An Algorithm for Full-Text Indexing

Tatu Ylönen*

Abstract

Conventional B-tree insertion algorithms typically require several disk accesses per insertion. This is inconveniently slow for full text databases. *Group update* can significantly speed up insertion. The idea is to collect and sort many keys in memory, and then merge them with the B-tree on disk.

This paper presents a concurrent group update algorithm for B^+-trees; in addition to allowing concurrent operation, the new algorithm provides 90 % CPU time savings compared to existing group update algorithms. Experimental results show that the algorithm uses on the order of 0.003 disk accesses per key in full-text indexing.

1 Introduction

In full text databases, any word in the stored documents can be used as a search key. A database of 100 megabytes of English text contains about 17 000 000 words and 210 000 unique word forms. The huge number of keys is problematic in conventional B-tree based indexing systems; assuming each insertion takes three disk accesses [2], the initial indexing of the 100 megabyte database would take about a week. Databases of several gigabytes are not uncommon; also, interactive applications need to be able to add new documents (often containing several thousand words) to the database while the user is waiting.

Signature files [4, 14] allow fairly efficient indexing and retrieval; however, they do not allow searching for truncated keys, which is very important in languages that have a richer morphology than English (for example, a Finnish noun can have 2 000 inflectional forms, and a verb as

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

many as 12 000 forms [9, p. 44]). Inverted files implemented as B-trees allow range queries (which can be used for searching inflected forms), but conventional insertion methods are fairly slow.

As pointed out in [18], B-tree indexing can be done much faster if the keys to be added are sorted in memory, and then merged with the B-tree on disk in sorted order. Srivastava and Ramamoorthy [18] presented two algorithms for merging the keys in memory to the tree on disk. They introduced the term *group update* for this approach. However, their algorithms are quite complicated and are not directly applicable to multi-user environments.

Cutting and Pedersen [3] presented a much simpler and equally efficient method for merging the keys in memory to the tree on disk. Their idea is simply to keep the nodes on the path from the root to the previously accessed leaf node in cache, and use the normal B-tree insertion algorithm to insert the keys in sorted order. This method is easy to implement, but cannot be used if several computers on a network need to access the index file simultaneously.

This paper presents a new algorithm for merging the keys to the B-tree. Unlike previous algorithms, this algorithm allows multiple concurrent updates and searches. This paper also introduces the idea of incremental construction of leaf nodes; it is shown that this optimization can reduce CPU time requirements by about 90 percent compared to previous group update algorithms. Compared to conventional B-tree insertion algorithms, the new algorithm is about 1 000 times faster when very many keys are to be inserted at the same time.

The new algorithm could also be used for deletion, and it would be possible to collect insertions and merges simultaneously in memory, as was done in [17].

2 A fast grouped B-tree insertion algorithm

This section presents a new grouped insertion algorithm for variable key length B^+-trees. The presentation is oriented towards full text databases, but the same algorithm could be used with more conventional databases as well.

Figure 1 illustrates the structure of the index file. A key corresponds to a word in a document, and occurrence data refers to application-dependent data used to describe in which document and where in the

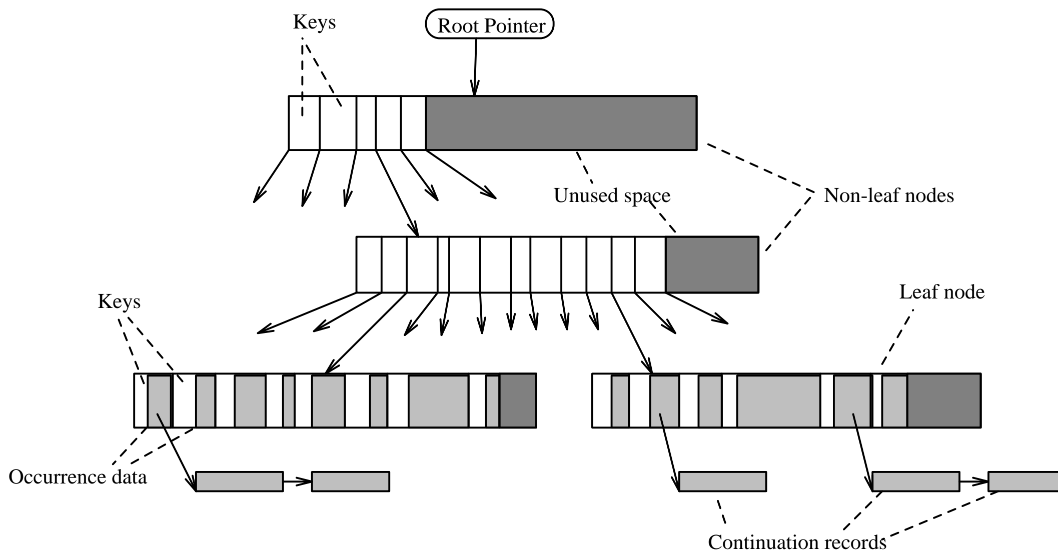

Figure 1: Index file structure.

document the word occurred. Some occurrence data is stored with the key; if a key has very many occurrences, it may have a list of continuation records containing more occurrence data.

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; see [6, 12]).

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

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 (it is known 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 and 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.

2.1 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
- combine multiple occurrences of a single key.

Keys are inserted to the memory buffer in an arbitrary order (quite often the textual order), and read out in the order used by the B-tree (usually the alphabetical order). The buffer can be implemented as a red-black tree [16, pp. 187–199], or using any other convenient data structure.

2.2 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 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. Incremental construction of leaf nodes results in further CPU time savings.

An outline of the algorithm is shown in figure 2.

```
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 2: The new merging algorithm.

If needed, non-leaf nodes are split when searching for a leaf node starting from the root. This ensures that a node can always be split and there will be enough space in the parent. There is no need to keep the ancestors of the parent of the leaf node locked. Nodes are always locked in a strict top-down order, and deadlocks are thus avoided.

2.2.1 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 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.

Figure 3 illustrates the algorithm for finding a leaf node.

```
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 3: Finding the leaf node and splitting non-leaf nodes.

2.2.2 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 2.4).

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 optimize inserting to the rightmost node of the index). Normally the next node will be found under the same parent as the previous node. It is locked and loaded, and it becomes the new current node.

Figure 4 illustrates the algorithm for finding the leaf node for the next key.

2.2.3 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 leaf 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.

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 3.*
else *Lock and load the new leaf node.*

Figure 4: Finding the node where the next key belongs.

2.2.4 File locking

All locks are allocated in top-down 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.

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 a continuation record chain will have the node pointing to the chain locked.

Searching should also allocate its locks in top-down order. 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.

2.3 Continuation records for occurrence data

It is best to store some of the occurrence data of a key in the node with the key [3]. 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 a 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 chain when there are sufficiently many occurrences in the node.

2.4 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 [10] for a different approach.)

The cost of multiple copying can be approximated by

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

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 [7], $\bar{U}_n$ is about $0.68S_n$ (S_n is the size of a node)). Thus,

$$\bar{t}_{mc} = \bar{N}_i \frac{0.34S_n}{C}. \tag{2}$$

A single copy of the node could be made in time

$$\bar{t}_{sc} = \frac{0.68S_n}{C}. \tag{3}$$

Assuming[1] $\bar{N}_i = 50$, $S_n = 8192$ and $C = 5MB/s$ (the data is usually unaligned), $\bar{t}_{mc} \approx 28ms$ and $\bar{t}_{sc} \approx 1ms$.

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

$$\bar{t}_t = \bar{t}_r + \bar{t}_p + \bar{t}_c + \bar{t}_w. \tag{4}$$

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

[1] 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.

$\bar{t}_w = 3ms$ (asynchronous writes), $\bar{t}_{t_{mc}} \approx 44ms$ and $\bar{t}_{t_{sc}} \approx 17ms$. CPU time usage $(\bar{t}_p + \bar{t}_c)$ is $31ms$ for multiple copying, and $4ms$ if multiple copying can be avoided. This is a saving of almost 90 percent. In real time, the merge phase becomes about twice as fast as with earlier algorithms.

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 in the node, 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.

If there isn't enough space in the new node to add a key or its occurrence data, the node must be split. This can be done 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.

Also, special care must be taken 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.

2.5 Optimizing index file size

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

$$S_i = S_t + S_c. \quad (5)$$

Using the results of [7],

$$S_t \approx \frac{N_u(\bar{L}_k + \bar{L}_b + \bar{L}_{on})}{0.68} \quad (6)$$

and

$$S_c \approx N_u \bar{L}_{oc} \quad (7)$$

The values $\bar{L}_{on}$ and $\bar{L}_{oc}$ are sometimes hard to obtain, and a reasonable estimate of the total index file size can also be obtained from

$$S_t \approx \frac{N_u(\bar{L}_k + \bar{L}_b)}{0.68} + N_o \bar{L}_o \quad (8)$$

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

In experimental analysis it was found that a 315 MB corpus of English text[2] 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.

Using equation 8 and assuming $N_u = 210\,000$, $N_o = 50\,000\,000$, $\bar{L}_k = 4.26$, $\bar{L}_b = 3$, and $\bar{L}_o = 4$, the size of the index file is approximately 202 megabytes. 2 megabytes are needed for 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 size of the index file, 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.

[1] suggests the *prefix omission method* for reducing the size of the index file. The idea is to store with each key the number of characters in common with the previous key, and only store the different characters. If

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

$\bar{L}_p$ is the average length of the common prefix, this methods saves about $S_p = N_u \bar{L}_p$ bytes. However, as noted earlier, the keys only use up about one percent of the size of of the index file. Thus, even if prefix omission could reduce the average stored key length to zero, no significant savings would be obtained. This results suggests that using prefix omission is waste of CPU time for large databases.

Preparatory splitting causes some space overhead in non-leaf nodes (note that it does not affect leaf nodes, as they do not have children that could be split). 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

$$N_n = \lceil \log_{\bar{b}} \frac{N_u(\bar{L}_k + \bar{L}_b + \bar{L}_{on})}{0.68 S_n} \rceil + 1 \tag{9}$$

The amount of disk space wasted by preparatory splitting,

$$S_w \approx N_n L_{max} \tag{10}$$

Experimental data suggests that with $S_n = 8192$, $\bar{L}_{on} \approx 50$ and the braching factor for non-leaf nodes $\bar{b} \approx 500$. Using equation 9 with the earlier example, there are about 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. Thus, the amount of space wasted by preparatory splitting is completely insignificant (cf. [8]).

3 Performance

The algorithm presented in this paper was implemented in C (about 3000 lines of code). The implementation was tested on an IBM RS6000 (192 MB physical memory, about 60 MIPS) and a Sun SPARCstation 1 (24 MB physical memory, about 12 MIPS). The RS6000 was 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 material was collected from the ClariNet newsfeed over the years 1990 and 1991. It consists of newspaper-like articles. The

	Eng 1 MB	10 MB	100 MB	Fin 11 MB
Words	172 978	1 731 478	17 551 625	1 712 668
Number of merges	1	3	26	5
Unique keys/merge	13 509	25 434	27 898	63 161
Words/unique key/merge	12.8	22.7	24.2	5.4
Node reads	2	488	19 437	2 436
Total reads	6	514	20 757	2 519
Node writes	198	1 014	19 168	3 406
Total writes	250	1 738	28 686	4 008
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	17 297	16 810	10 264	8 078
SPARC keys/real sec	9 609	5 676	6 366	3 780
RS6000 CPU time (min)	0:04	0:41		1:13
RS6000 real time (min)	0:22	3:43		6:40
RS6000 keys/CPU sec	43 244	42 231		23 461

Table 1: Indexing performance.

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 disk space limitations.

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.

Table 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 5 000 keys/second (or over 100MB/hour) even for large databases. Earlier results with the 315MB database suggest that indexing speed does not significantly change 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; most of the CPU time was spent collecting and sorting the keys in memory.

In all of the above tests, only about 0.003 disk accesses were needed

Buffer memory size	300kB	1MB	2 MB	5 MB
Number of merges	86	18	8	3
Node reads	16 405	3 359	1 468	500
Total reads	16 799	3 446	1 510	523
Node writes	16 907	3 934	2 027	1 017
Total writes	17 999	4 720	2 768	1 738
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).

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 [3] and [18] use similar methods, and are able to obtain most of the speed improvements. However, their algorithms do not address concurrent operation. The optimization presented in section 2.4 improves over their methods by a factor of two or three, and avoiding a full search from the root for every node results in further speedups.

4 Conclusion and further research

A very fast B-tree insertion algorithm has been presented. The algorithm allows concurrent access to the index file; previous group update algorithms have not addressed concurrency. The algorithm improves in-

dexing speed by a factor of about 1000 over conventional B-tree insertion algorithms, and reduces CPU time usage by 90 percent compared to earlier group update algorithms by incrementally constructing new versions of leaf nodes.

It was also shown that top-down rebalancing does not cause a significant space penalty, and that the prefix omission technique [1] does not result in significant space savings in large indexes.

The present algorithm is based on top-down rebalancing of the B-tree. B^{link}-trees [11] or relaxed balance schemes [15, 5, 13] could probably be used to improve the behaviour of the new algorithm in high concurrency environments; however, for most applications of full-text databases even the current level of concurrency would be quite sufficient.

References

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

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

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

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

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

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

[7] Johnson, T., Shasha, D. *Utilization of B-trees with Inserts, Deletes and Modifies*, Proc. ACM Principles of Database Systems 1989, pp. 235–246.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

[14] 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.

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

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

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

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