The VLDB Journal (2005) 14: 257–277 / Digital Object Identifier (DOI) 10.1007/s00778-004-0140-6

Concurrency control and recovery for balanced B-link trees

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

[1] Department of Computer Science and Engineering, Helsinki University of Technology, P.O. Box 5400, 02015 HUT, Finland (e-mail: {ijaluta, ess}@cs.hut.fi)
[2] Department of Computer Science, University of Helsinki, P.O. Box 68 (Gustaf Hällströmin katu 2b), 00014 University of Helsinki, Finland (e-mail: sippu@cs.helsinki.fi)

Edited by B. Seeger. Received: October 8, 2003 / Accepted: February 19, 2004
Published online: September 14, 2004 –

Abstract. In this paper we present new concurrent and recoverable B-link-tree algorithms. Unlike previous algorithms, ours maintain the balance of the B-link tree at all times, so that a logarithmic time bound for a search or an update operation is guaranteed under arbitrary sequences of record insertions and deletions. 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 guaranteed by key-range locking. The algorithms apply the write-ahead logging (WAL) protocol and the steal and no-force buffering policies for index and data pages. Record inserts and deletes on leaf pages of a B-link tree are logged using physiological redo-undo log records. Each structure modification such as a page split or merge is made an atomic action by keeping the pages involved in the modification latched for the (short) duration of the modification and the logging of that modification; at most two B-link-tree pages are kept X-latched at a time. Each structure modification brings the B-link tree into a structurally consistent and balanced state whenever the tree was structurally consistent and balanced initially. Each structure modification is logged using a single physiological redo-only log record. Thus, a structure modification will never be undone even if the transaction that gave rise to it eventually aborts. In restart recovery, the redo pass of our ARIES-based recovery protocol will always produce a structurally consistent and balanced B-link tree, on which the database updates by backward-rolling transactions can always be undone logically, when a physical (page-oriented) undo is no longer possible.

Keywords: B-link tree – Transaction – Concurrency control – Tree-structure modifications – Recovery

1 Introduction

The B-link-tree structure was introduced by Lehman and Yao [14]. A B-link tree is a B$\pm$tree in which the nodes on each level of the tree are linked from left to right. Each node of the B-link tree stores a high-key value that serves as an upper bound for the key values stored in the subtree rooted at that node. In the B-link-tree algorithms in [14], a search operation does not lock any nodes during the tree traversal, while an insert operation S-locks one node at a time. An insert operation releases its lock on a parent node before acquiring a lock on the node's child during tree traversal. When a leaf node is reached, the insert operation X-locks that node. In these algorithms, a split operation is executed in two phases: a half split followed by a complete split. The advantage of the link technique is that a search operation is compensated for a missed split of a node by following the right link of that node. The major problem with the B-link algorithms in [14] is that no provision is made for handling underflown nodes. The authors proposed that the B-link tree be rebalanced offline.

Sagiv [26] showed how to implement the B-link algorithms in [14] in such a way that search and insert operations lock only one node at a time. That is, a lock on a node is released before acquiring a lock on the node's child. The main contribution of the B-link algorithms in [26] is the introduction of an independent compression process. This process can run concurrently with search and insert operations, and it restructures the B-link tree by merging the underflown nodes. However, the algorithms in [26] suffer from the following drawbacks. First, it is not clear how often such a compression process should be run. The B-link tree may get out of balance and its performance may deteriorate before the compression process is run the next time. Second, a search process may end up in a wrong node because underflown nodes are merged in a nonuniform manner: entries are sometimes moved to the right, sometimes to the left. Also, inserters may end up trying to update a node far to the right. Third, merging of an underflown node by the compression process involves X-locking three nodes simultaneously on two adjacent levels of the tree.

Lanin and Shasha [13] presented B-link-tree algorithms that treat deletions symmetrically with insertions. These algorithms use the same locking protocol for search and insert operations as those in [26]. The merge operation in [13] is performed in two phases: a half-merge followed by a complete-merge. A half-merge X-locks two nodes at the same level of the tree simultaneously, while a complete-merge X-locks one node at a time. Due to the early-lock-releasing strategy followed in these algorithms, the complete-split and

complete-merge operations may encounter "inconsistent" situations when they try to update a parent node at a higher level of the tree. That is, a complete-split operation may find that the key value it is trying to insert into the parent node is already (still) there, and a complete-merge operation may find that a key to be deleted from the parent node does not exist there (yet) [29]. These algorithms have the drawback that a complete-split or a complete-merge may have to be restated repeatedly by traversing the tree from the root whenever such an operation reaches an inconsistent situation. Moreover, a node in the B-link tree may have a long chain of right sibling nodes that are not linked to their parent node. As a result, the tree balance may be lost and the search time may no longer be logarithmic in the number of database records stored in the tree.

The improvements to Sagiv's compression process presented in [3,5] guarantee a filling factor of at least 50% for B-link-tree nodes and avoid a search operation from reaching a wrong node. The periodic tree-restructuring process in [3] merges underflown nodes and performs node splits above the leaf-node level of the tree. An insert operation may split a node at the leaf-node level only. The periodic restructuring process in [3] cannot be run concurrently with search, insert, and delete operations. It is run asynchronously, and thus tree balance cannot be guaranteed at all times.

In [4], concurrent algorithms for doubly-linked B$\pm$trees (i.e., the tree nodes on each level are linked from left to right and from right to left) are presented. A search operation does not lock any nodes during the tree traversal, while updaters lock one at a time. A split operation X-locks two nodes at a time and never X-locks the newly allocated node, while a merge operation X-locks four nodes at most simultaneously. In these algorithms, the maintenance (tree balancing, garbage collection) is done by the insert and delete operations themselves. Many unrealistic assumptions, such as that the B-link-tree height never increases or decreases, are made in these algorithms.

Despite the enormous amount of work done in the area of concurrency control of B-trees in centralized systems [1–7, 11–14,16,17,19,23,24,26,27], only a few studies have considered recovery. In [2,6,7] recovery protocols for B$\pm$trees in centralized systems are sketched. Mohan and Levine [21] were the first to present detailed B$\pm$-tree concurrency-control and recovery protocols for centralized systems. The recovery protocol is based on ARIES [22]. Concurrency and recovery for B-link trees in centralized systems are considered by Lomet and Salzberg [16,17] and for generalized search trees by Kornacker et al. [11]. However, in the algorithms in [7,11,16,17, 21], a tree-structure modification interrupted by a transaction abort or a system failure always has to be rolled back (undone).

We summarize the main problems associated with the B-link algorithms in [3–5,13,14,26] as follows.

1. Variable-length keys cannot be dealt with.
2. When deleting the rightmost child of a parent node P in a B-link tree, the updating of the high-key value in P may propagate up the tree.
3. A split operation X-locks just the (full) node to be split and never X-locks the newly allocated node. This is not safe when a transaction contains key-range-scan operations (see [18]).
4. Tree-structure modifications (node splits or merges) are part of the transaction that triggered such a structure modification, which limits the degree of concurrency.
5. The concurrency control is at the node (page) level, and thus the degree of concurrency is very limited.
6. The transaction considered usually consists only of a single operation, and no key-range scans are included.
7. Empty nodes in the B-link tree cannot be deallocated immediately, and a node in the tree can have a long chain of right sibling nodes that are not linked to their parent node. Hence, the B-link tree may get out of balance, and its performance may deteriorate.
8. No recovery protocols to deal with transaction aborts or system failures are described.
9. No formal correctness proofs and no worst-case analyses are included.

The early-lock-releasing strategy followed in the B-link-tree algorithms in [13,14,26] can provide a high degree of concurrency. But the B-link tree can get out of balance due to the presence of many empty nodes and long chains of parentless nodes, and hence the tree performance may deteriorate over time. Moreover, designing recovery protocols for B-link trees to deal with transaction aborts and system failure could be quite complex and could constrain the degree of concurrency. Therefore, many studies [16,17,28–30] have proposed the use of lock coupling in B-link-tree algorithms to avoid the problems associated with the early-lock-releasing strategy. In [28,29], it is suggested that updaters hold an IS lock on a newly split or merged node of the B-link tree while acquiring an X lock on the appropriate parent node, i.e., lock coupling on the way up. In [30], updaters use lock coupling to propagate node splitting or merging up the B-link tree. In a study on real-time databases [8], lock-coupling algorithms were found to perform better than early-lock-releasing (link) algorithms [13,14,26].

Lomet and Salzberg's algorithms [17] are the most closely related to our paper. In the algorithms presented in [16,17], a tree-structure modification involving several levels of the tree is divided into smaller structure modifications (atomic actions). All structure modifications on levels of the tree above the leaf level are independent of database transactions and are of short duration. Lomet and Salzberg proposed that a tree-structure modification such as a node split or merge could be implemented as an atomic action using a separate database transaction, a special transaction, or as a "nested top action" (for nested top actions see [19–22]). Each atomic structure modification is logged using redo-undo log records. Searchers and updaters in [16,17] employ a latch-coupling protocol with S and U latches, respectively, where a U latch is compatible with an S latch but incompatible with an X latch and a U latch. When a page needs to be updated, the U latch on that page is upgraded to an X latch. The algorithms in [17] avoid many of the problems found in earlier B-link-tree algorithms. However, these algorithms still suffer from the following problems.

1. No key-range scans are included.
2. It is assumed that splitting index nodes requires only the node being split to be X-latched, but this is not safe when a transaction contains key-range-scan operations (see [18]).
3. Node merging or redistributing still needs three nodes to be X-latched on two adjacent levels of the tree.

4. A node split done on a single level of the tree is decoupled from the linking (posting) of the new child to its parent (i.e., the posting is never made unless a transaction follows a side pointer). Due to the decoupling, it is possible that arbitrarily long chains of sibling nodes are created that are not directly linked to their parents, and hence tree balance is not guaranteed. That is, no logarithmic bound for B-link-tree traversal is guaranteed.
5. Interrupted structure modifications always have to be rolled back (undone). This is inefficient because undoing a structure modification is expensive in terms of time, effort, and concurrency.
6. The proposed restart recovery protocol [15] is slow because two passes over the log have to be made during the undo phase of the restart recovery. In the first pass, all interrupted structure modifications are undone, and in the second pass, all aborted transactions are rolled back.

In this paper, we present new B-link-tree algorithms that improve concurrency, simplify recovery, and reduce the amount of logging. Our algorithms avoid all the problems in the previously published B-tree algorithms. In our algorithms, deletions are handled uniformly with insertions. A B-link-tree structure modification involving several levels of the tree is divided into smaller structure modifications (atomic actions). We present a new technique to implement a small structure modification such as a page split or a page merge as an atomic action, so that interrupted structure modifications are never rolled back (undone). Each structure modification X-latches and updates at most two pages on a single level of a B-link tree at a time for a short duration and is logged using a single redo-only log record. Hence, each successfully completed structure modification is regarded as a committed "nested top action" [19–22] and will not be undone even if the transaction that gave rise to it eventually aborts. Structure modifications can run concurrently with other structure modifications, whether on the same path or not. Each successfully completed structure modification brings the B-link tree into a structurally consistent and balanced state whenever the tree was structurally consistent and balanced initially.

The balance conditions of our B-link tree include that no level of the tree may contain two successive pages that are not both linked to their parents. This guarantees (in the worst case) that the search path of any database record is at most twice the height of the B-link tree. Record inserts and deletes on leaf pages of a B-link tree are logged using physiological redo-undo log records as in [7,17,19–22]. Our recovery protocol supports page-oriented redo, page-oriented undo (when possible), and logical undo. Recoverability from system failures of both the tree structure and the logical database is guaranteed because the redo pass of our ARIES-based [22] recovery protocol [9] will now produce a structurally consistent and balanced tree, thereby making possible the logical undo of record inserts and deletes. In our algorithms, structure modifications are never undone, which simplifies recovery and increases concurrency. Our algorithms work with the steal-and-no-force buffering policy for index and data pages and are deadlock free.

The paper is organized as follows. In Sect. 2 we present our database and transaction model. In Sect. 3 we introduce the definition of our balanced B-link tree. In Sect. 4 we introduce some basic concepts used in the paper. In Sect. 5 we present our new single-level structure-modification operations and logging in balanced B-link trees. In Sect. 6 we present our read-mode and update-mode traversal algorithms for the B-link tree. In Sect. 7 we present the fetch, insert, and delete algorithms. In Sect. 8 we describe page-oriented redo, page-oriented undo, and logical undo, and we present our undo-insert and undo-delete algorithms. In Sect. 9 we describe transaction execution and give general results that show the deadlock freedom of our algorithms and the maintenance of the structural consistency and balance of the B-link tree under concurrent transactions. In Sect. 10 we describe our ARIES-based restart recovery protocol [9] and show a recoverability result. In Sect. 11 conclusions are drawn.

2 Database and transaction model

We assume that our logical database D consists of *database records* of the form (k, x), where k 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, $\leq$, 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 $(\infty,$ null) which is never inserted or deleted. The database operations are as follows.

(1) **Fetch[k, θ u, x]:** Fetch the first matching record (k, x). Given a key value u $< \infty$, find the least key value k and the associated data value x such that k satisfies $k\theta u$ and the record (k, x) is in the database. Here θ is one of the comparison operators "$\geq$" or ">". To simulate the fetch-next operation [19–21] on a key range $[u, k]$, the fetch operation is used as follows. To fetch the first record in the key range, a transaction T issues Fetch[k_1, $\geq u$,x_1].To fetch the second record in the key range, T issues Fetch[k_2, $> k_1$,x_2].To fetch the third record in the key range, T issues Fetch[k_3, $> k_2$,x_3], and so on. The fetch operation Fetch[$k, \theta u, x$] *scans the key range* $[u, k]$ (if θ is "$\geq$") or $(u, k]$ (if θ is ">").

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

(3) **Delete[k, x]:** Delete the record with the key value k. Given a key value k, delete the record, (k, x), with key value k from the database if k appears in the database. If the database does not contain a record with key value k, 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. A *backward-rolling transaction* is of the form $B\alpha\beta A\beta^{-1}$, where $B\alpha\beta$ is a forward-rolling transaction, A denotes the *abort* operation, 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 nonempty operation string βo, where o is a single operation, is defined as $o^{-1}\beta^{-1}$. Here o^{-1} is the inverse of o, also denoted by undo-o. The inverses of our set of database operations are defined as follows.

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

A backward-rolling transaction $B\alpha\beta A\beta^{-1}$ thus denotes a transaction that has *undone* a suffix, β, of its forward-rolling phase, while the prefix, α, is still not undone. A transaction of the form $B\alpha A\alpha^{-1}R$, where R denotes the *rollback-completed* operation, is a *rolled-back transaction*. A forward-rolling or a backward-rolling transaction (that is, a transaction that does not contain the operation C or R) is called an *active transaction*, while a committed or rolled-back transaction is called a *terminated transaction*. A transaction is an *aborted transaction* if it is backward rolling or rolled back.

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* [25] 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 $H\gamma$ is a *completed history* for H.

3 Balanced B-link trees

Our B-link tree has a structure similar to a B-link tree [14] in that the pages on each level are linked from left to right and the link of the rightmost page at each level is set to null. That is, each interior page stores the Page-id of its successor page (on the same level) in its Page-link field. These links are called *sideways links*. We use our B-link tree as a sparse database index to the database so that the leaf pages store the database records. In addition, we assume the B-link tree is a unique index and that the key values are of variable length [10]. To handle variable-length key values, we assume that a database or an index record with the largest possible key value never occupies more than one sixth of the space in a data or index page.

Formally, a B-link tree is an array $B[0, \ldots, n]$ of *disk pages* $B[P]$ indexed by unique *page identifiers* (*Page-ids*) $P = 0, \ldots, n$. The page $B[P]$ with Page-id P is called *page* P for short. A page (other than page 0) is marked as *allocated* if that page is currently part of the B-link 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 nonleaf page and contains a list of *index records* of the form (k_1, P_1), (k_2, P_2), $\ldots$, (k_n, P_n), where k_1, k_2, $\ldots$, k_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 k_i in the index record (k_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 (k_1, x_1), (k_2, x_2), $\ldots$, (k_n, x_n), where $k_1, k_2, \ldots, k_n$ 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-link tree is (∞, null). The set of database records in the data pages of a B-link tree B is called the *database represented by* B and denoted by db(B).

A child page Q of a page P is a *direct child* of P if P contains a child link to Q. Otherwise, Q is an *indirect child* of P. The eldest (or leftmost) child is always a direct child. Indirect children of P can be accessed by first accessing some elder direct child of P and then following sideways links. The page R next to page Q is the *right sibling* of Q, and Q is the *left sibling* of R, if Q and R have the same parent. We even allow the root to have a right sibling page sideways linked to it.

In our B-link tree, each leaf page stores its high-key value so that for each leaf page it can be deduced whether or not that page has a right sibling that is an indirect child of its parent. The interior pages of the B-link tree do not store their high-key values because the current highest key value of an interior page can tell whether or not that page has a right sibling that is an indirect child of its parent (as a result of performing the structure modifications top-down in our approach). Hence, there is no need for an interior page to store its high-key value. For presentation consistency, we will use the term "high-key value" for an interior page P to mean the highest key value currently in P.

An example of such a B-link tree is shown in Fig. 1, where the data values of the database records in leaf pages are not shown. In this case, the root page $P1$ has no right sibling and each nonroot page is a direct child of its parent, except the leaf page $P8$, which is an indirect child of its parent $P3$.

A B-link tree is *structurally consistent* if it satisfies the basic definition of the B-link tree, so that each page can be accessed by following a child link to the eldest child and then following the sideways links, level by level. A structurally consistent B-link tree can contain underflown pages and chains of successive sibling pages that are indirect children of their parent. We say that a structurally consistent B-link tree is *balanced* if (1) none of its nonroot pages is underflown and (2) no indirect child page has a right sibling page that is also an indirect child.

We assume that each B-link-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

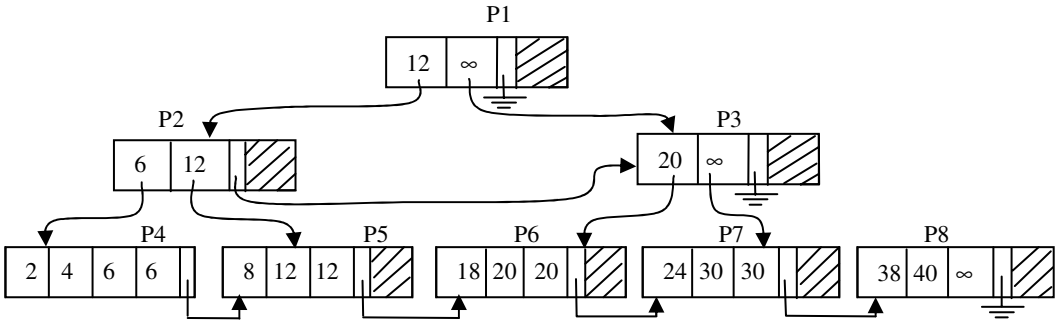

Fig. 1. A B-link tree

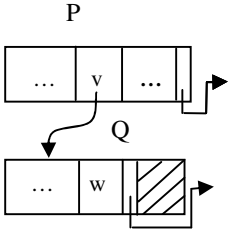

Fig. 2. Page P and its direct child Q

load factors for a nonroot leaf page and a nonroot index page, respectively. Choosing the loading factors m_1 and m_2 as above lets us avoid the problems associated with merge-at-half [9].

We say that a B-link-tree page P is *about to underflow* if (1) P is the root page and contains only one child link, (2) P is a nonroot leaf page and contains only m_1 database records, or (3) P is a nonroot index page and contains only m_2 child links. A B-link-tree page P is *about to overflow* if there is no room in it for the insertion of a record with maximum-length key.

We say that a B-link-tree page P is *safe* if one of the following conditions holds: (1) P is the only allocated page in the tree; (2) P is not about to underflow and not about to overflow, so that one record with a maximum-length key value can be inserted into P without causing P to overflow or one record can be deleted from P without causing P to underflow; or (3) P is not about to underflow (so that one record can be deleted), and both P and its right sibling (if one exists) are direct children of P's parent (so that P can be split if it cannot accommodate the insertion of a record).

As we shall see, the balance of a B-link tree will be maintained under updates by requiring that each update-mode traversal, when encountering an unsafe child page in the search path, turn it into a safe one. Doing this may cause the parent page to become unsafe, but the balance conditions of the B-link tree are still guaranteed to hold.

Let Q be a direct child of P and (v, Q) the index record in P associated with Q (Fig. 2). Q has a right sibling R that is an indirect child of P if and only if the highest key value w in Q is less than v. Here w is the key value associated with the last child link in Q if Q is a nonleaf page and the high-key value in Q if Q is a leaf page. This important property of our B-link tree is used in the update-mode traversal algorithm and in the repair-page-underflow algorithm (Sects. 5 and 6).

4 Latches, locks, logging, and buffering

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* [7,19–22] 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 [7,17,19–22]. A U latch, or an update-mode latch [16,17], is compatible with an S latch but incompatible with another U latch or an X latch. A U latch can be upgraded to an X latch, while an S latch cannot be upgraded. 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 [19–22]. 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. If a requested conditional lock cannot be granted immediately, then the requesting transaction is not made to wait; instead, the lock request returns with the exception "the lock cannot be granted", after which the requesting transaction continues its normal processing. Moreover, a lock may be held for different durations. 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 use *physiological logging* [7,19–22] as a compromise between physical logging and logical logging. We assume that the buffer manager employs the *steal* and *no-force* buffering policies [7,22], which do not put any restrictions on the buffer manager.

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 the type of the log record: "begin", "abort", "commit" or "rollback completed" (transaction-control log records), "insert" or "delete" (redo-undo log records for up-

dates in the forward-rolling phase), "undo-insert" or "undo-delete" (redo-only compensation log records for inverse operations in the backward-rolling phase), or "split", "link", "unlink", "merge", "redistribute", "increase-tree-height" or "decrease-tree-height" (redo-only log records for structure modifications).

Page-id(s): The identifier(s) of the page(s) to which updates of this log record were applied; present in redo-only, redo-undo, and CLRs.

LSN: The log sequence number of the log record (a monotonically increasing value).

Prev-LSN: The LSN of the preceding log record written by the same transaction.

Undo-Next-LSN: This field appears only in a CLR generated for a backward-rolling transaction. The Undo-Next-LSN is the LSN of the log record for the next update to be undone by the transaction.

Data: The record(s) inserted to or deleted from the page(s) and any additional information such as high-key value(s) and a page link that might be needed to redo or undo the log record.

A *redo-only* log record only contains redo information, while a *redo-undo* log record contains both redo and undo information. Therefore, any update logged by a redo-only or redo-undo log record is *redoable*, and any update logged by a redo-undo log record is *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 [7,19–22]. A CLR 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 (example 2, Sect. 10).

Each B-link-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. The recovery manager uses the Page-LSN field to keep track of the page's state. To guarantee a correct recovery, the *write-ahead-logging* (*WAL*) *protocol* [7,22] is applied. That is, an index page or a data page with Page-LSN n may be flushed onto the disk only after flushing first all log records with LSNs less than or equal to n.

Let $r = (k, x)$ be a database record. The redo-undo log records generated for the database operation insert$[k, x]$ and delete$[k, x]$ by transaction T in its forward-rolling phase are denoted by

$$n :< T,\ \text{insert},\ P,\ (k, x),\ m >,$$
$$n :< T, \text{delete},\ P,\ (k, x),\ m >,$$

respectively, where n is the LSN of the log record, P is the Page-id of the data page on which the update operation was performed, and m is the Prev-LSN of the log record. The LSN-value n is assigned to the Page-LSN field of page P.

The redo-only CLR generated for the inverse operations undo-insert$[k, x]$ and undo-delete$[k, x]$ by a backward-rolling transaction T in its backward-rolling phase are denoted by

$$n :< T,\ \text{undo-insert},\ P,\ k,\ m >,$$
$$n :< T, \text{undo-delete},\ P,\ (k, x),\ m >,$$

respectively, where n is the LSN of the log record, P is the Page-id of the data page on which the inverse operation was performed, and m is the Undo-Next-LSN of the log record. The LSN-value n is assigned to the Page-LSN field of page P. Note that to redo an undo-insert it is sufficient to record only the key value k of the database record (k, x) in the CLR.

In the next section we will define seven structure-modification operations that modify the structure of the B-link tree. Each of these operations is logged using a single redo-only log record that contains the Page-ids of the B-link-tree pages modified in the operation and the set of records moved from one page to another. The exact form of the log record for each operation is given in the algorithm for that operation in Sect. 5. For example, the operation split(Q), which splits a full page Q by allocating a new page Q' and moving half of the records from page Q to page Q', is logged as

$$n :< \text{split},\ Q,\ Q',\ M,\ V >,$$

where M is the Page-id of the storage-map page and V is the set of records moved from Q to Q'. The LSN value n is assigned to the Page-LSN fields of pages Q, Q', and M.

To keep track of active transactions and modified buffer pages, two main memory tables are used. The *active-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 log record written by the transaction, and, in the case of a backward-rolling transaction, (4) the *Undo-Next-LSN* field, which contains the LSN of the next log record to be processed during the rollback. 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 into 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 that 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 needed when the system crashes, the system takes *checkpoints* [1,7,22] periodically during normal processing. When a checkpoint is taken, a *checkpoint log record* is generated that includes copies of the contents of the transaction and modified-page tables. Taking a checkpoint involves no flushing of pages.

5 Single-level structure-modification operations and logging in the B-link tree

We handle B-link-tree structure modifications using an approach similar to that of [16,17], but we go further by making

a structure modification to involve only a single level of the tree and X-latching two pages at most on that level for short duration. Each structure modification is logged using a single redo-only log record, and hence, in the event of a system crash, either the log on disk contains no trace of the structure modification (and, accordingly, no trace of it in the database page on disk either), or the entire redo-only log record is found on disk. In the later case, the redo-only log record may have to be replayed during the redo pass of the restart recovery. Thus, each successfully completed structure modification is regarded as a committed "nested top action" [19–22] and will not be undone even if the transaction that gave rise to it eventually aborts. Also, an interrupted structure modification is never rolled back because it has no effects on the stable database, and hence recovery is simplified and concurrency increased. This in contrast to the algorithms in [7,11,16,17,20,21] in which an interrupted structure modification always has to be rolled back.

In our approach, each successfully completed structure modification brings the tree into a structurally consistent and balanced state whenever the tree was structurally consistent and balanced initially. Structure modifications can run concurrently with other structure modifications without the need for special tree latches. The execution of non-leaf-level structure modifications is separated from the execution of leaf-level updates that give rise to the structure modifications. For example, when a transaction T wants to insert (resp. to delete) a record r with key value k, T traverses the tree in update mode using the latch-coupling protocol [16,17] with U latches to reach the target leaf page Q and performs structure modifications along its search path, if there is a need for them (see the update-mode-traverse( ) algorithm in Sect. 6). To insert a record r into a full leaf page Q, T splits Q by moving the upper half of the records into a new leaf page Q' and then inserts r into the proper leaf page (Q or Q'). The action of linking Q' to its parent will be executed by the next transaction T' that traverses the same path in update mode and runs into Q.

The structure of a B-link tree can be modified by one of our seven structure-modification operations – *split*, *link*, *unlink*, *merge*, *redistribute*, *increase-tree-height*, and *decrease-tree-height*. Each operation when applied to a structurally consistent and balanced B-link tree performs a single update involving one or two pages on a single level of the tree, generates a single redo-only log record, and produces a structurally consistent and balanced B-link tree as a result.

To simplify the presentation of our algorithms, we use the following notations.

S-latch(P): Fix page P and acquire an S latch on P.
U-latch(P): Fix page P and acquire a U latch on P.
X-latch(P): Fix page P and acquire an X latch on P.
unlatch(P): Release the latch on page P and unfix P.
S-lock(r): Acquire an unconditional S lock on record r.
X-lock(r): Acquire an unconditional X lock on record r.
unlock(r): Release the acquired lock on record r.
upgrade-latch(P): Upgrade the U latch on page P to an X latch.
downgrade-latch(P): Downgrade the X latch on page P to a U latch.
allocate(P): Find some page P marked as unallocated in the storage-map page M and mark P as allocated in M.

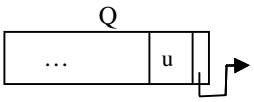

Fig. 3. Page Q is full

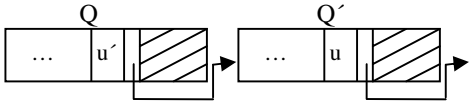

Fig. 4. Page Q is split at key value u' into Q and Q'

format(P): Format P as an empty B-link-tree page.
deallocate(P): Mark page P as unallocated in the storage-map page M.
log(n, R): Generate the log record R, place it in the log buffer, and return the LSN n of the log record.

The operation *split*(Q) is used to split a full page Q when Q is safe, that is, both Q and its right sibling (if one exists) are direct children of their parent (Fig. 3). It is assumed that Q is U-latched. The operation allocates a new page Q', moves the upper half of the records from Q to Q', and makes Q' a right sibling of Q. At the end of the operation, the parameter Q denotes the page (Q or Q') that covers the current search key value; this page remains U-latched while the latch on the other page is released (Fig. 4).

Split(Q) {
  upgrade-latch(Q); X-latch(M);
  allocate(Q'); X-latch(Q'); format(Q');
  u'= the key value that splits Q evenly;
  let V be the set of all records in Q with key values $> u'$;
  move the records in V from Q to Q';
  if (Q is a leaf page) high-record$(Q) = (u', Q')$;
  else Page-link$(Q) = Q'$;
  log(n, <split, Q, Q', M, V>); Page-LSN$(M) = n$;
  Page-LSN$(Q) = n$; Page-LSN$(Q') = n$; unlatch(M);
  if (Q covers the current search key value) {
    downgrade-latch(Q); unlatch(Q');
  } else {
    downgrade-latch(Q'); unlatch(Q); $Q = Q'$;
  }
}

The operation *link*(P, Q, R) is used to link the right sibling page R of page Q to the parent page P when Q is a direct child and R an indirect child of P (Fig. 5). The pages P and Q are assumed to be U-latched. It is assumed that page P can accommodate the insertion of the index record (v, R) and the change of the index record (v, Q) to (u, Q), where u is the high-key value of Q and v is the high-key value of R. At the end of the operation, P and Q remain U-latched (Fig. 6).

Link(P,Q,R) {
  upgrade-latch(P);
  insert the index record (v, R) into P and change the index record (v, Q) to (u, Q) in P;
  log(n, <link, P, (u, Q), (v, R)>);
  Page-LSN(P)= n; downgrade-latch(P);
}

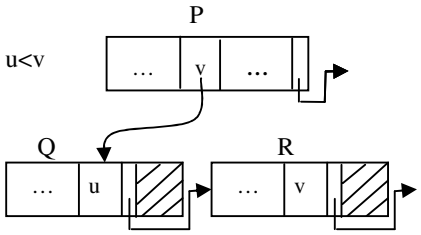

Fig. 5. Page Q has a right sibling R that is an indirect child of its parent P

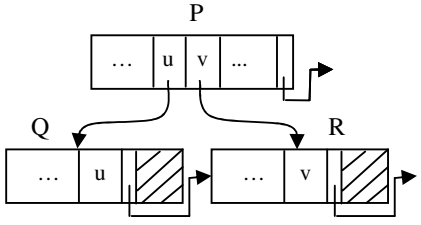

Fig. 6. The right sibling R of Q is linked to the parent P

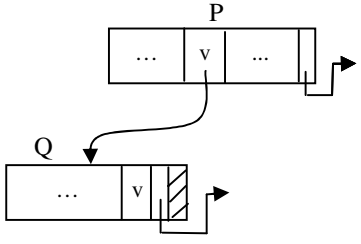

Fig. 7. Page R (Fig. 5) has been merged into its left sibling page Q

The operation *unlink*(P, Q, R) is used to unlink the right sibling page R of page Q from the parent page P when P is not about to underflow and the right sibling of R (if one exists) is not an indirect child (Fig. 6). The purpose of this operation is to make possible the merging or redistributing of Q and R. Pages P, Q, and R are assumed to be U-latched. The U latches on Q and R are needed to prevent another transaction from simultaneously splitting, merging (or redistributing) Q or R. At the end of the operation, the latch on P is released while the U latches on pages Q and R are retained (Fig. 5).

Unlink(P,Q,R) {
  upgrade-latch(P);
  delete the index record (v, R) from P;
  change the index record (u, Q) to (v, Q) in P;
  log(n, <unlink, P, (u, Q), (v, R)>);
  Page-LSN(P) $=$ n; unlatch(P);
}

The operation *merge*(Q, R) is used to merge page R into its left sibling page Q when R is an indirect child of the parent page and the contents of R and Q fit in a single page (Fig. 5). The pages Q and R are assumed to be U-latched. At the end of the operation, Q remains U-latched while the latch on R is released and R is deallocated (Fig. 7).

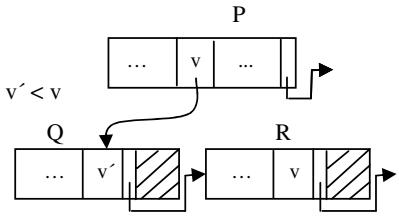

Fig. 8. Records in Q and R (Fig. 5) have been redistributed

Merge(Q,R) {
  upgrade-latch(Q); upgrade-latch(R); X-latch(M);
  if (Q is a leaf page) delete high-record(Q);
  else delete Page-link(Q);
  let V be the set of all records in R;
  move the records in V from R to Q; deallocate(R);
  log(n, <merge, Q, R, M, V>);
  Page-LSN(M) $=$ n; Page-LSN(Q) $=$ n;
  Page-LSN(R) $=$ n;
  unlatch(M); unlatch(R); downgrade-latch(Q);
}

The operation *redistribute*(Q, R) is used to redistribute the contents of page Q and its right sibling page R when R is an indirect child of the parent page but Q and R cannot be merged (Fig. 5). Pages Q and R are assumed to be U-latched. At the end of the operation, the U-latch on the page (Q or R) that covers the current search key value is retained while the latch on the other page is released, and the parameter Q is set to the page (Q or R) that covers the current search key value (Fig. 8).

Redistribute(Q,R) {
  upgrade-latch(Q); upgrade-latch(R);
  v' = the key value that redistributes records in Q and R;
  u = high-key(Q);
  if (Q is a leaf page) delete high-record(Q);
  else delete Page-link(Q);
  if ($v' > u$) { $Y = Q$;
    move all records with key values <= v' from R to Q;
  }else { $Y = R$;
    move all records with key values > v' from Q to R;
  }
  let V be the set of records moved;
  if (Q is a leaf page) high-record(Q) $=$ (v', R);
  else Page-link(Q) $=$ R;
  log(n, <redistribute, Q, R, V, Y>);
  Page-LSN(Q) $=$ n; Page-LSN(R) $=$ n;
  if (Q covers the current search key value) {
    downgrade-latch(Q); unlatch(R);
  } else {
    downgrade-latch(R); unlatch(Q); Q $=$ R;
  }
}

The operation *increase-tree-height*(P, P') is used to increase the height of the tree when the root page P has a right sibling page P' (Fig. 9). The pages P and P' are assumed to be U-latched. A new page P'' is allocated. P remains the root of the tree and is made the parent of P'' and P'. At the end of the operation, the latch on P is released, the U latch on the child

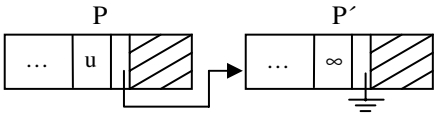

Fig. 9. Root page P has a right sibling P'

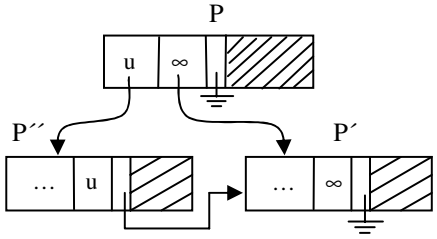

Fig. 10. Tree height is increased by one and P remains the root of the tree

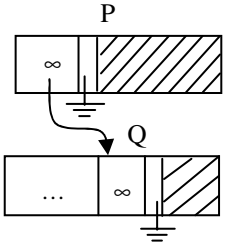

Fig. 11. Root page P has no right sibling and P has only one child Q

page of P (i.e., P' or P'') that covers the current search key value is retained, the latch on the other child page is released, and parameter P is set to the page (P' or P'') that covers the current search key value (Fig. 10).

Increase-tree-height(P, P') {
upgrade-latch(P); X-latch(M); allocate(P'');
X-latch(P''); format(P'');
let V be the set of all records in P;
move the records in V from P to P'';
insert index records (u, P'') and (∞, P') into P;
Page-link(P) = null;
log(n,<increase-tree-height, $P, P'', P', M, (u, P''), (\infty, P'), V >$);
Page-LSN(M) = n; Page-LSN(P) = n;
Page-LSN(P'') = n; unlatch(M); unlatch(P);
if (P'' covers the current search key value) {
downgrade-latch(P''); unlatch(P'); $P = P''$;
} else {
unlatch(P''); $P = P'$;
}
}

The operation *decrease-tree-height*(P, Q) is used to decrease the height of the tree when the root page P has only one child page, Q (Fig. 11). It is assumed that P and Q are U-latched and that P has no right sibling. At the end of the operation, P remains as the root of the tree, the U latch on P is retained, and the latch on Q is released and Q is deallocated (Fig. 12).

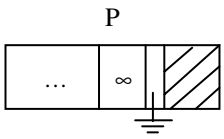

Fig. 12. Tree height is decreased by one and P remains the root of the tree

Decrease-tree-height(P,Q) {
upgrade-latch(P); upgrade-latch(Q); X-latch(M);
delete the index record (∞, Q) from P;
let V be the set of all records in Q;
move the records in V from Q to P; deallocate(Q);
log(n, <decrease-tree-height, P, Q, M, (∞, Q), V>);
Page-LSN(M) = n; Page-LSN(P) = n;
Page-LSN(Q) = n;
unlatch(M); unlatch(Q); downgrade-latch(P);
}

Lemma 1. *Let B be a structurally consistent and balanced B-link tree. Then any of the structure-modification operations link(P, Q, R), unlink(P, Q, R), split(Q), merge(Q, R), redistribute(Q, R), increase-tree-height(P, P'), and decrease-tree-height(P, Q), whenever the preconditions for the operation hold, produce a structurally consistent and balanced B-link tree when run on B.*

Proof. From the algorithms for the operations we see immediately that, if the preconditions for the operation are satisfied, then the operations indeed can be run on B and retain the structural consistency and balance of B. For link(P, Q, R) the preconditions state that the child page R is an indirect child of its parent P and that P has room for the child link (v, R) to be added, so that the operation can indeed be run on B. For unlink(P, Q, R), the structural consistency and balance of the resulting tree follows from the preconditions stating that P will not underflow if the child link (v, R) is deleted and that the right sibling of R (if any) is not an indirect child, so that the unlinking will not produce a chain of two successive indirect child pages. For split(Q), the preconditions state that Q is full and safe. As Q is full, the safety of Q implies that Q is not an indirect child and that the right sibling of Q (if any) is neither an indirect child nor a sibling of the root. Thus the splitting of Q will not create a chain of two indirect child pages. As Q is full, the two sibling pages created by the split are both guaranteed to be not about to underflow and not about to overflow, that is, both pages are safe. For merge(Q, R) and redistribute(Q, R), the preconditions state that R is a right sibling of Q and an indirect child of Q's parent. Thus the structural consistency of the tree clearly cannot be violated when Q and R are merged or redistributed. For increase-tree-height(P, P'), the preconditions state that P is the root of the tree and that P' is its right sibling. Since the contents of P are just moved to a new page P'' and the pages P'' and P' are linked as the only children of P, the result is a structurally consistent and balanced tree. For decrease-tree-height(P, Q), the preconditions state that Q is the only child of P. The contents of P are replaced by those of Q, thus producing a structurally consistent and balanced tree.

In a context of concurrent operations triggered by different transactions, the structural consistency and balance of the

resulting tree is guaranteed by the fact that the process performing a structure-modification operation keeps X-latched all the pages modified by the operation and keeps U-latched any additional page involved. During the modification of the parent page P in link(P, Q, R), P is kept X-latched and the direct child Q is kept U-latched. The right sibling R of Q need not be latched. Thus a simultaneous link(R, Q', R') or unlink(R, Q', R') may occur. However, the latch on Q prevents a simultaneous merge(Q, R) or redistribute(Q, R), which need both Q and R to be X-latched. The preconditions of split prevent a simultaneous split(R), and the balance conditions prevent the occurrence of a simultaneous merge(R, S) or redistribute(R, S).

During the modification of the parent page P in unlink(P, Q, R), P is kept X-latched and both direct child pages Q and R are kept U-latched. The latching of R is needed to prevent a simultaneous split(R), which would result in a chain of two indirect child pages.

The operation increase-tree-height(P, P') keeps P and the new page P'' X-latched and P' U-latched. The operation decrease-tree-height(P, Q) keeps both P and Q X-latched. The storage-map page is kept X-latched during the operations split(Q), merge(Q, R), increase-tree-height(P, P'), and decrease-tree-height(P, Q), thus guaranteeing the consistency of storage allocation and deallocation. □

When a transaction T traversing a B-link tree in update mode encounters an about-to-underflow child page Q of parent P, T executes the *repair-page-underflow*(P, Q) algorithm in order to merge or redistribute Q with its sibling. Pages P and Q are assumed to be U-latched, and P is assumed to be safe. Page Q can be either a rightmost or a nonrightmost child of P, and hence there are two cases to consider. When Q is not the rightmost child of P, then there are three subcases to consider depending on the current state of the right sibling page R of Q, that is, whether (1) R is an indirect child of P, (2) R is a direct child of P and has a right sibling page S that is an indirect child of P, or (3) R is a direct child of P and has a right sibling page S that is a direct child of P. When Q is the rightmost child of its parent P, then there are two subcases depending on the state of the left sibling page of Q, that is, whether (4) the left sibling page of Q is a direct child of P or (5) the left sibling page of Q is an indirect child of P.

At the end of the algorithm, the latch on the parent page P is released, the U latch on the child page (Q or its sibling) that covers the current search key value is retained and the latch on the other page is released, and the parameter Q is set to the page (Q or its sibling) that covers the search key value. Page Q is now safe.

```
Repair-page-underflow(P,Q) {
  if (high-key(Q) < high-key(P)) {
    /* Q is not the rightmost child of its parent P */
    let (v, Q) be the index record associated with Q in P;
    R = the Page-id of the right sibling of Q;
    U-latch(R); u = high-key(Q);
    if (u < v) { /* R is an indirect child of P */
                 /* Case 1, Fig. 13 */
      unlatch(P);
     if (Q and R can be merged) merge(Q, R);
     else redistribute(Q, R);
      return;
    } else { /* R is a direct child of P */
             /* Case 2 or Case 3 */
      let (w, R) be the index record associated with R in P;
      S = Page-id of the right sibling of R; v = high-key(R);
      if (v < w) { /*Case 2, Fig. 14*/
        if (P cannot accommodate the insertion of
        (w, S) or the change of (w, R) to (v, R)) {
        split(P);
        if (R is not a child of P anymore) {
          unlatch(R); restart the algorithm;
        }
       }
       link(P, R, S); unlink(P, Q, R); /*Fig. 16*/
       if (Q and R can be merged) merge(Q, R);
       else redistribute(Q, R); return;
       } else { /* Case 3, Fig. 15*/
         unlink(P, Q, R); /* Fig. 13 */
         if (Q and R can be merged) merge(Q, R);
         else redistribute(Q, R); return;
       }
    }
  }else { /* high-key(Q) = = high-key(P); hence Q    */
         /* is the rightmost child of its parent P; */
         /* Fig. 17 or Fig. 19                      */
    let (v, L) be the index record immediately preceding
    (w, Q) in P; unlatch(Q); U-latch(L);
    U-latch(the right-sibling page of L);
    if (high-key(L) = = v) { /* Case 4, Fig. 17 */
         /* the right sibling of L is Q */
         if (Q is no longer about to underflow) {
               unlatch(P); unlatch(L); return;
         }else {
           unlink(P, L, Q); /* Fig. 18 */
           if (L and Q can be merged) merge(L, Q);
           else redistribute(L, Q);
           Q = L; return;
         }
      } else { /* high-key(L) < v, Case 5, Fig. 19 */
               /* the right sibling of L is N */
        if (P cannot accommodate the insertion of
        (v, N) or the change of (v, L) to (u, L)) {
          split(P);
        }
        link(P, L, N); unlatch(L); U-latch(Q);
        if (Q is no longer about to underflow) {
          unlatch(P); unlatch(N); return;
        } else {
          unlink(P, N, Q);
          if (N and Q can be merged) merge(N, Q);
          else redistribute(N, Q); Q = N;
        }
      }
    }
  }
```

Lemma 2. *Let B be a structurally consistent and balanced B-link tree and let P and Q be pages of B such that P is safe, Q is about to underflow, Q is a direct child of P, P covers key value k, Q is the next page in the search path of k, and both P and Q are U-latched by the process that generates transaction T.*

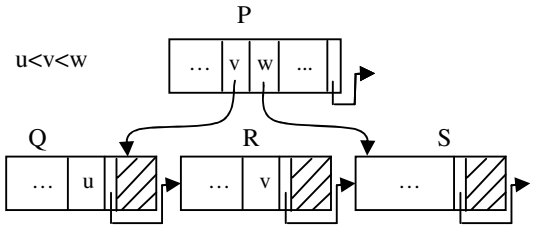

Fig. 13. The right sibling page R of an about-to-underflow page Q is an indirect child of its parent P

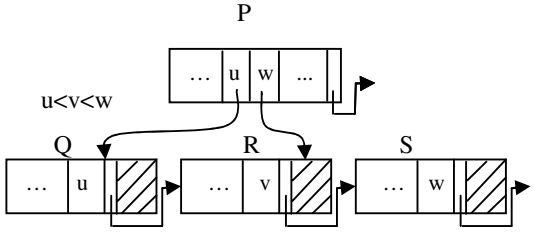

Fig. 14. The right sibling page R of an about-to-underflow page Q has a right sibling S, which is an indirect child of its parent P

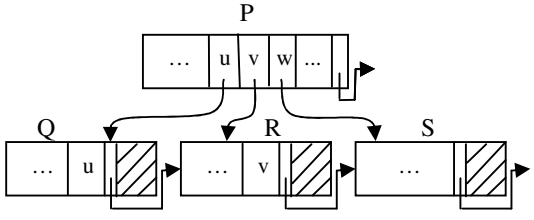

Fig. 15. The right sibling page R of an about-to-underflow page Q has a right sibling S, which is a direct child of its parent P

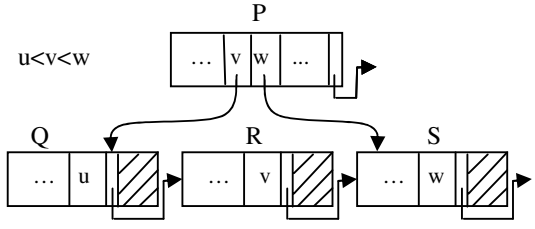

Fig. 16. Page S has been linked to its parent P and page R has been unlinked

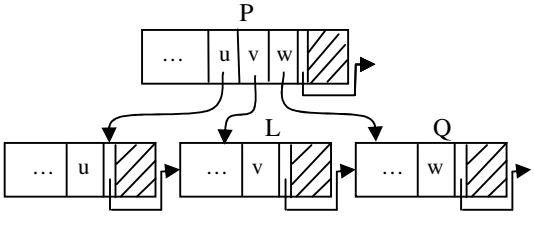

Fig. 17. The about-to-underflow page Q is the rightmost child of its parent page P and has a left sibling page L, which is a direct child of parent P

Then the repair-page-underflow(P, Q) algorithm, when run on B on behalf of T in search of key value k, generates a sequence of structure-modification operations such that, at the end of the algorithm, the resulting B-link tree is structurally consistent and balanced, the page denoted by the parameter P covers k,

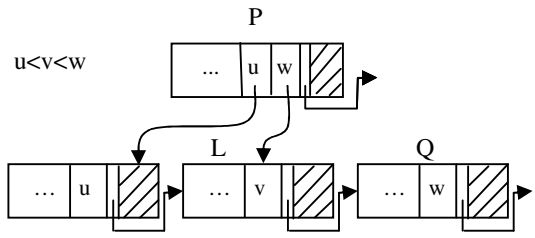

Fig. 18. The about-to-underflow rightmost child Q of P is unlinked from its parent

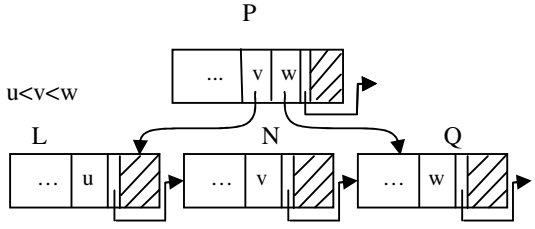

Fig. 19. The about-to-underflow page Q is the rightmost child of its parent page P and has a left sibling page N, which is an indirect child of parent P

and the page denoted by parameter Q is a safe direct child of P and is the next page on the search path for k.

Proof. The sequences of structure-modification operations generated by the repair-page-underflow(P, Q) algorithm in each of the five cases are given below, where R denotes the right sibling of Q (if any). We first assume that T runs the algorithm in isolation without any other transaction in progress.

(1) R is an indirect child of P (Fig. 13): merge or redistribute(Q, R). Page P is not modified and page Q is still on the search path for k because records are moved from R to Q and not vice versa.

(2) R is a direct child of P and has a right sibling page S that is an indirect child of P (Fig. 14): split(P) (if P is full), possibly followed by link(P, R, S); unlink(P, Q, R); merge or redistribute(Q, R). As P is assumed to be safe, it may be split into pages P and P', if it does not have enough room for the child link (w, S). Split(P) returns the one of pages P and P' that covers k. In any case, P remains safe after the operation link(P, R, S). Thus the tree is also guaranteed to remain structurally consistent and balanced after the unlink(P, Q, R) operation. The case thus reduces to case 1.

(3) R is a direct child of P and has a right sibling page S that is also a direct child of P (Fig. 15): unlink(P, Q, R); merge or redistribute(Q, R). As P is safe, it will not underflow when the child link (v, R) is deleted from P. The case reduces to case 1.

(4) Q is the rightmost child of P and the left sibling L of Q is a direct child of P (Fig. 17): unlink(P, L, Q); merge or redistribute(L, Q). As P is safe, it will not underflow when the child link (v, L) is deleted from P. At the end of the algorithm, the parameter Q is set to L, which is on the search path, as required. However, as Q is unlatched and latched again after latching L, it may happen that Q is no longer about to underflow. In that case the algorithm performs no structure modifications, and the parameter Q will stay as it is.

(5) Q is the rightmost child of P, the left sibling N of Q is an indirect child of P, and L (which then must be a

direct child of P) denotes the left sibling of N (Fig. 19): split(P) (if P is full); link(P, L, N); unlink(P, N, Q); merge or redistribute(N, Q). The reasoning proceeds as in case 2. We note that at the end of the operation, the parameter Q is set to N. However, as Q is unlatched and latched again after latching N, it may happen that Q is no longer about to underflow. In that case the algorithm does not perform the unlink(P, N, Q) and merge or redistribute(N, Q) operations, and parameter Q will stay as is.

Next we note that the above reasoning also holds in the context of concurrent operations by other transactions. This follows from the fact that the involved pages are all kept U-latched over the entire sequence of structure modifications. Case 1 is clear by Lemma 1 since it consists of only a single structure modification. In case 2, pages P, Q, and R are latched. Note that, if P is split, then the split(P) call leaves the page that covers k (P or its right sibling) U-latched. In case 3, pages P, Q, and R are latched. In case 4, pages P, L, and Q are latched when performing unlink(P, L, Q) followed by merge or redistribute(L, Q). In case 5, pages P, L, and N are latched while performing the possible split(P) followed by link(P, L, N), and pages P, N, and Q are latched while performing unlink(P, N, Q) followed by merge or redistribute(N, Q).

To prevent a deadlock and to decrease the number of page latches held simultaneously by the process that generates transaction T, the U latch held on Q (when the repair-page-underflow algorithm is entered) is released in cases 4 and 5, and Q is relatched after first latching its sibling L or siblings L and N. Because page P, which contains the child link to Q, is kept constantly latched, no other transaction can simultaneously deallocate Q or unlink it from its parent. After relatching Q, it is checked if Q is still about to underflow, and if not, the unlinking and merging/redistributing of Q is not done. Actually, this checking is unnecessary when the repair-page-underflow algorithm is only used in the update-mode traversal and when all transactions use the same update-mode traversal algorithm: it can be shown that other transactions cannot modify Q while Q is temporarily unlatched. □

6 B-link-tree traversals

To perform a Fetch$[k, \theta u, x]$ operation, a transaction T takes as input a key value u and uses the *read-mode-traverse*(u, P) algorithm to find the target leaf page P that covers the database record with key value u. The algorithm traverses the tree using the latch-coupling protocol with S latches and returns the Page-id P of the S-latched leaf page that covers u. If during the traversal T runs into a page that does not cover the search key value u, then T follows the sideways link to the next page on the same level.

```
Read-mode-traverse(u,P) {
  P = the Page-id of the root page of the tree; S-latch( P);
  do {
    v = high-key(P);
    if (u > v) {
      P' = the Page-id of the right sibling page of P;
      S-latch(P'); unlatch(P); P = P';
    }
    if (P is not a leaf page) {
      search P for the child page Q covering u;
      S-latch(Q); unlatch(P); P = Q;
    }
  } while (P is not a leaf page);
}
```

To perform a Insert$[k, x]$ or Delete$[k, x]$ operation, a transaction T takes as input the key value k and uses the *update-mode-traverse*(k, P) algorithm to find the target leaf page P that covers the database record with key value k. The algorithm traverses the tree using the latch-coupling protocol with U latches and returns the Page-id P of the U-latched leaf page that covers k. All unsafe pages on the search path are turned into safe ones.

```
Update-mode-traverse(k,P) {
  P = the Page-id of the root page of the tree; U-latch( P);
  if (P has a right sibling) {
    P' = the Page-id of the right sibling of P;
    U-latch(P'); increase-tree-height(P, P');
  }
  if (P is a leaf page) return;
  search P for the child page Q covering k; U-latch(Q);
  if (Q is the only child of P and Q has no right sibling) {
    decrease-tree-height(P, Q);
  }
  while (P is not a leaf page) {
    search P for the child page Q covering k;
    U-latch(Q);
    if (Q is about to underflow) {
      repair-page-underflow(P, Q); P = Q;
    } else {
      let (v, Q) be the index record associated with Q
      in P; u = high-key(Q);
      if (u < v) {
        /* Q has a right sibling page R which is an */
        /* indirect child of P, Fig. 5              */
        if (P cannot accommodate the insertion of (v, R)
        or the change of (v, Q) to (u, Q)) {
          split(P);
        }
        link(P, Q, R);
      }
      if (Q covers k) {
        unlatch(P); P = Q;
      } else {
        R = the Page-id of the right sibling of Q;
        unlatch(P); U-latch(R); unlatch(Q); P = R;
      }
    }
  }
}
```

Lemma 3. *Let B be a structurally consistent and balanced B-link tree. Assume that a transaction T performs an update-mode traversal on B. Then the tree produced is structurally consistent and balanced, and the leaf page covering the search key value is safe.*

Proof. The claim follows from Lemmas 1 and 2 by induction on the height of B because the update-mode traversal algorithm, when advancing from one page to the next on the search path, always turns an unsafe child page into a safe one, possibly thereby causing the parent to become unsafe but never leaving behind a page that would violate the balance conditions. First, at the top level of the tree, the increase-tree-height(P, P') structure modification is performed if the root page has a right sibling, or the decrease-tree-height(P, Q) structure modification is performed if the root page has only one child and no right sibling. Whenever the traversal proceeds from a page P to a direct child page Q that is about to underflow, the repair-page-underflow(P, Q) algorithm is called. Whenever the traversal proceeds from a page P to a direct child page Q that is not about to underflow but has a right sibling R that is an indirect child of P, the split(P) (if needed) and link(P, Q, R) structure modifications are called. At each step, we may assume as an induction hypothesis that page P on the previous level on the search path is safe, thus satisfying a required precondition of the structure modifications. □

Lemma 4. *Let B be a structurally consistent and balanced B-link tree of height h. Any read-mode traversal on B on behalf of transaction T accesses at most $2h$ pages of B and keeps at most two of those pages S-latched at a time. Any update-mode traversal on B on behalf of T accesses at most $4h$ pages of B and keeps at most two of those pages X-latched and at most two U-latched at a time. In addition, the storage-map page may be accessed and X-latched h times during an update-mode traversal.*

Proof. The result for a read-mode traversal follows immediately from the definition of a structurally consistent and balanced B-link tree and from the latch-coupling protocol that uses S latches. The worst case of accessing $2h$ pages occurs when at each level the sideways link to an indirect child page has to be followed.

The number of pages accessed by an update-mode traversal follows from the fact that in the worst case the number of pages accessed at one level of the tree is four. This happens in the repair-page-underflow(P, Q) algorithm (case 5) when the U latch on the rightmost child page Q of the parent P is released temporarily and the two sibling pages (L and N) of Q are U-latched and Q is relatched. In the increase-tree-height(P, P') operation the number of pages accessed is three, including the new page P''. The storage-map page is accessed when a new page needs to be allocated in the split(Q) and increase-tree-height(P, P') operations and when a page needs to be deallocated in the merge(P, R) and decrease-tree-height(P, Q) operations.

The number of U and X latches held simultaneously in each of the structure modifications is evident from the algorithms (also see the proof of Lemma 2). The link(P, Q, R) operation X-latches the parent page P while keeping the child page Q U-latched simultaneously. The unlink(P, Q, R) operation X-latches the parent page P while keeping both the Q and R child pages U-latched simultaneously. The split(Q) operation X-latches Q and the new (right sibling) page Q'. The merge(Q, R) and redistribute(Q, R) operations X-latch the sibling pages Q and R. The increase-tree-height(P, P') operation X-latches the root page P and the new page P'' and keeps U-latched the right sibling page P'. The decrease-tree-height(P, Q) operation X-latches the root page P and its child page Q.

The repair-page-underflow(P, Q) algorithm at any phase of its execution keeps at most two pages X-latched and two pages U-latched simultaneously. The worst case occurs in case 5 when the parent page P is split. In fact we could easily do with fewer latches here if we release the latches on the child pages for the duration of the split(P) operation and, after that is done, we reacquire the latches on the child pages. □

Lemma 5. *Read-mode traversals and update-mode traversals are deadlock free.*

Proof. A transaction performing a read-mode traversal acquires S latches on the search path in a top-down, left-to-right order, so that a nonroot page is latched only after latching either the parent or left sibling page first. A transaction performing an update-mode traversal acquires U latches on the search path, and on all the adjoining pages that need a structure modification, in a top-down, left-to-right order. Note that to achieve this property in the repair-page-underflow(P, Q) algorithm it is necessary to release the U latch on Q when Q is the rightmost child of its parent P and the left sibling of Q must be latched. As only U latches are ever upgraded to X latches, we may thus conclude that our page-latching protocol is deadlock free. □

7 B-link-tree fetch, insert, and delete

When the concurrency control is performed at the record level, transactions rely on record locking. However, S-locking only the found record and X-locking only the inserted (or deleted) record for commit duration does not guarantee repeatable reads when key-range fetch operations are present. 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 [7, 19–21].

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 completed its rollback.

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.

The Fetch$(T, k, \theta u, x)$ algorithm implements the Fetch$[k, \theta u, x]$ database operation. Given a key value $u < \infty$, the operation fetches the database record $r = (k, x)$ with the least key value k satisfying $k\theta u$. The fetched record r is S-locked for transaction T for commit duration.

Fetch(T,k, θ u,x){
if (the Page-id P of a leaf page has been saved as a result of a previous call to Fetch( )) {
/* guess that the record to be fetched resides in the */
/* same page as the previously fetched record */
S-latch(P);
if (P is not a leaf page of the B-link tree any more or u is less than the least key value in P or u is greater than the greatest key value in P) {
unlatch(P);
}
} else {
L1: read-mode-traverse(u, P);
}
Q = null;
L2: let v be the greatest key value in P;
if ($u > v$ or $u == v$ and θ ==">") {
/* the record to be fetched resides in P', */
/* the page next to P */
P'= the Page-id of the page next to P;
S-latch(P'); $Q = P$; $P = P'$;
save Page-LSN(Q); unlatch(Q);
}
L3: ;
/* now P contains the record to be fetched */
determine the record r in P with the least key value k satisfying $k\theta$ u;
request a conditional S lock on r;
if (the S lock is granted right away) {
save Page-id(P); unlatch(P); return with r;
}
if (r is the lowest record in P and Q ! = null) goto L5;
save Page-LSN(P); unlatch(P);
/* request an unconditional S lock on r */
S-lock(r); S-latch(P);
L4: ;
if (Page-LSN(P) has not changed) {
save Page-id(P); unlatch(P); return with r;
}
if (P does not cover r or P is not a leaf page or P is not part of the B-link tree any more) {
unlatch(P); unlock(r); goto L1;
}
if (r is present in P and the key value k of r is still the least key value in P satisfying $k\theta$ u) {
save Page-id(P); unlatch(P); return with r;
} else {
unlock(r); goto L2;
}
L5: ;
/* 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 Page-LSN(P); unlatch(P); S-lock(r);
S-latch(Q); S-latch(P);
if (Page-LSN(Q) has not changed) {
unlatch(Q); goto L4;
}
if (Q does not cover u or Q is not a leaf page or Q is no longer part of the B-link tree) {
unlatch(Q); unlatch(P); unlock(r); goto L1;
}
/* Page-LSN(Q) has changed but Q still covers u */
let v be the greatest key value in Q;
if ($u == v$ and θ == " >=") {
unlatch(Q); unlatch(P);
$P = Q$; save Page-id(P);
return with the record with key value v;
}
if ($u == v$ and θ ==">") {
unlatch(Q); goto L4;
}
/* now $u < v$ in Q */
unlatch(P); unlock(r); $P = Q$; goto L3;
}

The Insert(T, k, x) algorithm implements the Insert$[k, x]$ database operation in the forward-rolling phase of transaction T. The given record $r = (k, x)$ is inserted into the database. The insertion is redo-undo-logged for transaction T, and the inserted record r is X-locked for T for commit duration.

Insert(T,k,x) {
update-mode-traverse(k, P);
if (P is full) split(P);
upgrade-latch(P);
P' = the Page-id of the page that holds the record r' with the least key value greater than k;
/* thus P' is P (when r' is found in P) or */
/* the page next to P (otherwise) */
X-latch(P'); lock-records(T, P, r, P', r');
if (the exception ''locks cannot be granted'' is returned) {
restart the insert operation;
}
search P for the position of insertion;
if (a record with key value k is found) {
terminate the insert operation; release the latches;
return with the exception ''uniqueness violation'';
} else {
insert r into P;
log(n, <T, insert, P, (k, x), Last-LSN(T)>);
Page-LSN(P)= n; Last-LSN(T)= n;
unlatch(P); unlatch(P'); unlock(r');
hold the X lock on r for commit duration;
}
}

The Delete(T, k, x) algorithm implements the Delete$[k, x]$ database operation in the forward-rolling phase of transaction

T. Given a key value k, the algorithm deletes the record (k, x) with key value k from the database. The deletion is redo-undo-logged for transaction T, and the record next to the deleted record is X-locked for T for commit duration.

```
Delete(T,k,x) {
 update-mode-traverse(k, P); upgrade-latch(P);
 P' = the Page-id of the page that holds the record r' with
    the least key value greater than k;
 /* thus P' is P (when r' is found in P) or */
 /* the page next to P (otherwise) */
 X-latch(P'); lock-records(T, P, r, P', r');
 if (the exception ''locks cannot be granted'' is returned) {
    restart the delete operation;
 }
 search P for a record with key value k;
 if (no record with key value k was found in P) {
   terminate the delete operation; release the latches;
   return with the exception ''record not found'';
 } else {
   delete r from P;
   log(n, <T, delete, P, (k, x), Last-LSN(T)>);
   Page-LSN(P)= n; Last-LSN(T)= n;
   unlatch(P); unlatch(P'); unlock(r);
   hold the X lock on r' for commit duration;
 }
}
```

The Lock-records(T, P, r, P', r') algorithm is used by transaction T to acquire X locks on record r and the record r' next to r when inserting or deleting 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 (in ascending key order). The algorithm acquires X locks on r and r' for T, if possible. Otherwise, it unlatches pages P and P' and returns with the exception "the locks cannot be granted".

```
Lock-records (T,P,r,P',r') {
 k = the key value of r; request conditional X
 locks on the records r in P and r' in P';
 if (the X locks are granted right away) return;
 save Page-LSN(P) and Page-LSN(P');
 unlatch(P); unlatch(P');
 /* try again by requesting unconditional X locks */
 X-lock(r); X-lock(r'); X-latch(P); X-latch(P');
 if (Page-LSN(P) and Page-LSN(P') have not changed) {
   return;
   }
   if (Page-LSN(P) has changed) {
      if (P does not cover r or P is not a leaf page or P
      is no longer part of the B-link tree) {
         goto out;
      }
   }
   /* now P still covers r */
   search P for a record r'' with the least key value > k;
   if (no such record is found in P) {
     if (P == P') goto out;
     else P'' = the Page-id of the page next to P;
   } else { /* now P contains r'' */
     if (P == P') {
       if (the key values of r'' and r' are equal) {
         /*the record r' next to r did not change*/
         return;
       } else {
         unlock(r); unlock(r'); r' = r'';
         restart the algorithm from the beginning;
       }
     } else {
       unlatch(P'); unlock(r); unlock(r'); P' = P; r' = r'';
       restart the algorithm from the beginning;
     }
   }
   if (P'' == P' and the key value of r' is equal to the
   key value of the first record in P') {
      return;
   }
   out: ;
   unlatch(P); unlatch(P'); unlock(r); unlock(r');
   return with exception ''the locks cannot be granted'';
}
```

Example 1. Let $T1$ and $T2$ be two transactions where $T1$ wants to delete a database record with key value 35 while $T2$ wants to insert a database record with key value 40 into the database represented by the B-link tree shown in Fig. 20. Assume that $T1$ starts first and U-latches the root page P of the B-link tree and traverses the tree to reach the leaf page $P4$.

Now the high-key value in $P4$ is less than the key value of the associated index record $(\infty, P4)$ in the parent page P. Hence, $T1$ must execute the link operation in order to make $P5$ a direct child of its parent. For that purpose, the root page P, which is full, must first be split. The split(P) and link$(P', P4, P5)$ operations result in the B-link tree shown in Fig. 21.

$T1$ now upgrades its U latch on $P4$ to an X latch and acquires X locks on the record to be deleted and the next record in $P4$, that is, on the records with key values 35 and 46. Then $T1$ deletes the record with key value 35 from $P4$, releases the X latch on $P4$ and the X lock on the deleted record, and holds the X lock on the record with key value 46 for commit duration.

When $T2$ starts and U-latches the root page P of the B-link tree of Fig. 21, it finds that the Page-link(P) != null (i.e., P has a right sibling). Hence, $T2$ executes the increase-tree-height(P, P') operation and traverses down the tree to $P4$. $T2$ upgrades its U latch on $P4$ to an X latch and acquires X locks on the record to be inserted and the next record in $P4$, that is, on the records with key values 40 and 46. When the record locks are granted, $T2$ inserts the record with key value 40 into $P4$, releases the X latch on $P4$ and the X lock on the (next) record with key value 46, and holds the X lock on the inserted record for commit duration. This results in the B-link-tree of Fig. 22.

8 Page-oriented redo and undo and logical undo

Page-oriented redo (or *physical redo*) means that, when a page update needs to be redone at recovery time, the Page-id fields of the log record are used to determine uniquely the affected pages. That is, no other page needs to be accessed or examined. Similarly, *page-oriented undo* (or *physical undo*) means that, when a page update needs to be undone during transaction rollback, the Page-id fields of the log record are used to determine the affected pages. The pages are accessed and the update is undone on these pages. 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) [22] 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 CLR 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* [20, 21] is used: the backward-rolling transaction retraverses the B-link 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 Undo-insert(T, P, k, m) algorithm implements the Undo-insert$[k, x]$ inverse operation in the backward-rolling phase of an aborted transaction T. Given is the Page-id P of the leaf page on which the record $r = (k, x)$ was inserted during the forward-rolling phase of T, the key value k of r, and the Prev-LSN m of the log record generated by T for the insertion. The algorithm deletes r from P if r is still there. Otherwise, the insertion of r is undone logically. That is, T retraverses the B-link tree in update mode to reach the leaf page Q that currently covers k and deletes r from there. The undone insertion is redo-only-logged for T.

Undo-insert(T,P,k,m) {
 X-latch(P);
 if (P still contains r and will not underflow if r is deleted) {
  $Q = P$;
 } else {
  unlatch(P); update-mode-traverse(k, Q);
  upgrade-latch(Q);
 }
 delete r from Q;
 log(n, <T, undo-insert, Q, k, m>);
 Page-LSN(Q) = n; Undo-Next-LSN(T) = m; unlatch(Q);
}

The algorithm Undo-delete$(T, P, (k, x), m)$ implements the inverse operation Undo-delete$[k, x]$ in the backward-rolling phase of an aborted transaction T. Given is the Page-id P of the leaf page from which the record $r = (k, x)$ was deleted during the forward-rolling phase of T, the record r, and the Prev-LSN m of the log record generated by T for the deletion. The algorithm reinserts r into P if P still covers k. Otherwise, the deletion is undone logically. That is, T retraverses the B-link tree in update mode to reach the leaf page Q that currently covers k and inserts r into Q. The undone deletion is redo-only-logged for T.

Undo-delete(T,P,(k,x),m) {
 X-latch(P);
 if (P still covers r and there is a room for r in P) {
  $Q = P$;
 } else {
  unlatch(P); update-mode-traverse(k, Q);
  if (Q is full) split(Q);
  upgrade-latch(Q);
 }
 insert r into Q;
 log(n, <T, undo-delete, Q, (k, x), m>);
 Page-LSN(Q) = n; Undo-Next-LSN(T) = m; unlatch(Q);
}

Lemma 6. *Let B be a structurally consistent and balanced B-link tree of height h and let T be a transaction. Any Fetch$[k, \theta u, x]$ operation by T on B accesses at most $2h + 1$ pages of B and keeps at most two of those pages S-latched for T at a time. Any Insert$[k, x]$ or Delete$[k, x]$ operation in the forward-rolling phase of T and any logically implemented inverse operation Undo-insert$[k, x]$ or Undo-delete$[k, x]$ in the backward-rolling phase of T accesses at most $4h$ pages of B, keeps at most two of those pages X-latched and at most two U-latched for T at a time, and produces a structurally consistent and balanced B-link tree. Any page-oriented inverse operation Undo-insert$[k, x]$ or Undo-delete$[k, x]$ in the backward-rolling phase of T accesses at most one page of B and produces a structurally consistent and balanced B-link tree.*

Proof. The claim for Fetch$[k, \theta u, x]$ follows immediately from Lemma 4 and the algorithm for the Fetch operation. The worst case of $2h + 1$ pages occurs when at each level the sideways link to an indirect child page has to be followed and when reaching the leaf level an extra step in the sideways-link chain is needed to locate the least key k satisfying $k\theta u$. The claims for Insert$[k, x]$, Delete$[k, x]$, Undo-insert$[k, x]$, and Undo-delete$[k, x]$ follow from Lemma 4 and the algorithms for the operations. By Lemma 3, the leaf page reached by an update-mode traversal is safe. Thus it can be split, if needed, into Insert$[k, x]$ and a logical Undo-delete$[k, x]$, and it will not underflow in Delete$[k, x]$ or in a logical Undo-insert$[k, x]$. A page-oriented Undo-insert$[k, x]$ always checks that the page will not underflow from the deletion of (k, x), and a page-oriented Undo-delete$[k, x]$ always checks that the page has room for (k, x). □

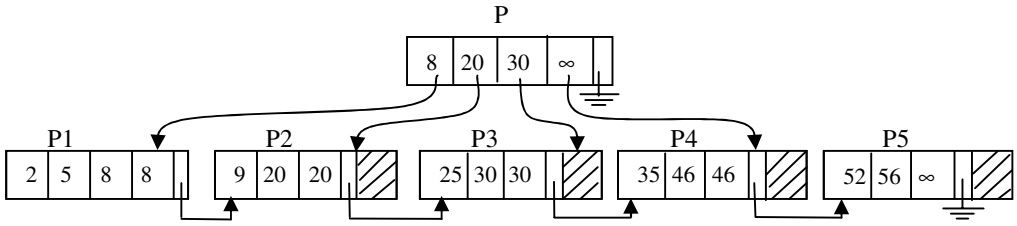

Fig. 20. An initial B-link tree, where leaf page $P5$ is an indirect child of its parent P

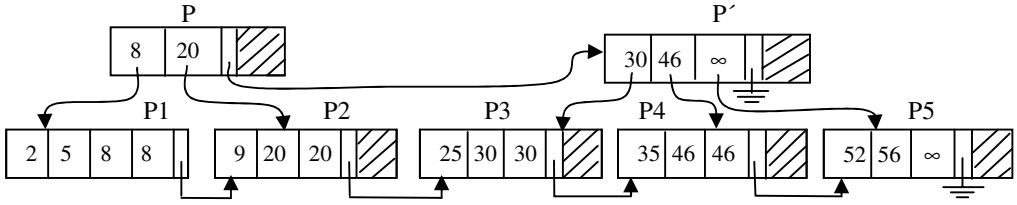

Fig. 21. The linking of indirect child $P5$ to its parent caused the split of root page P into P and P'

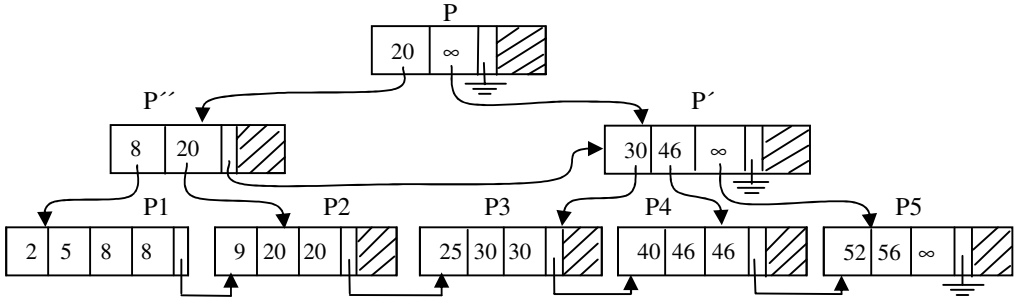

Fig. 22. The height of the B-link tree is increased by one

9 Transaction execution

Using the algorithms given in the previous sections, a transaction is executed as follows:

```
Execute-transaction( ) {
 begin(T);
 rollforward(T);
 if (T is to be committed) {
   commit(T);
 } else {
   abort(T);
   rollback(T, Undo-Next-LSN(T));
   rollback-completed(T);
 }
}
```

The call **begin**(T) generates a new transaction ID T, generates the log record $<T$, begin$>$, and inserts T into the active-transaction table as a forward-rolling transaction with Last-LSN = the LSN of $< T$, begin$>$. The call **commit**(T) generates the log record $<T$, commit$>$, flushes the log, releases all locks held by T, and removes T from the active-transaction table. The call **abort**(T) changes T to a backward-rolling transaction in the active-transaction table with Undo-Next-LSN(T) = Last-LSN(T) and generates the log record $< T$, abort$>$. The call **rollback-completed**(T) generates the log record $<T$, rollback-completed$>$, flushes the log, releases all locks held by T, and removes T from the active-transaction table. The actions contained in **rollforward**(T) represent the forward-rolling phase of transaction T and are determined by the application process that is generating T. The forward-rolling phase contains zero or more calls of the algorithms Fetch(), Insert(), and Delete(), with different arguments. The call **rollback**(T, n) performs the backward-rolling phase of T, when n = Undo-Next-LSN(T):

```
Rollback(T,n) {
 get the log record r with LSN = n;
 while (r is not <T, begin>) {
  if (r is <T, insert, P, (k, x), m>) {
    Undo-insert(T, P, k, m);
  } else if (r is <T, delete, P, (k, x), m>) {
    Undo-delete(T, P, (k, x), m);
  }
  get the log record r with LSN = m;
 }
}
```

In Lemma 5, we have shown that read-mode traversals and update-mode traversals are deadlock free. Thus we conclude that no deadlocks can occur between page latches. No deadlocks can occur between record locks, either, between single executions of the algorithms Fetch(), Insert(), Delete(), Undo-insert(), and Undo-delete(). Note that S locks on records are never upgraded and that a Fetch() call locks only one record, an Insert() call or a Delete() call locks two records in ascending key order, and the calls Undo-insert() and Undo-delete()

do not acquire record locks at all. No deadlock can be caused by the interaction of record locks and page latches because no transaction is made to wait for a lock on a record while holding a latch on some page. For transactions containing multiple fetch, insert, or delete operations in their forward-rolling phase, a deadlock can occur only if two transactions operate on two records in reverse key orders. We have:

Theorem 7. *Assume that each transaction in its forward-rolling phase accesses records in ascending key order. Then no deadlocks can occur when the concurrent transactions are executed using the Execute-transaction() algorithm given above.* □

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-link tree with $db(B1) = D1$. Let H' be a string of record-fetch, record-update, page-read, page-modification, and transaction-control actions executed on $B1$ by a set of concurrent processes generating transactions using the Execute-transaction() algorithm given above. Here a record-fetch action consists of reading a database record from an S-latched data page, a record-update action consists of inserting or deleting a database record in an X-latched data page or undoing the insertion or deletion of a database record in an X-latched data page, a page-read action consists of inspecting the contents of a latched page, a page-modification action consists of modifying the contents of an X-latched page in a structure modification, and a transaction-control action consists of writing a log record of type begin, commit, abort, or rollback-completed into the log buffer. We say that H' is an *implementation* of H on $B1$ if the sequence of record-fetch, record-update, and transaction-control actions in H' represents H.

Theorem 8. *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-link tree with $db(B) = D$, and let H' be an implementation of H on B. Then H' produces a structurally consistent and balanced B-link tree.* □

Proof. A formal proof would use induction on the number of actions in H'. For the purpose of the proof we may regard each sequence of page-modification actions generated by one of the structure-modification operations split(), link(), unlink(), merge(), redistribute(), increase-tree-height(), or decrease-tree-height() as an atomic action. This is because all the pages modified in a structure modification are kept simultaneously X-latched during the modification and the logging of that modification. Also, the action of inserting a record (k, x) into a leaf page P, or the action of deleting a record (k, x) from a leaf page, or the undoing of such an insertion or a deletion is an atomic action because the leaf page in question is kept X-latched during the update and the logging of that update. The structural consistency and balance of the tree produced by H' thus follows from Lemmas 1 and 6. □

Theorem 9. *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-link tree with $db(B) = D$. Then there exists an action string H' that implements H on B. Moreover, the implementation of each action in H' includes at most one traversal of B.*

Proof. In one such H', the implementations of individual operations Fetch$[k, \theta u, x]$, 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. The worst case, when the greatest number of tree traversals is needed, occurs when each inverse operation in the backward-rolling phase of any aborted transaction is implemented logically in H'; this includes a single traversal of the tree from the root down to the leaf page that currently covers the record insertion or deletion being undone. No latch waits can occur during the traversals at any level of the tree because all U and X latches on pages are only held for the time of the record-update or page-modification action in question and because any S latch on a page is released as soon as the next page in the search path has been latched (when latch-coupling down the tree) or as soon as a record has been fetched from the page (when performing a record-fetch action on a leaf page). As H can be run on D under the key-range locking protocol, no record-lock waits can occur in H' either. □

10 Restart recovery

We use a redo and undo recovery protocol for handling the system failures. Our restart recovery protocol supports page-oriented redo, page-oriented undo (when possible), logical undo, concurrency control at the record level, and the steal-and-no-force policies for buffer management. The restart recovery protocol is based on ARIES [22] and consists of three passes: an analysis pass, a redo pass, and an undo pass. But because the redo pass of our restart recovery always produces a structurally consistent and balanced B-link tree, new transactions could be admitted to the system as soon as the redo pass has been completed and the needed X locks on records have been reacquired (from the log) for the backward-rolling transactions. (However, to make possible the reacquisition of the X lock on the record r' of a deleted record r, the lock name (a hash value computed from the key) of r' should also be included in the log record generated for Delete$[r]$.)

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. The modified-page table of pages that were potentially more up to date in the buffer than in stable storage and the active-transaction table of transactions that were active (i.e., forward-rolling or backward-rolling) at the time of the crash are reconstructed 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 in the reconstructed modified-page table. In other words, the analysis pass determines the starting point of the redo pass in the log and the list of the transactions that need to be rolled back or whose rollback needs 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. If the modified-page table is empty, then the redo pass is

skipped. Otherwise, for each redoable log record, the following operation is performed. If the page P mentioned in the log record is not in the modified-page table or P is there and the Rec-LSN(P) is greater than the log record's LSN, then the logged update does not require redo. Otherwise, page P mentioned in the log record is X-latched and Page-LSN(P) is compared with the log record's LSN in order to check whether or not page P already contains the update, that is, whether or not the updated page P was written to the disk before the system failure. If Page-LSN(P) is less than the log record's LSN, then the logged update is redone, that is, applied physically (in a page-oriented fashion) to page P and Page-LSN(P) is set to the log record's LSN (without performing any logging). Otherwise, Rec-LSN(P) in the modified-page table is set to Page-LSN(P) + 1 (i.e., page P was written to disk after the checkpoint but before the system failure). Then, the X latch on P is released. By the end of the redo pass, the B-link tree will be structurally consistent and balanced.

The redo pass of our restart recovery uses the Redo-record-update() algorithm for redoing a record insert, delete, undo-insert, or undo-delete operation $o[v]$ and the Redo-structure-modification() algorithm for redoing a structure-modification operation $o[P1, ..., Pn, V]$.

```
Redo-record-update(n,o,P,v) {
 if (P is in the modified-page table and
 Rec-LSN(P) <= n) {
   X-latch(P);
   if (Page-LSN(P) < n) {
     perform the operation o[v] on P;
     Page-LSN(P) = n;
   }
   unlatch(P);
 }
}
```

For example, to redo an undo-delete logged as n:<T, undo-delete, P, (k, x), m >, the call Redo-record-update(n, undo-delete, P, (k, x)) is issued.

```
Redo-structure-modification(n,o,P1,...,Pn,V) {
 for (i = 1, ... , n) {
   if (P_i is in the modified-page table and
   Rec-LSN(P_i) <= n) {
    X-latch(P_i);
    if (Page-LSN(P_i) < n) {
      install the effect of the operation o[P1, ..., Pn, V]
      on P_i; Page-LSN(P_i) = n;
    }
    unlatch(P_i);
    }
  }
}
```

For example, to redo a split logged as n :<split, Q, Q', M, V >, the call Redo-structure-modification(n, split, Q, Q', M, V) is issued, where M is the storage-map page and V is the set of records moved from page Q to page Q'.

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.

```
Undo-pass( ) {
 for (each forward-rolling transaction T) {
    abort(T);
 }
 while (active-transaction table is not empty) {
   n = max{Undo-Next-LSN(T) | T is in the active-
        transaction table};
   get the log record r with LSN n;
   switch (r) {
     case ''<T, insert, P, (k, x), m>'':
          Undo-insert(T, P, k, m);
     case ''<T, delete, P, (k, x), m>'':
          Undo-delete(T, P, (k, x), m);
     case ''<T, begin>:
          rollback-completed(T);
   }
  }
}
```

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

Example 2. Assume that during the normal processing the stable log contains the following log records.

10: <T1, begin>
20: <T1, delete, Q, (k_1, x_1), 10>
30: <T2, begin>
40: <T1, insert, Q,(k_2, x_2), 20>
50: <T2, insert, R, $(k_3, x_3$), 30>
60: <split, Q, Q', M, V >
70: <T2, delete, R, $(k_4, x_4$), 50>
80: <T1, insert, Q', $(k_5, x_5$), 40>

Assume that after installing the log record with LSN = 80 in the stable 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 following log records are generated in the undo pass:

90: <T1, abort>
100: <T2, abort>
110: <T1, undo-insert, $Q', k_5, 40$ >
120: <T2, undo-delete, $R, (k_4, x_4), 50$ >
130: <T2, undo-insert, $R, k_3, 30$ >
140: <T1, undo-insert, $Q, k_2, 20$ >
150: <T2, rollback-completed>
160: <T1, undo-delete, $Q, (k_1, x_1), 10$ >
170: <T1, rollback-completed>

Theorem 10. *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-link tree with $db(B1) = D1$. Further, let H' be an implementation of H on $B1$ and let L be the sequence of log records generated 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-link*

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-link 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 of operations that implements some completion string for $H1$.

Proof. The way in which update and structure-modification operations are performed 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-link tree that is the result of running some prefix of H' on $B1$. By Theorem 8, $B1$ is structurally consistent and balanced. The rest of the theorem follows from Theorem 9. □

11 Conclusion

In the previously published B-link-tree algorithms, tree-structure modifications remain a challenge to concurrency control, recovery, and tree balancing. In [16,17], a B-link-tree structure modification involving several levels of the tree is divided into smaller structure modifications (atomic actions) in which a page split done on a single level of the tree is decoupled from the linking of the new child to its parent. Thus, tree balance is not guaranteed because it is possible that arbitrary long chains of sibling pages are created that are not directly linked to their parent. Moreover, the proposed restart recovery protocol is not efficient because a second pass over the log has to be made during the undo pass of restart recovery in order to roll back interrupted structure modifications.

In our new B-link-tree algorithms, the recoverability and concurrency problems were solved by defining each structure modification as a small atomic action. Each atomic action X-latches and updates at most two pages on a single level of the B-link-tree at a time for a short duration. Each action retains the structural consistency and balance of the tree and is logged using a single redo-only log record. Thus, in restart recovery, the redo pass of the ARIES algorithm [22] will always produce a structurally consistent and balanced tree on which the database updates by aborted transactions can be undone logically. In our algorithms, structure modifications can run concurrently with other structure modifications and leaf-page updates. Also, in our algorithms, tree-structure modifications interrupted by a system failure are never rolled back during restart recovery, and hence concurrency is increased and recovery simplified. This is in contrast to the algorithms in [7,11,16,17,20,21] in which an interrupted tree-structure modification always have to be rolled back during restart recovery.

In our algorithms, record deletions are handled uniformly with record insertions using a structure modification that merges two sibling pages or redistributes records between two sibling pages. The balance conditions of the B-link tree include that at no level of the tree must there be two successive pages that are both indirect children of their parent. This guarantees that the search path of any database record is at most twice the height of the tree. To maintain the balance conditions under record updates and tree-structure modifications, we defined the concept of a "safe page" and required that each transaction doing an update-mode traversal must always turn each encountered unsafe page into a safe one by performing a suitable structure modification. A full page is safe if is not an indirect child of its parent and if it does not have a right sibling page that is an indirect child. Using the safety concept we gave a rigorous proof that the balance of the B-link tree is indeed maintained under all circumstances. Our algorithms improve concurrency, simplify recovery, reduce the amount of logging, and are deadlock free.

Acknowledgements. The authors would like to thank the reviewers and the associate editor for their valuable comments, which helped to improve the paper.

References

1. Bernstein P, Hadzilacos V, Goodman N (1987) Concurrency control and recovery in database systems. Addison-Wesley, Reading, MA
2. Biliris A (1987) Operation specific locking in B-trees. In: Proc. 1987 ACM international conference on principles of database systems, pp 159–169
3. Chen I, Hassan S (1995) Performance analysis of a periodic data reorganization algorithm for concurrent B-link trees in database systems. In: Proc. 1995 ACM symposium on applied computing, pp 40–45
4. Cosmadakis S, Ioannidou K, Stergiou S (2001) View serializable updates of concurrent index structures. In: Proc. 2001 DBPL international workshop on database programming languages, pp 247–262
5. De Jonge W, Schijf A (1990) Concurrent access to B-trees. In: Proc. 1990 PARBASE international conference on databases, parallel architectures and their applications, pp 312–320
6. Fu A, Kameda T (1989) Concurrency control for nested transactions accessing B-trees. In: Proc. ACM SIGACT-SIGMOD-SIGART international conference on management of data, pp 270–285
7. Gray J, Reuter A (1993) Transaction processing: concepts and techniques. Morgan Kaufmann, San Francisco
8. Goyal B, Haritsa J, Seshadri S, Srinivasan V (1995) Index concurrency control in firm real-time DBMS. In: Proc. 21st VLDB conference, pp 146–157
9. Jaluta I (2002) B-tree concurrency control and recovery in a client-server database management system. Ph.D. thesis and report TKO-A37/02, Department of Computer Science and Engineering, Helsinki University of Technology. http://lib.hut.fi/Diss/2002/isbn9512257068/
10. Keller AM, Wiederhold G (1988) Concurrent use of B-trees with variable-length entries. SIGMOD Rec 17(2):89–90
11. Kornacker M, Mohan C, Hellerstein J (1997) Concurrency and recovery in generalized search trees. In: Proc. ACM SIGMOD international conference on management of data, pp 62–72
12. Kwong Y, Wood D (1982) A new method for concurrency in B-trees. IEEE Trans Softw Eng 8:211–222
13. Lanin V, Shasha D (1986) A symmetric concurrent B-tree algorithm. In: Proc. fall joint computer conference, pp 380–389

14. Lehman P, Yao S (1981) Efficient locking for concurrent operations on B-trees. ACM Trans Database Sys 6:650–670
15. Lomet D (1992) MLR: a recovery method for multi-level systems. In: Proc. ACM SIGMOD international conference on management of data, pp 185–194
16. Lomet D, Salzberg B (1992)) Access method concurrency with recovery. In: Proc. ACM SIGMOD international conference on management of data, pp 351–360
17. Lomet D, Salzberg B (1997) Concurrency and recovery for index trees. VLDB J 6:224–240
18. Mohan C (1989) ARIES/KVL: a key-value locking method for concurrency control of multi-action transactions operation on B-tree indexes. IBM Research Report RJ7008, IBM Almaden Research Center
19. Mohan C (1990) ARIES/KVL: a key-value locking method for concurrency control of multi-action transactions operation on B-tree indexes. In: Proc. 16th VLDB conference, pp 392–405
20. Mohan C (1996) Concurrency control and recovery methods for B±tree indexes: ARIES/KVL and ARIES/IM. In: Kumar V (ed) Performance of concurrency control mechanisms in centralized database systems. Prentice-Hall, Upper Saddle River NJ, pp 248–306
21. Mohan C, Levine F (1992) ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In: Proc. ACM SIGMOD international conference on management of data, pp 371–380
22. Mohan C, Haderle D, Lindsay B, Pirahesh H, Schwarz P (1992) ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans Database Sys 17:94–162
23. Mond Y, Raz Y (1985) Concurrency control in B±tree databases using preparatory operations. In: Proc. 11th VLDB conference, pp 331–334
24. Nurmi O, Soisalon-Soininen E, Wood D (1987) Concurrency control in database structures with relaxed balance. In: Proc. ACM international conference on principles of database systems, pp 170–176
25. Papadimitriou C (1986) The theory of database concurrency control. Computer Science Press, Rockville, MD
26. Sagiv Y (1986) Concurrent operations on B*-trees with overtaking. J Comput Sys Sci 33:275–296
27. Setzer V, Zisman A (1994) New concurrency control algorithms for accessing and compacting B-trees. In: Proc. 20th VLDB conference, pp 238–248
28. Srinivasan V, Carey M (1991)) Performance of B±tree concurrency control algorithms. In: Proc. ACM SIGMOD international conference on management of data, pp 416–425
29. Srinivasan V, Carey M (1993) Performance of B±tree concurrency control algorithms. VLDB J 2:361–406
30. Sullivan M, Olson M (1992) An index implementation supporting fast recovery for the POSTGRES storage system. In: Proc. 8th IEEE data engineering conference, pp 293–300