An Algorithm for Full Text Indexing

Tatu Ylönen

Helsinki University of Technology
Laboratory of Information Processing Science
SF-02150 Espoo, Finland
E-mail: ylo@cs.hut.fi

February 27, 1992

Master's Thesis

Abstract

A fast B-tree based indexing algorithm is presented. In some applications, such as full text indexing or indexing of very large tables, the new algorithm can be orders of magnitude faster than conventional B-tree insertion algorithms, while still allowing concurrent access. A similar algorithm can be used for deletion.

The algorithm works by collecting in memory thousands or hundreds of thousands of keys to be added. The keys are sorted and multiple occurrences of a key are collected together. The keys in memory and the B-tree on disk are then merged. These phases are repeated until all data has been indexed.

Experimental results suggest that indexing full-text data using the new algorithm requires only 0.01 to 0.001 disk accesses per key, even for large databases. CPU time usage per key is also reduced compared to conventional B-tree algorithms.

Contents

1 Introduction **1**
1.1 Full text retrieval as an information retrieval method . . . 1
1.2 Full text indexing methods . . . 1
1.3 B-tree indexing in full text retrieval systems . . . 3
1.4 B-trees . . . 4
1.4.1 History . . . 4
1.4.2 Definition of the B-tree . . . 4
1.4.3 B*-trees . . . 6
1.4.4 B+-trees . . . 7
1.4.5 B-trees with variable length keys . . . 9
1.4.6 Concurrency control: Bottom-up algorithms (Bayer–Schkolnick) . . . 9
1.4.7 Concurrency control: Preparatory operations (Top-down insertion) . . . 9
1.4.8 Concurrency control: B^link-trees . . . 10
1.4.9 Concurrency control: Relaxed balance (Nurmi–Soisalon-Soininen–Wood) . . . 12
1.4.10 Comparison of concurrency control methods . . . 12
1.4.11 Alternatives for storing occurrence data . . . 13
1.4.12 Optimizations: Disk cache . . . 14
1.4.13 Optimizations: Group update . . . 14
1.4.14 Optimizations: Prefix omission . . . 15
1.5 The new insertion algorithm . . . 15
1.6 Application of the new algorithm to deletion . . . 15
1.7 Scientific results . . . 16
1.8 Topics for further research . . . 16

2 The new algorithm **17**
2.1 Overview . . . 17
2.2 Buffering data in memory . . . 18
2.3 Merging the keys in memory to the B-tree on disk . . . 18
2.3.1 Finding a leaf node, starting from the root . . . 20
2.3.2 Locating the node where the next key should be inserted . . . 21
2.3.3 Inserting a key to a leaf node . . . 21
2.3.4 Continuation records for occurrence data . . . 22
2.3.5 Optimizing multiple insertions to a node . . . 22
2.3.6 Optimizing index file size . . . 24

2.3.7 File locking and concurrency 26
2.3.8 Fault Recovery 27

3 Description of the current implementation 28
3.1 Overview 28
3.2 Interfaces 28
3.3 The in-memory data structure 29
3.4 Overview of merge implementation 29
3.5 Performance 29
3.6 Commercial references 32

4 Conclusion 33

A Pseudocode listing of the implementation 34
A.1 Variables 34
A.2 Functions 36

List of Figures

1 The B-tree can be thought of as a binary tree where many nodes have been grouped together. 5
2 The B-tree data structure ($m = 4$). 5
3 The B^+-tree data structure ($m = 4$). 7
4 The B^+-tree insertion algorithm. 8
5 The B^+-tree deletion algorithm. 9
6 The B^{link}-tree data structure ($m = 3$). 10
7 Index file structure. 17
8 The new insertion algorithm. 19
9 Finding the leaf node and splitting non-leaf nodes. 20
10 Finding the node where the next key belongs. 21
11 Structure of a key in index file. 28

List of Tables

1 Indexing performance. 30
2 The effect of buffer memory size on indexing speed (10MB English corpus). 31

Acknowledgements

I would like to thank my professor, Elias Soisalon-Soininen, for his help and suggestions. His long-term experience with B-trees and database technology provided me with many valuable references, pointers, and ideas.

I would also like to thank all those who have helped me on the field of information retrieval. Pasi Tyrväinen and Petteri Saarinen of Nokia Research Center have kept me up to date with many developments on the field during the last few years, and introduced me to many new approaches, including the SIMPR project [24] and many other commercial and research systems. Kalervo Järvelin and Eero Sormunen of University of Tampere have provided me with many interesting articles.

I would like to thank all those who have participated in building, selling, buying and using the commercial versions of my system. They have made this work financially possible, and have provided me with lots of real-world expierience, test material and ideas. Kari Hiekkanen and Jukka Partanen (NGS), Jukka Rekula, Pekka Helle and Tuomo Telkkä (Monigraaf, now part of Siemens-Nixdorf), Osmo Moisio (State Printing Centre), Kristiina Pellas and Jarmo Lehtonen (Lääketietokeskus), Pekka Kekolahti and Matti Natunen (Grafimedia, a subsidiary of ICL Data), Esko Brotherus (Nokia Corporation Head Office), Ilkka Nousiainen (Otava), Markku Ylinen (Aamulehti), Jarmo Lahti (Talentum), and Kai Linnilä (Finlandia-Arkisto) are just some of the names I should mention.

I wish to thank my teachers at the Helsinki University of Technology, including Heikki Saikkonen, Ken Rimey, and many others, for all they have done to help me. I want to thank the Laboratory of Information Processing Science for allowing me to use their computers and facilities. I want to thank Kimmo Koskenniemi of University of Helsinki for guiding me in morphological processing of natural language.

I thank my friends at the Helsinki University of Technology, including Kenneth Oksanen, Johannes Helander, Tero Kivinen, Hannu Aronsson, Ken Rimey, Jukka Virtanen, Tero Mononen, Pekka Nikander, and many others, for all the discussions, ideas, support, as well as for the less scientific activities.

I want to apologize to my parents for so many times disturbing their sleep by using the computer till four o'clock in the morning. I want to thank them for their understanding and support.

1 Introduction

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. Many commercial databases allow full-text search, either as the primary search method, or as a method that can be used together with other searching methods. The trend in the industry has been towards increasing use of full text retrieval.

1.1 Full text retrieval as an information retrieval method

Within the information retrieval community, there has been considerable controversy on the usability of full text retrieval. Some people claim that full text retrieval produces too many irrelevant matches or is unable to find the relevant documents [6]; others argue that full text retrieval is able to find documents not found using other searching methods [11]. A conclusion appears to be that best results are achieved using many methods simultaneously [48, 13]. A recent comparison of information retrieval methods can be found in [29].

Despite of its weaknesses, full text search is an extremely useful tool in information retrieval, and it is an important component in most of today's commercial information retrieval systems. It provides capabilities not currently achievable using any other method, and thus full text retrieval is likely to remain a mainstream retrieval method for some time. New methods are being developed, mostly based on linguistic analysis of the source texts (eg. [24]), but most of the new developments only complement full text retrieval.

1.2 Full text indexing methods

Full text retrieval systems allow searching for stored documents using any words as search keys. Most systems also allow some kind of "wildcard search", proximity search, and offer boolean logic for combining search keys.

Fast retrieval from a large database requires efficient indexing techniques. Full text indexing is a slightly different problem from conventional database indexes: even a single document a few pages long contains thousands of search keys. The archive of an average newspaper editorial

system contains millions to hundreds of millions of search key occurrences; larger databases easily contain billions of search key occurrences.

Several methods are used to implement full text search [15, 29].

- Full text scanning. This requires a sequential scan through the database, and is not applicable to large databases.
- Inversion of terms. The idea is to list the words occurring in the document, and for each word list the places where it occurs. The list of terms can be organized using any method, such as B-trees, TRIEs, hashing, or variations and combinations of these. Variations of this approach are used in most commercial systems.
- Signature files. The idea is to compute a hash function from each word of the document and bitwise-or together the signatures of each word. Multiple levels can be used. For more information, see eg. [41].

This paper concentrates on inverted list methods, and in particular on inverted lists implemented using a B-tree. The B-tree has several desirable properties, including

- fast searching,
- the ability to search for truncated keys (using range queries),
- fast indexing,
- dynamic insertion and deletion,
- simplicity, and
- concurrency.

Hash tables are not well suited for searching for truncated keys. Signature files have problems with truncated keys and with deletion. Trie methods require large index files and are slower in searching and indexing.

Many languages have complicated inflection. For example, a Finnish noun can have approximately 2 200 forms, and a Finnish verb about 12 000 forms [22, pp. 356-357]. Some kind of morphological processing and truncated searches are essential for finding all occurrences of a word. Word inflection also affects the size of the index file and the number of distinct keys. Inflection can be handled either at search time (eg. using

FINSTEMS [23, pp. 81–92]), or at indexing time by reducing word forms to their base form (eg. using TWOL [27]).

The reader should realize that the problems in full text indexing are not the same as in conventional databases. Even a fairly modest database, say ten megabytes, contains about million search keys. Adding a new document (a few pages) means inserting thousands of new keys. Similarily, removing a document often means deletion of several thousand keys. In interactive personal computer applications it is desirable to be able to add a new document while the user is waiting. This is simply not possible using methods that require several disk accesses per key. Also, full text databases often contain hundreds of megabytes or several gigabytes of data. Just the initial indexing of such a database would take months using conventional methods.

1.3 B-tree indexing in full text retrieval systems

Many full text retrieval systems use B-tree based indexes. B-trees have many desirable features – they are compact, efficient, easy to manipulate, their properties are well known, and they adapt reasonably well to concurrent systems. Most importantly, they deliver sufficient searching performance, and also allow range queries, which can be used to implement wildcard search. The trend in the industry has been towards increasing use of B-tree based indexing.

Typical B-tree insertion time is on the order of 5 disk accesses (about 0.05 seconds), or one disk access (about 0.01 seconds) if non-leaf nodes of the tree can be kept in memory. B-tree search times are of the same order of magnitude. In general, search and insertion times are logarithmic on the number of different keys, and the constant factor is quite small.

However, in full text databases, the number of keys to be added is usually large. Adding a single document, say 10 000 words, requires 10 000 insertions and takes about 8 minutes (about 2 minutes if the upper levels are kept in memory). Similarily, building the index for a database of a few hundred megabytes (quite common in newspapers) could take a couple of months. Databases of 500 megabytes are not uncommon today, and after a couple of years the larger databases will be many gigabytes.

These figures illustrate that conventional B-tree algorithms will not be sufficient for full text databases in the long run. [47], [12] and [46] present partial solutions, and an improved solution is presented in section 2 in this paper.

1.4 B-trees

1.4.1 History

B-trees were first presented in 1971 by Bayer and McCreight [3]. A more theoretical analysis was presented by Bayer in 1972 [2]. The B-tree received lots of interest, and many papers were published in the next few years. Knuth [28, pp. 471–479] gives a good presentation of the early work, including the B*-tree (nodes at least $\frac{2}{3}$ full) and B$^+$-tree (all data is in leaf nodes) variants (although the name B$^+$-tree came to use later; [10] discusses some of the confusion about the names).

Bayer and Schkolnick [4] presented in 1977 some schemes for concurrent operations on B-trees. Guibas and Sedgewick [18] presented a top-down approach (see also [37]), which provided fairly good concurrent performance. Lehman and Yao [34] presented B$^{\text{link}}$-trees, where much of the locking is avoided by the addition of an extra link in each node. Lanin and Shasha [33] presented an improved version of B$^{\text{link}}$-trees. The idea of a separate process for rebalancing the tree was introduced by Sagiv [42], and generalized by Nurmi and Soisalon-Soininen [39, 40] (see also [25]).

Mathematical analyses of the dynamics of B-trees have been presented eg. by Zhang and Hsu [51], Baeza-Yates [1], Johnson and Shasha [20], and Matsliasch [36].

Eiter, Shrefl and Stumptner [14] compare the operation of B-tree concurrency control algorithms. Srinivasan and Carey [45] compare B-tree concurrency control algorithms and find the B$^{\text{link}}$-tree [34] to perform best.

Silberschatz, Stonebraker and Ullman [44] predict the growth of databases, and point out the need for efficient indexing algorithms for very large numbers of keys. Srinivasan and Carey [46] give some new solutions for the initial indexing of large tables.

1.4.2 Definition of the B-tree

B-trees are an extension of binary trees, where each physical node can contain multiple keys. In many respects, B-trees resemble binary trees where several nodes have been grouped to a single larger node. Figure 1 illustrates this idea. If all nodes in a group are read from the disk in a single access, searches can be done much faster than using conventional binary trees.

A B-tree consists of nodes, each of which may have at most m sons (figure 2). More formally, a B-tree of order m is a tree which satisfies the

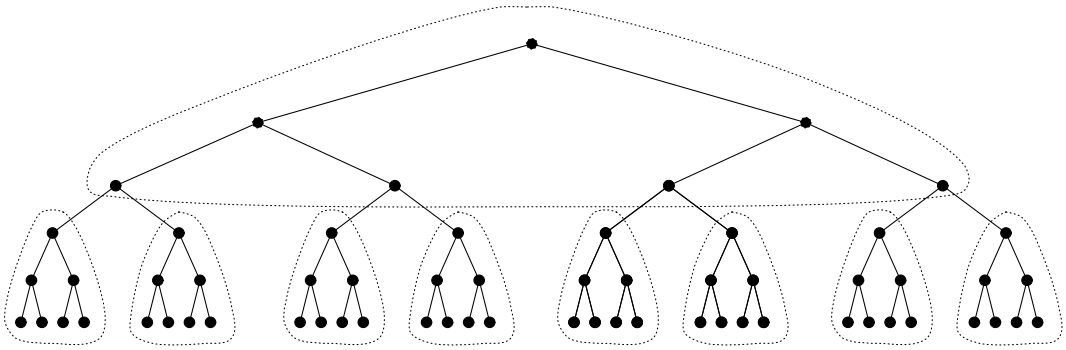

Figure 1: The B-tree can be thought of as a binary tree where many nodes have been grouped together.

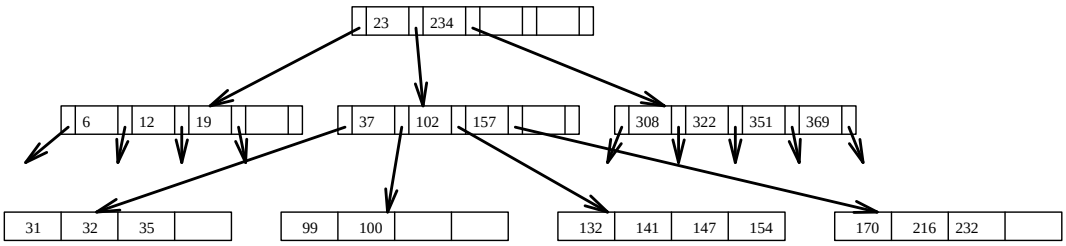

Figure 2: The B-tree data structure ($m = 4$).

following properties:

- Every node has $\leq m$ sons.
- Every node, except for the root and the leaves, has $\geq m/2$ sons.
- The root has at least 2 sons (unless it is a leaf).
- All leaves appear on the same level.
- A non-leaf node with k sons contains $k - 1$ keys.

A node which contains j keys and $j + 1$ pointers can be represented as

$$\boxed{P_0, K_1, P_1, K_2, \ldots, P_{j-1}, K_j, P_j}$$

where $K_1 < K_2 < \ldots < K_j$, and P_0 points to the subtree containing keys $K < K_1$, P_j to the subtree containing keys $K > K_j$, and P_i $(0 < i < j)$ points to the subtree for keys $K : K_i < K < K_{i+1}$.

Searching in a B-tree is quite straightforward: after a node has been fetched into the internal memory, the given argument is searched among the keys K_1, K_2, ..., K_j. If the searched key equals one of K_i, the search is complete. Otherwise, if the key lies between K_i and K_{i+1}, then the node indicated by P_i is fetched, and the process is continued. If the key is less than K_1, then P_0 is used, and if the key is greater than K_j, then P_j is used. If P_i is a null pointer, the search is unsuccessful.

B-tree insertion is also quite simple. In most cases, the new key can simply be inserted in the appropriate leaf node, and no further action is required. However, if the node where the key should be inserted is already full, the node must be split, and the middle key of the node must be inserted in the parent node. One can think of the insertion as creating a temporary node, which can be larger than the normal nodes, and if the temporary node is in fact too large, it is split to the two nodes shown below (here, m is the number of keys in the temporary node).

$P_0, K_1, P_1, \ldots, K_{\lceil m/2 \rceil - 1}, P_{\lceil m/2 \rceil - 1}$	$P_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, P_{\lceil m/2 \rceil + 1}, \ldots, K_m, P_m$

The key $K_{\lceil m/2 \rceil}$ is inserted to the parent of the original node (effectively replacing a pointer P in the parent node by the sequence $P, K_{\lceil m/2 \rceil}, P'$). This insertion may cause the father node to contain too many keys, in which case it needs to be split, and so on. If the node which gets too full is the root node, the root node must be split, and a new root node must be created, containing only the single key $K_{\lceil m/2 \rceil}$; the tree gets one level higher in this case.

The insertion procedure preserves all of the B-tree properties.

A deletion algorithm can also be constructed for the B-tree. However, the deletion algorithm must be able to handle the case when a key in a non-leaf node is deleted. In this case, it is necessary to move a key adjacent to it in the ordering of the tree to its place. This means moving a key up from a leaf node. The deletion algorithm for B^+-trees is simpler, and B^+-trees are often used instead of ordinary B-trees.

1.4.3 B^*-trees

B^*-trees are very much like ordinary B-trees, except that each node (except for the root) must be at least $\frac{2}{3}$ full. The properties of B^*-trees can be formulated as follows

- Every node has $\leq m$ sons.

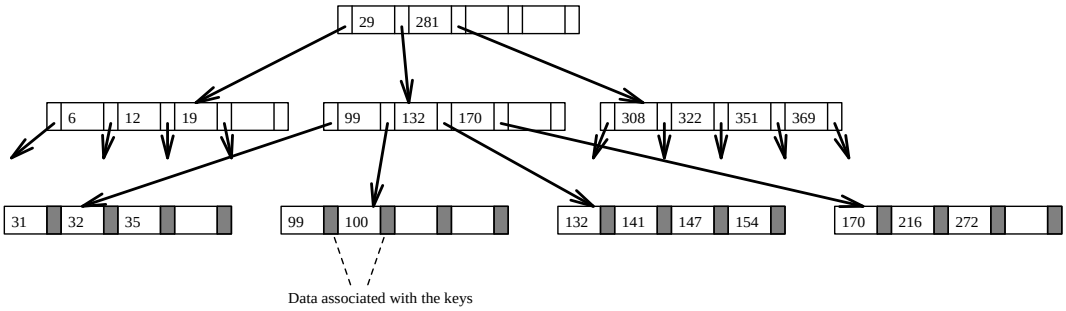

Figure 3: The B$^+$-tree data structure ($m = 4$).

- Every node, except for the root and the leaves, has $\geq (2m-1)/3$ sons.
- The root has at least 2 and at most $2\lceil(2m-2)/3\rceil + 1$ sons.
- All leaves are on the same level.
- A non-leaf node with k sons contains $k-1$ keys.

No changes are needed to the searching algorithm; however, the insertion algorithm is more complicated. It is no longer possible to just split when the node becomes full. Instead, the insertion algorithm must first look at either brother of the node, and if there is space in the brother, move some data to the brother node. If both the node and its brother are full, both must be split, creating one new node. As a result, all three nodes will be $\frac{2}{3}$ full. The deletion algorithm needs corresponding changes.

1.4.4 B$^+$-trees

Usually, some additional data is stored in the B-tree with the keys. This data can be, for example, the address of the record containing the key value.

B$^+$-trees are a variant of B-trees where all keys and their associated data are stored in the leaf nodes. Some of the keys (but not their data) is duplicated in the upper levels of the tree. Figure 3 illustrates the structure of a B$^+$-tree.

A non-leaf node in a B$^+$-tree is of the format

$$\boxed{P_0, K_1, P_1, K_2, \ldots, P_{j-1}, K_j, P_j}$$

```
Find the leaf node where the key should be inserted.
Insert the key (and its data) in the node in the appropriate place.
if the node is too full
  then begin
         Split the node in two.
         Insert the first key of the latter node in the parent.
       end
```

Figure 4: The B$^+$-tree insertion algorithm.

where K_1 is a copy of the first key in the node pointed to by P_1, K_2 is the first key in the node pointed to by P_2, etc. The subtree pointed to by P_i contains all keys K: $K_i \leq K < K_{i+1}$. The subtree pointed to by P_0 contains only keys $K < K_1$, and the subtree pointed to by P_j contains only keys $K \geq K_j$.

A leaf node in a B$^+$-tree can be represented as

$$\boxed{K_1, D_1, K_2, D_2, \ldots, D_{j-1}, K_j, D_j}$$

where D_i is the data associated with K_i (in the above discussion of ordinary B-trees, the D field was considered a part of the key).

Insertion in a B$^+$-tree is similar to normal B-tree insertion. However, when a node is split, all keys are left in the leaf nodes, and the first key of the "latter" node is copied to the parent. Figure 4 outlines the insertion algorithm.

Deletion in a B$^+$-tree is significantly easier than in an ordinary B-tree. In most applications, deletion is allowed to leave the node less than half full. Analytical and experimental results indicate that under random insertions and deletions the tree will be about 68% full, even if nodes are freed only when they get empty [20]. Figure 5 illustrates the deletion algorithm (a version which deletes nodes only when they get empty). Note that it is possible that after deletion the key in the parent is no longer the same as the first key of the node. However, this is no problem, as all operations still go to the correct node.

B$^+$-trees have become an important B-tree variant in database applications. Handling of a B$^+$-tree (especially deletion) is easier than handling an ordinary B-tree, the branching factor of the tree is higher due to having shorter keys in non-leaf nodes, and it is easier to make all nodes of the same size.

Find the leaf node where the key to be deleted resides.
Delete the key (and its data) from the node.
if *the node is empty after deletion*
  then begin
    Free the node.
    Delete the pair (K_i, P_i) *for this node in the parent.*
  end

Figure 5: The B^+-tree deletion algorithm.

1.4.5 B-trees with variable length keys

The ideas behind B-trees can also be generalized to trees with variable-length keys (or variable-length data). In this case, the number of keys in a node varies, and the balance conditions for the tree must be reformulated. The idea is to no longer put the bounds $\lfloor m/2, m \rfloor$ on the number of sons of each node; instead, it is just stated that each node should be at least about half full of data. The insertion algorithm needs to be changed so that about half of the data in a node goes to one node, and the other half to the other node. The conditions for deletion can also be formulated in terms of the available space in the node.

Variable-length keys can be used both with conventional B-trees and B^+-trees.

1.4.6 Concurrency control: Bottom-up algorithms (Bayer–Schkolnick)

In [4], Bayer and Schkolnick presented three algorithms for concurrency control in B-trees. Their idea is to lock the leaf node, its parent, and grandparents until a "safe" node is reached (safe here means a node in which a key can be inserted without splitting – a node with free space for at least one more key). The nodes are locked on the way down the tree (away from root), and rebalancing goes up the tree (towards root).

1.4.7 Concurrency control: Preparatory operations (Top-down insertion)

In [18], Guibas and Sedgewick presented an insertion algorithm for balanced trees in which rebalancing was done on the way down the tree, and

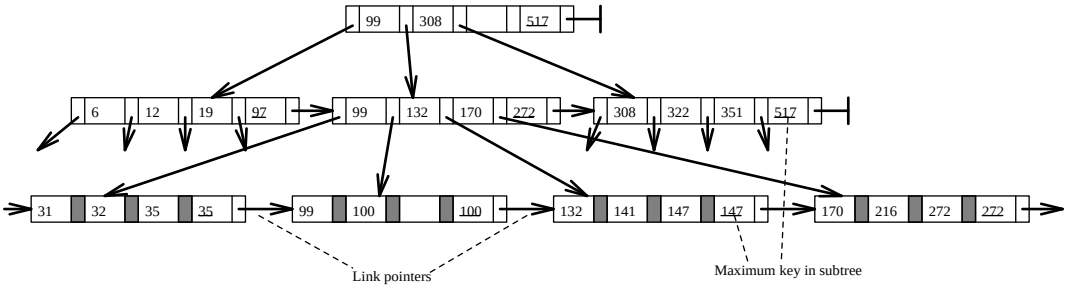

Figure 6: The B^{link}-tree data structure ($m = 3$).

thus no bottom-up rebalancing was needed. Their algorithm only needs to keep two nodes of the tree locked simultaneously.

Mond and Raz [37] generalized this framework for B^+-trees. Their algorithm splits each full node on the way down the tree, as a preparatory operation which guarantees that there will be no need to propagate splits up the tree. At most two nodes of the tree need to be kept locked simultaneously. This allows fairly high performance at the cost of a minor space overhead.

1.4.8 Concurrency control: B^{link}-trees

Lehman and Yao [34] presented a new B-tree variation, the B^{link}-tree, in which no locking is needed for the readers, and only a small constant number of locks is needed for an update. The locks are only used to block other updates; readers do not care about them.

The B^{link}-tree (see figure 6) is a variation of the B^+-tree[1]. Each node of the tree has an extra link pointer and an extra key containing the maximum key value in the rightmost subtree.

The extra link pointer points to the next node at the same level of the tree as the current node, except that the link pointer of the rightmost node on a level is a null pointer. All nodes on the same tree level form a linked list.

The purpose of the extra link pointer is to provide an additional method for reaching a node. When a node is split because of data overflow, a single node is replaced by two new nodes. The link pointer of the first new node points to the second node; the link pointer of the second node contains the contents of the link pointer field in the old node. Usually, the first new node is stored in the same physical page as the old node.

[1] [34] uses older terminology: the term B^*-tree is used for what is in modern terminology called the B^+-tree.

The intent of this scheme is that the two nodes, since they are connected by a link pointer, are functionally essentially the same as a single node until the proper pointer can be added in their parent.

For any node in the tree (except for the first node on any level) there are two pointers in the tree that point to that node (a "son" pointer from the parent and a link pointer from the left brother of the node). The link pointer is always created first, and thus it is legal for a node to have a link pointer pointing to it, but no son pointer. The new node will be reachable through the link pointer in its left brother, and if the searching, insertion and deletion algorithms are updated to use the node pointed to by the link pointer if the key is greater than the "maximum" key in the node, everything will work properly no matter how long the delay is from creation of the node to its insertion in the parent. Thus, only the link pointer needs to be updated in a split, and the first key of the second new node can be inserted in the parent later.

It is desirable to insert a new node in its parent fairly soon, as an extra disk access is needed whenever a process needs to use the link pointer. The B^{link}-tree insertion algorithm resembles the normal bottom-up B^+-tree insertion algorithm. After it has split a node, it locks its parent, and inserts the proper key in that node. Note however, that there was no need to do any locking at all on the way down. All locks are acquired in strict bottom-up order (thus preventing deadlock). It is possible that other processes use the node being locked and the node which was just split, before the lock on the parent is granted. However, since the B^{link}-tree structure is consistent even when the new key has not yet been inserted in the parent, this does not cause any problems.

The result of all this is that searching processes never need to wait for locks, and conflicts between insertion processes are very rare. A very high level of concurrency can be achieved.

Deletion in B^{link}-trees does not care about rebalancing the tree. Deletion can create nodes which have less than $\lfloor m/2 \rfloor$ keys. In most database systems insertions occur more frequently than deletions, and thus this will usually not cause problems.

Since B^{link}-trees continue to operate properly even if not fully balanced, balancing can be delayed until later. Sagiv [42], and Goodman and Shasha [17] proposed separating rebalancing from update. The rebalancing could be done by a separate process, running either concurrently with other processes, or outside rush hours.

1.4.9 Concurrency control: Relaxed balance (Nurmi–Soisalon-Soininen–Wood)

In the relaxed balance scheme [39], the balance conditions of the B^+-tree are allowed to be violated in a controlled manner, and a later balancing process is used to restore the balance. No extra links are needed in the index. A mark bit is used in the description of the algorithm; however, in an actual implementation it can be avoided. During updates, only one node needs to be locked at any one time.

During insertion, only the leaf node where the key is to be added needs to be locked. If the leaf node gets too full, two new nodes are created, the keys are distributed between the nodes, and the contents of the old node are replaced by just one key and pointers to the two new nodes. The old node is also marked as needing rebalancing.

During deletion, nodes are never freed. Instead, if a node gets too empty, it is simply marked as needing rebalancing.

The rebalancing process looks up the marked nodes in the index, and does any necessary rebalancing. The rebalancing can be done either by starting a rebalancing process as soon as the tree gets unbalanced, or later outside rush hours.

The relaxed balance scheme may have problems when very many keys are inserted in sorted order. If the index is constructed by inserting keys in alphabetical order, the depth of the tree grows linearily as a function of the number of keys in the tree. This worst-case behaviour may make the method less desirable in some applications, and it makes optimizing insertion by sorting many keys before insertion difficult.

1.4.10 Comparison of concurrency control methods

A recent comparison of B-tree concurrency control methods was presented by Srinivasan and Carey in [45]. They compared the Bayer-Schkolnick algorithms, top-down algorithms, B^{link}-tree algorithms, and a new algorithm of their own. However, when examining their results, one should note that the size of cache memory in their experiments was about 75% of the tree size, which is much more than in typical datbase applications. It is unclear how much this affects the reliability of their results.

They conclude that the B^{link}-algorithm provides clearly the best overall performance. They summarize the results as follows.

1. In a system with a single CPU and disk, there is no significant performance difference between the various algorithms.

2. Lock-coupling with exclusive locks is generally bad for performance. Even for workloads dominated by searches, algorithms in which updaters use such lock-coupling strategy cannot take full advantage of CPU and I/O parallelism, even in systems with only a few CPUs and disks.

3. The extent to which an overhead like a restart or a link-chase directly affects performance depends more on the number of conflicts that it creates than on the extra resources used.

In a more analytical study [21], Johnson and Shasha have also concluded that the B^{link}-tree provides the best performance.

1.4.11 Alternatives for storing occurrence data

In a full text database, the words usually occur many times. The indexing system must be able to store the occurrence data efficiently. [12] discusses some possible methods for maintaining the occurrence data for the keys.

- Each occurrence can be stored as a separate key in the index. This wastes lots of space, and may cause complications in the B-tree algorithms.

- Occurrence data for a key may be stored in the node with the key. However, due to limited node size, only a limited amount of occurrence data can be stored.

- Each key in the index can contain a pointer to an external block used to store the occurrence data. This requires an extra disk access every time a key is inserted or accessed. The size of the external block can be doubled when it becomes full. At least half of the space in the block is always in use.

- Some occurrence data is stored in the key, and when enough data has been collected, some of it is moved to an external block. In [12], this technique is called *pulsing*.

However, even the last method presented in [12] has problems. If there are very many occurrences (there can easily be tens of thousands, or even millions if stop words are not used), the block becomes impractically large. Also, managing blocks of many sizes may be complicated.

The method used in the system described later in this paper is to store some data in the node, and when enough data has accumulated,

move some of it to an extension block, which is put at the front of a chain of extension blocks for that key. The extension blocks will always be completely full. (See figure 7 for an overview of the structure of the index.)

1.4.12 Optimizations: Disk cache

B-tree operations are significantly faster if the upper layers of the tree can be kept in memory. A standard LRU disk cache performs well for this purpose, as the upper layers of the tree are used very often.

Assuming that the tree is of order m, just one buffer block (for the root) can save one disk access for each operation, and $m+1$ buffer blocks can save two disk accesses for every operation. In many systems the nodes are large enough that there are only three levels in the tree, and the number of disk accesses needed for searching, insertion and deletion is reduced to one (not counting writes, which are usually asynchronous).

If the keys are inserted in order, only as many buffers are needed as there are levels in the tree [28, p. 479]. In most applications, the depth of the tree is 3 or 4.

One should note that efficient use of a disk cache may be difficult in a networked environment with many processes accessing the index concurrently. The problem is particularly difficult in microcomputer environments, which lack sophisticated multiprocessing and interprocess communication facitities.

1.4.13 Optimizations: Group update

Creating a B-tree index is very fast if the keys are available in sorted order [28, p. 479], which was exploited for instance in [46].

The same idea can be used if very many keys are to be inserted into an existing index in a single operation, which is very common in full text databases. By collecting many keys in memory, multiple occurrences of a key can be combined, and the keys can be inserted in order, caching the path from the root to the current node. Very often, many keys are to be inserted in the same node. In this case, there is no need to reread the node from disk; instead, it is readily available in main memory. Thus, keys can be added in less than one disk access per key. This idea was used in [47] and [12].

In this paper a new algorithm is presented for merging the buffered keys in memory to the B-tree on disk. The new algorithm allows concurrent access and provides better CPU time efficiency than the straight-

forward algorithms used in [47] and [12]. With sufficently large buffer memories, CPU time becomes the dominating factor in indexing speed.

1.4.14 Optimizations: Prefix omission

Very often, two adjacent keys in a node share a common prefix. The prefix omission technique [9] means that the common prefix is not stored; instead, each key contains the count of characters common with the previous key. These characters need not be stored in the latter key.

The calculations in section 2.3.6 indicate that prefix omission does not provide significant space savings in large full text databases.

1.5 The new insertion algorithm

A new B-tree insertion algorithm is presented in this thesis. Much of its speed is gained using the methods presented in 1.4.13. In addition, a novel algorithm is presented for merging the buffered keys in memory to the B-tree on disk. The merging algorithm allows concurrent processing, and is very efficient CPU-wise.

The new merging algorithm keeps at most two nodes of the B-tree locked at any one time, allowing other processes to freely access other parts of the index file. It starts to look for the node where the next key belongs starting from the node where the previous key was inserted. Almost always, the new key belongs to a node under the same parent node as the previous node. In this case, the new leaf node can be found in a single disk access, without the need for caching (and thus locking) the entire path from the root to the previous leaf node. Also, since only the immediate parent of the leaf node needs to be searched, some CPU time can be saved.

Inserting many keys to a node is also optimized by constructing the new node incrementally. As shown in 2.3.5, this can provide significant savings.

1.6 Application of the new algorithm to deletion

The new algorithm can be extended to deletion, and in fact it is possible to collect both insertions and deletions in memory at the same time, and merge them both simultaneusly to the B-tree on disk.

1.7 Scientific results

The idea of constructing an index efficiently from a sorted list of keys is very old (eg., [28, p. 479]). Basically the same idea is used in [46], where the keys are explicitly sorted before the index is constructed.

Also, the idea of buffering many keys in memory has been presented earlier in [47] and [12]. In [47] two grouped update algorithms are presented; however, no attempt is made to handle concurrent operation. In [12], the buffered keys are simply inserted in order, caching the path from the root to the leaf node. Their method cannot easily be adapted for concurrent operation.

The new merging algorithm appears to be novel, and the first group update algorithm that allows concurrent operation. Also, the new algorithm provides significant CPU time savings, which has become important due to the small number of disk accesses needed in group update.

The statistical analysis of prefix omission in section 2.3.6 gives results which are quite surprising, considering that prefix omission has been treated in quite positive light in earlier publications [9, 7].

The discovery that preallocating space for a maximal key in non-leaf nodes incurs negligible storage overhead (see section 2.3.6) contradicts with some earlier statements in the literature, eg. in [25].

1.8 Topics for further research

Currently, the new algorithm uses locks much in the spirit of the preparatory operations B-tree. However, since two nodes are kept locked for a fairly long time, the granularity of the locking is fairly coarse. Also, each process blocking on a lock of another process will have one more node locked futher up the tree. Only as many processes as there are levels in the tree are needed until the root node gets locked (which will block all operations on the index until the lowermost locks are released).

It would probably be possible to extend the ideas presented here to use B^{link}-trees, at the same time avoiding the need for cascading locks.

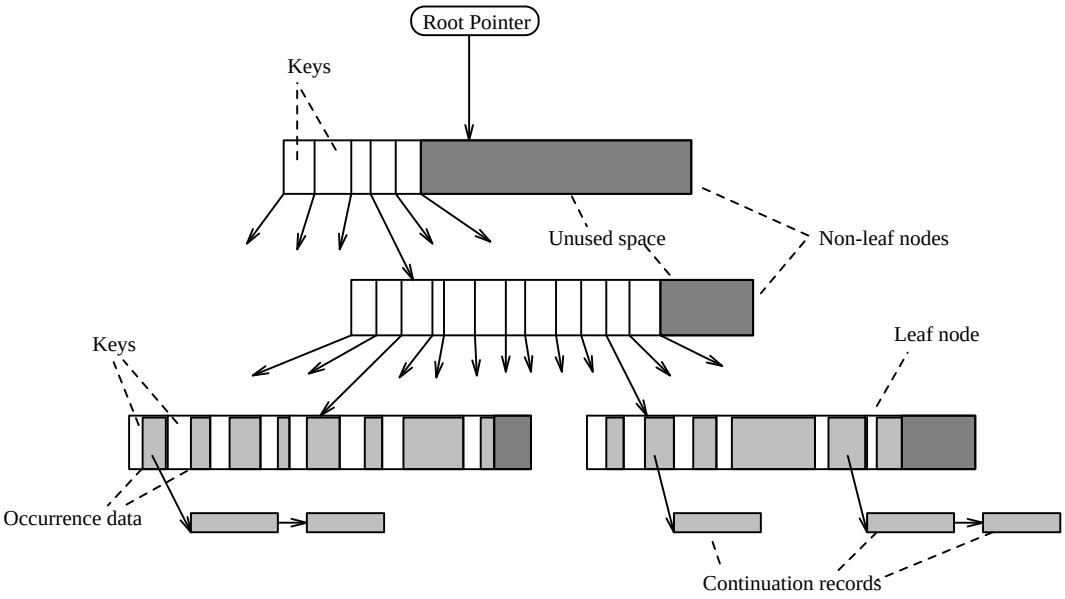

Figure 7: Index file structure.

2 The new algorithm

This section describes the new indexing algorithm in detail. Figure 7 presents an overview of the structure of the index file. It is basically a B^+-tree with variable length keys. Some occurrence data is stored in the nodes, and more may be stored in continuation records. The continuation records of a key form a linked list; all continuation records are always full.

2.1 Overview

The new indexing algorithm works in two alternating phases. In the first phase, keys and occurrence data are collected in 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 memory reaches a specified limit, the keys and occurrences in memory are merged with the B-tree on disk. Keys are read out of the memory structure one at a time (in the same order in which the B-tree on disk is ordered), and the node where the key should be inserted is located. Any nodes on the path from the root to the leaf node will be split if it cannot be guaranteed that there is enough space in the node for inserting at least one more key (top-down rebalancing).

If the key being inserted already exists in a leaf node, new occurrence

data will be appended to the data already in the node. If the key does not yet exist, it (and its occurrence data) is added to the node. If the key has very much occurrence data, some of it is stored in continuation records outside the node.

If there isn't enough space in the leaf node to add the new key and its occurrence data, the node must be split. A new leaf node is created, approximately half of the data in the old node is copied to it, and it is added under the parent (we know that there is enough space in the parent). If, after adding the new key, there still remains space for at least one more key in the parent, adding can be continued where it was before split. On the other hand, if there isn't enough space, indexing must be restarted from the root.

When a key has been added, processing continues with the next key. Quite often, the next key should be added to the same node as the previous key, or to a node under the same parent as the previous key. In the former case, processing of the node simply continues. In the latter case, a disk access is needed to load the new node, but CPU time (and concurrency & caching complications) needed for a full lookup are avoided. If the next key is not under the same parent as the previous one, merging must be restarted from the root.

2.2 Buffering data in memory

The purpose of buffering many keys in memory is to

- sort the keys into the same order that the B-tree on disk uses, and
- coalesce multiple occurrences of a single key by combining their occurrence data.

The keys are inserted to the memory buffers in arbitrary order (quite often the textual order), and read out in the order used by the B-tree (usually the alphabetical order).

In the current implementation, the memory buffers are implemented using a red-black tree [43, pp. 187–199].

2.3 Merging the keys in memory to the B-tree on disk

The insertion algorithm makes use of the knowledge that it receives the keys in the same order that the B-tree is in. When many keys are inserted

```
while there are more keys to insert do
 begin
  Get next key.
  if there is no previous node
    then Find the node where key belongs, starting from the root.
    else if the new key belongs to a different node
           then Find the node where the key belongs.
  if there is enough space for the key in the current node
    then Insert the key in the current node.
    else begin
           Split the current leaf node.
           if there might not be enough space in the parent node
             then Find the node where the key belongs
                    starting from the root.
           Insert the key in the current node.
         end
 end
```

Figure 8: The new insertion algorithm.

in the same node, significantly less than one disk access per key is needed. Also, by not starting the insertion from the root every time, significant amounts of CPU time can be saved, at the same time making concurrent operation a lot easier by not requiring locking of the whole path from the root to the leaf node.

The insertion algorithm consists of the following major operations:

1. Finding a leaf node starting from the root.
2. Splitting a non-leaf node.
3. Inserting a key in a leaf node (or, if the key already exists, adding its occurrence data).
4. Finding the node where the next key belongs.
5. Splitting a leaf node.

The basic algorithm is outlined in figure 8 (for a more detailed description, see appendix A).

Non-leaf nodes are split only when searching for a leaf node starting from the root. This ensures that a node can always be split and there

```
Lock and read the root node address.
Lock and load the root node.
while the node is not a leaf node do
 begin
   if there isn't enough free space in the node
    then begin
           Split this node.
           Release all locks and restart from the root
         end
   Unlock the root node address or the parent node.
   Find the child pointer where the current key belongs.
   Lock and load the child node.
 end
```

Figure 9: Finding the leaf node and splitting non-leaf nodes.

will be enough space in the parent. There is no need to keep the parents locked. Nodes are always locked in a strict top-down order, and deadlocks are thus avoided.

2.3.1 Finding a leaf node, starting from the root

To find the leaf node where the current key should be inserted, the B-tree searching algorithm is used. Any nodes which are too full to hold one more key will be split on the way down the tree. If the root node is split, a new root node will be created; otherwise a key will be moved to the parent node (we have just come from there, and know that there is enough space for the key). At this point, the parent node (or the root node pointer) is still locked. After it is known that there is enough space in the node, and it is a non-leaf node, the parent can be unlocked.

When the leaf node where the key should be inserted is found, both it and its immediate parent are left locked.

Note that splitting non-leaf nodes is very rare, and it is quite acceptable to restart the search from the root after a split for the sake of simplicity. Also note that keeping space in non-leaf nodes for an extra key does not cause any significant waste of disk space (a rough calculation in section 2.3.6 indicates that the wasted space is on the order of 0.01 percent of the index file size).

Figure 9 illustrates the algorithm for finding a leaf node.

if *the key belongs to the same node as the previous key*
  then return
Unlock the previous leaf node.
Find in the parent node the child pointer under which
  the key belongs.
if *the child pointer is the last pointer in the parent node*
  then *Unlock everything and use the algorithm of figure 9.*
  else *Lock and load the new leaf node.*

Figure 10: Finding the node where the next key belongs.

2.3.2 Locating the node where the next key should be inserted

In most cases, the next key is to be inserted in the same node as the previous key. The implementation should try to optimize this case (see section 2.3.5).

The next common case is that the key belongs to a node under the same parent node as the previous key. This can be tested by looking for the key in the parent node. If the current key is greater than the greatest key in the parent node, it must be looked up starting from the root (to speed up the first merge phase of a new index file, an implementation might choose to optimize inserting to the rightmost node of the index). Normally the next node will be found under the same parent. It is locked and loaded, and it becomes the new current node.

Figure 10 illustrates the algorithm for finding the leaf node for the next key.

2.3.3 Inserting a key to a leaf node

When inserting a new key to a leaf node, or adding occurrence data to an existing key, it is possible that space in the current node runs out. In this case, the current node must be split. It is known at this point that there is space in the parent node for inserting the key which is passed up. However, after insertion, it must be checked that there remains enough space in the parent for one more key. If not, processing of the next key must start from the root.

Note that it would also be possible to implement the algorithm without reserving extra space in non-leaf nodes. However, in that case it would be necessary to restart insertion from the root in the middle of a new key if the current node would have to be split. If each key is inserted into the current node separately, this will not cause any problems. However, if

the optimization of section 2.3.5 is used, it is desirable to be certain that there is space in the parent.

2.3.4 Continuation records for occurrence data

It is best to store some of the occurrence data of a key in the node with the key [12]. Most keys appear only a few times, whereas some keys can appear millions of times unless they are filtered out as "stop words". Even for valuable search keys the number of occurrences can vary from one to several thousand. It is impractical to prepare to store that many occurrences in the node – there can be more occurrences than is the size of the node.

Continuation records are the solution to the problem of storing occurrences. If a key has very many occurrences, some of them are moved to separate records containing only occurrence data. These records are chained together, and the data for the key in the node contains a pointer to the head of the chain. Continuation records can easily be implemented in such a way that the chain never needs to be accessed during insertion. New records are added to the front of the list when there are sufficiently many occurrences in the node.

2.3.5 Optimizing multiple insertions to a node

In the most common case, many keys are inserted into the same node. Regardless of whether the key is new or already exists in the node, insertion typically involves inserting data at some point in the middle of the node. The rest of the node must be moved correspondingly, for each key being added to the node. (See [30] for a different approach.)

The cost of multiple copying can be approximated by

$$\bar{t}_{mc} = \bar{N}_i \frac{\frac{1}{2}\bar{U}_n}{C}$$

where $\bar{t}_{mc}$ is the average time needed for multiple copying, $\bar{N}_i$ is the average number of insertions to a node, $\bar{U}_n$ is the average number of bytes used in the node, and C is the speed at which the CPU can copy data (bytes/sec). From [20], $\bar{U}_n$ is about $0.68S_n$ (S_n is the size of a node)). Thus,

$$\bar{t}_{mc} \approx \bar{N}_i \frac{0.34S_n}{C}. \quad (1)$$

A single copy of the node could be made in time

$$\bar{t}_{sc} \approx \frac{0.68 S_n}{C}. \tag{2}$$

Assuming[2] $\bar{N}_i = 50$, $S_n = 8192$ and $C = 10MB/s$ (the data is usually unaligned), $\bar{t}_{mc} \approx 14ms$ and $\bar{t}_{sc} \approx 0.6ms$.

The total time $\bar{t}_t$ needed for processing a node is

$$\bar{t}_t = \bar{t}_r + \bar{t}_p + \bar{t}_c + \bar{t}_w. \tag{3}$$

Assuming the time for reading the node $\bar{t}_r = 10ms$, the time for other processing (compares etc.) $\bar{t}_p = 3ms$ and the time for writing the node $\bar{t}_w = 3ms$ (asynchronous writes), $\bar{t}_{t_{mc}} \approx 30ms$ and $\bar{t}_{t_{sc}} \approx 16.6ms$.

From these calculations it can be concluded that if the node can be copied only once, about half of the total merge time can be saved (in CPU time savings this is much more than a half).

Multiple copying can be avoided by constructing a new version of the node incrementally. A new copy of the node is created in memory when the first key is inserted to the node. When insertion moves to another node, the new version of the node is written to disk overwriting the old version.

For incremental copying to work, the keys in each node must be kept sorted (which is a good idea anyway). When a key is to be inserted, all keys up to that key are copied to the new version of the node. If the key already exists in the node, it too is copied. If it does not exist, it is created at the end of the new version. If key layout is designed properly, new occurrence data can be simply appended to the key on the new node. When the next key is to be added, keys are copied up to that key, etc. When all keys belonging to the same node have been added, the remaining keys are copied from the old node to the new node, and the new node is written to the index file (overwriting the old node).

Special care must be taken when there isn't enough space in the new node to add a key or its occurrence data. In this case, the node must be split. This can be done simply by allocating a fresh node and writing about half of the data in the new node to the allocated node. Remaining data in the new node is moved to the beginning of the node, the first key of the newly allocated node is inserted in the parent, and processing can continue.

[2] In experiments, with 5MB of buffer memory and 8kB node size, $\bar{N}_i$ was 159 for a 10MB file of English text, 40 for a 100MB file of English text, and 157 for an 11MB file of Finnish text.

Also, special care must be taken to handle the case when there is no longer enough space in the parent node. It is possible that the new node is full, most of the old node is still unprocessed, and there is space in the parent node for only one more entry. In this case, the new node cannot be simply split in half; instead, the data on the new node and the remaining data on the old node must be divided between the new node and the old node, both nodes written to disk, and then inserting must be restarted from the root.

2.3.6 Optimizing index file size

The optimization of section 2.3.5 makes prefix omission easy to implement. Since keys are always appended to the end of the node (conceptually, at least), and it is easy to compute the prefix from the previous key on the new node.

However, the overall effect of prefix omission on index file size is quite small for large databases. With small databases it is much more important. This is because the vocabulary of normal language is limited to a few hundred thousand word forms. Space savings from prefix omission are relative to the number of unique word forms appearing in the database, whereas the size of the index file is mostly proportional to the total number of words in the database.

Space is needed in the index file for both the B-tree of the unique keys and for the occurrences. Let us denote the space used by the B-tree with S_t and the space used for continuation records by S_c. Let N_u denote the number of unique keys, $\bar{L}_k$ the average length of a key, $\bar{L}_b$ the average overhead per key for bookkeeping, $\bar{L}_{on}$ the average amount of occurrence information per key in the node, $\bar{L}_{oc}$ the average amount of continuation data per key in continuation records. The total size of the index file

$$S_i = S_t + S_c. \tag{4}$$

Using the results of [20],

$$S_t \approx \frac{N_u(\bar{L}_k + \bar{L}_b + \bar{L}_{on})}{0.68} \tag{5}$$

and

$$S_c \approx N_u \bar{L}_{oc} \tag{6}$$

An estimate of the total index file size can also be obtained from

$$S_t \approx \frac{N_u(\bar{L}_k + \bar{L}_b)}{0.68} + N_o \bar{L}_o \tag{7}$$

where N_o is the number of occurrences, and $\bar{L}_o$ is the average length of occurrence data per occurrence. (This estimate gives smaller results than reality, because some of the occurrence data is in the nodes, and use more space than the above equation suggests.

In experimental analysis it was found that a 315 MB corpus of English text[3] contained about 50 000 000 word occurrences and approximately 210 000 unique word forms, and the average length of a word was 4.26 characters.

Assume $N_u = 210\,000$, $N_o = 50\,000\,000$, $\bar{L}_k = 4.26$, and $\bar{L}_b = 3$. Using equation 7, the size of the index file is approximately 202 megabytes. 2 megabytes are needed for the B-tree containing the keys, and 200 megabytes for the occurrences.

With this material (English newspaper articles), saving a single bit on the average in occurrence data results in a saving of about 3 percent in the total index size, whereas saving a byte in key length on the average only saves about 0.1 percent. Even if the space requirement of the keys could be reduced to zero, only about one percent would be saved in index file size.

It may be interesting to compare these figures with those in [9]. Very close attention should be paid to exactly what the percentages in the article really refer to.

The calculations above show that prefix omission is completely useless and waste of CPU time for large databases of English text. However, the situation might be different with smaller databases, or if the database is in a language like Finnish, where a single word can have literally thousands of inflectional forms. (Many Finnish text database systems use morphological processing to normalize the words to their base form before indexing.)

Preparatory splitting increases the use of disk space in non-leaf nodes. Assuming a maximum key length L_{max} (including worst-case overhead), a node size S_n, an average branching factor $\bar{b}$, and N_u unique keys, the number of non-leaf nodes

$$N_n = \lceil \log_{\bar{b}} \frac{N_u(\bar{L}_k + \bar{L}_b + \bar{L}_{on})}{0.68 S_n} \rceil + 1 \tag{8}$$

The amount of disk space wasted by preparatory splitting,

$$S_w \approx N_n L_{max} \tag{9}$$

[3]Newspaper-like articles extracted from the ClariNet newsfeed; this material contains only the actual articles (all UseNet news headers have been stripped).

Experimental data suggests that $\bar{L}_{on} \approx 50$ and the braching factor for non-leaf nodes $\bar{b} \approx 500$. Using equation 8, there are approximately 2158 leaf nodes, and the number of non-leaf nodes $N_n = 3$.

As this example demonstrates, the number of non-leaf nodes is very small (3) compared to the index file size (202MB). If $L_{max} = 256$, the amount of space wasted by preparatory splitting in the 202MB index file is less than 1kB (note that extra space only needs to be reserved in non-leaf nodes). Thus, the amount of space wasted by preparatory splitting is completely insignificant.

2.3.7 File locking and concurrency

All locks are allocated in order from the root pointer to the leaf nodes. When a node is locked, its parent (or the root pointer) will always be locked. The parent (or the root pointer) can be unlocked if there is enough space in the child node (and the child node is a non-leaf node). This allows other processes to access other parts of the index while indexing is in progress.

The parent of the leaf node and the leaf node will both be kept locked, since the parent might need to be modified as a result of the child splitting, and the child must be locked before reading because someone else might have been using it. There is no need to lock continuation records, as any process accessing continuation record chains will have the node pointing to the chain locked.

Searching should be implemented so that it adheres to the same top-down locking practice. Searching processes use shared locks and indexing processes use exclusive locks. Deadlocks are not prossible since all locking is done in strict top-down order.

The indexing algorithm allows other processes to access other parts of the index file. However, since the index is quite shallow, and there aren't that many non-leaf nodes, the granularity of the locking is still quite rough. The probability of other processes stumbling into the locked parent node is quite high. Also, inserting data to all nodes under a parent node may take several seconds. This may be too long a wait in some applications.

Locking with this granularity would probably not be unacceptable in a relational database. However, in a full text database the speed of insertion is much more important due to the huge number of keys, and delays of a few seconds are usually acceptable, especially since they are worst-case delays – in most cases, there will be no delay, even if someone is indexing

at the same time. Even if some searching processes will have to wait every now and then, the indexing will be over much faster that it would be with the conventional algorithms, and the total system performance will be improved. Also note that multiple searching processes will not slow each other down, except by using more CPU resources and I/O bandwidth.

2.3.8 Fault Recovery

The indexing algorithm itself takes no special precautions about fault recovery. If the system crashes, disk space fills up, or the indexing program is aborted violently, the index file may be left in an inconsistent state.

A high level of fault tolerance can be achieved by doing all input and output to the index file in atomic transactions. Any method for transaction implementation can be used. The entire merge phase can be made one large transaction, or it can be divided into several transactions. A transaction can be ended and a new transaction started whenever the indexing algorithm restarts from root.

The new insertion algorithm, especially with the optimization of section 2.3.5, fits nicely in a transaction-based implementation. As seen by the transaction system, each node is modified only once, thus reducing transaction overhead. The current implementation of the indexing algorithm does in fact use transactions to recover from faults without the need for a full reindex.

Note that due to the buffering of keys in memory, errors may get reported much later than at the key at which they actually occur. If indexing does report failure, the current transaction should be aborted, and the application program can assume that none of the words added since the last merge have been added. (This is easily accomplished if the text database and the index use the same transaction system, because then the indexing transaction will actually be the same transaction as the database update transaction.)

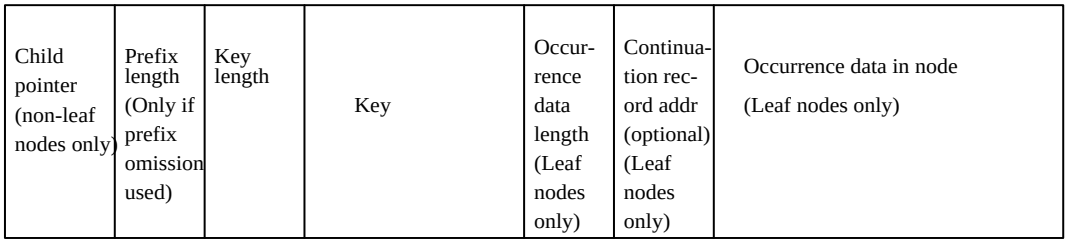

Figure 11: Structure of a key in index file.

3 Description of the current implementation

3.1 Overview

The new insertion algorithm has been implemented in C. About 1000 lines of code are used to implement the merging algorithm, about 500 lines for handling the in-memory data structure, and about 1000 lines for various support functions. The index is constructed on top of a simple shadow-paging transaction implementation (1500 lines). These figures do not include any higher level support such as query processing.

In the current implementation, the merge phase is implemented as a function which is called once for every key being added. The merge function keeps its state in variables global to it (actually these variables are part of the data structure used to represent the index file in memory). Separate functions are used to initialize the variables and to finish merging when all keys have been processed.

Figure 7 shows the structure of the index file. Figure 11 illustrates the structure of a single key in a node.

3.2 Interfaces

Most applications generate keys to be inserted in some arbitrary order, one key at a time. The the indexing system has the following entry points.

start_insert The indexing subsystem is informed that keys are to be inserted. Data structures needed for the memory buffers will be allocated and initialized.

insert Adds one key to the memory structure. If the key cannot be inserted in the memory structure, keys in memory will be merged

with the B-tree on disk, the memory will be freed, and the key will then be stored in memory.

end_insert The indexing subsystem is informed that no more keys are to be inserted at this time. Keys in memory will be merged to the tree on disk, and data structures will be freed.

In addition, inferfaces exist for searching, opening, closing and creating the index file.

3.3 The in-memory data structure

The buffers in memory are currently implemented as a red-black tree [43, pp. 187–199]. Some occurrence data is stored in the nodes themselves (since many words occur only once in the text that is processed between merges). Additional occurrence data is be stored in a linked list of blocks.

Experimental data indicates that indexing speed depends almost linearily on the amount of memory used. Reasonable sizes for the memory data structures are from a few hundred kilobytes to a few tens of megabytes. For most applications in the unix environment, a limit of 5 MB both provides good indexing performance and doesn't yet disturb other users. On large machines with big indexing runs, using even more memory may be justified. On the other hand, in the microcomputer world, especially in the MSDOS environment, one has to settle with what is available.

3.4 Overview of merge implementation

The merging algorithm (**merge_key**) as described below is applied to each key in memory, one at a time. The algorithm keeps some data in "global" variables between calls (these are actually fields of the structure used for representing the index). Merging must be started by executing **merge_start** and ended by calling **merge_end** after all keys have been processed with **merge_key**.

See appendix A for a more detailed pseudocode listing of the current implementation.

3.5 Performance

The performance of the current implementation was tested experimentally, both on an IBM RS6000 (192 MB physical memory, about 60 MIPS)

	English 1 MB	10 MB	100 MB	Finnish 11 MB
Words	172978	1731478	17551625	1712668
Number of merges	1	3	26	5
Unique keys/merge	13509	25434	27898	63161
Words/unique key/merge	12.8	22.7	24.2	5.4
Node reads	2	488	19437	2436
Total reads	6	514	20757	2519
Node writes	198	1014	19168	3406
Total writes	250	1738	28686	4008
Disk accesses/key	0.0015	0.0013	0.0028	0.0038
SPARCstation CPU time (min)	0:10	1:43	28:30	3:32
SPARCstation real time (min)	0:18	5:05	45:57	7:33
SPARC keys/CPU sec	17297	16810	10264	8078
SPARC keys/real sec	9609	5676	6366	3780
RS6000 CPU time (min)	0:04	0:41		1:13
RS6000 real time (min)	0:22	3:43		6:40
RS6000 keys/CPU sec	43244	42231		23461

Table 1: Indexing performance.

and a Sun SPARCstation 1 (24 MB physical memory, about 12 MIPS). The RS6000 was fairly heavily loaded, and the indexing process got only a fraction of the total CPU time. On the SPARCstation the indexing process was the only CPU-time consuming process.

The English test materials were collected from the ClariNet UseNet newsfeed over the years 1990 and 1991. It consists of newspaper-like articles. The test material contains only the bodies of the articles; all usenet news headers have been removed. Some of the articles appear more than once due to cross-posting to several newsgroups. A total of 315MB was collected; however, only up to 100MB was used in the final tests due to the very tight schedule for this thesis.

The Finnish test material consists of articles from Suomen Kuvalehti, a weekly magazine, over the years 1988 and 1989. The size of the Finnish material is about 11 megabytes.

The table in figure 1 summarizes the results of the test runs (5MB of buffer memory was used in all tests). The results indicate that the SPARCstation is able to index over 5000 keys/second (or over 100MB/hour) even for very large databases. Earlier results with the 315MB database suggest that indexing speed does not significantly change

Buffer memory size	300kB	1MB	2 MB	5 MB
Number of merges	86	18	8	3
Node reads	16405	3359	1468	500
Total reads	16799	3446	1510	523
Node writes	16907	3934	2027	1017
Total writes	17999	4720	2768	1738
Disk accesses/key	0.020	0.005	0.002	0.001
RS6000 CPU time (min)	1:36	0:54	0:46	0:42
RS6000 real time (min)	10:24	4:15	3:20	2:54

Table 2: The effect of buffer memory size on indexing speed (10MB English corpus).

after 100MB.

The results for the RS6000 are harder to analyze due to the heavy loading of the machine; also, the machine had a large memory which it could use to cache disk accesses. It could index over 40 000 words of English text per CPU second in the 10MB test. Best performance on a machine with this much main memory (192MB) would be achieved if very much buffer memory (on the order of 100MB) was used.

In all of the above tests, only about 0.003 disk accesses were needed per key. This compares very favorably with conventional B-tree insertion algorithms, which typically require one to three disk accesses per key – the new algorithm is about 1 000 times faster than the conventional algorithms.

The effect of buffer memory size on indexing speed is studied in table 2. All test were run on an IBM RS6000 using the 10MB English corpus.

The results in table 2 could be summarized by saying that the size of buffer memory affects indexing speed almost linearily. The reason for this is that buffering the keys in memory takes about constant time regardless of the number of merge phases, but the merge phases will have to access almost every node of the index regardless of buffer memory size. The number of merge phases is linearily determined by buffer memory size, and their duration is fairly constant (although it does increase with index file size; however, for the 100MB index, about 90 percent of the file size consists of continuation records and thus the growth is fairly slow).

The algorithms in [12] and [47] use similar methods, and are able to obtain most of the speed improvements. However, their algorithms do not address concurrent operation. Also, the optimizations presented in

section 2.3.5 improve over their methods by a factor of two or three.

3.6 Commercial references

Any commercial full text retrieval system contains many more features than have been described here. In fact, the low-level indexing algorithm is only a few percent of the total code size of the current commercial version. The needed support features include query processing, occurrence data extraction, database management, linguistic processing, other searching methods, user interface, and many others.

The current commercial system has about 1000 licenced end users in Finland and in Sweden, and has been in use since 1987. Applications include

- Newspaper editorial system archives. The system is installed in a few dozen newspapers in Finland and Sweden, ranging from one user to over hundred users. Among the larger are Keskisuomalainen, Norrtälje Tidningen, Suomen Tietotoimisto, Talentum, Keski-Uusimaa, Keski-Pohjanmaa.

- Medical information systems. This includes "Elektroninen Pharmaca Fennica", which contains a 15 MB database and a searching program, and is used by doctors and pharmacists on microcomputers.

- Microcomputer archival programs for searching for data from files on the users disk (or on a network server). The program (MOPSI) builds indexes from the files, understands most of the generally used word processor file formats, and has a very easy-to-use user interface. The program uses linguistic processing to handle the morphology of Finnish.

For more information on the commercial version, contact the author or New Generation Software (NGS) Oy, Inc., Tekniikantie 17, SF-02150 Espoo, Finland.

4 Conclusion

A very fast indexing algorithm has been presented in this thesis. The algorithm allows concurrent access to the index file; previous group update algorithms have not addressed concurrency. The new algorithm is about a thousand times faster than conventional B-tree insertion algorithms.

This paper has also provided new information on the effects of variable length keys on index file size, as well as on the effect of prefix omission in large indexes.

The new indexing algorithm has also proved successful in the "real world" and there exist several years of experience on its behaviour.

A Pseudocode listing of the implementation

A.1 Variables

`at_top`: `boolean` If this is true, there is no current node, and next key must be looked up starting from the root.

`at_dir_end`: `boolean` True indicates that the current node is the last (rightmost) node of the index. The purpose of this variable is to speed up the first merge phase on a newly created index.

`data_node`: `node` The `node` structure contains a copy of the disk block, and the following fields.

`node_address`: `disk_address`

`num_keys`: `integer`

`bytes_used`: `integer`

`is_leaf_node`: `boolean`

`data_node` describes the current leaf node, if one exists (as indicated by `at_top`).

`dir_node`: `node` This variable describes the parent of the current leaf node. The node is valid only if `at_top` is false.

`data_pos`: `pos` The `pos` structure points to a key in a `data_node` or `dir_node`. The following information is accessible through the `pos` structure.

`key_start`: `pointer` points to beginning of key in node

`key_end`: `pointer` to the end of the key

`at_end`: `boolean` is true if this points to end of node (in which case only the child pointer address is valid)

`child`: `disk_address` is the address of the child node (meaningful for non-leaf nodes only)

`key` is the full key (for prefix omission, see 2.3.6)

`key_len` is the number of characters in `key`

`key_num` the number of this key in the node (it is only stored here to simplify splitting operations)

`occurrence_data` provides access to the occurrence information associated with the key (the occurrence information may actually continue in continuation records; the implementation is hidden under a bit stream abstraction).

`data_pos` variable points to the current key in `data_node`. The value is valid only if `at_top` is false.

`dir_pos: pos` This variable points to the key under which `data_node` is positioned in `dir_node`. The value is valid only if `at_top` is false.

`dir_node_changed: boolean` `dir_node` is not written to disk immediately when changed. Instead, `dir_node_changed` is set to true, and `dir_node` will be saved the next time merging restarts from root. The motivation for this is that several children of `dir_node` may get split during the merge phase, and it is desirable that `dir_node` be written just once.

`new_node: node` The merging algorithm is implemented so that it doesn't actually modify `data_node`. Instead, it builds a new node on `new_node`. If splitting needs to be done, it is done by writing half of `new_node` to a new disk block and updating `dir_node`. When merging moves to a new node, the node being constructed on `new_node` will overwrite the old node. This approach removes the need to move data around multiple times as multiple keys are added to the node.

`new_pos: pos` always points to the last key on `new_node`. If prefix omission is used, prefix length can be computed from this key. Keys being moved or added to `new_node` are just appended to the end of this key and `new_pos` is moved to point to the new last key; similarily, new occurrence data is just appended to the last key.

`new_split_pos: pos` When appending to `new_node` crosses the halfway of the node, the current position is copied to `new_split_pos`. If the node later needs to be split, it can be split at `new_split_pos` without needing to compute the position.

A.2 Functions

```
merge_start:
    at_top := false;

merge_end:
    if ¬at_top
      then begin
             merge_finalize_node();
             if dir_node_changed
               then save_node(dir_node);
             unlock_node(dir_node);
             unlock_node(data_node);
           end

merge_key(key, occurrence_data):
  restart_from_root:
    if at_top
      then merge_find_from_top(key)
      else merge_find_next(key);
  restart_after_split:
    while ¬data_pos.at_end ∧ data_pos.key < key do
    if ¬merge_copy_current_key()
      then goto split;
    if data_pos.at_end ∨ data_pos.key ≠ key
      then begin
             if ¬merge_append_new_key(key)
               then goto split;
           end
      else begin
             if ¬merge_copy_current_key()
               then goto split;
           end
    merge_append_occurrence_data(occurrence_data);
    return;
  split:
    merge_split_current_node();
    if at_top
      then goto restart_from_root;
    goto restart_after_split;
```

```
merge_find_from_top(key):
    var node_address:  disk_address;
        have_prev_dir_node:  boolean;
        prev_dir_node:  node;
        prev_dir_pos:  pos;
  restart_from_root:
    at_dir_end := true;
    have_prev_dir_node := false;
    node_address := get_and_lock_root_address();
    dir_node := load_and_lock_node(node_address);
  process_dir_node:
    if (not enough space in dir_node to add at least one key)
      begin
        split_dir_node(have_prev_dir_node, prev_dir_node,
                       prev_dir_pos);
        if have_prev_dir_node
          unlock_node(prev_dir_node);
        unlock_node(dir_node)
        goto restart_from_root;
      end;
    if have_prev_dir_node
      then unlock_node(prev_dir_node)
      else unlock_root_address();
    dir_pos := find_key(dir_node, key);
    if ¬dir_pos.at_end
      then at_dir_end := false;
    data_node := load_and_lock_node(dir_pos.child);
    if ¬data_node.is_leaf
      then begin
             prev_dir_node := dir_node;
             prev_dir_pos := dir_pos;
             have_prev_dir_node := true;
             dir_node := data_node;
             goto process_dir_node;
           end;
    data_pos := find_first_key(data_node);
    clear_node(new_node);
    new_pos := find_first_key(new_node);
    at_top := false;
```

```
merge_find_next(key):
    if at_dir_end
      then return;
    if dir_pos.at_end
      then begin
             merge_finalize_node();
             if at_top
               then begin
                      merge_find_from_top(key);
                      return;
                    end;
             unlock_node(data_node);
             goto restart_from_top;
           end;
    if dir_pos.key > key
      then return;
    merge_finalize_node();
    if at_top
      then begin
             merge_find_from_top(key);
             return;
           end;
    unlock_node(data_node);
    dir_pos := find_next_key(dir_pos);
    while ¬dir_pos.at_end ∧ dir_pos.key ≤ key do
      dir_pos := find_next_key(dir_pos);
    if dir_pos.at_end
      then goto restart_from_top;
    data_node := load_and_lock_node(dir_pos.child);
    data_pos := find_first_key(data_node);
    clear_node(new_node);
    new_pos := find_first_key(new_node);
    return;
  restart_from_top:
    if dir_node_changed
      then save_node(dir_node);
    unlock_node(dir_node)
    merge_find_from_top(key);
```

```
merge_finalize_node:
  continue_after_split:
    if at_top
      then return;
    while ¬data_pos.at_end do
      if ¬merge_copy_current_key()
        then goto split;
    new_node.node_address := data_node.node_address;
    save_node(new_node);
    return;
  split:
    merge_split_current_node();
    goto continue_after_split;

merge_copy_current_key: boolean
    if (there isn't enough space in new_node)
      then return false;
    Copy key from data_pos to end of new_pos. Compute prefix
    from previous key if using prefix omission.
    data_pos := find_next_key(data_pos);
    new_pos := find_next_key(new_pos);
    return true;

merge_append_new_key(key):
    if (there isn't enough space in new_node)
      then return false;
    Create new key in new_node. Compute prefix from previous
    key if using prefix omission.
    new_pos := find_next_key(new_pos);
    return true;

merge_append_occurrence_data(occurrence_data):
    while (data in node + new data length geq
        size of continuation record) do
      Create a continuation record.
    Append data to node.

split_current_node:
    Save data from new_node up to new_split_pos in a new node.
    Move data in new_node after new_split_pos to beginning of node.
```

```
    If prefix omission in use, expand prefix of new first key.
    Update new_pos to reflect the movement.
    Insert the new first key in the parent node with child pointer to
    the newly created node.
    if (there isn't enough space in parent node for at least
      one more key)
      then begin
             Remove data that has been processed from data_node.
             save_node(dir_node);
             unlock_node(dir_node);
             save_node(data_node);
             unlock_node(data_node);
             at_top := true;
           end

split_dir_node(have_prev_dir_node, prev_dir_node,
                                   prev_dir_pos):
    Create a new code containing about half of the data in
      dir_node.  Save the new node.
    if have_prev_dir_node
      then Insert a key in prev_dir_node.
      else Create new root node with just one key.;

find_first_key(node): pos
    Return pos that points to the first key of node.

find_next_key(pos): pos
    Return pos that points to the next key.

find_key(node, key): pos
    Return pos that points to the smallest key greater than
    or equal to key.
```

References

[1] Baeza-Yates, R. *Expected Behaviour of* B^+*-trees under Random Insertions*, Acta Informatica, Vol. 26, 1989, pp. 439–471.

[2] Bayer, R. *Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms*, Acta Informatica, Vol. 1, pp. 290–306.

[3] Bayer, R., McCreight, E. M. *Organization and Maintenance of Large Ordered Indexes*, Acta Informatica, Vol. 1, No. 3, 1972, pp. 173–189.

[4] Bayer, R., Schkolnick, M. *Concurrency of Operations on B-trees*, Acta Informatica, Vol. 9, No. 1, 1977, pp. 1–21.

[5] Bayer, R., Unterauer, K. *Prefix B-trees*, ACM Transactions on Database Systems, Vol. 2, No. 1, 1977, pp. 11–26.

[6] Blair, David C. *Full Text Retrieval: Evaluation and Implications*, Int. Classif. Vol. 13, No. 1, 1986, pp. 18–23.

[7] Bratley, P., Choueka, Y. *Processing truncated terms in document retrieval systems*, Inf. Processing & Management, Vol. 18, 1982, 257–266.

[8] Brown, M. R. *A Storage Scheme for Height-Balanced Trees*, Information Processing Letters, Vol. 7, No. 5, 1978, pp. 231–232.

[9] Choueka, Y., Fraenkel, A., Klein, S. *Compression of Concordances in Full-Text Retrieval Systems*, Proc. ACM SIGIR 1988, pp. 597–612.

[10] Comer, D. *The Ubiquitous B-tree*, ACM Computing Surveys, Vol. 11, No. 2, 1979, pp. 121–137.

[11] Cotton, P. L. *Where full-text is viable*, Online Review, Vol. 11, No. 2, 1987, pp. 87–93.

[12] Cutting, D., Pedersen, J. *Optimizations for Dynamic Inverted Index Maintenance*, Proc. ACM SIGIR 1990, pp. 405–411.

[13] Dubois, C. P. R. *Free text vs. controlled vocabulary; a reassessment*, Online Review, Vol. 11, No. 4, 1987, pp. 243–253.

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

[15] Faloutsos, Cristos. *Access Methods for Text*, ACM Computing Surveys, Vol. 17, No. 1, 1985, pp. 49–74.

[16] Ford, R., Calhoun, J. *Concurrency Control Mechanism and the Serializability of Concurrent Tree Algorithms*, Proceedings of the ACM Sigact-Sigmod Symposium on the Priciples of Database Systems 1984, pp. 51–60.

[17] Goodman, N., Shasha, D. *Semantically-based Concurrency Control for Search Structures*, Proc. ACM Conf. on Principles of Database Systems 1985, pp. 8–19.

[18] Guibas, L. J., Sedgewick, R. *A Dichromatic Framework for Balanced Trees*, Proc. 19th IEEE Symp. on Foundations of Computer Science 1978, pp. 8–21.

[19] Held, G., Stonebraker, M. *B-trees Re-examined*, Communications of the ACM, Vol. 21, No. 2, 1978, pp. 139–143.

[20] Johnson, T., Shasha, D. *Utilization of B-trees with Inserts, Deletes and Modifies*, Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems 1989, pp. 235–246.

[21] Johnson, T., Shasha, D. *A Framework for the Performance Analysis of Concurrent B-Tree Algorithms*, Proc. 9th Symp. on Principles of Database Systems, 1990.

[22] Karlsson, Fred. *Suomen kielen äänne- ja muotorakenne*, WSOY, 1982.

[23] Karlsson, F. (ed.) *Computational Morphosyntax*, Report on Research 1981–84, Publications of the Department of General Linguistics, No. 13, University of Helsinki, 1985.

[24] Karlsson, F., Voutilainen, A., Heikkilä, J., Anttila, A. *SIMPR: Structured Information Management: Processing and Retrieval (ESPRIT II Project 2083)*, SIMPR Document no: SIMPR-RUCL-1990-13.4e, Research Unit for Computational Linguistics, University of Helsinki, Finland, 1991.

[25] Keller, A. M., Wiederhold, G. *Concurrent Use of B-trees with Variable-Length Entries*, ACM SIGMOD Record, Vol. 17, No. 2, 1988, pp. 89–90.

[26] Korth, H. F., Silberschatz, A. *Database System Concepts*, McGraw-Hill, 1988.

[27] Koskenniemi, Kimmo. *TWO-LEVEL MORPHOLOGY: A General Computational Model for Word-Form Recognition and Production.* Doctoral dissertation. Publications of the Department of General Linguistics, University of Helsinki, No. 11, 1983.

[28] Knuth, D. *The art of computer programming, Vol. 3: sorting and searching*, Addison-Wesley, 1973.

[29] Kujala, Teija. *Joustavan tiedonhaun tekniikat*, Laudaturtutkielma, sarjanumero C-1991-66, Tietojenkäsittelyopin laitos, Helsingin yliopisto, 1991.

[30] Kwong, Y. S., Wood, D. *Concurrent Operations in Large Ordered Indexes*, Proc. International Symposium on Programming, Paris, April 1980, pp. 207–222.

[31] Kwong, Y. S., Wood, D. *Approaches to Concurrency in B-trees*, Proc. 9th Symposium on Mathematical Foundations of Computer Science 1980, pp. 402–413.

[32] Kwong, Y. S., Wood, D. *A New Method for Concurrency in B-Trees*, IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, 1982, pp. 211–222.

[33] Lanin, V., Shasha, D. *A Symmetric Concurrent B-tree Algorithm*, Proceedings of the Fall Joint Computer Conference 1986, pp. 380–389.

[34] Lehman, P. L., Yao, S. B. *Efficient Locking for Concurrent Operations on B-trees*, ACM Transactions on Database Systems, Vol. 6, No. 4, 1981, pp. 650–670.

[35] Lomet, D. B. *Digital B-trees*, Proc. International Conference on Very Large Data Bases 1981, pp. 333–344.

[36] Matsliach, G. *Performance Analysis of File Organizations that Use Multi-Bucket Data Leaves with Partial Expansions*, Proc. 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1991, pp. 164–180.

[37] Mond, Y., Raz, Y. *Concurrency Control in* B^+*-trees Databases Using Preparatory Operations*, Proc. VLDB 1985, pp. 331–334.

[38] Nievergelt, J. *Binary Search Trees and File Organization*, ACM Computing Surveys, Vol. 6, No. 3, 1974, pp. 195–207.

[39] Nurmi, O., Soisalon-Soininen, E., Wood, D. *Concurrency Control in Database Structures with Relaxed Balance*, Proc. ACM Principles of Database Systems 1987, pp. 170–176.

[40] Nurmi, O., Soisalon-Soininen, E. *Uncoupling Updating and Rebalancing in Chromatic Binary Search Trees*, Proc. 10th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, 1991, pp. 192–198.

[41] Sacks-Davis, R., Kent, A., Ramamohanarao, K. *Multikey Access Methods Based on Superimposed Coding Techniques*, ACM Transactions on Database Systems, Vol. 12, No. 4, 1987, pp. 655–696.

[42] Sagiv, Y. *Concurrent Operations on B*-Trees with Overtaking*, Journal of Computer and System Sciences, Vol. 33, 1986, pp. 275–296.

[43] Sedgewick, R. *Algorithms.* Addison-Westley, 1984.

[44] Silberschatz, A., Stonebraker, M., Ullman, J. D. *Database Systems: Achievements and Opportunities*, ACM SIGMOD Record, Vol. 19, No. 4, 1990.

[45] Srinivasan, V., Carey, M. *Performance of B-Tree Concurrency Control Algorithms*, Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, 1991, pp. 416–425.

[46] Srinivasan, V., Carey, M. J. *On-Line Index Construction Algorithms*, Computer Sciences Technical Report #1008, University of Wisconsin-Madison, March 1991.

[47] Srivastava, J., Ramamoorthy, C. V. *Efficient Algorithms for Maintenance of Large Database Indexes*, Proc. IEEE Conf. on Data Engineering 1988, pp. 402–408.

[48] Tenopir, Carol. *Full text database retrieval performance*, Online Review, Vol. 9, No. 2, 1985, pp. 149–164.

[49] Wright, William. *Some Average Performance Measures for the B-tree*, Acta Informatica, Vol. 16, 1985.

[50] Yao, Andrew. *On Random 2-3 Trees*, Acta Informatica, Vol. 9, 1978, pp. 159–170.

[51] Zhang, B., Hsu, M. *Unsafe Operations in B-trees*, Acta Informatica, Vol. 26, 1989, pp. 421–438.

[52] Zobel, J., Thom, J., Sacks-Davis, R. *Efficiency of Nested Relational Document Database Systems*, Proc. VLDB 1991, pp. 91–102.