strategies.

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

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

6.1 Other Candidate Algorithms

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

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

While we have not explicitly simulated the two [Moha91] algorithms in our experiments, we believe that their performance can be inferred from that of the Index-C-Basic and the List-C-Basic algorithms. First, the performance of our Index-C-Basic algorithm should be comparable to (or better than) the performance of their index-based algorithm. This is because the list of concurrent

updates is inserted into the index built with the scanned entries in the Index-C-Basic algorithm, while in their index-based algorithm the sorted scanned entries are inserted concurrently into the index built with concurrent updates. In realistic situations, the list of concurrent updates is likely to be much smaller than the list of scanned entries, leading Index-C-Basic to perform better than their index-based algorithm.

Turning to their list-based algorithm, the qualitative performance of their list-based algorithm seems to be similar to that of our List-C-Basic algorithm. Their algorithm should perform better than List-C-Basic (in the best case having about half of the response time of List-C-Basic) since updaters selectively (rather than always) record updates to the update-list. Note, however, that our list-based and index-based algorithms can also be modified to make use of a similar optimization [Srin91b]. Since we found the performance of List-C-Basic to be an order of magnitude worse than that of the other on-line algorithms (Figures 5 and 7), their list-based algorithm can be expected to perform similar to List-C-Basic relative to the other on-line algorithms (once they too are optimized to selectively record updates). Finally, due to the multi-phase catch-up strategy used in their list-based algorithm, a race condition could occur if the rate of concurrent updates to the update-list happens to be higher than the rate at which the build process can apply these updates to the intermediate index; under such circumstances, the build process may never terminate. This problem condition cannot occur in the *-C-* algorithms of Section 2 due to the fact that the build process and the concurrent updaters share the same index during the catchup phase.

7 Conclusions

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

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

In terms of further work, this study has only examined concurrency issues that affect the performance of on-line index construction algorithms. As mentioned earlier, index construction for a terabyte relation may take days, which means that the index building process must be able to complete without having to restart from scratch after every crash. Appropriate recovery strategies thus have to be designed for this purpose. (Recovery strategies are described for the algorithms in [Moha91] and could be adapted for the algorithms studied in this paper.) Also, 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 parallel database systems. Still another interesting issue is to extend our on-line index construction strategies to work for indices other than B-tree indices; a further generalization would be to extend the ideas employed here for use in computing aggregate functions on a relation with minimal interference. These issues are all promising candidates for future work.

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),** (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.

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

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

[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).

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

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

[Srin91a] Srinivasan, V. and Carey, M. J. "Performance of B-tree Concurrency Control Algorithms", *Proceedings of the ACM SIGMOD Conference,* May 1991.

[Srin91b] Srinivasan, V. and Carey, M. J. "On-Line Index Construction Algorithms", *Proceedings of the High Performance Transaction Systems Workshop,* September 1991, to appear.

[Ston88] Stonebraker, M., Katz, R., Patterson, D. and Ousterhout, J. "The Design of XPRS", *Proceedings of the 14th VLDB Conference,* Los Angeles, CA, August 1988.

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