Batch Updates and Concurrency Control in B-Trees

Kerttu Pollari-Malmi

Helsinki University of Technology
Laboratory of Information Processing Science
P.O.Box 5400, FIN-02015 HUT, Espoo Finland

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 15th of April, 2002, at 12 noon.

ISBN 951-22-5919-2
ISSN 1239-6885

Otamedia Oy
Espoo 2002

Abstract

When a large number of new keys are to be inserted (or deleted) into a B-tree at about the same time, it is often profitable to sort the keys in the main memory before performing updates to the B-tree on disk. Thus, updates falling into the same leaf of the B-tree can be performed simultaneously and disk accesses are saved. In this thesis, new batch-update algorithms for B-trees are presented and analyzed.

First, an efficient batch-insertion algorithm for a situation where no concurrency control is needed is presented and analyzed. The algorithm is asymptotically optimal in the number of structure modification operations needed to rebalance the tree after the batch insertion.

Next, new batch-update algorithms for different types of B-trees such that the batch update can be performed concurrently with other actions are presented. Experimental tests have been performed to compare the new algorithms with an earlier algorithm. According to simulation results, the new algorithms allow much more concurrency than the previous algorithm.

Finally, a differential indexing system is presented. In the system a database index is divided into two parts: the main index located on disk and the differential index in the main memory. Periodically, writes of committed transactions are transferred from the differential index to the main index as a batch-update operation.

UDK: 004.651.4:004.422.635.32
Keywords: B-tree, concurrency control, batch update, differential index.

Preface

The research presented in this thesis was carried out at Helsinki University of Technology in the Laboratory of Information Processing Science. The work was partially financed by the Academy of Finland, by the Emil Aaltonen Foundation, and by the Foundation of Technology in Finland.

I wish to thank Professor Eljas Soisalon-Soininen, the supervisor of this thesis. Without the ideas he gave me this thesis would have never been started nor completed. I am grateful to him for his guidance, comments and patience during this work.

I would also like to thank Tatu Ylönen for his help and for the information he gave me about his full-text-indexing system and Jarmo Ruuth for the information he gave me about his differential indexing system.

I wish to thank Professor Heikki Saikkonen for his guidance during my post-graduate studies.

I thank all my present and former colleagues in the Laboratory of Information Processing Science for creating an excellent work environment. In addition, I want to thank especially Pekka Mård, Esko Nuutila, Kenneth Oksanen, and Satu Virtanen for their help.

I thank professors Matti Linna and Kim S. Larsen for their thesis review and their valuable comments.

I am grateful to Diane Pilkinton-Pihko for the checking of the English language.

I present special thanks to Irmeli Borgman who has worked as the day-care mom of my children for many years. Because of her help, I have been able to dedicate myself to my work.

Finally, I want to thank my husband, Lauri, not only for his comments on this thesis, but above all for his love and support during all these years we have lived together. I thank our children Katri, Hanna-Mari and Mikko for their love and for the joy they have provided in my life.

Otaniemi, March 2002.

Kerttu Pollari-Malmi

Contents

Chapter 1

Introduction

1.1 Dictionaries and Batch Update

When a large database is used, one or more dictionaries for each relation are usually needed to assure an efficient access to a desired data item. A dictionary is a data structure that supports the actions *search* (often called *member*), *insert* and *delete*. Action search searches the dictionary for a data item having a certain value as its key or in another field. The answer returned may just tell whether the data item was found or it may tell the location of the item in the physical memory. Insert and delete are used to insert and delete data items into or from the dictionary.

The dictionary can be either a *primary key index* or a *secondary index*. In a primary key index all data items are indexed according to their key values, which are unique for this relation. In a secondary index the data items are indexed according to a field value that may be non-unique. Thus there may be several data items belonging to the same index key value.

If the dictionary is in the external memory, an efficient and widely used data structure to implement it is a B-tree [3]. Another widely used solution is hashing [15, 30] and its variants, but in this thesis we concentrate on B-trees.

The efficiency of B-trees comes from the fact that only $O(\log_k N)$ disk accesses are needed for each dictionary operation, where N is the number of keys in the dictionary and k is the minimum number of keys in one node. Moreover, because in practice k usually varies between the range of 100–1000, B-trees usually have only three levels, and two upper levels can often be kept in the main memory. However, even one disk access per operation may be too inefficient if a large number of new keys are to be inserted (or deleted) at about the same time.

A significant improvement can be achieved by inserting the keys as a *batch*. This means, basically, that the keys to be inserted are sorted before insertion and all keys that should be inserted into the same leaf of the B-tree are inserted simultaneously.

Moreover, the search for the next leaf is started from the parent of the previous leaf, not from the root. This leads to significant savings in disk accesses.

There are typical applications in which batch updates are important. One of them is full-text indexing when the inverted-index technique [7, 19, 53, 67] is applied. For each index term (i.e. for each word that occurs at least once in the documents), a variable length occurrence list is created that gives all documents in which the term occurs. The index terms are usually organized as a B-tree, such that given a particular index term, the corresponding occurrence list is found in the leaf where this term (key) has been stored. It has been reported that one drawback of the inverted-index technique is that when inserting a new document, an update of the inverted index is required for each term in the document [17, 19]. However, this is a typical situation where a batch update can be used, leading to considerable improvements in the efficiency. Experience with a commercial text database system presented in [53, 67] confirms this—actually updating the index was intolerably slow when batch update was not in use. An approach similar to batch update was taken in [10].

A more general use of batch update comes from deferring the installation of single updates. At specific time intervals all pending updates are collected as a batch and brought to the main index [34, 52, 63]. This requires, however, that two indexes are in use, the differential index residing in the main memory and the main index usually residing on the disk.

Another application of batch update comes from on-line index construction [48, 60, 61]. If a new index is required for, e.g., a large relation in a relational database, then in on-line indexing this relation can be accessed and updated concurrently with the index construction. All updates are collected into a separate structure, which—after the main construction has finished—will be merged with the main index.

A recent application of batch update is data warehousing, e.g. [33]. A data warehouse is a repository of integrated information from distributed, autonomous and possibly heterogeneous sources. The warehouse stores one or more materialized views of the source data. The views are refreshed periodically by maintenance transactions, which propagate large batch updates from the base tables.

In this thesis, we address the concurrency control problem together with efficient batch update. This problem occurs, for example, in a text database system supporting a search engine for WWW pages. New WWW pages are written constantly and old existing pages are modified. Therefore, the indexers of search engines should work constantly and at the same time, the users of the search engine should be able to perform searches as efficiently as possible. When a new WWW page is indexed, the corresponding index terms are collected as a batch. Concurrently with this batch insertion, searches of the database through the same index must be allowed.

Another need for concurrency control occurs in on-line index construction, for example, when during the build phase (updates are merged with the main index) new updates are allowed. However, only special purpose solutions are presented in [48, 60, 61], without analyzing the tradeoff between the efficiency of batch update and the allowed degree of concurrency.

In this thesis we give new algorithms for the batch update such that it can be performed concurrently with other operations (possibly with other batch updates) while preserving its efficiency. Allowing concurrency means, however, that some of the efficiency of the batch update may be lost. In [67], Ylönen presented a version where the batch update is kept as efficient as possible, while allowing a reasonable amount of concurrency. This is obtained by performing atomically (in most cases) all updates to be done in leaf nodes below a single parent node. In the algorithms presented in this thesis, we improve concurrency by using two basic solutions. First, we use a B-tree type which allows readers in the parent node while its children are being updated. Second, we perform atomically only those updates that are to be done in single leaf nodes. Intuitively, and confirmed by our experiments, these solutions allow much more concurrency than the previous algorithm, but the previous algorithm is more efficient if the speed of the batch update is considered alone.

We also present a differential indexing system where a database index is divided into two parts: the main index located on disk and the differential index in the main memory. Periodically, writes of committed transactions are transferred from the differential index to the main index as a batch-update operation.

1.2 Contributions

The main contributions of this thesis are the following:

- We present a new efficient batch-insertion algorithm for a situation where no concurrency control is needed and analyze the time complexity of the algorithm.

- We present new batch-update algorithms for different types of B-trees such that the batch update can be performed concurrently with other actions. We have implemented a simulation system to compare the new algorithms with an earlier algorithm presented in [67]. The simulation results show that the new algorithms allow much more concurrency for other actions performed during the batch update than the previous one.

- We present a differential indexing system where a database index is divided into two parts: the main index located on disk and the differential index in the main memory. This system was originally constructed and implemented by Jarmo Ruuth [55]. This thesis provides a more formal presentation of the differential indexing system including the correctness proofs of the concurrency control algorithms.

Parts of these contributions have been published in [52] and [53].

1.3 Contents of the Thesis

This thesis is organized as follows. The B-tree and some of its versions are presented in Chapter 2. Concurrency control in B-trees and in databases in general is discussed in Chapter 3. In Chapter 4 we define what is meant by a batch update of a B-tree, and also review the relevant literature. In Chapter 5 we present a batch-insertion algorithm without concurrency and analyze the time complexity of the algorithm. Chapter 6 is devoted to the issues relevant to the concurrency control and its correctness when batch updates are present. We also present different concurrent batch-update algorithms for B-trees. In Chapter 7 we discuss how to use the batch update in full-text indexing. We present some experimental results in Chapter 8. Chapter 9 presents a differential indexing system and Chapter 10 is the conclusion.

Chapter 2

B-Trees

In this chapter, we describe different types of B-trees and explain how the actions search, insert and delete are implemented in them.

2.1 B$^+$-Trees

An ordered multi-way tree, called a B-tree, was originally introduced in [3]. Henceforth, we call this type of B-tree a *classical B-tree*. However, this chapter is started by describing the most frequently used type of the B-tree. It is a leaf-oriented B-tree, often referred to as a B$^+$-tree [9, 30]. Leaf-oriented means that all data are stored in the leaves and the internal nodes contain only pointers and routing information to aid the search in the tree.

An internal node can be represented by a tuple $[P_0, K_1, P_1, K_2, P_2, \ldots, K_n, P_n]$, where K_i is a router and P_i is a pointer. $P_{i-1}, 1 < i < n$, points to the subtree where all keys are larger than K_{i-1} but smaller than or equal to K_i. P_0 points to the subtree where all keys are smaller than or equal to K_1 and P_n points to the subtree where all keys are larger than K_n.

A B-tree is balanced if each node except the root is at least half full and the length of the path from the root to a leaf is equal for all leaves.

More formally, let u be a node. Denote by $c(u)$ the number of children of u if u is an internal node, and the number of keys in u if u is a leaf. If u is not the root, we denote its parent by φu. The function $l(u)$, defined by

$$l(u) = \begin{cases} 0, & \text{if } u \text{ is the root,} \\ l(\varphi u) + 1, & \text{otherwise,} \end{cases}$$

is the level of u.

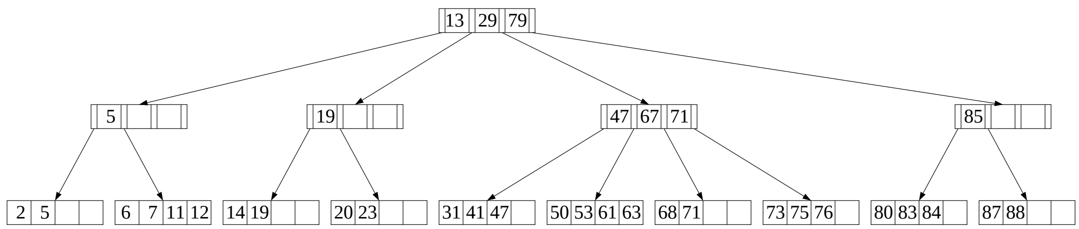


Figure 2.1: A B^+-tree.

Now let $k \geq 2$ be an integer. A B^+-tree of order $2k$ is an ordered multi-way leaf-oriented tree with the following properties:

B1. $2 \leq c(u) \leq 2k$, if u is the root,
$k \leq c(u) \leq 2k$, otherwise.

B2. If u_1 and u_2 are two leaves of the tree, then
$l(u_1) = l(u_2)$.

An example of the B^+-tree is presented in Figure 2.1.

The normal operations for B-trees are search(k), insert(k) and delete(k), where k is the key that is searched for, inserted or deleted. Some papers, for example [46], also present operation *fetch next*, but we do not cover it in this thesis.

Searching for the key k is performed as follows: starting from the root, each node is searched for the pointer to the next node according to the router values until a leaf node is reached. In each internal node, the largest router value K_i that is smaller than k is searched for and the next node will be the node pointed to by corresponding pointer P_i. If $k \leq K_1$, the next node will be the node pointed to by P_0. Finally, the leaf found is searched for the key value and according to the result either a positive or negative answer is returned.

Insertion is started by searching for the leaf node u where the key belongs. If the key is already in the node, nothing is done. Otherwise, if there is enough space in the node for the new key, i.e. $c(u) < 2k$, the key is inserted into the node. If the node is full, it must be split. A new node is created and about half of the contents of the original node is transferred to the new node. The key is inserted into one of the nodes. A router and a pointer to the new node are inserted into the parent node. If there is not enough space for them in the parent node, it must be split in the same way. In the worst case the splitting advances up to the root. Figure 2.2 presents the result when a key is inserted into the B^+-tree of Figure 2.1.

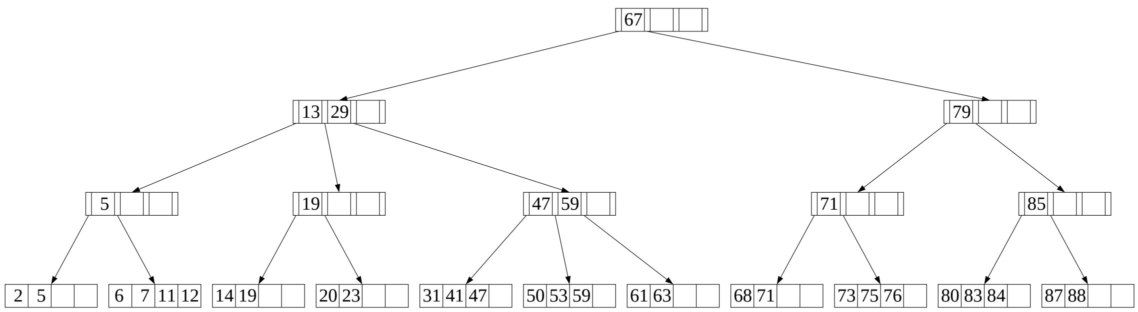


Figure 2.2: The B$^+$-tree of Figure 2.1 after inserting key 59.

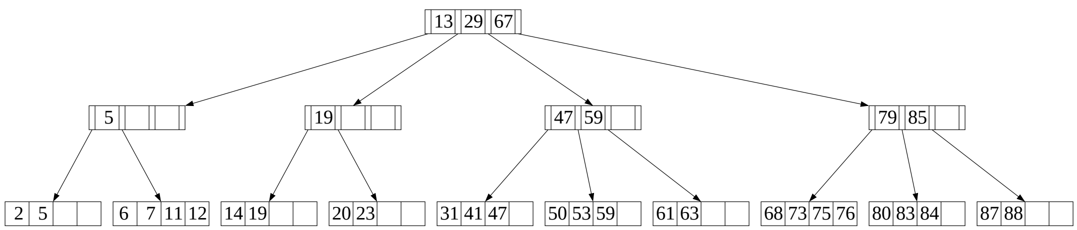


Figure 2.3: The B$^+$-tree of Figure 2.2 after deleting key 71.

Deletion is also started by searching for the leaf node v where the key should be. If the key is not found in the node, nothing is done. Otherwise, the key is removed. If the node remains full enough after that, i.e. $c(v) \geq k$, the deletion is finished. Otherwise, the node must either be combined with its sibling or the contents of the node and its sibling must be redivided between these nodes. The action chosen depends on the total number of keys in the two nodes. If the contents of the nodes are divided between these nodes, the router for the right sibling must be adjusted accordingly in the parent node. If the node is combined with its sibling, the router and the pointer to the deleted node must be deleted from the parent. If the parent does not contain enough keys after that, it must be adjusted with its sibling. Again, this may advance to the root, see Figure 2.3.

We described above what to do if the node is too full for the insert operation or remains too empty after the delete operation. The described way to perform balancing is called *bottom-up balancing*. The other possibility is to use *top-down balancing*. For insertion, when the search process traverses downwards in the tree, in each internal node a check is performed that there is enough space for at least one pointer and router. Otherwise, the node is split and the new pointer is inserted into the parent. (Because of top-down balancing, there is always enough space for it in the parent.) If the leaf node is full before the insertion, it is split and the new pointer and the new router are inserted into the parent. For deletion, the corresponding check for each internal node is that there are enough routers so that one can be removed. Otherwise, the contents of the node are transferred to its sibling or the total contents of these two nodes are divided between these nodes. The pointer and the router to the node are removed or modified accordingly. Because of top-down balancing, this is always possible.

The classical B-tree, presented in [3], differs from the B$^+$-tree in the sense that all its nodes contain actual key values. Insertions are always done into leaves, but searches and deletions may end in any node. If the key to be deleted is in a non-leaf node, it is replaced with the next highest key of the tree. The next highest key is always in a leaf node, and after the replacement it is deleted from the leaf.

A B-tree where each node is required to be at least two thirds full is called a *B^*-tree* [30]. The advantage of B*-trees is that they use less extra space than normal B-trees. However, this is achieved at the cost of more complex operations. If a node has to be split in insertion, its sibling must be accessed and if a node remains too empty after deletion, two sibling nodes may have to be accessed. Because even B$^+$-trees achieve 69 percent space utilization when random insertions and deletions are performed and the number of insertions slightly exceeds the number of deletions (see [28]), B*-trees are rarely used. Some papers, for example [56], use the term B*-tree when they refer to a B$^+$-tree.

Henceforth, when we use the term B-tree in this thesis, we mean a B$^+$-tree unless otherwise noted.

2.2 Relaxed B-Trees

The relaxed B-tree was introduced in [50]. The main idea of the relaxed B-tree is that some balance conditions are relaxed in such a way that the tree can be easily rebalanced with local operations. The motivation for this is that in situations where many processes work concurrently it is profitable if only local operations are needed for rebalancing. To make this possible a *tag* with value -1 or 0 is added to each node.

Let u be a node of a tree. The relaxed level of u, denoted by $rl(u)$, is defined by:

$$rl(u) = \begin{cases} 0, & \text{if } u \text{ is the root,} \\ rl(\varphi u) + 1 + tag(u), & \text{otherwise,} \end{cases}$$

where φu denotes the parent of u.

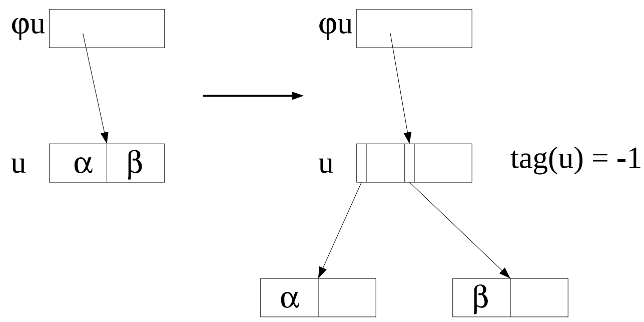


Figure 2.4: Split of u when key k to be inserted does not fit. The node u is first split (shown in the figure) and then k is inserted.

Denote by $c(u)$ the number of pointers in u if u is an internal node, and the number of keys in u if u is a leaf. Let $k \geq 1$ be an integer. An ordered multi-way leaf-oriented tree is a *relaxed B-tree of order* $2k$, if it has the following properties:

RB1. $0 \leq c(u) \leq 2k$, if u is a leaf,
$1 \leq c(u) \leq 2k$, otherwise.

RB2. If u_1 and u_2 are two leaves of the tree, then
$rl(u_1) = rl(u_2)$.

Clearly, a B-tree is a relaxed B-tree. A relaxed B-tree is a B-tree, if the tag of each node is zero and the density condition of B-trees (all the nodes except the root are at least half full) is satisfied.

The search for a key is carried out in exactly the same way as in B-trees. The update operations are designed in such a way that they preserve the properties RB1 and RB2. These operations make only local changes in the tree, see [50].

Insert(k): The insertion first searches for the leaf u where the new key k should be stored. If there is enough space for the new key, it is inserted. Otherwise, we introduce two new leaves, move the leftmost half of the contents of u into the first node, and the rightmost half into the second new node. The new key is then inserted into the first or second new leaf. Then the node u is put as the parent of the new leaves and the routers of u are set accordingly. Last, the tags of the leaves are set to zeroes, and the tag of u is set to -1. The insertion in an overflow situation is presented in Figure 2.4.

Delete(k): The leaf having the key k to be deleted is searched for. If k is found, it is deleted from the leaf. No further action takes place.

Two local operations, split and compress are used for changing a relaxed B-tree closer to a conventional B-tree.

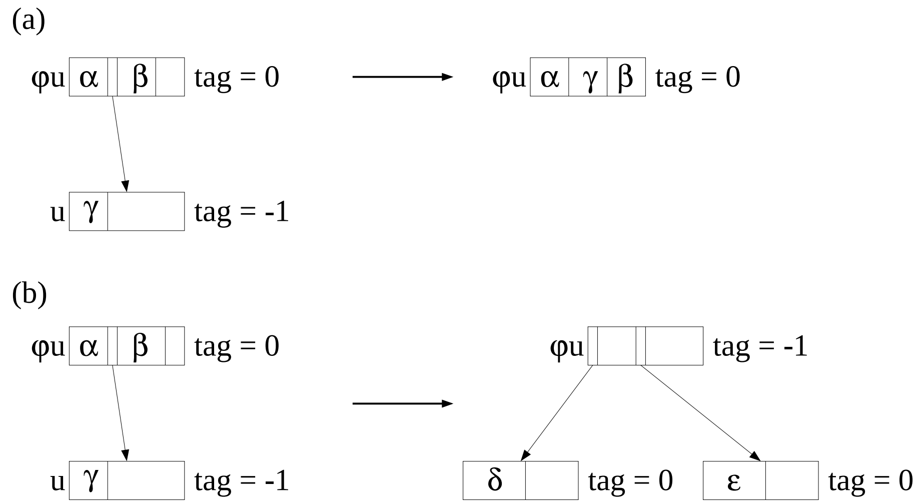


Figure 2.5: The split operation in a relaxed B-tree: $\alpha\gamma\beta = \delta\epsilon$

Split: Split is used to merge a node having a tag -1 with its parent node. If u is a node with the tag -1 and φu is its parent, the contents of u are moved into φu. If there is enough space in φu, the node u is deleted (see Figure 2.5(a)). If there is not enough space in φu, a new node is created. The contents of φu and u are divided between u and the new node. The node φu will have only two pointers, one to u and one to the new node. The tags of u and the new node are set to zero and the tag of φu is set to -1, if φu is not the root (see Figure 2.5(b)).

Compress: Let u be the node that is too empty, but has a tag value 0, and let φu be its parent. The contents of u are moved to its left or right sibling, if they have enough empty space, and node u is deleted. The pointer to u in φu and the corresponding router are also deleted. If the contents of u do not fit into its sibling, u and the contents of its left (or right) sibling are equally divided between these two nodes. The routing information in φu is updated accordingly. The compress operation can be performed only, if u has a sibling with tag value 0. If u has only siblings with tag value -1, a split operation in a sibling has to be carried out before the compress operation in u can be performed.

If enough split and compress operations are performed into a relaxed B-tree, it will eventually be modified into a B^+-tree. Larsen and Fagerberg have shown in [39] that when enough insertions and deletions are performed to a relaxed B-tree, the amortized rebalancing cost per insertion or deletion is constant.

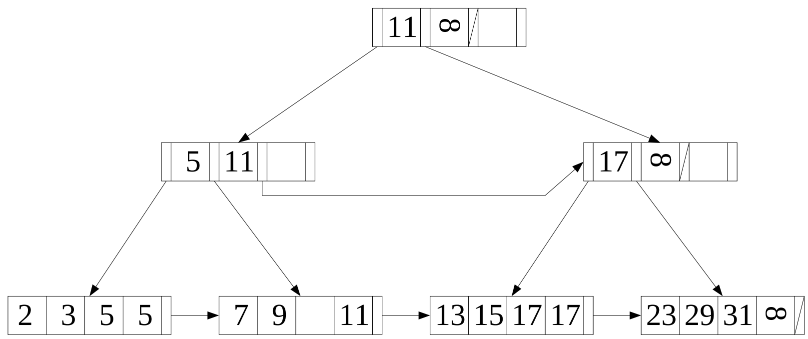


Figure 2.6: A B^{link}-tree.

2.3 B^{link}-Trees

B^{link}-trees were originally introduced in [40] and some improvements were proposed in [35, 56]. B^{link}-trees differ from B-trees in that each node contains an extra router, called a *highvalue*, and an extra pointer, called a *link*. The highvalue tells the highest key value that can be found in the subtree whose root is this node. The link is a pointer to the right sibling of this node. An example of a B^{link}-tree is presented in Figure 2.6. Because the nodes form a linked list in each level, the search structure is valid even if a parent node does not contain pointers to all its children. The validity of the search structure means that every key which is in the tree can be found starting from the root.

Search in a B^{link}-tree proceeds in the same way as in a B-tree, except when the search key is larger than the highvalue of the current node. In that case, the next node is obtained by following the link to the right sibling. Insertion is started by searching for the leaf node where the key belongs. If there is enough space in the node, the key is inserted. Otherwise, a new node is created and half of the contents of the current node is transferred to the new node. The key is inserted into one of these nodes. The new node receives the link and the highvalue of the current node. The link of the current node is modified to point to the new node, and the highvalue of the current node is changed. Now the search structure is already valid, see Figure 2.7(a). After that the router and the pointer to the new node are inserted into the parent, see Figure 2.7(b). If there is not enough space in the parent, it must be split in the same way.

In [40] it is proposed that a delete operation just remove the key from the leaf node and that no further action takes place. This means that it is not required that each leaf should be at least half full. Sagiv [56] proposes another approach: Nodes that are

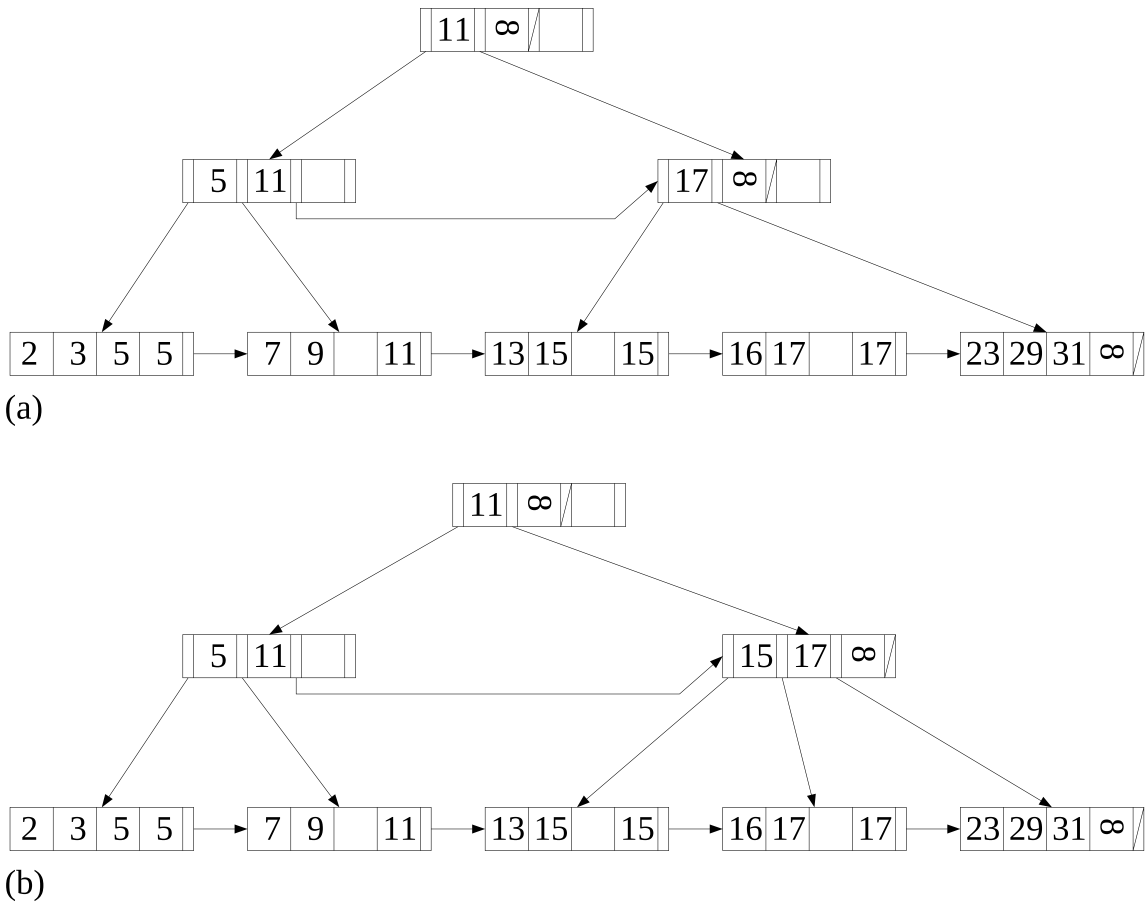


Figure 2.7: The B$^{\text{link}}$-tree of Figure 2.6 after inserting the key 16. (a) The insertion has been finished, but the balancing is not yet done. (b) The balancing has been finished.

not full enough are adjusted with their siblings by a separate process that traverses each level of the tree and searches for the nodes needing adjusting. When such a node is found, an operation similar to the operation compress of the relaxed B-tree is performed. In [35] the node merge is combined with deletion: If two siblings are merged together, their contents are transferred to the left sibling. A special link, outlink, is inserted into the deleted node to point to the left sibling so that other processes which have read a pointer to the deleted node can find the sibling. Modifications needed in the parent node and possibly higher in the tree are performed as in an insert operation.

2.4 (a, b)-Trees

An (a, b)-tree [25, 44] is a more general type of multi-way tree. A B-tree is a special case of (a, b)-trees.

Let a and b be integers with $a \geq 2$ and $b \geq 2a - 1$ and u a node of a multi-way search tree T. We denote by $c(u)$ the number of children of u, if u is an internal node and the number of keys in u, if u is a leaf node. Tree T is an (a, b)-tree, if the following conditions are satisfied:

1. All paths from the root of T to a leaf node have the same length.
2. $2 \leq c(u) \leq b$, if u is the root of T.
3. $a \leq c(u) \leq b$ for all nodes u except the root.

Otherwise, an (a, b)-tree is similar to a B-tree. Operations search, insert, and delete are implemented as described in Section 2.1, except that a compress operation is performed if a leaf node contained less than a keys or an internal node had less than a children after a deletion. It is also possible to construct relaxed (a, b)-trees [38, 39] and $(a, b)^{\text{link}}$-trees.

Chapter 3

Concurrency Control in B-Trees and Databases

There are many applications in which several users want to access a database and its dictionary concurrently. An example of this is an airline reservation system, where many sales agents may be selling tickets. They want to be able to reserve seats without extra delays, but selling the same seat twice by different sales agents has to be prevented.

Another example is a text database supporting a newspaper house. When a new article is added to the database, the corresponding index terms are collected as a batch. The articles obtained from a news agency should be inserted into the database at once. Concurrently with this batch insertion, searches of the database through the same index must be allowed.

When concurrency control is needed for a database having B-tree dictionaries, there are two different levels to consider. Firstly, it must be ensured that the B-tree structure behaves correctly in concurrent situations. This means, for example, that an operation searching for data object x really finds x if it is in the dictionary. Secondly, in the database level it is possible to join several actions together to form a single *transaction* [14]. In concurrent situations we demand that although several transactions may be working concurrently, it will seem to every transaction T that each of the other transactions is completely performed either before or after T. This is called serializability.

In this chapter we first discuss concurrency control of B-trees and then concurrency control of databases in general. A more detailed description about concurrency control methods of B-trees can be found in [13].

3.1 Concurrency Control in B-Trees

When a database has a dictionary which is implemented as a B-tree, the actions that use the dictionary have to fulfill certain requirements to make it possible to achieve correct behavior in concurrent situations. First, for any set of actions that access the B-tree and for any possible schedule that can implement it, there must be a serial schedule of the same actions so that each action returns the same value in both schedules and the logical data in the B-tree is the same after both schedules. In addition to that, the validity of the search structure has to be preserved all the time. The search structure is valid if a key that is in the tree is always found if it is searched for.

An example of a situation where one of the actions returns an incorrect answer when concurrency control is not used is presented in Figure 3.1. The processes insert(7) and search(8) run concurrently. The latter action returns a wrong answer, because it has read a pointer to the node D before this node was split, but it searches D for key 8 after the split. Of course, we want to prevent situations like this and that is why concurrency control is needed.

By using *locks*, one action can prevent other actions from accessing a certain part of the dictionary that it is processing. Normally, the action keeps locks on one or more nodes in the tree.

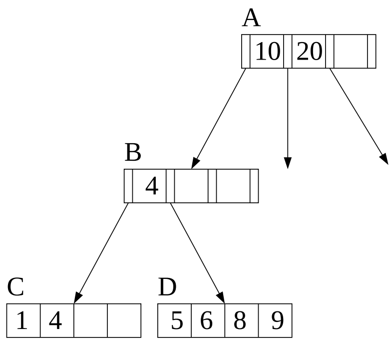
Most concurrency control methods use *lock-coupling* when advancing in the tree. When the parent is searched for the pointer to a child, the parent is kept locked. When the child is found, it is locked. The lock of the parent can be released after the lock on the child has been admitted. The lock of the parent is not released earlier, because otherwise another process could change the range of keys that can be found in the sub-tree whose root is the child.

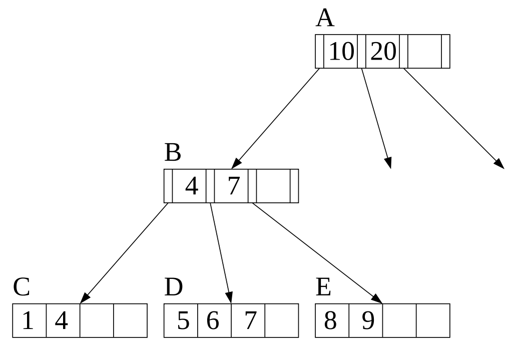
The processes accessing the tree can be divided into two categories, readers and writers. A reader is a process that performs searches in the tree, but does not modify the tree. A writer is a process that may modify the tree.

The simplest locking method uses only one kind of lock. Both readers and writers use exclusive locks and thus no two processes can access any node simultaneously. However, this is unnecessarily restrictive, because there is no need to prevent two or more readers from accessing the same node simultaneously.

A more advanced method uses two types of locks: *read locks* (called here r-locks) and *exclusive locks* (called here x-locks). Only one process can hold an x-lock on a node, but any number of processes can hold r-locks on a node. If a process holds an x-lock on a node, no other process can have an r-lock on the same node. Readers use r-locks when advancing in the tree and writers use x-locks.

It may seem that we could allow more concurrency by letting a writer first use r-locks when advancing in the tree and later change this r-lock to an x-lock when it is





insert(7)	search(8)
	node<- A
	read(A)
node<- A	
read(A)	
parent<- A	
node<- search(7,A) = B	
read(B)	
	parent<- A
	node<- search(8,A) = B
	read(B)
parent<- B	
node<- search(7,B) = D	
read(D)	
	parent<- B
	node<- search(8,B) = D
create E	
transfer 8 and 9 into E	
insert 7 and pointer to E into B	
insert 7 into D	
write(B)	
write(D)	
write(E)	
	read(D)
	search D for 8
	return NOT FOUND

Figure 3.1: An example of a situation where one of the concurrent processes returns a wrong answer because concurrency control is not used.

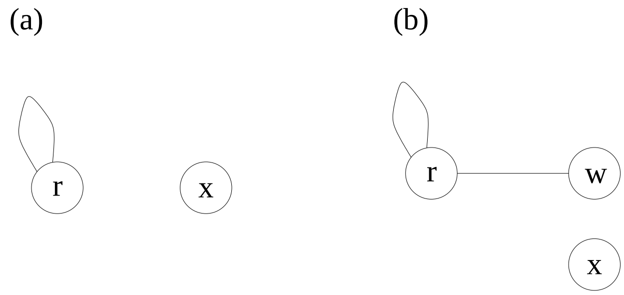


Figure 3.2: Compatibility graphs of two locking schemes: (a) two types of locks, (b) three types of locks.

clear that the current node has to be modified. However, this is not possible, because it may lead to *deadlock*. Consider a situation where two writers have an r-lock on the same node. Both writers want to modify the node and thus they try to change their r-lock to an x-lock. However, neither process succeeds, because the other process holds r-lock on the node. On the other hand, neither process releases its r-lock before it gets an x-lock.

The problem can be solved by introducing a third type of lock, a *w-lock*. A writer uses a w-lock to exclude other writers, but to allow readers to access the same node. Thus, if a process has a w-lock on a node, other processes can r-lock but not w-lock nor x-lock the node. Accordingly, a process can w-lock a node if it is not w-locked nor x-locked by another process. A writer advances in the tree using w-locks. When it finds a node it wants to modify, it changes its w-lock to an x-lock. This locking scheme was first introduced in [11, 12] and was originally developed for AVL and 2-3 trees.

It is also possible to describe locking schemes by using compatibility graphs. Such a graph contains a node for each type of lock used. An edge between any two nodes in this graph means that two different processes may simultaneously hold these locks on the same node of the tree. Figure 3.2 shows compatibility graphs for locking schemes using two and three types of locks.

When there are several concurrent processes accessing the B-tree simultaneously, the efficiency of the processes highly depends on the number of nodes that each process has to keep simultaneously locked and on the duration of the locking times. In particular, the time each process keeps the root node locked should be as short as possible.

The number of nodes that have to be locked simultaneously depends on the balancing method used. If bottom-up balancing is used, in the worst case a writer has to lock the whole path from the root to the leaf where the modification should be done. This is because if it is necessary to split a leaf node or to combine it with its sibling, in the worst case the splitting or combining may advance back to the root. The locks can be released after the insertion or the deletion and the balancing have been finished. For searches, it is enough to keep at most two nodes locked simultaneously, because search processes use lock-coupling when they advance in the tree.

When top-down balancing is used, it is enough to keep at most two nodes locked simultaneously. The processes use lock-coupling when advancing, but if it is necessary to split a node or to combine it with its sibling, it is always possible to make the necessary modifications in the parent node without splitting or combining the parent. Thus, the balancing never advances upwards in the tree. However, the parent has to be kept locked until it has been checked whether the child needs a balancing operation.

When relaxed B-trees are used, at most two nodes have to be kept locked simultaneously, because lock-coupling is used. However, the lock of the parent can be immediately released after the child has been locked. We do not have to check balance conditions before releasing the lock of the parent as we have to do when using top-down balancing. The rebalancing operation split has to lock two nodes (the node with tag value -1 and its parent) and the operation compress has to lock three nodes (the node that is not full enough, its sibling and parent).

B^{link}-trees differ from other B-tree structures presented here so that lock-coupling is not used. When advancing in the tree, a process locks a node, finds a pointer to the next node and releases the lock of the node. After that the next node is locked. This is possible, because if another process changes the range of the next node between the locks, a new node can be found by using the highvalue and the next pointer of the node. Also, a writer uses r-locks when advancing in the tree. When the writer has found a leaf node it wants to modify, it releases the r-lock and after that takes an x-lock on the same node. If the key searched for is not in the range of this node any more (because another process has modified the node between), the process advances in the tree to the right by using next pointers and x-locks, but not lock-coupling. If the node has to be split, the nodes that have to be modified higher in the tree are x-locked one at a time. Because lock-coupling is not used, a node cannot be deleted as long as there are active processes which might read this node, see Section 2.3.

3.2 Transaction Model and Concurrency Control

On the database level, it is not enough that single actions are performed correctly in concurrent situations. For example, consider a situation where two people want to pay into the same bank account simultaneously. The updating of the bank database consists of reading the balance of the account to a variable, incrementing this variable by the sum paid and copying the new value of the variable to the database. If one of the processes is completely performed after the other process has read the balance but before it has written the updated value of the balance, the effects of the first process are totally lost. Although each of the read and write actions is performed correctly, the final result is incorrect.

To prevent anomalies like this, a set of actions is joined together to form a *transaction* [14]. The last command of a successful transaction is always *commit*. If it is not possible to finish a transaction successfully, it must be *aborted*. The commands commit and abort can only appear as the last command of a transaction. Each transaction is required to have four properties: atomicity, consistency, isolation and durability. Atomicity means that either the whole transaction is performed or none of its actions is done. Consistency means that the transaction preserves the consistency of the database. What this actually means depends on the application. Isolation means that although several transactions may be working concurrently, it seems to every transaction T that each of the other transactions is completely performed either before or after T. Durability means that the results of committed transactions do not disappear from the database, not even if a physical crash of the system occurs.

If several transactions are allowed to perform concurrently, some concurrency control method is needed to guarantee isolation. Isolation can be achieved if it is demanded that any possible execution of any set of transactions is serializable. This means that if we take any set of transactions and any possible schedule that implements this set, there is always a serial schedule of the same transactions so that each action returns the same value in both schedules. By a schedule we mean the order in which the elementary operations of the transactions are performed. A schedule is serial if for every pair of transactions all actions of one transaction are executed before any of the actions of the other in the pair.

Next, we want to define these ideas more formally. When constructing our model, we have used terms presented in [6, 58, 66].

A database consists of *data objects*. Each data object consists of a unique *key* and of a *value*. The possible *actions* for a data object x are Read(x), Insert(x), Write(x) and Delete(x). Read(x) means that the value of object x is assigned to a local variable. Write(x) means that a new value is assigned to the object x. Insert(x) means that the object x is inserted into the database and a value is assigned to it. Delete(x) means that the object x is deleted from a database.

A *transaction* T is defined to be a finite sequence of actions $(A_1, A_2, \ldots, A_m)$ where A_i is either Read, Write, Insert or Delete, when $1 \leq i \leq m-1$, and A_m is either Abort or Commit. Each action A_i $(1 \leq i \leq m-1)$ has associated with it a data object x_i. We require that the transaction T contains at most one of the actions Write, Insert and Delete for a certain data object x. Abort and Commit are special actions that have no data objects as arguments. Commit means that the effects of the transaction are made permanent in the database. Abort means the the effects of the transaction are cancelled.

We use the following symbols for actions belonging to transaction T_i: c_i denotes a Commit action, a_i denotes an Abort action, $w_i(x)$ denotes one of the actions Insert(x), Write(x) and Delete(x) and $r_i(x)$ denotes the action Read(x).

The data objects are accessed through an index. The index consists of *index objects* (e.g. nodes in a B-tree.) An *execution of an action* is a sequence of *index operations*. The group of the possible index operations varies according to the index structure used, but it usually contains operations like read node, write node, delete node, insert key to node and so on. A certain action may have several possible executions depending on how the action interacts with other actions, for example. An *execution of a transaction* is a sequence of executions of the actions belonging to that transaction.

A *transaction system* τ is a set $\{T_1, T_2, \ldots, T_d\}$ of transactions. A *history* H of τ is any shuffle of the actions of the transactions that belong to τ.

Next, we want to define when a history H of a transaction system τ is correct. We have chosen *view serializability* as our main correctness criteria. This is because we want a criteria that is also applicable to multiversion systems where the same data object may have several versions. The following definitions are from [6].

A history H of the transaction system τ is called *serial*, if H contains executions of all transactions of τ without any interleavings.

By writing $A_i \prec A_j$ we mean that A_i precedes A_j in the history we are considering. We say that T_j *reads* x *from* T_i in history H if

1. $w_i(x) \prec r_j(x)$
2. a_i does not precede $r_j(x)$ and
3. if there is some $w_k(x)$ such that $w_i(x) \prec w_k(x) \prec r_j(x)$, then $a_k \prec r_j(x)$.

A write operation $w_i(x)$ in history H is called a *final write* if $a_i \notin H$ and for any $w_j(x) \in H$, $j \neq i$, either $w_j(x) \prec w_i(x)$ or $a_j \in H$.

The *committed projection* of history H, denoted by $C(H)$, is the history obtained from H by deleting all actions that do not belong to some transaction T_i such that $c_i \in H$.

Two histories H and H' are *view equivalent*, if

1. they are over the same set of transactions and have the same actions.
2. for any T_i, T_j such that $a_i, a_j \notin H$ (hence $a_i, a_j \notin H'$) and for any x, if T_i reads x from T_j in H then T_i reads x from T_j in H'.

3. for each x, if $w_i(x)$ is the final write of x in H then it is also the final write of x in H'.

Definition 3.1 *Let H be a history of a transaction system $\tau = \{T_1, T_2, \ldots, T_d\}$. H is* VIEW SERIALIZABLE, *if for every prefix H' of H there exists a serial schedule H'_S such that the committed projection of H' and H'_S are view equivalent.*

Next, we consider concurrency control methods, which are used to achieve serializability when several transactions are performed simultaneously. They are usually divided into three classes: optimistic methods, timestamp methods and locking. The algorithms presented in this thesis use either locking or timestamps and we introduce only briefly the main ideas of optimistic methods.

In optimistic methods [31] transactions consist of three phases: a read phase, a validation phase and a write phase. In the read phase the transaction reads all data items it needs and makes changes into their local copies. In the validation phase the transaction locks briefly these data items and checks that the assumptions (e.g. certain values of the data items) it needed to make the changes are still valid. In the write phase it makes the changes to the actual database. If the validation does not succeed, the transaction is aborted and restarted.

In timestamp methods (see e.g. [6]) each transaction has a unique timestamp. Every data item x also has two timestamps: $RTM(x)$ is the highest timestamp of all transactions that have read x, and $WTM(x)$ is a timestamp of the last transaction that has written x. Assume a transaction has a timestamp TS. If the transaction wants to read the data item x, it is allowed only if $TS \geq WTM(x)$. The write is allowed only if $TS \geq WTM(x)$ and $TS \geq RTM(x)$. Otherwise, the transaction is aborted and restarted. Thus, the timestamps are used to assure that a data item is not read by a transaction that has a lower timestamp than the last transaction that has written its value and that a data item is not written by a transaction if its value is read or written by another transaction having a higher timestamp. So, it is always possible to find an equivalent serial order of the same transactions. This order is the same as the order of the timestamps.

In addition to this basic timestamp method there are a lot of variations. Some of the variations use multiversion objects. This means that the same data object may have several versions in the database. In Chapter 9 we present a differential indexing system where a multiversion timestamp based method is used for concurrency control.

It is possible to achieve serializability of transactions by using the same locks on nodes which are used to assure the validity of the search structure. However, in a general case where there are many simultaneous transactions having several actions, this usually leads to poor performance in concurrent situations, because then the transactions have to take long-lasting locks. In Chapter 6 we consider a special case where only one of the simultaneous transactions accesses more than one key and only node level locking is used.

Key-range locking is an alternative for locking whole nodes. It means that either a certain key value or a range of key values is locked. Usually, this is not enough to assure that the index structure (B-tree, for example) preserves its validity, and another system, a separate node level locking, for example, is needed for this purpose. Key-range locking usually allows more concurrency, because the node locks can be released as soon as the update to the node and possible balancing operations has been performed. Usually, a separate data structure, called a lock table, is used to maintain key-value locks. The lock table is often implemented by a hash table. Sippu and Soisalon-Soininen present in [59] a transaction model where the actual transactions are nested transactions and search structure modification operations are subtransactions which can commit even though the parent transaction aborts.

Mohan [46] presents a key-range locking method called ARIES/KVL. Mohan uses two lock types, latches and locks. Latches are used to lock whole nodes for short periods, and locks are used to lock certain key values. Controlling latches is much easier than controlling locks, because latches are used for whole nodes. Latches are used to preserve the physical consistency of the search structure, and locks are used to guarantee the serializability of the transactions.

The processes advance in the tree by using latch-coupling. The processes use shared latches higher in the tree, but insert and delete processes take an exclusive latch when they see that the next node is a leaf node. When a desired leaf has been reached, an attempt is made to lock a certain key value. If this succeeds immediately, the latch of the leaf is released. Otherwise, the latch is also released to avoid deadlock, but the version number of the leaf is saved. When it is possible to lock the key value, it is checked that the version number has not changed. If the version number has changed, the leaf is searched again or the process is restarted from an upper level of the tree depending on the situation.

The search operation takes an r-lock to the key value if the value is found in the leaf. If the requested key value is not found, the next key value is locked. In both cases the locks are held until the transaction is committed.

Insert and delete operations usually lock, in addition to the desired key value, the next key value in the leaf. This is done to avoid difficulties that may follow from the interaction of uncommitted transactions. The modes of locks used and their duration depend on the type of the index (primary vs. secondary) and possible earlier locks on the key by the same transaction. A detailed description can be found in [46].

If node splits or compressions are needed, they are made step by step. A special bit is set in the node whose balancing is still unfinished, e.g. a node has been split, but no router to the current node has yet been inserted into the parent. In addition, the balancing operation takes an exclusive latch on the whole tree. This latch prevents only such operations that want to take a latch on the whole tree themselves. Node latches are given normally. If another operation notices that the balancing in its current node is unfinished, this operation waits until it is admitted a shared latch on the whole tree and restarts the operation from the root or from another suitable node. So an operation that reaches a node whose balancing is still unfinished waits until the balancing has been done, but other operations are not prevented by the balancing.

Chapter 4

Batch Update in B-trees

In this chapter, we explain what is meant by batch update. We also present some applications of batch update and review relevant literature.

4.1 Definition of Batch Update

The usual maintenance actions for dictionaries are Member, Insert and Delete. Here we will also consider actions called *batch insert* and *batch delete*. By batch insertion we mean a sequence of insertions:

$$I(k_1, \ldots, k_n) = (Ik_1, Ik_2, \ldots, Ik_n),$$

$n \geq 1$, where Ik_i denotes the insertion of key k_i into the dictionary. Batch delete is defined correspondingly. More generally, a *batch update* means any sequence of single insertions or deletions. We assume that the dictionary is implemented as a B-tree.

There are several ways to implement a given batch insertion $I(k_1, \ldots, k_n)$. The simplistic one is to perform n insertions in the given order into the underlying B-tree. A much more efficient method is to sort the keys before insertion. Then all keys belonging to the same leaf node can be inserted without consulting any other leaf. Thus, when the B-tree or at least its leaf nodes are stored on disk, substantial savings may be obtained in disk I/O by keeping the path to the current leaf in the main memory. An obvious inefficiency remains, however, because the search path from the root to the leaf is traversed for each key. The algorithm BATCHINSERT given in Figure 4.1 is efficient in this respect because it performs the search one time, at most, for all keys going to the same leaf.

To give an impression of the number of disk accesses it is possible to save by using an efficient batch-update algorithm, we consider an example. Assume that the number of leaves to be updated in the B-tree is l and the size of the batch is ml keys distributed

```
algorithm BATCHINSERT
begin
    sort the batch insertion and store it in the list L[1], ..., L[n];
    p := root;
    i := 1;
    while i ≤ n do begin ;
        starting from p search for the leaf node having L[i] in its range
        pushing the nodes read into stack;
        leaf :=  the found leaf node;
        j :=  the largest index such that L[j] is in the range of leaf;
        insert L[i], ..., L[j] into leaf;
        if leaf contains more than 2k keys after insertion then begin
            split leaf and rebalance in the usual way popping nodes from stack;
            p :=  the node where splitting ends
        end
        else
            p := POP(stack);
        i := j + 1;
        while i ≤ n and L[i]  is not in the range of p do
            p := POP(stack);
    end
```

Figure 4.1: Algorithm for batch insertion.

evenly over these leaves. If the upper levels of the tree are in the main memory and only the leaf nodes are on the disk, $2ml$ disk accesses are needed if the update is done one at a time in a random order. (We assume that no leaf splits or compressions are necessary.) If an efficient batch-update algorithm is used, only $2l$ disk accesses are needed. Thus the savings are $2l(m-1)$ disk accesses, assuming that $m > 1$. If $m \leq 1$, it is profitable to use a special batch-update algorithm only if the batch is very unevenly distributed over these leaf nodes.

The scheme for the batch insertion given in Figure 4.1 is inefficient when there are several concurrent processes accessing the underlying B-tree. This is because the above algorithm actually means that no other process can access the tree during the batch insertion. Solutions for the concurrent environment can be obtained by applying B-trees that do not require immediate rebalancing, such as relaxed B-trees[50] or $\mathrm{B}^{\mathrm{link}}$-trees[40, 56]. Another possibility to achieve a reasonable degree of concurrency is to apply top-down balancing [8, 22, 49] of the B-tree while performing the batch insertion. Details of these solutions are presented in Chapter 6.

There are some papers that give batch-insertion algorithms similar to the algorithm of Figure 4.1 [1, 5, 10, 20, 27, 32, 34, 48, 60, 61, 63]. In [34, 63] the general idea of a two-level index with the principal and differential indexes is introduced. In this scheme, updates are collected in a differential index that resides in the main memory. When the differential index becomes too large, it will be merged with the principal index on disk. In [27] a new B-tree structure, called Y-tree, is introduced. The purpose of the structure is to make very fast batch inserts in data warehousing systems possible. A batch update algorithm for a multidimensional index structure is presented in [1, 5, 20]. In [10] a text database indexing system is considered, in which the batch insertion consists of all words of the text item to be inserted. In [48, 60, 61] batch-update-like algorithms are used for on-line indexing. On-line indexing means that a relation can be accessed and updated concurrently with the construction of a new index for this relation. All updates performed during the index construction are collected into a separate structure, which will be merged with the main index after the original relation has been indexed. In [32] a batch-insert algorithm for an index of the real-time database system is presented.

4.2 Applications of the Batch Update

4.2.1 Using Batch Update with Differential Files

A general use of the batch update is to defer the installation of single updates. The updates are stored in a relatively small storage area, called a differential file. The purpose here is to obtain more efficiency, because it is faster to perform an update into a small index in the main memory than into a B-tree located in the disk. If a timestamp based concurrency control method is used, the differential indexing method can also be used to fasten the checks needed by the concurrency control algorithm, see Chapter 9.

At certain time intervals the differential file and the database must be merged. The batch update can be used to do this merge efficiently. When searches are performed, it has to be taken into account that a part of the data items is located in the differential file and a part on the disk. Search actions first search for the desired data item in the differential file. If the data item is not found there, the main index on the disk is searched.

Lang and Driscoll present a batch insertion algorithm for merging the differential file with the B-tree [34]. The whole path to the current leaf is kept in the memory. Each node is in a buffer whose size is twice the node size. As long as the next key to be inserted belongs to the current node, the search is not started from an upper level. The keys are inserted into the buffer one at a time. If the buffer becomes full, half of its contents are removed and written as a node to the disk. In this case, a router and a pointer to the new node are inserted in the buffer containing the parent node. When the next key k from the differential file does not belong to the current leaf node, the remaining records in the buffer are written out to the disk (split into two halves if necessary). The path is backtracked up until a B-tree node is reached which includes k in its range. All nodes that are left behind are written out to the disk. Then a new path of B-tree nodes is found to the leaf level based on the next key, and the previous insertion process repeats. Concurrency control is not considered in [34].

Lang and Driscoll [34] also present some analysis of batch update in their paper. They analyze how many disk accesses are saved by using a batch update. It is assumed that the whole B-tree, except the path to the current node, is kept on disk. They also derive an analytic formula to calculate the optimum time interval to merge the differential file to the data base and compare it with the optimum interval to perform a total reorganization of the data base, if the batch update is not used.

Srivastava and Ramamoorthy present two other algorithms, called GUA1 and GUA2, to perform the batch update [63]. GUA1 works as follows: first the list of all keys in the update is divided into groups according to the routers in the root node. If there are n pointers in the root node, the list is divided into n groups. Each group is passed to the node pointed to by the pointer. This is done recursively in every node when descending the tree until the leaf node is reached. Then, the remaining keys are inserted in the leaf.

The algorithm GUA2 requires each B-tree index node to store a pointer to its right sibling. In this algorithm the leaf node where the first key belongs is first searched from the root. Then, a left-to-right scan is done at the lowest level, using the right sibling pointers, to insert the rest of the tuples. It is not explained in the paper how the balancing is done if a node has to be split. Srivastava and Ramamoorthy do not consider concurrency control either. They present analytical results about the savings in disk accesses achieved by using GUA1 compared to a situation where the updates are performed one at a time in a random order, and simulation results about saved disk accesses when using GUA2.

4.2.2 Batch Updates and Full-text Indexing

Cutting and Pedersen present a batch-update algorithm [10] to update a full-text index that is implemented as a B-tree inverted index. When the keys to be inserted are sorted in the main memory, the occurrence data belonging to the same key are combined. (Occurrence data means the application-dependent data used to describe in which document and where in the document the word occurred.) Keys are inserted into the B-tree with their occurrence data one at a time, but the path from the root to the current leaf is kept in the main memory and thus extra disk accesses are avoided. Cutting and Pedersen show that more disk accesses are avoided by sorting the keys and keeping the path to the current leaf in the main memory than by doing the updates in occurrence order and keeping the upper levels of the B-tree in the main memory. Cutting and Pedersen do not discuss concurrency control during the batch update.

Tatu Ylönen presents a concurrent full-text indexing algorithm in [67]. His batch-update algorithm uses top-down balancing. Again, when the keys to be inserted are sorted in the main memory, the occurrence data belonging to the same key are combined. The leaf where the first key should be inserted is searched for. All keys that belong to this leaf are inserted simultaneously. After that the search for the next key is usually started from the parent of the leaf, not from the root. A commercial text database system that uses this algorithm has also been implemented. This algorithm is described in detail in Chapter 7.

4.2.3 Data Warehouses and Multidimensional Index Structures

A data warehouse (see [33], e.g.) is a repository of integrated information from distributed, autonomous and possibly heterogeneous sources. Actually, the warehouse stores one or more materialized views of source data. It is usual that huge numbers of additions, for example millions per day, are made to a data warehouse. Thus, algorithms that perform updates to indices one at a time cannot be used and batch-update algorithms are necessary, if indices are used when implementing a data warehouse.

Jermaine et al. [27] present a special type of the B-tree, called *Y-tree*, to make very fast batch updates to an index possible. In a Y-tree, each pointer in an internal node has a *bucket* attached to it. The bucket contains key values that should be inserted to the subtree pointed to by this pointer. When keys are inserted in a batch, in each non-leaf node only one pointer is followed. Other key values to be inserted are inserted to the buckets in this node. Only the pointer with a bucket having the greatest number of keys in this node is followed to the next level and the keys in this bucket are inserted to the buckets in the node in the next level. This means that each batch insertion (containing typically hundreds of keys) reads and writes nodes on at most one path from the root to the leaf level in the tree. On the other hand, searches in the tree are slower than in a normal B-tree because the structure of the nodes is more complex and the size of the nodes is larger. In addition to searching for correct router-pointer pair in an internal node, the search process has to examine the corresponding bucket to see if the key searched for is there.

A multidimensional index structure is an alternative to the set of many one-dimensional secondary indices in data warehouses. Bayer [2] introduced the UB-tree which is a multidimensional index structure based on B-tree. The data space is partitioned and the order of the parts is determined according to a Z-curve [51]. Otherwise, the tree is implemented as a normal B-tree. In [20], a batch-update algorithm (called bulk loading in the paper) for UB-tree is presented, but without any information about concurrency control.

Another widely used multidimensional index structure is an R-tree [23]. In [1, 5] batch-update algorithms for the R-tree are introduced. In the algorithms, the data to be inserted are not sorted before insertion, but a special buffer tree is constructed. The data items to be inserted are at first inserted to the root of the buffer tree. When the number of data items in the root exceeds a specified limit, the data items are transferred to lower levels in the buffer tree. Finally, the data items are transferred to the R-tree. The algorithm in [5] can only be used in a situation where the batch insertion is performed into an originally empty R-tree and no other actions can be performed concurrently with the batch insertion. The algorithm presented in [1] can be used to perform batch updates into a non-empty R-tree and it allows concurrent search operations with batch update.

4.2.4 Batch Updates with On-Line Indexing

Another application of a batch update comes from on-line index construction. When it is noticed that a new index for a certain relation is needed, the easiest way to perform index construction is to do it in an off-line manner (i.e., by locking the relevant relation). This approach is not suitable, if the size of the data base is very large and the index construction takes too long time. Srinivasan, Carey [60] and Mohan [48] present a few on-line indexing algorithms and some of their algorithms use batch update. We present these algorithms in this section.

Srinivasan and Carey introduce several algorithms for on-line index construction [60]. All algorithms have either two or three phases: a scan phase, a build phase and possibly a catch-up phase. During the scan phase the relation is scanned and the keys are extracted and sorted. In some cases an intermediate index is built, but the index is not yet visible for other processes. The updates during the scan phase are gathered into a temporary list or index. During the build phase the scanned keys and the temporary list or index are combined to form the index. The temporary list may be sorted to make the build phase more efficient. If an intermediate index of the scanned keys has been made, a batch update is actually performed. However, no concurrency control is needed because the index is not yet visible for other processes. In addition to this, combining the indices is performed using standard B-tree algorithms, i.e. the batch updates are implemented by perfoming single insertions or deletions.

The algorithms are divided into X-algorithms and C-algorithms according to whether they allow other update transactions during the build phase. The X-algorithms allow no update transactions, and thus the index is ready and can be made available for all transactions after the build phase has been finished. C-algorithms allow other update transactions during the build phase. Modifications made by these update transactions are gathered into a list S. After the build phase the list S is sorted and inserted into the main index in a catch-up phase. During the catch-up phase the main index is visible to other update transactions and thus concurrency control is needed. Srinivasan and Carey use here standard B^{link}-tree algorithms. Because the paper does not give exact information about how the batch update is performed, it is probable that the batch update is implemented by perfoming single insertions or deletions.

Mohan introduces two on-line indexing algorithms [48]. One of them, called SF, builds the index using a side-file to gather insertions and deletions that occur while the indexing process is active. At the end, the side-file is processed to bring the index up to date. The other algorithm, NSF, does not use a side-file. While the indexing process is inserting keys in the index, transactions can be inserting and deleting keys from the same index tree. The NSF algorithm resembles a batch-update algorithm when building the index. First all data pages are scanned and the keys are extracted and sorted. Then the keys are inserted into the index. To make the insertion efficient, the index manager will accept multiple keys in a single call. Tree traversals are avoided most of the time by remembering the path from the root to the leaf and by exploiting that information during a subsequent call. However, the insertion does not begin before the whole relation has been scanned and all keys have been sorted. This is how NSF differs from a general batch-update.

The SF algorithm works as follows: the index-builder first builds the index tree bottom-up without any interference caused by direct key inserts or deletes in the index by transactions. If the index under construction is already visible to a transaction, the key inserts and deletes by the transaction for this index are appended to a side file. (The index becomes visible to a transaction when the index-builder has started to scan a record affected by the transaction. However, this does not mean that the transaction could already read the index. The term visible only means here that the transaction may perform write operations that affect the part of the index that has been constructed or is under construction.) After building the index, the index-builder processes the side-file from the beginning to the end. The keys in the side file are inserted into or deleted from the index. Until the index-builder reaches the last entry in the side-file, other transactions may still be appending new entries to the side-file. When all the entries in the side-file have been processed, the index is made available to other transactions. Thus, no concurrency control for the index is needed during the batch update. The side-file may be sorted to improve performance, but by the time the updates of the sorted entries to the index is completed, some more pages might have been added to the side-file. They are processed sequentially.

4.2.5 Batch Updates in Real-time Databases

In a real-time database system the transactions have deadlines on their completion times. Any deadline violation of a hard real-time transaction may result in disaster. Kuo et al. [32] present a real-time batch-update algorithm which is based on the batch-update algorithms presented in this thesis and on priority inheritance [57].

In the algorithm, the transactions are given priorities according to their deadlines. If there is a lock conflict and the priority of the lock-holding transaction T_i is lower than the priority of the lock-requesting transaction T_j, the priority of transaction T_i will be raised up to that of transaction T_j. The purpose of this is to speed up the performance of transaction T_i so that it can quickly release the requested lock.

4.2.6 Discussion

As we have seen, some papers about batch updates and B-trees have been published during the last 15 years. However, most of these papers do not consider concurrency control. Even in the papers where concurrency control is covered, there are no comparisons between different B-tree structures. Moreover, the tradeoff between the efficiency of a batch update and the allowed degree of concurrency has not been analyzed.

In this thesis, we present concurrent batch-update algorithms for different types of B-trees and we also compare their efficiency and the degree of concurrency they allow.

In [34, 63] the idea of a two-level index with the principal and differential indexes is introduced. In this thesis, we present a system based on differential indexing, which contains concurrency control both in transaction and index levels.

Chapter 5

Batch Insertion and Rebalancing without Concurrency

In this chapter, we present a batch-insertion algorithm for a situation where the sizes of subgroups which should be inserted into the same leaf are large and no other concurrent operations are active during group insertion. We present the algorithm for a more general type of multi-way trees usually called (a, b)-trees [25, 44], which were presented in Section 2.4. A B-tree is a special case of (a, b)-trees.

The basic idea of our algorithm is that in the search phase starting from the first unprocessed key all keys are determined that have the same *position leaf*, i.e., the leaf where the search of these keys ends. Then, an (a, b)-tree, called a *subgroup tree*, is constructed from this set of keys together with the keys already present in the position leaf. The next task is to "merge" the subgroup tree with the original tree, i.e., to partially cut the original tree at the position leaf and to lift the subgroup tree at the correct place. Finally, rebalancing is needed in order to get a valid (a, b)-tree.

Assume that a group insertion of m sorted keys is to be performed, and assume that the group is partitioned into k subgroups such that the keys in each subgroup have the same position leaf. The worst case time complexity of the algorithm (without the initial construction of subgroup trees in time $O(m)$) is

$$O(\Sigma_{i=1}^{k} \log m_i + |L|),$$

where m_i denotes the size of the ith subgroup and $|L|$ denotes the total number of nodes that appear in search paths from the tree root to the position leaves. The same expression also bounds the number of structure changing operations needed in the worst case for balancing the tree after the group insertion. Both results can be expressed by saying that the additional cost when compared to the case in which each subgroup contains only one key is at most $O(\Sigma_{i=1}^{k} \log m_i)$.

We have also studied the amortized rebalancing cost for a situation where single insertions and deletions mixed up with group insertions are performed into originally empty (a, b)-tree. Insertion and deletion have constant amortized rebalancing cost and the rebalancing cost of insertion of a subgroup is logarithmic in the size of the group.

Larsen [36, 37] has obtained similar results for the amortized cost of rebalancing after batch updates: If one subgroup is of size m, then the amortized rebalancing cost is $O(\log m)$. There are earlier papers [24, 43] presenting batch-update algorithms for red-black trees and AVL-trees which require the additional work $O(\Sigma_{i=1}^{k} \log^2 m_i)$ when compared to the case each subgroup has only one element. The result in [24] was received for red-black trees. Because it is possible to represent (a, b)-trees where $a = 2$ and $b = 4$ as red-black trees, the result of this chapter can be applied to red-black trees. However, the result in [43] was obtained for AVL-trees, and the result of this chapter does not directly carry over to AVL-trees.

5.1 Batch-Insertion Algorithm

We present an algorithm for inserting a sorted sequence $s_1, s_2, \ldots, s_m$ of keys into an (a, b)-tree T. We assume that the keys are given in ascending order. We also assume that all keys of the tree are stored in leaf nodes.

The batch insertion searches, from the unprocessed suffix of the input, for the maximal set of keys (including the first key of this unprocessed suffix) with the same position leaf, say l. On this set of keys, the keys originally in l are combined, and then an (a, b)-tree, denoted by B, is constructed. If this subgroup tree B consists of one leaf only, the leaf l in the original tree is replaced by B.

If B consists of more than one node, B is inserted into T so that the leaves of B are in the same depth as the other leaves of T. To perform this, the nodes in the path from leaf l towards the root are split up to the height h of B, see Figure 5.1. The split is done so that at every level the nodes of B can be put between the split nodes.

After this, the root of B is inserted as a new child of the node, denoted p, of height $h + 1$ in the path from the root of T to leaf l, see Figure 5.2. After the insertion, rebalancing operations may be needed. First, there may be nodes which were split before the insertion and which have less than a children after the split. Secondly, it is possible that there is not enough space in node p for inserting a new child. Thus, after B has been inserted, the split nodes directly to the left or to the right from B are traversed bottom-up, and for each node it is checked whether or not it has less than a children. If necessary, the node is compressed with the leftmost or the rightmost node of B at the same level, see Figure 5.3. The root of B may also have less than a children and it must be compressed with one of its siblings. After there are no nodes having too few children, it is checked whether or not there is enough space in node

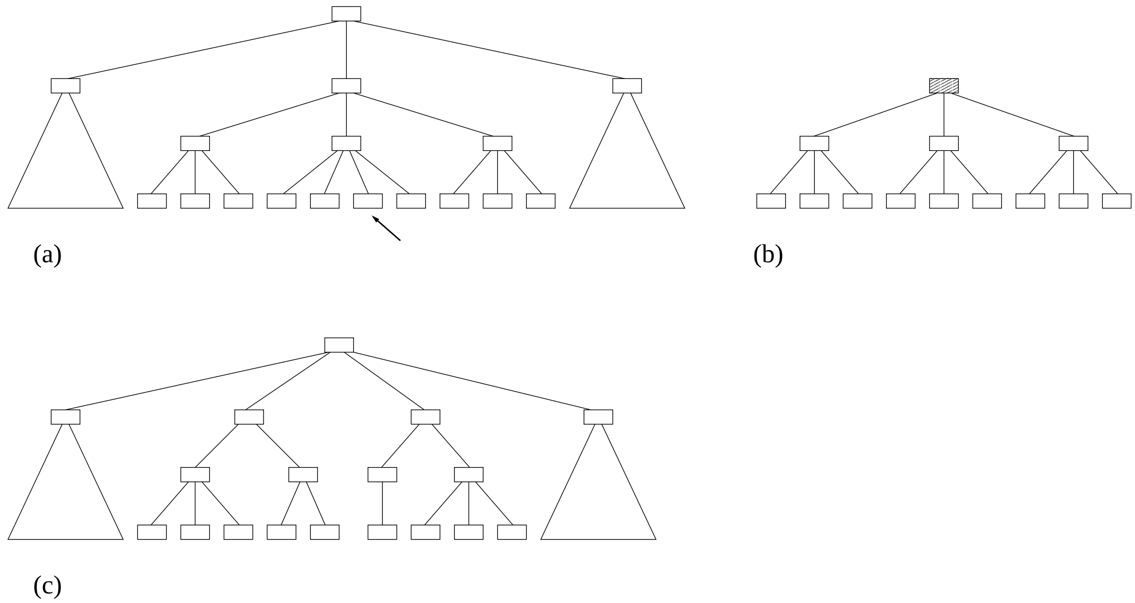


Figure 5.1: Splitting nodes to make space for a subgroup tree. $a = 2$ and $b = 4$. (a) The original tree. The leaf node where the group should be inserted is pointed by an arrow. (b) The subgroup tree. (c) The tree after the nodes have been split.

p for the pointers and routers to be inserted there. If there is not enough space, p is split and the splits are propagated as high as necessary.

Next, we give a more exact presentation of the algorithm. For node q, the term *range(q)* means the set of keys which belong to the subtree rooted by q or which would belong to this subtree if they were inserted into the tree.

Algorithm GI

Input: An ordered sequence $s_1, s_2, \ldots, s_m$ of keys and (a, b)-tree T into which the keys should be inserted.

Step 1. Let s_i be the first key to be inserted and q the root.

Step 2. Starting from q, search for the position leaf of s_i. Denote the leaf by l. The path from q to l is stored into stack.

Step 3. Let s_j be the last key which is in the range of l. Construct a subgroup tree B of keys $s_i, \ldots, s_j$ and the keys originally belonging to l. If B consists of only one leaf, replace l by B, denote the parent of l by t and go to Step 8. Otherwise, go to Step 4.

Step 4. Denote by h the height of B, and *cut* tree T starting from the parent of l up to the ancestor of l at height h as follows: At each level split the node such that the left part contains all pointers to children that store keys smaller than the smallest key

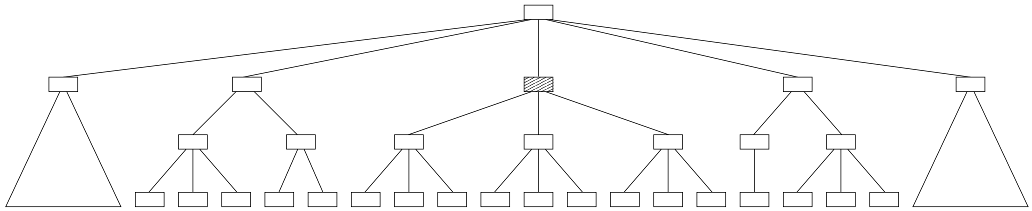


Figure 5.2: The tree of Figure 5.1 after the subgroup tree has been inserted. The root node has temporarily more than b children.

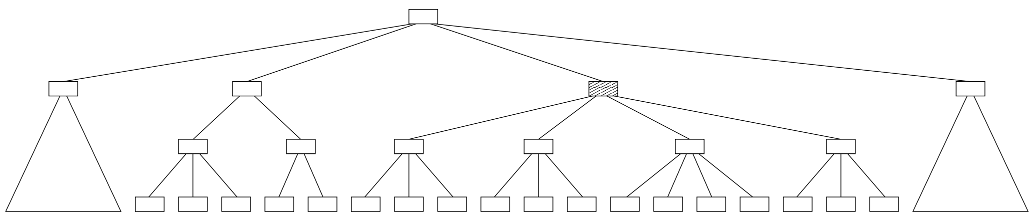


Figure 5.3: The tree of Figure 5.2 after rebalancing.

in B, and the right part pointers to children that store keys larger than the largest key in B, see Figure 5.1. Observe that at each level one of the parts may be empty.

Step 5. Denote by p the ancestor of l in T at height $h+1$, and by p' the ancestor at height h. Delete from p the pointer (and the corresponding router) to p', and insert into p the pointers (and the corresponding routers) to the left part of p' after the split, to the root of B, and to the right part of p'. If these pointers do not fit into p, let p be temporarily overfull, see Figure 5.2.

Step 6. Starting from the leaf level up to height h, compress the split left part with the leftmost node of B at the same height, and compress the split right part with the rightmost node of B at the same height, so that all nodes except possibly p fulfill the balance conditions of an (a, b)-tree, see Figures 5.3, 5.4, and 5.5. Notice that the router values before and after the pointer to the root of B have to be updated.

Step 7. Split p, if necessary, and continue splitting upwards as high as necessary. Denote by t the node where the balancing ends.

Step 8. If all keys have been inserted, stop. Otherwise let s_i be the first key which has not been inserted yet. If s_i is in the range of t, let q be t. Otherwise, advance upwards in T to the first ancestor of t which has s_i in its range. Denote this node by q. Go to Step 2.

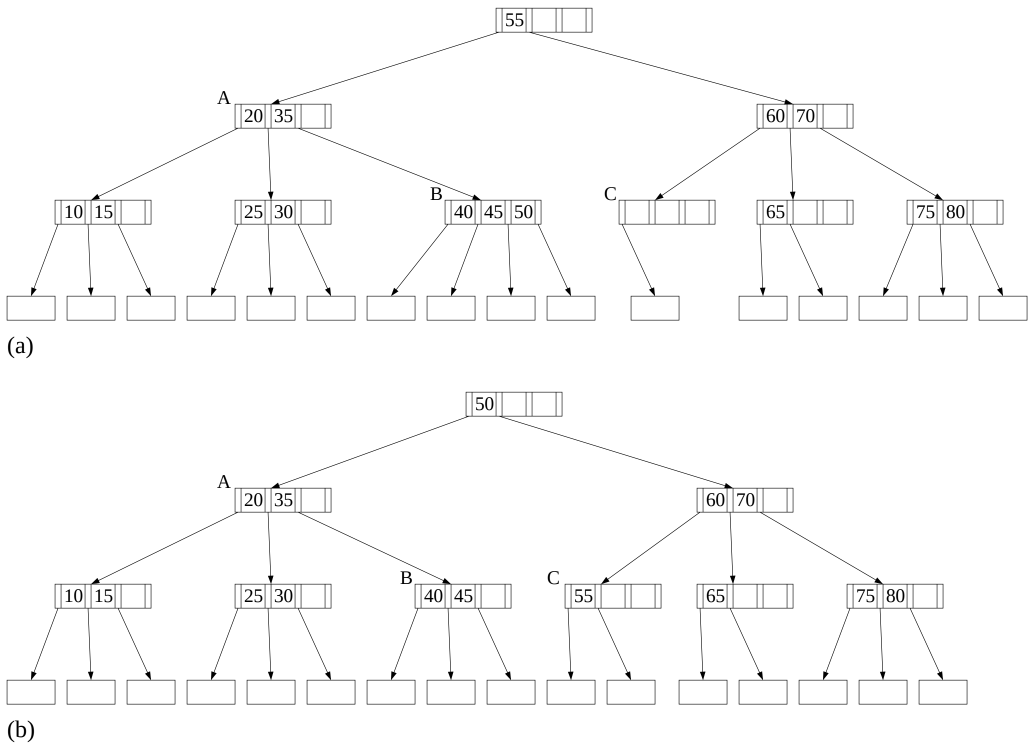


Figure 5.4: An example of how Step 6 of Algorithm GI works. A is the root of the subgroup tree, and B and C are the nodes which should be compressed. The contents of the leaf nodes are not shown. Diagram (a) shows the situation where the contents of B and C should be redivided. Diagram (b) shows the tree in (a) after the compress operation.

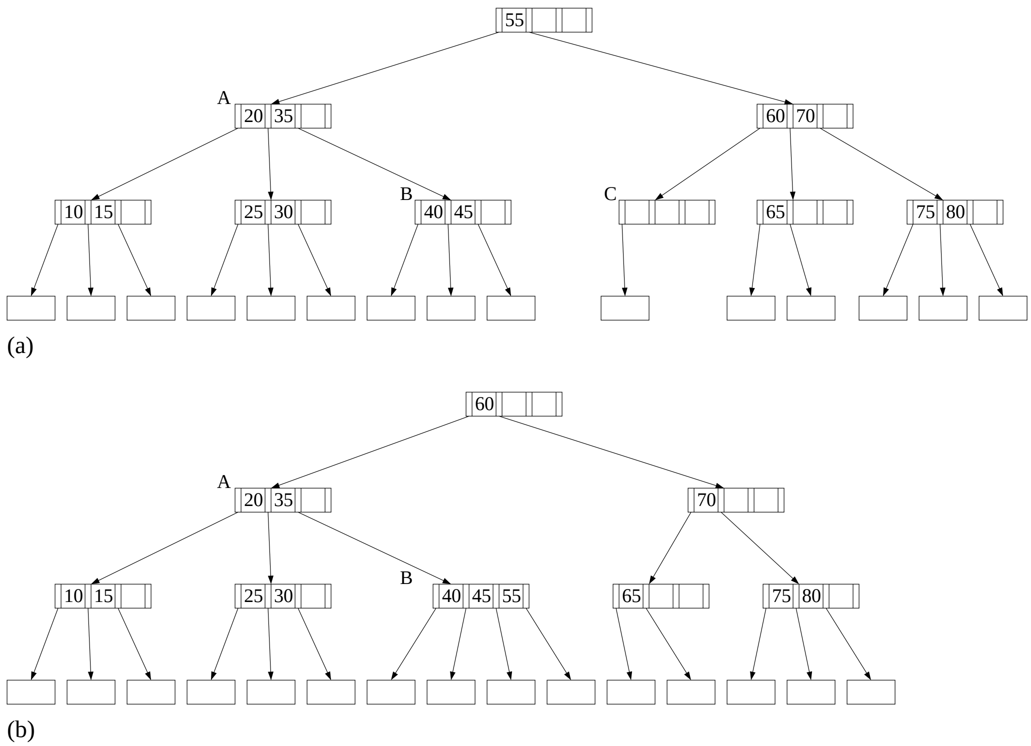


Figure 5.5: An example of how Step 6 of Algorithm GI works. A is the root of the subgroup tree, and B and C are the nodes which should be compressed. The contents of the leaf nodes are not shown. Diagram (a) shows the situation where the contents of B and C should be combined. Diagram (b) shows the tree in (a) after compress operations. Node C has been deleted.

5.2 Analysis

In this section, we analyze the time complexity of the algorithm GI and we also estimate the number of structure modification operations needed to rebalance an (a, b)-tree T when k subgroups, of sizes $m_i, 1 \leq i \leq k$, are inserted into T. By structure modification operations we mean the split and compress operations. By group-search path we mean the subtree of T whose root is the root of T and whose leaves are the position leaves of the subgroups.

Theorem 5.1 *Assume that the algorithm* GI *divides the input into k subgroups, and for $i = 1, \ldots, k$ the size of the ith subgroup is m_i. The number of structure modification operations needed in* GI *to rebalance the tree is $O(|L| + \Sigma_{i=1}^{k} \log m_i)$, where $|L|$ is the number of different nodes in the group-search path.*

Proof. When a group of size m_i is inserted, it means insertion of the subgroup tree of height $O(\log m_i)$. After insertion, there may be nodes containing less than a pointers on both sides of the subgroup tree. Thus, on both sides of the subgroup tree, at most $O(\log m_i)$ compress operations have to be performed in Step 6 of Algorithm GI. Together, this means $O(\log m_i)$ compress operations.

Denote by p the node which becomes the parent of the root of the subgroup tree. At most two new pointers are inserted into p. It is possible that there is not enough space in p for these pointers. In that case, p has to be split, and in the worst case, every node in the path from p to the root of T has to be split. However, when the next subgroup tree is inserted, if there are common nodes in the path from the parent node to the root, these nodes do not have to be split again. After the first split there is enough space for new insertions in the split node.

When several subgroups are inserted, it is possible that every node in the group-search path must be split. If we denote the group-search path by L and the number of nodes in it by $|L|$, this means $|L|$ possible split operations.

It is also possible that some nodes in the group-search path must be split more than once. Next we want to estimate the maximum number of these extra split operations.

When a (router, pointer) pair is inserted into a full node, the node must be split. After the split, the node contains at most $\frac{b}{2} + 1$ pointers, if b is even and $\frac{b+1}{2}$ pointers if b is odd. Thus, a new split is not needed before $\frac{b}{2}$ (b even) or $\frac{b+1}{2}$ (b odd) new insertions.

Together, if j insertions are made to a node, then the maximum number of extra splits needed is $\lfloor \frac{j}{\alpha} \rfloor$ where $\alpha = \frac{b}{2}$, if b is even, and $\alpha = \frac{b+1}{2}$, if b is odd. Each extra split implies the insertion to a parent node. Thus, the maximum number of extra splits in the next level is $\lfloor \frac{j}{\alpha^2} \rfloor$ and at height i $\lfloor \frac{j}{\alpha^i} \rfloor$. When the number of extra splits is calculated, all splits of the nodes which already have been split is taken into account.

For example, if node n is split to nodes n_1 and n_2 and later both n_1 and n_2 are split, this is counted as two extra splits.

When the batch-insertion is performed, all subgroup trees are not of the same height. We denote by k_i the number of subgroup trees of height i, $1 \leq i \leq h$, and $k = \Sigma_{i=1}^{h} k_i$. Each group of height i to be inserted causes at most 2 insertions at height $i+1$. Then, the number of extra splits at height 2 is at most $\frac{2k_1}{\alpha}$, at height 3 at most $\frac{2k_1}{\alpha^2} + \frac{2k_2}{\alpha}$ and at height $h+1$ at most $\frac{2k_1}{\alpha^h} + \frac{2k_2}{\alpha^{h-1}} + \ldots + \frac{2k_h}{\alpha}$. Let the splits advance to height $\gamma + 1$. Then, the number of splits at height $\gamma + 1$ is $\frac{2k_1}{\alpha^\gamma} + \frac{2k_2}{\alpha^{\gamma-1}} + \ldots + \frac{2k_h}{\alpha^{\gamma-h+1}}$.

The total number of extra splits is obtained by summing up all levels where splits are performed (notice that $\alpha \geq 2$):

$$\begin{array}{cc} & \Sigma_{i=1}^{\gamma} \frac{2k_1}{\alpha^i} + \Sigma_{i=1}^{\gamma-1} \frac{2k_2}{\alpha^i} + \ldots + \Sigma_{i=1}^{\gamma-h+1} \frac{2k_h}{\alpha^i} \\ \leq & \Sigma_{i=1}^{\gamma} \frac{2k_1}{\alpha^i} + \Sigma_{i=1}^{\gamma} \frac{2k_2}{\alpha^i} + \ldots + \Sigma_{i=1}^{\gamma} \frac{2k_h}{\alpha^i} \\ = & \Sigma_{i=1}^{\gamma} \frac{2k}{\alpha^i} \\ = & 2k \frac{\alpha^\gamma - 1}{\alpha^{\gamma+1} - \alpha^\gamma} \\ < & 2k \frac{1}{\alpha - 1} \\ \leq & 2k. \end{array}$$

There is still one possible source of extra splits which has not been taken into account yet. Suppose that when a certain subgroup tree is inserted, a node above the root of this subgroup tree has to be split. It is possible that the split node is compressed when another subgroup tree is inserted later. If a third subgroup tree is inserted below the compressed node, it may lead to an extra split in this node. However, the number of these extra splits cannot be larger than the number of compress operations, which is $O(\Sigma_{i=1}^{k} \log m_i)$.

By summing up the maximum number of all rebalancing operations, we obtain the bound

$$|L| + 2k + O(\Sigma_{i=1}^{k} \log m_i),$$

where $|L|$ is the number of nodes in the group-search path, k is the number of subgroups and m_i is the number of keys in group i. Thus, the maximum number of rebalancing operations is $O(|L| + \Sigma_{i=1}^{k} \log m_i)$. □

Next, we want to estimate the worst case time complexity of the batch-insertion algorithm GI. The complexity of the algorithm is bounded by the number of structure modification operations needed and the number of node visits during the group insertion. We use five lemmas to estimate the number of node visits.

Definition 5.1 *The* DISTANCE *of two nodes p and q in a B-tree is the length of the shortest path from node p to node q if the B-tree is considered as an undirected graph.*

Lemma 5.1 *If there is enough space for all keys to be inserted into the position leaves, the number of visits in nodes during group insertion is $O(|L|)$ where $|L|$ is the number of different nodes in the group-search path.*

Proof. If all subgroup trees consist of only one node, no new nodes are inserted into the B-tree during group insertion. Each internal node n in the group search path is visited at most $c+1$ times where c is the number of children of n that belong to the group search path. Thus, the number of visits in internal nodes is less than $2|L|$. In addition, each leaf node in the group search path is visited once. This means k visits in leaf nodes. Because $k < |L|$, the number of visits in nodes is $O(|L|)$.□

Lemma 5.2 *If a subgroup tree of height $h_i, i \geq 1$ is inserted into the B-tree and the number of node splits needed above the root of the subgroup tree after insertion is s_i, the number of visits in nodes after the position leaf is found until the balancing is finished, i.e. in Steps 4, 5, 6 and 7 of algorithm* GI, *is $O(h_i + s_i)$.*

Proof. When the subgroup tree of height h_i is inserted and the position leaf for the tree has been found, all nodes in the path from the position leaf to the root up to height h_i in the original B-tree have to be visited in Step 4 of algorithm GI. After the insertion, the split nodes in this path are visited again in Step 6 of the algorithm. Together, this means $O(h_i)$ node visits. It is also possible that the parent of the root of the subgroup tree has not enough space for the pointer to the root. In that case, the parent has to be split and it is possible that the splits advance higher. The number of node visits because of the splits in Step 7 of Algorithm GI is $O(s_i)$. Thus, the total maximum number of node visits is $O(h_i + s_i)$.□

Lemma 5.3 *Assume that the distance of two successive position leaves l and l' is d just before the subgroup tree B_i, whose position leaf is l, is inserted. If the subgroup tree has the height h_i and the number of split operations needed above the root of B_i after insertion is s_i, then at most $h_i + s_i + d$ nodes are visited after the balancing ends and before l' is reached, i.e. in Steps 8 and 2 of Algorithm* GI.

Proof. Denote by n the node where the balancing ended after the insertion of B_i. If l' is a descendant of n, the number of nodes to be visited is the height of n, which is at most $h_i + s_i$. If l' is not a descendant of n, then n must lie on the original path from l to l'. Because there are no new nodes above n in this path, the path from n to l' is not longer than the original path from l to l'. Thus, the number of nodes to be visited in Steps 8 and 2 of Algorithm GI is at most $h_i + s_i + d$.□

Lemma 5.4 *When a subgroup tree of height h_i has been inserted, the compress operations which are performed in the execution of Step 6 of Algorithm* GI *may increase the distance of two other successive position leaves by at most $2h_i$. In addition, each split above the subgroup tree may lead to at most two extra node visits later during the group insertion.*

Proof. After the insertion of the subgroup tree, the compress operations on the right-hand side of the tree may lead to a situation where an ancestor of a certain position leaf is combined with the right-most node of the subgroup tree in some level, but ancestors of the next position leaf are not combined with any node of this subgroup tree. In this case, the first position leaf becomes a descendant of the root of the subgroup tree, but the second position leaf does not. After the compress operation, the first common ancestor of these position leaves is the parent of the root of the subgroup tree. Thus, the distance of these two position leaves may be increased by at most $2h_i$. The increase does not affect the distances between other position leaves, because the group insertion advances from left to right and there is no need to return back to the left after the distance has been traversed once.

Next, we consider the number of node visits created by the splits above the inserted subgroup tree. Assume that l and l' are two successive position leaves and the distance between l and l' is d before the group insertion. Assume that after the insertion of the subgroup tree, a node n in the path from l to l' had to be split. We consider first the case where no nodes above n had to be split. If only one of l and l' was descendant of n before the split, then the distance between l and l' does not change after the split. If both l and l' were descendants of n before the split, and n were split to nodes n' and n'' such that l is the descendant of n' and l' the descendant of n'', then the distance between l and l' is increased by two (node n'' and its parent). Next, we consider the case where the split of n led to other splits in ancestors of n. Splits in nodes that are not common ancestors of both l and l' do not affect the distance between l and l'. Denote by c the number of common ancestors of l and l' which are split so that after the split one of the new nodes is an ancestor of l and another is an ancestor of l'. In that case, the distance between l and l' is increased by $c + 1$.

If the split of some node has increased the distance between l and l', it does not increase the distance between any other position leaves, because the group update advances from left to right. □

Lemma 5.5 *Assume that the algorithm* GI *divides the input into k subgroups, and for $i = 1, \ldots, k$ the size of the ith subgroup is m_i. The number of node visits during group insertion is $O(|L| + \Sigma_{i=1}^{k} \log m_i)$ where $|L|$ is the number of different nodes in the group-search path.*

Proof. We denote by s the total number of split operations performed during group insertion. By Lemma 5.1, the number of visits in nodes during group insertion is $O(|L|)$, if it is possible to perform all insertions directly to leaves and no structure modification operations are needed.

By Lemmas 5.2 and 5.3, the sum of the number of extra node visits caused by the insertions of all subgroup trees, performed after the position leaf for each subgroup tree has been found until the next position leaf is found, is $O(\Sigma_{i=1}^{k} \log m_i + s)$. By Lemma 5.4, the total number of extra node visits caused by an earlier insertion of another subgroup tree, when the batch-insertion advances from one position leaf to the next position leaf is $O(\Sigma_{i=1}^{k} \log m_i + s)$.

By Theorem 5.1, $s = O(|L| + \Sigma_{i=1}^{k} \log m_i)$. Thus, the total number of node visits performed during group insertion is $O(|L| + \Sigma_{i=1}^{k} \log m_i)$.□

Theorem 5.2 *Assume that the algorithm* GI *divides the input into k subgroups, and for $i = 1, \ldots, k$ the size of the ith subgroup is m_i. The worst case time complexity of the algorithm (except the construction of subgroup trees) is $O(|L| + \Sigma_{i=1}^{k} \log m_i)$ where $|L|$ is the number of different nodes in the group-search path.*

Proof. The complexity of the algorithm is bound by the number of structure modification operations needed and the number of node visits during the group insertion. By Theorem 5.1, the number of structure modification operations needed is $O(|L| + \Sigma_{i=1}^{k} \log m_i)$. By Lemma 5.5, the total number of visited nodes during group insertion is $O(|L| + \Sigma_{i=1}^{k} \log m_i)$. Thus, the total time complexity of the algorithm is $O(|L| + \Sigma_{i=1}^{k} \log m_i)$. □

Next, we want to estimate the amortized number of rebalancing operations needed when an arbitrary sequence of i insertions, d deletions, and insertions of k subgroups of sizes m_j, $1 \leq j \leq k$, is performed in an initially empty (a, b)-tree. When deletions are present, the best complexities cannot be obtained if $b = 2a - 1$ [44]. Thus, we consider only the case $b \geq 2a$.

We apply the potential function technique of [64]. For a node u, we denote by $c(u)$ the number of keys in u, if u is a leaf, and the number of children of u if u is an internal node. For the analysis, we assume that the insertion operation may leave the node temporarily overfull. If for node u, $c(u) = b + 1$ just after an insertion, a split operation is immediately performed. The parent of a split node may also have $b + 1$ children temporarily before it is split. Similarly, deletion may leave a node underfull to be corrected by the following compress operation. During group rebalancing a node may also be temporarily underfull.

We define the potential $\Phi(u)$ of a node u as follows, if u is not the root:

$$\Phi(u) = \begin{cases} 0, & a < c(u) < b \\ 1, & c(u) = a \\ 2, & c(u) = b \\ 3, & c(u) < a \\ 4, & c(u) > b \end{cases}$$

If u is the root, then $\Phi(u)$ is defined similarly, but a is substituted by the constant 2. The potential $\Phi(T)$ of an (a, b)-tree T is defined as the sum of the potentials of its nodes.

Theorem 5.3 *Let $b \geq 2a$ and $a \geq 2$. Consider an arbitrary sequence of i insertions, d deletions, and insertions of k subgroups of sizes m_j, $1 \leq j \leq k$, into an initially empty (a, b)-tree. Assume that the subgroups are inserted by using the algorithm* GI *and insertions and deletions are performed using standard algorithms. Then, the rebalancing phases of insertion and deletion have amortized constant complexity, and rebalancing after insertion of each subgroup is amortized logarithmic in the size of the group.*

Proof. We first show that each insertion and deletion operation increases the potential by at most a constant and the insertion of subgroup tree of height h increases the potential by at most $C_1 h + C_2$ where C_1 and C_2 are constants. Then, we show that each rebalancing operation decreases the potential of the tree by at least 1. By definition, the potential of the tree is at least 0. Thus, the number of rebalancing operations cannot be larger than the amount of potential inserted into the tree by insertion, deletion and group-insertion operations.

We consider the different operations and how they change the potential of the tree.

Insertion affects only the potential of the node where the key is inserted. It may increase the potential of this node by at most 2.

Deletion affects only the potential of the node where the key is deleted. It may increase the potential of this node by at most 2.

When a *group insertion* of a subgroup tree of height h is performed, h nodes are split before the subgroup tree is inserted. This may lead to at most $2h$ nodes which have less than a children. The total potential of those nodes is at most $6h$. We assume that the subgroup tree is constructed so that at most two nodes at each level contain exactly a or b keys or have exactly a or b children. This is possible since $a \geq 2$ and $b \geq 2a$, see [26]. All other nodes in the subgroup tree have potential 0. Thus the total potential of the subgroup tree is at most $4(h+1)$. After the insertion, the parent of the root of the subgroup tree may have at most $b + 2$ children. Thus the potential of the parent may be at most 4. Together, the insertion of the subgroup tree may increase the total potential of the tree by $10h + 8$. If the number of keys in the subgroup tree is m, then $h = O(\log m)$. Thus, the insertion of the subgroup tree containing m keys may increase the potential of the tree by $O(\log m)$.

Split affects the potential of the node to be split and its parent. We denote by q the node to be split and by p its parent. Before a split, $c(q)$ is either $b+1$ or $b+2$ and thus $\Phi(q) = 4$. After the split, the total potential of the two nodes resulting from the split is at most 1. Thus, the potential decrease is at most 3. Inserting a new router and a pointer into p may increase the potential of p, but only by 2. Thus, the total decrease is at least 1.

Compress can be performed by using either *sharing* or *fusing*. Sharing means that the contents of two nodes are equally redivided between those nodes. Fusing means that the contents of two nodes are combined into one of the nodes and the other node is removed. We denote by q' the node for which $c(q') < a$ and by q'' the node with which q' is compressed. The parent of q' is denoted by p'. After a single deletion, a node is compressed with its sibling, if it contains less than a keys. In the insertion of the subgroup tree, a node which was split in Step 4 of Algorithm GI and has less than a children is compressed with the left-most or right-most node of the subgroup tree at the same level. We assume that fusing is used if $c(q') + c(q'') \leq b - 1$. Otherwise, sharing is performed.

When sharing is performed, the number of children of p' does not change. It is enough to consider the change of the potential in nodes q' and q''. Before sharing, $\Phi(q') + \Phi(q'') \geq 3$, because $\Phi(q') = 3$. After sharing $\Phi(q') + \Phi(q'') \leq 2$, because $a \leq c(q') < b$ and $a \leq c(q'') < b$. Thus, the potential is decreased by at least 1.

Fusing is performed so that the contents of q' are transferred to q'' and q' is removed. In the operation, $\Phi(q'')$ does not increase, because $c(q'') < b$ after fusing. Removing q' decreases the potential of the tree by 3. Deleting the router and the pointer to q' from p' may increase the potential of p', but at most by 2. Thus, the potential of the tree is decreased by at least 1.

We have shown that each insertion and deletion increases the total potential of the (a, b)-tree by at most a constant and the group insertion of the subgroup tree containing m keys by at most $O(\log m)$. We have also shown that each rebalancing operation decreases the potential of the tree by at least 1. Thus, if i insertions, d deletions and insertions of k subgroup trees of sizes m_j, $1 \leq j \leq k$ are performed in an originally empty (a, b)-tree, then the total number of rebalancing operations is $O(i + d + \Sigma_{j=1}^{k} \log m_j)$. $\Box$

In the amortized analysis calculated in Theorem 5.3, the length of the group search path does not affect the number of rebalancing operations needed. This comes from the fact that the amortized analysis considers a situation where at first a number of keys are inserted into an originally empty (a, b)-tree and after that a group insertion is performed. Rebalancing operations performed in the group search path are paid by the insert operations that needed no rebalancing.

5.3 Related Work

In this chapter we have presented a batch-insertion algorithm for (a, b)-trees. In the algorithm a small (a, b)-tree, called a subgroup tree, is constructed together for each set of keys which should be inserted into the same leaf, and for the keys originally present in this leaf. Next, the original tree is partially cut and the subgroup tree is inserted into the correct place. Finally, rebalancing is performed to get a valid (a, b)-tree. Our algorithms for cutting the original tree and inserting the subgroup trees are related to the split and join algorithms presented in [45] for 2–3 trees.

In [36], Larsen presents a batch-insertion algorithm for relaxed (a, b)-trees [38, 39]. Larsen analyzes the number of rebalancing operations needed, when individual updates mixed with group insertions are performed. According to [36], only a constant number of rebalancing operations is needed after an insertion and a deletion in the amortized sense and the number of rebalancing operations needed after a group insertion of one subgroup is $O(\log m)$, where m is the size of the subgroup. Thus, Larsen achieves the same bound for amortized complexity as we do, but we also give the worst case bound for a situation where a group insertion is performed in an arbitrary tree.

The sizes of the subgroups to be inserted are smaller in Larsen's algorithm than in the algorithm presented in this chapter. In Larsen's algorithm, each subgroup is constructed from the keys that belong between two successive key values in some leaf. In our algorithm, all keys that should be inserted into the same leaf belong to the same subgroup.

On the other hand, in Larsen's algorithm the rebalancing consists of small local operations between which the search structure is valid (except balance conditions). Thus, in concurrent situations only a small number of nodes at a time are locked during rebalancing. In our algorithm, a larger part of the tree has to be locked. But, in contrast to our algorithm, Larsen's algorithm does not guarantee a logarithmic search time.

Chapter 6

Concurrency Control with Batch Updates

In this chapter we focus on the correctness issues of the concurrency control when batch updates are present. We present concurrent batch update algorithms for different types of B-trees and show their correctness. Here we consider only algorithms where the whole concurrency control is provided by locking B-tree nodes. In Chapter 9 we present an algorithm that uses separate transaction level locks in addition to B-tree node locks.

6.1 Framework for Correctness Proofs

Using the framework of Shasha and Goodman [58], we show that any schedule consisting of one batch-update transaction of the form

$$b(K) = (U(k_1), \ldots, U(k_n))$$

($U(k_i)$ denotes the insertion or deletion of k_i, $K = \{k_1, \ldots, k_n\}$) and several single-action transactions $a_1(x_1), \ldots, a_m(x_m)$ is serializable, if the batch-update algorithm used satisfies certain conditions given below. By serializability we mean view serializability, which was defined in Section 3.2.

We start with some definitions from [58] that are needed in the proof. Because we consider only B-trees instead of general search structures, we simplify some definitions. The elements that can be inserted into and deleted from a B-tree are called *keys*. We assume these keys are drawn from a possibly infinite set of keys called *KeySpace*. A *state* of a search structure consists of a five-tuple (N, E, *root*, *contents*, *edgeset*), where N is a set of nodes, E is a set of directed edges, and *root* is a distinguished member of N. *Contents* is a function from nodes to subsets of *KeySpace*. The *global contents* is the union of the contents in all the nodes. *Edgeset* is a function from edges to subsets of *KeySpace*.

Inset and *keyset* are functions from nodes to subsets of *KeySpace*. For a leaf node n, *keyset(n)* is those values that are either in n or would be in n if they were inserted into the B-tree. For a non-leaf node n, *keyset(n)* is empty. *Inset(n)* is the union of *keyset(m)* for all nodes m that belong to the subtree whose root is n.

A *good state* for a search structure satisfies the following conditions:

1. $\{keyset(n) \mid n \in N\}$ partitions *KeySpace*.

2. $\forall n \in N\ [contents(n) \subseteq keyset(n)]$.

The user can modify and get information from the search structure by performing *actions*. An action is implemented as a program, which consists of *operations*. An action a is called the *parent action* of an operation o, if o belongs to the the program implementing a.

A *computation* is a quadruple $(s,\ s',\ E,\ <)$ where s is the search structure state before the computation begins, s' is the state after the computation ends, E is the set of action executions occurring in the computation and $<$ is a partial order on the operations of E.

An operation o is atomic, if it can be guaranteed that no instruction outside o can modify any data accessed by any instruction of o while o is performed. A computation c is *well-formed* if all search structure operations in c are atomic.

In an execution of a single-key dictionary action $a(x)$, it is useful to distinguish one operation as the *decisive operation* for a. The decisive operation carries the semantics of its parent action. For example, the decisive operation of insert(x) inserts x to the node which has x in its keyset. A decisive operation of $a(x)$ is called *proper* if it is the physical operation after which the key x has been inserted into or deleted from a leaf node n such that x belongs to *keyset(n)* (insert and delete actions) or after which it is possible to decide whether the key really exists in the tree (member action). (For a more detailed definition, see [58].) For example, if the decisive operations of action insert(x) inserted x into a node which does not contain x in its keyset, it would be an improper decisive operation.

A batch-update action $b(X)$ has a set of decisive operations $o_1(X_1), \ldots, o_m(X_m)$ where each X_i is a set of keys belonging to some *keyset(n)*. Decisive operation $o_i(X_i)$ of $b(X)$ is proper if after $o_i(X_i)$ all keys in X_i have been inserted in or deleted from the corresponding node. The operations which are not decisive are called non-decisive operations. For example, searching for a router value in an internal node or a insertion of a (router, pointer) pair into an internal node after node split are non-decisive operations. Non-decisive operations do not change the global contents of the search structure and they do not influence the return value of their parent dictionary action.

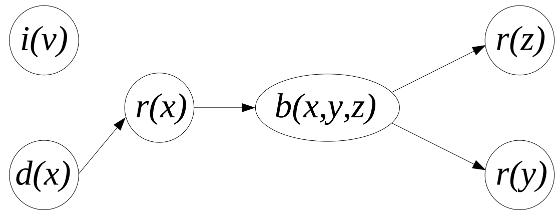


Figure 6.1: Example of a conflict graph when the set of actions is $\{d(x), i(v), b(x,y,z), r(x), r(y), r(z)\}$ and the order of decisive operations is $o(d(x)), o(r(x)), o_1(x,y), o(i(v)), o(r(y)), o_2(z), o(r(z))$ where $o(A)$ means the decisive operation of action A, $o_1(x,y)$ is the first and $o_2(z)$ the second decisive operation of the batch update action $b(x,y,z)$.

The interleaving of decisive operations from different dictionary actions of a computation c can be described by using a decisive operation *conflict graph*. This graph has a node for each action of c and an edge (a, a') if at least one of a or a' is an insert, delete or batch-update action and there is a decisive operation $o(X)$ or $o(x)$ of a and $o'(X')$ or $o'(x')$ of a' such that $o(X)$ or $o(x)$ precedes $o'(X')$ or $o'(x')$ and either $X \cap X' \neq \phi$, $\{x\} \cap X' \neq \phi$, $X \cap \{x'\} \neq \phi$ or $x = x'$, respectively. (X and X' refer to sets, x and x' to single values.) An example of a conflict graph is presented in Figure 6.1.

We consider a well-formed computation c that contains an arbitrary number of batch-update actions $b_1(X_1), \ldots, b_m(X_m)$ and single-key actions $a_1(x_1), \ldots, a_n(x_n)$. According to [58], c is a *generalized keyset computation*, if the following conditions KS1–KS5 hold:

KS1 The first state of c is a good state and each operation in c maps a good state to a good state.

KS2 The execution of every batch-update action $b_j(X)$ has a set of proper decisive operations $\{o_1(X_1), \ldots, o_k(X_k)\}$ in c such that $\{X_1, \ldots, X_k\}$ is a partition of X. The return value of $b_j(X)$ is the union of the return values of its decisive operations.

KS3 Each single-key action $a_i(x)$ has exactly one decisive operation and it is proper decisive.

KS4 Non-decisive operations in c do not change the global contents of the structure.

KS5 The conflict graph for c is acyclic.

It has been proved in [58] that if c is a generalized keyset computation, then c is serializable. (Actually the proof is given for computations having batch-insertion and batch-deletion actions, but because the proof does not depend on whether the actions in the batch are insertions or deletions, the result can be generalized for computations having also mixed batch-update actions.)

Lemma 6.1 *Consider a well-formed computation consisting of one batch-update action $b(X)$ and an arbitrary number of single-key actions $a_1(x_1), \ldots, a_n(x_n)$. If this computation satisfies the conditions KS1–KS4, it also satisfies the condition KS5.*

Proof. If any two decisive operations have the same key as an argument and at least one of the operations contains writing of a key value, then one of the operations ends before the other begins, because the computation is well-formed. Then it is possible to construct a conflict graph as follows:

1. There is one node for the batch-update action and one node for each single-key action in the graph.
2. There is an edge $(a_i(x), a_j(x))$ in the graph if at least one of $a_i(x)$ and $a_j(x)$ is an insert or delete action and the decisive operation of $a_i(x)$ precedes the decisive operation of $a_j(x)$.
3. The graph has an edge $(a_i(x), b(X))$, if the decisive operation of $a_i(x)$ precedes some decisive operation $o_j(X_j)$ of $b(X)$ and $x \in X_j$.
4. The graph has an edge $(b(X), a_i(x))$, if some decisive operation $o_j(X_j)$ of $b(X)$ precedes the decisive operation of $a_i(x)$ and $x \in X_j$.

The graph has only one node for an action having several keys as arguments. All other nodes describe single-key actions. Assume there were a loop in the graph. The loop would contain at most one node for batch-update action and one or more nodes for single-key actions $a_i(x)$ that all have the same key x as an argument. Each single-key action has only one decisive operation and the batch-update action has at most one decisive operation that has x as an argument. The loop would mean that there would be decisive operations o_i and o_j having the same argument such that o_i would precede o_j and o_j would precede o_i. This is in contradiction to the assumption that the computation is well-formed. Thus the decisive operation conflict graph for this computation is acyclic. □

If there are several concurrent batch updates, we cannot assure serializability in all cases. As an example consider the batch updates $b_1 = (I(k_1), I(k_2))$ and $b_2 = (D(k_1), D(k_2))$, i.e., b_1 inserts the keys k_1 and k_2, and b_2 deletes the same keys.

Consider the following order of decisive operations:

$$\begin{array}{llllll} b_1: & I(k_1) & & & I(k_2) \\ b_2: & & D(k_1) & D(k_2) & \end{array}$$

(horizontal axis is time). The result of this order is that key k_2 is in the dictionary but k_1 is not. Thus, this computation is not equivalent with either of the two serial computations.

In conclusion, we can show the serializability, that is, the correctness of a computation consisting of one batch-update action and several single-key actions. This is the case for which our algorithms are designed. This is also motivated by practical needs because, in practice, it is most important to be able to perform efficient searches during batch insertions. If efficient algorithms allowing more complex transactions are preferred, we need a separate transaction level concurrency control in addition to node locks.

6.2 Concurrent Batch Update Using Top-Down Balancing

The algorithm that was presented in Figure 4.1 is a description of a very efficient way to perform a batch insertion. It can be implemented in such a way that each edge of the underlying B-tree is traversed only once, unless rebalancing operations are needed. For efficient concurrency control, however, we should divide the batch insertion into smaller pieces so that other operations can access the B-tree between them. This means that some of the efficiency of the batch insertion will be lost, but reasonable efficiency is retained whenever at least all keys that should be inserted in the same leaf are inserted at the same time.

In this section we present an algorithm, called TD that inserts atomically all keys belonging to the leaves having the same parent. By "atomic" we mean that no other process can access this part of the tree during the insertion process, provided that there is enough space in the parent node for possible new routers and corresponding pointers. By top-down balancing, i.e. while advancing down the tree all full non-leaf nodes are split, we guarantee that at least one router will fit in the parent node. This batch insertion algorithm was originally introduced by Tatu Ylönen, who used it in his full-text indexing system [67].

The batch insertion algorithm TD is presented in Figure 6.2. We assume that the keys to be inserted are sorted and stored in the list $L[1], \ldots, L[n]$. The algorithm makes use of procedure SEARCH(*parent*, *leaf*, $L[i]$) which, starting from the *parent*, advances down the tree using the search key $L[i]$. When the leaf node in which $L[i]$ belongs is

```
algorithm TD
begin
    i := 1;
    parent := root;
    X_LOCK(parent);
    repeat
        SEARCH(parent, leaf, L[i]);
        /* Because of top-down balancing at least one new router will fit in parent */
        starting from L[i] insert keys into leaf until all keys belonging to leaf
        have been added or parent gets full because of leaf splitting;
        i := the index of the smallest key not yet inserted;
        RELEASE_X_LOCK(leaf);
        if (L[i] is in the range of parent and parent is full)
        or L[i] is not in the range of parent then begin
            RELEASE_X_LOCK(parent);
            parent := root;
            X_LOCK(parent);
        end
    until all keys have been inserted
end
```

Figure 6.2: Batch insertion when keys under the same parent node are inserted atomically. List L contains the keys to be inserted. Procedure SEARCH is presented in Figure 6.3.

reached, its parent node is stored in the variable *parent*. When advancing down the tree we apply the lock-coupling strategy of nodes with exclusive locks. Moreover, the top-down balancing strategy of B-trees is applied. When insertion takes place in a leaf, the leaf and its parent are locked. The parent is kept locked until the search is restarted from the root. The procedure SEARCH calls the procedure SEARCH_NEXT(*parent*, *key*) which returns the child of *parent* which has *key* in its *range*, i.e. *key* belongs to the subtree rooted by this child, if the tree contains *key*.

It is possible to perform a batch deletion in a similar way to the batch insertion. Only the balance condition to be checked is different. For each non-leaf node that is not the root, a check is made whether the node is full enough for at least one router and pointer to be removed from it. The deletion continues under the parent of a leaf node until all keys that should be deleted from the leaves having this parent have been deleted or the parent becomes so empty that it is not possible to remove a router from this parent any more. Then, the search is started from the root again.

```
procedure SEARCH(parent, node, key)
begin
    node := SEARCH_NEXT(parent, key);
    X_LOCK(node);
    while node is not a leaf node do begin
        if node contains 2k keys then begin
            split node;
            insert a router and a pointer to the new node into parent;
            if key belongs to the range of the new node then begin
                RELEASE_X_LOCK(node)
                node :=   the new node;
                X_LOCK(node);
            end;
        end;
        RELEASE_X_LOCK(parent);
        parent := node;
        node := SEARCH_NEXT(parent, key);
        X_LOCK(node);
    end
end;
```

Figure 6.3: Procedure SEARCH called by algorithm TD in Figure 6.2. The maximum number of children of an internal node is denoted by $2k$.

When a general batch update is done, we must check that each non-leaf node is neither too full nor too empty. Here we must relax the balance conditions presented in Section 2.1. Assume that the maximum number of pointers in a non-leaf node is $2k$ and denote by $c(u)$ the number of pointers in the node u. In the batch insertion, a non-leaf node u is split, if $c(u) = 2k$. In the batch deletion, a non-leaf node u, (not the root) is compressed with its sibling, if $c(u) \leq k$. If we compressed the node u having k pointers with its sibling when performing a general batch update, we might end with a node having $2k$ pointers. But this is too full for the batch update. So we cannot compress an internal node having k pointers, but we have to compress the node u if $c(u) \leq k - 1$. This means that after the batch update the B-tree may contain internal nodes having only $k - 1$ pointers. When a compress operation is performed, we must not construct an internal node containing $2k$ pointers. If the sum of pointers in the nodes to be compressed is $2k$, two nodes both containing k pointers are constructed. The balance condition for leaf nodes is not changed. A general batch update algorithm using top-down balancing, TD-UPDATE is presented

```
algorithm TD-UPDATE
begin
    i := 1;
    parent := root;
    X_LOCK(parent);
    repeat
        SEARCH2(parent, leaf, L[i]);
        starting from L[i] insert or delete keys into/from leaf until all keys belonging
        to leaf have been added/deleted or parent has either k − 1 or 2k children;
        i := the index of the smallest key not yet inserted;
        RELEASE_X_LOCK(leaf);
        if (L[i] is in the range of parent and parent has either k − 1 or 2k children)
        or L[i] is not in the range of parent then begin
            RELEASE_X_LOCK(parent);
            parent := root;
            X_LOCK(parent);
        end
    until all keys in L have been inserted or deleted
end
```

Figure 6.4: A general batch-update algorithm using top-down balancing. List L now contains the keys that have to be inserted and deleted. Each key is attached information about whether it should be deleted or inserted. The keys have been sorted. The maximum number of children of an internal node is denoted by $2k$.

in Figure 6.4. The procedure SEARCH2(*parent*, *leaf*, $L[i]$) is otherwise similar to the procedure SEARCH, but in addition to splitting full nodes the procedure compresses nodes having $k-1$ children.

The algorithms given in Figures 6.2 and 6.4 are efficient in the sense that the edges arriving at leaf nodes are traversed at most once. However, in a concurrent environment we have to keep the parent of the leaf node exclusively locked until all leaves under this node have been updated, which may take an unacceptably long period of time. In the next sections we present batch-update algorithms which allow more concurrency, but which may be less efficient than the algorithms presented in this section.

Next, we use the framework presented in Section 6.1 to prove the correctness of the computation consisting of a batch-update action TD and several single-key actions.

Theorem 6.1 *Any computation c which consists of a batch-update action* TD *and an arbitrary number of single-key actions is serializable, if all actions use lock-coupling technique as described in Section 3.1.*

Proof. We shall show that the conditions KS1–KS5 of Section 6.1 are satisfied. From this we obtain the theorem. First, each operation of c maps a good state to a good state. This comes from the fact that when any action splits or compresses a node, it also performs the necessary modifications in the parent to preserve the validity of the search structure. The node and its parent are kept exclusively locked until the modifications are ready. Because all actions use lock-coupling strategy, no action can get lost during these modifications.

Second, the operation which consists of instructions X_LOCK(*leaf*); insert keys $L[i], \ldots, L[j]$; RELEASE_X_LOCK(*leaf*); is a proper decisive operation for the batch insertion. It can be seen as follows: Every operation that reduces *inset(n)* for some node n exclusively locks both node n and its parent. Thus, *inset(parent)* and *inset(leaf)* cannot be reduced by another action while the parent is locked. This means that the keys $L[i], \ldots, L[j]$ belong to *keyset(leaf)* when the operation is performed and the operation is a proper decisive operation for the batch insertion action.

Third, using a similar argument as above, we can see that each single-key action $a(x)$ ends in a node n such that $x \in$ *keyset(n)*, which ensures that a decisive operation of $a(x)$ is proper. Fourth, there are no non-decisive operations that would change the global contents of the B-tree. By Lemma 6.1, since c satisfies the conditions KS1–KS4, it also satisfies the condition KS5.□

6.3 Concurrent Batch Update for Relaxed B-Trees

In this section we want to maximize the degree of overall concurrency in conjunction with batch updates. Basically this means that when a batch update is in progress, only the current leaf node is locked from other processes, not the parent of the leaf. This means, however, that the batch update itself becomes somewhat slower because we either have to search the correct leaf separately for each subgroup belonging to a certain leaf or the tree may be temporarily unbalanced.

Having this goal in mind, there are some suitable basic concurrency control strategies for B-trees, e.g., $\mathrm{B}^{\mathrm{link}}$-trees [40, 56, 35] or relaxed B-trees [50], that can be used. We present a batch-update algorithm using relaxed B-trees in this section and an algorithm for $\mathrm{B}^{\mathrm{link}}$-trees in the next section.

6.3.1 Algorithms

The first batch-insertion algorithm, called REL1, is presented in Figure 6.5. The procedure SEARCH_LEAF(*leaf*, $L[i]$) starts from the root and advances down the tree using the search key $L[i]$. When the leaf node in which $L[i]$ belongs is reached, it is stored in variable *leaf*. When advancing down the tree lock-coupling strategy with x-locks is used. When the leaf node is reached, the lock on its parent is immediately released.

Keys belonging to the *leaf* are inserted. If there is not enough space in the leaf to insert all keys, two new leaf nodes are created and the contents of the node are moved to them as explained in Section 2.2. When all keys belonging to this leaf have been inserted, the lock of the leaf is released and the search starts from the root again. During the insertion, it is enough to keep the leaf node locked, because in case the leaf is split, no modifications are made in the parent node.

It is also possible to modify the algorithm REL1 so that all keys belonging to the leaves having the same parent are inserted without restarting the search from the root node. Because of relaxed balance, it is enough to keep the parent r-locked. We call the version that keeps the parent of the current leaf node r-locked algorithm REL2 .

First, the leaf where the first key is to be inserted is searched for. When the leaf has been found and x-locked, the lock of its parent is not released, but changed to an r-lock instead. If the next key does not belong to the same node, the search of the new node is started from the parent (which is in the main memory already), not from the root. Only if the next key does not belong to the range of the parent, the lock of the parent is released and the search is started from the root. The case of overflow is handled as in the first algorithm.

The degree of concurrency obtained by this algorithm can be enhanced by using three kinds of locks, as explained in Section 3.1. When using this locking strategy, the batch-update process traverses down in the tree by using w-locks. When a leaf is found, its w-lock is changed to an x-lock and the w-lock of its parent is changed to an r-lock. Notice that the only process that is blocked because of the r-lock of the parent node is the rebalancing process. All other processes can advance in the tree in the same way as in the algorithm REL1.

This version of REL2 is presented in Figure 6.6. Starting from the root, the procedure SEARCH_W(*parent*, *leaf*, $L[i]$) searches the leaf node having $L[i]$ in its range and stores the leaf node into variable *leaf* and its parent into variable *parent*. Lock-coupling with w-locks is used. The procedure SEARCH_NEW_LEAF(*parent*, *leaf*, $L[i]$) starts from the parent and searches for the leaf where $L[i]$ belongs. The leaf found is stored in the variable *leaf*, but the value of *parent* is not changed. This procedure uses lock-coupling with w-locks under *parent*, which is left r-locked.

```
algorithm REL1
begin
    i := 1;
    repeat
        SEARCH_LEAF(leaf, L[i]);
        while L[i] belongs to leaf do begin
            starting from L[i] insert keys into leaf until leaf gets full or all keys
            belonging to leaf have been added;
            i :=  the index of the smallest key not yet inserted;
            if L[i] belongs to leaf then begin
                old_leaf := leaf;
                split leaf;
                leaf :=  one of the new nodes that has L[i] in its range;
                X_LOCK(leaf);
                RELEASE_X_LOCK(old_leaf);
            end
        end
    until all keys have been inserted
end
```

```
procedure SEARCH_LEAF(node, key)
begin
    node :=  root;
    X_LOCK(node);
    repeat
        parent := node;
        node := SEARCH_NEXT(parent, key);
        X_LOCK(node);
        RELEASE_X_LOCK(parent)
    until node is a leaf node;
end;
```

Figure 6.5: Batch-insertion algorithm REL1 using relaxed B-trees. List L contains the keys to be inserted. The degree of concurrency can be enhanced by using w-locks instead of x-locks when traversing down in the tree. In that case, when a leaf node is found, its lock is upgraded to an x-lock.

```
algorithm REL2
begin
    i := 1;
    repeat
        SEARCH_W(parent, leaf, L[i]);
        CHANGE_TO_X_LOCK(leaf);
        CHANGE_TO_R_LOCK(parent);
        while L[i] belongs to leaf do begin
            starting from L[i] insert keys into leaf until leaf gets full or all keys
            belonging to leaf have been added;
            i :=  the index of the smallest key not yet inserted;
            if L[i] belongs to leaf then begin
                old_leaf :=  leaf;
                split leaf;
                leaf :=  the new node that has L[i] in its range;
                X_LOCK(leaf);
                RELEASE_X_LOCK(old_leaf);
            end
            else if L[i] belongs to the range of parent then begin
                RELEASE_X_LOCK(leaf);
                SEARCH_NEW_LEAF(parent, leaf, L[i]);
                CHANGE_TO_X_LOCK(leaf);
            end
        end
        release all locks
    until all keys have been inserted
end
```

Figure 6.6: Batch-insertion algorithm REL2 using relaxed B-trees. List L contains the keys to be inserted.

```
procedure SEARCH_W(parent, node, key)
begin
    parent :=   root;
    W_LOCK(parent);
    node := SEARCH_NEXT(parent, key);
    W_LOCK(node);
    while node is not a leaf node do begin
        RELEASE_W_LOCK(parent);
        parent := node;
        node := SEARCH_NEXT(parent, key);
        W_LOCK(node);
    end
end;


procedure SEARCH_NEW_LEAF(parent, node, key)
begin
    /* parent is r-locked here */
    p := parent;
    node := SEARCH_NEXT(p, key);
    W_LOCK(node);
    while node is not a leaf node do begin
        if p ≠ parent then
            RELEASE_W_LOCK(p);
        p := node;
        node := SEARCH_NEXT(p, key);
        W_LOCK(node);
    end;
    if p ≠ parent  then
        RELEASE_W_LOCK(p);
end;
```

Figure 6.7: Procedures SEARCH_W and SEARCH_NEW_LEAF called by algorithm REL2.

It is very easy to construct algorithms like REL1 and REL2 for batch delete and for general batch update. In batch deletion the only difference is that if a leaf node remains too empty after deletion, nothing is done. In batch update, a full leaf node is split when a key should be inserted into it and nothing is done if a leaf node remains too empty. Everything else is done by a rebalancing process.

6.3.2 Rebalancing

The insertions and deletions performed by the batch-update algorithms REL1 and REL2 may leave the B-tree unbalanced. There may be nodes containing too few keys or pointers and nodes having tag value -1. The B-tree is rebalanced by a separate rebalancing process that uses operations split and compress presented in Section 2.2.

The main problem of rebalancing is how to find the unbalanced nodes. It would be possible to let rebalancing processes traverse the tree all the time searching for candidates. This would lead to many unnecessary disk reads, however. Most of these unnecessary operations can be avoided if another approach is chosen: the candidates for rebalancing are queued when they arise.

If a node must be split during an insert operation or it remains too empty after a delete operation, the node and its parent are put into a rebalancing queue. The lock on the node is released. The rebalancing process takes candidates from this queue, checks (after having x-locked both nodes) whether the rebalancing operation is still needed, performs the operation and releases the locks. If the parent node has to be split or remains too empty in the rebalancing operation, the parent node and its parent are inserted into the rebalancing queue. This means that the grandparent node has to be searched starting from the root and using lock-coupling with r-locks. The parent node is kept r-locked until the grandparent is found. When the nodes have been inserted into the rebalancing queue, all locks are released. If only two lock types (r- and x-locks) are in use, the lock of the parent has to be released if some node cannot be immediately locked when a grandparent is searched for. Otherwise, deadlock is possible.

Because the nodes in the rebalancing queue are not locked, it is possible that the parent-child relation is destroyed while the nodes are waiting for rebalancing. This is always checked by the rebalancing process. If the relation is not valid any more, the nodes are removed from the queue and no further actions take place.

When rebalancing operations are performed, two or three nodes must be x-locked. If some of the nodes cannot be locked because they are locked by another process, all locks taken by the rebalancing process are immediately released and the operations will be tried later again. In this way the rebalancing operations do not delay the search and update operations too much and deadlocks are avoided.

6.3.3 Correctness

Next, we use the framework presented in Section 6.1 to prove the correctness of the computation consisting of a batch-update action REL1 or REL2 and several single-key actions.

Theorem 6.2 *Any computation c that consists of one batch-update action* REL1 *or* REL2 *and an arbitrary number of single-key actions and rebalancing operations implemented by standard algorithms for relaxed B-trees is serializable.*

Proof. We shall show that conditions KS1–KS5 of Section 6.1 are satisfied. From this we obtain the theorem. First, the rebalancing processes in c perform only non-decisive operations into the search structure. If they split or compress nodes, they make the necessary modifications to preserve the validity of the structure. Thus, the good state is preserved.

Second, when relaxed balancing is used, the *inset(n)* for any node n of a B-tree may be decreased by compress operations only. (If the contents of two nodes are redivided between these two nodes in the compress operation, the *inset* of one of the nodes decreases.) In addition to that, rebalancing processes may delete some nodes. In both cases, the node which is deleted or whose inset decreases is exclusively locked as well as its parent. Thus, the inset of any leaf is not decreased by another action after the batch-update action has found the pointer to it, until the lock of this leaf has been released. This makes the operation consisting of X_LOCK(*leaf*); insert keys $L[i], \ldots, L[j]$ into *leaf*; RELEASE_X_LOCK(*leaf*); a proper decisive operation for the batch-insertion action.

Third, every single-key action $a(x)$ in relaxed B-trees uses lock-coupling. When it finds a pointer to a certain node, the inset of this node is not decreased by another action until it has released the lock on the node (see the argument above). This ensures that the action ends in node n such that $x \in$ *keyset(n)*. Thus, the decisive operation of every single-key action is a proper decisive operation. Fourth, there are no non-decisive operations in c that would change the global contents, i.e. the keys, in a relaxed B-tree. Fifth, since c satisfies the conditions KS1–KS4, it also satisfies the condition KS5 by Lemma 6.1. □

6.4 Concurrent Batch Update for B$^{\text{link}}$-Trees

The batch-insertion algorithm for the B$^{\text{link}}$-tree, called LINK is presented in Figure 6.8. The procedure MOVE_DOWN_AND_STACK(*leaf*, *key*) starts from the root and searches for the leaf having *key* in its range. The procedure uses r-locks, but not lock-coupling. The leaf found is stored in variable *leaf*. In addition, the last node visited at each non-leaf level is pushed into stack. The stack is used to locate the parent node if *leaf* has to be split and when a new leaf has to be found after the insertion into *leaf* has been finished. When a leaf node is reached, its r-lock is released. After that the process takes an x-lock on the node. It is checked whether the key still belongs to the leaf. (It is possible that another process has changed the leaf in the time interval the leaf was neither r-locked nor x-locked by the batch-insertion process.) If the key does not belong to the leaf, its x-lock is released and the process advances to the right with the procedure MOVERIGHT until the desired leaf is reached. Procedure MOVERIGHT always releases x-lock of a node, x-locks the right sibling of the node and checks whether the key belongs to the sibling node and so on until the desired node is reached.

To add desired keys into *leaf*, a temporary node, *temp_node*, is created in the main memory. The new keys and the keys of *leaf* are gradually merged into *temp_node*. When all keys belonging to *leaf* have been inserted or *temp_node* has grown too large, *temp_node* is copied into the disk (in the place of the old leaf node), the lock of *leaf* is released and the parent of *leaf* is popped from the stack. The search is continued from the parent, because in most cases the new leaf is a child of the same parent as the previous leaf. This search is done by the procedure SEARCH_LEAF_AND_STACK, which is similar to the procedure MOVE_DOWN_AND_STACK, except that the search is started from the parent instead of the root.

If *temp_node* becomes too full, it has to be split when it is copied to the disk. *Temp_node* is allowed to become at most twice as large as a normal node. Again, the parent of *temp_node* is located by popping a node from the stack. An x-lock is taken on the popped node. If this node is not the parent, it is searched by MOVERIGHT. The pointer-router pair is inserted into the node. If there is not enough space in the node, the process is continued by splitting the node and popping its parent from the stack. If there is no node in the stack (this may happen if another process has split the root), the search is started from the leftmost node at the next higher level. This can be done if we maintain an array of the leftmost nodes at each level. It is quite easy, because this array will be changed only when a number of levels in the B^{link}-tree is increased. The balancing process is continued as long as we have to split the node where the new router is inserted. When all routers have been inserted, the batch-insertion process is restarted from the root.

The batch-insertion algorithm presented here is nearly the same as the standard insertion for B^{link}-trees presented in [35, 40, 56]. The only differences are: (1) more than one key may be inserted into the current leaf simultaneously, (2) the temporary node is used to merge the old leaf node and the keys to be inserted and (3) the stack is used to find the parent of the current leaf and the search for the new leaf is usually started from the parent of the current leaf, not from the root. The standard insertion

```
algorithm LINK
begin
    i := 1;
    MOVE_DOWN_AND_STACK(leaf, L[i]);
    /* leaf is now x-locked */
    repeat
        MOVERIGHT(leaf, L[i]);
        /* leaf is now x-locked */
        copy leaf to temp_node;
        starting form L[i] merge keys to temp_node until temp_node becomes too
        large or all keys to be inserted into leaf have been inserted;
        i := the index of the smallest key not yet inserted;
        if size of temp_node ≤ 2k then begin
            write temp_node to disk (in the place of leaf);
            RELEASE_X_LOCK(leaf);
            if there are keys to be inserted left then
                SEARCH_LEAF_AND_STACK(leaf, L[i]);
        end;
        else begin
            split leaf and transfer the contents of temp_node into the two nodes;
            RELEASE_X_LOCK(leaf);
            add needed routers and pointers into upper levels;
            if there are keys to be inserted left
                MOVE_DOWN_AND_STACK(leaf,L[i]);
        end
    until all keys have been inserted
end
```

Figure 6.8: Batch-insertion algorithm LINK for B^{link}-trees. List L contains the keys to be inserted. The maximum number of keys in a leaf node is denoted by $2k$.

also uses the stack, but only to find the parent node when the current node has to be split.

The batch deletion and the general batch update in B^{link}-trees can be performed in nearly the same way as batch insertion. The only difference is that if a node remains too empty after deletion, the tree has to be compressed. The simplest way to do this is to have a separate process to do compressing, as proposed by Sagiv [56]. The compressing process can either traverse each level of the tree searching for nodes that are too empty, or deleters can put the candidate nodes into a queue for the compressing process.

In both cases the single compression is performed as follows: a pair of nodes, say A and B is examined. Assume that the maximum number of keys or router-pointer pairs in one node is $2k$. If A and B have together $2k$ or fewer pairs, then all data is moved to one of them and the other is deleted. If one of them has fewer than k pairs but together they have more than $2k$ pairs, then the data is redistributed between them so that each one will have at least k pairs. Deleted nodes cannot be released immediately, because there may be other processes that have found a pointer to them before deletion but have not yet read them. Thus, a deleted node can be removed when all processes that have started before its deletion have been finished.

The algorithm LINK starts the search for a new leaf from the parent of the current leaf, when all keys belonging to the current leaf have been inserted. Another possibility would be to follow the level link to the next leaf and not to consult the parent. This is profitable, if a typical batch update contains updates to nearly every leaf. However, there are many applications that typically make several updates to the same leaf, but leave many leaves without an update. Following leaf-level links could then lead to too many unnecessary disk reads.

In the original algorithms for B^{link}-trees [40, 56] only writers used locks, and it was assumed that the whole B-tree would be kept on the disk and an entire node could be read or written in one indivisible operation. Each process would read the node from the disk and process its own local copy of the node in the main memory. If the process made changes in the node, it would write the node back to the disk. Thus, only writers had to exclude each other, but the readers could always read an older version of the node from the disk although another operation would process its own copy of the same node in the main memory.

However, it is not always reasonable to assume the availability of atomic node reads or writes. It is usually profitable to keep part of the B-tree, for example the two upper levels, in the main memory. This means that the readers cannot be allowed to attach a certain node while a writer is making changes in the same node. That is why we have used a more general locking scheme, as in [35]. It is simple to convert the algorithms to fit the [40, 56] model when desired.

Next, we use the framework presented in Section 6.1 to prove a correctness of the computation consisting of a batch-update action LINK and several single-key actions. We give the proof only for situations where no $\mathrm{B}^{\mathrm{link}}$-tree nodes are deleted even if they have fewer than k children or keys. If node deletions were assumed, we should give the exact rules when the deleted nodes can be actually removed. Because $\mathrm{B}^{\mathrm{link}}$-tree algorithms do not use lock-coupling, the algorithms cannot assure that a deleted node will not be accessed. Additional rules are needed for this purpose, for example a deleted node can be removed when every process that started before the parent of the deleted node was updated has been finished.

Theorem 6.3 *Any computation c which consists of a batch-update action* LINK *and an arbitrary number of single-key actions implemented by standard* $\mathrm{B}^{\mathrm{link}}$*-tree algorithms is serializable, if the nodes of the* $\mathrm{B}^{\mathrm{link}}$*-tree are not compressed or deleted.*

Proof. We shall show that the conditions KS1–KS5 of Section 6.1 are satisfied. From this we obtain the theorem. First, the only operation that modifies the keyset of any node is splitting a leaf node. If a leaf node n is split and the new nodes are denoted by n_1 and n_2, then $\mathit{keyset}(n) = \mathit{keyset}(n_1) \cup \mathit{keyset}(n_2)$ and $\mathit{keyset}(n_1) \cap \mathit{keyset}(n_2) = \phi$. Thus, if the first condition of good state is satisfied before the split, it is also satisfied after the split.

Second, for any node n, no operation reduces $\mathit{inset}(n)$. If an action has decided that key k belongs to $\mathit{inset}(n)$ then k is never removed from $\mathit{inset}(n)$. When any action searches for a node where k belongs to or should be inserted to, it proceeds from node n_1 to node n_2 such that $k \in \mathit{inset}(n_1)$ and $k \in \mathit{inset}(n_2)$. The search is finished when a leaf node n having high value larger or equal to k is encountered. Because then $k \in \mathit{inset}(n)$ and $k \leq \mathit{highvalue}(n)$, $k \in \mathit{keyset}(n)$. When it is checked that $k \leq \mathit{highvalue}(n)$, the node n is kept exclusively locked during the check and until the key k has been possibly inserted or deleted. Thus, $\mathit{keyset}(n)$ is not changed before the actual insertion or deletion and the second condition of good state is preserved.

Third, the operation consisting of instructions X_LOCK(n); insert keys $L[i], \ldots, L[j]$ to n; RELEASE_X_LOCK(n); is the proper decisive operation for the batch insertion. This follows from the fact that the keys to be inserted belong to $\mathit{keyset}(n)$ (see the argumentation above). Similarly, each single-key action $a(x)$ ends in a node n such that $x \in \mathit{keyset}(n)$, which ensures that a decisive operation of $a(x)$ is proper.

Fourth, there are no nondecisive operations which change the global contents (the keys) in the $\mathrm{B}^{\mathrm{link}}$-tree. By Lemma 6.1, since c satisfies conditions KS1–KS4, it also satisfies the condition KS5.$\square$

Chapter 7

Application: Full-Text Indexing

An important application, in which a large number of keys can be inserted at the same time into a B-tree dictionary, is full-text indexing. Full-text retrieval is an information retrieval strategy where the user can search for information using any words of the original documents as search keys, instead of using only some predetermined key words. The search engines for web pages have greatly increased the importance of full-text indexing. To be able to implement full-text search efficiently, some kind of structure is needed to find documents containing the words searched for fast. The most important methods used are signature files [18] and inversion of terms [7, 19].

The signature file method is a technique that encodes the textual documents using hashing. The code is used to find documents that contain the word searched for. Usually, a word signature is created by setting m bit positions in an F bits long bit vector (m and F are design parameters) using a hash function. A document signature is an OR-ing of all word signatures in the document, see Figure 7.1. Searching documents containing a certain word is started by calculating the signature of the word. Then, the document signatures are scanned to eliminate non-qualified ones. The remaining documents are examined entirely to find out if they indeed contain the word searched for.

Traditionally it has been believed (see [16], e.g.) that when using signature files, only a small space overhead is needed and it is very easy and fast to insert a new document into the data base. On the other hand, searching for qualifying documents is usually slow. The search times can be enhanced by arranging the signatures in the disk so that it is not necessary to scan them all, for example by storing the signatures bit-wise, i.e. by clustering each bit position of all signatures. However, an ineffiency of searches remains when comparing signature file methods with inverted index methods. Zobel et al. [68] also shows that the claims about small space overhead and fast insertion do not hold for today's large text databases.

Word	Signature
free	001 000 110 010
text	000 010 101 001
Document signature	001 010 111 011

Figure 7.1: Example of a signature of a document containing only the words free and text.

When compared with signature files, inverted indexes have the advantage of very fast searching time. On the other hand, the space overhead of the index is often quite high, because the size of the index may be nearly as large as the actual database. However, it is possible to decrease the space overhead considerably by using efficient compression techniques [65]. It is also slow to insert a new document into an inverted index file, but a tolerable indexing performance can be achieved by using batch update.

In the inversion of terms the idea is to list the words occurring in the document, and for each word to list the places where it occurs. The list of words can be organized using any method, such as B-trees or hashing. We describe below a method that uses B-trees. It was originally constructed by Tatu Ylönen for his commercial full-text indexing system [67] that uses batch insertion with top-down balancing.

The index file is a B-tree, whose keys correspond to the words in documents. With each key, some occurrence data is stored which refers to application-dependent data used to describe in which document and where in the document the word occurred. Some occurrence data is stored with the key; if a key has very many occurrences, it may have a list of continuation records containing more occurrence data outside the leaf. The structure of the index is presented in Figure 7.2.

The indexing can be performed with any of the batch-update algorithms described in Chapter 6. In the first phase, keys and occurrence data are collected in the main memory. The keys are sorted, and each key is stored only once. Occurrence data for multiple occurrences of a key is combined.

When the size of the structure in the memory reaches a specified limit, the keys and occurrences in memory are merged into the B-tree on the disk using a batch-insertion algorithm.

If the key being inserted already exists in the leaf node, new occurrence data will be appended to the data already in the node. If the key does not exist, the key and its occurrence data are added to the node. If the key has very much occurrence data in the node, some of it is moved to continuation records outside the node.

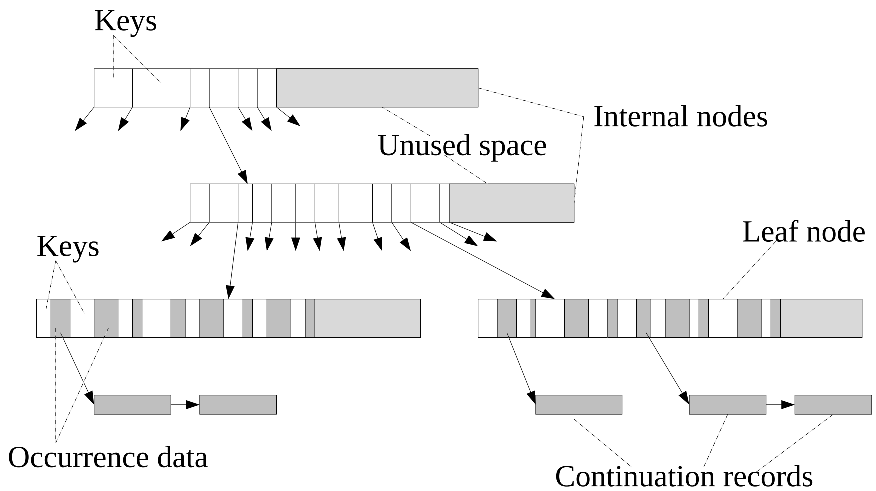


Figure 7.2: Structure of an inverted index file.

The commercial system by Tatu Ylönen [67] performs the indexing by using top-down balancing. The problem here is that the exclusive locking of the parent of the leaf node is quite bad in highly concurrent situations. A text database is an application where the size of the index grows very slowly after the first 10 MB of text has been indexed. Experience from commercial use suggest that the B-trees have usually only three levels and that there are about three nodes in the middle level. Locking the parent of the leaf node means that one third of the whole tree is locked. This problem can be avoided when using relaxed B-trees or B^{link}-trees and the batch-update algorithms described in Sections 6.3 and 6.4. In these algorithms, the middle node is either not locked at all or it is locked only in shared mode during most of the time span of the batch-update process.

Markku Rossi, in his Masters's Thesis [54], implemented a prototype for a concurrent full-text database system using the batch update algorithm for B^{link}-trees presented in this thesis. Rossi's implementation uses transaction level locks in addition to locking nodes and allows several concurrent batch update processes.

Chapter 8

Experimental Results

In this chapter we present experimental results about the behavior of the batch-update algorithms introduced in Chapter 6. In the experiments our goal was to investigate two things. At first, we wanted to know how efficient the algorithms are when the batch update is performed alone. Secondly, we wanted to know how much the batch update delays other processes, especially searches which are performed concurrently with the batch update. The purpose of the experiments was not to obtain exact numerical data of the performance of a certain algorithm, but to compare the algorithms.

Some comparisons on B-tree concurrency control algorithms can be found in the literature. Srinivasan and Carey [62] present a large experimental comparison of B-tree concurrency control algorithms. However, their research does not include relaxed B-trees and their comparison does not cover batch operations.

An alternative to experimental comparison is to compare the concurrency control algorithms analytically. This approach has been chosen in [28, 29] where Johnson and Shasha analyze the performance of B-tree concurrency control methods by using queuing theory. Unfortunately, the method used in [28, 29] is not directly applicable when batch updates are present. The analysis by Johnson and Shasha is based on the assumption that the processes using a B-tree can be modelled as Poisson processes. In addition to this, they calculate the maximum possible arrival rate of processes by finding the situation where the queues of processes waiting for locks on each node are all stable. When we want to analyze the concurrency during the batch-update process, the situation is rather different. There is one special process (batch-update process) that cannot be modelled as a Poisson process. The batch-update process traverses the whole B-tree and the performance of the other processes heavily depends on whether they encounter the batch-update process or not. The queues of processes waiting for locks on each node also vary depending on the current position of the batch-update process and there is no steady state for the queues.

In this chapter we first present the simulation model used in the experiments. Then, we present the results of the simulations for B-trees containing simple integers as keys and finally the results of the simulations for full-text-indexing systems.

8.1 Simulation Model

We wrote a simulation program to compare experimentally the batch-insertion algorithm using top-down balancing with the algorithms using relaxed B-trees and B^{link}-trees. In the simulation model used, each process has its own processor. The index structure is in the disk memory that is common to all processes. However, we assumed that the root of the index would be in the main memory which is common to all processors, but that no buffering would be used for other nodes. We assumed infinite disk resources, i.e., the processes had zero disk access waiting time. It was assumed that all disk writes were synchronous. When we simulated the algorithms REL1 and REL2 , we used versions that use three types of locks. The main parameters used in the simulation are presented in Table 8.1. For simplicity, we assumed constant times for all operations that usually take varying amounts of time.

8.2 Experiments for Numerical Data

The first simulations were performed on the three-level B-trees having integers as their keys. Batch inserts of different sizes were performed into a B-tree that had originally 60 000 keys. These keys were random integers between 0 and 400 000. The keys in the batches were also random integers from the same interval. Two different node sizes, 100 keys and 200 keys were used to show how the number of nodes in the middle level affected the results. When the node size 100 was used, the original B-tree had 12 nodes in the middle level; when the node size 200 was used, the original B-tree had 3 nodes in the middle level. To demostrate the effect of the original tree on batch-insertion times, the simulations were performed using two different B-trees of each node size.

Locking time	0.05 ms
Time to search for a key in a node	0.5 ms
Batch insert time for one leaf node	2.5 ms
Node split time	1.5 ms
Time to transfer a temporary node to its parent in a relaxed B-tree	1.5 ms
Disk access time	10 ms

Table 8.1: Main parameters used in the simulation.

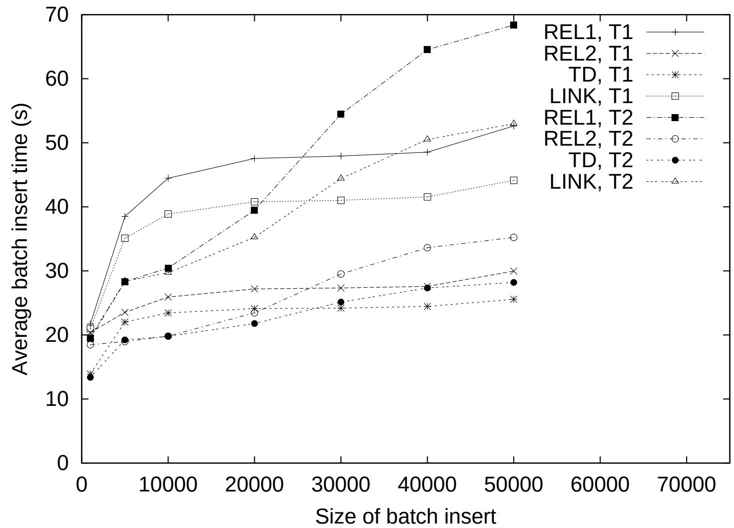


Figure 8.1: Average batch-insertion times for different algorithms. Node size was 100.

These trees were generated by using different sets of keys as the original keys, which had an effect on the number of the nodes and on the filling factor of the nodes. The trees are denoted by T1 and T2 in the tables and figures.

First the behavior of the indexing processes without any concurrent operations except rebalancing was compared. Batches of seven different sizes were inserted into both B-trees. For each batch size, 10 different key sets were generated. Thus, batch insert of each size was performed 10 times such that the key set was different at each time. The number of disk accesses needed and the total batch insert time were calculated. The average batch-insertion time for each batch size for the B-trees having node size 100 keys is presented in Figure 8.1 and more exact results for three different batch sizes in Table 8.2. The results for the B-trees having node size 200 keys are presented in Figure 8.2 and Table 8.3. In the figures and tables TD refers to the algorithm that uses top-down balancing, REL1 and REL2 refer to the algorithms that use relaxed balancing and LINK to the algorithm that uses B^{link}-trees.

When comparing the results, we see that the algorithm TD using top-down balancing is the most efficient when batch insertion is considered alone. REL2 is a little slower than TD, but significantly faster than REL1 and LINK. However, the latter difference would be much smaller if the middle level nodes were kept in the main memory. When batches were inserted to tree T1, the batch insertion time increased little after the batch size of 10 000 keys. The reason for this is that the number of leaf nodes where new keys were inserted did not grow much after that. When the batches were inserted to tree T2, the situation was different because the number of node splits

	T1				T2			
	REL1	REL2	TD	LINK	REL1	REL2	TD	LINK
$N = 1000$								
$T_{g,\text{ave}}$	21.7	20.4	13.9	21.0	19.3	18.5	13.4	19.7
$T_{g,\text{min}}$	21.1	20.3	13.5	20.4	19.0	18.3	13.0	19.2
$T_{g,\text{max}}$	22.1	20.5	14.2	21.3	20.0	18.7	13.7	20.2
D_{ave}	1939	1944	1206	1856	1726	1638	1159	1732
$D_{\text{ave,key}}$	1.94	1.94	1.21	1.86	1.73	1.64	1.16	1.73
$N = 5000$								
$T_{g,\text{ave}}$	38.5	23.5	22.0	35.1	28.3	19.0	19.2	28.4
$T_{g,\text{min}}$	38.1	23.3	22.0	34.9	28.1	18.9	19.2	28.3
$T_{g,\text{max}}$	40.0	23.7	22.2	35.4	28.5	19.0	19.3	28.5
D_{ave}	3471	2644	1909	3120	2513	1713	1666	2499
$D_{\text{ave,key}}$	0.69	0.53	0.38	0.62	0.50	0.34	0.33	0.50
$N = 20000$								
$T_{g,\text{ave}}$	47.6	27.2	24.1	40.8	39.5	23.5	21.8	35.3
$T_{g,\text{min}}$	47.4	24.1	24.1	40.7	38.9	23.2	21.7	34.9
$T_{g,\text{max}}$	47.7	27.3	24.1	40.9	39.8	23.6	21.9	35.5
D_{ave}	4333	3474	2091	3649	3572	2745	1889	3139
$D_{\text{ave,key}}$	0.22	0.17	0.11	0.18	0.18	0.14	0.094	0.16

Table 8.2: Time and disk accesses needed by batch update and rebalancing processes. Node size was 100 keys. The size of the batch is denoted by N, average batch-insertion time (in seconds) by $T_{g,\text{ave}}$, average minimum batch-update time by $T_{g,\text{min}}$ and average maximum batch-update time by $T_{g,\text{max}}$. D_{ave} refers to the average number of disk accesses needed by a batch-insertion process and $D_{\text{ave,key}}$ to the average number of disk accesses for an indexed key occurrence. The B-tree originally had 60 000 keys, three levels and 12 nodes in the middle level.

	T1				T2			
	REL1	REL2	TD	LINK	REL1	REL2	LINK	TD
$N = 1000$	14.6	10.4	9.1	13.9	12.5	9.2	8.6	12.6
$T_{g,\text{ave}}$	13.8	10.2	8.8	13.3	12.2	9.1	8.3	12.3
$T_{g,\text{min}}$	13.8	10.2	8.8	13.3	12.2	9.1	8.3	12.3
$T_{g,\text{max}}$	14.8	10.6	9.2	14.1	12.7	9.2	8.7	12.8
D_{ave}	1298	1031	787	1227	1107	810	741	1109
$D_{\text{ave,key}}$	1.30	1.03	0.79	1.23	1.11	0.81	0.74	1.11
$N = 5000$								
$T_{g,\text{ave}}$	21.5	12.6	11.4	18.8	13.9	9.3	9.4	14.0
$T_{g,\text{min}}$	21.3	12.5	11.4	18.7	13.8	9.3	9.4	13.9
$T_{g,\text{max}}$	22.0	12.7	11.5	19.1	14.0	9.3	9.5	14.0
D_{ave}	1938	1516	988	1674	1235	834	817	1227
$D_{\text{ave,key}}$	0.39	0.30	0.20	0.34	0.25	0.17	0.16	0.25
$N = 20000$								
$T_{g,\text{ave}}$	23.8	13.5	11.9	20.2	18.8	11.3	10.5	16.9
$T_{g,\text{min}}$	23.8	13.5	11.9	20.2	18.1	11.0	10.4	16.5
$T_{g,\text{max}}$	23.9	13.5	11.9	20.3	18.1	11.4	10.6	17.1
D_{ave}	2161	1731	1034	1809	1694	1284	911	1503
$D_{\text{ave,key}}$	0.11	0.087	0.052	0.090	0.085	0.064	0.046	0.075

Table 8.3: Time and disk accesses needed by batch update and rebalancing processes. Node size was 200 keys. The size of the batch is denoted by N, average batch-insertion time (in seconds) by $T_{g,\text{ave}}$, average minimum batch-update time by $T_{g,\text{min}}$ and average maximum batch-update time by $T_{g,\text{max}}$. D_{ave} refers to the average number of disk accesses needed by a batch-insertion process and $D_{\text{ave,key}}$ to the average number of disk accesses for an indexed key occurrence. The B-tree originally had 60 000 keys, three levels and 3 nodes in the middle level.

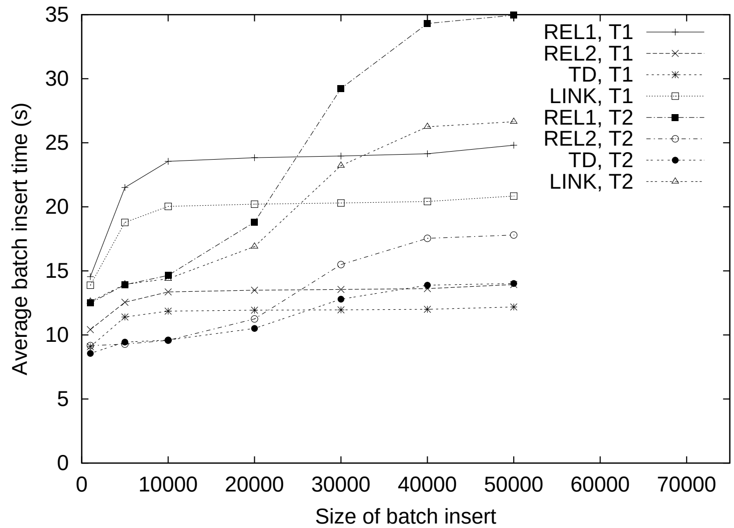


Figure 8.2: Average batch-insertion times for different algorithms. Node size was 200.

increased when the size of the batch increased.

Next the degree of concurrency allowed by the batch-update algorithms was compared. Search processes were started in groups containing 1 or 10 processes every 0.5 seconds during the batch-insertion process. The average and maximum times used by the search processes were calculated. The original B-trees and the keys in the batches were the same as in the first simulation. Again, batch insertion of each size was run with 10 different key sets. The average search times are presented in Figures 8.3, 8.4, 8.5 and 8.6 and some numerical results more exactly in Tables 8.4 and 8.5.

When considering the average and maximum search times during the batch insertion, the algorithms using relaxed balancing or B^{link}-trees allow considerably higher degree of concurrency than the algorithm TD. This can be clearly seen from Tables 8.6 and 8.7 where we compare average and maximum search times when using the algorithm TD and when using other algorithms. Actually, the average search time when using TD is often many times longer than the maximum search time when using the other algorithms. The average search time when using TD depends distinctly on the number of the nodes in the middle level (the fewer nodes in the middle level, the longer the search times). When using the other algorithms, the average search time is practically constant. This comes from the fact that in connection with TD the search time is dominated by the time to wait for the lock of the middle node, while in connection with the other algorithms the search time is dominated by the two disk accesses needed by every search. When searches are performed simultaneously with batch insertion TD, the search times have wide variation, because the time of a

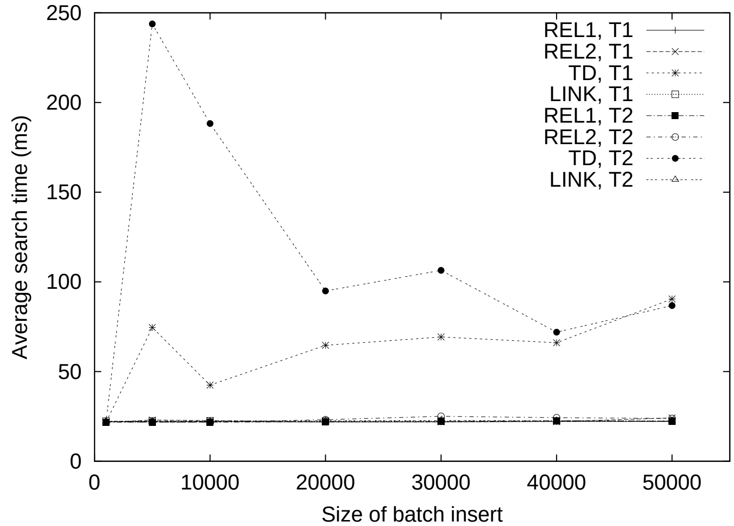


Figure 8.3: Average search times when the searches are performed concurrently with batch insertion. A search process was started every 0.5 seconds. Node size was 100.

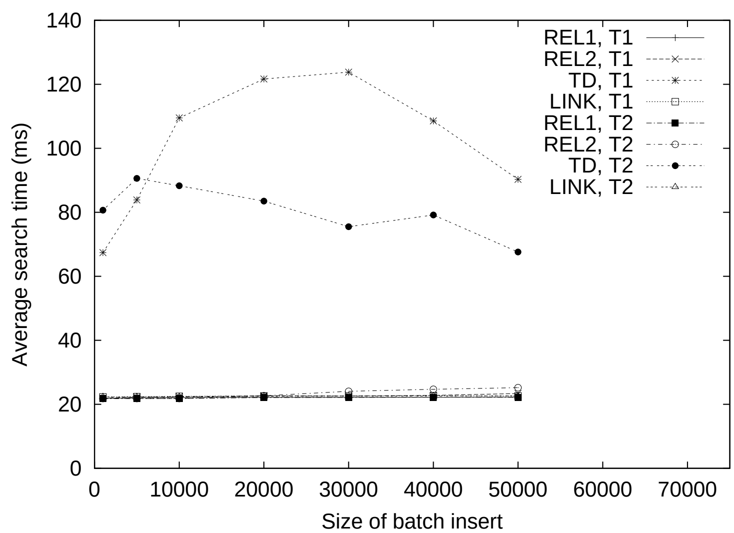


Figure 8.4: Average search times when the searches are performed concurrently with batch insertion. Search processes were started in the groups of 10 processes every 0.5 seconds. Node size was 100.

	T1				T2			
	REL1	REL2	TD	LINK	REL1	REL2	TD	LINK
$N = 1000$								
$t_{s,\mathrm{ave}}$	22.0	21.8	21.8	22.3	21.8	21.8	21.8	22.3
$t_{s,\mathrm{min}}$	21.8	21.8	21.8	22.3	21.8	21.8	21.8	22.3
$t_{s,\mathrm{max}}$	22.5	21.8	22.0	22.5	22.1	21.8	22.2	22.3
t_{max}	51.25	21.8	28.45	32.5	33.8	21.8	33.5	22.3
$N = 5000$								
$t_{s,\mathrm{ave}}$	22.0	23.0	74.6	22.4	21.8	21.8	243.8	22.3
$t_{s,\mathrm{min}}$	21.8	22.1	71.4	22.3	21.8	21.8	236	22.3
$t_{s,\mathrm{max}}$	22.6	24.8	79.7	22.6	21.8	21.9	248	22.3
t_{max}	53.2	61.1	1170	42.1	21.8	25.5	1820	22.3
$N = 20000$								
$t_{s,\mathrm{ave}}$	21.9	22.0	64.7	22.4	22.0	23.2	95.0	22.6
$t_{s,\mathrm{min}}$	21.8	22.0	62.9	22.4	21.8	22.4	89.9	22.3
$t_{s,\mathrm{max}}$	22.4	22.0	66.4	22.6	22.2	24.4	100	22.9
t_{max}	53.9	30.4	938	35.1	53.2	63.0	1980	49.3

Table 8.4: Search times when searches are made concurrently with a batch insertion. One search was started every 0.5 seconds during the batch insertion. Node size was 100 keys. The B-tree originally had 60 000 keys, three levels and 12 nodes in the middle level. The size of the batch is denoted by N, the average search time (in milliseconds) by $t_{s,\mathrm{ave}}$. The minimum and maximum average search times during all batches of the same size are denoted by $t_{s,\mathrm{min}}$ and $t_{s,\mathrm{max}}$, respectively. The maximum search time during all batch insertions is denoted by t_{max}.

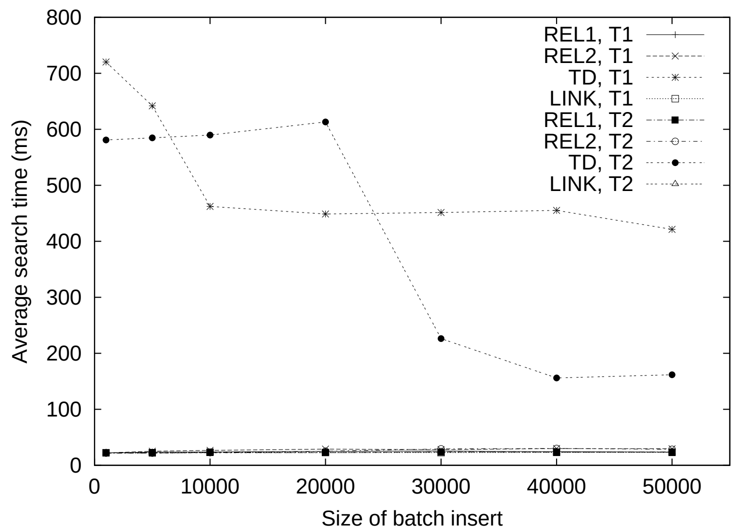


Figure 8.5: Average search times when the searches are performed concurrently with batch insertion. A search process was started every 0.5 seconds. Node size was 200.

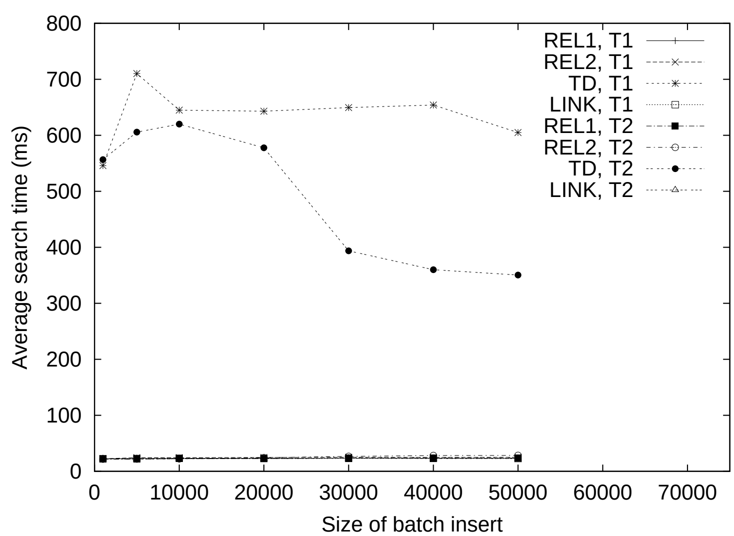


Figure 8.6: Average search times when the searches are performed concurrently with batch insertion. Search processes were started in the groups of 10 processes every 0.5 seconds. Node size was 200.

	T1				T2			
	REL1	REL2	TD	LINK	REL1	REL2	TD	LINK
$N = 1000$								
$t_{s,\text{ave}}$	22.2	22.0	720	22.5	21.8	21.8	581	22.3
$t_{s,\min}$	21.8	21.8	665	22.3	21.8	21.8	559	22.3
$t_{s,\max}$	23.6	22.6	768	23.0	21.8	21.8	606	22.3
$t_{\max}$	52.7	38.6	2900	42.5	21.8	21.8	3020	22.3
$N = 5000$								
$t_{s,\text{ave}}$	23.6	25.2	642	22.8	21.9	21.8	585	22.4
$t_{s,\min}$	21.8	23.7	491	22.4	21.8	21.8	582	22.3
$t_{s,\max}$	25.9	27.6	668	23.2	22.4	21.8	589	22.6
$t_{\max}$	53.1	54.0	3810	38.0	37.7	21.8	3300	30.5
$N = 20000$								
$t_{s,\text{ave}}$	24.9	28.9	449	23.0	22.6	25.1	613	22.6
$t_{s,\min}$	24.7	28.3	448	23.0	21.8	23.7	583	22.3
$t_{s,\max}$	25.0	29.1	450	23.0	24.2	27.7	640	23.0
$t_{\max}$	52.5	62.9	3450	38.5	70.8	84.4	3600	39.5

Table 8.5: Search times when searches are made concurrently with a batch insertion. One search was started every 0.5 seconds during the batch insertion. Node size was 200 keys. The B-tree originally had 60 000 keys, three levels and 3 nodes in the middle level. The size of the batch is denoted by N, the average search time (in milliseconds) by $t_{s,\text{ave}}$. The minimum and maximum average search times during all batches of the same size are denoted by $t_{s,\min}$ and $t_{s,\max}$, respectively. The maximum search time during all batch insertions is denoted by $t_{\max}$.

		Ratios of average search times		
Node size	Batch size	TD / REL1	TD / REL2	TD / LINK
100	1000	3.1	3.1	3.0
	5000	3.8	3.8	3.7
	10000	5.0	4.9	4.9
	20000	5.5	5.4	5.4
	30000	5.6	5.5	5.5
	40000	4.9	4.8	4.8
	50000	4.1	3.9	4.0
200	1000	25	24	24
	5000	31	29	31
	10000	28	26	28
	20000	28	26	28
	30000	28	26	28
	40000	28	26	28
	50000	26	24	26

Table 8.6: Ratios of average search times when searches are made concurrently with a batch insertion into tree T1. Ten search processes were started every 0.5 seconds. Node size means the maximum number of keys in a node. The B-tree originally had 60 000 keys.

certain search process heavily depends on whether this process has to wait for a lock kept by batch-insertion process or not. Because the number of search processes which have to wait for the lock varies, this causes wide variation in the average search times, especially when the number of search processes is low (Figures 8.3 and 8.5). In the simulations where 10 search processes were started every 0.5 seconds, the variation was smaller because the larger number of processes smoothed the results (Figures 8.4 and 8.6).

8.3 Simulation Results for Full-Text Indexing

In the next simulations, full-text-indexing systems using different batch-insertion algorithms were compared. The simulations were performed using different node sizes and both English and Finnish texts. There is a considerable difference depending on the language used. When the text is in English, the number of unique keys in the index is much smaller than when the text is in Finnish. This is because of the richer morphology of the Finnish language.

		Ratios of maximum search times		
Node size	Batch size	TD / REL1	TD / REL2	TD / LINK
100	1000	31	27	37
	5000	36	36	47
	10000	36	32	47
	20000	40	36	46
	30000	38	33	41
	40000	28	30	39
	50000	30	25	38
200	1000	67	66	76
	5000	72	63	89
	10000	61	62	81
	20000	70	57	84
	30000	70	58	84
	40000	69	58	85
	50000	62	57	87

Table 8.7: Ratios of maximum search times when searches are made concurrently with a batch insertion into tree T1. Ten search processes were started every 0.5 seconds. Node size means maximum the number of keys in a node. The B-tree originally had 60 000 keys.

	TD	REL1	REL2	LINK
NODE SIZE 4096 BYTES				
Total number of disk accesses	14148	24039	22207	21548
Disk accesses / indexed key occurrence	0.012	0.020	0.019	0.018
Total time (4 merges), s	157.7	262.4	231.4	237.3
NODE SIZE 6144 BYTES				
Total number of disk accesses	10538	16915	15891	15259
Disk accesses / indexed key occurrence	0.0090	0.014	0.013	0.013
Total time (4 merges), s	116.1	180.9	155.5	166.9

Table 8.8: Disk accesses and time needed by batch-insertion and rebalancing processes. A Finnish text of 11 MB was indexed in 5 batch-insertion merges using two different node sizes. Results from the last 4 merges are shown in the table.

	TD	REL1	REL2	LINK
Total number of disk accesses	8450	11758	10970	10852
Disk accesses / indexed key occurrence	0.0063	0.0087	0.0082	0.0081
Total time (4 merges), s	90.0	121.1	105.2	115.8

Table 8.9: Disk accesses and time needed by batch-insertion and rebalancing processes. An English text of 10 MB was indexed in 5 batch-insertion merges using node size 4096 bytes. Results from the last 4 merges are shown in the table.

The texts in English were collected from newspaper-like articles in the Internet. The test material contained only the bodies of the articles; all usenet news headers had been removed. The Finnish test material consisted of articles from a weekly magazine.

Again, the behavior of the batch-insertion processes without any concurrent operations except rebalancing was compared first. The indexed data consisted of 11 MB of Finnish text and of 10 MB of English text. Both types of text were indexed in five batch-update merges. The number of disk accesses needed by the batch-update and rebalancing processes was counted. For the simulations using relaxed B-trees, the first batch-update merge was done with the TD algorithm, because the algorithms using relaxed balancing are not suitable for situations where the size of the batch update is many times larger than the size of the B-tree. This is not a serious problem because the first merge can always be done in a bottom-up manner very efficiently. The disk operations needed and the words indexed in the first merge were not calculated in the results. The results are presented in Tables 8.8 and 8.9.

It can be seen that only 0.0063–0.020 disk accesses were needed for an indexed key occurrence. This is a considerable saving when compared with the situation where the key occurrences would be indexed in random order. The latter would mean at least 3 disk accesses for each key occurrence. When comparing the efficiency of the different algorithms, we see that the algorithm TD is the most efficient again. However, the algorithm REL2 is now not much faster than the algorithms REL1 and LINK. This is because the second batch-update merge is about the same size as the B-tree constructed during the first merge. Nearly all leaf nodes are split during the second merge. The r-locking of the middle node in REL2 blocks the balance process from doing anything under that node. So, the B-tree may be locally more imbalanced than when using the algorithms REL1 and LINK. This increases the number of disk accesses needed and lengthens the batch-update time.

Next the degree of concurrency allowed by the batch-update algorithms was compared. Search processes were started in the groups of 10 processes every 0.5 seconds during the batch-update processes. The searches were performed during the fourth and fifth merges of the Finnish text and during the fifth merge of the English text. For relaxed B-trees, previous merges were performed with the TD algorithm to make the start situations comparable. However, this could not be done for B^{link}-trees, because the structure of a B^{link}-tree differs from the structure of a normal B-tree having the same keys and the same node size. This difference results from a B^{link}-tree having an extra value, highvalue, and an extra pointer, link, in every leaf node. So, less space remains for actual keys. In all simulations the B-tree had three levels when the merges started.

The search keys were randomly picked from a text file that was already indexed. The average and maximum times used by the search processes were calculated. The results are presented in Tables 8.10, 8.11 and 8.12. The time needed to read the possible continuation records from the disk was not counted into the search times, because it heavily depends on the words to be searched for.

The ratios of concurrent search times are summarized in Tables 8.13 and 8.14. Again, the concurrent searches are much slower when TD is used than when the other algorithms are used, and the search times during TD depend on the number of the nodes in the middle level. In these simulations, the algorithm REL2 performs the batch update more efficiently than the algorithms REL1 and LINK. This is because the number of the leaf nodes that have to be split is not so high.

To summarize the results, they correspond to what we expected. The algorithm that uses top-down balancing is the most efficient if the batch update is considered alone. It needs fewer disk accesses than the algorithms using relaxed B-trees and B^{link}-trees. On the other hand, the algorithms using relaxed B-trees and B^{link}-trees allow a better degree of concurrency. The average search times are distinctly better

	TD	REL1	REL2	LINK
FOURTH BATCH-INSERTION MERGE				
Fraction of split nodes, %	11.9	11.9	11.9	18.2
Time of the batch-insert, s	47.3	70.7	59.3	69.0
Average search time, s	0.32	0.022	0.022	0.022
Maximum search time, s	6.9	0.044	0.053	0.042
FIFTH BATCH-INSERTION MERGE				
Fraction of split nodes, %	6.5	6.5	6.5	4.9
Time of the batch-insert, s	51.9	74.1	61.3	73.5
Average search time, s	0.15	0.022	0.022	0.022
Maximum search time, s	5.2	0.036	0.052	0.034

Table 8.10: Search times when searches are made concurrently with a batch insertion. The batch insertion was made with Finnish text and node size used was 4096 bytes. The B-tree had 7 or 8 nodes in the middle level before the fourth merge and 8–10 nodes before the fifth merge.

	TD	REL1	REL2	LINK
FOURTH BATCH-INSERTION MERGE				
Fraction of split nodes, %	7.3	7.3	7.3	11.0
Time of the batch-insert, s	35.0	49.1	41.0	48.5
Average search time, s	0.94	0.022	0.022	0.022
Maximum search time, s	8.3	0.046	0.050	0.042
FIFTH BATCH-INSERTION MERGE				
Fraction of split nodes, %	3.8	3.8	3.8	3.2
Time of the batch-insert, s	37.3	50.9	42.8	50.9
Average search time, s	0.59	0.022	0.022	0.022
Maximum search time, s	6.6	0.051	0.033	0.051

Table 8.11: Search times when searches are made concurrently with a batch insertion. The batch insertion was made with Finnish text and a node size used was 6144 bytes. The B-tree had 3 or 4 nodes in the middle level before the fourth merge and 5 or 6 nodes before the fifth merge.

	TD	REL1	REL2	LINK
Fraction of split nodes, %	10.1	10.1	10.1	12.4
Time of the batch-insert, s	28.1	36.8	29.8	36.2
Average search time, s	1.6	0.022	0.022	0.022
Maximum search time, s	10.8	0.050	0.073	0.097

Table 8.12: Search times when searches are made concurrently with a batch insertion. The batch insertion was made with English texts and the node size used was 4096 bytes. The B-tree had 3 nodes in the middle level before the batch-insertion merge.

				Ratios of average search times		
Language	Node	d	Batch size	TD / REL1	TD / REL2	TD / LINK
Finnish	4096	7–8	294300	15	15	15
Finnish	4096	8–10	294281	6.8	6.8	6.8
Finnish	6144	3–4	294300	43	43	43
Finnish	6144	5–6	294281	27	27	27
English	4096	3	325598	73	73	73

Table 8.13: Ratios of average search times when searches are made concurrently with a batch insertion into a full-text index. Language means the language of the indexed text, node means the node size in bytes, d refers to the number of nodes in the middle level before the batch insertion and batch size is the number of key occurrences in the batch insertion.

				Ratios of maximum search times		
Language	Node	d	Batch size	TD / REL1	TD / REL2	TD / LINK
Finnish	4096	7–8	294300	160	130	160
Finnish	4096	8–10	294281	140	100	150
Finnish	6144	3–4	294300	180	170	200
Finnish	6144	5–6	294281	130	200	130
English	4096	3	325598	220	150	110

Table 8.14: Ratios of maximum search times when searches are made concurrently with a batch insertion into a full-text index. Language means the language of the indexed text, node is the node size in bytes, d refers to the number of nodes in the middle level before the batch insertion and batch size is the number of key occurrences in the batch insertion.

than when using top-down balancing. If the number of the nodes that have to be split during the batch insertion is not very high and only the root of the B-tree is kept in the main memory, REL2 performs the batch update more efficiently than REL1 and LINK . However, this advantage is lost when the fraction of the split nodes increases.

Although there are many simplifications in the simulation model used (for example, a constant time is assumed for disk operations), the used values are near enough to realistic values that the order of magnitude in the results is realistic. It is also worth noting that the used parameter values (except the node size) do not affect the number of disk accesses needed by the batch-update algorithms. If the very simple buffering model (only the root is in the main memory) were changed to more realistic model where the buffer pool is managed in LRU fashion, the number of disk accessess and thus batch-update times would even decrease.

Chapter 9

Batch Updates and Differential Indexing

In this chapter we present an indexing system where a database index is divided into two parts: the main index located on disk and the differential index in the main memory. Periodically, writes of committed transactions are transferred from the differential index to the main index as a batch-update operation. We consider a system where there are many simultaneous transactions each possibly consisting of several actions. In addition to B-tree node level locking, the system uses another concurrency control strategy to assure consistency on transaction level: either key-value locking or a strategy based on multiversion timestamp ordering. We also consider recovery after system crash. The system described in this chapter was originally constructed and implemented by Jarmo Ruuth [55], but we have designed the formal presentation and the correctness proofs in this chapter.

In our abstract model we assume that all database objects have a unique key and all objects are accessed through one search structure, a B-tree. This search structure has a differential file containing the active objects, i.e., objects written by the active transactions. Objects that are no more active (or insertions or deletions of no more active objects) will be transferred to the main index by a separate background process. This can be done efficiently using batch updating of B-trees.

In database systems where conflicts between transactions are rare, a timestamp based concurrency control system may be preferred to locking based concurrency control. However, if the data objects accessed by transactions are on disk, the checks which are performed by a concurrency control system are slow because of disk reads. In the differential indexing system presented in this chapter the key observation is that—on the transaction level—only those database objects that are accessed by some active transaction may conflict. This means that the whole concurrency control may be limited to the part of the database accessed by the active transactions. The idea is then to keep all objects written by the active transactions in a differential file, which is usually small compared with the whole database. It is then possible to construct a timestamp based concurrency control strategy where transaction concurrency control operations can be limited to the differential file.

If a system crash occurs when a database system is active, recovery of the database has to be performed before the database can be used. First, all data written by committed transactions has to be written to the database, if it is not there. Secondly, if the database contains data written by aborted transactions, it has to be removed. The most common way to take care of this is to use a log file. The log file contains a list of actions the transactions have performed. It also contains information needed to undo or redo the actions (for example, new or old values of data objects). After a system crash, the log file is examined. The writes of the aborted transactions are undone and the writes of the committed transactions are redone. This usually takes a long time because the data objects to be written may be scattered around the disk. By planning carefully the order in which the data objects are written, the writes are reported to the log and the transactions are committed or aborted, it is possible to make either undo or redo operations unnecessary, but not both.

The differential indexing makes a faster recovery possible. In our scheme all changes to the main index will be performed only after the corresponding transactions have been committed. A major benefit here is that no potentially aborted transaction can perform a change in the main index structure. Thus, after a system crash we can simply continue working with the old main index, and build a new differential index according to the transactions that were committed but not transferred to the main index before the crash. No undo operations are needed and the redo operations are fast, because they write to the differential index in the main memory.

9.1 Database Actions for Differential Files

In this section, we explain how to implement database actions using differential files. We concentrate on how to implement actions when transactions are executed one at a time. The situation where several transactions are executed concurrently is explained in the next sections.

We assume that all database objects have a unique key and all objects are accessed through one index. This index consists of two parts: The main index M residing on disk and a differential file D containing objects written by active and recently committed transactions. The differential file is usually in the main memory. The objects written by committed transactions are periodically transferred from D to M. This process can be done concurrently with transaction actions.

A transaction action $A(x)$ is started by performing a search for x in D. If D contains x, the value of x is used by the action. If x is not found in D, the search is performed in M. If the transaction action is insert or write, a new value is written for the data object x and inserted into D. (This means a new search for x in D, if a search in M had to be performed.) For a delete action, x is inserted into D with a special delete mark. Active transactions make all their writes in D.

Periodically, part of the contents of the differential file is transferred to the main index. This process is called *merging*. Only values that have been written by transactions committed before the merge begins are transferred. It depends on the concurrency control method used if all such values are transferred or if part of the committed writes are still left in the differential file. No values written by an uncommitted transaction are transferred to the main index.

Depending on the concurrency control method used, D may contain several *versions* of the same data object, for example, a version that has been written by a committed transaction and another version that has been written by a still active transaction. The versions are used either to make aborts easier or as a part of the concurrency control in a multiversion timestamp ordering system. If versions are used, each version is associated with the label of the transaction which wrote the record. The transaction labels are unique. A version is called *valid for transaction* T, if the actions in transaction T are allowed to read this version. Whether a certain version is valid for a certain transaction or not, depends on the concurrency control method used and is explained in sections considering concurrency control. There is always at most one valid version of a certain data object for transaction T. M contains at most one version of a certain data object and all data objects in M are written by committed transactions.

When versions are used, the actions are performed as explained above, but only valid versions of data objects are considered. A transaction action $A(x)$ is started by performing a search for x in D. If D contains a valid version of x, this version is used by the action. If there is no valid version of x in D, the search is performed in M. In that case, if a version is found in M, it is always valid.

In the merging process, no versions written by uncommitted transactions are transferred to the main index. When a data object has several versions in D which should be transferred, only the version that has been written by the most recently committed transaction is written to M. The earlier versions are deleted from D.

Both the main index and the differential file are implemented as B-trees. There are some advantages from implementing the differential file as a B-tree and not as a binary tree: First, it is easy to have many versions of the same object in a B-tree. Second, should the differential file grow too large to be kept in the main memory, it would be easy to transfer it temporarily to disk. Third, when the differential file is implemented as a B-tree, it is easy to transfer the objects written by committed transactions efficiently to the main index concurrently with executing transaction actions.

9.2 Differential Files and Two Phase Locking

Next, we consider a situation where several transactions may be executed concurrently. In this section, we explain how to implement concurrency control based on two phase locking.

To allow efficient concurrency control, two levels of locks are needed. Transaction level locks are used to make all possible executions of concurrent transactions serializable. These locks are taken on data objects and they are maintained by a lock table, which is implemented as a separate data structure in the main memory. Node level locks are used to preserve the validity of the B-tree search structures. These locks are taken on index objects. Notice that a transaction level lock never conflicts with a node level lock and vice versa. The distinction between transaction level locks and node level locks is also made by Mohan in [46], which describes algorithms offering a wide set of operations like range scans. However, Mohan's work is based on the normal index structure, not on differential files.

To assure correct execution of concurrent transactions, the following rules are followed:

1. Strict two phase locking protocol is used for transaction level locks, i.e. if a transaction takes a lock on a data object, the lock is not released until the transaction is either committed or aborted.

2. Some standard B-tree concurrency control strategy (e.g., naive lock-coupling [4], top-down [49], relaxed B-trees [50], optimistic decent [3] or B^{link}-trees [35, 40, 56]) is used to assure that the index structure preserves its validity. A certain transaction action has locks in only one B-tree simultaneously. For example, all locks in D are released before the search in M is started.

3. When a data object x is transferred from the differential file to the main index, the differential index node containing x is exclusively locked. The lock is not released until the data object is written into the main index.

A transaction takes a transaction level r-lock on every data object it refers to. When the transaction notices that it will write the value of a certain object, it converts the r-lock to an x-lock. The locks are released after the transaction has committed. If the transaction has to be aborted, all of its locks are immediately released. If a transaction T wants to read a data object x and there are several versions for x in D, the valid version for T is always the version written by a transaction that has been committed most recently. Because write actions keep x-locks on the data objects they write until commit, it is not possible to have two versions of the same data object written by transactions committed simultaneously.

As stated above, any standard B-tree algorithm can be used to perform the transaction and transfer actions. In Figure 9.1, we describe the merging algorithm for a B^{link}-tree. The transaction actions are then implemented by using standard B^{link}-tree procedures [35]. The advantage of implementing the differential file as a B^{link}-tree is that it is easy to collect the object values for merging. Links in the leaf level are followed to find all object values written by committed transactions. Only when a leaf becomes totally empty (or alternatively, when a leaf becomes less than half full), it is necessary to return to the upper levels of the tree to remove the leaf by standard B^{link}-tree procedures.

The most efficient way to write the new versions to the main index depends on the size of the differential file and on the distribution of the data objects in it. If the differential index is small compared to the main index and the objects are evenly distributed, it is probably better to make the updates one at a time. However, if the differential file is large or object values are near each other, a batch-update algorithm may be used to improve the efficiency. A merging algorithm for a situation where the updates are made one at a time is described in Figure 9.1.

9.3 Concurrency Control Method Based on Multiversion Timestamp Ordering for Differential Files

In this section, a concurrency control strategy based on timestamps for differential files is explained. We denote this method by MVTO in the later sections. In this method, no transaction level locks are used, but timestamps are used to assure the transaction level serializability. Otherwise, the transaction actions are performed as explained in the previous sections.

Each transaction T_i is given a unique *transaction number*, t_i. The transaction numbers are used to set the conflicting transactions in serial order. The transaction number is assigned to the transaction when the transaction commits. If $t_i < t_j$, then transaction T_i has commited before transaction T_j.

In addition to the transaction number, each transaction T_i has a *transaction start number TSN_i*. The transaction start number is chosen so that every transaction T_j for which $t_j < TSN_i$ has been either committed or aborted before the transaction T_i begins.

```
algorithm MERGE
begin
    C_transactions :=  the set of committed transactions whose writes are
    transferred to the main index in this merge;
    x_key :=  a key smaller than the smallest possible data object key;
    SEARCH(x_key, leaf); /* leaf x-locked here */
    while (leaf is not NULL) do begin
        while (there are objects whose keys are larger than x_key in leaf) do begin
            x :=  next object whose key is larger than x_key  in leaf;
            x_key :=  the key of x;
            if x has values written by C_transactions then begin
                x_value :=  value of x written by the last C_transaction;
                remove all values of x written by C_transactions and other
                committed transactions preceding the last C_transaction;
                if there remains no value for x in leaf then
                    remove x from leaf;
                if x_value is a delete entry then
                    DELETE(x_key, main_index);
                else
                    INSERT(x_key, x_value, main_index);
            end;
        end;
        if leaf is empty then begin
            delete leaf by standard B-link operations;
            /* all locks are released here */
            SEARCH(x_key, leaf);
            /* leaf is x-locked if it is not NULL */
        end;
        else begin
            old_leaf := leaf;
            leaf :=  link pointer in leaf;
            RELEASE_X_LOCK(old_leaf);
            if leaf is not NULL then
                X_LOCK(leaf);
        end
    end
end.
```

Figure 9.1: Merging the differential index with the main index. Here the differential file is implemented as a B^{link}-tree and the updates into the main index are done one by one. Procedure SEARCH(*key*, *leaf*) is a standard B^{link}-tree search procedure which starting from the root searches the leaf where *key* belongs and stores it in the variable *leaf*.

When a transaction writes a record in a differential file, it saves its identifier with the record. When a transaction T_i makes a read operation, it considers only the values written by itself or by transactions that have transaction number smaller than TSN_i.

If the transaction T_i tries to write a version of object x and it notices that x already has a record written by transaction T_j such that $TSN_i \leq t_j$, transaction T_i is aborted. Because the conflicting writes are always in the differential index, this check can always be done without reading the main index. This makes the checking fast.

Because the checks that the transaction T_i performs when it reads or writes objects are all based on the transaction start number TSN_i, it is possible to make these checks although the transaction number of T_i is not known yet when T_i is active.

When a transaction T_i containing both read and write actions commits, one check is still needed to assure serializability. Each data object x read by the transaction is checked to see that there is no version of x written by a transaction T_j so that $t_j \geq TSN_i$ and T_j has committed. If there is such a version of x, transaction T_i is aborted. When T_i is active, all values written by transactions having a transaction number greater than or equal to TSN_i are still in the differential file. Thus, no disk accesses are needed for this check. This check is not needed if T_i contains only read actions and no write actions.

When a merge is made, only object values written by committed transactions whose transaction number is smaller than the smallest transaction start number among active transactions are transferred. Otherwise, the merge is executed in the same way as explained in the previous section. In the main index the object values are written without transaction numbers. Only one version of each object is saved.

9.4 Correctness of the Timestamp Method

In this section, we prove the MVTO algorithm described above to be correct. By correctness we mean that every committed projection $C(H)$ of history H produced by MVTO algorithm is equivalent to a serial single-version history of the same transactions. In a single-version history only one version of each data object is used. We use terms and methods given in [6]. By denotation x_i we mean a version of x written by transaction T_i. Thus, the read action of transaction T_k where a version of x written by T_i is read is denoted by $r_k(x_i)$. First, we define two terms needed in the correctness proofs.

Definition 9.1 *A serial multiversion history is* ONE-COPY SERIAL, *if for all i, j and x, if T_i reads x from T_j, then $i = j$, or T_j is the last transaction preceding T_i that writes any version of x.*

Definition 9.2 *A multiversion history is* ONE-COPY SERIALIZABLE *if its committed projection is equivalent to a one-copy serial multiversion history.*

It has been shown in [6] that if H is a multiversion history over transaction system T, its committed projection $C(H)$ is equivalent to a serial, single-version history over T if and only if H is one-copy serializable.

In order to determine if a multiversion history is one-copy serializable, a *multiversion serialization graph* can be used. To define the graph for H, we need a *version order*, $\ll$, for data items in H. Given a multiversion history H and a data item x, a version order $\ll$ for x in H is a total order of versions of x in H. A version order $\ll$ for H is the union of the version orders for all data items.

Given a multiversion history H and a version order $\ll$, the *multiversion serialization graph for H and* $\ll$, MVSG(H,$\ll$) is defined as follows: MVSG(H,$\ll$) is a directed graph whose nodes are the transactions in T that are committed in H. The graph has two kinds of edges, read edges and version order edges. For each read operation $r_k(x_j), k \neq j$, there is a read edge $T_j \to T_k$. Version order edges are added for each pair $r_k(x_j)$ and $w_i(x_i)$ in $C(H)$ where i, j and k are distinct. If $x_i \ll x_j$, then $T_i \to T_j$ is added, otherwise $T_k \to T_i$ is added. No other edges are included in MVSG(H,$\ll$).

It has been shown in [6] that a multiversion history H is one-copy serializable if and only if there exists a version order $\ll$ such that MVSG(H,$\ll$) is acyclic.

Next, we want to show that the MVTO algorithm given in the previous section is correct by showing that all histories produced by MVTO have an acyclic MVSG. We start by listing the essential characteristics of every MVTO history H over a transaction system $\{T_0, \ldots T_n\}$. We define the version order as follows: $x_i \ll x_j$ iff $t_i < t_j$. By denotation $A_i \prec A_j$ we mean that action A_i precedes action A_j. The commit action of transaction T_i is denoted by c_i and the abort action by a_i.

STO1 For each T_i, there is a unique timestamp t_i; that is, $t_i = t_j$ iff $i = j$.

STO2 For each T_i, there is a transaction start number TSN_i. For every T_j, such that $t_j < TSN_i$, and for every $A_i \in H$, either $c_j \prec A_i$ or $a_j \prec A_i$.

STO3 For every $r_k(x_j) \in H$, $w_j(x_j) \prec r_k(x_j)$ and $t_j < TSN_k$.

STO4 For every $r_k(x_j)$ and $w_i(x_i) \in H$, $i \neq j$, either (a) $t_i < t_j < TSN_k$ or (b) $t_i > t_j$, $TSN_k \leq t_i$ and in addition, $t_k < t_i$ if T_k contains write action(s) or (c) $i = k$ and $r_k(x_j) \prec w_i(x_i)$.

STO5 If $w_i(x_i) \in H$ and $w_j(x_j) \in H$, $i \neq j$, then either $t_i < TSN_j$ or $t_j < TSN_i$.

Properties STO1 and STO2 follow from the way transaction numbers and transaction start numbers are given to the transactions. STO3 follows from the definition of the valid version for transaction T_k. STO5 is assured by the check that is made when a transaction wants to write a data object. STO4 lists the reasons why transaction T_k reads a version written by T_j and not a version written by T_i. There are three possible cases:

1. $x_i \ll x_j$ and x_j is a valid version of x for T_k. By definition of $\ll$, $t_i < t_j$ and $t_j < TSN_k$, because otherwise x_j would not be valid for T_k.

2. Transaction number t_i is so high that x_i is not valid for T_k. In that case, $TSN_k \leq t_i$. If T_k contains at least one write action, there is a validation check when T_k commits. It is checked that there is no version of x written by a committed transaction having a transaction number higher than t_j. This implies that all transactions writing x and having a transaction number higher than t_j must commit after T_k commits. Thus, $t_k < t_i$. If T_k contains no write actions, the validation check is not made and it is possible that $t_k > t_i$.

3. T_k writes x after it has read a version of x written by another transaction.

We want to prove that every committed projection of history H satisfying properties STO1 - STO5 and thus every history produced by MVTO algorithm is one-copy serializable. We first show that MVSG(H',$\ll$) is acyclic. H' is a history that is otherwise the same as H, but all transactions that contain only read actions are removed. We call those transactions *read-only transactions*. After that, we show that including nodes and edges belonging to read-only transactions does not cause any cycles to the MVSG.

Lemma 9.1 *Let H be any history satisfying the properties STO1–STO5 and H' a history that is otherwise the same as H, but all read-only transactions of H are removed. Let the version order $\ll$ be defined as follows: $x_i \ll x_j$ iff $t_i < t_j$. Then, MVSG(H',$\ll$) is acyclic.*

Proof. We prove that MVSG(H',$\ll$) is acyclic by showing that for every edge $T_i \rightarrow T_j$ in MVSG(H',$\ll$), $t_i < t_j$. We first consider edges $T_i \rightarrow T_j$ originating from read actions $r_j(x_i)$. By STO3, $t_i < TSN_j$ and thus $t_i < t_j$. Next, we consider the version order edges $T_i \rightarrow T_j$ and $T_k \rightarrow T_i$. As defined above, $x_i \ll x_j$ iff $t_i < t_j$. Let $r_k(x_j)$ and $w_i(x_i)$ be in H'. There are two cases: (1) $x_i \ll x_j$, which implies the edge $T_i \rightarrow T_j$ in MVSG(H',$\ll$); and (2) $x_j \ll x_i$, which implies the edge $T_k \rightarrow T_i$ in MVSG(H',$\ll$). In case (1), by definition of $\ll$, $t_i < t_j$. In case (2), by STO4 either $t_i < t_j$ or $t_k < t_i$. The first option is impossible, because $x_j \ll x_i$ implies $t_j < t_i$. So $t_k < t_i$ as desired. $\square$

Lemma 9.2 *Let H be any history satisfying the properties STO1–STO5 and H' a history that is otherwise the same as H, but all read-only transactions of H are removed. Let the version order $\ll$ be defined as follows: $x_i \ll x_j$ iff $t_i < t_j$. If MVSG(H',$\ll$) is acyclic, then MVSG(H,$\ll$) is also acyclic.*

Proof. We show that adding nodes and edges belonging to read-only transactions to MVSG(H',$\ll$), transforming it to MVSG(H,$\ll$) does not bring any cycles to the graph. Consider a node for read-only transaction T_k. All incoming edges for node T_k are of the form $T_l \rightarrow T_k$ coming from read action $r_k(x_l)$. According to STO3, $t_l < TSN_k$.

All edges leaving T_k are of the form $T_k \rightarrow T_i$ where T_k contains the action $r_k(y_j)$, T_i contains an action $w_i(y_i)$, $y_j \ll y_k$ and thus $t_j < t_i$. By STO4, $TSN_k \leq t_i$. Thus, for every path $T_l \rightarrow T_k \rightarrow T_i$ of length 2 and having a read-only transaction node as its middle node, $t_l < TSN_k \leq t_i$ and $t_l < t_i$. Suppose that there is a cycle $T_i \rightarrow T_k \rightarrow T_i$ of length 2 in MVSG(H,$\ll$) and that T_k is a read-only transaction. Then, $t_i < TSN_k \leq t_i$ and thus $t_i < t_i$. But, this is impossible and thus MVSG(H,$\ll$) does not contain any cycles of length 2.

Now, consider an arbitrary path $T_1 \rightarrow T_2 \rightarrow \ldots T_n$ in MVSG(H,$\ll$), where $n \geq 3$ and T_1 and T_n are not read-only transactions. Because read-only transaction nodes do not have leaving edges to any read-only transaction nodes, it is impossible to have two consecutive read-only transaction nodes in the path. Thus, $t_n \geq t_1 + \frac{n}{2}$ if n is even and $t_n \geq t_1 + \frac{n-1}{2}$ if n is odd.

Suppose that there is a cycle $T_1 \rightarrow \ldots T_n \rightarrow T_1$ in MVSG(H,$\ll$) and $n \geq 2$. Without loss of generality, we can assume that T_1 is not a read-only transaction. Thus, we get $t_1 \geq t_1 + \frac{n}{2}$ if n is even and $t_1 \geq t_1 + \frac{n+1}{2}$ if n is odd. But this is impossible and thus MVSG(H,$\ll$) is acyclic. □

Theorem 9.1 *Every committed projection $C(H)$ of history H produced by MVTO algorithm is one-copy serializable.*

Proof. Let H be any history produced by MVTO algorithm and H' a history that is otherwise the same as H, but all read-only transactions of H removed. Let the version order $\ll$ be defined as follows: $x_i \ll x_j$ iff $t_i < t_j$. Properties STO1–STO5 have been defined so that any history H produced by MVTO algorithm satisfies them. According to Lemma 9.1, the multiversion serialization graph MVSG(H',$\ll$) is then acyclic. According to Lemma 9.2, if MVSG(H',$\ll$) is acyclic, then MVSG(H,$\ll$) is also acyclic. According to [6], a multiversion history H is one-copy serializable, if MVSG(H,$\ll$) is acyclic. □

9.5 Correctness of Index Operations

We have shown above that the concurrency control algorithms presented in this chapter allow only serializable histories. In the proofs, we have assumed that all index operations are performed correctly. However, although the correctness of concurrent index operations in a single B-tree is controlled by well-known concurrency control strategies, we must show that our algorithms assure the correct behavior when the index consists of two separate B-trees.

We first define what we mean by correct index operations.

Definition 9.3 *Consider a transaction system $\{T_1, T_2, \ldots, T_d\}$ and a history H of it. The index operations in executions of actions in H are* PERFORMED CORRECTLY, *if*

1. *For every i and k, if T_k reads x from T_i, then $r_k(x)$ returns the value written by $w_i(x)$ or "not found", if $w_i(x)$ is a delete action.*

2. *For every data item x written in H: if $w_i(x_i)$ is a final write of x in H and we add an extra operation $r(x)$ that is performed after all the actions of H are performed, then $r(x)$ returns the value written by $w_i(x)$ or "not found", if $w_i(x)$ is a delete action.*

When differential indexing is used, there are two possible problems: Either the reader may read an old version of the data item although there is a newer valid version of the same data item, or the reader may not find the data item at all, although it should be in the index. We explain next how these problems are avoided in our system.

When the merge is not operating, if there are any versions of the data item x in the differential file, all these versions are always newer than the version that is in the main index. All actions start by searching the differential file. So, they always find the newest versions first.

When the merge is running, the index operations must be performed carefully enough to prevent the following scenario: The merging process reads the node p containing data item x in the differential file. The merging process deletes x from p, because it wants to transfer x into the main index. After that, another action A_i searching for x reads p, but does not find x. A_i continues its search in the main index, reading the node P where x should be. But because the merge has not written x in the main index yet, A_i does not find x in P or it finds an older version of x. In the end, the merge writes x to P.

The problem with the scenario above is that A_i makes its reads both in the differential file and in the main index between the writes the merge makes in the trees. In our algorithms, the merge process prevents this by keeping the node p in the differential file exclusively locked until it has written the node P in the main index. Thus, if A_i

reads p after merge has removed x from p, then A_i reads also P after the merge has inserted x into P.

The standard concurrency control algorithms take care that other actions do not proceed to wrong nodes although the merge should divide or combine the nodes it writes.

9.6 Recovery

One of the main advantages of using differential files is the simple recovering scheme they allow. Because only committed transactions are transferred to the main index, no undo operations are needed. In addition, redo operations need only write to the main memory. This makes recovery fast.

We assume that the system uses write-ahead logging. That is, before the transaction commits, it writes redo records for all objects it has written into the log and the log is forced to the disk. We also assume that before the merge starts, identifiers of all transactions whose writes are transferred to the main index are written to the log. When the merge finishes, the end is reported to the log.

We use shadows checkpoint strategy for the main index. Checkpoint is a consistent state of the index that is guaranteed to be stored in the disk. The index tree nodes in a checkpoint are not overwritten until a new checkpoint is done. When an index tree node that belongs to an old checkpoint is changed, a new disk block is allocated for it, see Figure 9.2. Because the merge is the only process that updates the main index and the merge performs its updates in sorted order, it is easy to use shadows checkpoint strategy when a differential index is used. For a description of a B-tree recovery when differential indices are not used, see [21, 41, 42, 47].

If a system crash occurs, the recovery can be done as follows: A new differential file is made to contain writes of all transactions that have been committed but whose writes were still in the differential file or were part of the merge that was active during the crash. This can be easily done with help of the log. After that the system is ready to operate normally. The checkpoint of the main index is used as the main index. Nothing is done for transactions that were active during the crash. Their writes were only in the old differential file that was lost in the crash.

Next, we want to compare briefly recovery times of a database system using a differential index and a traditional database system where the redo operations have to be written to the disk. The recovery time consists of reading the log file from the disk, analyzing the log file and performing redo operations. If checkpoints in the traditional system are taken as often as merges are performed in a differential index based system, the time to read and analyze the log is about the same in both systems. (It may be a little longer in a differential index based system using the timestamp

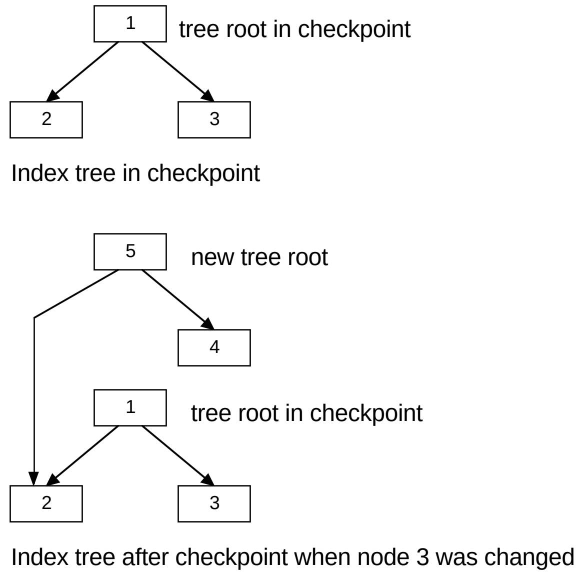


Figure 9.2: Checkpoint strategy. Numbers inside blocks are disk addresses.

based concurrency control strategy, because writes of some committed transactions cannot be transferred into the main index in the next merge following the commit.) However, each redo operation in the traditional system requires at least a disk read (to see if the write is on disk or not) whereas in the differential index based system it is enough to make an update in the main memory tree. Because the write in a main memory tree is about thousand times faster than a random disk access, this part of recovering is considerable faster in the system using differential index.

The shadows checkpoint strategy has some drawbacks in a concurrent environment. First, B^{link}-trees are shown to perform very efficiently in concurrent situations. When the shadows checkpoint mechanism is used, B^{link}-trees are not a practical choice. When a node is changed for the first time after the last checkpoint, a change also has to be made to all nodes that contain a pointer to that node. When a B^{link}-tree is used, most nodes are pointed to by two other nodes. Thus, one change in a leaf node could result in changes to numerous other nodes. Second, when the shadows checkpoint strategy is used, sometimes a large part of the B-tree has to be exclusively locked. For example, when the first change after the checkpoint is made, the whole path from the root to the leaf has to be locked. Every time a leaf is changed for the first time after the checkpoint, the parent of the leaf has to be locked. However, because the merge is the only process that makes updates in the main index and thus the only process that takes exclusive locks on the main index nodes, long lock waiting times are usually avoided.

Chapter 10

Conclusion

When there is a large number of updates into a B-tree, the usual insert and delete operations are not efficient enough, because each operation takes $O(\log N)$ disk accesses, where N is the number of the keys in the database. The performance of such an index can be improved by using batch-update operations. Some batch-update algorithms have been presented in the literature, but most of them do not consider concurrency control. However, there are applications like text database systems where the users want to be able to perform other operations, for example searches, during the batch update.

In this thesis, we have presented new algorithms for batch insertion such that it can be performed concurrently with other operations. We have also considered applications where batch update is useful, full-text indexing and a differential file system. First, we presented a batch-insertion algorithm for (a, b)-trees without concurrency and analyzed its time complexity. If a batch insertion of m sorted keys is to be performed and the group is partitioned into k subgroups such that the keys in each subgroup have the same position leaf, then the worst case time complexity of the algorithm (without the initial construction of subgroup trees in time $O(m)$) is $O(\Sigma_{i=1}^{k} \log m_i + |L|)$, where m_i denotes the size of the ith subgroup and $|L|$ denotes the total number of nodes that appear in search paths from the root to the position leaves. If individual insertions and deletions mixed with insertions of k subgroups are performed into an originally empty (a, b)-tree, then only a constant number of rebalancing operations is needed after an insertion and a deletion in the amortized sense and the number of rebalancing operations needed after an insertion of each subgroup is amortized logarithmic in the size of the group.

Next, we presented three concurrent batch-update algorithms for B-trees and gave experimental results on their behavior. Two of the algorithms, REL1 and REL2 use relaxed B-trees. The third algorithm LINK uses the $\mathrm{B}^{\mathrm{link}}$-tree. These algorithms are also compared with the algorithm TD that has been presented earlier in [67] and that uses top-down balancing.

Simulation results show that TD is efficient if the batch-update process works alone or the speed of the batch-update process is very important. REL1, REL2 and LINK are a little slower, but they allow a much higher degree of concurrency. The average times for concurrent searches are usually many times faster if REL1 REL2 or LINK are used for batch update than when TD is used. In addition, the average search times in connection with TD distinctly depend on the number of the nodes in the middle level of the B-tree being updated, while they are practically constant in connection with the other algorithms.

Finally, we presented a differential indexing system where the index is divided into two parts. The main index is on disk and a smaller differential index in the main memory. The active transactions perform their writes to the differential index. Periodically, writes of committed transactions are transferred from the differential index to the main index as a batch-update operation. This system offers a possibility for a simple timestamp based concurrency control strategy. Because all objects written by active transactions are in the differential index, the checks needed by the timestamp based concurrency control strategy can be limited to the differential index. The system also offers a simple recovery strategy after a system crash. No undo operations are needed in the main index and the redo operations can be performed on the differential index which is located in the main memory.

Bibliography

[1] L. Arge, K.H. Hinrichs, J. Vahrenhold, and J.S. Witter, Efficient bulk operations on dynamic R-trees. In: *Algorithm Engineering and Experimentation International Workshop ALENEX'99*, Lecture Notes in Computer Science 1619, Springer-Verlag, 1999, pp. 328–348.

[2] R. Bayer, The universal B-tree for multidimensional indexing: general concepts. In: *Proceedings of World-Wide Computing and Its Applications '97 (WWCA '97)*, Lecture Notes in Computer Science 1274, Springer-Verlag, 1997, pp. 198–209.

[3] R. Bayer and E. McCreight, Organization and maintenance of large ordered indexes. *Acta Informatica* **1** (1972), 173–189.

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

[5] J. van der Bercken, B. Seeger, and P. Widmayer, A generic approach to bulk loading multidimensional index structures. In: *Proceeding of the 23rd International Conference on Very Large Data Bases*, 1997, pp. 406–415.

[6] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[7] A.F. Cardenas, Analysis and performance of inverted data base structures. *Communications of the ACM* **18**:5 (1975), 253–263.

[8] M. Carey and C. Thompson, An efficient implementation of search trees on $\lceil lgN+1 \rceil$ processors. *IEEE Transactions on Computers* **C-33** (1984), 1038–1041.

[9] D. Comer, The ubiquitous B-tree. *ACM Computing Surveys* **11**:4 (1979), 121–138.

[10] D. Cutting and J. Pedersen, Optimizations for dynamic inverted index maintenance. In: *Proceedings of ACM SIGIR 1990, International Conference on Information Retrieval*, 1990, pp. 405–411.

[11] C.S. Ellis, Concurrent search and insertion in AVL trees. *IEEE Transactions on Computers* **C-29** (1980), 811–817.

[12] C.S. Ellis, Concurrent search and insertion in 2-3 trees. *Acta Informatica* **14** (1980), 63–86.

[13] T. Eiter, M. Schrefl, and M. Stumptner, Sperrverfahren für B-Bäume im Vergleich. *Informatik-Spektrum* **14** (1991), 183–200.

[14] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, The notions of consistency and predicate locks in a database system. *Communications of the ACM* **19** (1976), 624–633.

[15] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, Extendible hashing – a fast access method for dynamic files. *ACM Transactions on Database Systems* **4**:3 (1979), 315–344.

[16] C. Faloutsos, Access methods for text. *ACM Computing Surveys* **17**:1 (1985), 49–74.

[17] C. Faloutsos and S. Christodoulakis, Design of a signature file method that accounts for non-uniform occurrence and query frequencies. In: *Proceedings of the 11th International Conference on Very Large Data Bases*, 1985, pp. 165–170.

[18] C. Faloutsos and S. Christodoulakis, Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions On Office Information Systems* **2**:4 (1984), 267–288.

[19] C. Faloutsos and H.V. Jagadish, Hybrid index organizations for text databases. In: *Advances in Database Technology—EDBT'92, 3rd International Conference on Extending Database Technology*, Lecture Notes in Computer Science 580, Springer-Verlag, 1992, pp. 310–327.

[20] R. Fenk, A. Kawakami, V. Markl, R. Bayer, and S. Osaki, Bulk loading a data warehouse built upon a UB-tree. In: *Proceedings 2000 International Database Engineering and Applications Symposium*, 2000, pp. 179–187.

[21] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[22] L.J. Guibas and R. Sedgewick, A dichromatic framework for balanced trees. In: *Proceedings of the 19th IEEE Symposium on Foundations of Computer Science,* 1978, pp. 8–21.

[23] A. Guttman, R-trees: A dynamic index structure for spatial data access methods. In: *Proceedings of the ACM SIGMOD Conference*, 1985, pp. 47–57.

[24] S. Hanke and E. Soisalon-Soininen, Group updates for red-black trees. In: *Proceedings of the 4th Italian Conference on Algorithms and Complexity*, Lecture Notes in Computer Science 1767. Springer-Verlag, 2000, pp. 253–262.

[25] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica* **17** (1982), 157–184.

[26] L. Jacobsen, K.S. Larsen, and M.N. Nielsen, On the existence and construction of non-extreme (a,b)-trees. Techincal report PP-2001-11, Department of Mathematics and Computer Science, University of Southern Denmark, Odense, 2001.

[27] C. Jermaine, A. Datta, and E. Omiecinski, A novel index supporting high volume data warehouse insertions. In: *Proceedings of the 25th International Conference on Very Large Data Bases,* 1999, pp. 235–246.

[28] T.J. Johnson, The performance of concurrent data structure algorithms. Ph.D. Thesis, New York University, 1990.

[29] T.J. Johnson and D. Shasha, The performance of current B-tree algorithms. *ACM Transactions on Database Systems* **18**:1 (1993), 51–101.

[30] D. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley Publishing Company, Reading, Massaschusetts, 1973.

[31] H.T. Kung and J.T. Robinson, On optimistic methods for concurrency control. *ACM Transactions on Database Systems* **6**:2 (1981), 213–226.

[32] T.-W. Kuo, C-H. Wei, and K.-Y. Lam. Real-time data access control on B-tree index structures. In: *Proceedings of the 15th International Conference on Data Engineering.* IEEE Computer Society, 1999, pp. 458–467.

[33] W.J. Labio, Y. Zhuge, J.L. Wiener, H. Gupta, H.Garcia-Molina, and J.Widom, The WHIPS prototype for data warehouse creation and maintenance. In *Proceedings of the ACM SIGMOD Conference*, 1997, pp. 557–559.

[34] S.D. Lang, J.R. Driscoll, and J.H. Jou, Batch insertion for tree structured file organizations—improving differential database representation. *Information Systems* **11**:2 (1986), 167–175.

[35] V. Lanin and D. Shasha, A symmetric concurrent B-tree algorithm. In: *Proceedings of the Fall Joint Computer Conference,* 1986, pp. 380–389.

[36] K.S. Larsen. Relaxed multi-way trees with group updates. In: *Proceedings of the 20th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM Press, 2001, pp. 93–101.

[37] K.S. Larsen. Relaxed red-black trees with group updates. Techincal report PP-2001-12, Department of Mathematics and Computer Science, University of Southern Denmark, Odense, 2001.

[38] K.S. Larsen and R. Fagerberg. B-trees with relaxed balance. In: *Proceedings of the 9th International Parallel Processing Symposium*, IEEE Computer Society Press, 1995, pp. 196–202.

[39] K.S. Larsen and R. Fagerberg, Efficient rebalancing of B-trees with relaxed balance. *International Journal of Foundations of Computer Science* **7**:2 (1996), 169–186.

[40] P. Lehman and S. Yao, Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems* **6**:4 (1981), 650–670.

[41] D. Lomet and B. Salzberg, Access method concurrency with recovery. In: *Proceedings of ACM SIGMOD'92 International Conference on Management of Data*, 1992, pp. 351–360.

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

[43] L. Malmi and E. Soisalon-Soininen. Group updates for relaxed height-balanced trees. In: *Proceedings of the 18th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM Press, 1999, pp. 358–367.

[44] K. Mehlhorn. *Data Structures and Algorithms, Vol. 1: Sorting and Searching,* Springer-Verlag, 1986.

[45] A. Moffat, O. Petersson, and N.C. Wormald, A tree-based mergesort. *Acta Informatica* **35** (1998), 775–793.

[46] C. Mohan, ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operation on B-tree indexes. In: *Proceedings of the 16th International Conference on Very Large Data Bases*, 1990, pp. 392–405.

[47] C. Mohan and F. Levine, ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In: *Proceedings of ACM SIGMOD'92 International Conference on Management of Data*, 1992, pp. 371–380.

[48] C. Mohan and I. Narang, Algorithms for creating indexes for very large tables without quiescing updates. In: *Proceedings of ACM SIGMOD'92, International Conference on Management of Data,* 1992, pp. 361–370.

[49] Y. Mond and Y. Raz, Concurrency control in B^+-trees databases using preparatory operations. In: *Proceedings of the 11th International Conference on Very Large Data Bases,* 1985, pp. 331–334.

[50] O. Nurmi, E. Soisalon-Soininen, and D. Wood, Concurrency control in database structures with relaxed balance. In: *Proceedings of the Sixth ACM Symposium on Principles of Database Systems,* 1987, pp. 170–176.

[51] J.A. Orenstein and T.H. Merrett. A class of data structures for associative searching. In *Proceedings of the Third ACM Symposium on Principles of Database Systems,* 1984, pp. 181–190.

[52] K. Pollari-Malmi, J. Ruuth, and E. Soisalon-Soininen, Concurrency control for B-trees with differential indices. In: *Proceedings of 2000 International Database Engineering and Applications Symposium,* 2000, pp. 287–295.

[53] K. Pollari-Malmi, E. Soisalon-Soininen, and T. Ylönen, Concurrency control for B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering* **8**:6 (1996), 975–984.

[54] M. Rossi, Concurrent full text database. Master's Thesis, Helsinki University of Technology, Department of Computer Science, 1997.

[55] J. Ruuth, Database indexing system prototype. Master's Thesis, Helsinki University of Technology, Department of Computer Science, 1994.

[56] Y. Sagiv, Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences* **33**:2 (1986), 275–296.

[57] L. Sha, R. Rajkumar, and J.P. Lehoczky, Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers* **39**:9 (1990), 1175–1185.

[58] D. Shasha and N. Goodman, Concurrent search structure algorithms. *ACM Transactions on Database Systems* **13**:1 (1988), 53–90.

[59] S. Sippu and E. Soisalon-Soininen, A theory of transactions on recoverable search trees. In: *Proceedings of the 8th International Conference of Database Theory – ICDT 2001*, Lecture Notes in Computer Science 1973, Springer-Verlag, 2001, pp. 83–98.

[60] V. Srinivasan and M.J. Carey, On-line index construction algorithms. In: *Proceedings of High Performance Transaction Systems Workshop*, 1991.

[61] V. Srinivasan and M.J. Carey, Performance of on-line index construction algorithms. In: *Advances in Database Technology—EDBT'92, 3rd International Conference on Extending Database Technology*, Lecture Notes in Computer Science 580, Springer-Verlag, pp. 293–309.

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

[63] J. Srivastava and C.V. Ramamoorthy, Efficient algorithms for maintenance of large database indexes. In: *Proceedings of the Fourth International Conference on Data Engineering*, 1988, pp. 402–409.

[64] R.E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods* **6** (1985), 306–318.

[65] I. Witten, A. Moffat, and T. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, New York.

[66] M. Yannakakis, Serializability by locking. *Journal of the ACM* **31**:2 (1984), 227–244.

[67] T. Ylönen, An algorithm for full-text indexing. Master's Thesis, Helsinki University of Technology, Department of Computer Science, 1992.

[68] J. Zobel, A. Moffat, and K. Ramamohanarao, Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems* **23**:4 (1998), 453–490.