Concurrency Control and I/O-Optimality in Bulk Insertion

Kerttu Pollari-Malmi and Eljas Soisalon-Soininen

Department of Computer Science and Engineering
Helsinki University of Technology, P.O.Box 5400, FIN-02015 HUT, Finland
{kerttu,ess}@cs.hut.fi

Abstract. In a bulk update of a search tree a set of individual updates (insertions or deletions) is brought into the tree as a single transaction. In this paper, we present a bulk-insertion algorithm for the class of (a, b)-trees (including B^+-trees). The keys of the bulk to be inserted are divided into subsets, each of which contains keys with the same insertion place. From each of these sets, together with the keys already in the insertion place, an (a, b)-tree is constructed and substituted for the insertion place. The algorithm performs the rebalancing task in a novel manner minimizing the number of disk seeks required. The algorithm is designed to work in a concurrent environment where concurrent single-key actions can be present.

1 Introduction

Bulk insertion is an important index operation, for example in document databases and data warehousing. Document databases usually apply indices containing words and their occurrence information. When a new document is inserted into the database, a bulk insertion containing words in this document will be performed. Experiments of a commercial system designed for a newspaper house in Finland [15] have shown that a bulk insertion can be up to two orders of magnitude faster than the same insertions individually performed.

Additions to large data warehouses may number in the hundreds of thousands or even in the millions per day, and thus indices that require a disk operation per insertion are not acceptable. As a solution, a new B-tree like structure for indexing huge warehouses with frequent insertions is presented in [8]. This structure is similar to the buffer tree structure of [1, 2]; the essential feature is that one advancing step in the tree structure always means a search phase step for a set of several insertions.

In this paper, we consider the case in which the bulk, i.e., the set of keys to be inserted, fits into the main memory. This assumption is reasonable in most applications. Only some extreme cases of frequent insertions into warehouses do not fulfil this requirement. We present a new bulk-insertion algorithm, in which the possibility of concurrent single-key operations are taken into account. The bulk insertion is performed by local operations that involve only a constant number of nodes at a time. This makes it possible to design efficient concurrency

A. Apostolico and M. Melucci (Eds.): SPIRE 2004, LNCS 3246, pp. 161–170, 2004.

control algorithms, because only a constant number of nodes need be latched at a time. Allowing concurrent searches is vital in document databases [15] and in www-search-engine applications.

The search trees to be considered are a class of multi-way trees, called (a, b)-trees [6, 12]. The class of (a, b)-trees is a generalization of B^+-trees: in an (a, b)-tree, $b \geq 2a - 1$, a and b denote the minimum and the maximum number of elements in a node. The trees considered are external, i.e., keys are stored in the leaves and internal nodes contain routing information.

For our model of I/O-complexity we assume that each node of the tree is stored in one disk page. We assume that the current path from the root to a leaf (or the path not yet reached a leaf but advancing towards a leaf) is always found in the main memory, but otherwise accessing a node requires one I/O-operation. Moreover, in our model we count writing (or reading) of several consecutive disk pages as one I/O-operation. This is justified whenever the number of consecutive pages is "reasonable" because the seek time has become a larger and larger factor in data transfer to/from disk [17]. In our paper this property of the model comes into use when a portion of the bulk goes into the same leaf, and this (usually a relatively small) part of the bulk will be written on disk.

2 General Bulk Insertion

In a *level-linked* (a, b)*-tree* [6, 12], $a \geq 2, b \geq 2a - 1$, all paths from the root to a leaf have the same length. The leaves contain at least a and at most b keys, and, similarly, the internal nodes have at least a and at most b children. The root of the tree is an exception and has at least 2 and at most b children. In leaves each key is coupled with a data record (or with a pointer to data). An internal node v with n children is of the form

$$(p_0)(r_1, p_1)(r_2, p_2) \ldots (r_n, p_n)(r_{n+1}, p_{n+1}),$$

where for $i = 1, \ldots, n$, p_i is the pointer to the ith child of v. This ith child is the root of the subtree that contains the keys in the interval $(r_i, r_{i+1}]$. Values r_i, $1 \leq i \leq n + 1$, in an internal node are called *routers*. We say that node v *covers* the interval $(r_1, r_{n+1}]$.

Router r_1 is smaller than any key in the subtree rooted at v, called the *lowvalue* of node v, denoted *lowvalue*(v), and router r_{n+1} is the largest possible key value in this subtree, called the *highvalue* and denoted *highvalue*(v). Pointer p_0 points to the node that precedes, and p_{n+1} points to the node that follows node v at the same level. If node v is the parent of a leaf l and pointer $p_i, i = 1, \ldots, n$, in v points to l, then the *lowvalue* of leaf l is r_i and the *highvalue* is r_{i+1}.

The basic idea of our I/O-optimal bulk-insertion algorithm is that the keys of the bulk sorted in the main memory will efficiently be divided into subsets, each of which contains keys that have the same insertion place (which is a node in the leaf level). From each of these sets, called *simple bulks*, together with the keys already in the insertion place, an (a, b)-tree, called an *insertion tree*, is constructed and substituted for the insertion place. After this process, called *bulk insertion without rebalancing*, has been completed, the structure contains

all keys of the bulk and can already be used as a search tree with logarithmic search time. In order to retain the (a, b)-tree properties the structure needs, of course, rebalancing.

Moreover, we aim at a solution where concurrency is allowed and concurrency control is *efficient* in the sense that each process will latch only a constant number of nodes at a time and that a latch on a node is held only for a constant time. The concurrency control needed before rebalancing is simple latch coupling in the same way as for single-key updates (insertions or deletions). The efficient latch coupling in the search phase of an update (bulk or single) applies *might-write* latches, which exclude other updates but allow readers to apply their shared latches. *Shared* or *read* latches applied by readers exclude only the exclusive latches required on nodes to be written.

Given an (a, b)-tree T and a bulk with m keys, denoted $k_1, \ldots, k_m$, in ascending order, the bulk insertion into T without rebalancing works as follows.

Algorithm BI (Bulk Insertion)

Step 1. Set $i = 1$, and set $p =$ the root of T.

Step 2. Starting at node p search for the insertion place l_i of key k_i. Push each node in the path from node p to l_i onto stack S. In the search process apply latch coupling in the might-write mode. When leaf l_i is found, the latch on it will be upgraded into an exclusive latch. The latch on the parent of l_i will be released.

Step 3. Let k_{i+j} be the largest key in the input bulk that is less than or equal to the highvalue of l_i. From the keys $k_i, \ldots, k_{i+j}$ together with the keys already in l_i, an (a, b)-tree B_i is constructed and substituted for l_i in T. This will be done by storing the contents of the root of B_i into node l_i, so that no changes is needed in the parent of l_i. Release the latch on the node that is now the root of B_i.

Step 4. Set $i = i + j + 1$. If $i > m$, then continue to Step 5. Otherwise pop nodes from stack S until the popped node p covers the key k_i. If such a node is not found in the stack (this may occur if the root has been split after the bulk insertion started), set p as the new root. The nodes which are popped from the stack are latched in the shared mode, but latch-coupling is not used. Return to Step 2.

Step 5. Now all insertion positions have been replaced by the corresponding insertion trees. Rebalance the constructed tree by performing Algorithm SBR (given below) for each B_i in turn.

If concurrent single-key updates are allowed, it may happen that in Step 4 the algorithm must return even to the root although in the original tree only a few steps upward would have been enough in order to find the node from which to continue.

The algorithm composed of the first 4 steps of the above algorithm is called Algorithm BIWR (Bulk Insertion Without Rebalancing). In the following discussion of the complexity of Algorithm BIWR we assume that the concurrency is limited to concurrent searches.

If searching must be done in the standard way, that is, only pointers from parents to children are followed, it is clear that Algorithm BIWR is optimal as to nodes visited in T and thus in the number of nodes accessed. This is because each node in the paths from the root to the insertion positions must be accessed at least once, and this is exactly what the above algorithm does. If parent links together with level links are applied, a better performance can be obtained in some special cases, but from results in [3] it is straightforward to derive that no asymptotic improvement can be obtained.

The insertion tree B_i is constructed in the main memory, but I/O-operations are needed in writing it on disk as a part of the whole tree. For doing this only one disk seek is needed and in our model thus only one operation. We have:

Theorem 1. *Let T be an (a, b)-tree, and assume that a bulk of m keys is inserted into T by Algorithm BIWR (Bulk Insertion Without Rebalancing, the first four steps of Algorithm BI). Then the resulting tree (which has logarithmic depth but does not fulfil the (a, b)-balance conditions) contains exactly the keys that were originally in T or were members of the bulk. The I/O-complexity of Algorithm BIWR is*

$$\Theta(k + L) = \Theta(L),$$

where k denotes the number of insertion trees B_i constructed in Step 3 of the algorithm and L denotes the number of different nodes that appear in the paths from the root of the original tree T to the insertion places l_i.

3 Rebalancing

Our next task is to perform rebalancing. Our solution for rebalancing is designed such that concurrent searches and single-key updates are possible.

We assume first that we are given a situation in which the whole bulk to be inserted has the same insertion place l_1, and Algorithm BIWR has produced a new tree T, in which l_1 has been replaced by an insertion tree B_1 (Step 3 in Algorithm BIWR). Before we can start the rebalancing task we must have obtained a shared lock on the key interval $[k, k']$, where k and k', respectively, are the smallest and the largest key in B_1. This lock is requested in Step 3 in Algorithm BIWR before the replacement of l_1 by B_1 can take place. This guarantees that no updates that would affect B_1 could occur during rebalancing, provided that performing updates requires obtaining an exclusive lock on the key to be inserted or deleted, see e.g. [9]. (The locks are not the same as the latches; latches are for physical entities of a database, and locks for logical entities. Latches are short duration semaphores, and locks are usually held until the commit of the transaction involved.)

Now if B_1 contains one leaf only, we are done, and the lock on $[k, k']$ can be released. Otherwise, we perform the *simple bulk rebalancing* in the following way.

Algorithm SBR (Simple Bulk Rebalancing)

Step 1. Latch exclusively the parent of the root of B_1 and denote the latched node by p. Set $h = 1$.

Step 2. Split node p such that the left part contains all pointers to children that store keys smaller than the smallest key in B_1, and the right part all pointers to children that store keys larger than the largest key in B_1. Denote the nodes thus obtained by p_l and p_r. Observe that both p_l and p_r exist; in the extreme case node p_l contains only the lowvalue and the level link to the left and p_r only the highvalue and the level link to the right. In all cases p is set to p_l; that is, p_l is the node that remains latched, and $p = p_l$ does not point to the root of B_1 (or its ancestor) any more. Moreover, notice that neither p_l nor p_r can contain more than b elements, even though, when returning from Step 3, p could contain $b + 1$ elements. See Fig. 1 for illustration.

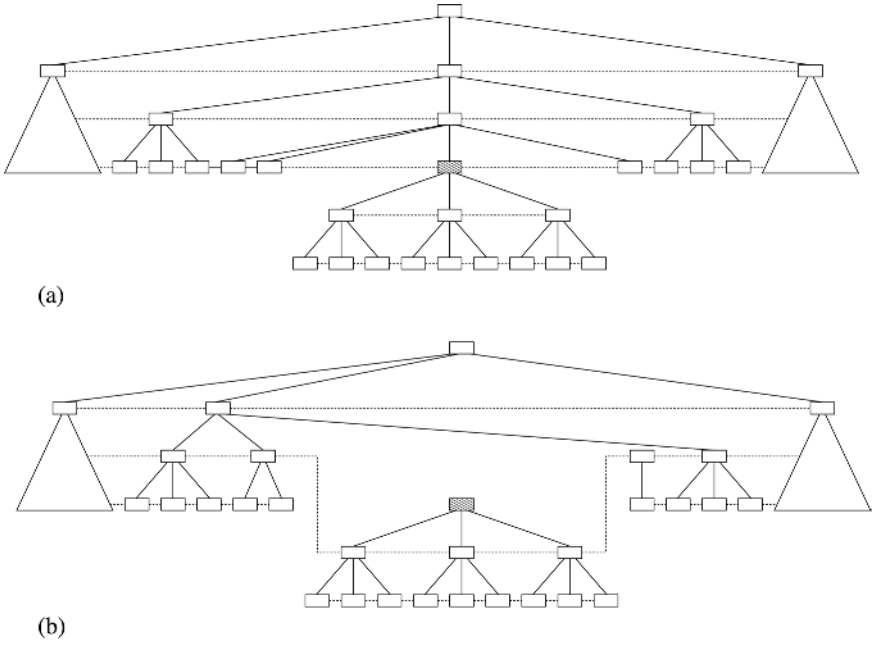

Fig. 1. Splitting the parent of the root of the insertion tree, $a = 2$ and $b = 4$. (a) Original tree. The root of the insertion tree is shaded. (b) Split tree with updated level links at height 1.

Exclusively latch p_r, and the leftmost and rightmost nodes, denoted q_l and q_r, respectively, at the height h in B_1. Then compress (by applying fusing or sharing) node p_l together with node q_l, node p_r together with q_r, and also adjust the level links appropriately. (The nodes in T and nodes in B_1 at height h are all linked together by level links and no violations against the (a, b)-tree property occur in these nodes.) Release all latches held.

Step 3. Set h = h + 1. At height h in T latch exclusively the node, denoted p, that has lowvalue smaller than the smallest key in B_1 and highvalue larger than the largest key in B_1. If in Step 2 node p_r, one level below, was not fused

with q_r but remained (perhaps shortened because of sharing), add this node as a new child to p. For the moment, allow p grow one too large, if necessary. If h is smaller than the height of B_1, then return to Step 2.

Step 4. Now the insertion tree B_1 has been properly level-linked with the rest of the tree, with the exception of the root and the leaf level. As in Step 2, split node p appropriately into p_l and p_r such that the root of B_1 can be put between them. The nodes at the leaf level which thus far has been level-linked to the root of B_1 must now be level-linked with the leftmost and, respectively, the rightmost leaf node in B_1. For the operation, all changed nodes must be exclusively latched. At the end, all latches held are released.

Step 5. The whole insertion tree B_1 has now been correctly level-linked, but it might be that the root of B_1 and its right brother have no parent, that is, they can be reached only by level links and not by child links from their parents. (See Fig. 2.) Thus rebalancing is still needed above the root of B_1, and the need of splits may propagate up to the root of the whole tree. In a concurrent environment this remaining rebalancing can be done exactly as for single inserts in $\mathrm{B}^{\mathrm{link}}$-trees [16]. Figure 3 shows the final rebalanced tree.

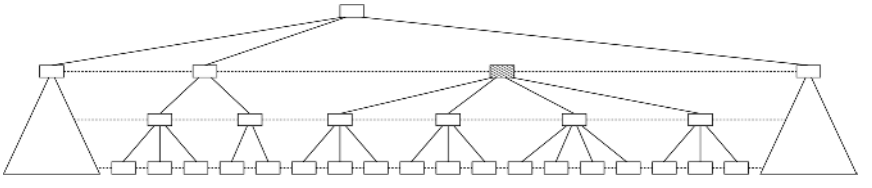

Fig. 2. Tree of Fig. 1 after the insertion tree has been correctly level-linked.

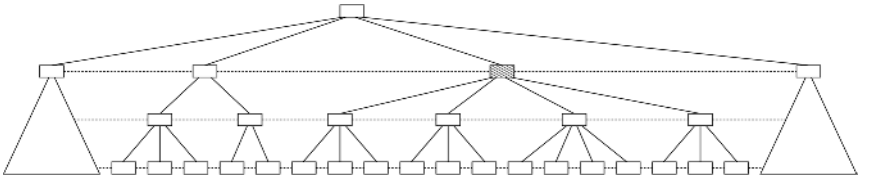

Fig. 3. Tree of Fig. 2 when bulk rebalancing has been finished.

It is important to note that, for rebalancing, we cannot simply cut the tree T starting from the insertion place up to the height of B_1, and then lift B_1 to its right position. Such an algorithm would need too much simultaneous latching in order to set the level links correctly. The level links are essential because they guarantee the correctness of concurrent searching at all times. The simpler solution to "merge" B_1 with T by cutting T at l_1, joining the left part with B_1, and joining the result with the right part [13] is not applicable in a concurrent environment, either.

First, for the correctness and complexity (Theorem 2–6), we consider Algorithm SBR in an environment, where only concurrent searches are allowed.

Notice that then node p as specified in Step 2 is directly obtained from the stack of nodes constructed in the search phase of Algorithm BI.

The following theorem is immediate. Notice that in Step 3 of Algorithm SBR compressing of nodes as described is always possible because the insertion tree B_1 is in balance. Compressing two nodes means, in the same way as in standard B-tree rebalancing, that two nodes are either made as one node (fusing) or their contents are redistributed (sharing) such that both nodes meet the (a, b)-tree conditions.

Theorem 2. *Let T be a tree yielded by Algorithm BIWR such that from the inserted bulk of size m only one insertion tree was constructed. Algorithm SBR (Simple Bulk Rebalancing) rebalances T, that is, yields an (a, b)-tree that contains exactly the keys originally in T. The worst case I/O-complexity of Algorithm SBR (when only concurrent searches are present) is $\Theta(\log m)$ (Steps 1–3), plus $\Theta(\log n)$ (Step 4), where n denotes the size of T.*

Notice that the worst case complexity $\Theta(\log n)$ of Step 5 comes from the fact that nodes above the root of B_1 may be full; this worst case may occur also for single insertions. Thus, and because it may be necessary to split h nodes, where h denotes the height of B_1, the above algorithm is asymptotically optimal.

Step 5 in Algorithm SBR can be considered as an elimination of an $b + 1$- or $b + 2$-node (node that contains $b + 1$ or $b + 2$ elements) from the tree. This is because the parent of the root of B_1 can have got one or two new children. But, as shown in [5], elimination of a $b+1$-node takes amortized constant time, provided that $b \geq 2a$. (By *amortized time* we mean the time of an operation averaged over a worst-case sequence of operations starting with an empty structure. See [12, 18].) The same holds, of course, for the elimination of a $b + 2$-node. Thus Theorem 2 implies:

Theorem 3. *Let T be a tree yielded by Algorithm BIWR such that from the inserted bulk of size m only one insertion tree was constructed. Algorithm SBR (Simple Bulk Rebalancing) rebalances T, that is, yields an (a, b)-tree that contains exactly the keys originally in T. The amortized I/O-complexity of Algorithm SBR (when only concurrent searches are present) is $\Theta(\log m)$.*

The result of Theorem 3 requires that each B_i in Algorithm BI is constructed so that at most two nodes at each level of B_i 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 [7].

Assume that a bulk insertion of m keys without rebalancing has been applied to an (a, b)-tree yielding a tree denoted by T. Assume that the bulk was divided into k insertion trees, denoted $B_1, B_2, \ldots, B_k$. Rebalancing T, that is, the final step of Algorithm BI, can now be performed by applying Algorithm SBR for $B_1, B_2, \ldots$, and B_k, in turn. The cost of rebalancing includes (i) the total cost of Steps 1–3 of Algorithm SBR for $B_1, \ldots, B_k$ and (ii) the total cost of rebalancing (Step 4) above node p_i that has become the parent of B_i, $i = 1, \ldots, k$. Part (i) has I/O-complexity $O(\Sigma_{i=1}^{k} \log m_i)$, where m_i denotes the size of B_i, and part (ii) has the obvious lower bound $\Omega(L)$, where L denotes the number of different

nodes appearing in the paths from the root to the insertion places in the original tree. It is easy to see that $O(L + \Sigma_{i=1}^{k} \log m_i)$ bounds from above part (ii). Of those nodes that are full before the rebalancing starts only L can be split because of rebalancing. Rebalancing of one B_i cannot produce more than $O(\log m_i)$ new full nodes that may need be split by rebalancing a subsequent B_j. Thus the total number by splits and also the number of I/Os needed for the whole rebalancing task is $O(L + \Sigma_{i=1}^{k} \log m_i)$.

We have:

Theorem 4. *Assume that a bulk insertion without rebalancing has been applied to an (a, b)-tree, and assume that the bulk was divided into k insertion trees with sizes $m_1, m_2, \ldots, m_k$. Then the worst case I/O-complexity of rebalancing (the final step of Algorithm BI), provided that only concurrent searches are present, is*

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

where L is number of different nodes in the paths from the root to the insertion places (roots of the insertion trees) before the rebalancing starts.

For the amortized complexity we have:

Theorem 5. *Assume that a bulk insertion without rebalancing has been applied to an (a, b)-tree, $b \geq 2a$, and assume that the bulk was divided into k insertion trees with sizes $m_1, m_2, \ldots, m_k$. Then the amortized I/O-complexity of rebalancing, provided that only concurrent searches are present, is*

$$\Theta(\Sigma_{i=1}^{k} \log m_i).$$

Theorems 1 and 4 imply:

Theorem 6. *The worst case I/O-complexity of a bulk insertion into an (a, b)-tree is*

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

where m_i is the size of the ith simple bulk and L is the number of different nodes in the paths from the root to the insertion places.

Our concurrent algorithm is meant to be used together with key searches that do not change the structure and with single-key operations. The pure searches are the most important operations that must be allowed together with bulk insertion. This is certainly important for www search engines, and it was vital for the commercial text database system reported in [15]. For the correctness, the issues to be taken care of are that no search paths (for pure searches or the search phases of insertions or deletions) cannot get lost, and that the possible splits or compress operations performed by concurrent single-key actions do not cause any incorrectness.

Because a shared lock on the key interval of the insertion tree must have been obtained before rebalancing, no changes in the interval trees caused by concurrent processes can occur during the bulk insertion. Thus the only possibility for

incorrectness (due to bulk insertion) is that a search path gets lost when a node above the insertion tree is split and the search would go through this node and end in a leaf of the insertion tree B_i, or in a leaf right to B_i, cf. Fig. 1 (b). But when this kind of a split occurs, the exclusive latches as defined in Step 3 of the algorithm prevents the search path losses. After Step 3 has been completed, all leaves of B_i, and all leaves to the right of B_i that have lost their parent path to the root (in Fig. 1 (b) one leaf to the right of the insertion tree) are again reachable because of the level links set. The possible splits or compress operations of concurrent single-key actions imply the possibility that the nodes of the search path to the insertion place pushed on stack are not always parents of the split nodes. Thus the parent must be searched, see Step 3, by a left-to-right traverse starting from the node that is popped from the stack. (Cf. [16].)

The pure searches and the search phases of single-key actions and the search phase of the bulk insertion apply latch coupling in the appropriate mode, and all changes in nodes are made under an exclusive latch, which prevents all other possible path losses.

We have:

Theorem 7. *The concurrent algorithm BI and standard concurrent searches and concurrent single-key actions all applied to the same level-linked* (a, b)*-tree run correctly with each other.*

4 Conclusion

We have presented an I/O-optimal bulk insertion algorithm for (a, b)-trees, a general class of search trees that include B-trees. Some ideas of the new algorithm stem from earlier papers on bulk updates [11, 15]. The new aspect in the present paper is that we couple efficient concurrency control with an I/O-optimal algorithm, and the I/O-complexity is carefully analyzed in both worst case and amortized sense.

The same amortized time bound has been proved for relaxed (a, b)-trees in [10], but with linear worst case time. In addition, although [10] gives operations to locally decrease imbalance in certain nodes, it does not give any deterministic algorithm to rebalance the whole tree after group insertion. Algorithms based on relaxed balancing also have the problem that they introduce new almost empty nodes at intermediate stages.

The idea of performing bulk insertion by inserting small trees [14] is independently presented for R-trees in [4]. In [4] concurrency control is not discussed, whereas our main contribution is to introduce efficient concurrency control into I/O-optimal bulk insertion.

Our method of bulk rebalancing can also be applied for buffer trees [1, 2]. Buffer trees are a good choice for efficient bulk insertion in the case in which the bulk is large and does not fit into the main memory.

References

1. L.Arge. The buffer tree: a technique for designing batched external data structures. *Algorithmica* **37** (2003), 1–24.

2. L.Arge, K.H.Hinrichs, J.Vahrenhold, and J.S.Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica* **33** (2002), 104–128.
3. M.R.Brown and R.E.Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal of Computing* **9** (1980), 594–614.
4. L.Chen, R.Choubey, and E.A.Rundensteiner. Merging R-trees: Efficient strategies for local bulk insertion. *GeoInformatica* **6** (2002), 7–34.
5. K.Hoffmann, K.Mehlhorn, P.Rosenstiehl, and R.E.Tarjan: Sorting Jordan sequences in linear time using level-linked search trees. *Information and Control* **68** (1986), 170–184.
6. S.Huddleston and K.Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica* **17** (1982), 157–184.
7. L.Jacobsen, K.S.Larsen, and M.N.Nielsen. On the existence and construction of non-extreme (a,b)-trees. *Information Processing Letters* **84** (2002), 69–73.
8. C.Jermaine, A.Datta, and E.Omiecinski. A novel index supporting high volume data warehouse insertion. In: *Proceedings of the 25th International Conference on Very Large Databases*. Morgan Kaufmann Publishers, 1999, pp. 235–246.
9. M.Kornacker, C.Mohan, and J.M.Hellerstein. Concurrency and recovery in generalized search trees. In: *Proceedings of the 1997 SIGMOD Conference, SIGMOD Record* **26**. ACM Press 1997, pp. 62–72.
10. K.S.Larsen. Relaxed multi-way trees with group updates. *Journal of Computer and System Sciences* **66** (2003), 657–670.
11. 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.
12. K.Mehlhorn. *Data Structures and Algorithms, Vol. 1: Sorting and Searching,* Springer-Verlag, 1984.
13. A.Moffat, O.Petersson, and N.C.Wormald. A tree-based mergesort. *Acta Informatica* **35** (1998), 775–793.
14. K.Pollari-Malmi. Batch updates and concurrency control in B-trees. Ph.D.Thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Report A38/02, 2002.
15. K.Pollari-Malmi, E.Soisalon-Soininen, and T.Ylönen. Concurrency control in B-trees with batch updates. *IEEE Transactions on Knowledge and Data Engineering* **8** (1996), 975–984.
16. Y.Sagiv. Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences* **33** (1986), 275–296.
17. Y.Tao and D.Papadias. Adaptive index structures. In: *Proceedings of the 28th Conference on Very Large Data Bases*. Morgan Kaufmann Publishers, 2002, pp. 418–429.
18. R.E.Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods* **6** (1985), 306–318.