Online B-Tree Merging

Xiaowei Sun   Rui Wang   Betty Salzberg *
College of Computer and Information Science, Northeastern University
{xwsun, bradrui, salzberg}@ccs.neu.edu

Chendong Zou
IBM
zouc@us.ibm.com

ABSTRACT

Many scenarios involve merging of two B-tree indexes, both covering the same key range. Increasing demand for continuous availability and high performance requires that such merging be done online, with minimal interference to normal user transactions. In this paper we present an online B-tree merging method, in which the merging of leaf pages in two B-trees are piggybacked lazily with normal user transactions, thus making the merging I/O efficient and allowing user transactions to access only one index instead of both. The concurrency control mechanism is designed to interfere as little as possible with ongoing user transactions. Merging is made forward recoverable by following a conventional logging protocol, with a few extensions. Should a system failure occur, both indexes being merged can be recovered to a consistent state and no merging work is lost. Experiments and analysis show the I/O savings and the performance, and compare variations on the basic algorithm.

1. INTRODUCTION

Many application and system maintenance scenarios require merging of two B-tree indexes covering *the same key range*. Two such scenarios occur during data migration in a parallel database system where data partitioning is not by key range, but by other methods such as hashing. First, for load balancing, data partitions may move from a "hot" node to a "cool" node. Second, when a node is to be deleted, the data on that node has to be distributed to other nodes. Accordingly, at the destination node, any *primary B-tree index* (with data in the leaf pages) and all *secondary B-tree indexes* (with references to data in the leaf pages) should be updated. For this purpose, temporary B-trees may need to be constructed on the moved data and merged later with the already existing B-tree indexes.

A third scenario occurs during batch data insertion in a centralized database system. For high efficiency, temporary B-tree indexes on newly inserted data may need to be constructed and merged into existing indexes. Yet another scenario occurs during the maintenance of a partitioned B-tree index [5]. In a partitioned B-tree, the whole tree is partitioned according to an artificial leading column.

Each partition is a B-tree. During system maintenance, different partitions may need to be merged.

1.1 Straightforward Approaches

First let us look at some straightforward approaches to B-tree merging and describe some of their drawbacks.

Off-line Approach: In off-line B-tree merging, both indexes are X-locked until the merging is finished. All incoming user transactions will wait until the locks are released. Thus user transactions are forbidden access to the indexes while the indexes are being merged. This is the simplest approach but may incur an unacceptable amount of waiting time for user transactions.

Eager Approach: Another approach requires all user transactions to access *both* indexes while merging is going on and has separate non-user transactions move batches of entries *eagerly*. The system source code for operations such as exact-match searching, range searching, insertion, deletion and updating on B-trees must be extensively changed to adjust to the requirement of accessing both indexes. This complicates user transaction logic (or B-tree access module logic if the B-tree access module accesses both indexes on behalf of user transactions).

Background Approach: A third approach makes a copy for each index, merges them in the background, then catches up changes by applying log records from the indexes in use, finally switches the merged index online. This approach involves higher complexity, and suffers from extra space requirements and I/O cost for copying (thus low efficiency), especially when the indexes are very large. In addition, as in the eager approach, user transactions must access both indexes while merging is taking place.

For continuous availability, the merging of two B-tree indexes should be done *online*, allowing concurrent user transactions, including searches and modifications. For stable system performance, the merging process should be *efficient*, without slowing down the concurrent user transactions too much. In addition, for *practicability*, the merging algorithm should be relatively easy to integrate into realistic DBMS implementations. None of the straightforward approaches just discussed satisfies all these requirements. We have sought to improve on the straightforward approaches.

1.2 Lazy Approach

This paper presents an online B-tree merging approach where merging is piggybacked *lazily* on user transactions. We designate one B-tree index as the **main-index** and the other (usually the smaller one) as the **second-index**. User transactions operate through the

main-index only. Upon reaching a leaf page, say *L*, of the main-index, if *L* has not yet been updated to include the entries from the second-index belonging to *L*'s key range, a system transaction takes over from the user transaction and performs the merging: moving all entries in *L*'s key range from the second-index to *L*, and splitting *L* if necessary. At this point, the user transaction resumes. In this way, only the system transaction deals with merging. The user transaction does not have to interface with both indexes as would occur in the *eager* and the *background approaches*.

Because the I/Os on the main-index required separately by the merging are piggybacked onto user I/Os, the I/O cost for merging is reduced considerably. Compared to the *eager* and the *background approaches*, our approach is more I/O efficient, especially when the second-index is much smaller than the main-index and large parts of it therefore can be found in buffer frequently during merging (true for most application and system maintenance scenarios). Furthermore, lazy merging is incrementally beneficial, because future user transactions which access merged parts of the main-index will have a shorter execution path.

We do not piggyback all of the merging however. To speed up completion of the merging, we also proactively merge index entries using a background thread which is usually activated when the system is not busy. This has an additional benefit of condensing the unmerged part of the second-index and further reducing the merging I/O cost.

So in our approach, a leaf page of the main-index is merged either when it is accessed by a user transaction (saving merging I/Os) or by the background thread (during low system load).

To provide high concurrency, only the smallest number of index pages required for merging one main-index leaf page correctly are made unavailable during a piggybacked merge operation. These pages are released as soon as the leaf page is merged and then control is returned to the user transaction. Compared to the *off-line approach*, our approach has much better availability.

Forward recovery for merging is made simple by following a conventional logging protocol with a few extensions. Should a system failure occur, both indexes being merged can be recovered to a consistent state. A system failure does not affect correctness, nor does it imply starting over again from the beginning.

In addition, since most modifications to DBMS code are only in the B-tree access module and no user transaction logic needs to be modified, the integration of this approach would be relatively easy.

In summary, the *contribution* of this paper is that it provides an *efficient*, *recoverable* and *practical online* B-tree merging method. It has cleaner code and less I/Os than the *eager* and the *background approaches*, and far better availability than the *off-line approach*.

1.3 Organization

The rest of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the lazy B-tree merging algorithm in detail. Section 4 explains how to complete the merging and discusses recovery issues. Section 5 presents the performance analysis and experimental results, including comparison with the *eager* and the *off-line approaches* (not with the *background approach* due to its obvious drawbacks). Section 6 concludes this paper. We use **boldface** when defining terms and *italics* for emphasis.
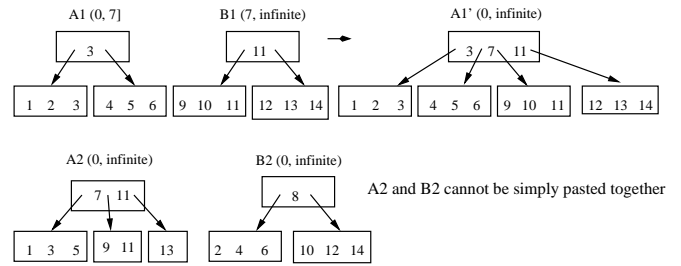

Figure 1: $A1'$ **is the tree after attaching** $B1$ **to** $A1$.

2. RELATED WORK

Several papers have considered creation and maintenance of B-tree indexes. A method to create indexes without disabling updates to the table was presented in [16]. Both [18] and [21] described methods to rebuild B-tree indexes online. Paper [4] presented a method to generate efficient execution plans for bulk deletes, where deleting entries from indexes in batch is done off-line. Without considering recovery issues, [1] described techniques to update indexes along with data movement in shared nothing parallel databases. These papers do not discuss B-tree merging.

Both [7] and [17] exploit the idea of merging in-memory data structures (possibly B-trees) with disk-resident data of large volume and user queries have to search both the persistent and in-memory data structures. In both the *k*-way merge algorithm in [7] and the rolling merge algorithm in [17], the only type of update by the user transactions is insertion into the in-memory data structures, and B-trees participating in the merge are never used for updates. Neither paper gives details about how to merge two B-trees online.

To migrate data in a parallel database system partitioned by key range, [9] proposed the idea of integrating one B-tree into another. The B-trees merged cover *disjoint key ranges*, thus one B-tree can be simply "attached" to the other. Such an attachment algorithm can not be used to merge two trees which have *the same key range*. This is illustrated in Figure 1. Trees $A1$ and $B1$ cover disjoint key ranges and thus they can be simply pasted together, while $A2$ and $B2$ cover the same key range and cannot use this method.

The idea of exploiting I/O from user transaction requests to piggyback necessary index updates was used in [22]. That work deferred the updates to references in indexes after the data records had been physically moved, and piggybacked them with user transactions. Although we also use the idea of piggybacking on user transactions, our algorithms are very different. We are dealing with merging two trees, not applying updates from a dynamically changing list, and we do not modify the buffer manager as is done in [22].

3. LAZY B-TREE MERGING

In this section, we present our B-tree merging algorithm, which is described as a **lazy** algorithm because merging is normally triggered by user transactions. Without loss of generality, we assume that the main-index has at least two levels. First we briefly present some background on B-trees and on the safe-node concurrency control protocol assumed in this section. Then we describe our merging algorithm in detail. In the end of this section, we show that our algorithm can be adapted to B-link trees [10], merging of which has simpler concurrency control.

3.1 Background: B-Tree Concurrency Control

We describe our algorithm in the context of B$^+$-trees [3]. Leaf pages in a B$^+$-tree are linked together left-to-right through forward side pointers, while pages at the same internal level are not linked together. We shall refer to B$^+$-trees simply as B-trees in the remainder of the paper.

We distinguish *latches* from *database locks*. **Latches** (sometimes called lightweight locks) are semaphores on buffer pages [6]. No deadlock detection will be made on latches. Instead latches are always obtained in a top-down or left-to-right (the leaf level) order to prevent deadlock. **Database locks** are locks held by transactions on data items. These locks are normally held to end of transactions and are kept in a lock table. For database locks, deadlocks are not prevented but are detected by standard techniques such as finding cycles in a waits-for graph.

The concurrency control for B-tree traversal on both indexes during merging is based on the **optimistic safe-node** latching protocol from [2]. In this protocol, *searchers* **latch-couple** (i.e. release the latch on a parent only after the latch on the child is acquired) down to the leaf level with S-latches. An *updater* latch-couples down to the parent-of-leaf level with S-latches and then obtains an X-latch at the leaf. In the case when an entry is to be inserted in a leaf page of a B-tree and insertion causes a split, the updater gives up all latches and starts again at the root, this time requesting X-latches and only releasing latches on the ancestors of the **safe node** (non-full page), thus ensuring that X-latches are held on all pages on the path to the leaf which might need updating. A similar protocol is followed for deleters, where the **safe node** in this case is a page which is not so sparse that it would require consolidation if an entry is removed.

3.2 Detection of Non-Merged Pages

To ensure that the benefit of the incremental merging is available to user transactions as early as possible, we need some mechanism to find out whether a main-index leaf page has already been merged or not, so that merging is not attempted where it is not needed.

We will use the **log sequence number** (**LSN**) of a page which indicates the number of the log record corresponding to the most recent update on the page. In the case the page is a leaf, we call this a **leafLSN**. We introduce a system variable, **reorgLSN** which is the system LSN at the time the whole merging process begins. Before the merging process begins, this reorgLSN is stored both on disk and in memory. Any main-index leaf page that is updated (due to merging or user index accesses) while the whole merging process is going on will have a leafLSN larger than the reorgLSN.

However, if a main-index leaf page has no matched entries in the second-index (thus need not be merged), and if the page is not updated by the user transaction index access either, it is possible that the page maintains a leafLSN smaller than the reorgLSN throughout the whole merging process. Subsequent user requests to this page, without any knowledge that this page has already been looked at, will continue to trigger merging (which is unnecessary). This will not be incorrect, but it will be inefficient.

Without introducing auxiliary durable structures, we use **no-op log records** to prevent unnecessary triggering of merging attempts. For a main-index leaf page having no matched entries, we can write a
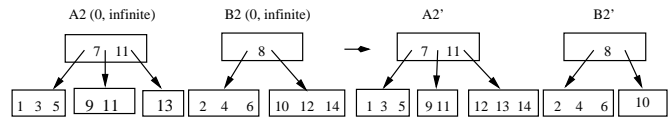

Figure 2: $A2'$ **and** $B2'$ **are the trees after the key range of the right most leaf page of** $A2$ **was merged with entries from** $B2$.

special no-op log record and increment the leafLSN just to indicate that this page has been looked at. We show later that in most cases, a large percentage of the main-index leaf pages require merging. Thus the number of no-op log records should be small.

3.3 Basic Lazy Merging Algorithm

In this section we present the basic algorithm for lazy merging. Suppose some B-tree access traverses down the main-index searching for key k. In order to completely merge the leaf L whose key range R contains k, we need to know this key range R. Normal B-tree top-down traversal can derive R by the time it reaches L. When L is found not merged yet, a system transaction (similar to the concept of nested top actions in [15]) is triggered to merge L. The system transaction merges all second-index entries in the key range R into the main-index and commits independently from the triggering user transaction, after which the normal B-tree access may resume. This system transaction always changes the leafLSN on L, either because L is updated with newly inserted entries from the second-index, or by using a no-op log record. If this merging triggers a split, the resumed B-tree access will retraverse the main-index.

Figure 2 shows an example: the right most leaf page of the main-index $A2$ was accessed by a user transaction, thus the key entries of $B2$ in that corresponding key range $(11, infinite)$ were merged into $A2$ by a system transaction.

Since this merging process involves concurrent accesses on two B-trees, we require that all latches on the main-index are issued before latches on the second-index. If one thread needs to request latches on the main-index while holding latches on the second-index, the latches on the second-index should be released first. This orders the resources being latched and prevents deadlocks involving both indexes.

The basic lazy merging algorithm used by a B-tree top-down access T looking for k is outlined as follows. Figure 3 illustrates the logic, with the modification to normal B-tree top-down access rendered inside the dashed-line box.

--Basic lazy merging

1. While holding an S-latch on the parent P of the leaf L, T acquires a latch on L (S-latch for search or X-latch for update). At this point, the key range R for L is known.

2. T checks the LSN of L, $leafLSN$:

   (a) If leafLSN $>$ reorgLSN, no merge is needed. T releases the S-latch on P and proceeds to step 5.

   (b) Otherwise (leafLSN $\leq$ reorgLSN), if the latch T has on L is an X-latch, T proceeds to step 3; otherwise, T releases the S-latch on L and re-acquires an X-latch on L, then repeats step 2. (The S-latch on the parent P is still held, so no retraversal is required.)
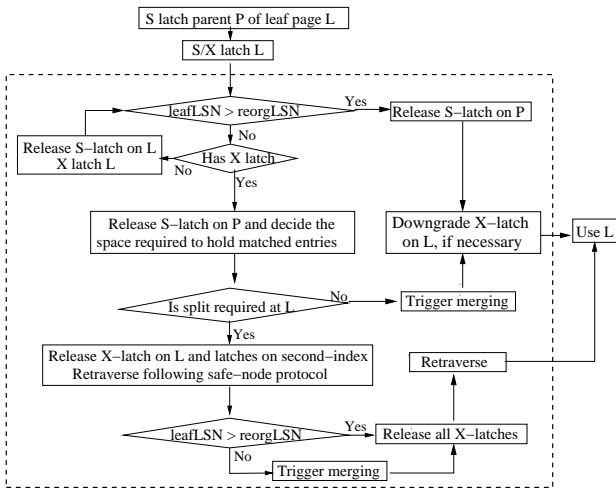
Figure 3: Basic lazy merging.

3. At this point T has an X-latch on L. T releases the S-latch on P and accesses the second-index in the key range R to decide the space required to hold entries to be moved in, then attempts to merge:

   (a) If L does not need to be split to hold the matched entries to be moved in, T triggers the merging as a system transaction. After the merging is done, T proceeds to step 5.

   (b) Otherwise (split needed), T releases the X-latch on L and all latches on the second-index, and follows the safe-node algorithm to retraverse looking for k, with all pages to be updated by this merging X-latched.

      i. If T finds that L has not been merged (by checking the leafLSN), T triggers the merging as a system transaction.

      ii. T releases all X-latches on the main-index and proceeds to step 4.

4. T retraverses the main-index using the same key k, and proceeds to step 6.

5. If T holds an X-latch on L and T's original request was an S-latch, T downgrades the X-latch to an S-latch.

6. T accesses page L, which is up to date at this point.

Step 2(b) utilizes the fact that keeping an S-latch on the parent P while releasing any latch on L itself ensures that L is neither split nor deallocated, so that L can be re-latched later without retraversal and k is still in L's key range. After retraversal in step 4, the leaf page whose key range contains k may not be L any longer.

Before the merging begins, all pages in the main-index to be updated have been X-latched. Before the merging ends, all latches on the second-index should be released. After a leaf page is merged, retraversal is required only when the merging involved a split.

The matched entries are removed from the second-index during the merging. It is tempting to try to make the second-index read-only, and just deallocate it at the end of merging. However, we would then be unable to tell when merging of all second-index entries has finished and the read-only second-index would occupy more buffer

Table 1: Interface for MergeEvents.

$set(L, latchType)$	request to set a mergeEvent
$notifySplit(L)$	notify a mergeEvent of a required split
$reset(L)$	reset a mergeEvent

space during the merging than otherwise (removing entries from the second-index during the merging), thus degrading performance.

When a split is required, it is possible that two or more concurrent B-tree accesses will attempt merging and splitting the same leaf page. For example, a B-tree access $T1$ gets the X-latch on L and finds that L needs to be merged. After accessing the second-index (to decide the space required to hold matched entries), $T1$ releases its X-latch because a split is needed, and further $T1$ is swapped out. Then another B-tree access $T2$ obtains the X-latch on L and also finds that L needs to be merged. After accessing the second-index, $T2$ also releases its X-latch. Whichever gets all required X-latches in step 3(b) first will perform the merging. Then the other one will also obtain the same X-latches later only to discover that the merging has been finished.

This scenario happens because there is no means for other concurrent B-tree accesses to identify whether the merging is in progress, accordingly there is no way to promptly stop their attempts to merge the same leaf page. In the following section we propose a low-cost optimization for this split-related situation.

3.4 Optimization for The Split Case

For this optimization, we introduce a new type of lightweight lock called a **mergeEvent**, to indicate the merging for a main-index leaf page is in progress. Using a mergeEvent, *only one* B-tree access will attempt to merge a given leaf page. Our later experiments verify that this optimization eliminates the CPU cost from unnecessary attempts in a high-split and high-contention situation without incurring much overhead in other situations.

The *mergeEvent* can be implemented simply by two semaphores [6]. It is dynamically created for any main-index leaf page, and requires no deadlock detection. The interface for mergeEvents is in Table 1. In this table, $latchType$ means the type of latch (S or X) currently on a leaf page L held by the requestor.

After determining that merging is needed on a leaf page L, a B-tree access will request to set a mergeEvent on L. When requesting to set a mergeEvent, the requestor holds an S-latch on the parent P. Only the first requestor on L will attempt to trigger the merging. Other subsequent requestors are immediately blocked until the merging is finished.

Suppose T_S is one of the subsequent requestors which are blocked while the merging of L takes place. We do not want T_S to be required to retraverse after the merging if the merging did not involve a split. So in this case (no split) we would like T_S to keep its S-latch on the parent P. Later T_S can re-latch L without retraversal and the search key k is still in the key range of L.

On the other hand, if a split is required, T_S must release its S-latch on P, so that the first requestor can X-latch P and do the split for merging. Since T_S does not access the second-index, it has no direct information about whether or not a split is required. This is the purpose of the call $notifySplit(L)$, by which the first requestor

indirectly notifies all subsequent requestors that a split is required.

The complete behavior of *mergeEvents* is as follows:

- For the first request to set a mergeEvent on page *L*, the call *set*(*L*, *latchType*) returns **1**. If the latchType on *L* held by this requestor is *S*, the S-latch is released and an X-latch is requested. The call does not return until the X-latch is granted (after other requestors (if any) have released latches on *L*).

- For any subsequent request to set the same mergeEvent, this requestor will be blocked. But before being blocked, two actions take place: (1) the latch on *L* is released so that the first requestor can obtain its X-latch, and (2) if there has already been a notification that a split is required for the merging, the S-latch on *P* is released.

- When the first requestor finds that a split is required, it calls *notifySplit*(*L*), which causes other requestors being blocked on the same mergeEvent to be unblocked, to release their S-latches on *P*, and then to be blocked again (entering a second waiting queue of this mergeEvent). This also provides a signal to subsequent requestors to release their S-latches on *P*.

- When the first requestor finishes the merging, it resets the mergeEvent, which causes other requestors being blocked on the same mergeEvent to be unblocked, with **0** returned to the call *set*(*L*, *latchType*) if there was no split, or **-1** returned if there was a split.

The lazy merging algorithm with mergeEvents used by a B-tree top-down access *T* is described as follows, with only step 2 and step 3 different from the basic algorithm.

Lazy merging with mergeEvents

2. *T* checks the LSN of *L*, *leafLSN*:

    (a) If leafLSN > reorgLSN, *T* releases the S-latch on *P* and proceeds to step 6.
    (b) If leafLSN ≤ reorgLSN, *T requests* to set a *mergeEvent* on *L*. If the request returns 1, *T* proceeds to step 3. If the request returns −1, *T* proceeds to step 4. Otherwise (returning 0), *T* re-requests X-latch (for update) or S-latch (for search) on *L*, then releases the S-latch on *P* and proceeds to step 6.

3. (Only the first requestor for the mergeEvent reaches this step.) *T* releases the S-latch on *P* and accesses the second-index in the key range *R* to decide the space required to hold entries to be moved in, and attempts to merge:

    (a) If *L* does not need to be split to hold the matched entries to be moved in, *T* triggers the merging as a system transaction. After the merging is done, *T resets* the *mergeEvent* on *L* and proceeds to step 5.
    (b) Otherwise (split case), *T notifies* the *mergeEvent* that split is required, releases the X-latch on *L* and all latches on the second-index, then follows the safe-node algorithm to retraverse looking for *k*, with all pages to be updated by this merging X-latched. Then *T* triggers the merging as a system transaction. After the merging is done, *T resets* the *mergeEvent* on *L*, releases all X-latches on the main-index and proceeds to step 4.
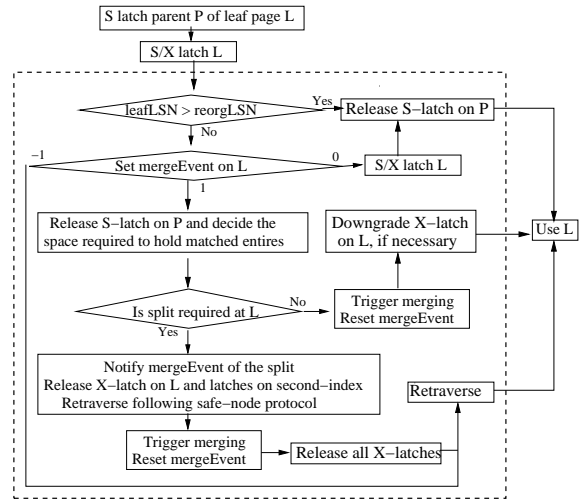

Figure 4: Lazy merging with mergeEvents.

Figure 4 illustrates the logic. It is unnecessary to check the leafLSN against the reorgLSN after *L* is X-latched again in step 3(b), because only the first requestor to set the mergeEvent will proceed to step 3 and trigger the merging, and the leafLSN does not change before *L* is re-latched again during the retraversal in step 3(b).

3.5 Range Scans with Lazy Merging

So far we have presented the two lazy merging algorithms with a B-tree top-down access. Another type of B-tree access is a range scan, which involves left-to-right access following forward side pointers at the leaf level.

Normally a range scan consists of two parts: first searching top-down from the root to the left-most leaf page in the search key range, then possibly scanning left-to-right at the leaf level. For the first part, when the left most leaf page *L* in the search key range is reached, the key range for *L* is known. Thus *L* can be merged using the lazy merging algorithms with a B-tree top-down access.

For the second part, when a leaf page L_S reached by a forward side pointer is not merged yet (by checking the leafLSN), we have to merge it first, but we cannot merge L_S if we do not know its key range.

One way to obtain the key range for L_S is to retraverse top-down from the root. To ensure to reach L_S (or if L_S is deleted by other user transactions, the next leaf in the search range) during retraversal, we utilize the key range *R* of the left sibling of L_S. Normally, leaf key ranges are half-open intervals. If *R* is of form $[k1, k2)$, we choose $k2$ as the search key during retraversal. Otherwise (*R* is of form $(k1, k2]$), we search for the smallest value which is greater than $k2$ (where duplicate keys are allowed, we make them unique by concatenating the B-tree keys with disk addresses (RIDs) or in a primary key (not RID) based system, concatenating the B-tree (secondary) keys with the primary keys [14]).

On the other hand, if each leaf page contains information about their own key range, e. g., if it contains the value of the *highest key* allowed in the leaf, we do know the range of L_S when we visit it via a forward side pointer. In this case, the left sibling of L_S plays the role of its "parent" in the lazy merging algorithm. Although

holding a latch on the left sibling of a leaf page does not prevent this leaf page from being split by another user transaction, this does not matter for range searches.

In addition, *backward range scans* are allowed to some degree in many realistic systems, where backward side pointers are implemented at the leaf level. To avoid deadlock with forward range scans, a backward scan does not use latch-coupling, instead, the whole tree is locked. Accordingly, the key range for a leaf page can be obtained easily by accessing its ancestor(s).

When a leaf page is split, the backward side pointer of its original right sibling should be updated to point to the new allocated page. In order not to complicate the presentation, we present only a simple solution: that backward side pointer is not updated immediately, instead, since we always have the correct forward side pointers from which all backward side pointers can be derived, that backward side pointer will be fixed later when a backward range scan occurs.

3.6 Deadlock-Freeness

Our concurrency control for merging uses latches and mergeEvents. The latching order follows top-down within an index and left-to-right at the leaf level. All latches on the main-index are obtained before latches on the second-index. Such resource ordering ensures that no deadlock is introduced by these latches.

If one B-tree access is to be blocked while requesting to set a mergeEvent (waiting for the completion of merging a main-index leaf page), before being blocked, all latches held by it which will conflict with required latches of the first requestor (the merger) are released. If two requestors are blocked (neither of them is the merger), the latches (if any) held by them do not conflict with each other (they are S-latches). So no deadlock is introduced by mergeEvents.

User transactions may of course deadlock on other database items, but not with latches or mergeEvents *used for merging*. *After merging*, the user thread may hold a latch on the merged leaf page. As is well known [15], this latch can cause deadlock if the user thread holds it while waiting for a database lock. In this case, the latch is released.

3.7 Lazy Merging of B-Link Trees

An important structural difference of B-link trees [10] from ordinary B-trees is that B-link trees have forward side pointers even at internal levels. Concurrency and recovery of B-link trees including node consolidations and deletions were studied in [11, 12, 13]. B-link trees were shown to have better concurrency than B-trees in [8, 20].

We assume that in B-link trees each page has an additional field called the **highest key**. (To keep the advantage of B-link trees, each page should contain information about its own key range *anyway*.) For a leaf page, this field indicates the highest key this page can hold. For an internal page, this field indicates the highest key the subtree rooted at this page can hold. Accordingly, the key range of each page is a half-open interval of form $(k1, k2]$.

Whenever a split occurs, we post to the parent the separation key for the split page. A natural choice of the separation key is the field *highest key* of this split page.

Any page in a B-link tree except the root can be reached either via a child pointer or via a forward side pointer. When a page is reached via a child pointer, we can know its key range from the information in the parent. When a page is reached via a forward side pointer, its key range is $(k1, k2]$, where $k2$ is its highest key, and $k1$ is the highest key of its left sibling.

The lazy merging algorithm used by a B-link tree top-down or left-to-right access T is described below, which is simpler and more efficient than B-tree merging.

Lazy merging of B-link trees

1. While holding a latch on the previous page P (either the parent or the left sibling of the leaf L), T acquires a latch on L (S-latch for search or X-latch for update). At this point, the key range R for L is known.
2. T checks the LSN of L, $leafLSN$:
   (a) If leafLSN > reorgLSN, no merge is needed. So T releases the latch on P and proceeds to step 4.
   (b) Otherwise (leafLSN $\leq$ reorgLSN), if the latch T has on L is an X-latch, T proceeds to step 3; otherwise, T releases the S-latch on L and re-acquires an X-latch on L, then repeats step 2. (The latch on P is still held, so that L can be re-latched directly.)
3. At this point T has an X-latch on L. T releases the latch on P and triggers the merging system transaction to merge the matched entries within the key range R from the second-index into the leaf level of the main-index. If L was split, posting of the split is scheduled in separate system transactions.
4. If T holds an X-latch on L and T's original request was an S-latch, T downgrades the X-latch to an S-latch. Then T accesses page L, which is up to date at this point.

It is unnecessary to immediately post a split boundary to the parent after the split takes place, because the forward side pointer allows a newly allocated page to be accessible from its left sibling. Accordingly it is not required to X-latch pages to be updated at upper levels at once before the actual merging (different from B-tree merging). It is also unnecessary to access the second-index before the actual merging to decide how much space is required to hold the matched entries and which upper levels should be updated. All matched entries are merged into the leaf level immediately at step 3. This results in shorter duration of X-latches at upper levels and thus higher concurrency.

With this lazy merging algorithm for B-link trees, only the first B-link tree access which finds that a main-index leaf page has not been merged and has X-latched this leaf page will attempt to access the second-index and do the merging, even if there will be a split. Other accesses which find that this leaf page has not been merged will wait on X-latching this leaf page and find later that this leaf page has already been merged. So unnecessary attempts are prevented. MergeEvents as used in the B-tree merging algorithm for the split case are unnecessary for the merging of B-link trees.

4. CLEANUP AND RECOVERY

In this section we describe how to merge entries into leaves which are never visited by user transactions to speed up completion of the merging and how to modify recovery to handle merging.

4.1 Background Cleanup Thread

One potential problem with lazy merging is that some main-index leaf pages may never be merged since they may never be accessed by user transactions, thus the second-index may exist forever. It is desirable to finish the merging process in a reasonable amount of time and bring the system back to its normal operating state.

To solve this problem, we use a special *background cleanup thread* which will trigger merging. It first looks up the *smallest key* in the second-index, then searches the main-index for this key. This will trigger the lazy merging of the main-index leaf page whose key range contains this key. (It releases latches on the second-index after reading the smallest key, to avoid deadlock with user threads which are issuing lazy merging system transactions.)

Normally the background cleanup thread will become active when the system is not busy, so that even if the merging triggered by the background thread causes disk I/Os, it does not affect performance experienced by user transactions.

When the background cleanup thread is used and keys are short (thus key comparison is not expensive), a further optimization is possible. The smallest key value in the second-index is regarded as the lower bound of the merging. Then when a main-index leaf page is found not merged yet, if the upper bound of its key range is smaller than this smallest key value, this page does not need to be considered by the lazy merging. This smallest key value is maintained as a system variable and updated by the background cleanup thread.

4.2 Recovery

Our algorithms require only a few mechanisms other than normal logging to ensure correct recovery. Changes to the main-index and the second-index during merging are in system transactions independent from user transactions which trigger the merging. Even if the triggering user transactions fail or the system crashes, these system transactions will complete eventually. In particular, any interrupted merging system transaction should be completed before the system resumes because otherwise the leafLSN of the leaf pages may be changed without the merging taking place. Thus merging requires *forward recovery* (this is different from ordinary system transactions required by tree structural changes such as splits).

After the system comes back from a crash, as is usual, first all actions are redone. Then as in [12, 15], any *finished* merging system transaction is not undone, even if the enclosing database transaction is unfinished and thus undone. This is because merging is an independent top-level system transaction. But we must go one step further. Any *incomplete* merging must be completed, because it may have already changed the leafLSN of the page to be merged. Thus the *begin-merge* log record (for the beginning of the merging system transaction) contains a key from the leaf page in the main-index to be merged. If the merge was not complete at the time of the crash, this leaf page would not have been updated or split by any other transaction because it would have been latched by the thread of the merging transaction at the time of the crash. After gathering all such keys, the recovery thread must merge these pages before new user transactions start. To make this simple, we assume that the incomplete system transaction is first undone as is usual [12, 15] (normally undo changes page LSNs), and then at the end of recovery, it is re-triggered using the key in the begin-merge log record. This avoids having to determine what part of the merging has completed before the crash.

Table 2: System Configuration.

CPU number	1
CPU service discipline	processor sharing
CPU speed	1GHz
disk number	1
disk service discipline	first come, first served
disk seek time	average 4ms
disk rotation delay	average 2ms
disk page size	4KB
disk transfer speed	75MBPS
buffer replacement policy	LRU

The re-triggered merging will not test the leafLSN against the reorgLSN, because it may have been modified and already be greater than the reorgLSN. The mergeEvents are unnecessary, because there is only one thread (the recovery thread) accessing the leaf page during recovery.

Since the deletion from the second-index and the insertion of those entries into the main-index are enclosed in forward recoverable system transactions, after the whole system recovers, the second-index will contain exactly those entries which had not yet been triggered to be merged at the time of the system failure.

5. PERFORMANCE ANALYSIS AND MEASUREMENTS

In this section, we present performance analysis and experimental results for the lazy merging approach. We will evaluate the effect of mergeEvents, analyze and measure I/O savings (compared with the *eager approach*), measure the degradation of the throughput and the response due to lazy merging (including comparison with the *off-line approach*), and finally measure the merging speedup by the background cleanup thread. The experimental setting is given first.

5.1 Experimental Setting

We implemented the top-down B-tree access with lazy merging on top of the CSIM-19 [19] simulator engine. This implementation also verifies correctness of the lazy B-tree merging algorithms. The configuration of the experimental system is listed in Table 2.

The fan out of both the main-index and the second-index is 100. Each leaf page in both indexes can hold 100 entries. The main-index is populated with 400K distinct odd integer keys whose values are uniformly chosen from 1 to 1800K. Each page of the main-index is 80% full. The second-index is populated with distinct even keys uniformly chosen from the same key range. Each page of the second-index is 100% full. For the **low-split** case, the second-index has 40K keys (10% of the main-index). For the **high-split** case, the second-index has 120K keys (30% of the main-index). After merging, the height of the main-index will keep at 3 for the low-split case, while for the high-split case, the height of the main-index will become 4. The size of the buffer is set to 10 times the number of all upper level pages of both indexes. In this way, throughout the merging process *almost all upper level pages and around 10% of all leaf pages could reside in buffer*.

Each top-down B-tree access issued by a user thread is counted as a **user transaction**. Each user transaction either modifies (50% possibility) or searches a merged leaf page. But the modification requested by user transactions does not change the structure of the main-index. We coded the experiments this way because user-

Table 3: Time Cost of Basic Operations.

hash access to locate a page in buffer or a mergeEvent	400
latch a page already in buffer without waiting	100
unlatch a page	100
find an empty slot in the buffer	200
find the page to be replaced by LRU	1,000
set a mergeEvent without waiting	200
reset a mergeEvent	200
create a mergeEvent	150
destroy a mergeEvent	100
notify a mergeEvent of a split	200
binary search in a page (about 100 entries per page)	1,000
calculate number of required new pages for a merge	250
modify a page including logging	2,000
copy a page	2,000
compare two index entries	150
disk seek	4,000,000
disk rotation	2,000,000
disk page transfer	53,000

Figure 5: CPU Time to finish merging 90%.

transaction-caused structural changes are not relevant to our study. All user threads run at the same priority and continuously access keys randomly chosen from 1 to 1800K.

The smallest time unit in the simulation is one *instruction time* (one second = 1G time units). The time cost of basic operations are listed in Table 3 (partly based on [20] and communication with commercial DBMS implementors). A mergeEvent is not a database lock and it can be implemented by two semaphores. Setting a mergeEvent without waiting costs about twice as much as latching a page already in buffer without waiting. The cost of merging a main-index leaf page depends on the number of second-index entries merged into the key range of this page. If the number of entries is small, they can be inserted one by one, and the cost for merging a page (the upper bound) is, $C_1 = NumOfEntries \cdot (cost_{binarySearch} + cost_{modifyPage})$. If, on the other hand, the number of entries is large enough, it is more efficient to simply create a copy of the page, do a sorted merge with the second-index entries, and write back the result. In this case the cost for merging a page (the upper bound) is, $C_2 = cost_{copyPage} + NumOfResultedPages \cdot (fan\_out \cdot cost_{entry-comparison} + cost_{copyPage})$. We use $minimum(C_1, C_2)$ as the merging cost.

5.2 Effect of MergeEvents

To see the effect of mergeEvents, we compared the lazy merging algorithm with mergeEvents to the basic lazy merging algorithm. Each user thread continuously runs until 90% of the entries in the second-index have been merged. (After 90%, few user transactions trigger merging.) We vary the number of concurrent user threads and compare the times required to finish. In this set of experiments, I/O cost is not included. It is reasonable to exclude I/O cost, because the two algorithms under comparison incur almost identical I/O cost due to the LRU buffer replacement policy. The same experiment is done under two settings: one is the combination of high-split and high-contention, and the other is the combination of low-split and low-contention. To *construct* a scenario of **high-contention**, all user threads access the same sequence of random keys at the same pace, while for the **low-contention** case, each user thread accesses a different sequence of random keys.

Figure 5 shows the result. For both settings, around 12K random keys should be accessed to finish merging 90% of second-index entries. For the setting of high-split and high-contention, because
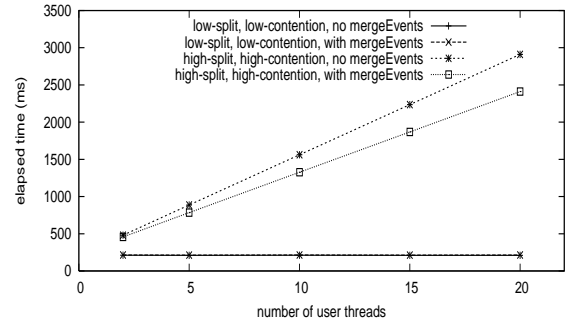
all threads access the same sequence of random keys, the time required to finish becomes longer as the number of threads increases. When the number of threads is greater than 1, the algorithm with mergeEvents takes less time than the basic algorithm, because in the algorithm with mergeEvents, only one user transaction will try to merge the same main-index leaf page, while in the basic algorithm, more than one user transaction will access the second-index and try to merge the same main-index leaf page. As the number of threads increases, the performance advantage becomes more obvious. When there are 20 threads, there is around 17% advantage.

For the setting of low-split and low-contention, the time required to finish keeps almost constant as the number of concurrent user threads increases, because each user thread accesses a different sequence of random keys, and 90% of the second-index will always be merged after around 12K user transactions have been issued no matter how many concurrent user threads are running. The algorithm with mergeEvents takes slightly more time to finish (invisible in Figure 5) because of the extra overhead of mergeEvents, regardless of the number of the user threads.

Although the algorithm with mergeEvents wins only when there is high-split and high-contention, in practice, it can be always adopted because the overhead of mergeEvents is minor. In later experiments, the algorithm with mergeEvents is always used and I/O cost is included.

5.3 I/O Saving

To analyze I/O saving due to lazy merging, we first make a probabilistic analysis for the number of merges, that is, the number of main-index leaves which require merging. Based on this, we predict the number of *no-op* log records (for leaves not needing merging) and estimate the number of I/Os we can save. Secondly, we present results from experiments which verify our analysis. Finally, we show the I/O saving improvement by the background cleanup thread.

5.3.1 Analysis for I/O Saving

Table 4 lists the parameters used in the analysis. We assume the keys in both the main-index and the second-index are uniformly distributed and we consider neither upper level index pages, nor the effect of the background cleanup thread.

To compute the number of merges, let us consider an arbitrary main-index leaf page, say p. This page will be accessed if it happens that a second-index key falls in the range of p. An arbitrary

Table 4: Parameters Used in Analysis.

m	number of leaf pages in main-index
s	number of leaf pages in second-index
n	number of key entries in second-index

Table 5: I/O Counts. *L* and *H* stand for low-split and high-split, respectively. *MI* and *SI* stand for main-index and second-index, respectively.

	lazy, L	eager, L	lazy, H	eager, H
read, MI	10,692	14,678	11,896	15,377
write, MI	7,699	9,770	12,063	13,474
total, MI	18,391	24,448	23,959	28,851
read, SI	3,865	2,295	8,573	6,097
write, SI	2,651	1,360	4,630	2,836
total, SI	6,516	3,655	13,203	8,933
total, MI+SI	24,907	28,103	37,162	37,784
allocate, MI	2	0	3,552	3,556
deallocate, SI	91	363	683	1,090
no-op log	0	0	0	0

key falls in the range of p with probability $1/m$ since there are m leaf pages and the keys are uniformly distributed. After we have examined n keys, the probability that at least one key falls in the range of p is $1 - (1 - \frac{1}{m})^n \approx 1 - e^{-n/m}$. The approximation is valid for large m. Since there are m main-index leaf pages, the total number of leaf pages that have at least one key falling in their ranges is approximately $m(1 - e^{-n/m})$.

This is the approximate number of merges. Intuitively, if $n \gg m$, then $m(1 - e^{-n/m}) \approx m$ and almost every leaf page of the main-index will have new keys falling in its range. On the other hand, if $n \ll m$, then $e^{-n/m} \approx 1 - \frac{n}{m}$ and $m(1 - e^{-n/m}) \approx n$.

Generally, as long as n is not too small, almost every leaf page of the main-index must be merged. Thus there are hardly any *no-op* log records. Each merge may involve two random I/Os (one read and one subsequent write) on a main-index leaf page. For the lazy merging approach, all such reads can be piggybacked in user read requests. Further if a user request is a write, the write for merging can also be piggybacked in the user write request. Thus compared to any other non-piggybacking approach, the number of I/Os on the main-index saved by the lazy approach is between m and $2 \cdot m$ (when all user requests are read-only, it is m; and when all user requests are writes, it is $2 \cdot m$).

On average, a leaf page of the second-index is merged with at most $\lceil m/s \rceil$ leaf pages of the main-index. The lazy approach merges one leaf page of the main-index at a time, thus each leaf page of the second-index will be accessed for $\lceil m/s \rceil$ times. Further since user transactions access leaf pages of the main-index randomly, the second-index is left sparse during the merging process. Accordingly, if the second-index is relatively large (compared to the main-index) and the buffer space is very tight for merging (due to the current system load), each access to a leaf page of the second-index may involve two random I/Os (one read and one subsequent write). Based on this analysis, in the worst case the lazy approach incurs around $2 \cdot s \cdot (\lceil m/s \rceil)$ I/Os on the second-index.

Any approach requires at least $2 \cdot s$ I/Os on the second-index (the theoretically minimal case). Thus the **I/O loss** on the second-index for the lazy approach (the extra I/Os needed beyond the minimal case) is $2 \cdot s \cdot (\lceil m/s \rceil - 1)$ in the worst case.

Combining the I/O saving on the main-index and the I/O loss on the second-index, the **total I/O saving** of the lazy approach (I/O saving on the main-index $-$ I/O loss on the second-index) in the worst case is $m - 2 \cdot s \cdot (\lceil m/s \rceil - 1)$ (which equals $2 \cdot s - m$ when m mod $s = 0$ and becomes negative further when $m > 2 \cdot s$). This might occur when all user transactions are read-only, the second-index is relatively very large and high-split is involved for merging. The total I/O saving of the lazy approach in the best case is $2 \cdot m$. This occurs when all user transactions are write, and the second-index is small and no splits are involved for merging so that the number of I/Os on the second-index approaches the minimal case.

However, user transactions usually involve read and/or write. When

half of the user transactions are write, the I/O saving on the main-index is $1.5 \cdot m$. And for most application scenarios (data migration and data batch insertion, etc.) where *the main-index is merged with a much smaller second-index* and thus normally the merging does not involve high-rate splits, the leaf pages of the second-index to be accessed can be found in buffer frequently. Thus the number of I/Os on the second-index is much smaller than $2 \cdot s \cdot (\lceil m/s \rceil)$. In this case the total I/O saving approaches $1.5 \cdot m$.

5.3.2 I/O Saving Measurement

To verify our analysis and provide more insight into practical settings, we carried out a number of experiments. We implemented an approach for merging without piggybacking, a very *simplified eager approach*, in which a merging thread *continuously* reads the smallest key from the second-index and then looks up that key in the main-index. Such a lookup will trigger the merging of the main-index leaf page whose key range contains that key. The user thread in the eager approach will *simply* access a random key in both indexes without merging, first the main-index, then the second-index. Each access is either search (50% possibility) or modification without structural changes. When the key to be accessed is smaller than the current smallest key in the second-index (maintained as a system variable and updated by the merging thread), the user thread will skip accessing the second-index.

Without loss of generality, in both the lazy and the eager approaches, there is only one user thread. In the lazy approach, there is no background cleanup thread and the user thread stops after 90% of second-index entries have been merged. In the eager approach, the user thread stops after the same number of user transactions as in the lazy approach have been issued, and the merging thread stops after 90% of second-index entries have been merged.

We measured the numbers of read and write I/Os, new allocated pages and/or deallocated pages for both the main-index and the second-index. The same experiment is done for both the low-split case and the high-split case. For either case, around 12K user transactions have been issued after 90% are merged. Table 5 lists the resulting I/O counts.

The number of leaf pages in the main-index is 5K. In the low-split case, the number of newly allocated pages in the main-index is around 2, while in the high-split case, the number of newly allocated pages in the main-index is around 3.5K. For either case, the exact numbers of newly allocated pages in the main-index for the lazy approach and for the eager approach are not equal, because the

Table 6: I/O Savings with Different Second-index Sizes. The percentage indicates the entry number in the second-index relative to that in the main-index.

percentage	5%	10%	30%	50%	80%	100%
I/O saving	5,328	3,196	622	247	33	13

Table 7: I/O Saving Improvement with Increase of Background Thread Activity (1 per *x* User Transactions). The number of entries in the second-index is 30% of that in the main-index. *Deallocate* means the number of pages deallocated in the second-index for the lazy approach.

x	infinite	100	50	20	10	0
deallocate	683	698	707	748	789	944
I/O saving	622	700	837	954	1,121	1,385

two approaches may have merged *different leaf pages* of the main-index and *different sets* of the entries in the second-index, and the last page merged may increase the proportion beyond 90%.

In the low-split case, for the lazy approach compared to the eager approach, the I/O saving on the main-index is 6057 ($24448 - 18391$) and the I/O loss on the second-index is 2861 ($6516 - 3655$), so the total I/O saving is 3196 ($6057 - 2861$). Notice that the LRU buffer replacement policy introduces some randomness in the results and that write I/Os are also saved sometimes by being piggybacked.

In the eager approach, as merging is done in key order, the second-index shrinks rapidly and once some entries in a leaf page of the second-index have been merged, the subsequent merges for the rest entries in that leaf page will find that leaf page in buffer with a large probability (**access locality**). In contrast, in the lazy approach, since we deallocate leaves of the second-index only-on-empty as usual and since the merging into the main-index is by random key ranges, the second-index becomes sparser and occupies more pages than in the eager approach. This is verified by the following data: after 90% of second-index entries have been merged, the number of deallocated pages in the second-index in the eager approach is 363, while the number in the lazy approach is only 91. The sparseness of the second-index and the relative non-locality of access on the second-index account for the I/O loss on the second-index for the lazy approach. However since the second-index is relatively small, the I/O loss is small.

In the high-split case, the second-index is relatively larger and many main-index leaf pages are split. The sparseness of the larger second-index and the newly allocated pages of the main-index cause the buffer space for merging to be tight. Thus the I/O loss on the second-index becomes larger ($4270 = 13203 - 8933$), and the I/O saving is much smaller ($622 = 37784 - 37162$) than the low-split case.

In addition, in all cases, the number of *no-op* log records is 0, which means all main-index leaf pages have at least one entry to be merged from the second-index. This verifies that the number of no-op log records is expected to be small.

We further measured the I/O savings when the second-index is even smaller or larger. Table 6 lists all the results (including previous results). When the second-index is much smaller than the main-index and the merging involves low-split (which occurs when the percentage is less than 20%, due to 80% fullness of the main-index), the I/O saving is large and along with the decease of the second-index size, the I/O saving becomes larger and approaches the theoretical maximum 6750 ($= 1.5 \cdot 5K \cdot 90\%$). This is the most likely setting for most applications. On the other hand, along with the increase of the second-index size, the I/O saving becomes smaller, especially when the merging involves high-split (when the percentage is greater than 20%). However, even with a relatively large second-index involving high-split to merge, the lazy approach incurs no more I/Os than the eager approach.

5.3.3 I/O Saving Improvement

The above results suggest that a benefit of the background cleanup thread in the lazy approach is to shrink the second-index and reduce the sparseness caused by random user-transaction-induced merges. This helps reduce the number of I/Os on the second-index. When the second-index is relatively large, the background cleanup thread can be made more active to shrink the second-index. In addition, when the background thread is active enough, its access on the second-index will present *locality* (finding second-index leaf pages in buffer frequently).

To demonstrate this, we make the frequency of the background thread activity to be one background transaction every *x* user transactions, where *x* changes from 0 to 100. When *x* is 0, it means the background thread continuously queries the smallest key of the second-index and triggers merging. This is the same as the merging thread in the eager approach. Both threads (1 user thread and the background cleanup thread) stop when 90% of second-index entries are merged. The counterpart in the eager approach issues the same number of user transactions as in the lazy approach.

Table 7 lists the total I/O savings for a case when the I/O loss on the second-index is large (thus the total I/O saving is small). When $x = infinite$, it means there is no background cleanup thread. As the frequency becomes higher, more pages in the second-index are deallocated (for comparison, the number of deallocated pages in the second-index for the eager approach is always 1090), and the I/O saving becomes larger. However the background thread should not be made too active or eager in practice, otherwise the throughput of concurrent user transactions will degrade.

5.4 Throughput and Response Degradation due to Lazy Merging

To see how much slowdown is incurred by concurrent user transactions due to lazy merging, we measured the throughput and the response time of user transactions during lazy merging where user transactions access a sequence of random keys on the main-index. This way is actually very "eager", because *most* user transactions will trigger merges at the early stage of merging. We first worked with one user thread (there is no background thread) and recorded the response time of each user transaction and throughput (the number of user transactions per second) from the beginning of lazy merging until all second-index entries are merged.

The same experiment is done for both the high-split case and the low-split case 20 times, with a different seed for the random key generator each time. For each run, the response time is taken as the average over every 600 user transactions and the throughput is taken as the average over every 10 seconds. Figures 6 and 7 show the results of one run for both cases. For either case, all 20 runs
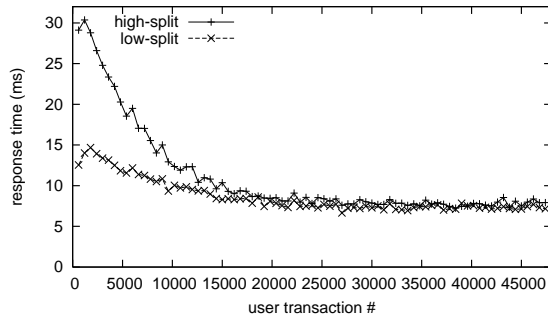
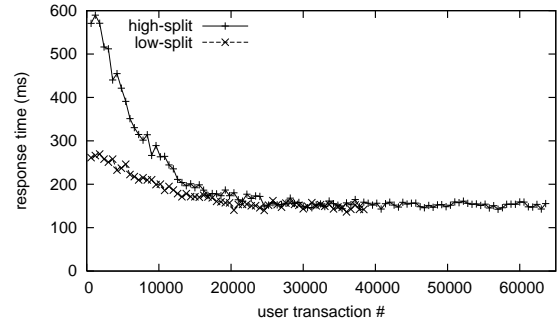**Figure 6: Response time during lazy merging (1 thread).**

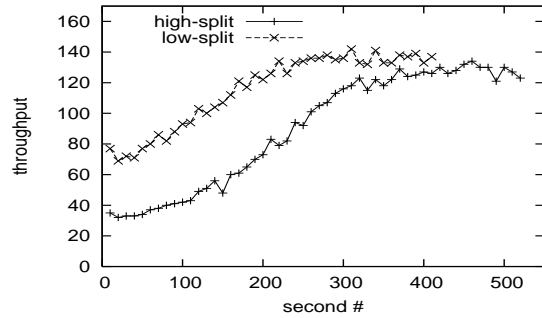**Figure 8: Response time during lazy merging (20 threads).**

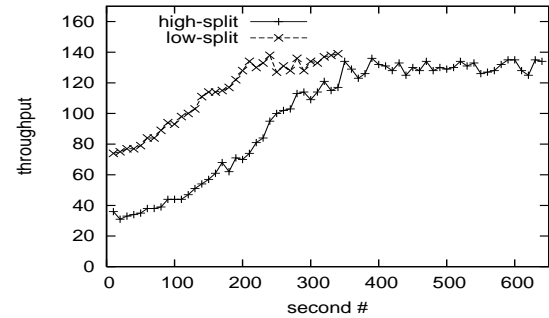**Figure 7: Throughput during lazy merging (1 thread).**

Figure 9: Throughput during lazy merging (20 threads).

have similar results.

The performance degradation is basically decided by the number of concurrent merges triggered. For either case, in the beginning of lazy merging, *many* user transactions triggered merging, so the response time is large and the throughput is low. As the merging progresses, *fewer* user transactions triggered merging, so the response time decreases and the throughput increases, especially quickly for the high-split case. This also verifies that the merging is incrementally beneficial. When around 25K user transactions are issued, 99% of second-index entries have been merged and the response time begins to become stable. The low-split case takes less time to finish, its throughput is higher and its response is quicker than the high-split case.

In addition, for both cases, we measured the time to finish merging by an *off-line approach*, where a thread eagerly merges entries from the second-index to the main-index, similar to the merging thread in the eager approach. (Sequential I/Os cannot be exploited, since the leaf pages of the main-index are not stored in order.)

Table 8 list performance values. All values for the lazy merging are averages taken over 20 runs. For the high-split case, the response time at the beginning (when almost every user transaction triggers a merge) is *less than four times* of that in the end (when hardly any user transaction triggers a merge), and for the low-split case, it is *less than two times*. These values are explained by the I/Os involved for merging and/or splits. Similarly, the throughput at the beginning is *more than one fourth* of that in the end for the high-split case, and *more than one half* for the low-split case. This shows that the amount of splitting has a large impact on the performance.

In the off-line approach, for either case, the total time (D_{off}) is less than the total time in the lazy approach (D_{laz}). However, all concurrent user transactions must wait for the entire merging time (D_{off}). This means for the high-split case that the response time for those user transactions is more than 115s, as opposed to 29.2ms, which is the response time of the worst case in the lazy approach.

We further measured the response time and the throughput during the lazy merging when there are 20 concurrent user threads. Each thread has a different seed for its random key generator. When there are multiple user threads, as shown in Figure 9, the throughput is basically the same as in the single thread case. However, the response time is around the number of threads times the response time of the single thread case, as shown in Figure 8. This is because all threads compete for the single disk and the single CPU concurrently and the response time is bottlenecked by disk I/Os.

Table 8: Performance Values. D_{off} **is the duration (total time) of off-line merging.** D_{laz} **is the duration of lazy merging.** *RE* **stands for the response time.** *T* **stands for the throughput. The subscripts** *b*, *o* **and** *e* **stand for the beginning of lazy merging, the time (during lazy merging) when off-line merging would be ending, and the end of lazy merging, respectively.** *H* **and** *L* **stand for high-split and low-split, respectively.**

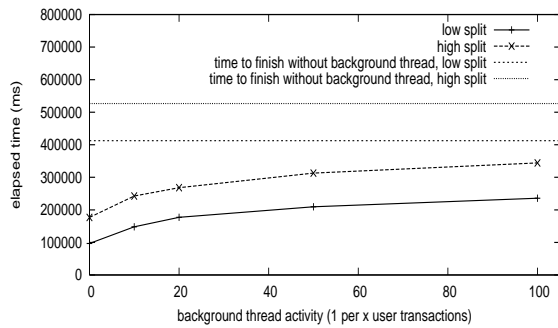|	D_{off}	D_{laz}	RE_b	RE_o	RE_e	T_b	T_o	T_e
H	115s	535s	29.2ms	20.7ms	7.7ms	36	47	129
L	61s	426s	13.7ms	12.0ms	7.3ms	73	84	137

Figure 10: Time to finish merging (1 user thread).

Notice that in Figure 8, for the high-split case, around 64K user transactions have to be issued before merging is finished, while only around 38K user transactions are required for the low-split case. This difference (26K) mainly lies in the difference of the numbers of user transactions that do not trigger merging after most merging has been done: After 90% merging has finished, only around 0.5K user transactions with merging are issued for either case, but the numbers of non-merging user transactions are different for the two cases. For the high-split case, around 51.5K non-merging user transactions are issued, while for the low-split case, only around 26.6K non-merging user transactions are issued. Correspondingly, the high-split case takes much longer time to finish as shown in Figure 9.

5.5 Merging Speedup by Background Cleanup Thread

To see how much speedup for merging is due to the background cleanup thread (which is usually activated when the system is not busy), we measured the times to finish merging by varying the frequency of the background thread activity (one background transaction every x user transactions). Without loss of generality, there is only one user thread. Both threads stop when all second-index entries are merged. We also obtained the time required when there is no background thread. In all runs, the user thread has the same seed for its random number generator. The same experiment is done for both the low-split case and the high-split case.

Figure 10 shows the results. When the frequency of background thread activity is the same, it takes more time to finish for the high-split case than for the low-split case. For either case, as the frequency becomes higher, it takes less time to finish. Even when the frequency is quite low such as one every 100 user transactions, the time required is already reduced by *more than one third* compared to the time required when there is no background thread.

6. CONCLUSIONS

We have presented three variations on lazy merging for trees with the same key range: the basic algorithm and the algorithm with mergeEvents for B-trees, and the algorithm for B-link trees. We have shown deadlock-freeness and we have discussed range searching, cleanup and recovery. Finally, we have analyzed and measured the performance of our lazy B-tree merging algorithms.

Our performance results show benefits in overhead reduction from mergeEvents, in I/O saving from lazy merging (especially for merging an index with another much smaller index), and in merging

speedup and in second-index shrinking (which affects I/O saving and performance) from the background cleanup thread. In addition, experiments showed that although running merges concurrently with user transactions (required for availability) slows down the system in the beginning, this effect is tolerable and transient.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] K. J. Achyutuni, E. Omiecinski, and S. B. Navathe. Two Techniques for On-Line Index Modification in Shared Nothing Parallel Databases. In *Proc ACM SIGMOD Conference*, pages 125–136, 1996.
[2] R. Bayer and M. Schkolnick. Concurrency of Operations on B-Trees. *Acta Inf.*, 9:1–21, 1977.
[3] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2), 1979.
[4] A. Gartner, A. Kemper, D. Kossmann, and B. Zeller. Efficient Bulk Deletes in Relational Databases. In *Proc IEEE ICDE Conference*, pages 183–192, 2001.
[5] G. Graefe. Sorting And Indexing With Partitioned B-Trees. In *Biennial Conference on Innovative Data System Research*, 2003.
[6] J. Gray and A. Reuter. *Transaction Processing: Techniques and Concepts*. Morgan Kaufmann, 1993.
[7] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proc VLDB Conference*, pages 16–25, 1997.
[8] T. Johnson and D. Shasha. A Framework for the Performance Analysis of Concurrent B-tree Algorithms. In *Proc Symposium on Principles of Database Systems*, pages 273–287, 1990.
[9] M.-L. Lee, M. Kitsuregawa, B. C. Ooi, K.-L. Tan, and A. Mondal. Towards Self-Tuning Data Placement in Parallel Database System. In *Proc ACM SIGMOD Conference*, pages 225–236, 2000.
[10] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.
[11] D. Lomet. Simple, Robust and Highly Concurrent B-trees with Node Deletion. In *Proc IEEE ICDE Conference*, pages 18–28, 2004.
[12] D. Lomet and B. Salzberg. Access Method Concurrency with Recovery. In *Proc ACM SIGMOD Conference*, pages 351–360, 1992.
[13] D. Lomet and B. Salzberg. Concurrency and Recovery for Index Trees. *VLDB Journal*, 6:224–240, 1997.
[14] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *Proc VLDB Conference*, pages 392–405, 1990.
[15] C. Mohan and F. Levine. ARIES/IM: an Efficient and High Concurrency Index Management Method Using Write-ahead Logging. In *Proc ACM SIGMOD Conference*, pages 371–380, 1992.
[16] C. Mohan and I. Narang. Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates. In *Proc ACM SIGMOD Conference*, pages 361–370, 1992.
[17] P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. Design, Implementation, and Performance of the LHAM Log-Structured History Data Access Method. In *Proc VLDB Conference*, pages 452–463, 1998.
[18] N. Ponnekanti and H. Kodavalla. Online Index Rebuild. In *Proc ACM SIGMOD Conference*, pages 529–538, 2000.
[19] H. Schmetman. CSIM19: A Powerful Tool for Building System Models. In *Proc Winter Simulation Conference*, 2001.
[20] V. Srinivasan and M. J. Carey. Performance of B$^+$ Tree Concurrency Algorithms. *VLDB Journal*, 2(4):361–406, 1993.
[21] C. Zou and B. Salzberg. On-line Reorganization of Sparsely-populated B$^+$ trees. In *Proc ACM SIGMOD Conference*, pages 115–124, 1996.
[22] C. Zou and B. Salzberg. Safely and Efficiently Updating References During On-line Reorganization. In *Proc VLDB Conference*, pages 512–522, 1998.