1977

Effects of Updates on Optimality In Tries and Doubly-Chained Trees

Douglas E. Comer
Purdue University, comer@cs.purdue.edu

Report Number:
77-256

Effects of Updates On Optimality

In Tries and Doubly-Chained Trees

CS TR-256

Douglas Comer

Computer Sciences Department

Purdue University

West Lafayette, Indiana 47907

November 1977

Effects of Updates On Optimality In Tries and Doubly-Chained Trees

Douglas Comer

Keywords and Phrases: Trie, Doubly-Chained Tree, Update, File Maintenance

CR Categories: 3.73, 3.74, 4.34

Abstract

The doubly-chained tree is a data structure for organizing files in a database system. We model such systems with a trie, a tree in which leaves correspond to records from a file. Retrieval proceeds by following a path in a trie from the root to a leaf, where the edge taken at each node is determined by some attribute value of the query. For a given file, altering the order in which attributes are tested can change the size of the resulting trie; tries with minimum size are considered optimum. We explore the preservation of optimality under the operations of inserting a record into the file, deleting a record from the file, and deleting any record from the file, showing that even for binary files a single update may be sufficient to make all optimum tries nonoptimum. The same results hold when the criterion of optimality is the average access time of a leaf.

Finding an indexing set for a file consists of finding a subset of the attributes which distinguish all records. We show that there are files for which a single insertion or deletion may make a minimum indexing set invalid so that no superset or subset, respectively, can be a minimum indexing set for the new file.

1 Introduction:

Intuitively, a file is a collection of records, each of which contains some information of an unspecified nature. Associated with a record is a key composed of values from one or more attributes. We assume that a record is uniquely identified by its key. A query poses a request for a record from the file by giving its key.

A trie[1] is a method of storing the keys from a file proposed by Fredkin [7]. A trie for a file F is a tree in which leaves correspond to records in F. Retrieval proceeds by following a path in the tree from the root to a leaf, where the edge taken at each node depends on some attribute value in the query. For example, Figure 1 shows a trie for the set of strings { "map", "mat", "mane", "many", "me" }. The path taken for the query "many" is darkened. Notice that all strings are padded to 4 characters by adding blanks where necessary.

The most straightforward implementation of a trie represents each nonleaf node by an array of pointers with one pointer for each possible attribute value. For the trie in Figure 1 there would be 27 elements in each array: one for each letter, and one for the blank character. An entry for the character "a" in the array representing a node u is a pointer to the son of u along an edge with label "a". With this tabular implementation, the desired son of a node can be found by one subscripting operation. Thus, searching proceeds rapidly. At the same time, most of the table entries are empty, so considerable space is wasted. In fact, the excessive storage cost detracts from this implementation.

---

[1] pronounced "try"

Several methods for reducing the space requirements of tries have been proposed. Fredkin observes that any chain leading to only a leaf may be pruned from the trie with the consequence that while lookup remains accurate, erroneous queries may not be detected until after the appropriate record is retrieved. Figure 2 shows the pruned trie for the set of strings in Figure 1. Taking a different approach, de la Briandais [5] proposes a binary tree representation for tries similar to the binary tree representation of a forest given in Knuth [8]. The idea is to place all sons of a node u on a linked list with u pointing to the first element of the list. Considerable savings in space results, especially if the degree of u is significantly smaller than the alphabet size. Sussenguth [12] coined the term "doubly-chained tree" for a modified linked list implementation which includes backpointers, and considers a hybrid scheme to conserve space in which the last few levels of the tree are replaced by a sequential search. Severence [11] gives a generalization of the hybrid approach, suggesting a TRIE-TREE to combine the tabular implementation and the doubly-chained tree. Knuth [9] reviews other space saving devices including a method for overlapping storage among sparse arrays in the tabular implementation.

Originally, tries were studied for the storage of character data as in our example, so the testing of attributes was left-to-right. But when a database system is organized as a doubly-chained tree (as in [2]), we think of a query as a k-tuple of unrelated values. It then makes sense to consider reordering the testing of attributes in the trie. deMaine and Rotwitt [6] observe that reordering attributes can decrease the size of the trie, and they give a heuristic for choosing a good order. Consider, for example, the trie shown in Figure 3 formed by

testing the third letter and then the fourth in the same set of strings given in Figure 1. The size, measured in internal nodes, decreases from 4 to 2. The same decrease occurs in the corresponding doubly-chained implementation.

We say that an ordering of attributes is optimum for a given file if the trie produced by that ordering requires least space. A trie produced by an optimum ordering will be referred to as an optimum trie.

The problem of producing an optimum ordering of attributes for a given file is known to be NP-Complete (see [4]). In this paper, we explore some of the effects of updates in the form of insertions or deletions to a file for which an optimum ordering is known. Clearly, if there were an efficient (polynomial) algorithm to maintain optimality of the ordering under insertion, we could use it to build an optimum order for a file in polynomial time by inserting records one at a time. In section 3 it is shown that the same holds true for deletions; an efficient algorithm to maintain optimality under deletion could be used to solve an NP-Complete problem efficiently. Thus, we expect that no such algorithms exist (on the assumption that $P \neq NP$). If, on the other hand, insertions and deletions are made to a file without attempting to maintain optimality, after some number of updates, the trie may be much larger than necessary. Cardenas [2] suggests reordering a tree structured database periodically to maintain a reasonable, if not optimum, size. The determination of when to reorder the database cannot be made a priori, and analysis is necessary to provide guidelines for doing so. This problem motivates our work.

Reordering attributes, in addition to changing the size of a trie, usually affects the average access time as well. Thus, the effects of

updates on retrieval speed will also be considered. For the tabular implementation, the time taken to access a leaf is proportional to its depth; the access time of a trie is given by the sum of the depths of all leaves. In [4] it is shown that the problem of finding an ordering of attributes which results in a minimum access time trie is also NP-Complete. Thus, maintaining a minimum access time ordering under operations like insertion or deletion may require exponential time, just as maintaining a smallest trie under the same updates does.

Since computing least space or least access time orderings may be costly, it is important that a fair balance be struck between the frequency with which the reorganization is performed and its overall worth. This paper begins the study of reorganization costs by exploring the number of updates necessary to cause a trie to become nonoptimum. We show that even for binary files, a single update is often sufficient to cause all optimum orderings to become nonoptimum. Although all our results are phrased in terms of tries, those concerned with space optimality apply equally to the doubly-chained implementation.

Another database problem is that of selecting a set of attributes over which to index. Schkolnick [10] gives a probabilistic algorithm, while Yao [13] considers statistical measures which include the clustering effect of attribute selections. As Yao points out, attribute selection in a tree structured database is inherently linked to attribute ordering. Unfortunately, the problem of finding an optimum set of attributes is NP-Complete even for a simple measure of optimality like minimum size [3]. In this paper we examine the effects of updates on minimum size indexing sets.

2 A Model for a Tree Structured Database:

In this section we define the terms file, key, query, and trie. We also pose questions about the effects of updates to be answered later.

A file will be thought of as a two-dimensional table with a row for each key and a column for each attribute. A query is a row in the table or it is not present in the file. Presumably, each row also contains a pointer to its record, which can easily be obtained once the row has been identified. But since the retrieval process itself is not germane to the problems at hand, we make no further distinction between records and their keys.

DEFINITION: Let $A_1, A_2, \ldots, A_k$ be a finite set of attributes, where attribute A_i takes on values from the finite set S_i, $1 \leq i \leq k$. A file F is a subset of $S_1 \times S_2 \times \ldots \times S_k$, and a key or record is an element of F. A query is an element of $S_1 \times S_2 \times \ldots \times S_k$. It may be that in a given file, not all elements of set S_i are used. The value set of attribute A_i in file F is $V_i = \bigcup_{r \in F} v_i$, where v_i is the actual value of the i^{th} attribute in r. Note that $V_i \subseteq S_i$, $1 \leq i \leq k$. The degree of a file F is given by $\max \{ \|V_1\|, \|V_2\|, \ldots, \|V_k\| \}$, where $\|V_i\|$ represents the number of elements in value set V_i. Files with degree 2 will be referred to as binary files. □

The basic notion is that if a file F has degree p then there is a file F' in which each entry is a nonnegative integer less than p, and F' is equivalent to F for the problems we consider.

Graph definitions used throughout this paper are standard, following

those in Aho et al [1].

DEFINITION: A full trie for a file F is a tree with all leaves at depth[2] k such that the following are true:

1. Let A_1, A_2, ..., A_k be the attributes of F and let π be a permutation of 1, 2, ..., k. All edges leaving a node at depth i - 1 have distinct labels chosen from $V_{\pi(i)}$ for all i, $1 \leq i \leq k$.
2. The labels encountered on each path from the root to a leaf correspond to an element of F, and, for each element of F there is such a path. □

Note that in order to specify a full trie all we need to do is give π which specifies the order in which attributes are tested. We can now give the definition of a trie from which leaf chains have been deleted, as in Figure 2.

DEFINITION: A node u in a tree is the head of a leaf chain if (a) the father of u has more than one son, and (b) u and all its descendants have at most one son.

A pruned trie for a file F is formed from a full trie for F by deleting the proper descendants of all nodes u such that u is the head of a leaf chain. □

We will be interested in the problem of maintaining least space pruned tries under the update operations of insertion and deletion. The following define these terms.

DEFINITION: The size of a pruned trie is the number of nonleaf nodes. A pruned trie is of optimum size for a file F if it is of minimum size

---

[2] the root of a tree lies at depth 0; the sons of a node at depth i - 1 lie at depth i.

over all pruned tries for F. An ordering of attributes π is optimum if it produces an optimum size trie. □

DEFINITION: Let F be a file and let r be a key in F. Then the result of deleting r from F is a file F' = F - {r}. F is the result of inserting r into F'. □

The following notions are used in describing minimum access time tries.

DEFINITION: Let T be a pruned trie for a file F. Then the access time of a leaf in T is given by its depth. The access time of T is the sum of the access times of all leaves. □

Note that the average access time of a leaf is obtained by dividing the access time by the number of leaves. Since the number of leaves is fixed, minimizing the access time also minimizes the average access time; the division is not necessary for our purposes.

The following provides terms that are useful in formulating the problem of indexing set selection.

DEFINITION: Let u be the head of a leaf chain in a full or pruned trie T, and let p be the record corresponding to the leaf which is an ancestor of u. Then p is said to be distinguished at u. Since all records in a file F are distinguished in a trie for F, we say that the trie indexes F.

A subset of the attributes of F over which a trie distinguishes all records in F is an indexing set. The size of an indexing set is the number of elements in it. An indexing set is minimum if it is of smallest size for F. □

In terms of the two-dimensional table for a file, an indexing set is a subset of the columns such that no two rows have identical values for all columns in the subset.

We can now state the questions considered in this paper.

QUESTION 1: Given a file F, and T, an optimum size pruned trie for F, can a single insertion (deletion) make T nonoptimum? □

QUESTION 2: Given a file F, can a single insertion (deletion) make all optimum size pruned tries nonoptimum? □

QUESTION 3: Given a file F, and T, an optimum size pruned trie for F, can the deletion of any record from F make T nonoptimum? □

Since we are interested only in pruned tries, the term "trie" will be used to mean "pruned trie" in the sequel.

Questions 1 - 3 will also be answered for updates to minimum access time tries, and updates to minimum indexing sets, using a restricted notion of indexing set optimality which eliminates trivial answers.

At this point the problem SAT3 (3-satisfiability) is introduced. SAT3 will be used to establish the difficulty of maintaining optimality under deletion.

Let n be a positive integer and $G_n = \{x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_n, \bar{x}_n\}$. The elements of G_n are called literals. Informally, a literal in G_n can either be true or false. In defining SAT3 we define clauses c_j, like x_1 or x_2 or $\bar{x}_3$. A clause is true if one of the literals in it is true.

We refer to the pair $(x_i, \bar{x}_i)$ as a variable. The complement of x_i is $\bar{x}_i$, and the complement of $\bar{x}_i$ is x_i. If a literal y is true, then

the complement of y is false and vice versa. Given a set of clauses $c_1, c_2, \ldots, c_m$, the clauses are <u>satisfiable</u> if, under some assignment to literals in G_n, all clauses are true. In the definition of SAT3, a set H will specify exactly which literals in G_n are true. In order for the truth assignment H to satisfy $c_1, \ldots, c_m$, for each c_j, one of the literals in H must also be in c_j, i.e. $H \cap c_j \neq \emptyset$.

<u>QUESTION</u> SAT3 (Satisfiability with three literals per clause): Given $I = \langle n, c_1, \ldots, c_m \rangle$, where n and m are positive integers, $n \leq 3m$, $c_j \subseteq G_n$, and $\|c_j\| = 3$, for $j = 1, 2, \ldots, m$. Does there exist a set $H = \{y_1, \ldots, y_n\}$ such that y_i equals x_i or $\bar{x}_i$ for $1 \leq i \leq n$, and $H \cap c_j \neq \emptyset$ for $j = 1, \ldots, m$? If there is such a set for a given I we say I is <u>satisfiable</u>. □

As is customary, we will use "+" to connect literals in a clause and "•" to connect clauses. Thus, an instance of SAT3 is $(x + \bar{y} + z) \bullet (\bar{x} + y + z) \bullet (\bar{x} + y + \bar{z})$.

A solution to SAT3 would be an algorithm that takes an instance I of SAT3 and answers true if and only if I is satisfiable. Informally, SAT3 is a difficult problem; no efficient solution is known. Aho <u>et al</u> [1] provides further details on SAT3 and the evidence that it is difficult.

In the next section we show that maintaining optimum size tries under update is at least as difficult as SAT3.

3 Maintenance of Optimum Tries Under Deletion:

In this section, the complexity of the problem of maintaining optimum tries under the operation of deletion is considered. It will be shown that if there is an efficient algorithm Z, which takes as input a file, an optimum order, and a deletion, and produces an optimum order for the updated file, then P = NP. In particular, it will be shown that the existance of such an algorithm leads to an efficient algorithm to solve SAT3, a known NP-Complete problem.

Figure 4a shows the construction of a file F from an instance of SAT3, B, that is crucial to our result. The following Lemma gives a property of such files.

LEMMA 1: (Comer and Sethi) Let B be an instance of SAT3, and let F be a file constructed from B as shown in Figure 4a. Then there exists a pruned trie for F with size no more than 2n + m iff B is satisfiable.

PROOF: Given in [4]. □

Now consider F', the result of inserting the records shown in Figure 4b into F. A trie T' for file F' can be generated by testing the following attributes left-to-right: all 2n attributes from set P, m attributes from set Q, one for each clause of B, and n attributes from set R, one for each pair of literals in B. It is easily verified that T' takes space $(r-1)/2$, and since any ternary tree of r leaves requires at least $(r-1)/2$ internal nodes (for odd r), T' is optimum. We will let π' denote the order which produces T'.

The following can now be stated.

THEOREM 1: Let Z be an algorithm which maps a file F', an optimum order T', and an integer d, denoting a record to be deleted, into an optimum order for F, the file formed by deleting record d from F'. If the time complexity of Z is $O(p(n))$, where $p(n)$ is a polynomial in n, then P = NP.

PROOF: Let B be an instance of SAT3. Peform the following steps to obtain an answer to B:

1. Construct a file F as shown in Figure 4a,
2. Insert the records as shown in Figure 4b to form a file F',
3. Generate π_0, the optimum order for F' given above,
4. For each record r in F' - F, apply algorithm Z to π_i to obtain π_{i+1}, an optimum order for F' - {r}, and delete r from F'.
5. Construct a trie for F given by the ordering π_{4n} (note that there are 4n records in F' - F) to determine its size, s.
6. From Lemma 1, B is satisfiable iff $s \leq 2n + m$.

Clearly, steps 1, 2, 3, and 6 require only polynomial time. Knuth [9] shows that step 5 requires only a linear number of steps in the size of F. Finally, Z has polynomial time complexity and is called only a polynomial number of times, so step 4 is also polynomial. Thus, a solution to SAT3 can be obtained in polynomial time, and since SAT3 is NP-Complete, the result follows. □

Similar results hold for deletion from a file for which a minimum access time order is known or for which a minimum indexing set is known. In each case, the construction is quite similar to the one presented here.

4 Updates and Least Space Pruned Tries:

In this section, questions 1 - 3 will be answered in the affirmitive. That is, there are files for which a single update is sufficient to destroy the optimality of all tries.

Consider the binary file shown in Figure 5. It is easy to verify that an optimum trie for this file has size 6. The orderings leading to such a size trie are: 1 2 3 4, 2 1 3 4, 1 4 3 2, and 4 1 3 2. Yet after the indicated record has been inserted, the optimum orderings become: 4 2 1 3, 4 2 3 1, 2 4 1 3, and 2 4 3 1. Thus, there is a file of 4 attributes such that 1 insertion makes all optimum orderings nonoptimum. By imbedding the 4-attribute file in a larger one as shown in Figure 6, the result can be extended to files of an arbitrary number of attributes. Note that the degree of the extended files is still 2. A binary file constructed as shown in Figure 6 is called an I(k) file; Lemma 2 gives a property of I(k) files needed to show that all optimum orderings become nonoptimum under 1 insertion.

LEMMA 2: Let F be an I(k) file constructed as shown in Figure 6. If π is an optimum ordering for F, then all attributes in set Q must be contiguous in π.

PROOF: Recall from Knuth [8] that any binary tree of r leaves must have at least r - 1 internal nodes. Since the order 1 2 3 ... k produces a trie with r - 1 internal nodes and r leaves, no smaller trie for F could exist. Now suppose that there is another optimum trie, T', produced by an order in which attributes from set Q are not contiguous. Since all nodes in T' must have 2 sons for it to be optimum, it is sufficient to show that at least one node in T' has only 1 son.

Once some attribute from Q has been selected in T', there will be at least two internal nodes in the trie until the last attribute from Q is tested. Each selection from Q distinguishes the records represented by one of the internal nodes and groups two more records together, forming a new internal node. But if a selection from P is made before all attributes from Q have been tested, an internal node representing records from the set K cannot be distinguished. Therefore, the selection from P would cause an internal node to have only 1 son. This is a contradication; T' cannot be optimum; and the lemma holds. □

LEMMA 3: Let F be an I(k) file constructed as shown in Figure 6, and let π be an optimum order for F. Then π is not optimum for F', the file formed by inserting the indicated record into F.

PROOF: From Lemma 2, all attributes from set Q must be contiguous in π. Moreover, the choices for Q must be one of the orders 1 2 3 4, 2 1 3 4, 1 4 3 2, or 4 1 3 2, or the trie could be made smaller by using one of them. But after the insertion is performed, the subtrie associated with the four selections from Q will no longer be optimum. Thus, the order given by π is not optimum for F'. □

The following corollary shows that there are binary files for which all optimum tries become nonoptimum under deletion.

LEMMA 4: Let F be an I(k) file constructed as shown in Figure 6, and let F' be the file formed by inserting the indicated record into F. If π' is an optimum ordering for F', then π' is not optimum for F.

PROOF: Observe that if all optimum orders for F become nonoptimum under the insertion producing F', then only nonoptimum orders for F became optimum under that insertion. Reversing the process, the

deletion of the same record from F' must make π' nonoptimum. From Lemma 3 we have that all optimum orders for F do become nonoptimum under the insertion, so the result holds. □

From the preceding, one can see that there are files for which no optimum trie is robust in the sense that a single insertion or deletion may destroy optimality. A related question of robustness will be addressed next, namely, whether there are files such that the deletion of any record can make some optimum trie nonoptimum.

Consider a binary file constructed as shown in Figure 7. Such files will be referred to as D(k) files, where k is the number of attributes. D(k) files have the property that there is an optimum order, π, such that π is not optimum for any file formed by deleting a record from a D(k) file. The next Lemmas establish our claim.

LEMMA 5: Let F be a D(k) file constructed as shown in Figure 7. Then π = 1 3 2 4 5 ... k is an optimum order for F.

PROOF: The profile of a trie is a sequence $< a_0, a_1, \ldots, a_{k-1} >$, where a_i represents the number of internal nodes at depth i in the trie. By definition, the size of a trie is given by the sum of all elements in its profile. A profile will be useful in describing an optimum trie for a D(k) file.

Claim: An optimum trie for a D(k) file has a profile < 1, 2, 3, 3, ..., 3, 3, 2 >.

To see this, think of Figure 7 as rolled into a cylinder with column k adjacent to column 1. Attribute i has exactly three 1's arranged so that the corresponding records interact with attribute i - 1, no attribute, and attribute i + 1, respectively. Thus, the file

is "symmetric" in that the first attribute chosen makes no difference. Moreover, any two choices produce a partial profile of at least < 1, 2 >. A third selection must add one more internal node, extending the partial profile to < 1, 2, 3 >. But two of the three internal nodes must correspond to sets of records that are not both overlapped by the same attribute. The profile can only be extended with 3's, since each attribute can distinguish at most one of the two sets corresponding to internal nodes, and always adds a new internal node itself. The only exception occurs when all attributes have been selected except for two. There must be an odd number of leaves at depth k - 1 because there are an odd number of records in F and the last depth in a pruned binary trie must have an even number of leaves. When the k - 1^{st} attribute is selected, one record is placed by itself and it becomes a leaf. Thus, the profile has the form < 1, 2, 3, 3, ..., 3, 3, 2 >.

Now suppose that at some depth an attribute was selected that did not distinguish records in an internal node. Then the profile would have a 4 at some point. Furthermore, no further choices could produce less than 3 internal nodes later. Thus, the trie could not be optimum, so the claim holds.

The order 1 3 2 4 5 ... k has a profile < 1, 2, 3, 3, ..., 3, 2 > and by the above is optimum for F. □

LEMMA 6: Let F be a D(k) file constructed as shown in Figure 7. Then there exists π, an optimum ordering for F, such that π is not optimum for any file F' formed by deleting a key from F.

PROOF: From Lemma 5, the order π = 1 3 2 4 5 ... k is optimum for F.

Now suppose that some record is deleted from F. Since the file is symmetric, we need only consider two cases: it is of the form of record 1

or it is of the form of record 2.

It is easy to verify that after record 1 is deleted there are only four optimum orderings:

1 2 3 4 ... k-3 k-2 k-1 k

2 1 3 4 ... k-3 k-2 k-1 k

1 k k-1 k-2 k-3 ... 4 3 2

k 1 k-1 k-2 k-3 ... 4 3 2

each of which produces a trie with no nodes having only one son. In general, the deletion of a record leaves some "starting" attribute so that proceeding around the cylinder in either direction produces an optimum order by insuring that no internal chains are generated. The only exceptions are the first two choices which can be made in either order.

For a deletion of a record like record 2, a similar situation occurs except that there is a "hole" allowing one to start on either side and proceed around the cylinder. Again, the first two choices can be made in either order. Thus, after record 2 is deleted, the optimum orders are:

2 3 4 ... k-3 k-2 k-1 k 1

3 2 4 ... k-3 k-2 k-1 k 1

1 k k-1 k-2 k-3 ... 4 3 2

k 1 k-1 k-2 k-3 ... 4 3 2

Therefore, π cannot be optimum for any file F' produced from F by 1 deletion. □

The following Theorem summarizes the results on optimality of pruned tries under updates.

THEOREM 2: There are files of degree d, $d \geq 2$, such that the following hold under the operations shown:

	1 Insertion	1 Deletion	Delete any Record
An Optimum Trie Becomes Nonoptimum	Yes, Lemma 3	Yes, Lemma 4	Yes, Lemma 6
All Optimum Tries Become Nonoptimum	Yes, Lemma 3	Yes, Lemma 4	?

PROOF: By the Lemmas indicated. □

5 Updates and Minimum Indexing Sets:

Recall that an indexing set is a subset of attributes which distinguish all records in a file. In terms of a trie for a file, the size of an indexing set I is the same as the maximum depth in any trie which uses the attributes in I.

It is trivial to give a file for which any indexing set becomes nonoptimum after 1 insertion if one simply chooses to add a record that is not distinguished by attributes in the trie. If the question is changed, however, to ask whether an indexing set can be extended (or contracted) to accommodate the insertion (deletion) it is not obvious that updates can have as drastic an effect.

An indexing set I is said to be valid for a file F' formed by inserting (deleting) a record into (from) a file F, if I is a minimum indexing set for F, and there is a minimum indexing set for F', I', such that $I \subseteq I'$ $(I' \subseteq I)$. We will show that there are a class of files for which 1 insertion or deletion can make a minimum indexing set invalid.

First, consider a single insertion. Figure 8 shows the construction

of a binary file in which 1 insertion invalidates an indexing set. Before any insertion, each attribute distinguishes exactly one record so any subset of k - 1 attributes is a minimum indexing set. In particular, $I = \{ 1, 2, 3, \ldots, k-1 \}$ is a minimum indexing set. After the record is inserted as shown, I is no longer an indexing set; attribute k must be added. Yet the set $I' = \{ k, 1, 2, \ldots, k-2 \}$ is an indexing set for F' which has only k elements. Therefore, I is not valid for F. We can therefore conclude the following.

LEMMA 7: Let k be a positive integer. There exists a file F, with k attributes, and a minimum indexing set for F, I, such that I is not valid for F', a file formed by inserting 1 record into F.

PROOF: Immediate from the above discussion. □

A file for which a single deletion invalidates a minimum indexing set is shown in Figure 9. Referring to the figure, one can see that any indexing set for such a file must include all attributes from set Q or records in set K could not be distinguished. Furthermore, at least three attributes from set P must be selected to distinguish the 5 records in set J. It follows that $I = \{ 1, 3, 4 \} \cup Q$ is a minimum indexing set for the file. After the indicated record has been deleted, however, I is not valid since no subset of I is an indexing set with as few elements as the indexing set $I' = \{ 1, 2 \} \cup Q$. These remarks are summarized in the next Lemma.

LEMMA 8: Let k be a positive integer. There exists a file F, with k attributes , and a minimum indexing set for F, I, such that I is not valid for F', a file formed by deleting a specified record from F.

PROOF: Immediate from the above discussion. □

Next we show that no file exists in which all minimum indexing sets become invalid after 1 insertion.

LEMMA 9: Let F be a file of k attributes and let R be the family of all minimum indexing sets for F. For any record r, some $I \in R$ is valid for F', the file formed by inserting r into F.

PROOF: The set of all tries for indexing sets in R must be the same height, say h. We claim that any insertion extends the height of one of these trees by at most 1, thus increasing the size of any indexing set in R by at most 1. Suppose that this were not true; that some trie was extended by 2 or more levels. Observe that a single insertion can extend the depth of at most one leaf in a trie. Therefore, the extended trie would have a node at depth h with only one son. But the attribute that was tested must have been superfluous and should not have been added to the indexing set. Continuing the process, attributes can be eliminated until there is an attribute tested at depth h which distinguishes the inserted record from the old one. So if I is a minimum indexing set for F, it can be extended to be a minimum indexing set for F' by adding only one element.

If all minimum indexing sets for F become invalid for F', then some indexing set, I', which was not minimum for F must become minimum for F'. But the size of I' is at least h + 1, or it would be minimum for F. Since an indexing set for F' can be obtained from a member of R by adding only 1 element, it will have size at most h + 1. So if I' is a minimum indexing set for F', I must be valid for F. The result follows.□

The next Lemma gives an immediate corollary for minimum indexing sets under the operation of deletion.

LEMMA 10: Let F be a file of k attributes and let R be the family of all minimum indexing sets for F. For any record r ∈ F, some I ∈ R is valid for F', the file formed by deleting r from F.

PROOF: Observe that if all minimum indexing sets for F under deletion of 1 record become invalid, then some indexing set which is not minimum must be valid for F'. Reversing the process, all minimum indexing sets for F' would become invalid under 1 insertion. From Lemma 9 we know that this is impossible. Thus, at least some I ∈ R is valid for F'. □

The following Theorem summarizes the results for minimum indexing sets under updates.

THEOREM 3: There are files of degree d, d ≥ 2, such that the following hold under the operations shown:

	1 Insertion	1 Deletion	Delete any Record
A Minimum Indexing Set Becomes Invalid	Yes, Lemma 7	Yes, Lemma 8	?
All Minimum Indexing Sets Become Invalid	No, Lemma 9	No, Lemma 10	No, Lemma 10

PROOF: By the Lemmas indicated. □

6 Updates and Minimum Access Time Pruned Tries:

We now turn to the problem of maintaining minimum access time tries under updates. Throughout this section we will use the term "optimum" to mean "minimum access time".

Recall the file shown in Figure 5. It is easy to verify that attri-

bute orderings which produce minimum access time pruned tries are exactly those orderings which produce least space pruned tries. Furthermore, after the indicated insertion is performed, all four optimum orders become nonoptimum. Thus, there is a 4-attribute file with the property that 1 insertion makes all minimum access time orderings nonoptimum.

Figure 10 shows an A(k) file, constructed by imbedding the 4-attribute file of Figure 5 in a larger one. The following Lemma establishes a property of A(k) files which will be used to show that 1 insertion can make all optimum (least access time) orderings nonoptimum.

LEMMA 10: Let $k \geq 6$ be an integer, and let F be an A(k) file constructed as shown in Figure 10. If π is an order which produces a minimum access time trie for F, then all attributes from set P must appear in π before any attribute in Q.

PROOF: A minimum access time trie for the file shown in Figure 5 has cost 21, and is achieved by the order 1 2 3 4. Delaying selections for the attributes in P by h depths adds 7h to this cost since attributes in P produce the same shape trie. Thus, selecting all attributes from P followed by those in set Q produces a trie with access time

$$h\,(2^h - 1) + (21 + 7h) \qquad (1)$$

where $h = ||P||$. If any attribute in Q preceded the selections from P, the cost would be at least

$$(h + 1)\,(2^h - 1) + (21 + 6h) \qquad (2)$$

Since for $h > 1$, (1) is less than (2), all attributes in P must appear before any attribute in Q. □

LEMMA 11: Let F be an A(k) file constructed as shown in Figure 10, and let π be an order which produces a minimum access time trie for F. Then π does not produce a minimum access time trie for F', the file formed by inserting the indicated record into F.

PROOF: Since the special case of k = 5 can be enumerated, we will present the case for $k \geq 6$.

From Lemma 10, all attributes from set P must be selected in π before any attributes from set Q. But then after 1 insertion, the cost of the trie can be reduced by reordering the selections in Q. Thus, π is not optimum for F' □

We can conclude the following Lemma for minimum access time orders under deletion from Lemma 11.

LEMMA 12: Let F be an A(k) file constructed as shown in Figure 10, and let F' be the file formed by inserting the indicated record into F. If π' is an ordering producing a minimum access time trie for F', then π' does not produce a minimum access time trie for F.

PROOF: From Lemma 11, all optimum orders for F become nonoptimum under the insertion which produces F'. It follows that π' could not be optimum after the deletion which yields F. □

Having answered questions 1 and 2 for minimum access time tries in the affirmitive, we turn to the problem of maintaining a minimum access time trie under the deletion of any record.

Recall that a D(k) file, constructed as shown in Figure 7, has the property that an order associated with a least space trie becomes nonoptimum under the deletion of any record. We will show that there is an order producing a minimum access time pruned trie which also

becomes nonoptimum after any deletion, in the sense that the resulting trie no longer has minimum access time.

From the proof of Lemma 5, any first choice of attributes leads to an optimum trie. At depth 2 there can be at most 1 leaf and at successive depths there can be at most 2 (except, of course, for the last two depths). Delaying an attribute A_i which produces a leaf does not shorten the path for other leaves, since A_i must be selected before the leaf can appear. Thus, selecting leaves as early as possible yields a minimum access time trie for a D(k) file. So the opimum order produces the following number of leaves at depth 0 to k: 0, 0, 1, 2, 2, ..., 3, 4. The order π' = 1 2 k 3 4 ... n-2 n-1 therefore produces a minimum access time trie.

Now consider F, a file formed from a D(k) file by the deletion of some record. From the proof of Lemma 6 there is an order π for F such that the number of leaves at depths 0 to k are: 0, 0, 2, 2, ..., 2, 3, 2. Let T denote the trie for order π, and compare it to T', the trie for order π'. Observe that T has 2 leaves at each depth except depths 2 and k - 1. Three cases arise:

Case 1: The first record of the file was deleted. Then T' has no leaves at depth 3,

Case 2: The second record of the file was deleted. Then T' has no leaves at depth 2.

Case 3: The i^{th} record was deleted, i > 2. Then T' has only one leaf at depth 2.

In all cases, at least one leaf in T' appears later than in T while no other leaves appear earlier. Therefore, T' has greater access time than T and cannot be optimum.

From the above discussion we can conclude:

LEMMA 13: Let F be a D(k) file constructed as shown in Figure 7. Then there exists π, an order producing a minimum access time trie for F, such that π does not produce a minimum access time trie for any file F' formed by deleting a record from F.

PROOF: Given in the discussion preceding the Lemma. □

We can summarize the results for minimum access time in tries in the following Theorem. The reader is reminded that these results apply only to the tabular implementation.

THEOREM 4: There are files of degree d, $d \geq 2$, such that the following hold under the operations shown:

	1 Insertion	1 Deletion	Delete any Record
A Minimum Access Time Trie becomes Nonoptimum	Yes, Lemma 11	Yes, Lemma 12	Yes, Lemma 13
All Minimum Access Time Tries Become Nonoptimum	Yes, Lemma 11	Yes, Lemma 12	?

PROOF: By the Lemmas indicated. □

7 Conclusions:

We have presented evidence that indicates the optimality in a trie, or related doubly-chained tree, is very sensitive to updates in the file. Even in the binary case, there are files for which a single insertion or deletion can cause all optimum tries to become nonoptimum.

For a database designer, this represents a warning that the restructuring to maintain optimality may need to be done quite frequently.

Minimum access time tries were also found to be sensitive to updates. While the access time results apply only to the tabular implementation, they are strong in that they hold even for binary files where the linked list implementation is not advantageous.

Although no results were presented on the relative cost of a nonoptimum vs. optimum trie for either access time or space measures of optimality, we can observe that the tries for D(k) files which were optimum before a deletion required almost 50% more space than necessary after only 1 deletion. Further study will almost certainly yield cases which have worse costs. More to the point, a look at the tries reveals that the internal chains account for the additional space, and it seems likely that such chains would be encountered in most files. Thus, in a database using the doubly-chained tree, one would expect to find a tradeoff between storage costs and the (possibly exponential) cost of restructuring often.

Before trie-based systems can be implemented that maintain a good balance between frequent updates and low storage costs, further analysis is needed.

Acknowledgement: The author would like to thank Christoph Hoffmann who showed that two insertions could make all optimum tries nonoptimum for a binary file, and encouraged further exploration.

References

1. Aho, A., Hopcroft, J., and Ullman, J., The Design and Analysis of Computer Algorithms, Addison Wesley, Reading, Mass., 1974.

2. Cardenas, A., and Sagamang, J., "Doubly-Chained Tree Data Base Organization -- Analysis and Design Strategies," The Computer Journal 20, 1 (Jan. 1977), pp. 15-26.

3. Comer, D., "Trie Structured Index Minimization," Ph.D. Dissertation, The Pennsylvania State University, 1976.

4. Comer, D., and Sethi, R., "The Complexity of Trie Index Construction," JACM 24, 3 (July 1977), pp. 428-440.

5. de la Briandais, R., "File Searching Using Variable Length Keys," Proc. Western Joint Computer Conference, IRE, New York, 1959, pp. 295-298.

6. deMaine, P.A.D., and Rotwitt, T. Jr., "Storage Optimization of Tree Structured Files Representing Descriptor Sets," Proc. ACM SIGFIDET Workshop on Data Description, Access and Control, November, 1971, pp. 207-217.

7. Fredkin, E., "Trie Memory," CACM 3, 9 (Sept. 1960), pp. 490-499.

8. Knuth, D.E., The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1968.

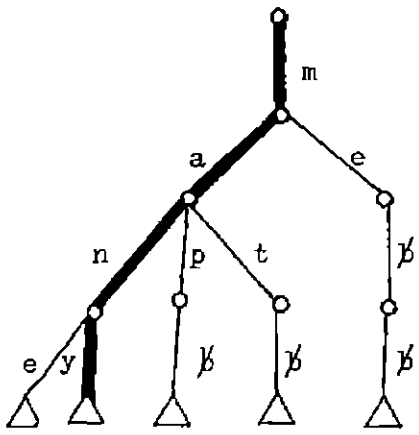
9. Knuth, D.E., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading Mass., 1973.

10. Schkolnick, M., "The Optimal Selection of Secondary Indices for Files," Information Systems 1 (1975), pp. 141-146.

11. Severence, D.G., "Identifier Search Mechanisms: A Survey and Generalized Model," Computing Surveys 6, 3 (Sept. 1974), pp. 175-194.

12. Sussenguth, E. Jr., "Use of Tree Structures for Processing Files," CACM 6, 5 (May 1963), pp. 272-279.

13. Yao, S.B., "Tree Structures Using Key Densities," ACM Annual Conf., Minneapolis, Minn., 1975, pp. 337-340.



(a)

m	a	p	ϕ
m	a	t	ϕ
m	a	n	e
m	a	n	y
m	e	ϕ	ϕ

(b)

Figure 1. (a) a full trie for the strings shown in the file (b).

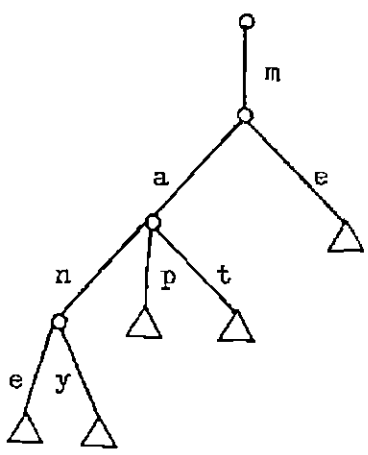


Figure 2. A pruned trie formed from the trie in Figure 1. Note the savings in space.

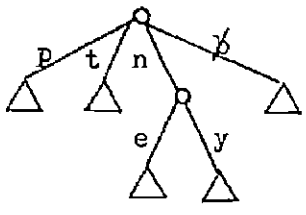
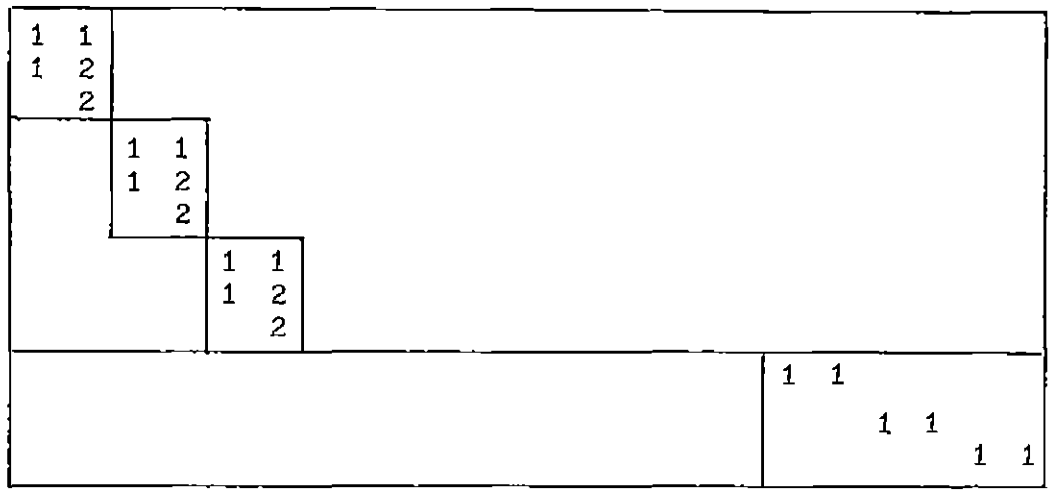


Figure 3. A pruned trie for the file shown in Figure 1 (b) formed by testing the third letter and then the fourth.

x	$\bar{x}$	y	$\bar{y}$	z	$\bar{z}$	$(x+y+z)(\bar{x}+\bar{y}+z)(x+\bar{y}+\bar{z})$	x	$\bar{x}$	y	$\bar{y}$	z	$\bar{z}$
1						1		1				
	1						1					
		1				1				1		
			1						1			
				1		1						1
					1	1					1	
						1 1 1						
						1 1 1						
						1 1 1						
2	2	2	2	2	2	2 2 2	2	2	2	2	2	2
		2	2	2	2	2 2 2	2	2	2	2	2	2
				2	2	2 2 2	2	2	2	2	2	2
						2 2 2	2	2	2	2	2	2
						2 2 2	2	2	2	2	2	2
						2 2 2	2	2	2	2	2	2
							2	2	2	2	2	2
									2	2	2	2
											2	2

(a)



(b)

Figure 4a. A sample file for the instance of SAT3 B = $(x + y + z) \bullet (\bar{x} + \bar{y} + z) \bullet (x + \bar{y} + \bar{z})$, and insertions (b) which make the problem of finding an optimum trie trivial.

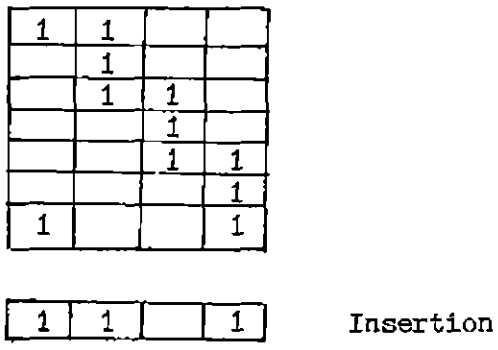


Figure 5. A binary file of 4 attributes such that 1 insertion makes all optimum tries nonoptimum.

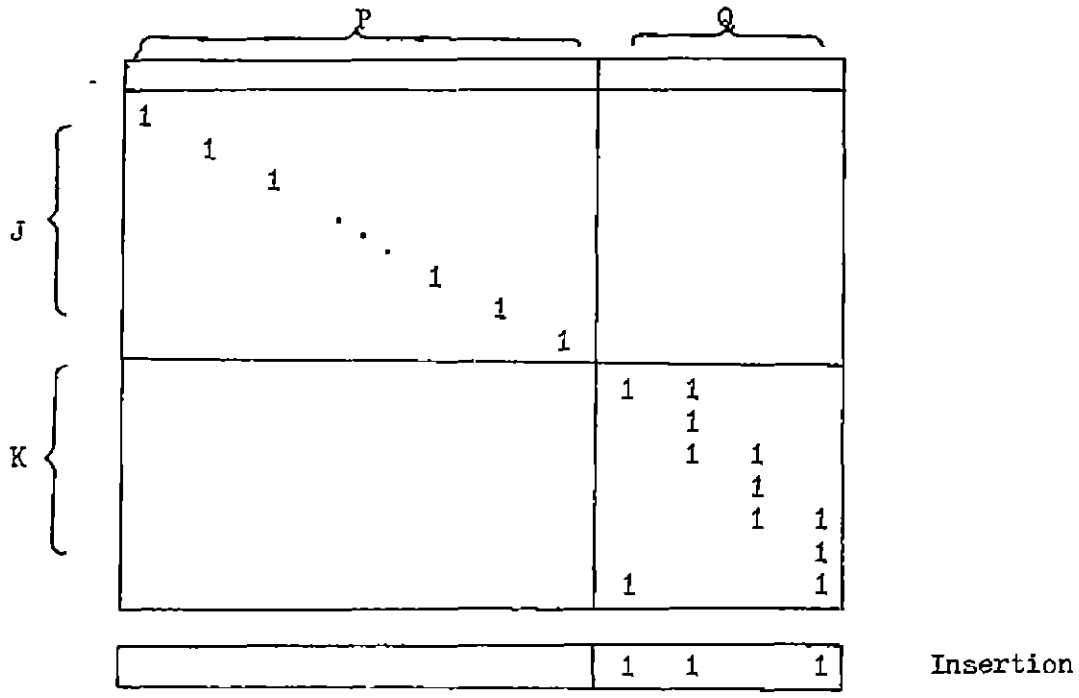


Figure 6. An I(k) file which has the property that 1 insertion makes all optimum tries nonoptimum. Note that the first record has all zero values. P can be extended to any number of attributes by adding records as indicated.

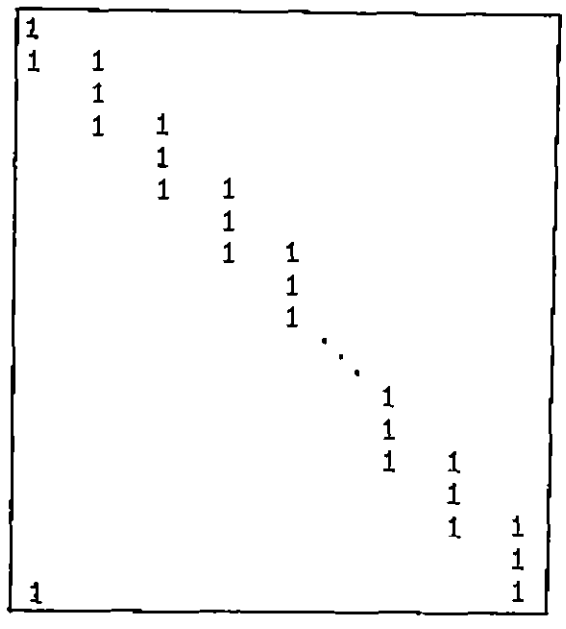
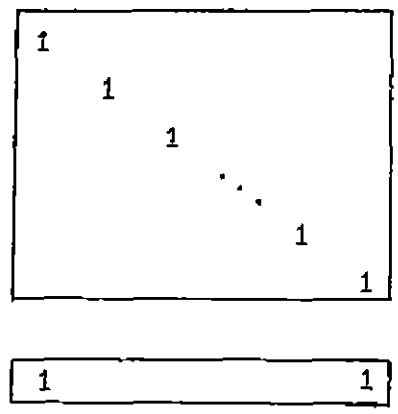


Figure 7. A D(k) file which has the property that the deletion of any record makes an optimum order nonoptimum.



Insertion

Figure 8. A file for which 1 insertion makes a minimum indexing set invalid.

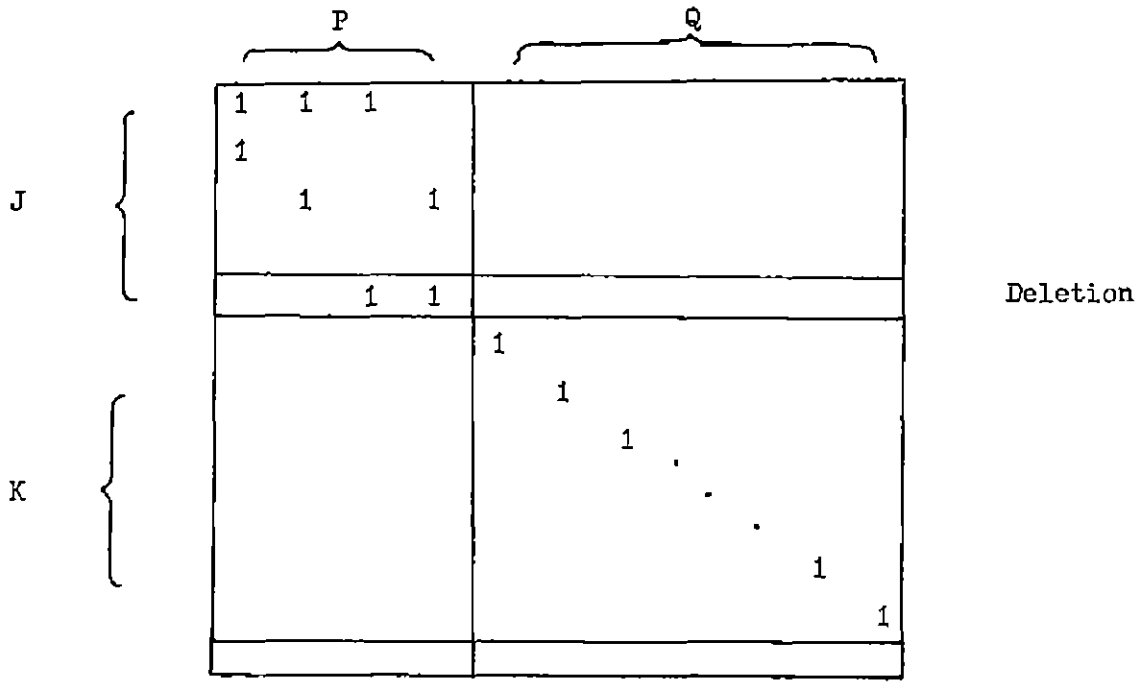


Figure 9. A file for which 1 deletion makes a minimum indexing set invalid. Note that the last record has all zero values.

P Q

J	1	1	1	1				
	1	1	1	0				
	1	1	0	1				
	1	1	0	0				
	1	0	1	1				
	1	0	1	0				
	1	0	0	1				
	1	0	0	0				
	0	1	1	1				
	0	1	1	0				
	0	1	0	1				
	0	1	0	0				
	0	0	1	1				
	0	0	1	0				
	0	0	0	1				
	0	0	0	0				
K					1	1		
						1		
						1	1	
							1	1
					1			1

Figure 10. A file for which 1 deletion makes all least access time tries nonoptimum. The set P generates a trie with many leaves and thus must be selected first.