Prefix B-Trees

RUDOLF BAYER and KARL UNTERAUER
Technische Universität München

Two modifications of B-trees are described, simple prefix B-trees and prefix B-trees. Both store only parts of keys, namely prefixes, in the index part of a B^*-tree. In simple prefix B-trees those prefixes are selected carefully to minimize their length. In prefix B-trees the prefixes need not be fully stored, but are reconstructed as the tree is searched. Prefix B-trees are designed to combine some of the advantages of B-trees, digital search trees, and key compression techniques while reducing the processing overhead of compression techniques.

Key Words and Phrases: B-trees, key compression, multiway search trees
CR Categories: 3.73, 3.74

1. SIMPLE PREFIX B-TREES

We assume that the reader is familiar with B-trees [4, 8] and with a variation of B-trees, called B^*-trees [8, 10]. In B^*-trees the records of a file together with the keys identifying them are only stored in leaf nodes of the tree structure. We call the nonleaves branch nodes or branch pages. Leaves can be linked to their neighbors to allow sequential processing of the leaves without using the branch nodes of the B^*-tree.

We call the part of a B^*-tree consisting only of the branch nodes the B^*-index and the ordered set of leaves the B^*-file. In the B^*-index some keys, which appear again in the B^*-file with their associated records, are repeated without their records. The following observation is important for the rest of this paper: The keys stored in the B^*-index are only used to direct the search algorithm and to determine in which subtree of a given branch node a key and its associated record will be found, if they are in the tree at all.

It is now a fairly obvious observation that we need not necessarily use the keys in the B^*-file to construct the B^*-index. Instead we can use other strings, constructed to have desired properties, for building up the equivalent of the B^*-index of a B^*-tree.

To give a simple example, we assume that a leaf is already full and contains keys

Most of the research reported in this paper was performed while the first author was visiting at IBM Research Laboratory, San Jose, CA 95193. The research of the second author was supported by the Sonderforschungsbereich 49—Elektronische Rechenanlagen und Informationsverarbeitung—of the Deutsche Forschungsgemeinschaft.
Authors' address: Institut für Informatik der Technischen Universität München, Arcisstrasse 21, D-8000 München 2, West Germany.

Bigbird, Burt, Cookiemonster, Ernie, Snuffleopogus

In order to insert the key "Grouch" with its record, we must split this leaf into two as follows:

Bigbird, Burt, Cookiemonster	Ernie, Grouch, Snuffleopogus

Instead of storing the key "Ernie" in the index, it obviously suffices to use one of the one-letter strings "*D*", "*E*" for the same purpose. In general we can select any string s with the property

$$\text{Cookiemonster} < s \leq \text{Ernie} \tag{1}$$

and store it in the index part to separate the two nodes. We call such a string s a *separator* (between Cookiemonster and Ernie). It seems prudent to choose one of the shortest separators.

Note. If the keys are words over some alphabet and the ordering of the keys is the alphabetic order, then the following property, called the *prefix property*, holds:

Let x and y be any two keys such that $x < y$. Then there is a unique prefix $\bar{y}$ of y such that (a) $\bar{y}$ is a separator between x and y, and (b) no other separator between x and y is shorter than $\bar{y}$. For the rest of this paper, we assume that the prefix property holds.

The technique of moving a shortest separator to the father node when a node is being split can be used only for splitting leaves, not branch nodes. When a branch node is being split, one of the separators on that node must be moved up one level in the tree.

As mentioned before, a B^*-tree can be considered as consisting of a B^*-index and a B^*-file. The B^*-index itself is just a conventional B-tree of a subset of the keys in the B^*-file together with the maintenance algorithms for B-trees described in [4].

Definition. A *simple prefix B-tree* is a B^*-tree in which the B^*-index is replaced by a B-tree of (variable length) separators.

Note. Since a key in a B^*-index is also a separator, although not necessarily a shortest possible separator, the class of simple prefix B-trees contains the class of B^*-trees.

Except for the slight complication of always having variable length separators, the search algorithm for simple prefix B-trees is exactly the same as for B^*-trees.

Split interval. The performance bottleneck of our trees is the number of accesses to the backup store needed for INSERT, DELETE, and RETRIEVE operations. This number is essentially determined by the height of the tree since the pages along the retrieval path for some key x from the root to some leaf are always needed for those three operations.

Performance can therefore be improved by making the trees as flat as possible, which can be achieved by making the branching degree of the nodes, especially in the upper parts of the tree (i.e. near the root), as high as possible. This branching degree is determined by the number of (separator, pointer) pairs that can be stored on a fixed size page. Pointers are generally rather short and have a fixed

size of a few bytes. Keys, however, tend to be rather long. The branching degree can therefore be increased by decreasing the length of the separators. This essentially is the rationale for storing only shortest separators instead of full keys in the index part of a simple prefix B-tree.

This idea can now be carried one step further if we do not insist on splitting a leaf precisely in the middle. Instead we could allow a certain *split interval* around the middle (or the median key) of a splitting leaf within which a split point (between two adjacent keys x_{i-1} and x_i) should be chosen so as to minimize the length of the shortest separator s_i. Note that because of the prefix property the resulting separator s_i can still be chosen as a prefix of the key x_i. The size of the split interval is determined by a parameter σ_l. σ_l is simply the number of separators (or bytes) around the middle of the page which we consider for choosing a suitable split point.

The same idea can be applied to splitting branch nodes. A certain interval of size σ_b around the middle of a page is considered for choosing a split point such that a shortest separator within this interval is moved to the father page.

Effect of σ_l and σ_b. An increase of σ_l should decrease the average length of the separators in the index part of the tree, thereby reducing the number of nodes in the index part. An increase of σ_b should favor the shorter separators in the index to be located near the root, thereby increasing the branching degree of nodes near the root, where a high branching degree is most beneficial.

Increasing both σ_l and σ_b causes two effects working against each other:

(a) It tends to decrease the height of the index part for the reasons just described.

(b) The storage utilization decreases (there can now be pages less than half full), which requires more pages in the file part and more but shorter entries in the index part of a tree.

We have not analyzed the influence of σ_l or σ_b on the performance of the trees. We expect such an analysis to be quite involved and difficult. We are quite confident, however, that small split intervals improve performance considerably. Sets of keys that arise in practical applications are often far from random, and clusters of similar keys differing only in the last few letters (e.g. plural forms) are quite common.

As an example, consider the key sequence

"On, Part, Problem, Problems, Solution, Solutions"

arising in the tree of Figure 1. Splitting this sequence in the middle between the third and fourth key would yield "Problems" as the shortest separator. Allowing a split point to be chosen one key to the left or to the right yields "*Pr*" or "*S*" as separators. The split point between "Problems" and "Solution" yields the shortest separator "*S*", which is a prefix of the full key "Solution" and appears in the tree of Figure 1.

2. ALGORITHMS FOR SIMPLE PREFIX B-TREES

2.1 Search Algorithm

This algorithm is the same as for B^*-trees with variable length keys.

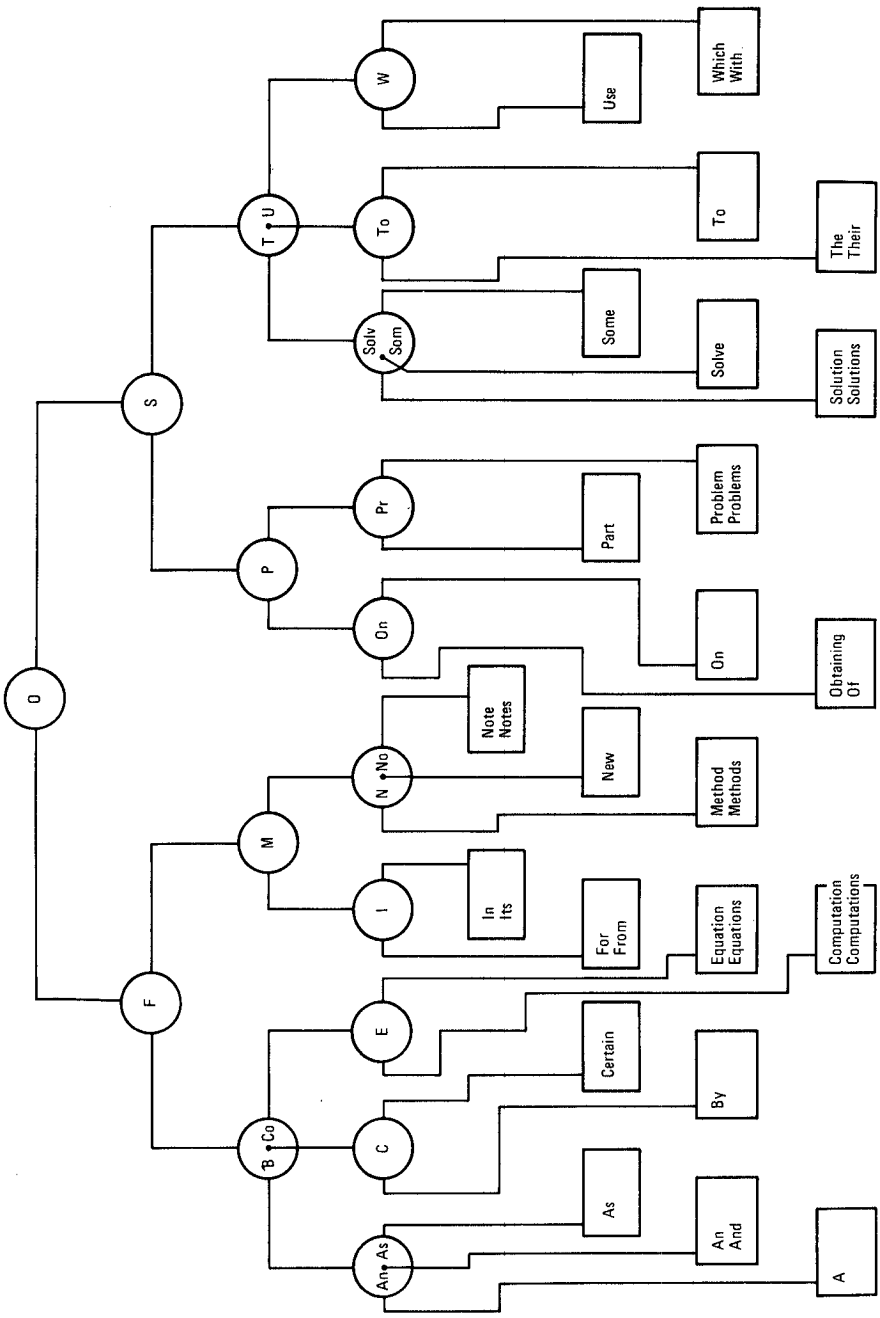


Fig. 1. Example of a simple prefix B-tree

2.2 Sequential Processing

This is very easy and efficient using either the index or the linked list of the leaves.

2.3 Insertion

An important part of the insertion algorithm is the search algorithm to determine the leaf on which the insertion must be performed. If this leaf must be split, choose within the split interval a split point yielding the shortest separator s, which as we know can be chosen to be a prefix of a key. Insert s into the index part, propagating splits toward the root if necessary. To split a branch node choose again a shortest separator within the split interval and move it to the father node.

2.4 Deletion

Deletion of a key and its associated record is always made from a leaf. Unless a merge or an underflow [4] of the leaf is required, the index part of the tree need not be affected by deletions. Thus deletions are simpler than in the original B-trees. If the deletion causes two leaves to be merged, simply delete the corresponding separator from the index part. This will always be a deletion from a leaf of the index part, which is organized as a B-tree. Thus these deletions are special cases and much simpler than general deletions from B-trees. Other separators in the index part of the tree are not affected by such a deletion.

Note. If the largest or the smallest key on a leaf is deleted, then a separator in the index part might be replaced by a shorter separator. Although this can easily be done, it is hardly worthwhile.

2.5 Overflow

So far we have disregarded overflows caused by insertions or deletions (where they are sometimes called underflows [4]). An overflow is performed by moving keys and records or separators from a node to a brother node in order to avoid splits or to balance storage utilization. Using variable length separators instead of fixed length keys means, however, that during an overflow a separator in the father page may be replaced by either a longer or a shorter separator. Obviously split intervals should also be applied to overflows.

As opposed to the original B-tree proposal based on fixed length keys, overflows may now propagate and cause further splits, merges, or overflows if a separator is replaced by a longer or shorter separator. Obviously such propagation will be infrequent.

Note. Rear compression of keys, described in [8] and [9], is a technique similar to using shortest prefixes. Rear compression, however, does not use the important device (see Section 3) of the split interval.

3. EXAMPLE OF A SIMPLE PREFIX B-TREE

To construct an example of a simple prefix B-tree, the keys of the KWIC index [8, p. 437] were inserted into the tree in the following random order: part, their, solve, for, to, an, some, certain, new, equations, problems, on, methods, as, obtaining, a, its, solutions, use, notes, by, computation, in, of, problem, from, note, and, solution, which, with, method, computations, the, equation.

The nodes are assumed to be able to hold up to two keys or separators and up to three pointers. When a leaf becomes too full, i.e. contains exactly three keys, a split after the first or second key is allowed to get the shortest separator. Branch nodes are split by moving the middle separator to the father. Thus our example demonstrates only the effect of the split interval on leaves; it ignores the interplay among a fixed page size for a node, the variable length separators, and the resulting variable branching degrees of the branch nodes. After inserting all keys, we obtain the tree of Figure 1.

The following observations can be made about our example. Any actual data will not quite satisfy the assumptions made for the analysis of simple prefix B-trees: that the keys are chosen at random. Thus the average length of a shortest separator for our set of key words is 2.88 rather than 1.47 as would be expected for random keys (see Section 6). This is due to the obvious phenomenon of several nouns appearing both in singular and plural forms. The long plural forms appear necessarily in the set of shortest separators. Our proposal of choosing a split point within a certain interval almost compensated for this effect, yielding an average length of 1.54 for the separators actually appearing in the index part of the tree. This is in good agreement with the expected length of 1.47.

As mentioned before, the influence of the size of the split interval on the length of separators is an open problem. We conjecture that the main benefits of the scheme can be obtained by a rather small interval. This means that the good storage utilization of B-trees in general will not be degraded appreciably.

4. PREFIX B-TREES

In this section we describe a modification of simple prefix B-trees with the goal of further reducing the size, mainly the height of the index part, of such trees. Assume that the ordering of the keys is the lexicographical order according to the collating sequence of the alphabet over which the keys are defined.

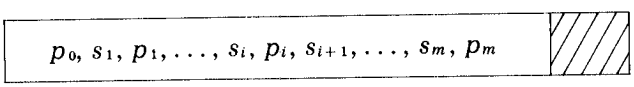
For an arbitrary page P, let $T(P)$ be the subtree of index and leaf pages with root P. Reconsider the index part of a simple prefix B-tree. The tree structure determines for each page P a largest lower bound $\lambda(P)$ and a smallest upper bound $\mu(P)$ such that, for all keys x or separators s which are or might be stored in $T(P)$, the following holds:

$$\lambda(P) \leq x < \mu(P), \quad \lambda(P) \leq s < \mu(P).$$

Let l_0 be the smallest letter in the alphabet and let ∞ be larger than any letter:

$$\lambda(R) = l_0, \quad \mu(R) = \infty.$$

To define λ and μ for other nodes, let P be a branch node with lower and upper bounds $\lambda(P)$ and $\mu(P)$, respectively, and the following structure:



$p_0 \ldots p_m$ are pointers to the sons of P which are branch or leaf nodes; $s_1 \ldots s_m$ are separators, s_m being the last one on P. Then $\lambda(P(p_i))$ and $\mu(P(p_i))$, also

denoted by $\lambda(p_i)$ and $\mu(p_i)$, are defined as:

$$\lambda(p_i) = \begin{cases} s_i \text{ for } i = 1, 2, \ldots, m, \\ \lambda(P) \text{ for } i = 0, \end{cases} \qquad \mu(p_i) = \begin{cases} s_{i+1} \text{ for } i = 0, 1, \ldots, m-1, \\ \mu(P) \text{ for } i = m. \end{cases}$$

Then obviously all separators or keys in $T(p_i)$ must have at least a common prefix $\kappa(p_i)$, which can be defined as follows: Let $\overline{k_i}$ be the longest common prefix (possibly the empty string) of $\lambda(p_i)$ and $\mu(p_i)$. Then

$$\kappa(p_i) = \begin{cases} \overline{k_i}l_j & \text{if } \lambda(p_i) = \overline{k_i}l_j z \text{ and } \mu(p_i) = \overline{k_i}l_{j+1}, \text{ where } z \text{ is an arbitrary} \\ & \text{string and the letter } l_{j+1} \text{ follows } l_j \text{ immediately in the col-} \\ & \text{lating sequence,} \\ \overline{k_i} & \text{otherwise.} \end{cases}$$

$\lambda(p_i)$ and $\mu(p_i)$, and therefore also $\kappa(p_i)$, can be derived by traversing the tree from the root to the node $P(p_i)$. Therefore there is no need to repeatedly store the common prefix $\kappa(p_i)$ in $P(p_i)$; it suffices to store it once and to store the rest $\hat{s}$ of the separators on $P(p_i)$. This holds even for the keys on leaves, although storage of full keys on leaves or at least the repetition of the common prefix on the leaves is desirable for sequential processing of the file without the use of the index part. A full separator s on $P(p_i)$ is easily reconstructed by the concatenation $s = \kappa(p_i)\hat{s}$.

For example consider the tree in Figure 1. All the keys in the subtree with the root containing "To" have the common prefix "*T*". This common prefix can be derived according to the above definitions when the father node which contains the pair (T, u) is examined. Therefore in a prefix B-tree we would not store the common prefix T with the separators in the subtree.

Using this prefix compression technique yields a new kind of tree which we call *prefix B-tree*. We present another explicit definition of prefix B-trees in Section 5 after discussing the maintenance algorithms.

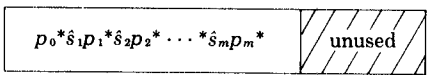
As mentioned before, we hope that prefix B-trees combine some of the advantages of B-trees, digital search trees, and key compression without sacrificing the basic simplicity of B-trees and the associated algorithms and without inheriting some of the disadvantages of digital search trees and key compression techniques. To substantiate these claims, let us first consider the algorithms for processing prefix B-trees.

5. ALGORITHMS FOR PREFIX B-TREES

5.1 Searching

To search for a key x on a page referenced by p:

(1) Determine $\kappa(p)$ according to its definition in Section 4.
(2) Remove $\kappa(p)$ from x, yielding $\hat{x}$.
(3) Let page $P(p)$ be organized as follows:



if $\hat{x} < \hat{s}_1$ **then** $q := p_0$;
if $\hat{s}_i \leq \hat{x} < \hat{s}_{i+1}$ **then** $q := p_i$;
if $\hat{s}_m \leq \hat{x}$ **then** $q := p_m$;

(4) **if** $P(p)$ is a leaf page, **then** retrieve the record q is referencing; it has key x if there is such a record at all
else begin $p := q$; **goto** step (1) **end**

Comments on the search algorithm.

Step (1): If x can be in $T(P(p))$ at all, it must have the prefix $\kappa(p)$. The same prefix $\kappa(p)$ was removed from the full separators s_j to obtain the partial separators $\hat{s}_j$ stored on $P(p)$. Therefore to search $P(p)$ use $\hat{x}$ as a search argument and compare $\hat{x}$ with the $\hat{s}_j$ on $P(p)$.

Step (3): Let $*$ be a special symbol which cannot appear in $\hat{s}_j$ or p_j. Even though the partial separators $\hat{s}_j$ have variable length, a calculated or binary search is still possible. We describe the binary search: Start the search in the middle of the used part of the page. With a short sequential character scan locate two neighboring $*$ symbols. Since the p_j have fixed length, we can determine $\hat{s}_j$, which is used for comparison. The iteration and termination of the scheme on one page is obvious.

Step (4): For simplicity we assumed that only pointers to the records are stored on the leaf pages.

When proceeding from P to a son page P', it is easy to construct $\lambda(P')$, $\mu(P')$, and $\kappa(P')$ and to iterate the search process. It is now clear that prefix B-trees avoid the main disadvantages of other compression techniques, namely the need to either decompress the keys on P first or to change the search argument to be used for comparison with each search step and to rely on additional structure information [9] to allow a faster than sequential, e.g. a quadratic, search of P. The way the prefixes $\kappa(P)$ are constructed is very reminiscent of the way of constructing prefixes in traversing digital search trees.

5.2 Insertion

A significant part of the process of inserting a record (x, a) with key x and associated information a is the search algorithm just described. In most cases, (x, a) will simply be inserted into a leaf and the insertion process is completed.

Node splitting. When a node P splits into P and P'', a separator s must be selected and a partial separator $\hat{s}$ must be inserted into the father page Q of the splitting page P. The prefix for the father page, i.e. $\kappa(Q)$, satisfies the property that $\kappa(Q)$ is a prefix of s. Thus $s = \kappa(Q)\hat{s}$, and the partial separator $\hat{s}$ can be inserted into Q without affecting any other separators on Q. The partial separators on P and P'' may now be shortened in case the new $\kappa(P)$ and $\kappa(P'')$ are longer than the old $\kappa(P)$ was. Similarly as for the original B-trees, splits may propagate toward the root and trigger further splits or overflows.

Overflow. Instead of splitting a full page P, an overflow from P into a brother page B can be attempted. Then the separator s between P and B must be replaced by another separator t, and accordingly $\hat{s}$ on the common father Q must be replaced by $\hat{t}$.

In this process the partial separators on P may shrink; those on B may expand. Therefore an overflow to B may not be possible even though B is not full. In this case P should be split. Replacing $\hat{s}$ by $\hat{t}$ may force further splits, merges, or overflows. This effect is analogous to the one already observed for simple prefix B-trees.

5.3 Deletion

Deletion of (x, a) will always be done on a leaf. If x is the first or the last key on a leaf, then the separator to the left or right brother, part of which is stored in the index part, might be shortened. But this is not necessary, since the longer separator will still remain a separator.

Node merging. Owing to the deletion of (x, a) from a leaf F, F might now be merged with a brother G. To merge F and G onto F, remove the corresponding partial separator from the father of F. This might trigger further merges or overflows. To merge branch nodes Q and B onto Q, delete the partial separator between Q and B on their common father node and recalculate the old and new partial separators on Q since they might expand. Obviously the condition for merging Q and B is that the expanded partial separators still fit onto Q.

With the discussion in the preceding sections, we have given a constructive definition of prefix B-trees: Prefix B-trees are exactly those trees that arise from applying arbitrary sequences of INSERT and DELETE operations to an empty file.

It may be helpful to present an explicit definition of prefix B-trees, defining precisely those properties which allow us to distinguish this kind of tree from other tree structures. This definition is rather difficult to grasp, however, without first understanding the discussion in the preceding sections and the constructive definition.

Definition. A *prefix B-tree* is a B^*-tree in which the B^*-index is replaced by the index part of a prefix B-tree.

Definition. The *index part of a prefix B-tree* is a directed tree structure together with a particular mode organization and additional properties as follows:

(a) Tree structure: (i) The tree is completely balanced with respect to path length. (ii) The degree d of a node is variable; $d \geq 2$, but otherwise d is determined by the internal organization of a node.

(b) Node organization: A node P contains an alternating sequence of references P_i to other nodes (the sons of P in the tree structure) and partial separators $\hat{s}_i$, which are variable length strings over some alphabet. The subsequence of partial separators is sorted. Denote the alternating sequence in a node as $p_0, \hat{s}_1, p_1, \hat{s}_2, p_2, \ldots, \hat{s}_m, p_m$.

(c) Additional properties: For every node p the following holds:

(i) Let the common prefix $\kappa(P)$ of P be as defined before. Then

$$s_i = \kappa(P)\hat{s}_i \quad \text{for} \quad i = 1, 2, \ldots, m$$

are called the separators of P.

(ii) Let $T(P(p_i))$ be the maximal subtree with root $P(p_i)$. Let $\kappa(p_i)$ be the set of all separators of nodes of $T(P(p_i))$ and of all keys on nodes of the B^*-file referenced from leaves of $T(P(p_i))$. Then the following hold:

$$\forall y \in \kappa(p_0) : y < s_1,$$

$$\forall y \in \kappa(p_i) : s_i \leq y < s_{i+1} \quad \text{for} \quad i = 1, 2, \ldots, m-1,$$

$$\forall y \in \kappa(p_m) : s_m \leq y.$$

(iii) If every node has a fixed storage capacity, the storage occupancy is so high that no two brothers can be merged.

We have claimed that prefix B-trees should combine some of the advantages of B-trees, digital search trees, and compression techniques without inheriting some of their less desirable properties. More precisely we mean the following:

(1) The basic advantages of B-trees are preserved:

—a balanced tree organization guaranteeing good worst-case performance,

—good storage utilization,

—maintenance algorithms which are only slightly more complicated than those for original B-trees with various length keys.

(2) The technique of choosing shortest separators and pruning off the common prefixes $\kappa(P)$ allows storing only partial separators in the index part of the tree. The two techniques could be applied independently to a B^*-tree. Choosing shortest separators is reminiscent of rear compression, and pruning off common prefixes is reminiscent of front compression of keys. However, the main disadvantages of other compression techniques are avoided.

(3) The technique of constructing prefixes while traversing the tree during a search is reminiscent of digital search trees. However, the danger of obtaining unbalanced trees is avoided.

Note. In certain cases, especially when insertions and deletions are fairly rare, it may be desirable to factor out the largest common prefix of the keys or separators actually stored in a node P instead of pruning off only $\kappa(P)$. Binary search of a page would still be possible. The advantage is additional storage saving; the disadvantage is additional processing required for some insertions or deletions which may alter the longest common prefix and therefore also the partial separators on a page.

Note. Recently some research has been done on performing concurrent operations on B-trees [6] and on enciphering B-trees [5]. All these results can be extended in a rather obvious way to apply to simple prefix B-trees and to prefix B-trees.

6. PERFORMANCE ANALYSIS OF SIMPLE PREFIX B-TREES

In Section 2 we have discussed why the performance of our trees depends heavily on the length of the separators stored in the B^*-index. In this section we therefore attempt an approximate analysis of the expected length of the separators in a simple prefix B-tree.

We restrict the analysis to a file with fixed length keys, but we see no reason why the results for variable length keys should be significantly different. We do take into account of course that we get variable length separators, but we do not consider the influence of the split intervals. For some comments on the general influence of the split interval, see Section 2 and the example in Section 3.

For our analysis we assume that the keys in the B^*-file are random. In practical applications this is often not the case, and it leads to a longer expected separator length if one considers a set S of separators containing exactly one shortest separator s_j for each consecutive key pair (x_{j-1}, x_j) in the sorted file. But only a rather small subset of S actually appears in the index part, and the technique of using the split interval for leaf splitting has the effect of selecting mostly the short sepa-

rators in S to be used as separators in the index part. We assume that the split interval roughly compensates for the nonrandomness of the keys of actual files and that the expected length of the separators in S is a useful first-order approximation for the separator length to be expected in practical applications in the index part of prefix B-trees.

Let $x_1, x_2, \ldots, x_{n-1}$ be the keys of a file in lexicographical order, and let ϵ be the empty word. Then the intervals $(\epsilon, x_1], (x_1, x_2], \ldots, (x_i, x_{i+1}], \ldots, (x_{n-1}, \infty]$ are called the *gaps* of the file. A file of cardinality $n - 1$ defines n gaps. We say that s is a separator for the gap $(x_j, x_{j+1}]$ or s is a separator between x_j and x_{j+1} or s fills the gap $(x_j, x_{j+1}]$ iff $x_j < s \leq x_{j+1}$.

Note. Each nonempty string is a separator for a unique gap defined by a file. For each gap there are one or several shortest separators. Choosing a shortest separator for each gap yields the set S mentioned before.

Assume that we have a set of n gaps and a set of m separators. If we assume that a separator fills a gap with probability $1/n$, then the probability that a particular gap is not filled by any of the m separators is $(1 - 1/n)^m$. Thus the expected number of gaps filled by m separators is

$$\omega(n, m) = n - n(1 - 1/n)^m.$$

Let the alphabet over which keys are formed have cardinality α. Then there are exactly α^l strings of length l. For a file of cardinality $n - 1$, let ω_i be the expected number of gaps of the file filled by the set of α^i separators of length exactly i. Then

$$\omega_i = \omega(n, \alpha^i) \quad \text{for} \quad i = 1, 2, \ldots.$$

Note. Let L be the fixed length of the keys of the file. Then each gap filled by a separator of length i will also be filled by separators of length $i + 1, i + 2, \ldots, L$. For example, the gap (Part, Prob] can be filled by "Pb", "Pba", "Pbaa" and also by "Pr", "Pro", "Prob" of lengths 2, 3, 4.

Let ν_i be the expected number of gaps actually filled by a separator of length i when the shortest possible separators are chosen. Then there will be $\nu_1 = \omega_1$ gaps filled by separators of length 1, $\nu_2 = \omega_2 - \nu_1$ gaps filled by separators of length 2, and in general $\nu_l = \omega_l - \sum_{i=1}^{l-1} \nu_i$, $l = 2, 3, \ldots$ gaps filled by separators of length l.

Since we have fixed length keys, each gap will eventually be filled by a separator of length at most L, and the expected length $E(s)$ of a shortest separator to fill a gap should be approximately

$$E(s) = 1/n \sum_{l=1}^{L} l\nu_l.$$

LEMMA 6.1. $\nu_l = \omega_l - \omega_{l-1}$ *for* $l = 2, 3, \ldots$.

PROOF. This follows directly from

$$\omega_l = \sum_{i=1}^{l} \nu_i.$$

Using $E(s) = (1/n)\sum_{l=0}^{L} l\nu_l$, $\nu_l = \omega_l - \omega_{l-1}$, and $\omega_l = \omega(n, \alpha^l) = n[1 - (1 - 1/n)^{\alpha^l}]$, we easily see that

$$E(s) = 1 + \sum_{l=1}^{L-1} (1 - 1/n)^{\alpha^l} - L(1 - 1/n)^{\alpha^L}.$$

Table I. Expected Length of Separators in the Index Part of a Simple Prefix B-Tree
α = cardinality of alphabet; n = cardinality of file.

	n					
α	10^3	10^4	10^5	10^6	10^7	10^8
2	9.634	12.955	16.277	19.599	22.921	26.243
8	3.546	4.645	5.747	6.855	7.968	9.083
10	3.263	4.262	5.262	6.262	7.262	8.262
16	2.775	3.638	4.476	5.283	6.080	6.903
26	2.483	3.104	3.842	4.615	5.258	5.929
36	2.238	2.884	3.614	4.140	4.843	5.529
256	1.774	1.976	2.517	2.936	3.180	3.845

For the following numerical examples the approximation

$$E(s) \approx 1 + \sum_{l=1}^{L-1} (1 - 1/n)^{\alpha}$$

was used. Furthermore it was assumed that $\alpha^L \gg n$. This means that the long separators do not significantly contribute to $E(s)$ and that $E(s)$ becomes nearly, i.e. within the accuracy calculated, independent of L. Table I presents $E(s)$ for a large range of alphabet sizes α and file sizes n.

7. HEIGHT OF A SIMPLE PREFIX B-TREE

According to [4], the minimal and maximal number of entries in a B-tree are

$$I_{\min} = 2(k + 1)^{h-1} - 1, \quad I_{\max} = (2k + 1)^h - 1.$$

We want to compare the number of keys or separators that can be stored in the index part—which is a B-tree—of a B^*-tree or a simple prefix B-tree for a fixed page size and a given height.

In [4] $I_{\min}$ and $I_{\max}$ were calculated for $k = 60$ and for experiments with an entry that required 14 bytes, e.g. 10 bytes for the key and 4 bytes for the page pointers, yielding a page size of 1684 bytes.

Using full bytes for the symbols of the alphabet ($\alpha = 256$), we see (Table I) that the expected length of a separator up to a file size 10^8 is less than 4. Thus let us assume pessimistically that a separator requires 4 bytes. Then a half-full page will have $k = 105$ entries, a full page 210 entries. We also calculate an average I_{av} assuming that each page is three-quarters full and contains 157 entries. (See Table II.)

8. PERFORMANCE OF PREFIX B-TREES

It seems quite difficult to give a precise analysis of the expected length of the common prefixes $\kappa(p_i)$ defined in Section 4 or equivalently of the amount of storage space that can be saved by using prefix B-trees rather than simple prefix B-trees. In [7] a rather crude worst-case analysis is carried out to obtain a lower bound on the expected length of the common prefix $\kappa(p_i)$. We omit the analysis here, but

Table II. Comparison of B^*-Trees and Simple Prefix B-Trees

Page size = 1684 bytes.

h	I_{min}	I_{av}	I_{max}	I_{min}	I_{av}	I_{max}
1	1	90	120	1	157	210
2	121	8280	14640	211	24963	44520
3	7441	753570	1771560	22471	3944311	9393930

B^*-tree		Simple prefix B-tree	
Key:	10 bytes	Separator:	4 bytes
Page pointer:	4 bytes	Page pointer:	4 bytes

Table III. Expected Length of the Common Prefix

	I				
k	10^3	10^4	10^5	10^6	
2	7.00	10.36	13.69	16.95	$\alpha = 2$
10	4.80	7.99	11.36	14.69	
100	1.90	4.80	7.99	11.36	
1000	0	1.90	4.80	7.99	
2	0.95	1.86	2.63	3.05	$\alpha = 26$
10	0.75	1.30	1.93	2.82	
100	0	0.75	1.30	1.93	
1000	0	0	0.75	1.30	
2	0.49	0.95	0.99	1.87	$\alpha = 256$
10	0	0.75	0.97	1.34	
100	0	0	0.75	0.97	
1000	0	0	0	0.75	

present the results for a representative collection of file sizes, typical page sizes, and alphabet sizes in tabular form. Although these results are probably poor lower bounds, they should be helpful design guidelines for the practitioner.

Table III gives lower bounds (arrived at in [7]) for the expected value of the length of the removable common prefix $\kappa(P)$. The parameters should be interpreted as follows:

α: size of the alphabet.

I: size of the separator set stored in the index part. (More precisely, I is only the size of the separator set stored in the leaf pages of the index part, but for typical applications this is within 1 percent of the total size of the index part.)

k: average number of partial separators stored on an index page. A crude estimate of k suffices for using the table.

9. PREFIX B-TREES AND DENSE INDEXES

Frequently a file for which an index must be built is not sorted according to the order of the keys being used for the index. In this case each key in the file must

also appear in the index. Such indexes are often called "dense indexes" or "secondary indexes" for obvious reasons.

It is interesting to observe now that for dense indexes it suffices to always store separators of two successive secondary keys in the index instead of full keys. Assume for example that the successive secondary keys on a leaf of the index would be $\cdots x_{i-1}, x_i, x_{i+1} \cdots$ with pointers $\cdots q_{i-1}, q_i, q_{i+1} \cdots$ to the actual records. It is assumed of course that the secondary keys are stored again with the records. Now assume that we construct separators s_j with the property

$$\cdots s_{i-1} \leq x_{i-1} < s_i \leq x_i < s_{i+1} \leq x_{i+1} \cdots$$

Then it suffices to construct the secondary index with the leaf pages containing only the separators rather than the full keys. The organization of a leaf page could then be

$$\cdots q_{i-2}, s_{i-1}, q_{i-1}, s_i, q_i, s_{i+1}, q_{i+1} \cdots .$$

Searching. To search for a record with key y between x_i and x_{i+1}, i.e. $x_i \leq y < x_{i+1}$, search the index of separators. This will yield $s_i \leq y < s_{i+1}$ and lead to the pointer q_i in the index. Then retrieve the record referenced by q_i and compare its secondary key with y. In case of equality we have found and retrieved the proper record; in case of inequality there is no record with the secondary key y in the file.

Updating. To perform an update, e.g. to insert a record with key y between x_i and x_{i+1}, construct a separator s_i' with the property

$$s_i \leq x_i < s_i' \leq y < s_{i+1} \leq x_{i+1} .$$

If s_i is not a prefix of y, then it is obviously possible to construct s_i' without retrieving x_i; otherwise x_i must first be retrieved to construct s_i'.

Deletion of a record works analogously.

The expected length of separators in a dense ($n \approx I$) index for a file of size n and the lower bounds—derived in Section 8—for the length of the removable common prefixes can now be used to get an estimate for the number of characters per key to be stored in the index. We imply subtract the lower bound for the length of the removable prefix in Table III from the expected length of the separator in Table I and obtain an upper bound for the expected length of the partial separators. Table IV contains those values for some representative parameters for prefix B-trees.

10. EXPERIMENTAL RESULTS

Simple prefix B-trees and prefix B-trees have been implemented to compare their performance against the performance of B^*-trees [1]. The main results concern computing time and saving of disk accesses.

Computing time. The time to perform the algorithms for simple prefix B-trees is nearly identical to the time for B^*-trees. Prefix B-trees need 50–100 percent more time. Considering the algorithms, this is unexpected, and we have, at the moment, no satisfactory explanation for this phenomenon.

Saving of disk accesses. For trees having no more than 200 pages no saving is achieved. For trees having between 400 and 800 pages, simple prefix B-trees re-

Table IV. Length of Partial Separators in a Prefix B-Tree

α = size of alphabet; k = average number of entries stored in a node; I = n = size of file and dense index.

	I				
k	10^3	10^4	10^5	10^6	
2	2.63	2.60	2.59	2.65	$\alpha = 2$
10	4.83	4.97	4.92	4.91	
100	7.73	8.16	8.29	8.24	
1000	9.63	11.06	11.48	11.61	
2	1.53	1.24	1.21	1.57	$\alpha = 26$
10	1.73	1.80	1.91	1.80	
100	2.48	2.35	2.54	2.69	
1000	2.48	3.10	3.09	3.32	
2	1.28	1.03	1.53	1.07	$\alpha = 256$
10	1.77	1.23	1.55	1.60	
100	1.77	1.98	1.77	1.97	
1000	1.77	1.98	2.52	2.19	

Table V. Experimental Results

B = B^*-tree; SPB = simple prefix B-tree; PB = prefix B-tree; n = cardinality of file; k = parameter as used in [4]; l = maximal length of keys.

n:	1000			5000			10000		
K:	30	20	10	30	20	10	30	20	10
Number of pages:	<80	<80	<80	≈125	≈200	≈380	≈230	≈380	≈760
Computing time, sec									
B, l = 9	20	20	20	150	150	150	330	320	350
SPB, l = 9	20	20	20	130	150	150	350	360	380
PB, l = 9	55	40	30	310	280	220	710	600	490
B, l = 15	25	20	15	180	170	170	450	370	400
SPB, l = 15	25	20	15	160	160	170	420	400	420
PB, l = 15	75	60	45	430	340	270	900	800	600
Disk accesses									
B, l = 9	800	800	900	4700	4800	6600	9600	10700	16400
SPB, l = 9	800	800	900	4700	4800	5400	9600	9900	13600
PB, l = 9	800	800	900	4700	4800	5200	9600	9700	13100
B, l = 15	800	800	900	4700	4800	6600	9600	10700	16400
SPB, l = 15	800	800	900	4700	4800	4900	9600		12100
PB, l = 15	800	800	900	4700	4800	4800	9600		12000
Length of separators									
SPB, l = 9		2,62			3,20			3,50	
SPB, l = 15		2,65			3,20			3,50	
Theoretical value		2,99			3,64			3,84	
Compression factor									
Measured	0,55	0,65	0,85	0,92	1,05	1,37	1,19	1,34	1,67
Theoretical value	0,37	0,48	0,73	0,92	1,00	1,34	1,19	1,34	1,61

quire 20–25 percent fewer disk accesses than B^*-trees. Compared to simple prefix B-trees, prefix B-trees need about 2 percent fewer disk accesses.

Length of separators. The average length of separators in simple prefix B-trees was in all cases about 0.35 less than the theoretical values. The compression factor in prefix B-trees corresponds with the theoretical results.

Numerical results. The numerical results listed in Table V were obtained in building up B^*-trees, simple prefix B-trees, and prefix B-trees by inserting n nodes into an initially empty tree. In all cases $\alpha = 13$ was chosen.

ACKNOWLEDGMENTS

We wish to thank Jim Gray and Mike Blasgen of IBM Research in San Jose for valuable discussions and comments about this paper. We also thank the referees for suggesting several important improvements to clarify the presentation of this paper.

REFERENCES

Note. References [2, 3] are not cited in the text.

1. AUER, R. Schlüsselkompressionen in B^*-bäumen. Diplomarbeit, Tech. U. München, München, Germany, 1976.
2. BAYER, R. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica 1* (1972), 290–306.
3. BAYER, R. Storage characteristics and methods for searching and addressing. Information Processing 74, North-Holland Pub. Co., Amsterdam, 1974, pp. 440–444.
4. BAYER, R., AND MCCREIGHT, E. Organization and maintenance of large ordered indexes. *Acta Informatica 1*, 3 (1972), 173–189.
5. BAYER, R., AND METZGER, J.K. On the encipherment of search tress and random access files. *ACM Trans. Database Syst. 1*, 1 (March 1976), 37–52.
6. BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-trees. IBM Res. Rep. RJ 1791, IBM Res. Lab., San Jose, Calif., May 1976.
7. BAYER, R., AND UNTERAUER, K. Prefix B-trees. IBM Res. Rep. RJ 1796, IBM Res. Lab., San Jose, Calif., June 1976.
8. KNUTH, D.E. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1972.
9. WAGNER, R.E. Indexing design considerations. *IBM Syst. J. 4* (1973), 351–367.
10. WEDEKIND, H. On the selection of access paths in a data base system. In *Data Base Management*, J.W. Klimbie and K.L. Koffeman, Eds., North-Holland Pub. Co., Amsterdam, 1974, pp. 385–397.

Received June 1976; revised November 1976