Recoverable B±trees in Centralized Database Management Systems

Ibrahim Jaluta[1], Seppo Sippu[2] and Eljas Soisalon-Soininen[1]

[1]Department of Computer Science and Engineering, Helsinki University of Technology, Konemiehentie 2, FIN-02015 HUT, Finland
ijaluta@cs.hut.fi
ess@cs.hut.fi

[2]Department of Computer Science, University of Helsinki, P. O. Box 26 (Teollisuuskatu 23), FIN 00014, University of Helsinki, Finland
sippu@cs.helsinki.fi

Abstract: The B±tree is the most widely used index structure in the current commercial database systems. This paper introduces new B±tree algorithms in which tree-structure modifications such as page splits or merges are executed as atomic actions. A B±tree structure modification, once executed to completion will never be undone no matter if the transaction that triggered such a structure modification commits or aborts later on. In restart recovery from a system crash, the redo pass of the recovery algorithm will always produce a structurally consistent B±tree, on which undo operations by backward-rolling transactions can be performed. A database transaction can contain any number of operations of the form "fetch the first (or next) matching record", "insert a record", or "delete a record", where database records are identified by their primary keys. Repeatable-read-level isolation for transactions is achieved by key-range locking.

Introduction

The B±tree is the most widely used index in the databases today. Each node (page) of the tree is a disk page (usually 4096 bytes). The leaves of the tree contain the database records. The leaves of the tree are all at the same level of the tree. Index pages (pages above the leaf level) contain only index records. The search for a single database record accesses exactly one node at each level of the tree. The B±tree clusters data in leaf pages in the order of the indexing attributes; it is therefore ideal for key-range searches. The B±tree is organized dynamically, and hence record inserts or deletes always leave the tree height balanced.

The B±tree structure modifications such as splits of overflown pages or merges of underflown pages present a challenge to the concurrency control and recovery in centralized Database management systems. In the literature there are two techniques designed to handle structure modifications such as page splits or page merges. In the first technique, a transaction X-latches the pages along the structure-modification path, and executes it before executing the insert/delete operation that triggered such a structure modification. In the second technique, a transaction acquires a tree latch before it executes a structure modification, triggered by an insert or a delete operation.

www.manaraa.com

In the technique presented in [1], a transaction X-latches the pages along the structure-modification path top-down, before it starts the execution of the structure modification bottom-up. Hence, two structure modifications can be executed concurrently only if they occur on completely distinct paths. Two structure modifications having at least one page in common on their paths will be serialized. X-latching the structure-modification path by an updating transaction T will prevent other concurrent transactions from reading or updating any of the pages on such a path. On the other hand, if the path were not X-latched at once by T, and if other transactions were allowed to update pages on the path, then T would wipe out updates of other transactions to these pages, when T aborts or the system fails.

This technique suffers from two problems. The first one is the reduction in concurrency due to the X-latching of all the pages on the structure-modification path at once. The second problem is that, if leaf-page updates and structure modifications are allowed to execute concurrently, then during restart recovery the B±tree could be structurally inconsistent and thus fail to perform logical undo operations [2], [3].

In the technique presented in [2], [3], [4], concurrent B±tree structure modifications are prevented through the use of a tree latch. That is, an updating transaction acquires a tree latch in X mode, before it starts the execution of a structure modification bottom up. Moreover, other transactions are prevented from performing any updating to leaf pages while the structure modification is still going on. This is because if transaction T1 updates a leaf page P and transaction T2 moves this update to another page P´ and commits while the structure modification is going on, then it is not possible to undo the update of transaction T1, if the system fails before the on-going structure modification is committed. During the undo pass of the restart recovery a page-oriented undo of the update by T1 would fail, and trying to undo the update logically would fail too, because the B±tree is not yet structurally consistent (also see [5]). When a transaction traversing the B±tree reaches an ambiguous situation (cannot decide which page to traverse next) while a structure modification is going on, then the traversing transaction must acquire a tree latch in S mode for short duration and retraverse the B±tree, so that the traversing transaction does not reach a wrong leaf page.

The pages along the structure-modification path are X-latched one page at a time bottom-up. When a page is modified during the structure modification, then the split-bit/delete-bit in that page is set. Hence, when an X latch on a modified page P is released by transaction T and the split-bit/delete-bit of P is set and the tree latch is still held by T, then no other transaction can update P. Therefore, uncommitted structure modifications can always be undone physiologically in a page-oriented fashion. The use of a tree latch serializes the structure modifications and guarantees a correct recovery.

This technique limits the degree of concurrency, because when a transaction T holds a tree latch in X mode to perform a structure modification, then new transactions will be prevented from accessing the B±tree, and at the same time other transactions are prevented from updating leaf pages.

If a transaction uses S latches to latch the pages along the split/merge path, then a deadlock may occur when two transactions try to upgrade their S latches on a common page in the path to X latches at the same time [6].

Traditionally, the space-utilization and balance conditions of the B±tree state that the pages are not allowed to fall below 50% full. When a full page P has been split, it has exactly that minimum space utilization. Now if a record is deleted from P, then P has to be merged back with its sibling page, since P fell below 50%. Also the page resulting from the merge can again be split after two consecutive inserts. Hence, the B±tree needlessly thrashes between splitting and merging the same page [7].

One approach to avoid the above problem is to allow pages to be emptied completely [1], [2] [4]. This issue was analyzed in [8], which concludes that page merging in a B±tree is not useful unless the pages are completely empty. That is, *free-at-empty* is much better (fewer

www.manaraa.com

access conflicts because of structure modification operations, and therefore higher concurrency on the B±tree) than *merge-at-half*. However, the free-at-empty approach does not guarantee (in the worst case) the logarithmic bound for the B±tree traversal algorithm; in addition, it leads to low disk utilization.

Our contributions are the following. We present new recoverable B±tree algorithms for centralized database management systems. In these algorithms, we introduce a new technique for handling B±tree structure modifications [9] that improves concurrency, simplifies recovery, avoids the problems which are associated with the merge-at-half and free-at-empty approaches, and reduces the amount of logging. Unlike in the algorithms in [1], [2], [3], [4], [10], [11], [12], we define a tree-structure modification (page split, page merge, record redistribution, increase-tree-height and decrease-tree-height) as a small atomic action. Each action affects only two levels of a B±tree. Each structure modification is logged using a single redo-only log record. Each successfully completed structure modification brings the B±tree into a structurally consistent and balanced state whenever the tree was structurally consistent and balanced initially. Record inserts and deletes on leaf pages of a B±tree are logged using physiological redo-undo log records as in [1], [2], [3], [4], [13]. Recoverability from system failures of both the tree structure and the logical database is guaranteed because the redo pass of our ARIES-based [14] recovery protocol [9] will now produce a structurally consistent and balanced tree, hence making possible the logical undo of record inserts and deletes.

Basic concepts

In a centralized database system, *locks* are typically used to guarantee the logical consistency of the database under a concurrent history of transactions, while *latches* are used to guarantee the physical consistency of a database page under a single operation. A latch is implemented by a low-level primitive (semaphore) that provides a cheap synchronization mechanism with S (shared), U (update), and X (exclusive) modes, but with no deadlock detection [1], [3], [13]. A latch operation typically involves fewer instructions than a lock operation, because the latch control information is always in virtual memory in a fixed place and directly addressable. On the other hand, storage for locks is dynamically managed and hence more instructions need to be executed to acquire and release locks and to detect deadlocks.

Lock requests may be made with the conditional or the unconditional option [2], [3], [4]. A *conditional* request means that the requestor (transaction) is not willing to wait if the lock cannot be granted immediately. An *unconditional* request means that the requestor is willing to wait until the lock is granted. Moreover, a lock may be held for different durations. An *instant-duration* lock is used to check for any current lock conflicts and to make the requestor wait if such a conflict is detected. However, no actual lock is retained on the data item. A *short-duration* lock is released after the completion of an operation and before the transaction that performed this operation commits. A *commit-duration* lock is released only at the time of termination of the transaction, i.e., after the commit or rollback is completed.

We assume that the buffer manager employs the steal and no-force buffering policy, because this policy does not put any restrictions on the buffer manager. It is very suitable during normal processing. To guarantee a correct recovery, the *write-ahead-logging* (*WAL*) *protocol* [1], [14] is applied. That is, an index or a data page with Page-LSN n my flushed onto the disk only after flushing first all log records with LSNs less than or equal to n. To facilitate the rollback of an aborted transaction, the log records of a transaction are chained via the Prev-LSN field of log records in reverse chronological order. The important fields that may be present in different types of log records are:

Transaction-id: The identifier of a transaction, if any, that wrote the log record.
Type: Indicates whether the record is “begin”, “abort”, “rollback-completed”, “commit”, “redo-undo”, “redo-only” or “compensation”.
Operation: Indicates the type of the update operation in the case of a redo-undo, redo-only or compensation log record.

www.manaraa.com

Page-id(s): The identifier(s) of the page(s) to which updates of this log record were applied; present in log records of types "redo-only", "redo-undo", and "compensation".
LSN: The log sequence number of the log record (monotonically increasing value).
Prev-LSN: The LSN of the preceding log record written by the same transaction in its forward-rolling phase.
Undo-Next-LSN: This field appears only in a compensation log record generated for a backward-rolling aborted transaction. The Undo-Next-LSN is the LSN of the log record for the next update to be undone by the backward-rolling transaction.
Data: Describes the update that was performed on the page.

A *redo-only* log record only contains redo information while a *redo-undo* log record contains both the redo and undo information. Therefore, every redo-only log record is *redoable*, and every redo-undo log record is both redoable and *undoable*. However, when the update logged using a redo-undo log record is undone, then the undo action is logged using a *compensation log record* (CLR). The Undo-Next-LSN of the generated CLR is set to the Prev-LSN of the log record being undone [1], [2], [3], [4]. A compensation log record is generated as a redo-only log record and hence an undo operation is never undone. Therefore, during the rollback of an aborted transaction the Undo-Next-LSN field of the most recently written CLR keeps track of the progress of the rollback, so that the rollback can proceed from the point where it left off should a crash occur during the rollback.

Each B±tree page contains a *Page-LSN* field, which is used to store the LSN of the log record for the latest update on the page. The LSN concept lets us avoid attempting to redo an operation when the operation's effect is already present in the page. It also lets us avoid attempting to undo an operation when the operation's effect is not present in the page. The recovery manager uses the Page-LSN field to keep track of the page's state.

The *transaction table* contains an entry for each active transaction. Each entry in the transaction table consists of four fields: (1) the *transaction-id*, (2) the *state* of the transaction (forward-rolling, backward-rolling), (3) the *Last-LSN* field which contains the LSN of the latest undoable non-CLR log record written by the transaction, and (4) the *Undo-Next-LSN* field which contains the Undo-Next-LSN of the latest CLR written by the transaction. The *modified-page table* is used to store information about modified pages in the buffer pool. Each entry in this table consists of two fields: the Page-id and *Rec-LSN* (recovery LSN). Both tables are updated during normal processing. When a transaction modifies a page P, then a new entry is created in the modified-page table, if there is no such entry for P. That is, the Page-id of P and the log-record address are inserted in the Page-id and Rec-LSN fields, respectively in the modified-page table. When a modified page is written to stable storage, then the corresponding entry in the modified-page table is removed. The value of Rec-LSN indicates from what point in the log there may be updates which possibly are not yet in the stable-storage version of the page. The minimum Rec-LSN value in the modified-page table determines the starting point for the redo pass during restart recovery.

To reduce the amount of recovery work that would be requested when the system crashes, *checkpoints* [Bern87, Moha92a, Gray93] are taken periodically during normal processing. When a checkpoint is taken, a *checkpoint log record* is generated which includes copies of the contents of the transaction and modified-page tables. Taking a checkpoint involves no flushing of pages.

Database and Transaction Model

We assume that our database D consists of *database records* of the form (v, x), where v is the *key value* of the record and x is the *data value* (the values of other attributes) of the record. The key values are unique, and there is a total order, ≤, among the key values. The least possible key value is denoted by $-\infty$ and the greatest possible key value is denoted by ∞. We assume that the database always contains the special record (∞, nil) which is never inserted or deleted. The database operations are as follows.

www.manaraa.com

1) ***Fetch*[v,θu, x]:** Fetch the first matching record (v, x). Given a key value u < ∞, find the least key value v and the associated data value x such that v satisfies vθu and the record (v, x) is in the database. Here θ is one of the comparison operators "≥" or ">". To simulate the fetch-next operation [2], [3], [4] on a key range [u, v], the fetch operation is used as follows. To fetch the first record in the key range, a transaction T issues Fetch[$v_1, \geq u, x_1$]. To fetch the second record in the key range, T issues Fetch[$v_2, > v_1, x_2$]. To fetch the third record in the key range, T issues Fetch[$v_3, > v_2, x_3$], and so on. The fetch operation Fetch[v,θu, x] is said to scan *the key range* [u, v] (if θ is "≥") or (u, v] (if θ is ">").

2) ***Insert*[v, x]:** Insert a new record (v, x). Given a key value v and a data value x, insert the record (v, x) into the database if v is not the key value of any record in the database. Otherwise, return with the exception "uniqueness violation".

3) ***Delete*[v, x]:** Delete the record with the key value v. Given a key value v, delete the record, (v, x), with key value v from the database if v appears in the database. If the database does not contain a record with key value v, then return with the exception "record not found".

In normal transaction processing, a database transaction can be in one of the following four states: forward-rolling, committed, backward-rolling, or rolled-back. A *forward-rolling transaction* is a string of the form $B\alpha$, where *B* denotes the *begin* operation and α is a string of fetch, insert and delete operations. A *committed transaction* is of the form $B\alpha C$, where $B\alpha$ is a forward-rolling transaction and *C* denotes the *commit* operation. An *aborted transaction* is one that contains the *abort* operation, *A*. A *backward-rolling transaction* is an aborted transaction of the form $B\alpha\beta A\beta^{-1}$, where $B\alpha\beta$ is a forward-rolling transaction and β^{-1} is the inverse of β (defined below). The string $\alpha\beta$ is called the *forward-rolling phase*, and the string β^{-1} the *backward-rolling phase*, of the transaction.

The *inverse* β^{-1} of an operation string β is defined inductively as follows. For the empty operation string, $\in$, the inverse $\in^{-1}$ is defined as $\in$. The inverse $(\beta o)^{-1}$ of a non-empty operation string βo, where o is a single operation, is defined as undo-o β^{-1}, where undo-o denotes the *inverse* of operation o. The inverses of our set of database operations are defined as follows.

1) ***Undo-fetch*[v, θu, x]** = $\in$.
2) ***Undo-insert*[v, x] = *Delete*[v, x]**.
3) ***Undo-delete*[v, x] = *Insert*[v, x]**.

A *history* for a set of database transactions is a string H in the shuffle of those transactions. Each transaction in H can be forward-rolling, committed, backward-rolling, or rolled-back. Each transaction in H can contain any number of fetch, insert and delete operations. The *shuffle* of two or more strings is the set of all strings that have the given strings as subsequences, and contain no other element. H is a *complete history* if all its transactions are committed or rolled-back.

For a forward-rolling transaction $B\alpha$, the string $A\alpha^{-1}R$ is the *completion string*, and $B\alpha A\alpha^{-1}R$ the *completed transaction*. For a backward-rolling transaction $B\alpha\beta A\beta^{-1}$, the string $\alpha^{-1}R$ is the *completion string*, and $B\alpha\beta A\beta^{-1}\alpha^{-1}R$ the *completed transaction*. A *completion string* γ for an incomplete history H is any string in the shuffle of the completion strings of all the active transactions in H; the complete history Hγ is a *completed history* for H.

B±trees

We use the B±tree [10] as a sparse database index to the database, so that the leaf pages store the database records. In addition, we assume the B±tree is a unique index and that the key values are of fixed length.

Formally, a B±tree is an array B[0,…,n] of *disk pages* B[P] indexed by unique *page identifiers* (*Page-ids*) P = 0,…,n. The page B[P] with Page-id P is called page P, for short. A

www.manaraa.com

page (other than page 0) is marked as an *allocated* if that page is currently part of the B±tree. Otherwise, it is marked as *unallocated*. Page M=0 is assumed to contain a *storage map* (a bit vector) that indicates which pages are allocated and which are unallocated. Page 1, the *root*, is always allocated. The allocated pages form a tree rooted at 1.

An allocated page is an index page or a data page. An *index page* P is a non-leaf page and it contains list of *index records* of the form $(v_1, P_1), (v_2, P_2), \ldots, (v_n, P_n)$ where $v_1, v_2, \ldots, v_n$ are key values, and $P_1, P_2, \ldots, P_n$ are page identifiers. Each index record (*child link*) is associated exactly with one child page. The key value v_i in the index record (v_i, P_i) is always greater than or equal to the highest key value in the page P_i. A *data page* P is a leaf page and contains a list of database records $(v_1, x_1), (v_2, x_2), \ldots, (v_n, x_n)$ where $v_1, v_2, \ldots, v_n$ are the key values of the database records and $x_1, x_2, \ldots, x_n$ denote the data values of the records. Each data page also stores its high-key record. The *high-key record* of a data page is of the form (high-key value, Page-link), where the *high-key value* is the highest key value that can appear in that data page and *Page-link* denotes the Page-id of the successor (right-sibling) leaf page. The high-key record in the last leaf page of the B±tree is (∞, nil). The set of database records in the data pages of a B±tree B is called the *database represented by* B and denoted by db(B).

We assume that each B±tree page can hold a maximum of $M_1 \geq 8$ database records (excluding the high-key record) and a maximum of $M_2 \geq 8$ index records. Let m_1, $2 \leq m_1 < M_1/2$, and m_2, $2 \leq m_2 < M_2/2$, be the chosen minimum load factors for a non-root leaf page and a non-root index page, respectively. We say that a B±tree page P is *underflown* if (1) P is a non-root leaf page and contains less than m_1 database records, or (2) P is a non-root index page and contains less than m_2 child links.

A B±tree is *structurally consistent* if it satisfies the basic definition of the B±tree, so that each page can be accessed from the root by following child links. A structurally consistent B±tree can contain underflown pages. We say that a structurally consistent B±tree is *balanced* if none of its non-root pages are underflown.

We say that a B±tree page P is *about to underflow* if (1) P is the root page and contains only two child links, (2) P is a non-root leaf page and contains only m_1 database records, or (3) P is a non-root index page and contains only m_2 child links.

When searching for key value u, then in non-leaf page we follow the pointer P_1, if $u \leq v_1$, and the pointer P_i, if $v_{i-1} < u \leq v_i$. An example of a B±tree is shown in Figure 1. The data values of the database records are not shown.

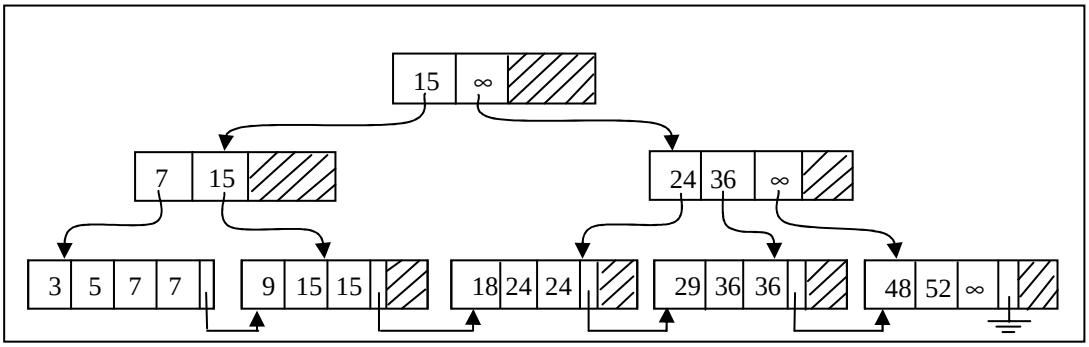

Figure 1. A B±tree that represent a database consisting of records with key values, 3, 5, 7, 9, 15, 18, 24, 29, 36, 48, 52.

Execution of B±tree Structure Modifications as Atomic Actions

To save time and efforts when a transaction aborts or the system fails, we would like the B±tree structure modifications made by a transaction T to be committed regardless of whether

www.manaraa.com

T will eventually commit or abort. That is, the structure modifications are executed as atomic actions [13], [15], and will never be undone once executed to completion. In the literature, a structure modification can be executed as an atomic action either by generating a special transaction [13], or by executing the structure modification as a nested top action [2], [3], [4].

In the special-transaction approach when an updating transaction T finds that it needs to execute a B±tree structure modification, then a new (dummy) transaction identifier T´ and the log record <T´, begin> are generated. The process that generates the transaction T executes the structure modification, updates the involved pages, generates log records and updates the Page-LSNs of the pages, on behalf of T´. When T completes the structure modification, then the log record <T´, commit> is generated. Therefore, if T were to rollback after the structure modification has been completed, that is, after generating the log record <T´, commit>, then the log records related to the structure modification will be bypassed, since these log records belong to the committed transaction T´. However, if the system fails before the log record <T´, commit> has been generated and written to disk, then the incomplete structure modification would have to be undone, since T´ was not committed when the system failure occurred. This guarantees that the structure modification either is executed to completion or all its effects are undone.

Nested top actions actually provide a lightweight implementation for the special transactions. When an updating transaction T needs to execute a B±tree structure modification, then T saves the LSN of the last log record it has generated, before starting the execution of the structure modification. Then T executes the structure modification, updates the involved pages, generates log records, and updates the Page-LSNs. When T completes the structure modification, it generates a dummy compensation log record (CLR), which is set to point to the log record whose LSN was saved previously, as shown in Figure 2.

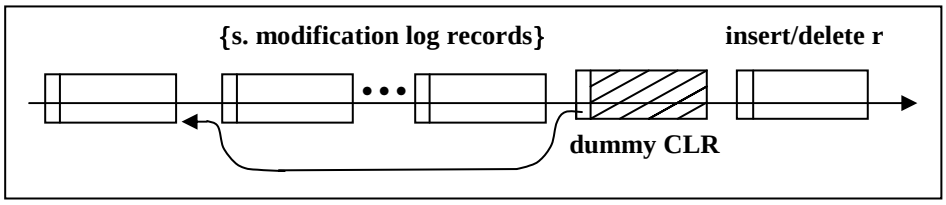

Figure 2. Part of the log showing a structure modification (triggered by an insert/delete of record r) which is executed as a nested top action.

Therefore, the CLR lets the transaction T, if it were to rollback after completing the structure modification, bypass the log records related to the B±tree structure modification. However, if a system failure were to occur before the dummy CLR is written to disk, then the incomplete structure modification will be undone. Again, the structure modification either is executed to completion or all of its effects are undone.

In the former approach, when a structure modification is completed, neither the log record <T´, commit> nor the other log records related to the structure modification need be forced to disk. Similarly, in the latter approach, when the structure modification is completed, the dummy compensation log record (CLR) is not forced to disk. The two approaches are equivalent but are not efficient compared to our new approach.

B±tree Structure-Modification Operations and Logging

We introduce a new approach for handling B±tree structure modifications. In our new approach, a structure modification involving several levels of the tree is divided into a sequence of small atomic actions, each of which involves X-locking three pages on two adjacent levels of the tree. Each structure modification is logged using a single redo-only log record. Thus a completed atomic structure modification need never be undone during normal processing or restart recovery. A completed atomic structure modification brings the tree into

www.manaraa.com

a structurally consistent and balanced state whenever the tree was initially structurally consistent and balanced. To execute atomic structure modifications we present five structure modification operations with their algorithms given below.

***Split*(P,Q).** Given are the Page-ids of an X-latched parent page P and a U-latched child page Q, where P is not full and Q is full. The algorithm allocates a new page Q´, splits Q into Q and Q´ and links Q´ to its parent P. Upon return, the parameter Q denotes the page (Q or Q´) that covers the search key value. That page is kept X-latched while the latches on the other two pages are released.

Step 1. Acquire X latches on the storage-map page M and some page Q´ marked as unallocated in M, and upgrade the U latch on Q to an X latch.

Step 2. Mark Q´ as allocated in M and format Q´ as an empty B±tree page.

Step 3. Let u´ be the current highest key value in Q, and let u be the key value that splits Q evenly. Move all the records with key values greater than u from Q to Q´ and keep the records with key values less than or equal to u in Q.

Step 4. Change the index record (u´, Q) associated with child page Q in the parent P into (u, Q) and inserts the index record (u´, Q´) associated with new child page Q´ into the parent P.

Step 5. If Q is a leaf page, then set the Page-link of Q´:= the Page-link of Q, the Page-link of Q:= Q´, and λ := the Page-link of Q´. Otherwise, set λ := nil.

Step 6. Generate the redo-only log record <T, split, P, Q, Q´, M, u, u´, λ, V, n> where V is the set of records moved from Q to Q´ and n is the LSN of the previous log record generated by T, and update the Page-LSNs of M, P, Q and Q´.

Step 7. Release the X latches on M, P and the page (Q or Q´) that does not cover the search key value. Set Q:= the page (Q or Q´) that covers the search key value. □

***Merge*(P,Q,R).** Given are the Page-ids of an X-latched parent page P and its U-latched child pages Q and R, where P is not about to underflow, R is the right sibling of Q, and the records in Q and R all fit in a single page. The algorithm merges R into Q and deallocates R. Upon return, Q remains X-latched while the latches on P and R are released.

Step 1. Acquire an X latch on the storage-map page M, and upgrade the U latches on Q and R to X latches.

Step 2. Move all the records from R to Q. If Q is a leaf page, then set the Page-link of Q:= the Page-link of R and set λ := the Page-link of R. Otherwise, set λ := nil.

Step 3. Let (u, Q) and (v, R) be the index records associated with the child pages Q and R, respectively in the parent P. Unlink the child page R from its parent P by deleting the index record (v, R) from P and changing the index record (u, Q) in P into (v, Q).

Step 4. Mark R as unallocated in M.

Step 5. Generate the redo-only log record <T, merge, P, Q, R, M, u, v, λ, V, n> where V is the set of records moved from R to Q and n is the LSN of the previous log record generated by T, and update the Page-LSNs of M, P, Q and R. Release the X latches on M, P and R. □

***Redistribute*(P,Q,R).** Given are the Page-ids of an X-latched parent page P and its U-latched child pages Q and R, where R is the right sibling of Q, and the pages Q and R cannot be merged. The algorithm redistributes the records in Q and R evenly. Upon return, the parameter Q denotes the page (Q or R) that covers the search key value; that page remains X-latched while the latches on the other two pages are released.

www.manaraa.com

Step 1. Upgrade the U latches on Q and R to X latches.

Step 2. Let u be the current highest key value in Q, and let u´ be the key value in the page (Q or R) that redistributes the records in these pages evenly.

Step 3. If u´ > u, then move all the records with key values less than or equal to u´ from R to Q. Otherwise, move all the records with key values greater than u´ from Q to R.

Step 4. Change the index record (u, Q) associated with the child page Q in the parent P to (u´, Q).

Step 5. Generate the redo-only log record <T, redistribute, P, Q, R, u´, V, Y, n> where V is the set of the records moved and Y is the page (Q or R) that received the records in V and n is the LSN of the previous log record generated by T, and update the Page-LSNs of P, Q and R.

Step 6. Release the X latches on P and the page (Q or R) that does not cover the search key value, and set Q:= the page (Q or R) that covers the search key value. □

***Increase-tree-height*(P).** Given is the Page-id of the X-latched full root page P. The algorithm allocates two pages P´ and P´´, distributes the records in P evenly between P´ and P´´ and makes P´ and P´´ children of P. Upon return, the parameter P denotes the page (P´ or P´´) that covers the search key value. That page is kept X-latched, while the latches on the other two pages are released.

Step 1. Acquire X latches on the storage-map page M and some pages P´ and P´´ marked as unallocated in M.

Step 2. Mark P´ and P´´ as allocated in M, and format P´ and P´´ as empty B±tree pages.

Step 3. Determine the key value u that splits P evenly, move all the records with key values greater than u from P to P´´, and move all the remaining records from P to P´.

Step 4. Let u and ∞ be the high-key values of P´ and P´´, respectively, and insert the index records (u, P´) and (∞, P´´) into P.

Step 5. Generate the redo-only log record <T, increase-tree-height, P, P´, P´´, M, u, ∞, V1, V2, n> where V1 is the set of records that moved from P to P´ and V2 is the set of records that moved from P to P´´, and n is the LSN of the previous log record generated by T, and update the Page-LSNs of M, P, P´, and P´´.

Step 6. Release the X latches on M, P and the page (P´ or P´´) that does not cover the search key value, and set P:= the page (P´ or P´´) that covers the search key value. □

***Decrease-tree-height*(P,Q,R).** Given are the Page-ids of the X-latched root page P and its U-latched child pages Q and R, where Q and R are the only children of P and can be merged. The algorithm moves the records in Q and R to P and deallocates Q and R. Upon return, P remains X-latched while the latches on the other two pages are released.

Step 1. Acquire an X latch on the storage-map page M, and upgrade the U latches on Q and R to X latches.

Step 2. Delete the only two remaining index records (u, Q) and (∞, R) associated with the child pages Q and R from the parent page P.

Step 3. Move all the records from Q and R to P. Mark Q and R as unallocated in M.

www.manaraa.com

Step 4. Generate the redo-only log record <T, decrease-tree-height, P, Q, R, M, u, ∞, V1, V2, n> where V1 is the set of records moved from Q to P, V2 is the set of records moved from R to P and n is the LSN of the previous log record generated by T, and update the Page-LSNs of M, P, Q and R.

Step 5. Release the X latches on M, Q and R. □

Lemma 1. Let B be a structurally consistent and balanced B±tree. Then any of the structure-modification operations split(P,Q), merge(P,Q,R), redistribute(P,Q,R), increase-tree-height(P) and decrease-tree-height(P,Q,R), whenever the preconditions for the operation hold, produces a structurally consistent and balanced B±tree when run on B by some transaction T.

Proof. From the algorithms for the operations, it is evident that if the preconditions for the operation are satisfied, then the operation can be run on B and preserves the structural consistency and balance of B. The preconditions for split(P,Q) state that Q is full and its parent P is not full. In Split(P,Q), Q is split into Q and the allocated page Q´, and Q´ is made as a child of P. Thus the resulting B±tree is structural consistent and balanced. The preconditions for merge(P,Q,R) and redistribute(P,Q,R) state that R is a right sibling of Q and their parent P is not about to underflow. In merge(P,Q,R), R is merged into Q and R unlinked from its parent P, while in redistribute(P,Q,R), the records in Q and R are redistributed evenly, and hence the resulting B±tree in either case is structurally consistent and balanced. The preconditions for increase-tree-height(P) state that P is the root of the tree and full. The operation distributes the records in P evenly between the allocated pages P´ and P´´ and makes P´ and P´´ as children of P. Thus the resulting B±tree is structurally consistent and balanced. The preconditions for decrease-tree-height(P,Q,R) state that Q and R are the only children of P, R is an about-to-underflow right sibling of Q, and the records in both pages Q and R can fit in one page. The operation replaces the contents of P by the contents of Q and R, and hence the resulting B±tree is structurally consistent and balanced. Concurrent B±tree structure-modification operations performed by transactions preserve the structural consistency of the tree, because the tree pages involved in each operation are kept X-latched for the duration of the operation. □

The following algorithm is used by a transactions T to execute a B±tree structure modification (page split) involving several levels of the tree as a sequence of small atomic actions.

Algorithm 1. Split the full pages on the split path starting from the highest non-full page down to the target leaf page P when these pages are U-latched by T.

Step 1. Let P:= the Page-id of the highest U-latched page on the split path.

Step 2. Upgrade the U latch on P to an X latch.

Step 3. If P is the root of the B±tree and P is full, then increase-tree-height(P).

Step 4. If P is a leaf page, then return.

Step 5. Determine the U-latched full child page Q of P that is on the split path.

Step 6. Split(P,Q), set P := Q, and go to Step 4. □

The following algorithm is used by a transactions T to execute a B±tree structure modification (page merge/redistribute) involving several levels of the tree as a sequence of small atomic actions.

Algorithm 2. Merge or redistribute the pages along the merge (or redistribute) path which are about to underflow with their sibling pages when these pages are U-latched by T.

www.manaraa.com

Step 1. Let P be the Page-id of the highest U-latched not-about-to-underflow page on the merge (or redistribute) path.

Step 2. Upgrade the U latch on P to an X latch.

Step 3. Determine the U-latched child page Q of P that is about to underflow and is on the merge (or redistribute) path.

Step 4. If Q is the rightmost child of its parent P, then go to Step 8.

Step 5. Determine the right sibling page R of Q and U-latch R.

Step 6. If P is the root of the B±tree and P has just two child pages Q and R which can be merged, then decrease-tree-height(P,Q,R) and go to Step 12.

Step 7. If Q and R can be merged, then merge(P,Q,R) else redistribute(P,Q,R). Set P:= Q and go to Step 12.

Step 8. Now Q is the rightmost child of its parent P. Determine the left sibling page L of Q.

Step 9. Release the U latch on Q (to avoid a deadlock), and U-latch L and Q (in this order).

Step 10. If P is the root of the B±tree and P has just two child pages L and Q which can be merged, then decrease-tree-height(P,L,Q) and go to Step 12.

Step 11. If L and Q can be merged, then merge(P,L,Q) else redistribute(P,L,Q). Set P:= L.

Step 12. If P is a leaf page, then return. Otherwise, go to Step 3. □

Lemma 2. Let B be a structurally consistent and balanced B±tree of height h. Assume that a transaction T performs tree-structure modifications on B using Algorithm 1 or 2. Then T accesses 2h+1 pages at most, keeps at most h pages U-latched and three pages X-latched at a time, and accesses and X-latches the storage-map page M at most h times.

Proof. It is clear from the algorithms of the structure-modification operations that each operation acquires three X latches on the pages involved in the operation. Hence, when transaction T performs tree-structure modifications using Algorithm 1 or 2, T X-latches three pages at a time, two of which are on the same level. Thus in the worst case, Algorithm 1 or 2 accesses and modifies 2h+1 pages. The rest of the pages on the structure-modification path are kept U-latched. The storage-map page is accessed when a new page needs to be allocated in the operations split(P,Q) and increase-tree-height(P) and when a page needs to be deallocated in the operations merge(P,Q,R) and decrease-tree-height(P,Q,R). Hence in the worst case, the storage-map page M may be accessed h times. □

Concurrency Control

We use the latch-coupling protocol [1], [2], [3], [4] for page-level concurrency control on index pages. For concurrency control on leaf pages we use the key-range locking protocol. Thus, different transactions can fetch, insert or delete database records from leaf pages concurrently. The latch-coupling protocol is assumed to be deadlock-free. This is guaranteed if an S latch is never upgraded to an X latch. We assume that a transaction T fixes an index or data page P in the buffer pool before it latches P and unfixes P after unlatching P.

A fetch operation by a transaction T in a history H is *an unrepeatable read* if some other transaction T´ updates (inserts or deletes) a record whose key belongs to the key range read by the fetch operation, before T commits or completes its rollback. To avoid unrepeatable reads, we use the *key-range locking protocol* [1], [2], [3], [4].

www.manaraa.com

In the key-range locking protocol, transactions acquire in their forward-rolling phase commit-duration X locks on inserted records and on the next records of deleted records, short-duration X locks on deleted records and on the next records of inserted records, and commit-duration S locks on fetched records. Short-duration locks are held only for the time the operation is being performed. Commit-duration X locks are released after the transaction has committed or completed its rollback. Commit-duration S locks can be released after the transaction has committed or aborted.

No additional record locks are acquired for operations done in the backward-rolling phase of an aborted transaction. To undo an insertion of record r, an aborted transaction T just deletes r under the protection of the X lock acquired during the forward-rolling phase on r. To undo a deletion of record r, T just inserts r under the protection of the X lock acquired during the forward-rolling phase on the next record r´.

There is one troublesome point in using the key-range locking protocol. Namely, if a transaction must wait for a lock on the next record r´, then when the lock is granted, the record r´ may no longer be the right record to lock. This is because another transaction may have deleted r´ or inserted a new record just before r´. Therefore, if a transaction T waits for a lock on the next record, then T must revalidate the next record when the lock is granted. If the next record has changed during the lock wait due to a page update, then T should release the lock on the old next record and request a lock on the current next record.

B±tree Traversals, Fetch, Insert and Delete

A transaction T takes as input a key value k and *traverses* the B±tree, using the latch-coupling protocol with S latches (Algorithm 3). When the leaf page P covering the search key value k is reached, then P is S-latched for fetching and X-latched for updating.

Algorithm 3. B±tree traversal using latch coupling with S latches.

Step 1. Set P:= the Page-id of the root page of the B±tree and S-latch P.

Step 2. If P is a leaf page, then go to Step 5.

Step 3. Search P for the child page Q and S-latch Q.

Step 4. Release the S latch on P, set P:= Q, and go to Step 2.

Step 5. Now P is the leaf page that covers k. If P is needed for updating, then X-latch P. □

When an updating T traversing the B±tree reaches a full leaf page P (when inserting) or an about-to-underflow leaf page P (when deleting), T releases its X latch on P (to avoid a deadlock with another transaction traversing the tree). Then T retraverses the B±tree, using Algorithm 4 which results in U-latching the reached leaf page P and its ancestors up to the first non-full (not-about-to-underflow) page.

Algorithm 4. B±tree traversal using latch coupling with U latches for updating.

Step 1. Set P:= the Page-id of the root page of the B±tree, and U-latch P.

Step 2. If P is a leaf page, then return.

Step 3. Search P for the child page Q covering the search key value k, and U-latch Q.

Step 4. If the child page Q is full (when inserting) or about to underflow (when deleting), then save the Page-id of P, set P:= Q, and go to Step 2.

www.manaraa.com

Step 5. If the child page Q is not-full (resp. not-about-to-underflow) and some ancestors of Q (whose Page-ids were saved previously) are still U-latched, then release the U latches on the ancestors of Q.

Step 6. Release the U latch on page P, set P:= Q, and go to Step 2. □

Lemma 3. Let B be a structurally consistent and balanced B±tree. Then the read-mode traversal (Algorithm 3), the update-mode traversal (Algorithm 4) and the tree-structure-modifications (Algorithms 1 and 2) are deadlock-free.

Proof. The read-mode traversal (Algorithm 3) and the update-mode traversal (Algorithm 4) employ the lock-coupling protocol and acquire page locks in a top-down order. In the Algorithms 1 and 2, the U locks on the pages involved in the tree structure modifications are acquired in a top-down, left-to-right order. When an about-to-underflow page Q is the rightmost child of its parent P, the U lock on Q is released and U locks are acquired on the pages L (the left sibling of Q) and Q (see Steps 8 and 9 of Algorithm 2). Again in Algorithms 1 and 2, only U locks are upgraded to X locks. Therefore, we may conclude that our read-mode-traversal, update-mode traversal and tree structure modification algorithms are deadlock-free. □

The following algorithm implements the operation Fetch[k,θu, x].

Algorithm 5. Given a key value u<∞, fetch the data record r = (k, x) with the least key value k satisfying kθu. As a result of a previous call to this algorithm, the Page-id P of the page where the previously fetched record resided may also be given as input.

Step 1. If P is not given as input, then go to Step 5. Otherwise, S-latch P.

Step 2. Find the lowest key value w, the highest key value v and the high-key value v´ in P.

Step 3. If u < w or u > v´, then go to Step 5.

Step 4. If u > v or u = v and θ = ”>”, then go to Step 6. Otherwise go to Step 7.

Step 5. Traverse the B±tree from the root using latch coupling with S latches (Algorithm 3) with u as the input key value. Search the reached target leaf page P, determine the highest key value v and the high-key value v´ in P, set Q:= nil, and go to Step 4.

Step 6. The record to be fetched resides in P´, the page next to P. S-latch P´, set Q:= P and P:= P´, and save the Page-LSN of Q.

Step 7. Now P contains the record to be fetched. Determine the record r in P with the least key value k satisfying kθu. Request a conditional S lock on r. If the lock is granted right away, then S-lock r, release the S latch on P, save the Page-id P, and return with r.

Step 8. If r is the lowest record in P and Q ≠ nil, then go to Step 13.

Step 9. Save the Page-LSN of P and release the S latch on P. Request unconditional S lock on r. When the lock is granted, S-latch P and search P for r.

Step 10. If the Page-LSN of P has not changed, then save the Page-id P, release the S latch on P, and return with r.

Step 11. If P does not cover r or P is not a leaf page or P is not part of the B±tree any more, then release the S latch on P, release the X lock on r, and go to Step 5.

www.manaraa.com

Step 12. If r is present in P and the key value k of r is still the least key value in P satisfying kθu, then release the S latch on P, save the Page-id P, and return with r. Otherwise, release the S lock on r, determine the highest key value v in P, and go to Step 4.

Step 13. The record r to be fetched is the lowest record in P, and r was found by entering P from its left sibling Q. Save the Page-LSN of P and release the S latch on P. Request unconditional S lock on r. When the lock is granted, S-latch P and Q, and search P for r.

Step 14. If the Page-LSN of Q has not changed, then release S latch on Q and go to Step 10.

Step 15. If Q does not cover the key value u or Q is not a leaf page or Q is not part of the B±tree any more, then release S latches on P and Q, release the X lock on r, and go to Step 5.

Step 16. The Page-LSN of Q has changed but Q still covers u. Determine the highest key value v in Q.

Step 17. If u = v and θ = "≥", then release the S latches on P and Q, set P:= Q, save the Page-id P, and return with r.

Step 18. If u = v and θ = ">", then release the S latch on Q and go to Step 10.

Step 19. Now u < v in the page Q. Release the S latch on P, release the S lock on r, set P:= Q, and go to Step 7. □

The next algorithm implements the operation Insert[k, x] in the forward-rolling phase of T.

Algorithm 6. Given a record r = (k, x) with key value k, insert r into the database.

Step 1. Traverse the B±tree using latch coupling with S latches (Algorithm 3) to reach the leaf page P that covers the key value k of record r.

Step 2. If the X-latched leaf page P is full, then go to Step 8.

Step 3. Determine the page P´ that holds the record r´ with the least key value greater than the key value k of r and X-latch P´. Thus P´ is P (when r´ is found in P) or the page next to P (otherwise).

Step 4. Lock-records(P,r,P´,r´). If the exception "records cannot be X-locked" is returned, then go to Step 1.

Step 5. Search P for the position of insertion of r. If a key value that matches the key value k of r is found, then terminate the insert operation, and return with the exception "uniqueness violation".

Step 6. Insert r into P, generate the redo-undo log record <T, insert, P, r, n> where n is the LSN of the previous log record generated by T, and update the Page-LSN of P.

Step 7. Release the X latches on P and P´, release the X lock on r´, and hold the X lock on r for commit duration and return.

Step 8. Release the X latch on P (to avoid a deadlock with another transaction traversing the B±tree) and retraverse the B±tree from the root page, using latch coupling with U latches (Algorithm 4), and set P:= Q (the reached leaf page).

Step 9. If P is full, then perform the needed page splits using Algorithm 1, and go to Step 3. Otherwise, upgrade the U latch on P to an X latch and go to Step 3. □

www.manaraa.com

The following algorithm implements the operation Delete[k, x] in the forward-rolling phase of a transaction T.

Algorithm 7. Given a key value k, delete the record r = (k, x) with key value k from the database.

Step 1. Traverse the B±tree using latch coupling with S latches (Algorithm 3) to reach the leaf page P that covers the key value k of record r.

Step 2. If P is about to underflow, then go to Step 8.

Step 3. Determine the page P´ that holds the record r´ with the least key value greater than the key value of r and X-latch P´. Thus P´ is P (when r´ is found in P) or the page next to P (otherwise).

Step 4. Lock-records(P,r,P´,r´). If the exception "records cannot be X-locked" is returned, then go to Step 1.

Step 5. Search P for a record with key value k. If no record with key value k was found in P, then terminate the delete operation, and return with the exception "record not found".

Step 6. Delete r from P, generate the redo-undo log record <T, delete, P, r, n> where n is the LSN of the previous log record generated by T, and update the Page-LSN of P.

Step 7. Release the X latches on P and P´, release the X lock on r, and hold the X lock on r´ for commit duration and return.

Step 8. Release the X latch on P (to avoid a deadlock with another transaction traversing the B±tree) and retraverse the B±tree from the root page, using latch coupling with U latches (Algorithm 4), and set P:= Q (the reached leaf page).

Step 9. If P is about to underflow, then perform the needed page merge/redistribute using Algorithm 2, and go to Step 3. Otherwise, upgrade the U latch on P to an X latch and go to Step 3. □

The following algorithm is used by an updating transaction to acquire X locks on the record r and the record r´ next to r when inserting or deleting r.

***Lock-records*(P,r,P´,r´):** Given are the Page-id P of an X-latched leaf page P covering record r and the Page-id P´ of an X-latched leaf page P´ containing the record r´ next to r. The algorithm acquires X locks on r and r´ if possible. Otherwise, it releases the X latches on P and P´ and returns with the exception "records cannot be X-locked".

Step 1. Request conditional X locks on the records r and r´.

Step 2. If the locks are granted right away, then return.

Step 3. Save the Page-LSNs of P and P´. Release the X latches on P and P´, and request unconditional X locks on r and r´. When the record locks are granted, X-latch P and P´.

Step 4. If the Page-LSNs of P and P´ have not changed, then return.

Step 5. If the Page-LSN of P has changed and either P does not cover r or P is not a leaf page or P is not part of the B±tree any more, then go to Step 12.

Step 6. Now P still covers r. Search P for a record r´´ with the least key value greater than the key value of r. If no such record is found in P, then go to Step 10.

www.manaraa.com

Step 7. Now P contains r´´. If P = P´, then go to Step 8. Otherwise go to Step 9.

Step 8. If the key values of r´´ and r´ are equal, then return. Otherwise, release the X locks on r and r´ in P, set r´:=r´´, and go to Step 1.

Step 9. Release the X latch on P´ and release the X locks on r and r´, set P´:= P and r´:= r´´, and go to Step 1.

Step 10. If P= P´, then go to Step 12. Otherwise, determine the page P´´ currently next to P.

Step 11. If P´´= P´ and the key value of r´ is equal to the key value of the first record in P´, then return.

Step 12. Release the X latches on P and P´, release the X locks on r and r´, and return with the exception "records cannot be X-locked". □

Page-Oriented Redo, Page-Oriented Undo and Logical Undo

Page-oriented redo means that, when a page update needs to be redone at recovery time, the Page-id field of the log record is used to determine uniquely the affected page. That is, no other page needs to be accessed or examined. Similarly, *page-oriented undo* means that, when page update needs to be undone during transaction rollback, the Page-id field of the log record is used to determine the affected page. The same page is accessed and the update is undone on that page. Page-oriented redo and page-oriented undo provide faster recovery, because only the pages mentioned in the log record are accessed.

The operations in the backward-rolling phase of an aborted transaction are implemented by the algorithms below. The algorithms are used during normal processing to roll back a transaction as well as during the undo pass of restart recovery to roll back all active transactions. An undo operation in the backward-rolling phase of an aborted transaction T is logged by writing a *compensation log record* (CLR) [14] that contains, besides the arguments needed to replay the undo operation, the LSN of the log record for the next database operation by T to be undone. Such a compensation log record is a redo-only log record.

Uncommitted updates by a transaction T may move to a different leaf page due to structure modifications triggered by other concurrent transactions. Thus, during recovery time a page-oriented undo can fail, because an update to be undone may no longer be covered by the page mentioned in the log record. When a page-oriented undo fails, a *logical undo* [2], [3] is used: the backward-rolling transaction retraverses the B±tree from the root page down to the leaf page that currently covers the update to be undone, and then that update is undone on that page. Naturally, the undo of such an update may trigger a tree-structure modification, which is executed and logged using redo-only log records. Logical undo provides a higher level of concurrency than would be possible if the system only allowed for page-oriented undo.

The following algorithm implements the inverse operation Undo-delete[k, x] in the backward-rolling phase of an aborted transaction T.

Algorithm 8. Undo the deletion of record r = (k, x). Given is a redo-undo log record <T, delete, P, r, n> where r is a record with key value k that was deleted by T from page P, and n is the LSN of the previous log record generated by T in its forward-rolling phase. The algorithm reinserts r into P if P still covers k. Otherwise, T retraverses the B±tree to reach the covering leaf page Q and inserts r into Q.

Step 1. X-latch P. If P still covers k and there is a room for r in P, then go to Step 4.

Step 2. Undo the deletion of r logically. Release the X latch on P, retraverse the B±tree from the root page using Algorithm 4 with k as an input key value to reach the covering leaf page Q, and set P:= Q.

www.manaraa.com

Step 3. If P is full, then perform the needed page splits using Algorithm 1. Otherwise, upgrade the U latch on P to an X latch.

Step 4. Insert r into P, generate the compensation log record <T, undo-delete, P, r, n>, update the Page-LSN of P, and release the X latch on P. □

The following algorithm implements the inverse operation Undo-insert[k, x] in the backward-rolling phase of an aborted transaction T.

Algorithm 9. Undo the insertion of record r. Given is a redo-undo log record <T, insert, P, r, n> where r is a record with key value k that was inserted by T into page P, and n is the LSN of the previous log record generated by T in its forward-rolling phase. The algorithm deletes r from P if P still contains r. Otherwise, T retraverses the B±tree to reach the covering leaf page Q and deletes r from Q.

Step 1. X-latch P. If P still contains r and will not underflow if r is deleted, then go to Step 4.

Step 2. Undo the insertion of r logically. Release the X latch on P, retraverse the B±tree from the root page using Algorithm 4 with k as an input key value to reach the covering leaf page Q, and set P:= Q.

Step 3. If P is about to underflow, then perform the needed page merging (or record redistribution) using Algorithm 2. Otherwise, upgrade the U latch on P to an X latch.

Step 4. Delete r from P, generate the compensation log record <T, undo-insert, P, r, n>, update the Page-LSN of P, and release the X latch on P. □

Lemma 4. Let T be a transaction and B a structurally consistent B±tree of height h. Any operation Fetch[k,θu, x] by T on B accesses at most h+1 pages of B and keeps at most two of those pages S-latched for T at a time. Any operation Insert[k, x] or Delete[k, x] in the forward-rolling phase of T and any logically implemented inverse operation Undo-delete[k, x] or Undo-insert[k, x] in the backward-rolling phase of T accesses at most 2h+1 pages of B, keeps at most three of those pages X-latched T at a time, and produces a structurally consistent and balanced B±tree. Any physiologically implemented inverse operation Undo-delete[k, x] or Undo-insert[k, x] in the backward-rolling phase of T accesses at most one page of B, and produces a structurally consistent and balanced B±tree.

Proof. From Algorithms 3 and 5, it follows that Fetch[k,θu, x] accesses h pages to reach leaf-page level while keeping two pages S-latched at a time. When a leaf page is reached, an extra page access may be needed to locate the least key value k satisfying kθu. The claims for Insert[k, x], Delete[k, x], Undo-delete[k, x] and Undo-insert[k, x] follow from Lemma 2 and Algorithms 6, 7, 8 and 9. A physiological Undo-delete[k, x] always checks that the page has room for (k, x), and a physiological Undo-insert[k, x] always checks that the page will not underflow from the deletion of (k, x). □

Transaction Abort and Rollback

A transaction T is rolled back during normal processing when T performs the “abort” operation or T gets involved in a deadlock or the flushing process of the log records of T is not completed successfully. During the rollback, the log records are undone in the reverse chronological order, and for each log record that is undone, a compensation log record (CLR) is generated. When a redo-only log record is encountered, the Prev-LSN of such log record is used to determine the next log record to be undone. If any structure modifications are executed during transaction rollback, then such structure modifications are logged using redo-only log records. Moreover, no B±tree structure modifications are undone during transaction rollback. The backward-rolling phase $A\alpha^{-1}R$ of an aborted transaction T = $B\alpha A\alpha^{-1}R$ is executed by the following algorithm.

www.manaraa.com

Algorithm 10. Rollback an aborted transaction T.

Step 1. Generate the log record <T, abort, n> where n is the LSN of the last log record generated by T during its forward-rolling phase (n is obtained from the transaction table).

Step 2. Get the log record generated by T with LSN equal to n.

Step 3. If the log record is <T, begin>, then go to Step 7.

Step 4. If the log record is a redo-only log record of type "split" or "merge" or redistribute" or "increase-tree-height" or "decrease-tree-height", then use the Prev-LSN of this log record to determine the next log record to be undone, and go to Step 3.

Step 5. If the log record is a redo-undo log record of type "delete" or "insert", then undo the logged update, using Algorithm 8 or 9, respectively.

Step 6. Get the next log record to be undone using the Prev-LSN of the log record being undone, and go to Step 3.

Step 7. Generate the log record <T, rollback-completed> and release all locks of T. □

Transaction Execution

In the forward-rolling phase of transaction T, the operations fetch, insert and delete are implemented by the algorithms 5, 6 and 7, respectively, while the operations begin and commit are implemented by the following algorithms.

Begin transaction: Get a new transaction identifier T, and generate the log record <T, begin>. □

Commit transaction T: Generate the log record <T, commit>, flush the log, and release all locks held by T. □

In the backward-rolling phase of transaction T, the operations Undo-delete[k, x] and Undo-insert[k, x] are executed by the algorithms 8 and 9, while the operations abort and rollback-completed are executed by the following algorithms.

Abort transaction T: Generate the log record <T, abort, n> where n is the LSN of the previous log record generated by T in its forward-rolling phase. □

Rollback-completed: Generate the log record <T, rollback-completed>, flush the log, and release all locks held by T. □

Theorem 5. Assume that each transaction in its forward-rolling phase accesses records in ascending key order and that the operations Fetch, Insert, Delete, Undo-delete and Undo-insert are implemented by Algorithms 5, 6, 7, 8 and 9. Then no deadlocks can occur.

Proof. The operations use the same read-mode and update-mode traversal algorithms 3 and 4. Thus we conclude by Lemma 3 that no deadlock can occur between page latches. No deadlocks can occur between record locks either, because the execution of single Fetch, Insert, Delete, Undo-delete and Undo-insert operations by Algorithms 5, 6, 7, 8 and 9 is deadlock-free. Note that S locks on records are never upgraded and that the Fetch operation locks only one record, the operations Insert and Delete lock two records in ascending key order, and the operations Undo-delete and Undo-insert acquire no new record locks at all. For transactions containing multiple operations in their forward-rolling phase, the assumption of accessing records in ascending key order is needed to avoid deadlocks that may occur when transactions lock two or more records in a different order. Finally, no deadlock can be caused

www.manaraa.com

by the interaction of record locks and page latches, because no transaction is made to wait for a lock on a record covered by page P while holding a latch. □

Let H be a history of forward-rolling, committed, backward-rolling and rolled-back transactions that can be run on database D1 and let B1 be a structurally consistent and balanced B±tree with db(B1) = D1. Let H´ be a string of B±tree operations containing begin, commit, abort, rollback-completed, traversal, structure-modification, fetch, insert and delete operations that can be run on B1. We say that H´ is an *implementation* of H on B1, if the subsequence of begin, commit, abort, rollback-completed, fetch, insert and delete operations included in H´ is equal to H.

Theorem 6. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back transactions that can be run on database D. Further let B be a structurally consistent and balanced B±tree with db(B) = D, and let H´ be an implementation of H on B using the above algorithms. Then H´ produces a structurally consistent and balanced B±tree.

Proof. A formal proof would use induction on the number of operations in H´. For the purposes of the proof we may regard each of the structure-modification operations split(P,Q), merge(P,Q,R), redistribute(P,Q,R), increase-tree-height(P) and decrease-tree-height(P,Q,R) as an atomic operation. Also the insertion of a record (k, x) into a leaf page P, and the deletion of a record from leaf page P are atomic operations. This is justified because structure modifications and leaf page updates are performed under the protection of X latches. The structural consistency and balance of the tree produced by H´ follows from Lemmas 1 and 4. □

Theorem 7. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back transactions that can be run on database D under the key-range locking protocol. Further let B be a structurally consistent and balanced B±tree with db(B) = D. Then there exists a B±tree operation string H´ that implements H on B using the above algorithms. Moreover, each operation implementation in H´ includes at most one traversal of B.

Proof. In one such H´, the implementations of individual operations Fetch[k,θu, x], Insert[k, x], Delete[k, x], Undo-insert [k, x], and Undo-delete[k, x] by different transactions are run serially, so that the implementations of different operations are not interleaved. In the worst case, each operation is implemented logically in H´ and includes a single traversal of the tree from the root down to the covering leaf page. No latch conflicts can occur during the traversals at any level of the tree, because an X latch on a page is only held for the time of the update operation in question and because any S latch on a leaf page held for longer duration can be released immediately after the Fetch operation is done. As H can be run on D under the key-range locking protocol, no lock conflicts can occur with the record locks either. □

Restart Recovery

The goal of recovery is to ensure that the B±tree only reflects the updates of committed transactions and none of those of uncommitted transactions. We use a redo and undo recovery protocol for handling the system failures. Our recovery protocol is based on ARIES [14], which support the steal and no-force policies for buffer management and fine-grained concurrency control. We assume that no new transactions are accepted to the system during restart recovery until the execution of the recovery protocol is completed. The restart recovery protocol consists of three passes: an analysis pass, a redo pass, and an undo pass.

In the *analysis pass*, the log is scanned forward starting from the start-checkpoint log record of the last complete checkpoint, up to the end of the log. A *modified-page list* of pages that were potentially more up-to-date in the buffer than in stable storage and an *active-transaction list* of transactions that were in progress (i.e., forward-rolling or backward-rolling) at the time of the crash are constructed, using the information in the checkpoint log record and the encountered log records. The analysis pass also determines the *Redo-LSN*, i.e., the LSN of the earliest log record that needs to be redone. The Redo-LSN is the minimum *Rec-LSN* (recovery LSN) in the created modified-page list. In other words, the analysis pass determines the

www.manaraa.com

starting point of the redo pass in the log, and the list of the transactions that need to be rolled back or whose rollback need to be completed.

The *redo pass* begins at the log record whose LSN equals to Redo-LSN, and then proceeds forward to the end of the log. For each redoable log record of the type "update" or "compensation", the following is performed. If the page mentioned in the log record is not in the modified-page list, then the logged update does not require redo. If the page mentioned in the log record is in the modified-page list and its Rec-LSN is greater than the log record's LSN, then the logged update does not require redo. Otherwise, the page mentioned in the log record is accessed and its Page-LSN is compared with the log record's LSN, in order to check whether or not the page already contains the update, that is, whether or not updated page was written to the disk before the system failure. If the Page-LSN is less than the log record's LSN, then the logged update is redone, that is, applied physiologically to the page and the Page-LSN is set to the log record's LSN. Otherwise, the logged update does not require redo. No logging is performed during the redo pass. By the end of the redo pass, the B±tree will become structurally consistent.

In the *undo pass*, all forward-rolling transactions are aborted and rolled back and the rollback of all backward-rolling transactions is completed. The log is scanned backward from the end of the log until all updates of such transactions are undone. When a log record of type "compensation" (CLR) is encountered, then no action is performed except that the value of the Undo-Next-LSN field of such a CLR is used to determine the next log record to be processed. When a redo-only log record of type "split" or "merge" or "redistribute" or "increase-tree-height" or "decrease-tree-height" is encountered, then no action is performed except that the value of the Prev-LSN of such log record is used to determine the next log record to be processed. When a redo-undo log record of type "delete" or "insert" is encountered, then the logged update is undone, using Algorithm 8 or 9, respectively, except that no X latch is acquired on the affected page.

To undo all uncommitted transaction updates in a single pass over the log during the undo pass of restart recovery, a list containing the LSNs of the next log records to be processed for each transaction being undone is constructed. When a log record is processed, the Prev-LSN (or Undo-Next-LSN, in the case of a CLR) is entered in the list as the next LSN of the log record to be processed. The next log record to be processed will be the log record with the maximum LSN on this list.

The following example shows how the aborted transactions are rolled back during restart recovery, and how the restart recovery is resumed in case of the system fails during the restart recovery.

Example 1. Assume that during the normal processing the stable log contains the following log records.

LSN	Log records
10	<T1, begin>
20	<T1, delete, Q, r_1, 10>
30	<T2, begin>
40	<T1, insert, Q, r_2, 20>
50	<T2, insert, R, r_3, 30>
60	<T1, split, P, Q, Q´, M, u, u', λ, V, 40>
70	<T2, delete, R, r_4, 50>
80	<T1, insert, Q´, r_5, 60>

Assume that after installing the log record <T1, insert, Q´, r_5, 60> in the log, the system fails. When the system is up again, all forward-rolling transactions are aborted and rolled back and the rollback of all backward-rolling transactions is completed. The compensation log records generated during the undo pass are as follows.

www.manaraa.com

90	<T1, abort, 80>
100	<T1, undo-insert, Q´, r_5, 60>
110	<T2, abort, 70>
120	<T2, undo-delete, R, r_4, 50>
130	<T2, undo-insert, R, r_3, 30>
140	<T1, undo-insert, Q, r_2, 20>

Now assume that the system fails again during the restart recovery after installing the generated compensation log record <T1, undo-insert, P, r_2, 20>. When the system is up again, the restart recovery is resumed and the following compensation log records are generated.

150	<T2, rollback-completed>
160	<T1, undo-delete, Q, r_1, 10>
170	<T1, rollback-completed>.

Theorem 8. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back transactions that can be run on database D1 and let B1 be a structurally consistent and balanced B±tree with db(B1) = D1. Further let H´ be an implementation of H on B1 using the above algorithms and let L be the sequence of log records written by the operations in H´. Given the prefix L1 of L stored in the stable log and the (possibly structurally inconsistent) disk version B2 of the B±tree at the time H´ has been run on B1, the redo pass of the ARIES algorithm will produce a structurally consistent and balanced B±tree B3, where db(B3) is the database produced by running on D1 a prefix H1 of H that contains all the database operations logged in L1. Moreover, the undo pass of ARIES will generate a string γ´ of B±tree operations that implements some completion string for H1.

Proof. The way in which updates and structure modifications are generated guarantees that the log records generated by each transaction T are written to the log in the order in which the corresponding operations appear in T. As leaf-page updates and structure modifications are protected by X latches, the log records for the updates and modifications on each page P appear in the log in the order in which the corresponding operations appear in H. Thus, given any prefix L1 of the log L, the redo pass of the ARIES algorithm will produce the B±tree that is the result of running some prefix of H´ on B1. By Theorem 6, B1 is structurally consistent and balanced. The rest of the Theorem follows from Theorem 7. □

Conclusion

Tree-structure modifications can limit the degree of concurrency and complicate the restart recovery. We introduced new B±tree algorithms in which tree-structure modifications are handled adequately. In our B±tree algorithms, tree-structure modifications are executed as atomic actions. Hence, a tree-structure modification which is executed to completion will never be undone regardless of the outcome of the transaction that triggered such tree-structure modification. Structure modifications are protected by X latches that are held for the duration of the structure modification. In our B±tree algorithms, each structure modification X-latches three pages at most simultaneously. This is an improvement over the algorithms presented in [1], [16], where structure modifications are performed unnecessarily or the whole search path is X-latched at once. In our B±tree algorithms, structure modifications can run concurrently with leaf-page updates as well as other structure modifications. This is in contrast to [2], [3], [4], where structure modifications are effectively serialized by the use of “tree latches”, in order to guarantee that logical undo operations always see a structurally consistent tree. A (leaf) page can contain uncommitted updates by several transactions, and uncommitted updates by one transaction can migrate to another page due to structure modifications triggered by other concurrent transactions. All our algorithms work with the steal and no-force buffering policies [1] and with the fine-granularity (record-level) locking protocol [3], [4].

www.manaraa.com

References

1. Gray, J., Reuter, A. *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

2. Mohan, C., Levine, F. "ARIES/IM: an efficient and high concurrency index management method using write-ahead logging", *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 371-380.

3. Mohan, C. "Concurrency control and recovery methods for B±tree indexes: ARIES/KVL and ARIES/IM", *Performance of Concurrency Control Mechanisms in Centralized Database Systems* (V.Kumar, ed.), Prentice Hall, 1996, pages 248-306.

4. Mohan, C. "ARIES/KVL: a key-value locking method for concurrency control of multi-action transactions operation on B-tree indexes", *Proc. of the 16th VLDB Conference*, 1990, pages 392-405.

5. Lomet, D. "Advanced recovery techniques in practice", *Recovery Mechanisms in Database Systems* (V. Kumar and M. Hsu, eds.), Prentice Hall PTR, 1998, pages 697-710.

6. Srinivasan, V., Carey, M. "Performance of B-tree concurrency control algorithms", *Proc. of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 416-425.

7. Maier, D., Salveter, S. "Hysterical B-trees", *J. Information Processing Letter* **12** (1981), pages 199-202.

8. Johnson, T., Shasha, D. "B-trees with inserts and deletes: why free-at-empty is better than merge-at-half", *J. Computer and System Sciences*, **47** (1993), pages 45-76.

9. Jaluta, I. "B-tree concurrency control and recovery in a client-server database management system" Ph.D. Thesis, Department of Computer Science and Engineering, Helsinki University of Technology, 2002. Available at: http://lib.hut.fi/Diss/2002/isbn9512257068/

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

11. Kwong, Y., Wood, D. "A new method for concurrency in B-trees", *IEEE Trans. Software Engineering*, **8** (1982), pages 211-222.

12. Bernstein, P., Hadzilacos, V., Goodman, N. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

13. Lomet, D., Salzberg, B. "Concurrency and recovery for index trees", *The VLDB Journal* **6** (1997), pages 224-240.

14. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging", *ACM Trans. Database Systems* **17** (1992), pages 94-162.

15. Lomet, D., Salzberg, B. "Access method concurrency with recovery", *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 351-360.

16 Mond, Y., Raz, Y. "Concurrency control in B±tree databases using preparatory operations", *Proc. of the 11th VLDB Conference*, 1985, pages 331-334.

www.manaraa.com