Helsinki University Of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science
Espoo 2002 TKO-A37/02

B-tree Concurrency Control and Recovery in a Client-Server Database Management System

Ibrahim Jaluta

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY

Helsinki University Of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science
Espoo 2002 TKO-A37/02

B-tree Concurrency Control and Recovery in a Client-Server Database Management System

Ibrahim Jaluta

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Department of Computer Science and Engineering for public examination and debate in Auditorium T2 at Helsinki University of Technology (Espoo, Finland) on the 8th of March, 2002, at 12 noon.

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory of Information Processing Science

Abstract

In this thesis, we consider the management of transactions in a data-shipping client-server database system in which the physical database is organized as a sparse B±tree or B-link-tree index. Client transactions perform their operations on cached copies of index and data pages. A client 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. Efficient implementation of these operations on B±tree and B-link trees are developed for page-server systems. Our algorithms guarantee recoverability of the logical database operations as well as the tree-structure modifications on the physical B±tree or B-link tree. Record updates and structure modifications are logged using physiological log records, which are generated by clients and shipped to the server. During normal processing client transaction aborts and rollbacks are performed by the clients themselves. The server applies the write-ahead logging (WAL) protocol, and the steal and no-force buffering policies for data and index pages. The server performs the restart recovery after a system failure using an ARIES-based recovery protocol.

Tree-structure modifications such as page splits and merges are defined as small atomic actions, where each action affects only two levels of a B±tree, or a single level of a B-link tree, while retaining the structural consistency and balance of the tree. In the case of a B-link tree, our balance conditions guarantee that all pages always contain at least m (a chosen minimum load factor) records and that the length of the search path of a database record is never greater than twice the tree height. Deletions are handled uniformly with insertions, so that underflows are disposed of by merging a page into its sibling page or redistributing the records between two sibling pages. Each structure modification is logged using a single physiological redo-only log record. Hence, structure modifications need never be undone during transaction rollback or restart recovery.

Our algorithms allow for inter-transaction caching of data and index pages, so that a page can reside in a client cache even when no transaction is currently active at the client. Cache consistency is guaranteed by a replica-management protocol based on

callback locking. In one set of our algorithms, both the concurrency-control and replica-management protocols operate at the page level. These algorithms are most suitable for object-oriented database and CAD / CAM applications, where long-lasting editing sessions are typical. Another set of our algorithms is designed for general database applications where concurrency and transaction throughput are the major issues. In these algorithms, transaction isolation is guaranteed by the key-range locking protocol, and replica management operates in adaptive manner, so that only the needed record may be called back. A leaf (data) page may now contain uncommitted updates by several client transactions, and uncommitted updates by one transaction on a leaf page may migrate to another page in structure modifications. Moreover, a record in a leaf page cached at one client can be updated by one transaction while other records in copies of the same page cached at other clients can simultaneously be fetched by other transactions.

Computing Reviews (1998) Categories and Subject Descriptions: H.2.2 [**Database Management**]: Physical Design–*access methods*, *recovery and restart*; H.2.4 [**Database Management**]: Systems–*concurrency*, *transaction processing*.
General Terms: Algorithms, Design.
Additional Key Words and Phrases: ARIES, B-link tree, B-tree, cache consistency, callback locking, client-server database system, key-range locking, page server, physiological logging, tree-structure modifications.

Preface

The research work for this thesis was carried out at the Laboratory of Information Processing Science at Helsinki University of Technology during the years 1997-2001. First of all, praise and thanks to God, who guided and provided me with power and means to accomplish this work.

I express my warm thanks to my supervisors, prof. Eljas Soisalon-Soininen and prof. Seppo Sippu, for their continued guidance and support during these years. Without their help and encouragement, I could not have completed this work.

I would like to thank Dr. Otto Nurmi for his valuable comments on my thesis. I also thank Pekka Mård, for his assistance.

I am very deeply indebted to my wife Fatma, for her support, encouragement and taking care of our children, Al-moatasem, Arrwa, Elaf and Ahmed, during these years.

Finally, my thanks to all who have in one way or another helped me during these years.

Otaniemi, December 2001

Ibrahim Jaluta

Contents

Chapter 1

Introduction

1.1 Background

In a data-shipping client-server database management system, database pages are shipped from the server to clients, so that clients can run applications and perform database operations on cached pages. Many client-server systems have adopted the page-server architecture, because it is the most efficient and the simplest to implement [DeWi90]. Having database pages available at clients can reduce the number of client-server interactions and offload server resources (CPU and disks), thus improving client-transaction response time. Local caching allows for copies of a database page to reside in multiple client caches. Moreover, when inter-transaction caching is used, a page may remain cached locally when the transaction that accessed it has terminated. Hence, in addition to concurrency control, a replica-management protocol for cached pages must be used, to ensure that all clients have consistent page copies in their caches and see a serializable view of the database.

Data caching has been found to offer significant performance gains despite the potential overhead associated with the concurrency-control and replica-management protocols [Wilk90, Care91a, Care91b, Wang91, Moha92a, Fran92b, Fran92c, Fran93, Care94b, Fran97]. These studies have examined caching of database pages in a general setting, without considering index (B-tree) pages in particular, and they do not consider recovery at all. Some client-server systems [Deux91, Fran92a, Care94a, Whit94] have adopted recovery protocols which are based on the ARIES algorithm [Moha92a], but very little has been published on these recovery protocols. Franklin et al. [Fran92a] and Mohan and Narang [Moha94] present two ARIES-based recovery protocols for client-server systems. Also these protocols discuss recovery in general without considering index recovery in particular.

B-tree concurrency-control, replica-management and recovery protocols for client-server systems have received little attention to date. In contrast, numerous concurrent B-tree algorithms have been developed for centralized systems [Baye77,

Lehm81, Kwon82, Mond85, Sagi86, Lani86, Bern87, Nurm87, Bill87, Shas88, John89, Setz94, Poll96]. Despite of the enormous amount of work done in the area of concurrency control of B-trees in centralized systems, most of these works do not consider recovery. Also the transactions considered usually consist only of a single operation, and no key-range scans are included. In [Fuka89, Bill87, Gray93, Lomet97] recovery protocols for B-trees in centralized systems are sketched. Mohan and Levine [Moha92b] were the first to present a detailed B±tree recovery protocol for centralized systems. That recovery protocol is based on ARIES [Moha92a]. Concurrency and recovery for B-link trees in centralized systems are considered by Lomet and Salzberg [Lome92, Lome97] and for generalized search trees by Kornacker, Mohan and Hellerstein [Korn97].

1.2 The Problem and Related Work

The current concurrency-control and replica-management protocols for client-server systems treat index pages and data pages in the same way, and use the two-phase-locking protocol on both data and index pages. However, index pages have more relaxed consistency requirements, and they are accessed more often than data pages and hence cached at clients more often than data pages. Therefore, applying the two-phase-locking protocol can lead to high data contention and decrease the concurrency in the system. Therefore, new and advanced B-tree concurrency-control, replica-management, and recovery protocols are needed to improve the performance of the current client-server systems. The works that are most closely related to the thesis appeared in [Gott96, Basu97, Zaha97].

In [Basu97], a comparative study of the performance of B±tree algorithms in a centralized system and in a client-server system is presented. The study is based on simulation. The lock-coupling and callback-locking protocols are used for concurrency control and replica management, respectively. Concurrency control for leaf pages is adaptive, so that a page-level lock held by a client transaction can be replaced by a set of record-level locks, should transactions at other clients want to simultaneously access other records in the locked page. A transaction T at client A can update any record in an X-locked data page P in the cache of A without contacting the server. Now if some other transaction T´, running at client B, wants to access page P, then the server sends a callback request for P to client A. Client A responds by requesting the server to get global X locks on the records in P updated by T. When the request for the X locks is granted, the updated records in P are X-locked locally and the local X lock on P is released. An adaptive callback-locking protocol is used for both index and data pages. That is, if a record r in a locally S-locked page P at client A is called back, then r is marked as unavailable in P when it is no longer in use at A; if no record in P is in use then P is purged from A’s cache.

In the centralized system used in the study, the operations fetch, insert, and delete a record are performed by the server on behalf of the clients, while in the client-server system, such operations are performed by the clients themselves. The client-server system assumes immediate propagation for updated pages. That is, when a client has updated a page, then a copy of that page is sent to the server with the corresponding log record(s), so that other clients waiting for reading, fetching, or updating the same page are allowed to proceed.

The study of [Basu97] is very limited and does not consider the main issues in B±tree algorithms, such as tree-structure modifications, or recovery. For example, the simulation only involves leaf pages, while page overflow and page underflow are not dealt with at all. The simulation does not consider repeatable reads and phantom avoidance, and it does not consider recovery in the face of client-transaction and system failures.

Gottemukkala et al. [Gott96] present a comparative study of two index cache-consistency (concurrency-control and replica-management) algorithms in client-server systems, using simulation. These algorithms are called callback-locking (CBL) and relaxed-index-consistency (RIC) algorithms, respectively. Both algorithms support client caching of index pages, and allow clients to perform fetch, fetch-next, and update operations. The CBL algorithm uses a strict consistency approach for index and data pages, so that clients always access up-to-date index and data pages. This is because the CBL algorithm uses a page-level callback-locking protocol for cache consistency. On the other hand, the RIC algorithm uses a strict-consistency approach for data pages and a relaxed-consistency approach for index pages. In the relaxed-consistency approach, clients are allowed to access out-of-date cached leaf pages as long as the records retrieved from these pages are the correct ones. This approach enhances concurrency, because an index page can be updated by one client transaction and read by many client transactions simultaneously. The RIC algorithm maintains index-coherency information, both at the server and at the clients, in the form of index-page version numbers. The server updates its index-coherency information whenever it receives a request for an X latch on an index page from a client. When a client sends a request to the server for a record lock, then the server sends the updated index-coherency information and the record lock when the request is granted. The client uses this index-coherency information for checking the validity of their cached copies of index pages.

The algorithms in [Gott96] suffer from the following problems. The callback-locking protocol that is used by both algorithms works at the page level, which can lead to high data contention. Moreover, this protocol cannot be extended to work at the record level or in an adaptive manner. In the RIC algorithm, interior pages are

allowed to become inconsistent, and hence a client transaction traversing the B±tree from the root is not guaranteed to reach a correct leaf page. These algorithms do not show how structure modifications are handled or how B±tree recovery is performed, and no explicit B±tree algorithms are presented. Finally, the RIC algorithm seems to be quite complex to implement in practice.

Zaharioudakis and Carey [Zaha97] present a simulation study with four approaches to B±tree concurrency in client server systems. The four approaches are called no-caching (NC), strict-hybrid-caching (HC-S), relaxed-hybrid-caching (HC-R), and full-caching (FC). In the NC approach, clients do not cache B±tree pages at all, and hence the server executes all tree operations on client requests. In the HC-S approach, clients are only allowed to cache leaf pages, and clients can perform fetch operations and to some extent also fetch-next operations. The server still performs page updating on client requests. The HC-R approach is similar to the HS-S approach, but HC-R uses a relaxed-consistency approach for cached leaf pages. Hence, when the server updates a leaf page, then copies of that page remain cached at other clients for reading, to enhance concurrency. In the FC approach, clients cache both interior and leaf B±tree pages and are able to perform all tree operations locally. This approach is a distributed version of the centralized B±tree algorithm presented in [Moha92b]. In the FC approach, the B±tree pages on each level are doubly linked, so that page underflow can be handled using the free-at-empty approach, and a lazier approach can be used for executing tree-structure modifications.

Potential concurrency in no-caching, strict-hybrid-caching, and relaxed-hybrid-caching approaches is very limited, because the server executes most of the B±tree operations on client requests. The FC approach is the one that is most closely related to the B±tree algorithms in the thesis. However, the FC approach suffers from problems that decrease concurrency and affect the performance of the B±tree in the client-server system: 1) Whenever a database page needs to be updated, then all its cached copies at other clients have to be called back, and hence a page cannot be updated by one client and read by many clients simultaneously. 2) Tree-structure modifications are serialized and no leaf-page updating can be executed while a structure modification is going on. 3) Using the free-at-empty approach does not guarantee a logarithmic upper bound for tree traversals.

1.3 Contribution

The thesis introduces new B±tree and B-link-tree algorithms with recovery for page-server systems where the physical database is organized as a sparse B±tree or

B-link-tree index. In these algorithms, a client transaction can contain any number of fetch, insert, or delete operations. Efficient implementation of these operations on B±trees and B-link trees are developed. Our algorithms allow for inter-transaction caching of data and index pages, so that a page can reside in a client cache even when no transaction is currently active at the client. Cache consistency is guaranteed by a replica-management protocol based on callback locking.

In the first two sets of our B±tree and B-link-tree algorithms, the cache-consistency (concurrency-control and replica-management) protocols are performed at the page level. These algorithms are most suitable for design applications (CAD and CAM) where a typical client transaction spends most of its time in editing a set of related objects. In such applications it is important that, once a page containing objects to be edited has been X-locked (locally and) at the server and cached at the client, no update on that page by the client transaction need be propagated to the server until the transaction has reached its commit point.

In the third set of our algorithms, developed for B-link trees, replica management is adaptive and concurrency is controlled by record-level locking. These algorithms are most suitable for general database applications where concurrency and transaction throughput are a major issue as in a general-purpose RDBMS (relational database management system) in which a typical transaction updates only a few records. Our algorithms guarantee repeatable-read isolation for client transactions and recoverability of the logical database operations as well as of the tree-structure modifications on the physical B±tree or B-link tree. The server employs the write-ahead logging (WAL) protocol and the steal and no-force buffering policies for index and data pages [Gray93].

We summarize our contribution in the following.

1) We improve the current page-level concurrency-control and the callback-locking protocols for page-server systems by augmenting both protocols with a page-level locking protocol that is based on update-mode locks (U locks) [Gray93]. When an updating client transaction traversing the B±tree or B-link tree acquires U locks on the pages in the search path, those pages can simultaneously be cached and read by transactions in other clients. Our improved concurrency-control and the callback-locking protocols increase concurrency in the system, and also prevent starvation. This is in contrast to the standard callback-locking protocol described in [Lamb91, Wang 91, Fran92b, Care94b] in which starvation is possible.

2) We give a simple recovery protocol for page-server systems that avoids the problems found in previous protocols [Fran92a, Moha94]. This protocol is based on

the ARIES recovery algorithm [Moha92a], and it deals with transaction aborts and system failure. In our protocol, client transaction aborts and rollbacks during normal processing are performed by the clients themselves in order to offload the server, while the server performs the restart recovery after a system failure.

3) We introduce a new technique for handling B±tree and B-link-tree structure modifications that improves concurrency, simplifies recovery, and avoids the problems which are associated with the merge-at-half and free-at-empty approaches. Unlike in the algorithms in [Baye77, Kwon82, Bern87, Moha90, Moha92b, Gray93, Moha96], we define a tree-structure modification (page split, page merge, record redistribution, link, unlink, increase-tree-height and decrease-tree-height) as a small atomic action. Each structure modification is logged using a single redo-only log record. Each successfully completed structure modification brings the B-tree into a structurally consistent and balanced state whenever the tree was structurally consistent and balanced initially. Record inserts and deletes on leaf pages of a B±tree or a B-link tree are logged using physiological redo-undo log records as in [Moha90, Moha92b, Gray93, Moha96, Lome97, Zaha97]. Recoverability from system failures of both the tree structure and the logical database is guaranteed because the redo pass of our ARIES-based recovery protocol will now produce a structurally consistent and balanced tree, hence making possible the logical undo of record inserts and deletes.

4) We design new B±tree algorithms for page-server systems to work with our page-level concurrency-control and callback-locking and recovery protocols. In these algorithms, record fetches on leaf pages are protected by commit-duration page-level S locks, and inserts and deletes on leaf pages are protected by commit-duration page-level X locks. Structure modifications are protected by page-level X locks that are held for the duration of the structure modification. For each structure modification, three pages at most are X-locked simultaneously. This is an improvement over the algorithms presented in [Mond85, Gray93], where structure modifications are performed unnecessarily or the whole search path is X-locked at once. In our algorithms, structure modifications can run concurrently with leaf-page updates as well as other structure modifications. This is in contrast to [Moha90, Moha92b, Moha96], where structure modifications are effectively serialized by the use of “tree locks”, in order to guarantee that logical undo operations always see a structurally consistent tree. Our B±tree algorithms can be modified to work with our adaptive concurrency-control and replica-management protocols.

5) To further increase the concurrency in a page-server system, we design new B-link-tree algorithms. In these algorithms, each structure modification now affects only a single level of the tree, and at most two pages need be X-locked

simultaneously. Again, each structure modification retains the structural consistency and balance of the tree. Here the balance of a B-link tree means that the length of the search path of any record is always logarithmic in the number of database records stored in the leaf pages. Our algorithms outperform those presented in [Lome92, Lome97, Zaha97], where the merging (or redistributing) of two pages of a B-link tree needs three pages to be X-locked simultaneously on two adjacent levels, and the balance of the tree is not guaranteed under all circumstances.

6) We design new adaptive concurrency-control and replica management protocols for page-server systems. These protocols avoid the data contention that may occur when concurrency control and replica management are performed at the page level. Unlike those presented in [Gott96, Basu97, Zaha97], page-level X locks are needed only for structure modifications, while record inserts and deletes on a leaf page need only a short-duration U lock on the affected page (besides the record-level X locks). Thus, uncommitted updates on a leaf page by one client transaction can migrate to another leaf page in structure modifications by other transactions, and a record in a U-locked leaf page can be updated by one client transaction while other records can simultaneously be fetched by transactions that cache a copy of the page in other clients. Page-level S locks on leaf pages are not necessarily held for commit duration, but are changed to a set of commit-duration record-level S locks at the server if the page needs to be updated at another client and is therefore called back. The record-level locking protocol is based on key-range locking [Moha90, Moha92b, Gray93, Moha96].

7) We modify the algorithms developed above for B-link trees, so that these algorithms work with our adaptive concurrency-control and replica-management protocols.

8) All our algorithms work in the general setting in which any number of forward-rolling and backward-rolling transactions may be running concurrently, each transaction may contain any number of record fetches, inserts and deletes, and the server may apply the steal and no-force buffering policies.

1.4 Thesis Contents

In Chapter 2 we introduce the basic concepts that are used throughout the thesis. Chapter 3 introduces the algorithms that work with conventional B±trees and perform concurrency control and replica management at the page level. Chapter 4 introduces the new B-link-tree algorithms that retain the balance of the tree in all situations. The algorithms in this chapter employ concurrency control and replica management at the page level. Chapter 5 finally modifies the B-link-tree algorithms

so as to include the key-range locking and callback-locking protocols. Chapter 6 summarizes the main contributions of the thesis.

Chapter 2

Transaction Model and Page-Server Database Systems

In this chapter, the logical database model, the transaction model, the client-server model, and other basic concepts that we shall use in the thesis are introduced. The logical database consists of records identified by unique keys. A transaction can contain any number of record fetches, inserts and deletes. Our client-server database system is a page-server in which transactions running at clients performs their operations on copies of database pages cached at clients. Different types of page servers are obtained when different levels of granularities (record or page) are used for concurrency control and replica management (cache coherency). Concurrency is controlled by page-level or record-level locking, and replicas of data items in client caches are managed at the page level or at the record level, applying callback locking. Transaction atomicity and durability are achieved by physiological logging. Client transactions generate log records for their updates and ship the log records and the updated pages to the server. The server buffer manager applies the steal and no-force buffering policies and flushes updated pages to the disk applying write-ahead logging (WAL). Checkpointing is done by the server.

2.1 Database and Transaction Model

We assume that our database D consists of *database records* of the form (v, x), where v is the *key value* of the record and x is the *data value* (the values of other attributes) of the record. The key values are unique, and there is a total order, $\leq$, 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 (∞, nil) which is never inserted or deleted. The database operations are as follows.

1) *Fetch*[v,θu, x]: Fetch the first matching record (v, x). Given a key value u < ∞, find the least key value v and the associated data value x such that v satisfies vθu and the record (v, x) is in the database. Here θ is one of the comparison operators "≥" or ">". To simulate the fetch-next operation [Moha90, Moha92b, Moha96] on a key range [u, v], the fetch operation is used as follows. To fetch the first record in the key range, a client transaction T issues Fetch[v_1,≥u,x_1]. To fetch the second record in the key range, T issues Fetch[v_2,>v_1,x_2]. To fetch the third record in the key range, T issues Fetch[v_3,>v_2,x_3], and so on. The fetch operation Fetch[v,θu, x] *scans the key range* [u, v] (if θ is "≥") or (u, v] (if θ is ">").

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

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

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

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

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

A backward-rolling transaction $Ba\,bAb^{-1}$ thus denotes an aborted transaction that has *undone* a suffix, b, of its forward rolling phase, while the prefix, a, is still not undone. An aborted transaction of the form $BaAa^{-1}R$ is a *rolled-back transaction* where *R* denotes the *rollback-completed* operation. 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 a rolled-back transaction is called a *terminated transaction*.

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* [Papa86] 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 Ba, the string $Aa^{-1}R$ is the *completion string,* and $BaAa^{-1}R$ the *completed transaction*. For a backward-rolling transaction $Ba\,bAb^{-1}$, the string $a^{-1}R$ is the *completion string,* and $Ba\,bAb^{-1}a^{-1}R$ 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.

Let D be a database and a a string of operations. We define when a *can be run* on D and what is the *database produced*. The empty operation string $\in$ can be run on D and produces D. Each of the operations *B, C, A* and *R* can be run on D and produces D. A fetch[v,θu, x] operation can always be run on D and produces D. An insert[v, x] operation can be run on D and produces $D \cup \{(v, x)\}$, if D does not contain a record with key value v. Otherwise, it produces D. A delete[v, x] operation can be run on D and produces $D \setminus \{(v, x)\}$, if D contains (v, x). Otherwise, it produces D. An operation string ao, where o is a single operation, can be run on D and produces D´ if a can be run on D and produces D´´ and o can be run on D´´ and produces D´.

2.2 The Client-Server Model

A *data-shipping client-server* system consists of one server connected to many clients via a local-area network (Figure 2.1). A data-shipping system is usually a page server, so that pages are used as units of data transfer between the server and the clients. The server manages the disk version of the database, the log, the buffer, the global locking across the clients, and the restart recovery procedure. The server also uses a copy table to keep track of the locations of the cached pages.

Each client has a local lock manager to manage the locally cached pages, and a buffer manager which manages the contents of the client buffer pool. Each client transaction performs its updates on the cached pages and produces log records, which are kept temporarily in the client cache until they are sent to the server (clients do not have disks for logging). All index operations are performed at the client. Client–server interactions during B-tree operations take place in terms of low-level physical requests (request for locks, missing B-tree pages, or new pages). In order to minimize server disk access, and to fully exploit the use of client main memory, clients are allowed to retain their cached pages across transaction boundaries (inter-transaction caching). For the sake of simple presentation and readability of our B-tree algorithms for client-server systems, we assume that each client runs only one transaction at a time.

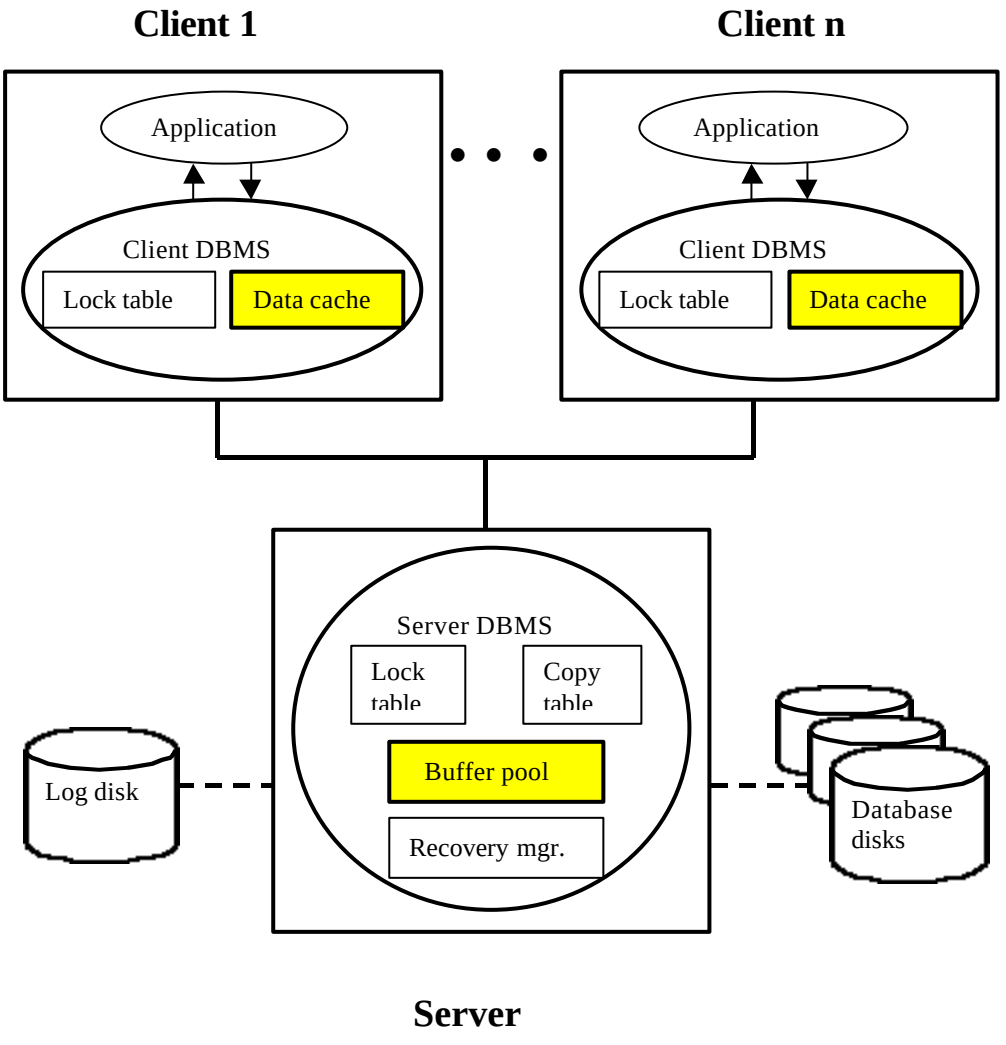

Figure 2.1. Architecture of a client-server DBMS system.

2.3 Locks and Latches

In a centralized database system, *locks* are typically used to guarantee the logical consistency of the database under a concurrent history of transactions, while *latches* are used to guarantee the physical consistency of a database page under a single operation. A latch is implemented by a low-level primitive (semaphore) that provides a cheap synchronization mechanism with S (shared), U (update), and X (exclusive) modes, but with no deadlock detection [Gray93, Moha96, Lome97]. 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.

In a client-server system, the consistency of database pages can be guaranteed either by latching or locking. To latch a page P at the server, P has to be resident in the buffer of the server, because latches are associated with buffer frames. Therefore, when client A requests an X latch or a U latch on a cached page P, then the server has to read P from the disk (stable storage) if P is not resident in the buffer pool. On the other hand, when client A requests an X lock or a U lock on a cached page P, then P need not be resident in the server buffer pool in order to lock P at the server. Hence, to avoid such disk reads at the server, we use page-level locks instead of latches in all our B-tree algorithms we develop in the thesis.

Lock requests may be made with the conditional or the unconditional option [Moha90, Moha92b, Moha96]. A *conditional* request means that the requestor (transaction) is not willing to wait if the lock cannot be granted immediately. An *unconditional* request means that the requestor is willing to wait until the lock is granted. Moreover, a lock may be held for different durations. An *instant-duration* lock is used to check for any current lock conflicts and to make the requestor wait if such a conflict is detected. However, no actual lock is retained on the data item. A *short-duration* lock is released after the completion of an operation and before the transaction that performed this operation commits. A *commit-duration* lock is released only at the time of termination of the transaction, i.e., after the commit or rollback is completed.

2.4 Page-Server Architectures

The three main functions of a client-server DBMS – data transfer between the server and clients, concurrency control (locking), and replica management (callback) – can be performed at different granularities, such as at the file, page, object or record

level. But experience has shown data transfer at the page level is the most efficient and the simplest to implement [DeWi90]. Hence the page-server architecture, in which data transfer takes place at the page level, was adopted in many client-server DBMSs. For concurrency control and replica management, different combinations of granularities can be used [Care94b], but care should be taken when the concurrency-control and replica-management granularities are smaller than the data-transfer granularity. The possible combinations of concurrency-control and replica-management granularities are shown in Figure 2.2. According to [Care94b], we could have four types of page servers (PS).

	Concurrency Control	Replica Management
Granularity	page	page
	record	record
	record	adaptive
	adaptive	adaptive

Figure 2.2. Possible combinations of concurrency control replica management granularities.

1) PS-PP. Both concurrency control and replica management are done at the page level (PP). This type of page server is called the *basic page server*. It is easy to implement and is very efficient in terms of the number of exchanged messages between the server and clients. An updating client holds an X lock on the updated page for commit duration. The drawback is that potential concurrency is low due to page-level locking.

2) PS-RR. Both concurrency control and replica management are done at the record level (RR). PS-RR combines the communication advantages of page transfers with the increased concurrency allowed by record-level locking and record-level callback. However, PS-RR suffers from the following inefficiency. Assume that transaction T1 at client A reads a given page P once, and then page P remains unused in A's cache, that is, none of the records in page P is being used at client A. If transaction T2 at client B wishes later to update some records on page P, then a separate callback request will be sent to client A by the server for each individual record that T2 wants to update. In addition the server copy table could consume a lot of memory space, because in PS-RR the information on the location of copies is kept at the record level.

3) PS-RA. Concurrency control is done at the record level (R), while replica management is done in an *adaptive* manner (A). That is, the callback granularity is

not static but dynamic. PS-RA avoids the inefficiency of a PS-RR server as follows. When a record r in page P at client A is called back and none of the records in page P is being used at A, then P is purged from A's cache. Otherwise, just the requested record r is called back when it is no longer in use at A, and the called-back record r is marked as *unavailable* at client A. Moreover, the server copy table size is reduced, because in PS-RA the information on the location of the copies is kept at the page level.

4) PS-AA. Both concurrency control and replica management are performed adaptively (AA), see Sections 5.2 and 5.3. In this server type, both the concurrency-control and replica-management granularities are dynamic and not static.

Performance of different page servers was analyzed in [Care94b, Zaha97]; these studies conclude that PS-AA outperforms the other page-server types.

2.5 The Standard Callback-Locking Protocol

Inter-transaction caching allows multiple copies of pages to reside in different client caches across transaction boundaries, so that a page can reside in a client cache even when no record in the page is currently locked by any transaction. Thus, in addition to concurrency control, replica management is needed for managing the page copies (replicas). For replica management a lock-based protocol known as *callback locking* [Howa88] is used. In our page-server model we will use a variant called the callback-read protocol [Lamb91, Wang91, Fran92b, Care94b].

The *callback-read protocol* guarantees that copies of pages in client caches are always valid, so that a client transaction can safely S-lock and read cached pages without server intervention. The server maintains the X locks while clients maintain the S locks. The standard callback-read protocol works as follows:

When a client transaction T at client A wants to read a page P that is not resident in A's cache, T requests from the server a copy of P. If P is not X-locked at the server by another client transaction, then the server returns the latest copy of page P to A. Otherwise, T must wait until the conflicting lock is released.

When a client transaction T at client A wants to update a page P which is not X-locked locally (i.e., at the client), T requests the server to get an X lock on P. If P is X-locked at the server, then T waits until the conflicting lock is released. When P is no longer X-locked at the server, the server issues callback requests to all other clients (except A) which hold a cached copy of P. At a client B, such a callback request is treated as follows. If the callback request cannot be granted immediately

(due to a local S lock held on the cached copy of P by an active transaction at client B), then client B responds to the server saying that P is currently in use, so that the server can detect any deadlocks. When P is no longer in use at B, P is purged from the client cache of B and an acknowledgement is sent to the server. Once all the callbacks have been acknowledged to the server, the server registers an X lock on P for the requesting client A and informs A that its request for an X lock on P has been granted. Subsequent read or update requests for P by transactions from other clients will then block at the server until the X lock is released by the transaction at A.

The server updates its copy table when a page copy is purged from a client cache, or when a page copy is sent to a client which does not have a cached copy of that page. When a client transaction completes its execution, the client sends copies of all the updated pages and the log records to the server with a request to release all its acquired locks. However, the client retains copies of all updated pages in its cache, and thus permission to grant local S locks on those pages to its transactions.

2.6 Logging and LSN Management in a Page-Server System

We use *physiological logging* [Moha90, Moha92b, Gray93, Moha96] as a compromise between physical logging and logical logging. A client buffers transaction log records in its virtual storage temporarily. When a client needs to send an updated page P to the server, then the client must also send all the log records generated for updates on P. At the commit of a transaction, the client sends copies of all the updated pages and the log records to the server with a commit request. On receiving the log records from a client, the server log manager appends these log records to its log buffer. To facilitate the rollback of an aborted client transaction, the log records of a transaction are chained via the Prev-LSN field of log records in reverse chronological order. A client must not discard a log record from its local log buffer until it gets confirmation from the server that the log record has been safely written onto stable storage at the server.

The server log manager forces the log records of a client transaction to stable log when the server receives copies of the updated pages and the log records with a commit request from the client. When the server buffer manager wants to write an updated page P to stable storage, the log manager has to force the log records up to and including those for P to the stable log. In addition to forces caused by client transaction commits and buffer manager activities, a background process may periodically force the log buffer as it fills up.

The important fields that may be present in different types of log records are:

Transaction-id: The identifier of a client transaction, if any, that wrote the log record.
Operation: Indicates the type of the update operation in the case of a redo-undo, redo-only or compensation log record.
Type: Indicates whether the record is “begin”, “abort”, “rollback-completed”, “commit”, “redo-undo”, “redo-only”, “compensation”.
Page-id(s): The identifier(s) of the page(s) to which updates of this log record were applied; present in log records of types “redo-only”, “redo-undo”, and “compensation”.
LSN: The log sequence number of the log record, which should be a monotonically increasing value.
Prev-LSN: The LSN of the preceding log record written by the same client transaction in its forward-rolling phase.
Undo-Next-LSN: This field appears only in a compensation log record generated for a backward-rolling aborted client transaction. The Undo-Next-LSN is the LSN of the log record for the next update to be undone by the backward-rolling client transaction.
Data: Describes the update that was performed on the page. That is, the record(s) that were inserted to or deleted from the page(s).

A *redo-only* log record only contains redo information while a *redo-undo* log record contains both the redo and undo information. Therefore, every redo-only log record is *redoable*, and every redo-undo log record is both redoable and *undoable*. However, when the update logged using a redo-undo log record is undone, then the undo action is logged using a *compensation log record* (CLR). The Undo-Next-LSN of the generated CLR is set to the Prev-LSN of the log record being undone [Moha90, Moha92b, Moha96, Gray93]. A compensation log record is generated as a redo-only log record and hence an undo operation is never undone. Therefore, during the rollback of an aborted transaction the Undo-Next-LSN field of the most recently written CLR keeps track of the progress of the rollback, so that the rollback can proceed from the point where it left off should a crash occur during the rollback. For an example, see Section 3.11.

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

In a client-server system, a client cannot afford to wait for a log record to be sent to the server and for the server to respond back with an LSN for the newly written log record before the updated page's Page-LSN is set to the correct value. Therefore, we allow clients to assign LSNs locally. However, if every client were allowed to assign LSNs independently of other clients, then it could happen that different clients update the same page creating a sequence of LSNs which may no longer be monotonically increasing. The property that the Page-LSNs are monotonically increasing is very important, because the recovery manager uses this property to track the page's state, and the log manager and the buffer manager at the server use this property as well.

In a client-server system a Page-LSN is used as an update sequence number and not as a direct log-record address. Hence, clients assign LSNs locally as in [Moha94]. When a client transaction is about to update a page, a client generates a new LSN value by

$$LSN := Max\ (Page\text{-}LSN, Local\text{-}LSN)+1,$$

where Page-LSN is the current Page-LSN value in the page to be updated and *Local-LSN* is the LSN value of the log record most recently generated by the local log manager. Therefore, when a client transaction T updates a page P, then a log record is generated, and the Page-LSN field of P and the generated log record are assigned the LSN value computed by the above formula. The Local-LSN field is then updated by

$$Local\text{-}LSN := LSN.$$

Generating LSNs as above guarantees that the values assigned to the Page-LSN field of page P by different clients always form a monotonically increasing sequence. This is because a page can reside for updating in the cache of at most one client at a time.

2.7 The Server Buffer Management

The policy used by the server buffer manager to manage the buffer is very important with respect to the recovery and the performance of the client-server system overall. If the server buffer manager allows pages updated by an uncommitted client transaction to be written to stable storage, then it is said that the buffer manager applies the *steal policy*. Otherwise, it is said that the buffer manager applies the *no-steal policy*. If the buffer manager forces (writes) pages updated by a client transaction to stable storage in order to commit the transaction, then it is said that

the buffer manager applies the *force policy*. Otherwise it is said that the buffer manager applies the *no-force policy.*

The concurrency-control protocol is responsible for maintaining the consistency and isolation properties of ACID (atomicity, consistency, isolation and durability) transactions, while the recovery protocol is responsible for maintaining the atomicity and durability properties of transactions. During restart recovery, the process of removing the effects of aborted transactions for preserving atomicity is known as *undo processing*, while the process of reinstalling the effects of committed transactions for preserving durability is known as *redo processing*.

If the server buffer manager uses the steal policy, then undo processing is needed during restart recovery; otherwise it is only needed in rolling back transactions during normal processing. If the server buffer manager uses the no-force policy, then redo processing is needed during restart recovery; otherwise it is not needed. Therefore, we conclude that the policy used by the buffer manager determines the restart recovery protocol to be used. A summary of the buffer policies and the needed recovery protocols [Bern87, Gray93] is shown in Figure 2.3. The combination of the steal and no-force policies does not put any restrictions on the buffer manager. It is very suitable for normal processing. A well-known recovery protocol that supports the steal and no-force policies and fine-grained concurrency control is called ARIES [Moha92a].

Buffer Policy	Needed Restart Recovery Protocol
Steal and No-Force	Undo and Redo
Steal and Force	Undo and No-Redo
No-Steal and No-Force	No-Undo and Redo

Figure 2.3. Restart-recovery protocols for combinations of buffer policies.

The buffer managers at the clients and the server use the least-recently-used (LRU) policy for page replacement. That is, the least-recently-used page of the B-tree will be replaced, when the buffer is full and there is a need for a buffer frame. Hence, the most recently used pages will remain in the cache. In a B-tree, the pages that are most heavily used by client transactions are the pages at the top levels of the B-tree. Therefore, such pages will remain in the client caches, and hence the LRU policy improves the system performance. The functions of a client buffer manager are

limited to page replacement, and installing pages received from the server, because the clients do not store database records on their local disks.

2.8 The Write-Ahead-Logging (WAL) Protocol

A recovery protocol is responsible for maintaining the atomicity and durability properties of an ACID transaction. Hence, for the sake of correct recovery, the server employs the *write-ahead-logging* (*WAL*) *protocol* [Moha92a, Gray93], so that, in the event of a system crash, any update in the database that is reflected on stable storage is also found recorded in the stable log. When the server buffer manager needs to write a modified page P to stable storage, the server log manager is requested to flush all the log records up to and including the log record with an LSN equal to the Page-LSN of P. Therefore, the log-record address, instead of the log record's LSN, has to be passed to the log manager.

In order that the WAL protocol would work correctly in a client-server system, the server uses an *LSN-address list* to map a log-record LSN to its actual address in the log [Moha94]. Each entry in the LSN-address list consists of a pair (log-record-LSN, log-record-address). Hence, when the server receives a modified page P and a log record for an update on P, then the server performs the following.
1) The received log record is appended to the log and the pair (log-record-LSN, log-record-address) is inserted into the LSN-address list.
2) Page P is installed in the server buffer pool, and the log-record-address is stored in the buffer-control block (BCB) associated with the buffer frame of P, using the LSN-address list.
3) The flag "modified" is set in the BCB of P, if it is not set already. However, if the flag "modified" was set, then the new log record address is not stored in the BCB. Thus, the BCB always contains the address of the log record for the update that made P modified.

Thus, before a modified page P is written to stable storage, the server executes the *WAL protocol* as follows:
Step 1. Determine the address of the log record for the latest update on P, using the log record address stored in the BCB associated with buffer frame of P.
Step 2. Flush all the log records up to and including the log record whose address is determined in Step 1. □

2.9 Transaction Table, Modified-Page Table and Checkpointing

The server maintains two tables called transaction table and modified-page table. The *transaction table* contains an entry for each active client transaction. Each entry

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

The server takes *checkpoints* [Bern87, Moha92a, Gray93] periodically during normal processing, to reduce the amount of recovery work that would be requested when the system crashes. When a checkpoint is taken, a *checkpoint log record* is generated which includes copies of the contents of the transaction and modified-page tables. Taking a checkpoint involves no flushing of pages. Clients need not take checkpoints, because clients use the *force-to-server-at-commit* policy. That is, when a client transaction T is about to commit, then the client sends copies of all pages updated by T, and the log records for these updates, to the server.

Chapter 3

B±tree Concurrency Control and Recovery in a PS-PP System

In this chapter, we describe the B±tree algorithms to implement transactions in a PS-PP page-server system in which both concurrency control and replica management are performed at the page level. We improve the standard callback-read locking protocol by augmenting it with the U-locking protocol [Gray93, Lome97], so that the concurrency-control protocol and the replica-management protocol work together correctly. The lock-coupling protocol [Baye77] is used for concurrency control on index pages; once the traversing transaction reaches the target leaf page, the leaf page is locked in the proper (S or X) mode for commit duration to guarantee transaction isolation. In the current B±tree algorithms for centralized and client-server DBMS, tree-structure modifications are still presenting a challenge to concurrency control, recovery and tree balancing. We introduce five structure modification operations, each of which performs an atomic structure modification involving three pages on two adjacent levels of the tree. Each atomic structure modification is logged using a single redo-only log record. Thus a completed atomic structure modification need never be undone during normal processing or restart recovery. A completed atomic structure modification brings the tree into a structurally consistent and balanced state whenever the tree was initially structurally consistent and balanced. Our B±tree algorithms increase concurrency in the system, provide simple recovery, and avoid the problems associated with the "merge-at-half" and the "free-at-empty" approaches [Moha90, Moha92b, Gray93, John93, Moha96]. The algorithms are most suitable for special design (CAD/CAM) applications in which data caching at clients is essential for efficient computation, while concurrency between different clients is not a major issue.

3.1 B±trees

We assume that our B±tree is similar to the B±tree of [Baye77]. We use this B±tree as a database index in the client-server DBMS. We also assume that the B±tree is a sparse index to the database, so that the leaf pages store the database records. In addition, we assume that the B±tree is a unique index and that the key values are of fixed length.

Formally, a B±tree is an array B[0,…,n] of *disk pages* B[P] indexed by unique *page identifiers* (*Page-ids*) P = 0,…,n. The page B[P] with Page-id P is called page P, for short. A page (other than page 0) is marked as an *allocated* if that page is currently part of the B±tree. Otherwise, it is marked as *unallocated*. Page M=0 is assumed to contain a *storage map* (a bit vector) that indicates which pages are allocated and which are unallocated. Page 1, the *root*, is always allocated. The allocated pages form a tree rooted at 1.

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

If searching for key value u, then in non-leaf page we follow the pointer P_1, if $u \leq v_1$, and the pointer P_i, if $v_{i-1} < u \leq v_i$. An example of a B±tree is shown in Figure 3.1. The data values of the database records are not shown.

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

database records, or (2) P is a non-root index page and contains less than m_2 child links.

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

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

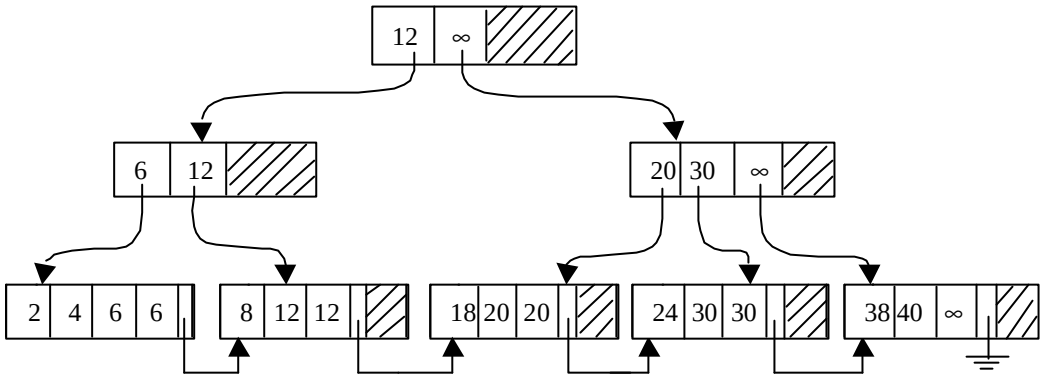

Figure 3.1. A B±tree that represent a database consisting of records with key values 2, 4, 6, 8, 12, 18, 20, 24, 30, 38 and 40.

3.2 B±tree Structure Modifications

Traditionally, the space-utilization and balance conditions of the B±tree state that the pages are not allowed to fall below 50 % full. When a full page P has been split, it has exactly that minimum space utilization. Now if a record is deleted from P, then P has to be merged back with its sibling page, since P fell below 50 %. Also the page resulting from the merge can again be split after two consecutive inserts. Hence, the B±tree needlessly thrashes between splitting and merging the same page [Maie81].

One approach to avoid the above problem is to allow pages to be emptied completely [Moha90, Moha92b, Gray93]. This issue was analyzed in [John93], which concludes that page merging in a B±tree is not useful unless the pages are completely empty. That is, *free-at-empty* is much better (fewer access conflicts because of structure modification operations, and therefore higher concurrency on the B±tree) than *merge-at-half*. However, the free-at-empty approach does not

guarantee (in the worst case) the logarithmic bound for the B±tree traversal algorithm; in addition, it leads to low disk utilization. Moreover, the free-at-empty approach is not applicable in a B±tree whose interior pages are one-way linked to the right unless all the nodes of a B±tree are doubly linked as in [Zaha97].

We introduce a new approach, which is a compromise between the free-at-empty and merge-at-half approaches to handle page merging (or record redistribution). This approach avoids thrashing and increases concurrency (less page merging), and guarantees the logarithmic bound for B±tree traversals. In our approach, the minimum number of records a B±tree page can hold is m_1, $2 \leq m_1 < M_1 / 2$, for a non-root leaf page and m_2, $2 \leq m_2 < M_2 / 2$, for a non-root index page, where m_1, m_2, M_1 and M_2 are defined as in the previous section.

The B±tree structure modifications present a challenge to the concurrency-control and recovery protocols. In the literature there are two techniques designed to handle the structure modifications. In the first technique, a transaction X-locks the pages along the structure-modification path, and executes it before executing the insert/delete operation that triggered such a structure modification. In the second technique, a transaction acquires a tree lock before it executes a structure modification, triggered by an insert or a delete operation.

In the technique presented in [Gray93], an updating transaction X-locks the pages along the B±tree structure-modification path top-down, before it starts the execution of the structure modification. Hence, two structure modifications can be executed concurrently only if they occur on completely distinct paths. Two structure modifications having at least one page in common on their paths will be serialized. X-locking the structure-modification path by an updating transaction T will prevent other concurrent client transactions from reading or updating any of the pages on such a path. On the other hand, if the path were not X-locked at once by T, and if other transactions were allowed to update pages on the path, then T would wipe out updates of other transactions to these pages, when T aborts or the system fails.

This technique suffers from two problems. The first one is the reduction in concurrency and the overhead due to the X-locking of the pages on the path in terms of the callbacks and waiting time. The second problem is that, if leaf-page updates and structure modifications are allowed to execute concurrently, then during restart recovery the B±tree could be structurally inconsistent and thus fail to perform logical undo operation [Moha92b, Moha96].

In the technique presented in [Moha90, Moha92b, Moha96], concurrent B±tree structure modifications are prevented through the use of a *tree lock* (or *a tree latch*).

That is, an updating transaction acquires a tree lock in X mode, before it starts the execution of a structure modification. Moreover, other transactions are prevented from performing any updating to leaf pages while the structure modification is still going on. This is because if transaction T1 updates a leaf page P and transaction T2 moves this update to another page P´ and commits while the structure modification is going on, then it is not possible to undo the update of transaction T1, if the system fails before the on-going structure modification is committed. During the undo pass of the restart recovery a page-oriented undo of the update by T1 would fail, and trying to undo the update logically would fail too, because the B±tree is not yet structurally consistent (also see [Lome98]). When a transaction traversing the B±tree reaches an ambiguous situation (cannot decide which page to traverse next) while a structure modification is going on, then the traversing transaction must acquire a tree lock in S mode for short duration and retraverse the B±tree, so that the traversing transaction does not reach a wrong leaf page. For the sake of a correct recovery, no leaf page updating is allowed while a structure modification is going on.

This technique is not suitable at all for use in a client-server system, because when a client transaction T holds a tree lock in X mode to perform a structure modification, then new client transactions will be prevented from accessing the B±tree, and at the same time other client transactions are prevented from updating leaf pages.

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

If a client transaction uses S locks to lock the pages along the split/merge path, then a deadlock may occur when two transactions try to upgrade their S locks on a common page in the path to X locks at the same time [Srin91]. In our algorithms, to avoid such deadlocks and to increase concurrency, an updating client transaction T uses U locks to lock the pages along the split/merge path as follows. T traverses the B±tree using the lock-coupling protocol with S locks. If the reached leaf page P is found to be full or about to underflow, then P is unlocked to avoid a deadlock with another transaction traversing the B±tree. Then, T retraverses the B±tree from the root page, now using lock coupling with U locks, and keeps a traversed full or about-to-underflow page U-locked. When a non-full or a not-about-to-underflow

page is encountered, T releases the acquired U locks on the ancestors of this page. If the reached leaf page P is found to be non-full (for insert) or not- about-to-underflow (for delete), then T releases all the acquired U locks on the ancestors of P and updates P. Otherwise, T keeps the acquired U locks on the ancestors of P and executes all the needed B±tree-structure-modification operations, and updates the proper leaf page.

3.3 Execution of B±tree Structure Modifications as Atomic Actions

To save time and efforts when a client transaction aborts or the system fails, we would like the B±tree structure modifications made by a client transaction T to be committed regardless of whether T will eventually commit or abort. That is, the structure modifications are executed as atomic actions [Lome92, Lome97], and will never be undone once executed to completion. In the literature, a structure modification can be executed as an atomic action either by generating a special transaction [Lome97], or by executing the structure modification as a nested top action [Moha90, Moha92b, Moha 96].

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

Nested top actions actually provide a lightweight implementation for the special transactions. When an updating client transaction T needs to execute a B±tree structure modification, then T saves the LSN of the last log record it has generated, before starting the execution of the structure modification. Then T executes the structure modification, updates the involved pages, generates log records, and updates the Page-LSNs. When T completes the structure modification, it generates a *dummy compensation log record* (CLR), which is set to point to the log record whose LSN was saved previously, as shown in Figure 3.2.

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

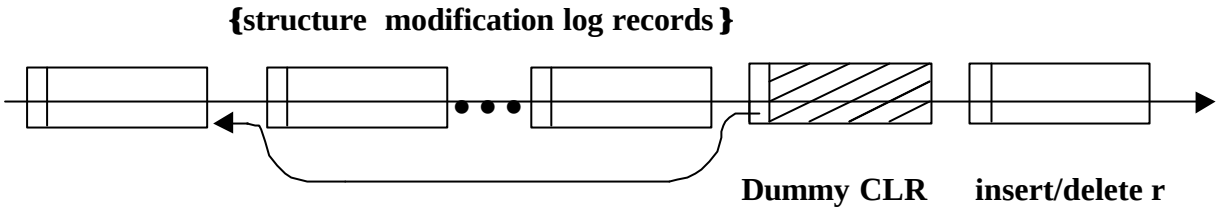

Figure 3.2. Part of the log showing a structure modification (triggered by an insert/delete of record r) which is executed as a nested top action.

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

In our new approach, a structure modification involving several levels of the tree is divided into smaller atomic actions, each of which involves X-locking tree pages on two adjacent levels of the tree. Each structure modification is logged using a single redo-only log record. Thus a completed atomic structure modification need never be undone during normal processing or restart recovery. A completed atomic structure modification brings the tree into a structurally consistent and balanced state whenever the tree was initially structurally consistent and balanced. When an insert or delete operation by a client transaction T triggers a tree-structure modification, then T U-locks all the pages along the structure modification path and executes the structure modification in a top-down order (Algorithm 3.6 or 3.7 in Section 3.9).

3.4 Page-Level Replica Management with U Locks in PS-PP

In the standard callback-read locking protocol, a client transaction can cache a page in shared (S) mode, if that page is not X-locked at the server by another client, while an X-locked page can only be cached at one client at a time. When an updating client transaction T wants to update a page P, it acquires an X lock on P from the

server, reads P, and then updates P if needed. Hence, other client transactions are prevented from reading P, while it is X-locked by T.

If concurrency control is performed at the record level and transaction T wants to update page P, then P need be X-locked by T for short duration only. Such page locking can still reduce concurrency, because the callbacks effectively prevent transactions in other clients from reading different records on such an X-locked page. When an updating client transaction X-locks the pages along a B±tree structure-modification path, concurrency would be decreased further. Moreover, If an updating client transaction uses S locks to lock the pages along the structure-modification path, then a deadlock could take place when the S locks on these pages need to be upgraded to X locks.

We use *update-mode locks* (*U locks*, for short) [Gray93] on the page level to avoid decrease in concurrency and deadlocks that could result from upgrading S locks to X locks on a page simultaneously cached at two clients. An updating client transaction acquires a U lock on page P, and later on upgrades the lock to an X lock if an update on P is needed. The server maintains the global locks, U and X, while clients maintain the local S locks. However, in addition to its local S locks, each client records locally the global locks its transactions hold.

The *U-locking protocol* in a centralized DBMS says that, once a transaction has acquired a U lock on a data item Q (in which case other transactions can only hold S locks on Q), then no new U locks or S locks can be granted on Q. However, in a client-server environment applying callback locking the granting of S locks is done locally at the clients. Therefore, it is not feasible to prevent a client B from S-locking an item Q cached at B when Q is already U-locked at a client A. On the other hand, it is feasible to prevent B from obtaining a data item Q with a U lock on it from the server if Q is U-locked at client A. In this case B is forced to wait. Thus in a client-server system, the compatibility matrix is as in Figure 3.3.

	S	**U**	**X**
S	Yes	Yes	No
U	Yes	No	No
X	No	No	No

Figure 3.3 Compatibility matrix for the U-locking protocol in a client server-system.

The compatibility matrix is symmetric, so that a U lock is compatible with an S lock and vice versa. This is in contrast to the standard U-locking protocol used in a centralized DBMS, when new S locks can no longer be granted on Q once some transaction has been granted a U lock on Q. In our protocol, a U-locked page can be cached at other clients. Hence a transaction T at client A can S-lock and read a cached page P, which is U-locked at the server by client B.

We improve the callback-read locking protocols for a PS-PP page-server system by augmenting the protocol with a page-level locking protocol that is based on U locks. When the callback locking protocol is augmented with the U-locking protocol, starvation may take place in the system. For example, assume that a copy of a B±tree page P is cached and U-locked at client A by transaction T while a copy of P is currently cached at client B. If T needs to update P, then T requests the server to upgrade its U lock on P to an X lock. The server sends a callback request for P to a client B. Now if other clients request the server for copies of page P before the callback on P from B is completed, then the server grants copies of P to the requesting clients as P is currently locked at the server by T in U mode only. Thus, if client requests for new copies of P keep coming and the server keeps granting these requests while P is still U-locked by T, then T may never get a chance to upgrade its U lock on P to an X lock, that is, T may starve.

In centralized systems that use the U-locking protocol, starvation is avoided by preventing a transaction from acquiring an S lock on a B±tree page which is currently locked in U-mode. Our improved protocol prevents starvation in a PS-PP page-server system as follows. When a client transaction T at client A requests the server to upgrade its U lock on page P to an X lock, the server acquires an X lock on P for T (to block new client requests for a copy of P) and sends a callback requests for P to all other clients (except A) that are caching copies of P. When all the callbacks have been acknowledged to the server, the server informs client A that its upgrade request has been granted. This is in contrast to the callback-read locking protocol described in [Lamb91, Wang 91, Fran92b, Care94b] in which starvation is possible.

The *callback-read locking protocol with U locks* works as follows. When a client transaction T at client A wants to read a page P, which is not resident in the client cache, then T requests from the server a copy of P. The server acquires an instant S lock on P. If the lock is granted right away, then the server returns the latest copy of P to client A. Otherwise, the server requests an unconditional S lock on P. When the lock is granted, the server returns the latest copy of P to client A and releases its S lock on P.

When a client transaction T at client A wants to acquire a U lock on page P, T requests the server to get a U lock on P. If P is neither X-locked nor U-locked at the server by another client, then the server grants a U lock on P and sends the latest copy of P to A. Otherwise, A waits until the conflicting lock is released. A installs the received copy of page P in its local cache and U-locks P locally when the U lock is granted.

When a transaction T at client A wants to update a cached page P on which it currently holds a U lock, T requests the server to upgrade its U lock on P to an X lock. The server acquires an X lock on page P and sends callback requests to all other clients (except A) which hold a cached copy of P. At a client B, such a callback is treated as follows. If the callback request cannot be granted immediately, due to a local S lock held on the cached copy of P by an active transaction at B, then B responds to the server saying that P is currently in use. When P is no longer S-locked at B, P is purged from the cache of B and an acknowledgement is sent to the server. Once all the callbacks have been acknowledged to the server, the server informs A that its upgrade request has been granted. As in the standard callback-read protocol, the server updates its copy table when a page copy is purged from a client cache, or when a page copy is sent to a client, which does not yet have a cached copy of the page.

Theorem 3.1. The improved callback-locking protocol with S, X and U locks is starvation-free.

Proof. The result follows from the fact that whenever a client transaction T at client A requests the server to upgrade its U lock on page P to an X lock, the server acquires an X lock on P for T (to block new client requests for a copy of P) and sends callback requests for P to all other clients (except A) that are caching copies of P. Thus, T can proceed as soon as the transactions that are currently holding P S-locked have terminated. □

Advantages of using U locks in a client-server environment include:

1) A U lock on a page P can be upgraded to an X lock without any deadlocks.

2) When a page P is U-locked by an updating client transaction, then P can be cached at other clients, which increases concurrency. For example, when an updating client transaction T traverses a B±tree using lock coupling with U locks, then the pages U-locked by T can remain cached at other clients.

3) When the concurrency control is performed at the record level (see Chapter 5), then a transaction T at client A can update a record r in a U-locked leaf page P cached at A, while transactions at other clients can simultaneously cache P and fetch available records other than r in P. This would not be possible if T should X-lock P in order to update it. Hence, concurrency is increased in the system, in addition to the savings in the number of the callbacks that would be performed otherwise.

4) U-locking pages along a B±tree structure modification path initially, instead of X-locking them all at once, allows other transactions to read these pages before the U locks are upgraded. Moreover, if the structure-modification path involves the root page, then X-locking the root page would prevent new transactions from accessing the B±tree during the execution of the structure modification.

However, there are also some drawbacks of using the U locks. Updating client transactions traversing the same path are serialized. Upgrading of a U lock needs an extra message to be sent to the server.

3.5 Page-Level Concurrency Control in PS-PP

A client transaction T uses the lock-coupling protocol to traverse the B±tree (Algorithm 3.1 or 3.2 in Section 3.7). When the target leaf (data) page P is reached, then P is S-locked locally for fetching and X-locked for updating, after acquiring an X lock from the server. In the case of the operation Fetch[k, θu, x], T searches the S-locked page P for a record r with the least key value k satisfying kθu, holds the S lock on P, and returns with r. Hence, other client transactions are prevented from updating records in P before T commits.

In the case of the operation Insert[k, x], T searches the X-locked page P for the position of insertion. If P already contains a record with the key value k, then the insert operation is terminated and the exception “uniqueness violation” is returned. Otherwise, T inserts (k, x) into P, and holds the X lock on P for commit duration. Hence, no other client transaction can fetch or update records in P until T commits.

Similarly, in the case of the operation Delete[k, x], the X-locked page P is searched for k. If no record with key value k is found in P, then the delete operation is terminated and the exception “record not found” is returned. Otherwise, T deletes the record from P, and holds the X lock on P for commit duration to prevent other client transactions from fetching or updating records in P before it commits.

No additional page locks are acquired for operations done in the backward-rolling phase of an aborted transaction T. To undo an insert (resp. a delete) of a record r, T

just deletes (resp. inserts) r under the protection of the X lock acquired on the covering page P during the forward-rolling phase.

Commit-duration S locks acquired by a client transaction T on the leaf pages covering the fetched records guarantee that no other client transaction can update records in these pages until T commits. Similarly, commit-duration X locks acquired by T on the updated leaf pages guarantee that no other client transaction can fetch or update records in those pages, until T commits.

Lock tables at the server and clients may be implemented as hash tables, because hash tables are fast for content retrieval. As *lock names* have to be of fixed length, we hash the Page-id to get a fixed-length lock name. That is, we form the lock name as follows.

Page-lock-name := Hash(Page-id).

3.6 B±tree Structure-Modification Operations and Logging

The structure of a B±tree is modified by the following structure-modification operations. Each operation affects three pages on two adjacent levels of the tree, produces a structurally consistent and balanced B±tree and is logged using a single redo-only log record. We assume that any step in the algorithms below that includes the sending of a newly generated log record and the associated updated page to the server also includes the sending of all log records up to the newly generated one together with associated (leaf) pages that have not yet been sent. Note that updates on leaf pages are not immediately propagated.

Split(P,Q). Given are the Page-ids of an X-locked parent page P and a U-locked child page Q, where P is not full and Q is full. The algorithm allocates a new page Q´, splits Q into Q and Q´ and links Q´ to its parent P. Upon return, the parameter Q denotes the page (Q or Q´) that covers the search key value. That page is kept X-locked while the locks on the other two pages are released.
Step 1. Request the server to get an X lock on the storage-map page M and a copy of M, to get an X lock on some page Q´ marked as unallocated in M, and to upgrade the U lock on Q to an X lock. When the request is granted, X-lock M, Q and Q´ locally.
Step 2. Mark Q´ as allocated in M and format Q´ as an empty B±tree page.
Step 3. Let u´ be the current highest key value in Q and let u be the key value that splits Q evenly.
Step 4. Move all the records with key values greater than u from Q to Q´ and keep the records with key values less than or equal to u in Q.

Step 5. Change the index record (u´, Q) associated with the child page Q in the parent P into (u, Q) and inserts the index record (u´, Q´) associated with new child page Q´ into the parent P.
Step 6. If Q is a leaf page, then set the Page-link of Q´ := the Page-link of Q, the Page-link of Q := Q´, and set *l* := the Page-link of Q´. Otherwise, set *l* := nil.
Step 7. Generate the redo-only log record <T, split, Q, u, Q´, u´, *l* , V, n> where V is the set of records moved from Q to Q´ and n is the LSN of the previous log record generated by T. Update the Page-LSNs of M, P, Q and Q´.
Step 8. Send copies of the updated pages M, P, Q and Q´, and the log record to the server with a request to release the U lock on M, to release the X lock on P, and to release the X lock on the page (Q or Q´) that does not cover the search key value. When the request is granted, release the local U lock on M, release the local X lock on P, and release the local X lock on the page (Q or Q´) that does not cover the search key value.
Step 9. Set Q := the page (Q or Q´) that covers the search key value. □

Merge(P,Q,R). Given are the Page-ids of an X-locked parent page P and its U-locked child pages Q and R, where P is not about to underflow, R is the right sibling of Q, and the records in Q and R all fit in a single page. The algorithm merges R into Q and deallocates R. Upon return, Q remains X-locked while the locks on P and R are released.
Step 1. Request the server to get an X lock on the storage-map page M and a copy of M, and to upgrade the U locks on Q and R to X locks. When the request is granted, X-lock M, Q and R locally.
Step 2. Move all the records from R to Q and mark R as unallocated in M.
Step 3. If Q is a leaf page, then set the Page-link of Q := the Page-link of R and set *l* := the Page-link of R. Otherwise, set *l* := nil.
Step 4. Let (u, Q) and (v, R) be the index records associated with the child pages Q and R, respectively in the parent P.
Step 5. Unlink the child page R from its parent P by deleting the index record (v, R) from P and changing the index record (u, Q) in P into (v, Q).
Step 6. Generate the redo-only log record <T, merge, Q, u, R, v, *l* , V, n> where V is the set of records moved from R to Q and n is the LSN of the previous log record generated by T. Update the Page-LSNs of M, P, Q and R.
Step 7. Send copies of the updated pages M, P, Q and R, and the log record to the server with a request to release the U lock on M and to release the X locks on P and R. When the request is granted, release the local U lock on M and release local X locks on P and R. □

Redistribute(P,Q,R). Given are the Page-ids of an X-locked parent page P and its U-locked child pages Q and R, where R is the right sibling of Q, and the pages Q and

R cannot be merged. The algorithm redistributes the records in Q and R evenly. Upon return, the parameter Q denotes the page (Q or R) that covers the search key value; that page remains X-locked while the locks on the other two pages are released.
Step 1. Request the server to upgrade the U locks on Q and R to X locks. When the request is granted, upgrade the local U locks on Q and R to X locks.
Step 2. Let u be the current highest key value in Q and let u´ be the key value in the page (Q or R) that redistributes the records in these pages evenly.
Step 3. If u´ > u, then move all the records with key values less than or equal to u´ from R to Q. Otherwise, move all the records with key values greater than u´ from Q to R.
Step 4. Change the index record (u, Q) associated with the child page Q in the parent P to (u´, Q).
Step 5. Generate the redo-only log record <T, redistribute, P, Q, u´, R, V, Y, n> where V is the set of the records moved and Y is the page (Q or R) that received the records in V and n is the LSN of the previous log record generated by T. Update the Page-LSNs of P, Q and R.
Step 6. Send copies of the updated pages P, Q, R and the log record with a request to release the X lock on P and to release the X lock on the page (Q or R) that does not cover the search key value. When the request is granted, release the local X lock on P and release the local X lock on the page (Q or R) that does not cover the search key value.
Step 7. Set Q := the page (Q or R) that covers the search key value. □

Increase-tree-height(P). Given is the Page-id of the X-locked full root page P. The algorithm allocates two pages P´ and P´´, distributes the records in P evenly between P´ and P´´ and makes P´ and P´´ children of P. Upon return, the parameter P denotes the page (P´ or P´´) that covers the search key value. That page is kept X-locked, while the locks on the other two pages are released.
Step 1. Request the server to get an X lock on the storage-map page M and a copy of M, and to get X locks on some pages P´ and P´´ marked as unallocated in M. When the request is granted, X-lock M, P´ and P´´ locally.
Step 2. Mark P´ and P´´ as allocated in M and format P´ and P´´ as empty B±tree pages.
Step 3. Determine the key value u that splits P evenly.
Step 4. Move all the records with key values greater than u from P to P´´.
Step 5. Move all the remaining records from P to P´.
Step 6. Insert the records (u, P´) and (∞, P´´) into P.
Step 7. Generate the redo-only log record <T, increase-tree-height, P, P´, u, P´´, ∞, V1, V2, n> where u and ∞ are the high-key values of P´ and P´´ respectively, V1 is the set of records that moved from P to P´ and V2 is the set of records that moved

from P to P´´, and n is the LSN of the previous log record generated by T. Update the Page-LSNs of M, P, P´, and P´´.
Step 8. Send copies of the updated pages M, P, P´ and P´´, and the log record to the server with a request to release the U lock on M, to release the X lock on P, and to release the X lock on the page (P´ or P´´) that does not cover the search key value. When the request is granted, release the local U lock on M, release the local X lock on P, and release the local X lock on the page (P´ or P´´) that does not cover the search key value.
Step 9. Set P:= the page (P´ or P´´) that covers the search key value. □

Decrease-tree-height(P,Q,R). Given are the Page-ids of the X-locked root page P and its U-locked child pages Q and R, where Q and R are the only children of P and can be merged. The algorithm moves the records in Q and R to P and deallocates Q and R. Upon return, P remains X-locked while the locks on the other two pages are released.
Step 1. Request the server to get an X lock on the storage-map page M and a copy of M, and to upgrade the U locks on Q and R to X locks. When the request is granted, X-lock M, Q and R locally.
Step 2. Delete the only two remaining index records (u, Q) and (∞, R) associated with the child pages Q and R from the parent page P.
Step 3. Move all the records from Q and R to P, and mark Q and R as unallocated in M.
Step 4. Generate the redo-only log record <T, decrease-tree-height, P, Q, u, R, ∞, V1, V2, n> where V1 is the set of records moved from Q to P, V2 is the set of records moved from R to P, and n is the LSN of the previous log record generated by T. Update the Page-LSNs of M, P, Q and R.
Step 5. Send copies of the updated pages M, P, Q and R, and the log record to the server with a request to release the U lock on M and to release the X locks on Q and R. When the request is granted, release the local U lock on M and release the X locks on Q and R. □

Lemma 3.2. Let B be a structurally consistent and balanced B±tree. Then any of the structure-modification operations split(P,Q), merge(P,Q,R), redistribute(P,Q,R), increase-tree-height(P) and decrease-tree-height(P,Q,R), whenever the preconditions for the operation hold, produces a structurally consistent and balanced B±tree when run on B by some client transaction T. When the operation is terminated, the pages modified by the operation are in the server's database buffer and the redo-only log record generated for the operation is in the server's log buffer.

Proof. From the algorithms for the operations, it is evident that if the preconditions for the operation are satisfied, then the operation can be run on B and preserves the

structural consistency and balance of B. The preconditions for split(P,Q) state that Q is full and its parent P is not full. In Split(P,Q), Q is split into Q and the allocated page Q´, and Q´ is made as a child of P. Thus the resulting B±tree is structural consistent and balanced. The preconditions for merge(P,Q,R) and redistribute(P,Q,R) state that R is a right sibling of Q and their parent P is not about to underflow. In merge(P,Q,R), R is merged into Q and R unlinked from its parent P, while in redistribute(P,Q,R), the records in Q and R are redistributed evenly, and hence the resulting B±tree in either case is structurally consistent and balanced. The preconditions for increase-tree-height(P) state that P is the root of the tree and full. The operation distributes the records in P evenly between the allocated pages P´ and P´´ and makes P´ and P´´ as children of P. Thus the resulting B±tree is structurally consistent and balanced. The preconditions for decrease-tree-height(P,Q,R) state that Q and R are the only children of P, R is an about-to-underflow right sibling of Q, and the records in both pages Q and R can fit in one page. The operation replaces the contents of P by the contents of Q and R, and hence the resulting B±tree is structurally consistent and balanced.

Concurrent B±tree structure-modification operations performed by client transactions preserve the structural consistency of the tree, because the tree pages involved in each operation are kept X-locked for the duration of the operation. Finally, as seen from the algorithms of the operations, both the modified pages and the generated redo-only log record are shipped to the server at the end of the operation. □

3.7 B±tree Distributed Traversals

A client transaction T takes as input a key value k and *traverses* the B±tree, using the lock-coupling protocol with S locks (Algorithm 3.1). When T needs to read a cached page P, then T S-locks P locally and reads P without server intervention (guaranteed by the callback-read protocol). If P is not resident in the client cache, then T requests from the server a copy of P. When the request is granted, P is S-locked locally, and read by T. When the leaf page P covering the search key value k is reached, then P is S-locked locally for fetching and X-locked locally for updating, after acquiring an X lock from the server.

Algorithm 3.1. B±tree traversal using lock coupling with S locks.
Step 1. Set P:= the Page-id of the root page of the B±tree.
Step 2. If P is cached locally, then S-lock P locally. Otherwise, request from the server a copy of P. When the request is granted, S-lock P locally.
Step 3. If P is a leaf page, then go to Step 6.

Step 4. Search P for the child page Q covering the search key value k. If Q is cached locally, then S-lock Q locally. Otherwise, request from the server a copy of Q. When the request is granted, S-lock Q locally.
Step 5. Unlock P locally, set P:= Q, and go to Step 3.
Step 6. Now P is the leaf page that covers the search key value k. If P is needed for updating, then request from the server an X lock on P. When the request is granted, X-lock P locally. □

If an updating client transaction T traversing the B±tree reaches a full leaf page P (when inserting) or an about-to-underflow leaf page P (when deleting), then T requests the server to release the X lock on P (to avoid a deadlock with another client transaction traversing the tree). When the request is granted, T releases the local X lock on P and retraverses the B±tree, using Algorithm 3.2 which results in U-locking the reached leaf page P and its ancestors up to the first non-full (non-underflow) page.

Algorithm 3.2. B±tree traversal using lock coupling with U locks for updating.
Step 1. Set P := the Page-id of the root page of the B±tree. If P is not U-locked locally for T, then request the server to get a U lock on P, and U-lock P locally when the lock is granted.
Step 2. If P is a leaf page, then request the server to upgrade the U lock on P to an X lock. When the request is granted, upgrade the local U lock on P to an X lock, and return.
Step 3. Search P for the child page Q covering the search key value k. Request the server to get a U lock on Q, and U-lock Q locally when the lock is granted.
Step 4. If the child page Q is full (when inserting) or about to underflow (when deleting), then save the Page-id of P, set P:= Q, and go to Step 2.
Step 5. If the child page Q is not-full (resp. not-about-to-underflow) and some ancestors of Q (whose Page-ids were saved previously) are still U-locked, then request the server to release the U locks on the ancestors of Q. When the request is granted, release the local U locks on these pages.
Step 6. Request the server to release the U lock on page P. When the request is granted, release the local U lock on P, set P:=Q, and go to Step 2. □

3.8 Transactional Isolation by Leaf-Page Locks

A record in a leaf page P updated by client transaction T can later be moved to another page by a structure modification performed by T. Therefore, in order to adapt the structure-modification operations of Section 3.6 to the page-level-locking protocol in a PS-PP page-server system, the following changes are needed in the structure-modification operations:

Split(P,Q): If Q is a leaf page S-locked or X-locked by T for commit duration, then both Q and the newly allocated page Q´ must be locked by T in the same mode for commit duration.

Merge(P,Q, R): If Q and R are leaf pages and one of them is S-locked or X-locked by T for commit duration then Q must be locked by T in the same mode for commit duration.

Redistribute(P,Q,R): If Q and R are leaf pages and one of them is S-locked or X-locked by T for commit duration then that lock must be retained for commit duration. Moreover, if the redistribution involves moving records from Q to R (resp. from R to Q) and if Q (resp. R) is locked for commit duration then also R (resp. Q) must be locked by T in the same mode for commit duration.

Increase-tree-height(P): If P is a leaf page and P is S-locked or X-locked by T for commit duration, then the lock on P (if any) is changed to commit-duration locks on P´ and P´´.

Decrease-tree-height(P,Q,R): If Q and R are leaf pages and one of them is S-locked or X-locked by T for commit duration, then the lock on one of these pages (if any) is changed to a commit-duration lock on P.

In the traversal algorithms 3.1 and 3.2, the action of releasing a page-level lock should never imply releasing a commit-duration lock.

3.9 B±tree Distributed Fetch, Insert and Delete

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

Algorithm 3.3. Given a key value $u < \infty$, fetch the data record r with the least key value k satisfying kθu. As a result of a previous call to this algorithm, the Page-id P of the page where the previously fetched record resided may also be given as input.
Step 1. If no P is given or if P is not cached and S-locked by T, then go to Step 4. Otherwise, determine the lowest key value w, the highest key value v, and the high-key value v´ in P.
Step 2. If $u < w$ or $u > v´$, then go to Step 4.
Step 3. If $u > v$ or $u = v$ and θ = ”>”, then go to Step 5. Otherwise go to Step 6.
Step 4. Traverse the B±tree from the root using lock coupling with S locks (Algorithm 3.1) with u as the input key value. Search the reached leaf page P, determine the highest key value v and the high-key value v´ in P, and go to Step 3.

Step 5. The record to be fetched resides in P′, the page next to P. If P′ is cached locally, then S-lock P′ locally. Otherwise, request from the server a copy of P′. When the request is granted, install the received copy of P′ in the cache, S-lock P′ locally, and set P:= P′.
Step 6. Now P is the page that contains the record to be fetched. Determine the record r in P with the least key value k satisfying kθu. Save the page-id P, hold the S lock on P for commit duration, and return with r. □

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

Algorithm 3.4. Given a record r = (k, x), insert r into the database.
Step 1. Traverse the B±tree, using lock coupling with S locks and acquire an X lock on the leaf page P that covers k (Algorithm 3.1).
Step 2. If P is full, then go to Step 5.
Step 3. Search page P for the position of insertion of record r. If a key value that matches the key value k is found, then return the "uniqueness violation" status, terminate the insert operation, and return.
Step 4. Insert r into P, generate a redo-undo log record <T, insert, P, r, n> where n is the LSN of the previous log record generated by T, update the Page-LSN of P, hold the X lock on P for commit duration, and return.
Step 5. Request the server to release the X lock on page P to avoid a deadlock with another client transaction traversing the B±tree. When the request is granted, release the local X lock on P, retraverse the B±tree, using lock coupling with U locks (Algorithm 3.2), and set P:= Q (the reached leaf page).
Step 6. If P is not full, then go to Step 3. Otherwise, perform the needed page splits using Algorithm 3.6, and go to Step 3. □

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

Algorithm 3.5. Given a key value k, delete the record r = (k, x) with key value k from the database.
Step 1. Traverse the B±tree, using lock coupling with S locks and acquire an X lock on the leaf page P that covers k (Algorithm 3.1).
Step 2. If P is about to underflow, then go to Step 5.
Step 3. Search page P for the record r with key value k. If no such record is found in P, then terminate the delete operation and return the exception "record not found".
Step 4. Delete r from P, generate the redo-undo log record <T, delete, P, r, n> where n is the LSN of the previous log record generated by T, update the Page-LSN of P, hold the X lock on P for commit duration, and return.

Step 5. Request the server to release the X lock on page P. When the request is granted, release the local X lock on P and retraverse the B±tree using lock coupling with U locks (Algorithm 3.2), and set P:= the reached leaf page Q.
Step 6. If P is not about to underflow, then go to Step 3. Otherwise, perform the needed page merging (or redistributing) using Algorithm 3.7, and go to Step 3. □

The following algorithm is used by a client transaction to execute the needed page splits along the search path in top-down manner.

Algorithm 3.6. Split the full pages on the split path starting from the highest non-full page down to the target leaf page P when these pages are U-locked by T.
Step 1. Let P := the Page-id of the highest U-locked page on the split path.
Step 2. Request the server to upgrade the U lock on P to an X lock. When the lock is granted, upgrade the local U lock on P to an X lock.
Step 3. If P is the root of the B±tree and P is full, then increase-tree-height(P).
Step 4. If P is a leaf page, then return.
Step 5. Determine the U-locked full child page Q of P that is on the split path.
Step 6. Split(P,Q), set P := Q, and go to Step 4. □

The following algorithm is used by a client transaction to perform page merging (or redistributing) along the search path starting from the highest not-about-to-underflow page down to the target leaf page P.

Algorithm 3.7. Merge or redistribute the pages along the merge (or redistribute) path which are about to underflow with their sibling pages when these pages are U-locked by T.
Step 1. Let P be the Page-id of the highest U-locked not-about-to-underflow page on the merge (or redistribute) path.
Step 2. Request the server to upgrade the U lock on P to an X lock. When the request is granted, upgrade the local U lock to an X lock.
Step 3. Determine the U-locked child page Q of P that is about to underflow and is on the merge (or redistribute) path.
Step 4. If Q is the rightmost child of its parent P, then go to Step 8.
Step 5. Determine the right sibling page R of Q. Request the server to get a U lock on R and a copy of R, and U-lock R locally when the lock is granted.
Step 6. If P is the root of the B±tree and P has just two child pages Q and R which can be merged, then decrease-tree-height(P,Q,R) and go to Step 12.
Step 7. If Q and R can be merged, then merge(P,Q,R) else redistribute(P,Q,R). Set P := Q and go to Step 12.
Step 8. Now Q is the rightmost child of its parent P. Determine the left sibling page L of Q.

Step 9. Release the local U lock on Q and request the server to release the U lock on Q (to avoid a deadlock) and to get U locks on L and Q (in this order). When the locks are granted, U-lock L and Q locally.
Step 10. If P is the root of the B±tree and P has just two child pages L and Q which can be merged, then decrease-tree-height(P,L,Q) and go to Step 12.
Step 11. If L and Q can be merged, then merge(P,L,Q) else redistribute(P,L,Q). Set P := L.
Step 12. If P is not a leaf page, then go to Step 3. □

Lemma 3.3. Let B be a structurally consistent and balanced B±tree of height h. Assume that a client transaction T performs tree-structure modifications on B using Algorithm 3.6 or 3.7. Then T accesses 2h+1 pages at most, keeps at most h pages U-locked and three pages X-locked at a time, and accesses and X-locks the storage-map page M at most h times.

Proof. It is clear from the algorithms of the structure-modification operations that each operation acquires three X locks on the pages involved in the operation. Hence, when transaction T performs tree-structure modifications using Algorithm 3.6 or 3.7, T X-locks three pages at a time, two of which are on the same level. Thus in the worst case, Algorithm 3.6 or 3.7 accesses and modifies 2h+1 pages. The rest of the pages on the structure-modification path are kept U-locked. The storage-map page is accessed when a new page needs to be allocated in the operations split(P,Q) and increase-tree-height(P) and when a page needs to be deallocated in the operations merge(P,Q,R) and decrease-tree-height(P,Q,R). Hence in the worst case, the storage-map page M may be accessed h times. □

Lemma 3.4. Let B be a structurally consistent and balanced B±tree. Then the read-mode traversal (Algorithm 3.1), the update-mode traversal (Algorithm 3.2) and the tree-structure-modifications (Algorithms 3.6 and 3.7) are deadlock-free.

Proof. The read-mode traversal (Algorithm 3.1) and the update-mode traversal (Algorithm 3.2) employ the lock-coupling protocol and acquire page locks in a top-down order. In the Algorithms 3.6 and 3.7, the U locks on the pages involved in the tree structure modifications are acquired in a top-down, left-to-right order. When an about-to-underflow page Q is the rightmost child of its parent P, the U lock on Q is released and U locks are acquired on the pages L (the left sibling of Q) and Q (see Steps 8 and 9 of Algorithm 3.7). Again in Algorithms 3.6 and 3.7, only U locks are upgraded to X locks. Therefore, we may conclude that our read-mode-traversal, update-mode traversal and tree structure modification algorithms are deadlock-free. □

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

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

The operations in the backward-rolling phase of an aborted transaction are implemented by the algorithms below. The algorithms are used during normal processing to roll back a transaction as well as during the undo pass of restart recovery to roll back all active transactions. An undo operation in the backward-rolling phase of an aborted transaction T is logged by writing a *compensation log record* (CLR) [Moha92a] that contains, besides the arguments needed to replay the undo operation, the LSN of the log record for the next database operation by T to be undone. Such a compensation log record is a redo-only log record. No new page locks are needed to roll back an aborted client transaction T, because the rollback is done under the protection of the page locks acquired by T in its forward-rolling phase.

When a client transaction T contains several insert or delete operations, then uncommitted updates by T can be moved to a different leaf page by structure modifications triggered by T. 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* [Moha92b, Moha96] is used: the backward-rolling client transaction retraverses the 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. As we shall see in Chapter 5, in a PS-AA page-server system a logical undo may also be needed in the case of client transactions whose forward-rolling phase contains only a single update operation. This is because uncommitted updates by one transaction may migrate due to structure modifications by other transactions when leaf pages are locked only for short duration. Logical undo provides a higher level of concurrency than would be possible if the system only allowed for page-oriented undo.

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

Algorithm 3.8. Undo the deletion of record r = (k, x). Given is a redo-undo log record <T, delete, P, r, n> where n is the LSN of the previous log record generated by T in the forward-rolling phase. The algorithm reinserts r into P if P still covers k. Otherwise, T retraverses the B±tree from the root page to reach the covering leaf page and inserts r there.
Step 1. Check if physiological undo is possible, that is, if P still covers k and there is a room for r in P. Note that T still holds a commit-duration X lock on the page that covers k. If physiological undo is possible, then go to Step 4.
Step 2. Undo the deletion of r logically. This step is needed only when T itself has performed a structure modification that has changed the key range covered by P. Retraverse the B±tree from the root page, using Algorithm 3.2 with k as an input key value to reach the covering leaf page Q, and set P:= Q.
Step 3. If P is full, then perform the needed page splits using Algorithm 3.6.
Step 4. Insert r into P, generate the compensation log record <T, undo-delete, P, r, n>, and update the Page-LSN of P. □

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

Algorithm 3.9. Undo the insertion of record r = (k, x). Given is a redo-undo log record <T, insert, P, r, n> where n is the LSN of the previous log record generated by T in the forward-rolling phase. The algorithm deletes r from P if P still holds r. Otherwise, T retraverses the B±tree from the root page to reach the covering leaf page Q and deletes r from there.
Step 1. Check if physiological undo is possible, that is, if P still contains r and will not underflow if r is deleted. Note that T still holds a commit-duration X lock on the page that holds r. If physiological undo is possible, then go to Step 4.
Step 2. Undo the insertion of r logically. This step is needed only when T itself has performed a structure modification that has moved r from its original page to another. Retraverse the B±tree from the root page, using Algorithm 3.2 with k as an input key value to reach the covering leaf page Q, and set P:= Q.
Step 3. If P is about to underflow, then perform the needed page merging (or redistributing) using Algorithm 3.7.
Step 4. Delete r from P, generate the compensation log record <T, undo-insert, P, r, n>, and update the Page-LSN of P. □

Lemma 3.5. Let T be a client transaction and B a structurally consistent B±tree of height h. Any operation Fetch[k,θu, x] by T on B accesses at most h+1 pages of B

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

Proof. From Algorithms 3.1 and 3.3, it follows that Fetch[k,θu, x] accesses h pages to reach leaf-page level while keeping two pages S-locked at a time. When a leaf page is reached, an extra page access may be needed to locate the least key value k satisfying kθu. The claims for Insert[k, x], Delete[k, x], Undo-delete[k, x] and Undo-insert[k, x] follow from Lemma 3.3 and Algorithms 3.4, 3.5, 3.8 and 3.9. A physiological Undo-delete[k, x] always checks that the page has room for (k, x), and a physiological Undo-insert[k, x] always checks that the page will not underflow from the deletion of (k, x). □

3.11 Transaction Abort and Rollback in PS-PP

A client transaction T is rolled back during normal processing when T performs the "abort" operation or T gets involved in a deadlock or the flushing process of the log records of T is not completed successfully. During the rollback, the log records are undone in the reverse chronological order, and for each log record that is undone, a compensation log record (CLR) is generated. The backward-rolling phase $A\alpha^{-1}R$ of an aborted transaction T = $B\alpha A\alpha^{-1}R$ is executed by the following algorithm.

Algorithm 3.10. Rollback an aborted transaction T.
Step 1. Generate the log record <T, abort, n> where n is the LSN of the last log record generated by T during its forward-rolling phase.
Step 2. Get the log record generated by T with LSN equal to n.
Step 3. If the log record is <T, begin>, then go to Step 7.
Step 4. If the log record is a redo-only log record of type "split" or "merge" or redistribute" or "increase-tree-height" or "decrease-tree-height", then use the Prev-LSN of this log record to determine the next log record to be undone, and go to Step 3.
Step 5. If the log record is a redo-undo log record of type "delete" or "insert", then undo the logged update, using Algorithm 3.8 or 3.9, respectively.
Step 6. Get the next log record to be undone using the Prev-LSN of the log record being undone, and go to Step 3.

Step 7. Generate the log record <T, rollback-completed> and send it to the server with a request to release all locks of T.
Step 8. When the server receives the log record <T, rollback-completed>, the server flushes all the log records up to and including this log record, releases the locks of T, and informs the client.
Step 9. When the client is acknowledged about the successful flushing of the log records of T, the client releases the local locks of T, and discards the log records of T. □

When the client A where a transaction T is running fails, then the server just removes the entry associated with T from the transaction-table.

3.12 Transaction Execution in PS-PP

In the forward-rolling phase of client transaction T, the operations fetch, insert and delete are implemented by the algorithms of Section 3.9, while the operations begin and commit are implemented by the following algorithms.

Begin transaction: A client initiates a transaction by sending a begin-transaction request to the server, the server responds by assigning a new transaction-id T. The client generates the log record <T, begin> when the request is granted. □

Commit transaction T: Generate the log record <T, commit>, and send all copies of updated pages and the log records including the commit log record to the server with request to release all locks held by T. □

In the backward-rolling phase of client transaction T, the operations Undo-delete[k, x] and Undo-insert[k, x] are executed by the algorithms of Section 3.10, while the operations abort and rollback-completed are executed by the following algorithms.

Abort transaction T: Generate the log record <T, abort, n> where n is the LSN of the previous log record generated for database operation (begin, insert or delete) by T. □

Rollback-completed transaction T: Generate the log record <T, rollback-completed> and send it to the server with a request to release all locks of T. □

Theorem 3.6. If the database operations Fetch, Insert, Delete, Undo-delete and Undo-insert are implemented by the algorithms of Sections 3.9 and 3.10 and if each client transaction accesses database records in an ascending key-value order, then our B±tree algorithms for a PS-PP page-server system are deadlock-free.

Proof. The assumption that client transaction access records in an ascending key-value order is needed to avoid deadlocks that may result when two transactions access two or more records in a different order. Therefore, the result follows from the assumption and Lemma 3.4. □

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

Theorem 3.7. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back client transactions that can be run on database D. Further let B be a structurally consistent and balanced B±tree in a PS-PP page-server system with db(B) = D, and let H´ be an implementation of H on B using the algorithms presented in this chapter. Then H´ produces a structurally consistent and balanced B±tree. The effects of all record inserts and deletes on B by the committed and rolled-back transactions in H, together with their log records, as well as the effects of all structure modifications on B by all the transactions in H, together with their log records, are found at the server.

Proof. A formal proof would use induction on the number of operations in H´. For the purposes of the proof we may regard each of the structure-modification operations split(P,Q), merge(P,Q,R), redistribute(P,Q,R), increase-tree-height(P) and decrease-tree-height(P,Q,R) as an atomic operation. Also the insertion of a record (k, x) into a leaf page P, and the deletion of a record from leaf page P are atomic operations. This is justified because the client transaction doing one of these operations will hold X locks on all the pages modified by the operation. The structural consistency and balance of the tree produced by H´ follows from Lemmas 3.2, and 3.5. As stated in Lemma 3.2, the effects of any structure-modification operation is propagated immediately to the server when the operation is completed. The propagation of record inserts and deletes however is not immediate and may be delayed until the transaction commits or completes its rollback. □

Theorem 3.8. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back client transactions that can be run on database D and let H_g be a completed history for H. Further let B be a structurally consistent and balanced B±tree in a PS-PP page-server system with db(B) = D, and let H´ be an

implementation of H on B using the algorithms presented in this chapter. Then there is a B±tree operation string g' such that H$'g'$ is an implementation of Hg on B. Moreover, if each transaction in H contains at most one update operation (i.e., an insert or a delete of a record), then all the inverse operations in g are performed physiologically in g'.

Proof. In one such g', the implementations of individual inverse operations Undo-insert[k, x] and Undo-delete[k, x] by different aborted transactions are implemented logically and are run serially, so that each inverse operation includes an update-mode traversal of the tree and the implementations of the operations are not interleaved. Thus, no lock conflicts can occur during the traversals at the non-leaf levels, nor can such conflicts occur at the leaf level because the aborted transaction is holding a commit-duration X lock on the leaf page, acquired for the corresponding forward-rolling operation. Recall that if the leaf page that received an update by a client transaction T is later split or redistributed by T, then also the other page, when it receives records from the updated page, is X-locked by T for commit duration.

A client transaction T that contains only one record insert or delete performs all its structure modifications before that insert or delete, as is indicated by Algorithms 3.4, 3.5, 3.6 and 3.7. Thus, after the insert or delete is done, T itself cannot change the leaf page P that received the update. As P is X-locked by T for commit duration no other transaction can access P either while T is active. Therefore, if T is active in H, the update by T (if any) can be undone physiologically on P (see Algorithms 3.8 and 3.9), in any order with respect to the inverse operations of the other active transactions in H. □

3.13 Restart Recovery

The goal of recovery is to ensure that the B±tree only reflects the updates of committed transactions and none of those of uncommitted transactions. We use a redo and undo recovery protocol for handling the system failures. Our recovery protocol is based on ARIES/CS [Moha94] and ARIES/IM [Moha92b, Moha96], which support the steal and no-force policies for buffer management. We assume that no new transactions are accepted to the system during restart recovery until the execution of the recovery protocol is completed. The restart recovery protocol consists of three passes: an analysis pass, a redo pass, and an undo pass.

In the *analysis pass*, the log is scanned forward starting from the start-checkpoint log record of the last complete checkpoint, up to the end of the log. A *modified-page list* of pages that were potentially more up-to-date in the server buffer than in stable

storage and an *active-transaction list* of client transactions that were in progress (i.e., forward-rolling or backward-rolling) at the time of the crash are constructed, using the information in the checkpoint record and the encountered log records. The analysis pass also determines the *Redo-LSN*, i.e., the LSN of the earliest log record that needs to be redone. The Redo-LSN is the minimum *Rec-LSN* (recovery LSN) in the created modified-page list. In other words, the analysis pass determines the starting point of the redo pass in the log, and the list of the transactions that need to be rolled back or whose rollback need to be completed.

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

In the *undo pass*, all forward-rolling transactions are aborted and rolled back and the rollback of all backward-rolling transactions is completed. The log is scanned backward from the end of the log until all updates of such transactions are undone. When a log record of type "compensation" (CLR) is encountered, then no action is performed except that the value of the Undo-Next-LSN field of such a CLR is used to determine the next log record to be processed. When a redo-only log record of type "split" or "merge" or "redistribute" or "increase-tree-height" or "decrease-tree-height" is encountered, then no action is performed except that the value of the Prev-LSN of such log record is used to determine the next log record to be processed. When a redo-undo log record of type "delete" or "insert" is encountered, then the logged update is undone, using Algorithm 3.8 or 3.9, respectively, except that no X lock is acquired on the affected page.

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

the list as the next LSN of the log record to be processed. The next log record to be processed will be the log record with the maximum LSN on this list.

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

Example 3.1. Assume that during the normal processing the server log contains the following log records.

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

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

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

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

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

Theorem 3.9. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back client transactions that can be run on database D1 and let B1 be a structurally consistent and balanced B±tree at the server of a PS-PP system with db(B1) = D1. Further let H´ be an implementation of H on B1 using the algorithms presented in this chapter and let L be the sequence of log records sent to the server by the operations in H´. Given the prefix L1 of L stored in the stable log and the (possibly structurally inconsistent) disk version B2 of the B±tree at the time H´ has been run on B1, the redo pass of the ARIES algorithm will produce a structurally consistent and balanced B±tree B3 at the server, 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 $g´$ of B±tree operations that implements some completion string for H1.

Proof. The way in which updates and structure modifications are propagated to the server guarantees that the log records generated by each client 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 locks, the log records for the updates and modifications on each page P appear in the log in the order in which the corresponding operations appear in H. Thus, given any prefix L1 of the log L, the redo pass of the ARIES algorithm will produce the B±tree that is the result of running some prefix of H´ on B1. By Theorem 3.7, B1 is structurally consistent and balanced. The rest of the Theorem follows from Theorem 3.8. □

3.14 Discussion

The B±tree algorithms that we have we developed above for a PS-PP page server system perform concurrency control and replica management at the page level, protecting leaf-page fetch, insert and delete operations by commit-duration page locks and postponing the propagation of updates and structure modifications until the commit of the transaction or the completion of a structure modification. These properties make the algorithms most suitable for design applications in which concurrency between transactions running at different clients is not the most important issue. In a typical CAD/CAM application built on a data-shipping OODBMS, at the beginning of an editing session, pages containing objects to be edited are first unloaded from the database and shipped to the clients, to be edited there by several (long) transactions.

Chapter 4

B-link-Tree Concurrency Control and Recovery in a PS-PP System

To increase concurrency in a PS-PP page-server system, we design new B-link- tree algorithms. In these algorithms, the execution of non-leaf-level structure modifications is separated from the execution of the leaf-level updates that give rise to the structure modifications. This is possible because in a B-link tree the pages on each level are linked sideways from left to right and interior pages are allowed to have indirect children that are only accessible from an elder sibling page via the sideways link. We introduce seven structure-modification operations, each of which performs a single modification involving one or two pages on a single level of the tree. This is an improvement over the algorithms in Chapter 3, where each structure modification affects two levels of the tree. Each successfully completed structure modification brings the tree into a structurally consistent and balanced state whenever the tree was structurally consistent and balanced initially. This is an improvement over previous B-link-tree algorithms, which may result in a degenerate tree with long chains of underutilized pages and indirect children. In this chapter we use these B-link-tree algorithms to implement transactions in a PS-PP system in which transaction isolation is guaranteed by page-level locking. The algorithms are most suitable for the same kind of CAD/CAM applications as those of Chapter 3 but allow more concurrency.

4.1 Single-Level Structure Modifications and Leaf-Page Updating

In the B±tree algorithms we developed for PS-PP, an updating client transaction T executes a B±tree structure modification triggered by a record insert (delete) to a leaf page P to completion before it updates P. The execution of a structure modification involves X-locking three pages along the split/merge path and keeping all the pages in the rest of the path U-locked by T, thus reducing the concurrency in the system. In the worst case, a B±tree structure modification may need to keep

three pages X-locked and *h*-1 pages U-locked simultaneously, where h is the height of the B±tree. In the B-link-tree algorithms presented in [Lome92, Lome97], a tree-structure modification involves three pages at most. However, merging (or redistributing) two pages still needs three pages to be X-locked on two adjacent levels. Also a page split done on a single level of the tree is decoupled from the linking of the new child (resulted from the split) to its parent. Due to the decoupling, it is possible in the algorithms in [Lome92, Lome97] that arbitrary long chains of sibling pages are created that are not directly linked to their parents, and hence tree balance is not guaranteed.

We handle tree-structure modifications using an approach similar to that of [Lome92, Lome97], but we go further by making a structure modification to involve only a single level of the tree and X-locking two pages at most on that level. 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 tree locks. The execution of non-leaf-level structure modifications is separated from the execution of the leaf-level updates that give rise to the structure modifications as follows. When a client transaction T wants to insert (resp. to delete) a record r with key value k, T traverses the tree using the lock-coupling protocol with U locks to reach the target leaf page P and performs structure modifications along its search path, if there is a need for them. To insert a record r into a full leaf page P, T splits P by moving the upper half of the records into a new leaf page P´ and inserts r into the proper leaf page (P or P´). The action of linking of P´ to its parent will be executed by the next updating client transaction T´ traversing the same path and running into P.

4.2 B-link-Tree Structure-Modification Operations and Logging

We modify our B±tree to a *B-link tree* by linking from left to right the interior pages that are at the same level and setting the link of the right-most page at each level to nil. 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*. 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 which 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 which 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 Figure 4.1, 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 non-root page is a direct child of its parent, except the leaf page P8, which is an indirect child of its parent P3.

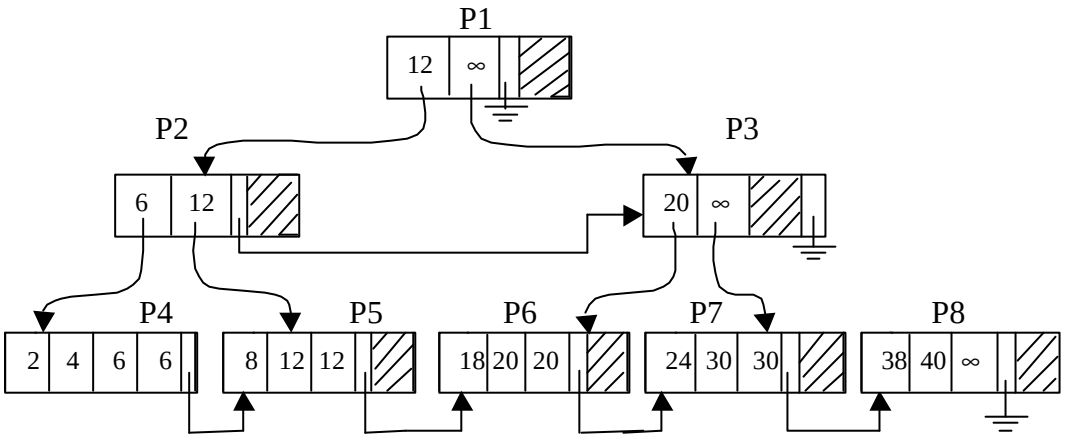

Figure 4.1. A B-link tree.

Let Q be a direct child of P and (v, Q) the index record in P associated with Q (Figure 4.2). 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 non-leaf 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 (Algorithm 4.2) and in the repair-page-underflow algorithm.

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 non-root pages are underflown and (2) no indirect child page has a right sibling page that is also an indirect child.

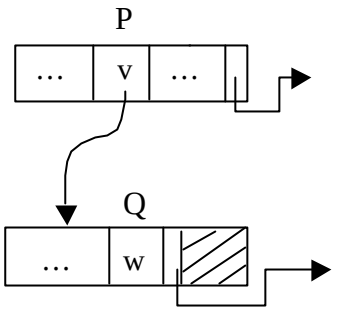

Figure 4.2. Page P and its direct child Q.

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 non-root leaf page and contains only m_1 database records, or (3) P is a non-root index page and contains only m_2 child links. A B-link-tree page P is *about to overflow* if there is no room in it for the insertion of a record.

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 can be deleted from P or inserted into P without causing P to underflow or overflow; 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 a record to be inserted).

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, turns 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.

The structure of a B-link-tree can be modified by one of seven structure-modification operations, *link, unlink, split, 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 and produces a structurally consistent and balanced B-link-tree as a result. As previously each structure-modification operation is logged using a single redo-only log record, and hence each successfully completed structure

modification is regarded as a committed “nested top action” and will not be undone even if the transaction that gave rise to it eventually aborts.

Below are algorithms that implement the structure-modification operations triggered by a client transaction T. As in the algorithms in Chapter 3, all page updates done in a structure modification are propagated to the server immediately after the structure modification is completed, whereas record inserts and deletes on leaf pages are not propagated until the transaction commits (or completes its rollback) or some structure modification follows, whichever happens first. As in Chapter 3, we assume that any step in the following algorithms that includes the sending of a newly generated log record and the associated updated page to the server also includes the sending of all log records up to the newly generated one together with associated (leaf) pages that have not yet been sent.

Link(P,Q,R): Make an indirect child page R a direct child of its parent P where Q is the left sibling of R (Figure 4.3). Given are the Page-id P of a U-locked parent page, the Page-id Q of a U-locked direct child of P, and the Page-id R of an indirect child of P where the high-key values of Q and R are u and v respectively. It is assumed that P can accommodate the index record (v, R). At the end of the link operation, P and Q remain U-locked.

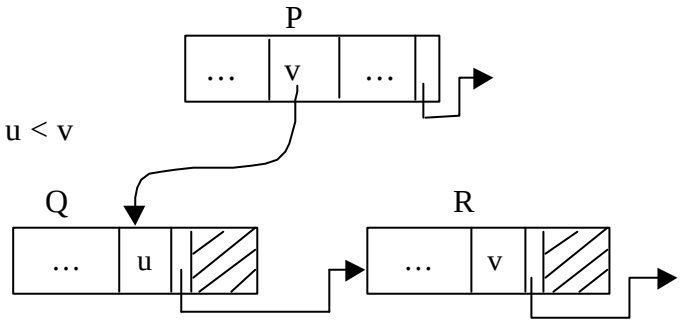

Figure 4.3. Page Q has a right sibling R that is an indirect child of its parent P.

Step 1. Request the server to upgrade the U lock on P to an X lock. When the request is granted, upgrade the local U lock on P to an X lock.
Step 2. Change the index record (v, Q) to (u, Q) in P and insert the index record (v, R) into P, see Figure 4.4.
Step 3. Generate the redo-only log record <T, link, P, Q, R, u, v, n> where n is the LSN of the previous log record generated by T, and update the Page-LSN of P.
Step 4. Send a copy of the updated parent page P and the log record to the server with a request to downgrade the X lock on P to a U lock. When the request is granted, downgrade the local X lock on P to a U lock. □

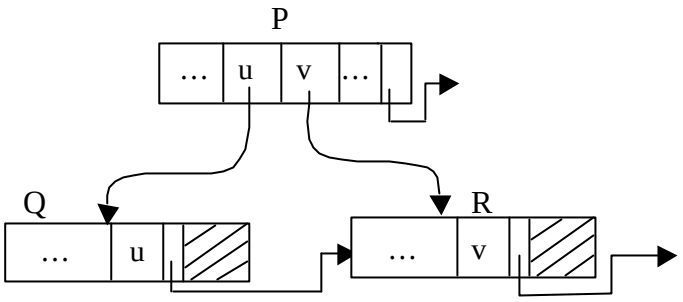

Figure 4.4. The right sibling R of Q is linked to the parent P.

Unlink(P,Q,R): Make a direct child page R an indirect child of its parent page P where Q is the left sibling of R. Given are the Page-id P of a U-locked parent page, the Page-ids Q and R of U-locked direct child pages of P where the high-key values of Q and R are u and v respectively (Figure 4.4). It is assumed that P is not about to underflow and that the right sibling of R (if any) is not an indirect child. The U locks on Q and R are needed to prevent another transaction from simultaneously splitting, merging (or redistributing) Q or R. At the end of the unlink operation, the lock on P is released while the U locks on the pages Q and R are retained.
Step 1. Request the server to upgrade the U lock on P to an X lock. When the lock is granted, upgrade the local U lock on P to an X lock.
Step 2. Delete the index record (v, R) from page P and change the index record (u, Q) to (v, Q) in P, see Figure 4.3.
Step 3. Generate the redo-only log record <T, unlink, P, Q, R, u, v, n> where n is the LSN of the previous log record generated by T and update the Page-LSN of P.
Step 4. Send a copy of the updated parent page P and the log record to the server with a request to release the X lock on P. When the request is granted, release the local X lock on P. □

Split(Q): Split the full page Q by allocating a new page Q´, by moving the upper half of the records from Q to Q´, and by making Q´ a right sibling of Q. Given is the Page-id Q of a U-locked full page (Figure 4.5). It is assumed that Q is safe. At the end of the split operation, the parameter Q denotes the page (Q or Q´) that covers the search key value; this page remains X-locked while the lock on the other page is released.

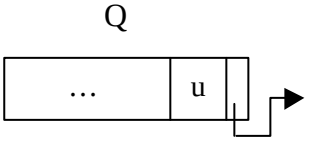

Figure 4.5. Page Q is full.

Step 1. Request the server to upgrade the U lock on Q to an X lock, to get an X lock on the storage-map page M and a copy of M, and to get an X lock on some page Q´

marked as unallocated in M. When the request is granted, X-lock M, Q and Q´ locally.
Step 2. Mark Q´ as allocated in M, and format Q´ as an empty B-link-tree page.
Step 3. Determine the key value u´ in Q that splits Q evenly.
Step 4. Move all the records with key values greater than u´ including the high-key value from Q to Q´, set the Page-link of Q´ := the Page-link of Q and keep the records with key values less than or equal to u´ in Q. Insert the high-key value u´ into Q if Q is a leaf page. Set the Page-link of Q := Q´ (Figure 4.6).

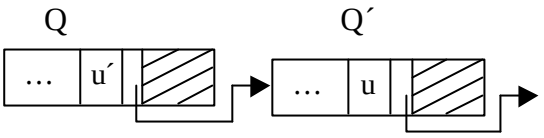

Figure 4.6. Page Q is split at key value u´ into Q and Q´.

Step 5. Generate the redo-only log record <T, split, Q, u´, Q´, V, n> where V is the set of records moved from Q to Q´ and n is the LSN of the previous log record generated by T, and update the Page-LSNs of M, Q and Q´.
Step 6. Send copies of the updated pages M, Q, Q´ and the log record to the server, with a request to release the U lock on M and to release the X lock on the page (Q or Q´) that does not cover the search key value. When the request is granted, release the local U lock on M, and release the X lock on the page (Q or Q´) that does not cover the search key value.
Step 7. Set Q:= the page (Q or Q´) that covers the search key value. □

Merge(Q,R): Merge page R into its left sibling page Q. Given are the Page-ids Q and R of U-locked sibling pages, such that R is an indirect child of the parent P of Q, where the high-key values of Q and R are u and v respectively (Figure 4.3). At the end of the merge operation, Q remains U-locked while the lock on R is released and R is deallocated.
Step 1. Request the server to upgrade the U locks on Q and R to X locks, and to get an X lock on the storage-map page M and a copy of M. When the request is granted, X-lock M, Q and R locally.
Step 2. If Q is a leaf page, then delete its high-key value. Move all records including the high-key value from R to Q, and set the Page-link of Q:= the Page-link of R (Figure 4.7).
Step 3. Mark R as unallocated in M.
Step 4. Generate the redo-only log record <T, merge, Q, R, V, n> where V is the set of the records moved from R to Q including the Page-link of R and n is the LSN of the previous log record generated by T, and update the Page-LSNs of M, Q and R.

Step 5. Send copies of the updated pages M, Q, R and the log record to the server with a request to release the X locks on M and R, and to downgrade the X lock on Q to a U lock. When the request is granted, release the local X locks on M and R, and downgrade the local X lock on Q to a U lock. □

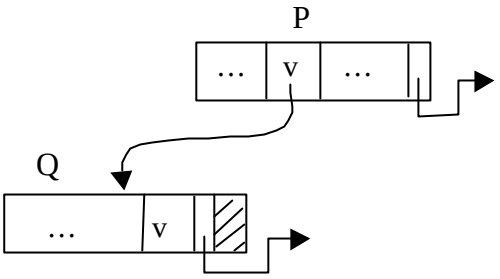

Figure 4.7. Page R (see Figure 4.3) has been merged into its left sibling page Q.

Redistribute(Q,R): Redistribute the records in sibling pages Q and R evenly. Given are the Page-ids Q and R of U-locked pages with high-key values u and v respectively, where R is the right sibling of Q, and R is an indirect child of the parent P of Q (Figure 4.3). At the end of the redistribute operation, the U-lock on the page (Q or R) that covers the search key value is retained while the lock on the other page is released and the parameter Q is set to the page (Q or R) that covers the search key value.
Step 1. Request the server to upgrade the U locks on Q and R to X locks. When the locks are granted, upgrade the local U locks on Q and R to X locks.
Step 2. Determine the key value v´ in the page (Q or R) that distributes the records in these pages evenly. Delete the high-key value of Q if Q is a leaf page.
Step 3. If v´ > u, then move the records with key values less than or equal to v´ from R to Q. Otherwise, move the records with key values greater than v´ from Q to R. In either case, if Q is a leaf page, set the high-key value of Q:= v´ (Figure 4.8).

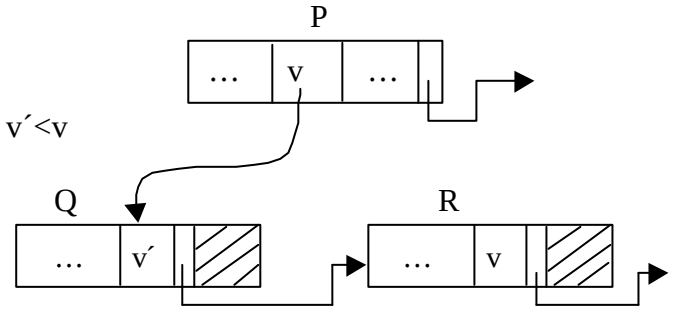

Figure 4.8. Records in Q and R (see Figure 4.3) have been redistributed.

Step 4. Generate the redo-only log record <T, redistribute, Q, R, V, Y, n> where V is the set of the records moved and Y is the page (Q or R) that received the records

in V and n is the LSN of the previous log record generated by T, and update the Page-LSNs of Q and R.
Step 5. Send copies of the updated pages Q and R and the log record to the server with a request to downgrade the X lock on the page (Q or R) that covers the search key value to a U lock, and to release the X lock on the other page. When the request is granted, downgrade the local X lock on the page (Q or R) that covers the search key value to a U lock, and release the local X lock on the other page.
Step 6. Set Q:= the page (Q or R) that covers the search key value. □

Increase-tree-height(P,P´): Increase the height of the B-link tree by one when the root page P has a right sibling P´. Given are the Page-ids P and P´ of the U-locked root page with high-key value u and the U-locked right sibling page with high-key value ∞ (Figure 4.9). At the end of the increase-tree-height operation, P remains as the root of the tree, the lock on P is released, the U lock on the child page of P (i.e., P´ or its newly allocated left sibling page P´´) that covers the search-key value is retained, the lock on the other child page is released, and the parameter P is set to the page (P´ or P´´) that covers the search-key value (Figure 4.10).

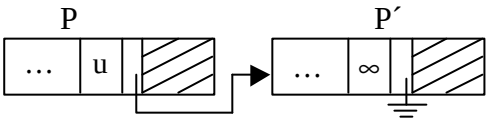

Figure 4.9. The root page P has a right sibling P´.

Step 1. Request the server to upgrade the U lock on P to an X lock, to get an X lock on the storage-map page M and a copy of M, and to get an X lock on some page P´´ marked as unallocated in M. When the request is granted, X-lock M, P and P´´ locally.
Step 2. Mark P´´ as allocated in M, and format P´´ as an empty B-link tree page.
Step 3. Move all index records from P to P´´, set the Page-link of P´´:= P´, insert the index records (u, P´´) and (∞, P´) into P, and set the Page-link of P := nil.

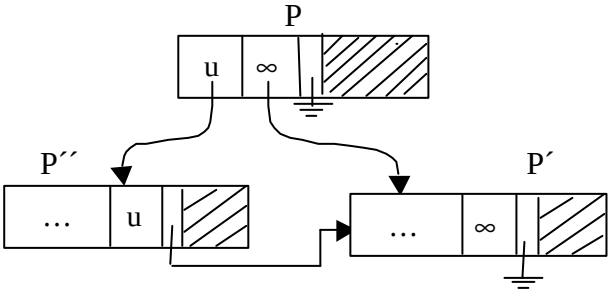

Figure 4.10. Tree height is increased by one and P remains as the root of the tree.

Step 4. Generate the redo-only log record <T, increase-tree-height, P, P′, P′′, V, n> where V is the set of index records moved from P to P′′ and n is the LSN of the previous log record generated by T. Update the Page-LSNs of M, P and P′′.
Step 5. If P′′ covers the search key value, then send copies of the updated pages M, P, P′′ and the log record to the server with a request to release the X locks on M and P, to release the U lock on P′, and to downgrade the X lock on P′′ to a U lock. When the request is granted, release the local X locks on M and P, release the local U lock on P′, downgrade the local X lock on P′′ to a U lock, set P:= P′′, and return.
Step 6. If P′ covers the search key value, then send copies of the updated pages M, P, P′′ and the log record to the server with a request to release the X locks on M, P and P′′. When the request is granted, release the local X locks on M, P and P′′, and set P:= P′. □

Decrease-tree-height(P,Q): Decrease the height of the B-link tree by one when the root page P has only one child page Q. Given are the Page-ids P and Q of U-locked pages such that P is the root of the B-link tree which has no right sibling, Q is the only child of P and Q has no right sibling (Figure 4.11). At the end of the decrease-tree-height operation, P remains as the root of the tree, the U lock on P is retained, and the lock on Q is released and Q is deallocated (Figure 4.12).

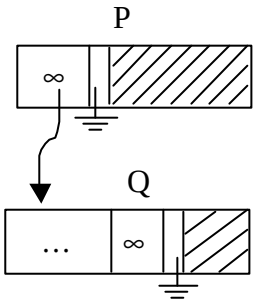

Figure 4.11. The root page P has no right sibling and P has only one child Q.

Step 1. Request the server to upgrade the U locks on P and Q to X locks, and to get an X lock on the storage-map page M and a copy of M. When the request is granted, X-lock M, P and Q locally.
Step 2. Delete the only remaining index record (∞, Q) from P, and move all index records from Q to P, see Figure 4.12.

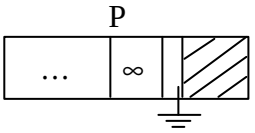

Figure 4.12. Tree height is decreased by one and P remains the root of the tree.

Step 3. Mark Q as unallocated in M.
Step 4. Generate the redo-only log record <T, decrease-tree-height, P, Q, V, n> where V is the set of index records moved from Q to P and n is the LSN of the previous log record generated by T, and update the Page-LSNs of M, P and Q.
Step 5. Send copies of the updated pages M, P, Q and the log record to the server with a request to release the X locks on M and Q, and to downgrade the X lock on P to a U lock. When the request is granted, release the local X locks on M and Q, and downgrade the local X lock on P to a U lock. □

Lemma 4.1. 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, produces a structurally consistent and balanced B-link tree when run on B by some client transaction T. When the operation is terminated, the pages modified by the operation are in the server's database buffer and the redo-only log record generated for the operation is in the server's log buffer.

Proof. From the algorithms for the operations we see immediately that if the preconditions for the operation are satisfied then the operation indeed can be run on B and retains 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 P´´ and P´ are linked as the only children of P, the result is 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 by different client transactions, the structural consistency and balance of the resulting tree is guaranteed by the fact that T keeps X-locked all the pages modified by the operation and U-locked any additional page involved that could otherwise make possible the violation of the balance conditions by a simultaneous modification by another transaction. During the modification of the parent page P in link(P,Q,R), P is kept X-locked and the direct child Q is kept U-locked. The right sibling R of Q need not be locked. Thus a simultaneous link(R,Q´,R´) or unlink(R,Q´,R´) may occur. However, the lock on Q prevents a simultaneous merge(Q,R) or redistribute(Q,R), which need both Q and R be X-locked. 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-locked and both the direct child pages Q and R are kept U-locked. The locking of R is needed to prevent a simultaneous split(R), which would result in a chain of two indirect child pages. When split(R) is called in update-mode traversal, its parent is no longer locked, so that it would be impossible to guarantee the safety of R if a simultaneous unlink(P,Q,R) were allowed (i.e., if unlink would not lock R).

The operation increase-tree-height(P,P´) keeps X-locked all the three pages (P, P´ and P´´) involved. The operation decrease-tree-height(P,Q) keeps X-locked both P and Q. The storage-map page is kept X-locked 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. Finally we note that at the end of each operation, both the modified pages and the generated redo-only log record are sent to the server. □

When a client 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* algorithm, in order to merge or redistribute Q with its sibling. The page Q can be either a rightmost or a non-rightmost 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 which is an indirect child of P, or (3) R is a direct child of P and has a right sibling page S which 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 direct child of P, or (5) the left sibling page of Q is an indirect child of P.

Repair-page-underflow(P,Q): Merge P's child page Q with its sibling page if possible. Otherwise, redistribute Q with its sibling page. Given are the Page-ids P and Q of a U-locked safe parent page and a U-locked direct child page that is about to underflow. At the end of the repair-page-underflow algorithm, the lock on the parent page P is released, the U lock on the child page (Q or its sibling) that covers the search key value is retained and the lock 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.
Step 1. If the high-key value in Q is equal to the high-key value in P, so that Q is the rightmost child of its parent page P, then go to Step 15.
Step 2. Request from the server a U lock on the right sibling page R of Q. When the lock is granted, U-lock R locally, and determine the high-key value v in R.
Step 3. If the high-key value u in Q is less than the key value v of the associated index record (v, Q) in the parent page P (Figure 4.13), then go to Step 6.

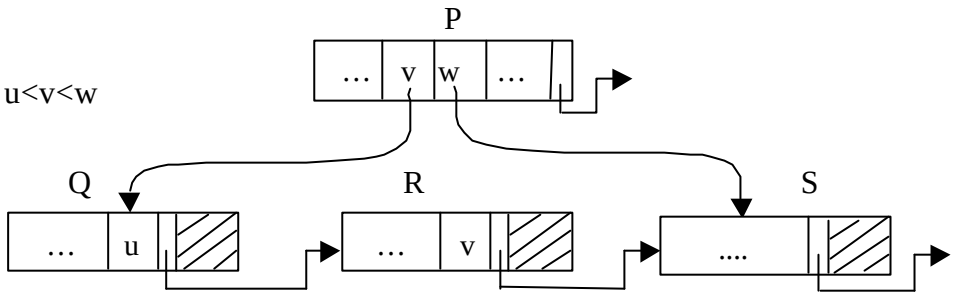

Figure 4.13. The right sibling page R of an about-to-underflow page Q is an indirect child of its parent P.

Step 4. If the high-key value v in R is less than the key value of the associated index record (w, R) in the parent page P (Figure 4.14), then go to Step 9.

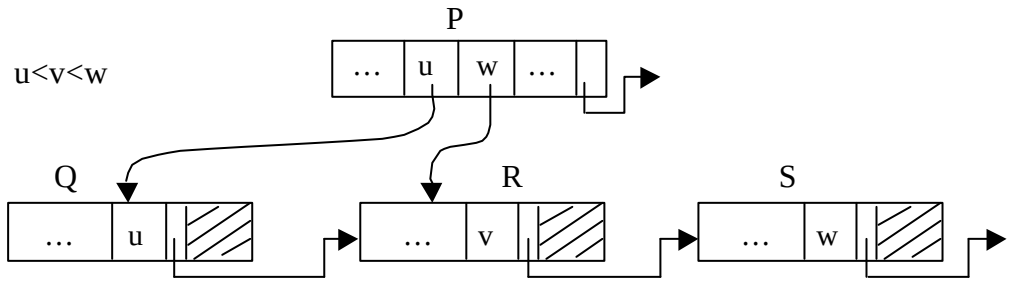

Figure 4.14. The right sibling page R of an about-to-underflow page Q has a right sibling S, which is an indirect child of its parent P.

Step 5. Now the high-key value v in R is equal to the key value of the associated index record (v, R) in the parent page P (Figure 4.15); go to Step 12.

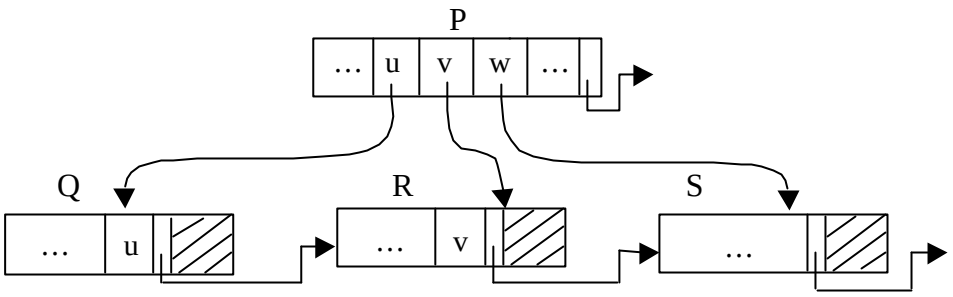

Figure 4.15. The right sibling page R of an about-to-underflow page Q has a right sibling S, which is a direct child of its parent P.

Step 6. This is Case 1 where R is an indirect child of P (Figure 4.13). Request the server to release the U lock on P. Release the local U lock on P when the request is granted.
Step 7. If Q and R can be merged then merge(Q,R) else redistribute(Q,R).
Step 8. Return.
Step 9. This is Case 2 where R is a direct child of P and R has a right sibling page S, which is an indirect child of P (Figure 4.14). If P cannot accommodate the index record (w, S), then split(P). Link(P,R,S) and unlink(P,Q,R), see Figure 4.16.
Step 10. If Q and R can be merged then merge(Q,R) else redistribute(Q,R).
Step 11. Return.
Step 12. This is Case 3 where R is a direct child of P and R has a right sibling page S which is also a direct child of P (see Figure 4.15). Unlink(P,Q,R), see Figure 4.13.
Step 13. If Q and R can be merged then merge(Q,R) else redistribute(Q,R).
Step 14. Return.

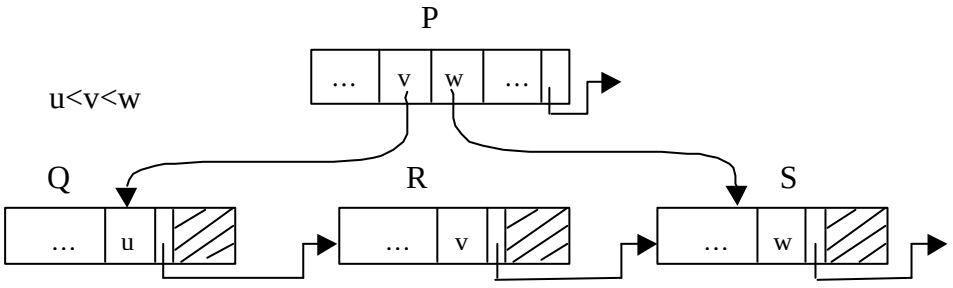

Figure 4.16. Page S has been linked to its parent P and page R has been unlinked.

Step 15. In this case Q is the rightmost child of its parent P (Figure 4.17 or Figure 4.19). Let (v, L) be the index record in P immediately preceding the index record (w, Q) in P. Request the server to release the U lock on Q (to avoid a deadlock) and to get U locks on the page L (direct child of P) and its right sibling page. When the

locks are granted, release the local U lock on Q, and U-lock L and its right sibling page locally.
Step 16. If the high-key value in page L is less than v, then go to Step 20.

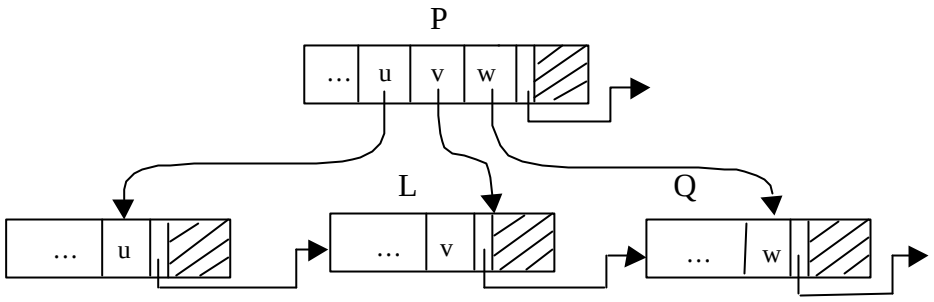

Figure 4.17. The about-to-underflow page Q is the rightmost child of its parent page P and has a left sibling page L, which is a direct child of parent P.

Step 17. This is Case 4 where the left sibling page L of Q is a direct child of P (Figure 4.17), because the high-key value v in L is equal to the key value of the associated index record (v, L) in the parent page P. If Q is no longer about to underflow, then go to Step 22. Otherwise, unlink(P,L,Q), see Figure 4.18.
Step 18. If L and Q can be merged then merge(L,Q) else redistribute(L,Q).
Step 19. Set Q:= L, and return.
Step 20. This is Case 5 where the right sibling page N of L is an indirect child of the parent P (Figure 4.19). If P cannot accommodate the index record (v, N), then split(P). Link(P,L,N), and request the server to release the U lock on L and to get a U lock on Q. When the request is granted, release the local U lock on L and U-lock Q locally. If Q is no longer about to underflow, then go to Step 22. Otherwise, unlink(P,N,Q).
Step 21. If N and Q can be merged then merge(N,Q) else redistribute(N,Q). Set Q:= N, and return.

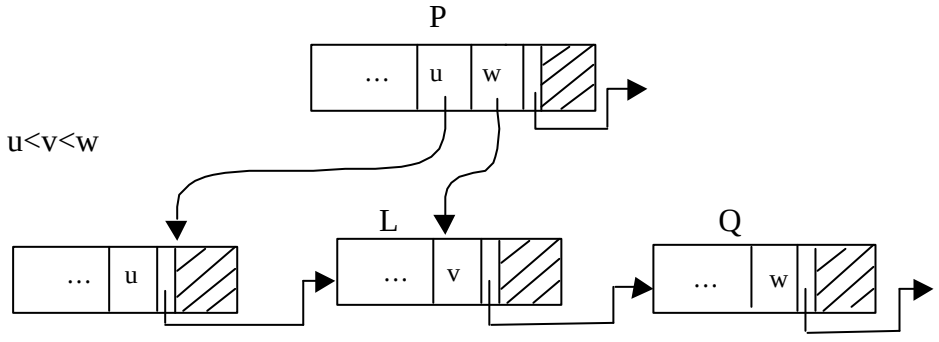

Figure 4.18. The about-to-underflow rightmost child Q of P is unlinked from its parent.

Step 22. Now Q is no longer about to underflow and therefore need not to be merged or redistributed. Request the server to release the U lock on the left sibling of Q (i.e., L when from Step 17 and N when from Step 20). When the request is granted, release the local U lock on the page. □

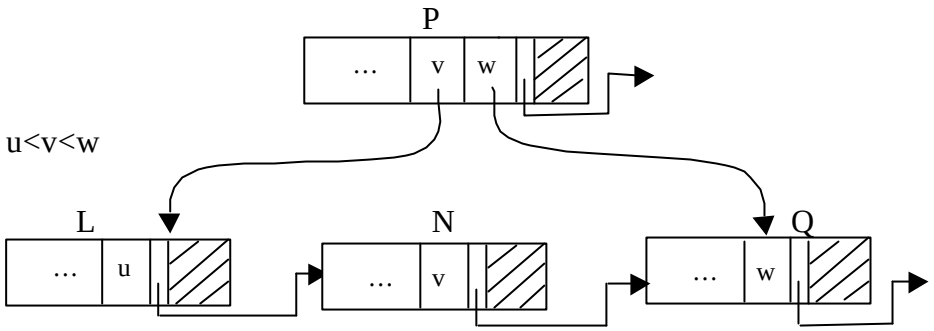

Figure 4.19. The about-to-underflow page Q is the rightmost child of its parent page P and has a left sibling page N, which is an indirect child of parent P.

Lemma 4.2. 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 k, Q is the next page in the search path of k and both P and Q are U-locked by client transaction T. Then the procedure repair-page-underflow(P,Q), when run on B on behalf of T in search for key k, generates a sequence of structure-modification operations logged for T such that, at the end of the procedure, the resulting B-link tree is structurally consistent and balanced, the page denoted by the parameter P covers k, and the page denoted by the parameter Q is a safe direct child of P and is the next page on search path for k.

Proof. The sequences of structure-modification operations generated by the repair-page-underflow(P,Q) procedure 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 procedure in isolation without any other transaction in progress.
(1) R is an indirect child of P (Figure 4.13): merge/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 which is an indirect child of P (Figure 4.14): split(P) (if P is full); link(P,R,S); unlink(P,Q,R); merge/redistribute(Q,R). As P is assumed to be safe, it may be split into pages P and P´, if it has not enough room for the child link (w, S). Split(P) returns the one of the 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 operation unlink(P,Q,R). The case thus reduces to Case 1.

(3) R is a direct child of P and has a right sibling page S which is also a direct child of P (Figure 4.15): unlink(P,Q,R); merge/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 (Figure 4.17): unlink(P,L,Q); merge/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 procedure, the parameter Q is set to L, which is on the search path, as required. However, as Q is unlocked and locked again after locking L, it may happen that Q is no longer about to underflow. In that case the procedure 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 (Figure 4.19): split(P) (if P is full); link(P,L,N); unlink(P,N,Q); merge/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 unlocked and locked again after locking N, it may happen that Q is no longer about to underflow. In that case the procedure does not perform the unlink(P,N,Q) and merge/redistribute(N,Q) operations, and the parameter Q will stay as it is.

Next we note that the above reasoning also hold in the context of concurrent operations by other client transactions. This follows from the fact that the pages involved are all kept U-locked over the entire sequence of structure modifications. Case 1 is clear by Lemma 4.1 since it consists of only a single structure modification. In Case 2, the pages P, Q and R are locked. Note that if P is split, then the split(P) call leaves the page that covers k (P or its right sibling) U-locked. In Case 3, the pages P, Q and R are locked. In Case 4, the pages P, L and Q are locked when performing unlink(P,L,Q) followed by merge/redistribute(L,Q). In Case 5, the pages P, L and N are locked while performing the possible split(P) followed by link(P,L,N), and the pages P, N and Q are locked while performing unlink(P,N,Q) followed by merge/redistribute(N,Q).

To prevent a deadlock and to decrease the number of page locks held simultaneously by T, the U lock on Q held when the procedure repair-page-underflow is entered is released in Cases 4 and 5, and Q is relocked by T after first locking its sibling L or siblings L and N. Because page P, which contains the child link to Q, is kept locked all the time, no other transaction can simultaneously deallocate Q or unlink it from its parent. After relocking 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 procedure repair-page-underflow is only used in the update-mode traversal (as shown in Algorithm 4.2 below) 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 unlocked by T. □

4.3 B-link-Tree Distributed Traversals

A client transaction T takes as input a key value u and uses the *read-mode traversal* algorithm (Algorithm 4.1) to find the target leaf page P that covers the record with key value u. If during the traversal of the B-link tree T runs into a page which does not cover the search key value u, then T follows the sideways link to the next page on the same level. When the leaf page P covering the search key value u is reached, then P is S-locked locally.

Algorithm 4.1: Read-mode traversal using lock coupling with S locks. Given a key value $u < \infty$, the algorithm searches for u and returns the Page-id P of an S-locked leaf page that covers u.
Step 1. Set P:= Page-id of the root page of the B-link tree. If P is cached locally, then S-lock P. Otherwise, request from the server a copy of P, and S-lock P locally when the request is granted.
Step 2. Determine the high-key value k´ in P.
Step 3. If u > k´ then if the right sibling page P´ of P is cached locally, then S-lock P´ locally else request from the server a copy of P´ and S-lock P´ locally when the request is granted. Unlock P locally, and set P:= P´.
Step 4. If P is a leaf page, then return.
Step 5. Search P for the child page Q covering the search key value u. If Q is cached locally, then S-lock Q locally. Otherwise, request from the server a copy of Q and S-lock Q locally when the request is granted.
Step 6. Unlock P locally, set P:= Q, and go to Step 2. □

When a client transaction T wants to insert or delete a database record with key value k, T takes as input the key value k and uses the *update-mode traversal* (Algorithm 4.2) to find the target leaf page that covers k.

Algorithm 4.2: Update-mode traversal, using lock coupling with U locks. Given a key value k, the algorithm searches for k by traversing the B-link tree and returns the Page-id of the U-locked leaf page that covers k. All unsafe pages on the search path are turned into safe ones.
Step 1. Set P:= Page-id of the root page of the B-link tree. If P is not U-locked locally for T, then request the server to get a U lock on P, and U-lock P locally when the lock is granted.
Step 2. If the Page-link of P = nil, then go to Step 4.

Step 3. Now the root page P has a right sibling page P´ and hence the tree height is increased. Request the server to get a U lock on P´. When the lock is granted, U-lock P´ locally and increase-tree-height(P,P´).
Step 4. If P is a leaf page, then return.
Step 5. Search P for the child page Q covering the search key value k, and request the server to get a U lock on Q. When the lock is granted, U-lock Q locally.
Step 6. If Q is the only child of the root page P so that Q has no right sibling, then decrease-tree-height(P,Q), and go to Step 4.
Step 7. If Q is about to underflow, then execute repair-page-underflow(P,Q), set P := Q, and go to Step 4.
Step 8. If the high-key value u in Q is less than the key-value of the associated index record (v, Q) in the parent P (i.e., Q has a right sibling page R which is an indirect child of parent P), then first split(P) if P cannot accommodate the index record (v, R), and then link(P,Q,R).
Step 9. If Q covers the search key value k, then request the server to release the U lock on P; when the request is granted, release the local U lock on P, and set P:= Q and go to Step 4. Otherwise, determine the right sibling page R of Q, and request the server to release the U locks on P, Q and to get a U lock on R. When the request is granted, release the local U locks on P and Q, U-lock R locally, set P := R, and go to Step 4. □

Lemma 4.3. Let B be a structurally consistent and balanced B-link tree. Assume that a client transaction T performs an update-mode traversal, using Algorithm 4.2 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 4.1 and 4.2 by induction on the height of B, because Algorithm 4.2, when traversing from one page to the next on the search path, always turns an unsafe child page to 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 structure modification increase-tree-height(P,P´) is performed if the root page has a right sibling, or the structure modification decrease-tree-height(P,Q) 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 procedure repair-page-underflow(P,Q) 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, then the structure modifications split(P) (if needed) and link(P,Q,R) are called. At each step, we may assume as an induction hypothesis that the page P on the previous level on the search path is safe, thus satisfying a required precondition of the structure modifications. □

Lemma 4.4. Let B be a structurally consistent and balanced B-link tree of height h. Any read-mode traversal on B by Algorithm 4.1 on behalf of client transaction T accesses at most 2h pages of B and keeps at most two of those pages S-locked for T at a time. Any update-mode traversal on B by Algorithm 4.2 on behalf of T accesses at most 4h pages of B and keeps at most two of those pages X-locked and at most two U-locked for T at a time. In addition, the storage-map page may be accessed and X-locked 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 lock-coupling protocol that uses S locks. The worst case of 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) procedure when two sibling pages (L and N) of the child page Q have to be accessed. This makes three pages accessed. The fourth page is accessed when Q, the next page in the search path, may have to be split because a child link has to be inserted there. This indeed can happen because the underflow of Q can be repaired by merging a sibling to it, which may result in a full page. Actually the number of page accesses (lockings) is five in the worst case because when Q is the rightmost child of P its lock is temporarily released and the page is locked and accessed again when coming from the left sibling of Q. In the increase-tree-height operation the number of pages accessed in the two topmost levels of the resulting tree is three. The storage-map page is accessed when a new page needs to be allocated in the operations split(Q) and increase-tree-height(P,P´) and when a page needs to be deallocated in the operations merge(P,R) and decrease-tree-height(P,Q).

The number of U and X locks held simultaneously in each of the structure modifications is evident from the algorithms (also see the proof of Lemma 4.2). The link(P,Q,R) operation X-locks at most two pages while keeping one page U-locked simultaneously. The worst case occurs when the parent page P has to be split, which needs two X-locks. In addition, the child page Q is kept U-locked. The unlink(P,Q,R) operation X-locks the parent page P while keeping both the child pages Q and R U-locked simultaneously. The split(Q) operation X-locks Q and the new right sibling page created for Q. The merge(Q,R) and redistribute(Q,R) X-lock the sibling pages Q and R. The increase-tree-height(P,P´) operation X-locks the old root page P and the new root page P´´ and keeps U-locked the right sibling page P´.

The decrease-tree-height(P,Q) operation X-locks the root page P and its child page Q. The repair-page-underflow(P,Q) procedure at any phase of its execution keeps at most two pages X-locked and two pages U-locked simultaneously. The worst case occurs in Case 5 when the parent page P is split. In fact we could easily do with fewer locks here if we released the locks on the child pages for the duration of split(P), and after that is done, reacquired the locks. However this would need additional message exchanges with the server. □

Lemma 4.5. Read-mode traversals and update-mode traversals by Algorithms 4.1 and 4.2 are deadlock-free.

Proof. A client transaction performing a read-mode traversal by Algorithm 4.1 acquires S locks on the search path in a top-down, left-to-right order, so that a non-root page is locked only after locking either the parent or left sibling page first. A client transaction performing an update-mode traversal by Algorithm 4.2 acquires U locks 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) procedure it is necessary to release the U lock on Q when Q is the rightmost child of its parent P and the left sibling of Q must be locked (see Step 15). As only U locks are ever upgraded to X locks, we may thus conclude that our page-locking protocol is deadlock-free. □

4.4 Transactional Isolation by Leaf-Page Locks

We shall use the structure-modification operations of Section 4.2 and the traversal algorithms of Section 4.3 both in a context in which transactional isolation is achieved by commit-duration leaf-page locks (see Sections 4.5 and 4.6) and in a context in which transactional isolation is achieved by a record-level locking protocol called key-range locking (see Chapter 5). Analogous to Section 3.8, in order to adapt the structure-modification operations to the page-level-locking context, the following changes are needed in the above structure modification operations:

Split(Q): If Q is a leaf page S-locked or X-locked by T for commit duration, then both Q and the newly allocated page Q´ must be locked by T in the same mode for commit duration. This is because a page-level lock does not imply the record(s) updated by T in Q or Q´.

Merge(Q, R): If Q and R are leaf pages and one of them is S-locked or X-locked by T for commit duration then Q must be locked by T in the same mode for commit duration.

Redistribute(Q,R): If Q and R are leaf pages and one of them is S-locked or X-locked by T for commit duration then that lock must be retained for commit duration. Moreover, if the redistribution involves moving records from Q to R (resp. from R to Q) and if Q (resp. R) is locked for commit duration then also R (resp. Q) must be locked by T in the same mode for commit duration.

Increase-tree-height(P,P´): If P and P´ are leaf pages and one or both are S-locked or X-locked by T for commit duration, then the lock on P´ (if any) is retained for commit duration and the lock on P (if any) is changed to a commit-duration lock on P´´.

Decrease-tree-height(P,Q): if Q is a leaf page S-locked or X-locked by T for commit duration, then the lock on Q is changed to a commit-duration lock on P.

In the traversal algorithms of Section 4.3, the action of releasing a page-level lock should never imply releasing a commit-duration lock.

4.5 B-link-Tree Distributed Fetch, Insert and Delete

To perform Fetch[k,θu, x], a client transaction T takes u as an input key value and traverses the B-link tree using the lock-coupling protocol with S locks (Algorithm 4.1). When the leaf page P covering u is reached, T performs the same steps as in Algorithm 3.3 to find the record (k, x) in P or in P´, the page next to P where k is the least key satisfying kθu. The page that holds (k, x) is kept S-locked for commit duration.

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

Algorithm 4.3. Inserts a new record r with key value k into the database.
Step 1. Traverse the B-link tree using lock coupling with U locks (Algorithm 4.2).
Step 2. Search the U-locked leaf page P for the position of insertion of record r.
Step 3. If a key value that matches the key value k is found, then return the "uniqueness violation" status, terminate the insert operation, and return.
Step 4. If P is full, then split(P), and go to Step 6.
Step 5. Request the server to upgrade the U lock on P to an X lock. When the request is granted, upgrade the local U lock on P to an X lock.
Step 6. Insert record r into P, generate the redo-undo log record <T, insert, P, r, n> where n is the LSN of the previous log record generated by T, update the Page-LSN of P, and hold the X lock on P for commit duration. □

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

Algorithm 4.4. Deletes a record r with key value k from the database.
Step 1. Traverse the B-link tree using lock coupling with U locks (Algorithm 4.2).
Step 2. Search the U-locked leaf page P for the record r to be deleted.
Step 3. If r is not found, then return the "not found" status, terminate the delete operation, and return.
Step 4. Delete r from P, generate the redo-undo log record <T, delete, P, r, n> where n is the LSN of the previous log record generated by T, update the Page-LSN of page P, and hold the X lock on P for commit duration. □

Example 4.1. Let T1 and T2 be two client transactions where T1 wants to delete a database record with key value 35 while T2 wants to insert a record with key value 40. Assume that T1 starts first and U-locks the root page P of the initial B-link tree shown in Figure 4.20, and traverses the tree to reach the leaf page P4.

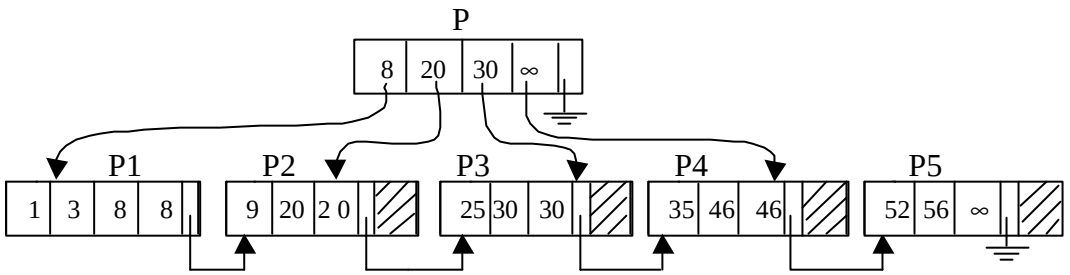

Figure 4.20. An initial B-link-tree, where leaf page P5 is an indirect child of its parent P.

Now the the high-key value in P4 is less than the key value of the associated record (∞, P4) in the parent page P. Hence, T1 executes link(P,P4,P5) and deletes the database record with key value 35 from P4. For execution of link operation, the root page P must first be split into P and P´, and thus the B-link-tree structure is changed to that shown in Figure 4.21. T1 keeps the X lock on P4 for commit duration.

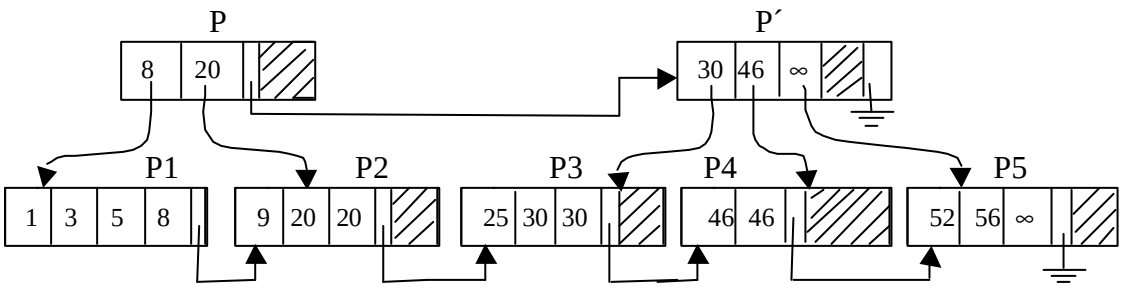

Figure 4.21. The linking of the indirect child page P5 to its parent caused the split of the root page P into P and P´.

When T2 starts and U-locks the root page P of the B-link tree of Figure 4.21, it finds that the high-key value in P is less than the search key value 40. Hence, T2 executes the increase-tree-height(P,P´) operation, and traverses down to P4. The request by T2 for a U lock on P4 is blocked at the server. When T1 has terminated and released its X lock on P4, T2 is granted the U lock, and its upgrade to an X lock. T2 inserts the new database record with key value 40 into P4. This results in the B-link-tree of Figure 4.22.

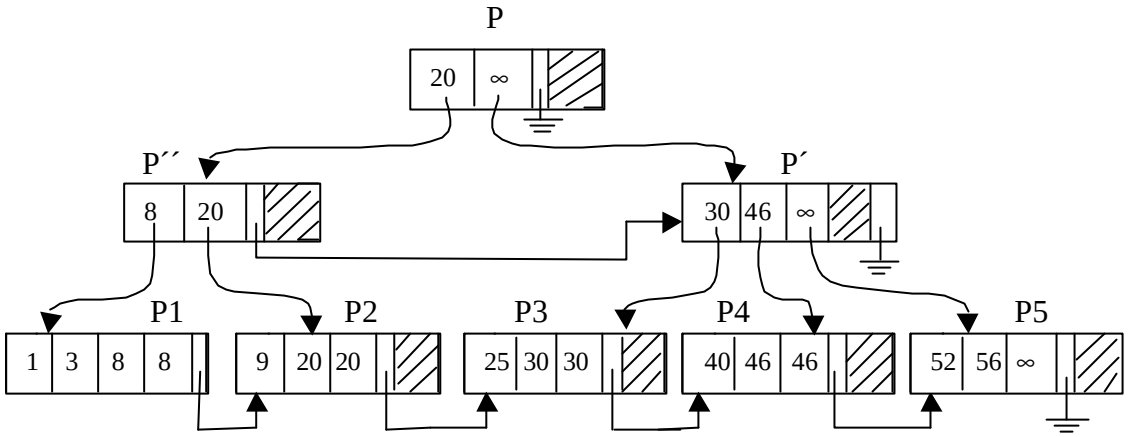

Figure 4.22. The height of the B-link tree is increased by one.

4.6 Undo of an Insertion or a Deletion of a Record

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

Algorithm 4.5. Undo the deletion of record r = (k, x). Given is a redo-undo log record <T, delete, P, r, n> where n is the LSN of the previous log record generated by T in the forward-rolling phase. The algorithm reinserts r into P if P still covers k. Otherwise, T retraverses the B-link tree from the root page to reach the covering leaf page Q and inserts r there.
Step 1. Check if physiological undo is possible, that is, if P still covers k and there is a room for r in P. Note that T still holds a commit-duration X lock on the page that covers k. If physiological undo is possible, then go to Step 4.
Step 2. Undo the deletion of r logically. This step is needed only when T itself has performed a structure modification that has changed the key range covered by P or when T has consumed the space occupied by r by inserting other records to P. Retraverse the B-link from the root page, using Algorithm 4.2 with k as an input key value to reach the covering leaf page Q; note that this includes the execution of any structure modifications that may be needed during the traversal. Set P:= Q (the reached covering leaf page).

Step 3. If P is full, then split(P).
Step 4. Insert r into P, generate the compensation log record <T, undo-delete, P, r, n>, and update the Page-LSN of P. □

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

Algorithm 4.6. Undo the insertion of record r = (k, x). Given is a redo-undo log record <T, insert, P, r, n> where n is the LSN of the previous log record generated by T in the forward-rolling phase. The algorithm deletes r from P if P still holds r. Otherwise, T retraverses the B-link tree from the root page to reach the covering leaf page Q and deletes r from there.
Step 1. Check if physiological undo is possible, that is, if P still contains r and will not underflow if r is deleted. Note that T still holds a commit-duration X lock on the page that holds r. If physiological undo is possible, then go to Step 3.
Step 2. Undo the insertion of r logically. This step is needed only when T itself has performed a structure modification that has moved r from its original page to another or when T has made P about-to-underflow by deleting other records from P. Retraverse the B-link from the root page, using Algorithm 4.2 with k as an input key value to reach the covering leaf page Q; again note that this includes the execution of any structure modifications that may be needed during the traversal. Set P:= Q (the reached covering leaf page).
Step 3. Delete r from P, generate the compensation log record <T, undo-insert, P, r, n>, and update the Page-LSN of P. □

4.7 Transaction Abort and Rollback in PS-PP

The B-link-tree transaction rollback in a PS-PP system during normal processing is almost identical to the B+tree transaction rollback in a PS-PP system (Algorithm 3.10), except that Step 4 in Algorithm 3.10 should be rewritten as follows. If the log record is a redo-only log record of type “link” or “unlink” or “split” or “merge” or “redistribute” or “increase-tree-height” or “decrease-tree-height”, then use the Prev-LSN of this log record to determine the next log record to be undone and its type, and go to Step 3. Moreover, in Step 5, Algorithms 4.5 and 4.6 are used to undo the logged updates.

4.8 Transaction Execution in PS-PP

The operations begin, commit, abort and rollback-complete of a client transaction T are executed as shown in Chapter 3. The commit and rollback-completed operations include the sending to the server of all the log records generated by T and the

associated updated (leaf) pages that have not yet been sent, and releasing of all the locks held by T.

The operations fetch, insert and delete in the forward-rolling phase of a client transaction T are executed by the algorithms of Section 4.5 and the undo-delete and undo-insert operations in the backward-rolling phase of an aborted transaction T are executed by the algorithms of Section 4.6.

Lemma 4.6. Let B be a structurally consistent and balanced B-link tree of height h and let T be a client transaction. Any operation Fetch[k,θu, x] by T on B accesses at most 2h+1 pages of B and keeps at most two of those pages S-locked for T at a time. Any operation Insert[k, x] or Delete[k, x] in the forward-rolling phase of T and any logically implemented inverse operation Undo-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-locked and at most two U-locked for T at a time, and produces a structurally consistent and balanced B-link tree. Any physiologically implemented 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,θu, x] follows immediately from Lemma 4.4 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 kθu. The claims for Insert[k, x], Delete[k, x], Undo-insert[k, x] and Undo-delete[k, x] follow from Lemma 4.4 and Algorithms 4.3, 4.4, 4.5 and 4.6. By Lemma 4.3, the leaf page reached by an update-mode traversal is safe. Thus it can be split, if needed, in Insert[k, x] and in a logical Undo-delete[k, x], and it will not underflow in Undo-delete[k, x] or in a logical Undo-insert[k, x]. A physiological Undo-insert[k, x] always checks that the page will not underflow from the deletion of (k, x), and a physiological Undo-delete[k, x] always checks that the page has room for (k, x). □

Theorem 4.7. If the database operations Fetch, Insert, Delete, Undo-delete and Undo-insert are implemented by the algorithms of Sections 4.5 and 4.6 and if each client transaction accesses database records in an ascending key-value order, then our B-link-tree algorithms for a PS-PP page-server system are deadlock-free.

Proof. The result follows from the assumption and Lemma 4.5. □

Theorem 4.8. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back client transactions that can be run on database D. Further let B be a structurally consistent and balanced B-link tree in a PS-PP system with db(B) = D, and let H´ be an implementation of H on B using the algorithms presented in this chapter. Then H´ produces a structurally consistent and balanced B-link tree. The effects of all record inserts and deletes on B by the committed and rolled-back transactions in H, together with their log records, as well as the effects of all structure modifications on B by all the transactions in H, together with their log records, are found at the server.

Proof. As the proof of Theorem 3.7, a formal proof would use induction on the number of operations in H´. For the purposes of the proof we may regard each of the structure-modification operations 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) as an atomic operation. Also the insertion of a record (k, x) into a leaf page P, and the deletion of a record from leaf page P are atomic operations. This is justified because the client transaction doing one of these operations will hold X locks on all the pages modified by the operation. The structural consistency and balance of the tree produced by H´ follows from Lemmas 4.1, 4.2, 4.3 and 4.6. As stated in Lemma 4.1, the effects of any structure-modification operation is propagated immediately to the server when the operation is completed. The propagation of record inserts and deletes however is not immediate and may be delayed until the transaction commits or completes its rollback. □

Theorem 4.9. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back client transactions that can be run on database D and let Hg be a completed history for H. Further let B be a structurally consistent and balanced B-link tree in a PS-PP system with db(B) = D, and let H´ be an implementation of H on B using the algorithms presented in this chapter. Then there is a B-link-tree operation string g´ such that H´g´ is an implementation of Hg on B. Moreover, if each transaction in H contains at most one update operation (i.e., an insert or a delete of a record), then all the inverse operations in g are performed physiologically in g´.

Proof. The proof is similar to that of Theorem 3.8. □

4.9 Restart Recovery in PS-PP

The restart recovery consists of three passes: the analysis, the redo, and the undo passes. The analysis and the redo passes are identical to the ones in Chapter 3. The B-link tree will be structurally consistent by the end of the redo pass. The undo pass

is very similar to the one in Chapter 3, except that when a redo-only log record of type “link” or “unlink” (in addition to “split” or “merge” or “redistribute” or “increase-tree-height” or “decrease-tree-height”) is encountered, then no action is performed except that the value of the Prev-LSN of such log record is used to determine the next log record to be processed. Moreover, when a redo-undo log record of type “delete” or “insert” is encountered, then the logged update is undone using Algorithm 4.5 or 4.6, respectively, where no X-locks are acquired on the affected pages.

Analogous to Theorem 3.9, we have:

Theorem 4.10. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back client transactions that can be run on database D1 and let B1 be a structurally consistent and balanced B-link tree in the server of a PS-PP system with db(B1) = D1. Further let H´ be an implementation of H on B1 using the algorithms presented in this chapter and let L be the sequence of log records sent to the server 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 at the server, 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 g' of B-link-tree operations that implements some completion string for H1.

Proof. The proof is similar to that of Theorem 3.9. □

Chapter 5

B-link-Tree Concurrency Control and Recovery in a PS-AA System

In this chapter, we describe our new adaptive replica-management (callback locking) and adaptive concurrency-control (key-range locking) protocols for a PS-AA page-server system. We modify our B-link-tree algorithms that we have developed for a PS-PP page-server system, so that these algorithms work with our new adaptive concurrency-control and replica-management protocols. These algorithms are suitable for general-purpose database applications where concurrency is a major issue as in a general-purpose RDBMS. These algorithms allow a higher degree of concurrency while still guaranteeing repeatable-read isolation. As before, structure modifications can run concurrently with other structure modifications and leaf-page updating without any need for tree locks. A leaf page may contain uncommitted updates by several client transactions, and uncommitted updates by one transaction on a leaf page may migrate to another page in structure modifications by other transactions. A record r in a U-locked leaf page P can be updated by one client transaction while available records other than r in P can simultaneously be fetched by transactions at other clients. Unlike in the algorithms of Chapters 3 and 4, record inserts and deletes on a leaf page are immediately propagated to the server.

5.1 Key-Range Locking

In a page server in which concurrency control and replica management are performed at the record level, client transactions rely on record locking. However, S-locking only the found record locally by a fetching transaction and X-locking only the inserted (or deleted) record by an updating transaction 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* [Moha90, Moha92b, Gray93, Moha96].

The idea of key-range locking is that the range of possible key values is divided into a set of *key ranges*, and to lock a key range we lock the last key value in the range, i.e., we apply *next-key locking*. A lock on a key value is really a range lock on the key values from the preceding key value up to the locked key value. For example, the set of key values v_1, v_2, v_3, v_4 is divided into the set of key-ranges, $(v_1, v_2]$, $(v_2, v_3]$, $(v_3, v_4]$. To lock the key range $(v_1, v_2]$, we lock the key value v_2. Hence a lock on v_2 locks everything after v_1 up to v_2. An alternative to next-key locking is called *previous-key locking*, where a lock on a key value starting the range is used as a lock on the whole range. Then, a lock on the key value v_1 is used as a lock on the key-range $[v_1, v_2)$. When the key-range locking protocol is used, then a lock on the key value k of the record r serves as a key-range lock, and at the same time as a lock on the record r. In other words, a lock on a record is actually a key-range lock.

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

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

There is one troublesome point in using the key-range locking protocol. Namely, if a client 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 client transaction may have deleted r´ or inserted a new record just before r´. Therefore, if a client 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 from the server.

As in Chapters 3 and 4, lock tables at the server and clients may be implemented as hash tables, and hence we hash the key value to get a fixed-length lock name. That is, the record-lock name is formed as follows.

Record-lock-name := Hash(Key-value)

5.2 Adaptive Replica Management (Callback Locking)

To allow different client transactions to update leaf pages concurrently, we have to modify our replica-management (callback-locking) and concurrency-control protocols, so that these protocols can work in an adaptive manner, that is, either at the page level or at the record level, according to the current needs. However, the concurrency control and replica management for index pages remains to work at the page level.

In a PS-AA system, a client transaction T always acquires global X locks from the server on records to be updated and on their next records, while T acquires just local S locks on records to be fetched if these records are available. Only if the records to be fetched are currently unavailable, T requests the server to get global S locks on those records. A record r in a copy of page P cached at client A is *unavailable at* A *for* client transaction T running at A if r is not locked by T and r was X-locked by some other transaction at some other client B at the time the copy of P was received from the server and installed in the cache of A. A record r in a cached copy of P at A is *available at* A *for* T if it is not unavailable at A for T.

We use *record marking* to indicate which records in a cached page copy are unavailable. Record marking allows client transactions to acquire just local S locks on the records to be fetched when these records are available. If record marking were not used, then client transactions would always have to acquire global S locks on the records to be fetched.

To implement record marking, we reserve one bit in each record as an *unavailability bit*. The unavailability bit of record r in page P is set in all cached copies of P if and only if some transaction T has acquired an X lock on r. Thus, r is unavailable for all transactions other than T. When T no longer holds an X lock on r, the unavailability bit in the server's copy of P is reset.

The *adaptive replica-management protocol* works as follows. A client transaction can safely S-lock (locally) and read cached pages without server intervention. If a transaction T at client A wants to fetch an available record r in a locally S-locked leaf page P, then T just acquires a local S lock on the record. To fetch an unavailable record r in P, T releases its local S lock on P and requests the server to get a *global* S lock on r. When r is no longer X-locked at the server by another client transaction, the server grants a global S lock on r and sends a new copy of P to A, with records X-locked by other clients marked as unavailable.

To update a record in a cached leaf page P, or to perform a structure modification on a cached leaf or non-leaf page P, a U lock on P must first be acquired from the server. If a transaction T at client A in its forward-rolling phase wants to update (or to acquire just an X lock on) record r in a locally U-locked leaf page P, then T requests the server to get an X lock on r in P. If r is X-locked or S-locked at the server by another client, then T waits until the conflicting lock is released. When r is no longer X-locked or S-locked at the server, the server acquires an X lock on r for T and issues callback requests to all other clients (except A) that hold a cached copy of P. At a client B, such a callback request is treated as follows. If the callback request cannot be granted immediately (due to a local S lock held on r by an active transaction at B), then B responds to the server saying that the record r is currently in use. When r is no longer S-locked at B, it is marked as unavailable in P at B. P remains cached and S-locked locally at B if there are still some other records in P which are currently only locally S-locked at B. However, if no record in P remains S-locked at B and hence also P is no longer S-locked locally at B, then P is purged from B's cache. Once all the callbacks have been acknowledged to the server, the server informs client A that its request for an X lock on r has been granted. Subsequent fetch or update operations for r from other client transactions will then block at the server, until the X lock on r is released by transaction T at client A. When client A sends a request to the server to release the X locks held by transaction T running at A on some records in page P, the server releases the X locks of T on these records and marks these records as available in its current copy of P. However, the server does not have to inform other clients which are caching copies of page P about the availability of these records.

If a transaction T at client A needs to perform a structure modification on a locally U-locked leaf page P, then T requests the server to get global S locks on the records in P that are currently only locally S-locked at A (if any) and to upgrade the U lock on P to an X lock. The server acquires the global record locks and the X lock on P for T which are granted right away and then sends callback requests for P to all other clients (except A) which hold a cached copy of P. At client B, such a callback is treated as follows. If B is currently only holding local S locks on some records in

P, and hence also a local S lock on P, then B responds by requesting the server to get global S locks on those records. When the global record locks have been granted, and when P is no longer in use at B, B releases its local S lock on P, and purges P from its cache. If, on the contrary, B has no local S locks on any record in P, then P is just purged from B's cache. Once all the callbacks have been acknowledged to the server, the server informs client A that its request for global record locks and for an X lock on P has been granted.

For example, when a client transaction T at client A during its forward-rolling phase has inserted a record r into a U-locked leaf page P which is also cached at client B, then A sends a copy of P and the log record to the server with a request to release the U lock held by T on P and the X lock held by T on the next record r´. The server installs the new copy of P in its database buffer and the log record in the log buffer, marks r´ as available in P, releases the U lock on P and the X lock on r´, and holds the X lock on r for commit duration. Now if a transaction T´ at client B needs to fetch the record r´ which is still marked as unavailable in the locally S-locked cached copy of page P, then T´ releases its local S lock on P and requests the server to get a global S lock on r´ and a new copy of P. The server sends a new copy of P and a global S lock on r´ to B when r´ is no longer X-locked at the server.

Theorem 5.1. The adaptive replica-management protocol is starvation-free.

Proof. In the adaptive replica-management protocol, when a client transaction T at client A requests the server to get an X lock on a record r in page P, the server acquires an X lock on r when r is not S-locked or X-locked and sends callback requests for r to all clients (except A) that are caching copies of P. Therefore, the result follows from Theorem 3.1 and the algorithm. □

5.3 Adaptive Concurrency Control (Key-Range Locking)

When a transaction T wants to fetch a record r with the least key value k from the database such that $k \theta u$ where $u < \infty$ is a given key value and θ is a comparison operator "≥" or ">", T searches the locally S-locked leaf page P if the Page-id of such a page is also given as input (remembered from the previous operation by T, if any). Otherwise, T searches the B-link tree from the root using lock coupling with S locks (Algorithm 4.1). When the target leaf page P is reached, it is S-locked locally and searched for r. If r is found in P and is available, then T S-locks r locally and holds the local S lock on r for commit duration; the local S lock on P is held by T until P is called back or until T commits or completes its rollback (whichever happens first). As explained above, if P is called back, then T requests the server to get a global S lock on r which is granted right away. If r is found in P but

unavailable, then T releases the local S lock on P and requests the server to get global S locks on r and other records in P which are currently only locally S-locked at A (if any) and for a copy of P. When the locks are granted, T S-locks P and the records locally and searches P for r and proceeds as above.

When a client transaction T at client A wants to insert a record r with key value k into the database during its forward-rolling phase, T traverses the B-link tree using lock coupling with U locks (Algorithm 4.2). When the target leaf page P is reached, it is U-locked locally. Then T determines the page P´ that holds the record r´ next to r, and requests the server for a U lock on P´ and U-locks P´ locally if P´ ≠ P. Afterwards T requests the server to get X locks on r and r´. When the locks are granted, the records r and r´ are X-locked locally, and T searches P for the position of insertion of r. If inserting r would violate the index uniqueness, then T requests the server to release the acquired locks on P, r, P´ and r´ and releases the local locks on P, r, P´ and r´ when the request is granted, the insert operation is terminated, and the exception “uniqueness violation” is returned. Otherwise, T inserts r into P and marks r as unavailable. A copy of the updated page P and the generated log record are sent to the server with a request to release the U lock on P and the X lock on r´. Thus, we use immediate update propagation. When the request is granted, T releases the local U lock on P and the local X lock on r´, and holds the X lock on r for commit duration. The U lock on P is only held for short duration, to allow other client transactions to update other records in P or to modify the structure of P while T is still active.

When a client transaction T at client A wants to delete a record r with key value k from the database during its forward-rolling phase, T traverses the B-link tree using lock coupling with U locks (Algorithm 4.2) and U-locks the reached leaf page P locally. Then T determines the page P´ that holds the record r´ next to r, requests the server for a U lock on P´ if P´ ≠ P, and U-locks P´ locally when the lock is granted. Afterwards T requests the server to get X locks on r and r´, X-locks r and r´ locally when the locks are granted, and searches P for r. If r is not found in P, then T requests the server to release the acquired locks on P, r, P´ and r´ and releases the local locks on P, r, P´ and r´ when the request is granted, the delete operation is terminated, and the exception “record not found’’ is returned. Otherwise, T deletes r from P. A copy of the updated page P and the generated log record are sent to the server with a request to release the U lock on P and the X lock on r. When the request is granted, T releases the local U lock on P and the local X lock on r, and holds the X lock on r´ for commit duration. As in the insert operation, the U lock on P is held for short duration only, to allow other client transactions to update or modify P concurrently.

When a client transaction T is aborted, then during the backward-rolling phase, T acquires just a U lock on the page P covering the update to be undone and no additional record locks. That is, an insert or a delete operation is undone under the protection of the commit-duration X lock acquired for the insert or delete operation during the forward-rolling phase.

5.4 Concurrent Leaf-Page Updates and Deadlocks

When different client transactions are allowed to update a leaf page P concurrently, a deadlock may occur. In the following we distinguish between four cases in which such a deadlock can occur. For simplicity and without loss of generality we assume that the inserted or deleted record and the next record are both on the same leaf page P.

Case 1: A deadlock involving two forward-rolling updating transactions T1 and T2.

First, T1 at client A inserts a new record r into a U-locked leaf page P, marks r as unavailable, generates a redo-undo log record, updates the Page-LSN of P, and sends a copy of the updated page P and the log record to the server, with a request to release the U lock on P and the X lock on the next record. T1 holds the X lock on r for commit duration.

Then T2 at client B requests from the server a copy of P and a U lock on P, installs P into the cache of B and U-locks P locally when the request is granted, and searches P for the position of insertion of a new record r´´ and determines its next record, which now happens to be r, the record inserted by T1 and hence unavailable. Then T2 requests the server to get an X lock on r in P, while P is still U-locked locally at B.

The server will block T2's request for an X lock on the record r, because r is still X-locked by T1, and hence T2 has to wait. Now, if T1 wants to update P again, T1 requests from the server a copy of P and a U lock on P. The server will block T1's request, because P is still U-locked by T2, and hence T1 has to wait. Therefore, we have a deadlock involving two forward-rolling updating transactions T1 and T2.

Case 2: A deadlock involving a forward-rolling updating transaction T1 inserting a record r and a fetching transaction T2 looking for record r.

First, T1 at client A inserts a new record r into a U-locked leaf page P, marks r as unavailable, generates a redo-undo log record, updates the Page-LSN of P, and sends a copy of the updated page P and the log record to the server with a request to

release the U lock on P and the X lock on the next record. T1 holds the X lock on r for commit duration.

Then T2 at client B requests from the server a copy of page P that covers the key value of the record r to be fetched, installs P into the cache of B and S-locks P locally when the request is granted, and searches P for r. Now r is found in P but it is unavailable. Thus T2 requests the server to get a global S lock on the record r, while P is still S-locked locally at B.

The server will block T2's request for a global S lock on the record r, because r is still X-locked by T1, and hence T2 has to wait. Now, if T1 wants to update P again, then T1 requests from the server a copy of page P and a U lock on P. When the request is granted, T1 installs P into the cache of A and U-locks P locally, and searches P, only to find that P is full. Therefore, T1 requests the server to upgrade its U lock on P to an X lock. Hence, the server sends callbacks to all other clients that are caching copies of P. However, client B where T2 is running will block the callback request and informs the server that P is still in use, because P is still S-locked locally by T2 which is waiting for a global S lock on the record r. As a result the server blocks T1's request to upgrade its U lock on P, and hence T1 has to wait. Therefore, a deadlock involving a forward-rolling updating transaction T1 and a fetching transaction T2 will take place.

Case 3: A deadlock involving an aborted transaction T1 and a forward-rolling updating transaction T2.

First, T1 at client A in its forward-rolling phase inserts a new record r into a U-locked leaf page P, marks r as unavailable, generates a redo-undo log record, updates the Page-LSN of P, and sends a copy of the updated page P and the log record to the server with a request to release the U lock on P and the X lock on the next record. T1 holds the X lock on r for commit duration.

Then T2 request from the server a copy of P and a U lock on P. When the request is granted, T2 installs P into the cache of B and U-locks P locally, searches P for the position of insertion of a new record r´´, and determines the next record, which is r. Then T2 requests the server to get an X lock on r in P, while P is still U-locked locally at B.

The server will block T2's request for an X lock on the record r, because r is still X-locked by T1, and hence T2 has to wait. Now, if T1 wants to abort, then T1 requests the server to get a U lock on page P and a copy of P in order to undo the insertion of r. The server will block T1's request, because P is still U-locked by T2, and hence

T1 has to wait. Therefore, the backward-rolling transaction T1 gets involved in a deadlock with the forward-rolling updating transaction T2.

Case 4: A deadlock involving an aborted transaction T1 which in its forward-rolling phase deletes a record r and a fetching transaction T2 looking for record r´, the record next to r.

First, T1 at client A in its forward-rolling phase deletes a record r from a U-locked leaf page P, generates a redo-undo log record, updates the Page-LSN of P, and sends a copy of the updated page P and the log record to the server with a request to release the U lock on P and the X lock on r. T1 holds the X lock on the next record r´ for commit duration.

Then T2 at client B requests from the server a copy of page P that covers the key value of the record r to be fetched. When the request is granted, T2 installs P into the cache of B and S-locks P locally and searches P for r´, which is unavailable in P. Thus T2 requests the server to get a global S lock on r´, while P is still S-locked locally at B.

The server will block T2's request for an X lock on the record r´, because r´ is still X-locked by T1, and hence T2 has to wait. If T1 wants to abort, then to reinsert r into P, T1 requests the server to get a U lock on page P and a copy of P. When the request is granted, T1 installs P into the cache of A and U-locks P locally, and searches P only to find that P is full. Hence T1 requests the server to upgrade its U lock on P to an X lock. The server sends callbacks to all other clients that are caching copies of P. However, client B where T2 is running will block the callback request and inform the server that page P is still in use, because P is still S-locked locally at A by T1 which is waiting for a global S lock on r´. The server will block T1's request to upgrade its U locks on P, and hence T1 has to wait. Therefore, a deadlock involving the aborted transaction T1 and the fetching transaction T2 will take place.

To avoid a deadlock that may arise in the above situations, we prevent a client transaction from holding a U lock on a leaf page P while it is waiting for a record lock from the server. That is, the transaction must release its U lock on P whenever it has to wait for an X lock on a record in P. Similarly, we prevent a client transaction from holding a local S lock on a leaf page P while it is waiting for a record lock from the server. That is, the transaction must release its local S lock on P whenever it has to wait for a global S lock on a record in P. This is in accordance with the policy followed in a centralized database system, where a transaction is

allowed to wait for a lock only if the process generating the transaction is not holding any latches [Moha90, Moha92b, Moha96, Lome97].

5.5 B-link-Tree Distributed Fetch, Insert and Delete

The following algorithm implements the operation Fetch[k,θu, x] by client transaction T on a B-link tree.

Algorithm 5.1. Given a key value u < ∞, fetch the data record r = (k, x) with the least key value k satisfying kθu. As a result of a previous call to this algorithm, the Page-id P of the page where the previously fetched record resided may also be given as input.
Step 1. If P is not given as input, then go to Step 5. If P is cached locally, then S-lock P locally. Otherwise, request from the server a copy of P. When the request is granted, install the received copy of P in the local cache and S-lock P locally.
Step 2. Determine the lowest key value w, the highest key value v and the high-key value v´ in P.
Step 3. If u < w or u > v´, then go to Step 5.
Step 4. If u > v or u = v and θ = ">", then go to Step 6. Otherwise go to Step 7.
Step 5. Traverse the B-link tree from the root using lock coupling with S locks (Algorithm 4.1) with u as the input key value. Search the reached target leaf page P, determine the highest key value v and the high-key value v´ in P, set Q := nil, and go to Step 4.
Step 6. The record to be fetched resides in P´, the page next to P. If P´ is cached locally, then S-lock P´ locally. Otherwise, request from the server a copy of P´. When the request is granted, install the received copy of P´ in the local cache and S-lock P´ locally. In either case, set Q:=P and P:=P´, and save the Page-LSN of Q.
Step 7. Now P is the page that contains the record to be fetched. Determine the record r in P with the least key value k satisfying kθu. If r is available, then S-lock r locally, save the Page-id P, and return with r.
Step 8. The record r to be fetched is found to be unavailable in P. If r is the lowest record in P and Q ≠ nil, then go to Step 13.
Step 9. Save the Page-LSN of P and release the local S lock on P. Request the server to get global S locks on r and other records in P that are currently only locally S-locked and a copy of P. When the request is granted, install the received copy of P in the cache, S-lock P and the records locally, and search P for r.
Step 10. If the Page-LSN of P has not changed, then save the Page-id P and return with r.
Step 11. If P does not cover r or P is not a leaf page or P is not part of the index any more, then request the server to release the global S lock on r. When the request is granted, release the local S locks on P and r and go to Step 5.

Step 12. If r is present in P and the key value k of r is still the least key value in P satisfying kθu, then save the Page-id P and return with r. Otherwise, request the server to release the global S lock on r. When the request is granted, release the local S lock on r, determine the highest key value v in P, and go to Step 4.
Step 13. The record r to be fetched is the lowest record in P and unavailable, and r was found by entering P from its left sibling Q. Save the Page-LSN of P and release the local S lock on P. Request the server to get global S locks on r and on other records in P that are currently only locally S-locked and copies of P and Q. When the request is granted, install the received copies of Q and P in the local cache, S-lock Q, P and the records locally, and search P for r.
Step 14. If the Page-LSN of Q has not changed, then go to Step 10.
Step 15. If Q does not cover the key value u or Q is not a leaf page or Q is not part of the index any more, then request the server to release the global S lock on r. When the request is granted, release the local S locks on P and r, and go to Step 5.
Step 16. The Page-LSN of Q has changed but Q still covers u. Determine the highest key value v in Q. If v = u and θ = "≥", then set P := Q, save the Page-id P, and return with r.
Step 17. If v = u and θ = ">", then go to Step 10.
Step 18. Request the server to release the global S lock on r. When the request is granted, release the local S locks on P and r, set P:= Q, and go to Step 7. □

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

Algorithm 5.2. Insert a new data record r with key value k.
Step 1. Traverse the B-link tree using lock coupling with U locks (Algorithm 4.2).
Step 2. If the U-locked target leaf page P is full, then split(P).
Step 3. Determine the page P´ that holds the record r´ with the least key value greater than the key value of r. Thus P´ is P (when r´ is found in P) or the page next to P (otherwise). If P´ is not U-locked locally for T, then request the server to get a U lock on P´ and U-lock P´ locally when the request is granted.
Step 4. Lock-records(P,r,P´,r´). If the exception "records cannot be X-locked" is returned, then go to Step 1.
Step 5. Search P for the position of insertion of r. If a key value that matches the key value k of r is found, then terminate the insert operation, and return with the exception "uniqueness violation".
Step 6. Insert r into P, mark r as unavailable, generate the redo-undo log record <T, insert, P, r, n> where n is the LSN of the previous log record generated by T, and update the Page-LSN of P.
Step 7. Send a copy of the updated page P and the log record to the server with a request to release the U locks on P and P´, and to release the X lock on r´. When the

request is granted, release the local U locks on P and P´, release the local X lock on r´, and hold the X lock on r for commit duration. □

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

Algorithm 5.3. Delete a data record r with key value k.
Step 1. Traverse the B-link tree using lock coupling with U locks (Algorithm 4.2).
Step 2. Determine the page P´ that holds the record r´ with the least key value greater than the key value of r. Thus P´ is P (when r´ is found in P) or the page next to P (otherwise). If P´ is not U-locked locally for T, then request the server to get a U lock on P´ and U-lock P´ locally when the request is granted.
Step 3. Lock-records(P,r,P´,r´). If the exception "records cannot be X-locked" is returned, then go to Step 1.
Step 4. Search P for a record with key value k. If no record with key value k was found in P, then terminate the delete operation, and return with the exception "record not found".
Step 5. Delete r from P, generate the redo-undo log record <T, delete, P, r, n> where n is the LSN of the previous log record generated by T, and update the Page-LSN of P.
Step 6. Send a copy of the updated page P and the log record to the server with a request to release the U locks on P and P´, and to release the X lock on r. When the request is granted, release the local U locks on P and P´, release the local X lock on r, and hold the X lock on r´ for commit duration. □

Lock-records(P,r,P´,r´): Given are the Page-id P of a U-locked leaf page P covering record r and the Page-id P´ of a U-locked leaf page P´ containing the record r´ next to r. The algorithm acquires X locks on r and r´ if possible. Otherwise, it releases the U locks on P and P´ and returns with the exception "records cannot be X-locked".
Step 1. Request the server to get conditional X locks on the records r and r´ in P and P´.
Step 2. If the locks are granted right away, then X-lock r and r´ locally and return.
Step 3. Save the Page-LSNs of P and P´. Release the local U locks on P and P´. Request the server to release the U locks on P and P´, to acquire unconditional X locks on r and r´, to acquire U locks on P and P´, and to get a copy of P and P´. When the request is granted, U-lock P and P´ locally, and X-lock r and r´ locally.
Step 4. If the Page-LSNs of P and P´ have not changed, then return.
Step 5. If the Page-LSN of P has changed and either P does not cover r or P is not a leaf page or P is not part of the tree any more, then go to Step 12.
Step 6. Now P still covers r. Search P for a record r´´ with the least key value greater than the key value of r. If no such record is found in P, then go to Step 10.

Step 7. Now P contains r´´. If P = P´, then go to Step 8. Otherwise go to Step 9.
Step 8. If the key values of r´´ and r´ are equal, then return. Otherwise, request the server to release the X locks on r and r´ in P. When the request is granted, release the local X locks on r and r´, set r´:= r´´, and go to Step 1.
Step 9. Request the server to release the U lock on P´ and the X locks on r and r´ in P and P´. When the request is granted, release the local U lock on P´ and the local X locks on r and r´, set P´:= P and r´:= r´´, and go to Step 1.
Step 10. If P= P´, then go to Step 12. Otherwise, determine the page P´´ currently next to P.
Step 11. If P´´= P´ and the key value of r´ is equal to the key value of the first record in P´, then return.
Step 12. Request the server to release the U locks on P and P´ and the X locks on r and r´. When the request is granted, release the local U locks on P and P´ and the local X locks on r and r´, and return with the exception "records cannot be X-locked". □

Example 5.1 Assume that there are two clients, A and B, in our page-server system. Also assume that the cached B-link-tree pages at the server, client A and client B are as shown in Figure 5.1.

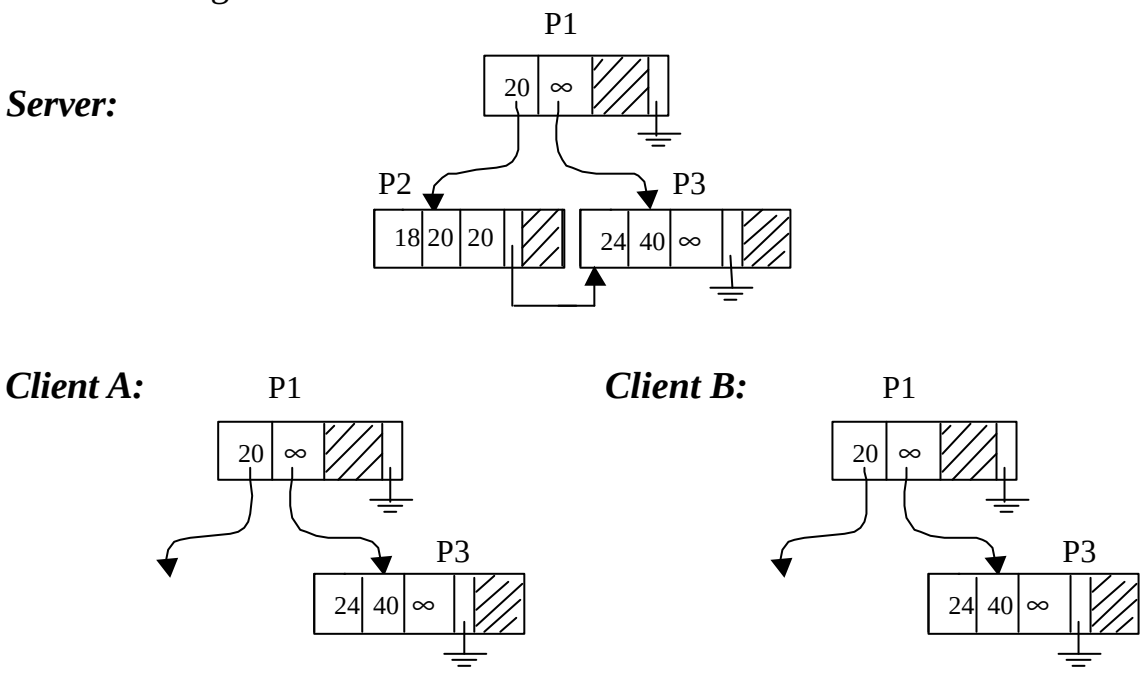

Figure 5.1. Cached B-link-tree pages at the server, client A and client B, where P3 is cached and U-locked at B and at the same time P3 is cached and S-locked locally at A.

1) Assume that transaction T1 at client A needs to fetch a record with the least key value k > 21. The record with key value k = 24 satisfies the fetch condition and it is available in P3, hence T1 acquires just a local S lock on this record.

2) Now if transaction T2 at client B needs to insert a new record with key value 38 into the cached and U-locked leaf page P3, then client B sends a request to the server for X locks on the new record and on the next record with key value 40. The server acquires X locks on the new record and on the next record and sends a callback request to client A for the record with key value 40 in its cached copy of P3. Client A responds by marking this record as unavailable in P3, because currently it is not in use, and A acknowledges the server. The server marks the next record as unavailable in its copy of page P3, and informs client B that its request for record locks has been granted. Client B X-locks the records locally and inserts the new record into P3, marks the inserted record as unavailable in P3, generates a redo-undo log record, and updates the Page-LSN of P3. Then client B sends copies of the updated page P3 and the log record to the server with a request to release the U lock on P3 and the X-lock on the next record. When the request is granted, client B releases the local U lock on P3 and the local X lock on the next record, and holds the X lock on the inserted record for commit duration. Now the cached B-link-tree pages at the server, client A and client B are as shown in Figure 5.2.

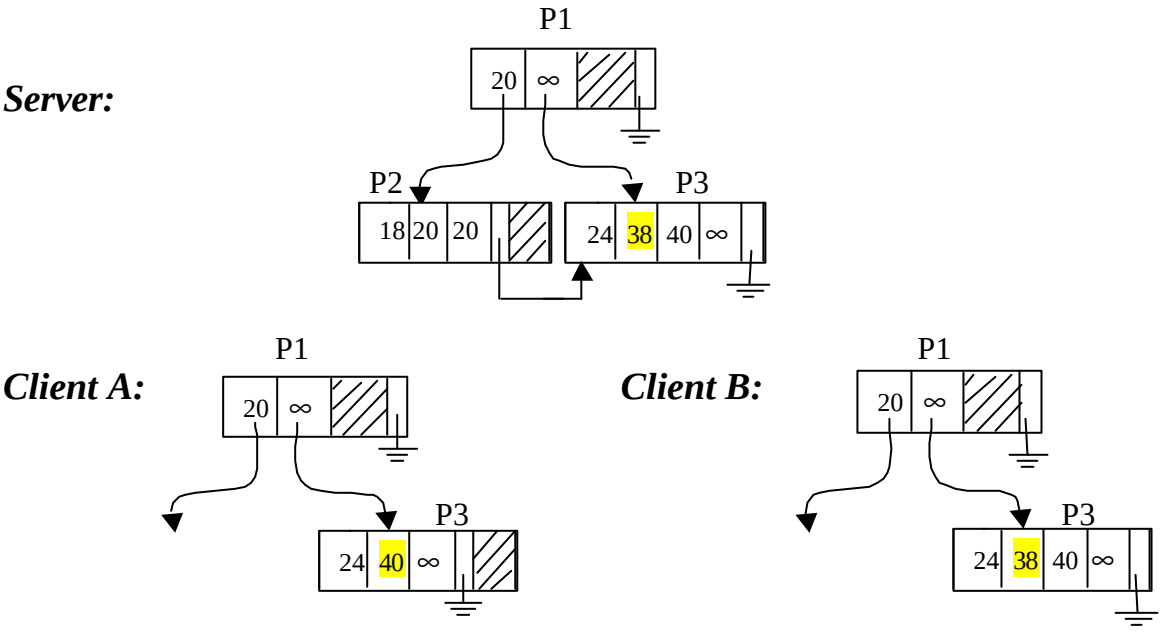

Figure 5.2. The server and client B are caching up-to-date copies of page P3, while client A is still caching an outdated copy of P3. The record with key value 40 is still marked as unavailable in A although it is no longer X-locked at the server by transaction T2 running at B.

3) Assume that transaction T1 at client A needs to fetch the record with the least key value k > 24 in the locally S-locked leaf page P3. When T1 searches P3, it finds a record with the least key value k = 40 such that k satisfies the search condition in P3, but it is unavailable. Therefore, client A releases its local S lock on P3 and sends a request to the server for global S locks on the record with key value 24 (because T1 has released its S lock on P3 and it has just a local S lock on this record) and on the record with key value 40, and for a new copy of P3. The server acquires global S locks on these records for T1, because none of these records is X-locked at the server. When the request is granted, client A installs the new copy of page P3 in its cache and S-locks P3 and the records with key values 24 and 40 locally (Figure 5.3), and searches P3 for a record with the least key value k such that k > 24. When T1 searches P3, it finds a record with the least key value k = 38 such that k satisfies the search condition k > 24, but it is unavailable. Therefore, client A releases the local S locks on P3 and on the record with key value k = 40 and sends a request to the server to get a global S lock on the record with key value k = 38, to release the global S lock on the record with key value k = 40, and to get a new copy of P3.

Client A:

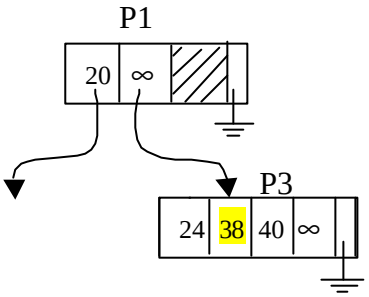

Figure 5.3. Client A caching up-to-date copy of page P3 where the record with key value k = 38 is still unavailable.

The server blocks A's request for a global S lock on the record with key value k = 38, because this record is still X-locked at the server by T2. When this record is no longer X-locked at the server by T2, the server grants a global S lock on this record to client A and T1 resumes the fetch operation. □

5.6 Undo of an Insertion or a Deletion of a Record

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

Algorithm 5.4. Undo the deletion of record r = (k, x). Given is a redo-undo log record <T, delete, P, r, n> where r is a record with key value k that was deleted by T from page P, and n is the LSN of the previous log record generated by T in its

forward-rolling phase. The algorithm reinserts r into P if P still covers k. Otherwise, T retraverses the B-link tree to reach the covering leaf page Q and inserts r into Q.
Step 1. If P is not currently U-locked locally for T, then request the server to get a U lock on P and a copy of P. When the request is granted, U-lock P locally.
Step 2. If P still covers k and there is room for r in P, then go to Step 5.
Step 3. Undo the deletion of record r logically. Retraverse the B-link tree from the root page, using Algorithm 4.2 with k as an input key value, and execute and log using redo-only log records any structure modifications that may be needed during tree traversal. Set P:= Q (the reached covering leaf page).
Step 4. If P is full, then split(P).
Step 5. Insert r into P, generate the compensation log record <T, undo-delete, P, r, n>, and update the Page-LSN of P.
Step 6. Send a copy of the updated page P and the CLR to the server, with a request to release the U lock on P. When the request is granted, release the local U lock on P. □

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

Algorithm 5.5. Undo the insertion of record r. Given is a redo-undo log record <T, insert, P, r, n> where r is a record with key value k that was inserted by T into page P, and n is the LSN of the previous log record generated by T in its forward-rolling phase. The algorithm deletes r from P if P still contains r. Otherwise, T retraverses the B-link tree to reach the covering leaf page Q and deletes r from Q.
Step 1. If P is not currently U-locked locally for T, then request the server to get a U lock on P and a copy of P. When the request is granted, U-lock P locally.
Step 2. If P still contains r and will not underflow if r is deleted, then go to Step 4.
Step 3. Undo the insertion of record r logically. Retraverse the B-link tree from the root page, using Algorithm 4.2 with k as an input key value, and execute and log using redo-only log records any structure modifications that may be needed during tree traversal. Set P:= Q (the reached covering leaf page).
Step 4. Delete r from P, generate the compensation log record <T, undo-insert, P, r, n>, and update the Page-LSN of P.
Step 5. Send a copy of the updated page P and the CLR to the server, with a request to release the U lock on P. When the request is granted, release the local U lock on P. □

5.7 Transaction Abort and Rollback in PS-AA

The log records are undone in the reverse chronological order, and for each log record that is undone, a CLR is generated. When a redo-only log record is

encountered, the Prev-LSN of such log record is used to determine the next log record to be undone. If any structure modifications are executed during transaction rollback, then such structure modifications are logged using redo-only log records. Moreover, no B-link tree structure modifications are undone during transaction rollback. The backward-rolling phase $Aa^{-1}R$ of an aborted transaction T = $BaAa^{-1}R$ is executed by the following algorithm.

Algorithm 5.6. Rollback an aborted transaction T.
Step 1. Generate the log record <T, abort, n> where n is the LSN of the last log record generated by T during its forward-rolling phase.
Step 2. Get the log record generated by T with LSN equal to n.
Step 3. If the log record is <T, begin>, then go to Step 7.
Step 4. If the log record is a redo-only log record of type "link" or "unlink" or "split" or "merge" or "redistribute" or "increase-tree-height" or "decrease-tree-height", then use the Prev-LSN of this log record to determine the next log record to be undone, and go to Step 3.
Step 5. If the log record is a redo-undo log record of type "delete" or "insert", then undo the logged update, using Algorithm 5.4 or 5.5, respectively.
Step 6. Get the next log record to be undone using the Prev-LSN of the log record being undone, and go to Step 3.
Step 7. Generate the log record <T, rollback-completed> and send it to the server with a request to release all locks of T.
Step 8. When the server receives the log record <T, rollback-completed>, the server flushes all the log records up to and including this log record, releases the locks of T, and informs the client.
Step 9. When the client is acknowledged about the successful flushing of the log records of T, the client releases the local locks of T, and discards the log records of T. □

When the client A where transaction T is running fails, then any leaf-page updates (record insert or delete) of T that have already been sent to the server are undone by the server.

5.8 Transaction Execution in PS-AA

The operations begin, commit, abort and rollback-complete of a client transaction T are executed as shown in Chapter 3. When the operation commit or rollback-complete is executed, T generates the log record <T, commit> or <T, rollback-completed> and sends it to the server with a request to release all its locks.

In the forward-rolling phase of a client transaction T, the operations fetch, insert and delete are executed by the algorithms of Section 5.5, while in the backward-rolling phase of an aborted transaction T, the operations undo-delete and undo-insert are executed by the algorithms of Section 5.6.

Theorem 5.2. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back client transactions that can be run on database D. Further let B be a structurally consistent and balanced B-link tree in a PS-AA system with db(B) = D, and let H´ be an implementation of H on B using the algorithms presented in this chapter. Then H´ produces a structurally consistent and balanced B-link tree. The effects of all record inserts and deletes on B by all transactions in H, together their log records, as well as the effects of all structure modifications on B by all the transactions in H, together with their log records, are found at the server.

Proof. The proof is similar to that of Theorem 4.8. Immediate update propagation guarantees that the effects of inserts and deletes by active transactions are also found at the server. □

Theorem 5.3. Let H be a history of forward-rolling, committed, backward-rolling and rolled-back client 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 in a PS-AA system with db(B) = D. Then there exists a B-link-tree operation string H´ that implements H on B using the algorithms presented in this chapter. Moreover, each operation implementation in H´ includes at most one traversal of B.

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

Theorem 5.4. Assume that each client transaction in its forward-rolling phase accesses records in ascending key order and that the operations Fetch, Insert, Delete, Undo-delete and Undo-insert are implemented by Algorithms 5.1, 5.2, 5.3, 5.4 and 5.5. Then no deadlocks can occur.

Proof. The operations use the same read-mode and update-mode traversal algorithms as the operations in Chapter 4. Thus we conclude by Lemma 4.5 that no deadlock can occur between page locks. No deadlocks can occur between record locks either, because the execution of single Fetch, Insert, Delete, Undo-delete and Undo-insert operations by Algorithms 5.1, 5.2, 5.3, 5.4 and 5.5 is deadlock-free. Note that S locks on records are never upgraded and that the Fetch operation locks only one record, the operations Insert and Delete lock two records in ascending key order, and the operations Undo-delete and Undo-insert acquire no record locks at all. For transactions containing multiple operations in their forward-rolling phase, the assumption of accessing records in ascending key order is needed to avoid deadlocks that may occur when transactions lock two or more records in a different order. Finally, no deadlock can be caused by the interaction of record locks and page locks, because no transaction is made to wait for a lock on a record covered by page P while holding a lock on P and because the S locks on other pages possibly held by a transaction waiting for a record lock can always be released should the pages be called back. □

5.9 Restart Recovery

The restart recovery is almost identical to the one in Chapters 3 and 4, except that when a redo-undo log record of type “delete” or “insert” is encountered, then the logged update is undone using Algorithm 5.4 or 5.5, respectively, without acquiring U locks on the affected pages. Theorem 4.9 also holds for the algorithms presented in this chapter for a PS-AA system.

5.10 Discussion

Our new B-link-tree algorithms for a PS-AA page server system in which concurrency control and the replica management performed adaptively are most suitable for a setting in which concurrency is a critical issue as in RDBMS applications. In these applications, a typical client transaction contains only few update operations and sometimes key-range scans. In these algorithms, we used the immediate-propagation policy. That is, when a client transaction updates an index or data page P, then copies of updated page P and the log record are sent to the server, so that another client transaction that is waiting for a copy of P can proceed. While

this policy is most suitable for index pages, a transaction performing n updates on a leaf page P needs to request the server n times to get a U lock on P and a copy of P.

Chapter 6

Conclusions

6.1 Summary of the Main Results

In the B-tree algorithms previously published for centralized and client-server systems, tree-structure modifications remain a challenge to concurrency control, recovery and tree-balancing. B±tree-structure modifications are not handled adequately in [Moha90, Moha92b, Moha96], because structure modifications are serialized and no leaf-page updating can be executed while a structure modification is going on. That is, for the sake of correct recovery, concurrency is sacrificed. In [Gray93], the whole search path need to be X-locked at once before executing the needed structure modifications. When leaf-page updating and structure modifications execute concurrently, correct recovery is not guaranteed. In [Lome92, Lome97], 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 parents. Moreover, no algorithms for implementing these atomic actions or recovery were described. In [Zaha97], a relaxed approach is used to execute tree-structure modifications by doubly-linking tree pages on all levels, but still structure modifications are serialized and tree balance is not guaranteed.

In the new B±tree and B-link-tree algorithms presented in this thesis, the recoverability and concurrency problems were solved by defining each structure modification as a small atomic action that updates at most three B±tree pages or at most two B-link-tree pages. Each structure modification 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 [Moha92a] will always produce a structurally consistent and balanced tree, on which the

database updates by aborted transactions can be undone logically (see Theorems 3.9 and 4.10). Our algorithms are deadlock-free (see Theorems 3.6, 4.7 and 5.4), and structure modifications can run concurrently with other structure modifications and leaf-page updates.

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. In the case of a B-link tree, the balance conditions include that at no level of the tree must there be two successive pages that both are indirect children of their parents. This guarantees that the search path of any database record is at most twice the height of the tree (see Lemma 4.4). To maintain the balance conditions under record updates and tree-structure modifications, we defined the concept of a "safe page" (see Section 4.2) 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 it 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 indeed is maintained under all circumstances (see Lemma 4.3 and Theorem 4.8).

We assumed that our database system is a page-server system, in which client transactions perform their operations on copies of B±tree or B-link-tree pages cached at the clients. In the current page-server systems that apply inter-transaction caching, cache consistency is maintained by page-level callback locking using S and X locks, and no difference is usually made between data pages and index pages. In these systems data contention may be high and client transactions may starve. Although some proposals exist for handling of B-trees in page-server environments [Gott96, Basu97, Zaha97], the proposed methods do not provide explicit recovery algorithms and do not discuss tree-structure recovery or transaction rollback in sufficient detail. Also the model of client transactions assumed seems to be rather restricted. This is in contrast to our model, in which each client transaction can contain any number of record fetch, insert and delete operations, and any number of forward-rolling and backward-rolling client transactions can be running concurrently (see Theorems 3.7, 3.8, 4.8 and 4.9).

To increase the concurrency in PS-PP page-server systems, we improved the current page-level concurrency-control and replica-management protocols for these systems by augmenting both protocols with a page-level U-locking protocol [Gray93]. Thus, when an updating client transaction traversing the B±tree or B-link tree acquires U locks on the pages in the search path, those pages can simultaneously be cached and read by transactions at other clients. The improved replica-management protocol is

starvation-free (see Theorem 3.1). To avoid data contention that may arise in PS-PP systems, we presented new adaptive replica-management and concurrency-control protocols for PS-AA systems, so that concurrency is enhanced and client transaction starvation is avoided (see Theorem 5.1).

6.2 Extensions and Future Research

We have assumed that our B±trees and B-link trees are sparse unique indexes to the database. Our algorithms are easily modified to work for dense indexes as well. In a dense index, the leaf pages of the tree contain index records of the form (k, t), where k is the unique key of the record and t is the tuple identifier of the database record (k, x) stored in a separate data file. In this case, the insertion of a new record (k, x) to the database by a client transaction T includes the insertion of (k, x) into the data file and the insertion of the corresponding index record (k, t) to the B±tree or B-link-tree index. Structure modifications needed to extend the data file, such as allocating a new page for a heap file structure, are handled using the same principle we have used for the tree-structure modifications: they are logged using a single redo-only log record. Adapting our algorithms to non-unique indexes involves more changes because the database operations have to be defined differently. The isolation anomalies are also different, and a version of the key-range locking protocol defined for non-unique indexes [Moha90, Gray93, Moha96] must be used.

To make the presentation of our B±tree and B-link-tree algorithms readable and simple, we assumed that each client can run only one transaction at a time. In this setting, running several transactions concurrently at one client is possible, if each application process has its own private cache. However, our algorithms can easily be adapted to PS-PP and PS-AA systems in which a client can run many transactions concurrently using a shared client cache. To do that, we have to include some server-side functionality to the client-side DBMS. The client-side DBMS must be able to manage the concurrent access by several processes to the shared cache. Only minor changes are needed in the concurrency-control and replica-management protocols. Each client transaction must first acquire the needed page or records locks locally and then globally. For example, when a client transaction T at client A needs to read a database page P, T must first acquire a local S lock on P and afterwards request the server for a copy of P if P is not cached locally. When T needs to update page P, T must first acquire the needed page or record locks locally and afterwards request the server for the corresponding global locks.

In our algorithms for a PS-AA system, updates by a client transaction T on a leaf page P are propagated immediately to the server. To postpone update propagation until P is called back or T commits, our replica-management and concurrency-

control protocols should be modified as follows. If transaction T2 at client B requests the server to get a U lock on a data page P and a copy of P where P is cached and U-locked by transaction T1 at client A, then instead of blocking T2's request, the server sends a callback request for page P to client A. If T1 has completed its updates on P, then T1 releases the local U lock on P and A sends the updated copy of P and the log record to the server, and purges P from its cache.

References

[Baye77] R. Bayer and M. Schkolnick, Concurrency of operations on B-trees. *Acta Informatica* **9** (1977), 1-21.

[Basu97] J. Basu, A. Keller, and M. Pöss, Centralized versus distributed index schemes in OODBMS – a performance analysis. *Proc. of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97)*, St.-Petersburg, 1997, pp 162-169.

[Bern87] P. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

[Bill87] A. Biliris, Operation specific locking in B-trees. In: *Proc. of the 1987 ACM International Conference on Principles of Database Systems*, pp 159-169.

[Care91a] M. Carey, M. Franklin, M. Livny, and E. Shekita, Data caching tradeoffs in client-server DBMS architectures. In: *Proc. of the 1991 ACM SIGMOD International Conference on Management of Data*, pp 357-366.

[Care91b] M. Carey and M. Livny, Conflict detection tradeoffs for replicated data. *ACM Trans. Database Systems*, **16** (1991), 703-746.

[Care94a] M. J. Carey, *et al.*, Shoring up persistent applications. In: *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data*, pp 383-394.

[Care94b] M. Carey, M. Franklin, and M. Zaharioudakis, Fine-grained sharing in a page server OODBMS. In: *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data*, pp 359-370.

[Deux91] O. Deux, The O_2 system. *Commun. ACM*, **34** (1991), 50-63.

[Dewi90] D. J. DeWitt, D. Maier, P. Futtersack, F. Velez, A study of three alternative workstation-server architectures for object-oriented

database systems. In: *Proc. of the 16th VLDB Conference*, 1990, pp 107-121.

[Fran92a] M. Franklin, M. Zwilling, C. Tan, M. J. Carey, D. DeWitt, Crash recovery in client-server EXODUS. In: *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, pp 165-174.

[Fran92b] M. Franklin, M. Carey, Client-server caching revisited. In: *Proc. of the 1992 International Workshop on Distributed Object Management*, Morgan Kaufmann, 1994, pp 57-78.

[Fran92c] M. Franklin, M. J. Carey, M. Livny, Global memory management in client-server DBMS architecture. In: *Proc. of the 18th VLDB Conference*, 1992, pp 596-609.

[Fran93] M. Franklin, M. J. Carey, M. Livny, Local disk caching for client-server database systems. In: *Proc. of the 19th VLDB Conference*, 1993, pp 641-654.

[Fran97] M. Franklin, M. J. Carey, M. Livny, Transactional client-server cache consistency: alternatives and performance. *ACM Trans. Database Systems* **22** (1997), 315-363.

[Fuka89] A. Fu, T. Kameda, Concurrency control for nested transactions accessing B-trees. In: *Proc. of the 1989 ACM SIGACT-SIGMOD-SIGART International Conference on Management of Data*, pp 270-285.

[Gott96] V. Gottemukkala, U. Ramachandran, E. Omiecinski, Relaxed index consistency for data-only locking in a client-server database. In: *Proc. of the 12th IEEE Data Engineering Conference*, 1996, pp 352-361.

[Gray93] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.

[Howa88] J. Howard, *et al.*, Scale and performance in a distributed file system. *ACM Trans. Computer Systems* **6** (1988), 51-81.

[John89] T. Johnson and D. Shasha, Utilization of B-tree with inserts, deletes and searches. In: *Proc. of the 1989 ACM International Conference on Principles of Database Systems*, pp 235-246.

[John93] T. Johnson and D. Shasha, B-trees with inserts and deletes: why free-at-empty is better than merge-at-half. *J. Computer and System Sciences*, **47** (1993), 45-76.

[Korn97] M. Kornacker, C. Mohan, J. Hellerstein, Concurrency and recovery in generalized search trees. In: *Proc. of the 1997 ACM SIGMOD International Conference on Management of Data*, pp 62-72.

[Kwon82] Y. Kwong and D. Wood, A new method for concurrency in B-trees. *IEEE Trans. Software Engineering*, **8** (1982), 211-222.

[Lamb91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, The ObjectStore database system. *Commun. ACM*, **34** (1991), 50-63.

[Lani86] V. Lanin and D. Shasha, A symmetric concurrent B-tree algorithm. In: *Proc. Fall Joint Computer Conference*, 1986, pp 380-389.

[Lehm81] P. Lehman and S. Yao, Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Systems* **6** (1981), 650-670.

[Lome92] D. Lomet and B. Salzberg, Access method concurrency with recovery. In: *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, pp 351-360.

[Lome97] D. Lomet, B. Salzberg, Concurrency and recovery for index trees. *The VLDB Journal* **6** (1997), 224-240.

[Lome98] D. Lomet, Advanced recovery techniques in practice. In: *Recovery Mechanisms in Database Systems* (V. Kumar and M. Hsu, eds.), Prentice Hall PTR, 1998, pp 697-710.

[Maie81] D. Maier, S. Salveter. Hysterical B-trees. *J. Information Processing Letter* **12** (1981), 199-202

[Moha90] C. Mohan, ARIES/KVL: a key-value locking method for concurrency control of multi-action transactions operation on B-tree indexes. In: *Proc. of the 16th VLDB Conference*, 1990, pp 392-405.

[Moha92a] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Systems* **17** (1992), 94-162.

[Moha92b] C. Mohan and F. Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In: *Proc. of the 1992 ACM SIGMOD International Conference on Management of Data*, pp 371-380.

[Moha94] C. Mohan, I. Narang, ARIES/CSA: a method for database recovery in client-server architectures. In: *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data*, pp 55-66.

[Moha96] C. Mohan, Concurrency control and recovery methods for B±tree indexes: ARIES/KVL and ARIES/IM. In: *Performance of Concurrency Control Mechanisms in Centralized Database Systems* (V.Kumar, ed.), Prentice Hall, 1996, pp 248-306.

[Mond85] Y. Mond and Y. Raz, Concurrency control in B±tree databases using preparatory operations. In: *Proc. of the 11th VLDB Conference*, 1985, pp 331-334.

[Nurm87] O. Nurmi, E. Soisalon-Soininen and D. Wood, Concurrency control in database structures with relaxed balance. In: *Proc. of the 1987 ACM International Conference on Principles of Database Systems*, pp 170-176.

[Papa86] C. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

[Poll96] K. Pollari-Malmi, E. Soisalon-Soininen and T. Ylönen, Concurrency control in B-trees with batch updates. *IEEE Trans. Knowledge and Data Engineering* **8** (1996), 975-983.

[Sagi86] Y. Sagiv, Concurrent operations on B*-trees with overtaking. *J. Computer and System Sciences* **33** (1986), 275-296.

[Setz94] V. Setzer and A. Zisman, New concurrency control algorithms for accessing and compacting B-trees. In: *Proc. of the 20th VLDB Conference*, 1994, 238-248.

[Shas88] D. Shasha and N. Goodman, Concurrent search structure algorithms. *ACM Trans. Database Systems* **13** (1988), 53-90.

[Srin91] V. Srinivasan and M. Carey, Performance of B-tree concurrency control algorithms. In: *Proc. of the 1991 ACM SIGMOD International Conference on Management of Data*, pp 416-425.

[Wang91] Y. Wang and L. Rowe, Cache consistency and concurrency control in a client-server DBMS architecture. In: *Proc. of the 1991 ACM SIGMOD International Conference on Management of Data*, pp 367-376.

[Whit94] S. J. White, D. J. DeWitt, QuickStore: a high performance mapped object store. In: *Proc. of the 1994 ACM SIGMOD International Conference on Management of Data*, pp 395-406.

[Wilk90] K. Wilkinson and M. Neimat, Maintaining consistency of client-cached data. In: *Proc. of the 16th VLDB Conference*, 1990, pp 122-133.

[Zaha97] M. Zaharioudakis, M. Carey, Highly concurrent cache consistency for indices in client-server database systems. In: *Proc. of the 1997 ACM SIGMOD International Conference on Management of Data*, pp 50-61.