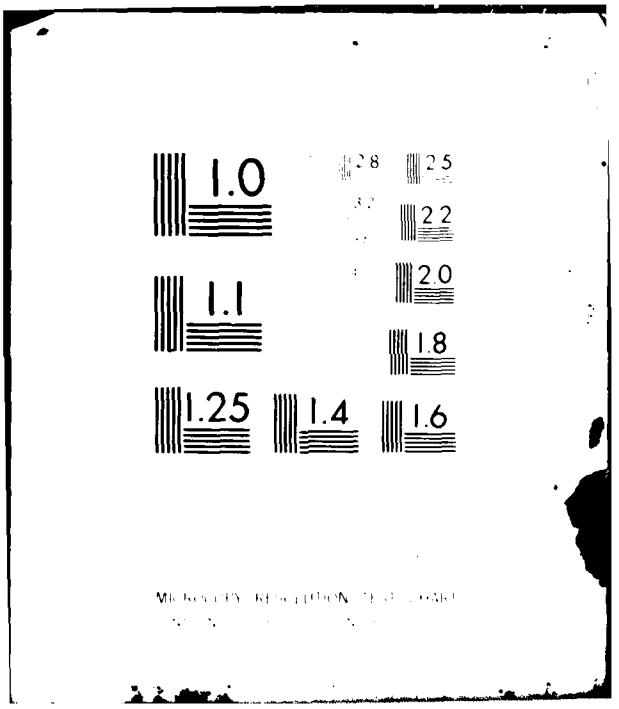
AD-A110 390 HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB F/G 5/2
NEW DATA STRUCTURES FOR ORTHOGONAL QUERIES.(U)
JAN 79 D E WILLARD N00014-76-C-0914
UNCLASSIFIED TR-22-78 NL

1 OF 1
AD A

END
DATE FILMED
02-82
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5

12

NEW DATA STRUCTURES FOR ORTHOGONAL QUERIES*

by

Dan E. Willard

TR-22-78

*This research was supported in part by the Office of Naval Research under contract N00014-76-C-0914.

Submitted January 12, 1979.

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A116390	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) "New Data Structures for Orthogonal Queries"	5. TYPE OF REPORT & PERIOD COVERED technical report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Willard, Dan E.	8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0914
9. PERFORMING ORGANIZATION NAME AND ADDRESS Harvard University Aiken Computation Laboratory Cambridge, Massachusetts 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research 800 North Quincy Street Arlington, Virginia 22217	12. REPORT DATE January 1979
	13. NUMBER OF PAGES 23 pages
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) unclassified
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

multidimensional searching
partial match retrieval
k-d tree
quad tree
bounded balance tree
B-tree
super-B-tree
$O((\log N)^{k-1})$

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The application of pyramid-like data structures to multidimensional queries has been explored in three recent papers (BS-77, Lu-78, Wi-78). ~~It will be shown here~~ that many of the earlier results (including some of our own) can be improved by a factor of log N with a slightly modified data structure that enables k-dimensional searches to be performed in $O(\log^{k-1} N)$ time. The new revised pyramid structure can be made sufficiently efficient in a dynamic environment to have an $O(\log^{k-1} N)$ record-insertion and deletion

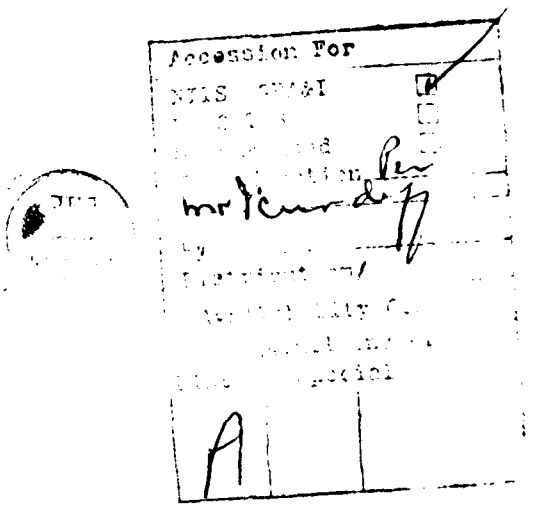
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

runtime. Queries for the special two-dimensional version of the proposed pyramid will have the same combination of $O(\log N)$ retrieval, insertion and deletion runtimes that has traditionally been associated with one-dimensional sorted lists. Our k-dimensional pyramid data structure will occupy $O(N \log^{k-1} N)$ space. The coefficient associated with its memory space utilization will only be approximately 50 percent larger than that of the otherwise considerably less efficient pyramids ~~of BS-77, Lu-78 and Wi-78.~~ Also, it ~~will be~~ shown ~~here~~ how the combination of the concepts of this paper ~~a~~long with ~~BE-75, Ri-76, Wi-78 and Wi-78a~~ other cited concepts can be used to develop very useful partial match data structures.

12



ONR/Code 411 IS, Ms. Laura Watson, has been notified that this report is Copyrighted.

INFORMATIVE ABSTRACT

NEW DATA STRUCTURES FOR ORTHOGONAL QUERIES

By Dan E. Willard
Harvard University

Computer Science classification: 3.73, 3.74
Keywords: multidimensional searching, partial match retrieval, k-d tree, quad tree, bounded balance tree, B-tree, super-B-tree

The application of pyramid-like data structures to multi-dimensional queries has been explored in three recent papers (BS-77, Lu-78, Wi-78). It will be shown here that many of the earlier results (including some of our own) can be improved by a factor of log N with a slightly modified data structure that enables k-dimensional searches to be performed in $O(\log^{k-1} N)$ time. The new revised pyramid structure can be made sufficiently efficient in a dynamic environment to have an $O(\log^{k-1} N)$ record-insertion and deletion runtime. Queries for the special two-dimensional version of the proposed pyramid will have the same combination of O(log N) retrieval, insertion and deletion runtimes that has traditionally been associated with one-dimensional sorted lists. Our k-dimensional pyramid data structure will occupy $O(N \log^{k-1} N)$ space. The

coefficient associated with its memory space utilization will only be approximately 50% larger than that of the otherwise considerably less efficient pyramids of BS-77, Lu-78 and Wi-78. Also, it will be shown here how the combination of the concepts of this paper along with Be-75, Ri-76, Wi-78 and Wi-78a can be used to develop very useful partial match data structures.

NEW DATA STRUCTURES FOR ORTHOGONAL QUERIES

By Dan E. Willard
Harvard University

There has long been an apparent need for an efficient data structure which supports retrievals on a conjunction of range predicates similar to

$a_1 < KEY.1 < b_1$ & $a_2 < KEY.2 < b_2$ & ... $a_k < KEY.k < b_k$

Following Knuth's suggestion (Kn-73), a series of articles has appeared within the last five years discussing this problem in the context of a data structure which occupies O(N) space (FB-74, BS-75, Be-75, LW-77, Wi-78a). More recently, several papers have begun to appear which discuss the improved retrieval time resulting from an allocation of $O(N \log^{k-1} N)$ memory space (BS-77, Wi-78, Lu-78). This article will show how a subtle change produces a dramatic improvement in the pyramid-like data structure of the latter series of articles.

In our discussion, L will denote the initial list of elements, L_v the subset of L that descends from tree-node v, and $P_o(k,L)$ the k-dimensional pyramid structure that was advocated in the previous articles. This pyramid will be inductively defined according to the value of k as follows:

1) If k=1, then the corresponding $P_0(k,L)$ pyramid will be defined as a tree representation of list L that has an O(log N) height and has sorted its records by increasing KEY.1 value.

2) Given that k-1 dimensional pyramids are previously defined, $P_0(k,L)$ will be inductively defined as a tree with O(log N) height that has the records of L sorted by increasing KEY.k value and which additionally associates each interior node v with an auxiliary $P_0(k-1, L_v)$ pyramid. This auxiliary pyramid was called an SDS field in Wi-78.

For a query q of the canonical form

$a_1 < KEY.1 < b_1$ & $a_2 < KEY.2 < b_2$ & ... $a_k < KEY.k < b_k$

the following terminology will be used:

i) SET(q) will denote the subset of the initial file that satisfies q

ii) COUNT(q) will denote the number of records belonging to SET(q)

iii) given a previously defined function F, SUM(q) will denote the sum of the F-values of those records belonging to SET(q)

The "locate-and-copy" time of a specified retrieval algorithm will be defined as the amount of runtime needed by the procedure

to find and transfer the members of SET(q) into the user's work-space. This concept is not very useful, because the degenerate case where COUNT(q) = N forces all procedures to have an O(N) worst-case locate-and-copy time. Consequently, another notion will be necessary in our worst-case analysis, and this paper will rely on the following two measurements:

i) the "locate" retrieval time of a search procedure will be defined as the difference obtained when subtracting COUNT(q) from the locate-and-copy time (worst-case analysis of locate runtime is meaningful because this quantity has been automatically adjusted to avoid the trivial degeneration that results when COUNT(q) is a large quantity)

ii) the aggregate-scan time of a retrieval algorithm is defined as the amount of time needed to scan the SET(q) collection of records for the purpose of calculating one of its aggregate values, such as SUM(q) or COUNT(q).

The application of the above two concepts to $P_0(k)$ pyramids was discussed in BS-77, Lu-78 and Wi-78. Some of the results obtained in these papers were quite similar, since they were written during overlapping time periods. What was known about pyramids previous to this article is given below:

1) SUM(q) and COUNT(q) can be calculated in $O(\log^k N)$ worst-case aggregate-scan time (BS-77, Lu-78, Wi-78).

2) SET(q) can be calculated in $O(\log^k N)$ worst-case locate time (observed in Wi-78 as a straightforward generalization of item 1).

3) If L is initially the empty set, and if a sequence of N insertion and deletion commands are subsequently given, then the total time needed to dynamically adjust $P_0(k,L)$ in response to this command sequence will have a worst-case $O(N \log^k N)$ magnitude. (First proposed in Wi-78. Several months later an independent derivation of a basically similar procedure was presented in a conference as Lu-78.)

4) The above result can be strengthened to indicate the existence of a procedure that executes individual insertion and deletion commands in $O(\log^k N)$ worst-case time (Wi-78; also in Wi-78b).

5) Several of the above results can have their runtime reduced by a factor of log N in a <u>batch environment</u> where N operations are simultaneously performed. Such batch procedures include:

   5a) an algorithm that constructs an entire $P_0(k,L)$ data structure in $N \log^{k-1} N$ time (BS-77, Lu-78)

   5b) a procedure that calculates ECDF statistics in $N \log^{k-1} N$ time (BS-77)

   5c) given n queries of $q_1\ q_2 \ldots q_n$, a procedure that calculates their SUM(q) and COUNT(q) values in $N \log^{k-1} N$ time (Wi-78)

The discussion in this paper will focus on topics 1 through 4 rather than the batch algorithms of topic 5. It will be shown here that the runtimes associated with topics 1-4 can <u>almost</u> be reduced by a factor of log N, thus deriving the new magnitude of $O(\log^{k-1}N)$. We say "almost" because the criterion used for measuring runtime here is slightly weaker than that in Wi-78 and the previous references. The distinction is that the earlier papers discussed worst-case optimization in a dynamic environment, whereas the improved results of this paper are either expected runtimes in a dynamic environment or worst-case runtimes in a static environment. Our new algorithm can be controlled to ensure that its worst-case performance will always be at least as efficient as that of Wi-78.

The symbols $P_e(k)$, $P_s(k)$ and $P_d(k)$ will denote the three modified versions of the $P_o(k)$ pyramid proposed in this article. All three will occupy the same $O(N \log^{k-1}N)$ quantity of memory space previously associated with $P_o(k)$, and each will solve a slightly different type of optimization problem. Below are listed the three main results that will be proven in this paper:

<u>Theorem 1</u>: The $P_s(k)$ pyramid (of definitions 2 and 5) will provide a static environment where SUM(q), COUNT(q) and SET(q) can be evaluated in $O(\log^{k-1}N)$ worst-case time.

<u>Theorem 2</u>: The $P_e(k)$ pyramid (of definitions 1 and 5) will provide a partially dynamic environment where SET(q) can be located in $O(\log^{k-1}N)$ <u>worst-case</u> time and where records can be inserted or deleted in $O(\log^{k-1}N)$ <u>expected</u> time.

Theorem 3: The $P_d(k)$ pyramid (of definitions 3 through 5) will provide a fully dynamic environment where record insertions, record deletions, and retrievals of SET(q) can be executed in $O(\log^{k-1} N)$ expected time and $O(\log^{k} N)$ worst-case runtime. (A comparison of theorems 2 and 3 indicates that $P_d(k)$ has better update and worse retrieval time than $P_e(k)$.)

In addition to discussing the above three classes of pyramids, this paper will also make brief mention of a new type of partial match and partial region query data structure which is quite similar to these pyramids. This new data structure (discussed in section 2) will enable the user to improve retrieval time by allocating $O(N \log N)$ additional units of space.

PART 1

The algorithms in this paper will make frequent subroutine-calls to the super-B-tree procedure introduced in Wi-78 (soon to be widely disseminated in Wi-78b). Because of its importance, the next several paragraphs will summarize the nature of this super-B-tree procedure.

In the forthcoming discussion as well as throughout this paper, it will be assumed that our trees have been structured so that there exists a one-to-one correspondence between the leaves of the tree and the record of the list it represents (as opposed to a pairing between general nodes and records). An SDS field will be defined as any auxiliary data structure which the user has created for the purpose of describing the descendants of a given interior node. A tree (which is a representation of a sorted list with O(log N) height) will be called an augmented tree if it assigns an SDS field to each of its interior nodes. For instance, the $P_o(k)$ pyramids (whose definitions were given in the second paragraph of this paper) are examples of an augmented tree.

The super-B-tree theorem describes the worst-case amount of runtime needed to insert and delete a record in augmented trees, in terms of a parameter w that denotes the amount of runtime needed to insert or delete a single record in an SDS field. The theorem states that arbitrary insertion and deletion operations can be performed within O(w log N) worst-case runtime.

This result is significant because the super-B-tree procedure simultaneously ensures that the augmented tree will have O(log N) worst-case height, and that no insertion or deletion command can cause the runtime involved in adjusting SDS fields to exceed the O(w log N) worst-case upper bound. (A traditional B-tree algorithm (AVL-62, NR-73, AHU-74) will not satisfy this condition because O(wN) worst-case time will be spent adjusting the SDS fields when "rebalancing" is performed.)

This paper's discussion of pyramids will begin with the $P_e(2,L)$ because it is the simplest of our various pyramids. The definition of $P_e(2,L)$ is given below:

Definition 1: A $P_e(2,L)$ pyramid will be defined as a two-part data structure consisting of a dictionary D and an augmented tree T. The former will be defined as a B-tree which has its records sorted by KEY.1 and which possesses pointers that map each record of the dictionary onto the location where the record is stored in the SDS field of the root of T. Here T will be defined as a tree which has its records sorted by KEY.2 and which uses the following rules to define the SDS fields of each of its nodes v:

A) SDS(v) will be a doubly-chained sorted list which has taken v's descendants (in T) and arranged them by order of increasing KEY.1 value.

B) In addition to containing its name, the entry for record R in SDS(v) will contain the following information:

i) pointers to the predecessor and successor of R in this SDS field

ii) a "LEFT.DOWN.POINTER" that contains the address of the least record in the SDS field of v's left son whose KEY.1 value is greater than or equal to that of R

iii) a "RIGHT.DOWN.POINTER" that similarly contains the address of the least record in the SDS field of v's right son whose KEY.1 value is greater than or equal to that of R

Our first lemma will discuss retrieval operations in $P_e(2,L)$ pyramids. In that discussion, as well as elsewhere in this paper, it will be necessary to speak of the nodes which are "critical" with respect to a range predicate such as $a<KEY<b$. An interior node v of a specified tree will be defined as critical whenever the following two conditions hold:

i) all leafrecords that descend from v satisfy this range condition;

ii) the same is not true for v's father (in other words, one of the father's descendants does not satisfy the range condition).

Lemma 1: Let q denote a two-dimensional query of the form $a_1<KEY.1<b_1$ & $a_2<KEY.2<b_2$. In the context of the $P_e(2,L)$ pyramid:

A) search operations for SET(q) can be executed in O(log N) worst-case locate time

B) insertions and deletions can be executed in O(log N) expected runtime

Proof: Only proposition A requires verification, since B is rendered trivial by the fact that it discusses only expected runtime. In our proof of A, the symbol INF(a_1,v) will denote the least record in SDS(v) whose KEY.1 $\geq a_1$. The search procedure that A needs to locate SET(q) will consist of the following three steps:

1) Find the address of INF(a_1, root of T) in log N time (by using dictionary D).
2) Let INF(a_1, critical) denote the union of the INF(a_1,v) elements of those nodes v in T that are critical with respect to $a_2 <$ KEY.2 $< b_2$. This step will construct the INF(a_1, critical) set in O(log N) time by using the binary tree that is rooted at INF(a_1, root of T) and generated by the LEFT.DOWN and RIGHT.DOWN pointers.
3) Construct the sought-after SET(q) in COUNT(q) runtime by making the obvious walk down the list of "successor" pointers that is generated by INF(a_1, critical).

The above algorithm obviously performs locate-and-copy operations for SET(q) in O(log N + COUNT(q)) time. Subtracting COUNT(q) from this quantity, we obtain the result that SET(q) has an O(log N) locate runtime. QED

The next objective of this paper will be to design and study a new pyramid that is capable of efficiently calculating SUM(q) and COUNT(q) values. This pyramid will be called $P_s(2,L)$ and is defined below:

Definition 2: The $P_s(2,L)$ pyramid will be defined as containing all the information of $P_e(2,L)$ plus two additional fields for each record R stored in SDS(v). These fields will be denoted as SUM(R) and COUNT(R). In the context of v's SDS field, these fields will specify the respective SUM and COUNT of the subset of SDS(v) whose KEY.1 value is greater than or equal to the KEY.1 value of R.

Lemma 2: In addition to satisfying part A of Lemma 1, the $P_s(2,L)$ pyramids will enable SUM(q) and COUNT(q) to be calculated in O(log N) worst-case aggregate-scan time.

Proof: Using reasoning similar to Lemma 1, it can be verified that all the members of the $INF(a_1, critical)$ and $INF(b_1, critical)$ sets can be found in log N time. The present lemma follows from this observation and the fact that

$$SUM(q) = \sum_{x \in INF(a_1,\ critical)} SUM(x) - \sum_{y \in INF(b_1,\ critical)} SUM(y)$$

$$COUNT(q) = \sum_{x \in INF(a_1,\ critical)} COUNT(x) - \sum_{y \in INF(b_1,\ critical)} COUNT(y)$$

QED

The next goal of this section will be to design a pyramid that optimizes on worst-case insertion and deletion time in addition to the expected time optimization mentioned in Lemma 1B. The proposed pyramid will be called $P_d(2,L)$. Our discussion commences with the following preliminary definition:

Definition 3: Let s denote an interior node of an augmented tree that is contained in a $P_e(2,L)$ pyramid, y a record in SDS(s), x the predecessor of y in this SDS field, and v the father of s. The symbol ASSOC(s,y) will denote the subset of SDS(v) whose records R satisfy the inequality $KEY.1(x) < KEY.1(R) \leq KEY.1(y)$.

Definition 4: A $P_d(2,L)$ pyramid will be defined as having a data structure identical to $P_e(2,L)$ in all respects but one. The distinction is that the $P_d(2,L)$ pyramid will not have any LEFT.DOWN.POINTER or RIGHT.DOWN.POINTER fields. Instead, each member of an SDS field of $P_d(2,L)$ will contain two new fields called LEFT.DOWN.LEAF and RIGHT.DOWN.LEAF, such that

i) each record y belonging to the SDS field of the left son of v will be associated with a 2-3 tree whose leaves are the LEFT.DOWN.LEAVES of the ASSOC [(left son of v), y] set and whose root points to y;

ii) each record y belonging to the SDS field of the right son of v will be associated with a similar 2-3 tree whose leaves are the RIGHT.DOWN.LEAVES of ASSOC [(right son of v), y].

The above 2-3 trees of the $P_d(2,L)$ pyramid will henceforth be called mapping trees. The runtime characteristics of this pyramid will be discussed in the next lemma. The proof of that lemma assumes that the reader is familiar with the characteristics of 2-3 trees that were discussed in AHU-74.

Lemma 3: Each of the following operations can be performed in $O(\log N)$ expected and $O(\log^2 N)$ worst-case time with the use of a $P_d(2,L)$ pyramid:

A) searches for SET(q);

B) insertions and deletions

Proof of A: The algorithm for performing searches in $P_d(2,L)$ pyramids is identical to that of $P_e(2,L)$, except that the former will traverse a path from a DOWN.LEAF to the root of the associated mapping tree on those occasions when the latter would simply advance to the position indicated by the corresponding LEFT or RIGHT.DOWN.POINTER. This difference cannot increase the runtime of the $P_d(2,L)$ procedure by a factor of more than log N (since 2-3 trees have log N worst-case heights). Furthermore, expected retrieval time should not increase at all, since the mapping trees in the present application will have an $O(1)$ expected height. Thus the previous Lemma 1 implies that $P_d(2,L)$ will have $O(\log N)$ expected and $O(\log^2 N)$ worst-case retrieval times.

QED

Proof of B: It is sufficient to confirm the proposition only for the deletion algorithm, since the insertion procedure is similar. Upon the user's command to delete a record R, the following three-step procedure will be executed.

1) Utilize dictionary D (of Definition 1) and the mapping trees to perform a straightforward search that finds all the entries for record R in the SDS fields of the $P_d(2,L)$ pyramid.
2) Repeatedly execute the following three substeps in order to remove R from each of the above SDS fields:
   a) Delete the LEFT.DOWN and RIGHT.DOWN leaves of R from their mapping trees;
   b) Merge the old mapping tree whose roots pointed to R into the mapping tree whose roots point to R's immediate predecessor (in the relevant SDS field);
   c) Deallocate R's memory space in the SDS field and make the predecessor and successor fields of its predecessor and successor point to each other.
3) Remove record R from $P_d(2,L)$'s augmented tree and use the super-B-tree algorithm to rebalance the augmented tree (so that it retains its C(log N) height).

The $C(\log^2 N)$ worst-case runtime of steps 1 and 2 can be understood given the observation that the time-consuming parts of these steps consisted of O(log N) invocations of certain specific 2-3 tree manipulation algorithms for which AHU-74

has verified an O(log N) worst-case runtime. The super-B-tree theorem indicates that the amount of time needed by step 3's rebalancing procedure must have the same magnitude as the SDS field updating which takes place in step 2. Thus the combined runtime of all three steps of our deletion algorithm has the $O(\log^2 N)$ worst-case magnitude which Lemma 3 attributed to it.

Similar reasoning can be used to confirm the O(log N) expected runtime of deletion operations. In essence, this runtime follows from the O(1) expected heights of the mapping trees. QED

The final goal of this section will be to generalize Lemmas 1 through 3 for k-dimensional pyramids. Below is our definition of k-dimensional pyramids:

Definition 5: Let P_i denote one of the symbols of P_e, P_s or P_d, and let L_v denote the subset of list L that is a descendant of v. The symbol $P_i(k,L)$ will denote a typical k-dimensional pyramid representation of L. If $k \geq 3$, then the associated $P_i(k,L)$ pyramid will be inductively defined as an augmented tree sorted by KEY.k whose SDS field equals $P_i(k-1, L_v)$.

Claim 1: The portions of Theorems 1 through 3 that discuss retrieval times of k-dimensional pyramids are valid. (The statement of these theorems can be found in the introductory portion of this paper.)

Proof: For $k \geq 3$, consider a retrieval procedure that locates the nodes which are critical with respect to $a_k < KEY.k < b_k$ and that recursively calls itself to search the SDS fields of these nodes. It is trivial to verify that such a procedure will cause $P_i(k)$ pyramids to have a retrieval time that exceeds $P_i(k-1)$ by a factor of log N. This fact, combined with Lemmas 1 through 3, easily inductively verifies the claim.

QED

Claim 2: The portions of Theorems 1 through 3 that discuss insertion and deletion runtime are also valid.

Proof: The super-B-tree theorem implies that the update time of a $P_i(k)$ pyramid will exceed that of $P_i(k-1)$ by a factor of log N. The claim follows from the conjunction of this fact, the principle of induction, and Lemmas 1 through 3.

QED

Although the discussion in this section was centered on measurements of CPU runtime, the proposed data structures are also useful in minimizing disc accesses. To illustrate this point, we consider the $P_e(2)$ pyramid.

In a paging environment, the SDS fields of $P_e(2)$ pyramids should be arranged so that consecutive records in their sorted lists appear on the same page. Let k denote the average number of records stored on a typical page, and r the fraction of the

file's total records that satisfy $c < KEY.2 < d$. A full locate-and-copy operation to retrieve the records satisfying $a < KEY.1 < b$ AND $c < KEY.2 < d$ from a $P_e(2)$ pyramid will require

$C_1 \log N + \frac{COUNT(s)}{k}$ worst-case page accesses (for some small constant C_1). In contrast, the same query would require

$C_2 \log N + \frac{COUNT(s)}{kr}$ expected page accesses with a (KEY.1-sorted) B-tree or some other conventional method of organizing a file. As r is always less than one and usually very small, the $P_e(2)$ pyramids produce a clear gain in efficiency.

Note Added February 15

At the time when the November draft of this paper was completed, I was aware of the possibility of slightly modifying the structure of the P_d pyramids by giving them mapping trees with a "multiway" rather than 2-3 structure. Multiway trees have been described in Kn-73, and they are the generalization of 2-3 trees that assign each interior mode between 2M and 2M-1 sons (for some fixed M). The employment of multiway mapping trees in the context of P_d pyramids would produce an improvement in the coefficient associated with retrieval time at the expense of the update runtime coefficient. Such a modification of the runtime coefficient was not mentioned in my entire draft because I did not consider it expecially subtle.

I now realize that multiway mapping trees are more important than I previously expected because they can be used to define new magnitudes of runtime. This can be done if M is treated as a variable rather than a constant. For instance, if M is defined as the least integer such that $M^M > n$ then the multiway mapping trees will produce a log log N improvement in retrieval time at the expense of an $\log N/(\log \log N)^2$ worsenning of update runtime. Hence, a $\log^k N/\log \log N$ retrieval and $\log^{k+1} N/(\log \log N)^2$ worst-case update can be associated with K dimensional pyramids. This change in worst-case runtime is produced without alterning the basic $\log^{k-1} N$ expected runtime that is associated with $P_d(k)$ pyramids. It appears that many users may desire to emplby this technique since a high priority is usually assigned to optimizing retrieval runtime. Further improvements in the magnitude of retrieval runtime do not appear possible without seriously damaging the worst-case update runtime associated with $P_d(k)$ pyramids.

PART 2

One serious disadvantage to the data structures of the preceding section is that they consume $O(N \log^{k-1} N)$ space. This allocation of memory space will generally be prohibitively expensive when k is greater than 3.

To save memory space, it is often useful to combine the theory of partial match data structures with the super-B-trees of Wi-78. A discussion of partial match data structures can be found in Be-75, Ri-76 and Wi-78a. These data structures are representations of k-dimensional files that occupy $O(N)$ space and associate $O(N^{1-j/k})$ retrieval time with requests of the form: $KEY.i_1 = C_1$ & $KEY.i_2 = C_2$ & ... $KEY.i_j = C_j$.

The distinction between the Be-75, Ri-76 and Wi-78 structures is rather technical. Ri-76 relies on hashing and consequently associates an $O(1)$ runtime with insertion and deletion operations. Be-75 utilizes tree representations for its partial match files, whose advantage is that they additionally enable searches to be done on a conjunction of range queries such as

$$a_1 < KEY.i_1 < b_1 \quad \& \quad a_2 < KEY.i_2 < b_2 \quad \& \quad \ldots \quad a_j < KEY.i_j < b_j$$

(discussed in detail in LW-77). Wi-78a describes a dynamic generalization of Be-75 designed to guarantee an $O(\log^2 N)$ worst-case insertion and deletion time.

Let A(KEY.0, k, α) denote an augmented tree which

i) has its records sorted by KEY.0

ii) has SDS fields that consist of partial match data structures describing the k Keys of KEY.1 KEY.2 ... KEY.k

iii) satisfies the Nievergelt-Reingold BB(α) condition (the nature of which can be explained if the ratio of v's left son's descendants over v's descendants is denoted as p(v): here BB(α) requires that all nodes of the tree satisfy $\alpha < p(v) < 1 - \alpha$)

Let M_{ab} denote the cardinality of the subset of our initial file that satisfies a<KEY.0<b. The theorems of Be-75, Ri-76 and Wi-78a can be easily generalized to show that A(KEY.0, k, α) associates an $O(M_{ab}^{1-j/k})$ worst-case retrieval time with queries of the form:

a<KEY.0<b & KEY.i_1 = C_1 & KEY.i_2 = C_2 & ... KEY.i_j = C_j

In contrast, the same query in traditional partial match files would require $O(N^{1-j/k})$ worst-case runtime (where N denotes the file's cardinality). Thus the A(KEY.0, k, α) data structure will have a significantly improved retrieval time, produced through an allocation of N log N additional units of memory space.

The point of this example is that the super-B-tree algorithm has many significant applications beyond the pyramids of the last section. In the present context, subroutine-calls to the super-B-tree algorithm will guarantee that any record can be inserted into and deleted from A(KEY.0, k, α) in O(log N) time if Rivest-like hash systems are used to define the SDS fields, and in $O(\log^3 N)$ worst-case runtime if the otherwise more flexible k-d trees of Be-75 and Wi-78a are employed. Several other useful applications of the super-B-tree procedure are discussed in Wi-78.

CONCLUSION

Our goal in this paper was to improve the runtime of multi-dimensional systems by employing data structures which require more than O(N) space. It was shown here that this could be done with data structures that occupy as little as O(N log N) or O(N $\log^2$N) space. This result could be quite significant if the cost of computer memory continues to drop at the same rate as it has in the past.

REFERENCES

AVL-62 G. M. Adel'son-Vel'skii and E. M. Landis, "An Algorithm for the Organization of Information," Sov. Math. Dokl., 3 (1962), pp. 1259-1262.

AHU-74 Alfred Aho, John Hopcroft, and Jeffrey Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley. Reading, Mass., 1974.

Be-75 Jon L. Bentley. "Multidimensional Binary Search Trees Used for Associative Searching, CACM. 18:9 (1975), pp. 509-517.

BF-78 Jon Bentley and Jerome Friedman, "Algorithms and Data Structures for Range Searching," Proceedings of the Computer Science and Statistics 11th Annual Symposium on the Interface (March 1978), pp. 297-307.

BS-75 Jon L. Bentley and Donald F. Stanat, "Analysis of Range Searches in Quad Trees," Inf. Proc. Letters, 3:6 (1975), pp. 170-173.

BS-76 Jon L. Bentley and Michael I. Shamos, "Divide-and-Conquer in Multidimensional Space," Proc. 8th Annual ACM Symposium on Theory of Computing, 1976. pp. 220-230.

BS-77 Jon Bentley and Michael Shamos, "A Problem in Multi-Variate Statistics: Algorithm, Data Structure, and Applications," Proceedings of the 15th Allerton Conference on Communications, Control, and Computing (Sept. 1977), pp. 193-201.

DL-76 David Dobkin and Richard Lipton, "Multidimensional Searching Problems," SIAM J. Comput., 5:2 (1976), pp. 181-186.

FB-74 R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," Acta Inf., 4 (1974), pp. 1-9.

K-73 Donald Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley. Reading. Mass., 1973.

Lu-78 George Lueker, "A Data Structure for Orthogonal Queries," 19th Symposium on Foundation of Computer Science (1978), pp. 28-34.

Lum-70 V. Y. Lum, "Multi-Attribute Retrieval with Combined Indexes," CACM (Nov. 1970), pp. 660-666.

LW-77 D. T. Lee and C. K. Wong, "Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees," Acta. Inf., 9(1977), pp. 23-29.

NR-73 J. Nievergelt and E. M. Reingold, "Binary Search Trees of Bounded Balance," SIAM J. Comput., 2:1 (1973), pp. 33-43.

Ri-76 Ronald Rivest, "Partial Match Retrieval Algorithms," SIAM J. Comput., 5:1 (1976), pp. 19-50.

Wi-78 Dan E. Willard. Predicate-Oriented Database Search Algorithm, Ph.D. Thesis for Harvard Mathematics Department; formally submitted May 3, 1978; accepted Sept. 28, 1978; published as an official Harvard Aiken Computer Lab report; to be disseminated as one of twenty volumes in Garland Publishing Company's series of 20 "Outstanding Dissertations in Computer Science."

Wi-78a Dan E. Willard, "Balanced Forests of K-d* Trees as a Dynamic Data Structure"; submitted to CACM; also will be a Harvard Aiken Computer Lab Research Report.

Wi-78b "Super-B-trees": two articles based on extracts from Wi-78 that will be completed and mailed to CACM and JACM before Dec. 31, 1978.

NOTE TO THE READER: Several references are made in this article to the super-B-tree algorithm. That algorithm was described in my dissertation (Wi-78), and a condensation (Wi-78b) should be available within a month's time. On completion, this condensation will be submitted to CACM for publication (also a second, much more technical description will go to JACM). I deliberately wrote this article and Wi-78a before producing a revised edition of my work on super-B-trees because I wanted to put my new ideas on paper before considering how to present the old ones more clearly.

:al Queries,"
:nce (1978),

)mbined

:s for Region
:al Binary Search
:(1977), pp. 23-29.

:rch Trees of
), pp. 33-43.

:ithms,"

Search Algorithm,
:nt; formally
78; published
port; to be
land Publishing
tions in

:es as a
also will
ort.

:cts from Wi-78
d J.CM before

this article
escribed in my
ould be available
tion will be
ch more technical
this article and
on super-B-trees
considering