MULTIKEY RETRIEVAL from K-d TREES and QUAD-TREES

D A Beckley
AT&T Bell Laboratories
Naperville, Ill 60566
(312) 979-2506

M W Evens
Computer Science Department
Illinois Inst of Technology
Chicago, Illinois 60616

V K Raman
Stuart School
Illinois Inst of Technology
Chicago, Illinois 60616

Abstract

Associative file structures are potentially valuable for many database and artificial intelligence applications, but very little information is available to database designers trying to choose an appropriate file structure for a particular problem This paper describes an experiment comparing the retrieval performance of K-d trees, quad-trees, and flat files, as measured by CPU time, wall clock time, and I/O operations Five types of queries are used exact match, partial match, range search, nearest neighbor, and best match The database used in this study is a static medical database of half a million characters with the patient information removed Results suggest that there is no one best type of file structure for all types of associative queries, quad trees dominated with some query classes, K-d trees with others

Motivation for Associative File Structures

Many data base applications are inherently multidimensional, but data has usually been forced into primary key file structures The main shortcoming of primary key structures for information processing applications is the absence of support for multikey retrieval functions Associative, or multikey, file structures can simplify the mapping of multikey data to physical storage as well as support a richer scope of access functions Data retrieval is based on content rather than address or file index and involves more substantive relations among the keys being matched It is not necessary to sequence a file with a unique key In addition to data base applications, associative file structures have found applications in several areas of artificial intelligence including image processing, robotics, cartography, pattern recognition, statistics, and information retrieval Knowledge-based systems and expert systems are often viewed as data base systems with added intelligence to augment the user interface, multikey retrieval capability is especially attractive for these applications

File design is not a simple problem [27], there are many relevant factors that can drastically affect performance [21] and cost. Data base designers faced with the chore of choosing an associative file structure have very few resources with which to attack this difficult task Few studies of the performance and cost characteristics of these new file structures are available This paper is an attempt to address this deficiency with a study of the performance of K-d trees and quad-trees on secondary storage with a direct access file organization The associative query algorithms for quad-trees that were used for these experiments are also given because they are not available elsewhere in the literature.

Review of Flat Files, K-d Trees, and Quad-Trees

Performance and cost characteristics are not always portable across operating environments This is partly due to implementation differences in file access methods Cardenas and Sagamang [10,11] pointed out that another very important performance factor is the manner in which lists or indexes are actually organized and managed In addition to file structure differences and environmental differences, the "best" or even an appropriate file structure [12] also depends on the data and users Guidelines and methodologies for the comparison of file structures in the targeted production environment are necessary

Sequential or "flat" files were chosen as a reasonable standard to use for comparison with other structures It is certainly the most basic of any file structure It is logically simple and it is easy to implement, however, for on-line applications, it is should be limited to small files because retrieval is slow Intuition would lead one to assume that a sequential file is not worthy of consideration for associative retrieval, but the simplicity of the structure makes it a viable solution for many applications

K-d trees [4] are multidimensional homogeneous binary trees At the root level, the first key is used as the primary discriminator for branching decisions At each successive level, the next key is used as the primary discriminator If more than k levels are necessary, the process repeats again beginning with the first key at the k+1 level This cycle continues for as many levels as necessary When the primary discriminator of a node is equal to the corresponding query key, secondary discriminators are used until a secondary key does not match the corresponding query key Secondary discriminators begin with the key after the primary discriminator and continue to the last

key, if still more discriminating keys are necessary, the first key through the key prior to the primary key are used The primary discriminator is related to the level in the tree and need not be stored as data As k gets larger, the levels of the tree necessary to cycle through all keys gets larger and a given key is thus used less often as a discriminator K-d trees work better when keys in queries are used as discriminators frequently [5]

The pseudo K-d tree [22] is nonhomogeneous modification of the K-d tree, all data resides in the terminal nodes This organization simplifies deletion and has been used to achieve some degree of balancing Pseudo K-d trees cannot handle multiple tuples with the same value in the same key Overmars and Van Leeuwen modified the pseudo K-d tree to get the extended pseudo K-d tree Additional pointers are added to each node for tuples lying on the axes or the origin

K-d trees are most often utilized as a primary storage structure, but the extended K-d tree [13,14,15] has been modified for secondary storage This structure uses the K-d structure as a directory to data stored in pages, but only the leaves of the directory contain page pointers Each internal node stores the discriminator field, although all nodes at the same level have the same discriminator Unlike the cyclic assignment of primary discriminators, the discriminator sequence can be assigned by a data base administrator Through the discriminator sequence function, the data base administrator can control the size of subfiles Like standard K-d trees, however, the discriminator sequence function assigns the same discriminator to all nodes on a given level

A second K-d tree hybrid, called the K-D-B tree [23,29,30], has evolved for very large indexes K-D-B trees combine features of B-trees [16] and K-d trees This structure has two types of nodes or pages which are called region pages and point pages Region pages define region boundaries and page ID's for each associated region The union of all region nodes under a given node is the region for the given node Point pages contain indexes to point data base records As in a B-tree, a point is inserted and splitting may ripple all the way to the root, thus the tree is always perfectly balanced and grows from the leaf towards the root The splitting, however, uses the discriminator concept when deciding which way to split a region A cyclic approach similar to the original K-d tree is a common approach

A third K-d tree hybrid is defined as an optimized K-d tree for finding best matches [19] This hybrid is a static structure Discriminators are not tagged by node position in the tree, in fact, nodes at the same level need not have the same discriminator The algorithm to build the optimized tree chooses the key with the largest spread as the discriminator, and the median of the discriminator key values as the partition

Quad trees [18] are an extension of the one-dimensional binary tree to a two-dimensional structure Each node in the tree has four sons to partition its scope of the universe The root for instance partitions the entire universe into NE (1), NW (2), SW (3), and SE (4) quadrants, with each son having one quadrant as its scope By convention quadrants (1) and (3) are closed and quadrants (2) and (4) are open, thus a point on the axis belongs to either (1) or (3) Insertion into a quad tree is similar to insertion in a binary tree Beginning at the root, a comparison is made to determine the subquadrant in which the new point resides If no point has been inserted in the subquadrant, the new point becomes the origin to partition the subquadrant for subsequent insertions Deletion is not so easy if the node is not a terminal node, in fact the common solution is to reinsert all points beneath a deleted node Samet [25] has studied the problem of deletion and proposed an algorithm that can reduce the number of nodes to be reinserted to 2/9 of the number to be reinserted with the original algorithm This algorithm proposes a definition for "closest" when determining the node to replace the deleted node Like binary trees, quad trees can also become readily skewed Finkel and Bentley have proposed simple balancing techniques for terminal nodes, but at times a tree built with the leaf balancing algorithm has a greater total path length than a tree built without balancing [18] Quad-trees are usually discussed for the two-dimensional case, but can be extended to arbitrary dimensions Another variation called pseudo quad-trees [22] has been proposed to make quad-trees more dynamic Instead of being a homogeneous structure the pseudo quad-tree is non-homogeneous Internal nodes are chosen as arbitrary points that suitably divide the subquadrant, and are not points in the file, thus all points are terminal nodes Deletion is merely the removal of a terminal node Pseudo quad trees have the restriction that no two points can have equal values in the same key Overmars and Van Leeuwen enhanced the pseudo quad tree with the extended pseudo quad tree (EPQ) structure This structure adds pointers for any points lying on an axis or equal to the origin, thus EPQ nodes have seven sons instead of four

Multikey Queries

The data base used for comparison in this experiment was the Michael Reese Hospital Stroke Registry, a static data base of a half million characters [1] The data consists of patient history information for stroke victims collected by the Neurology Department of Michael Reese Hospital There are seven retrieval keys used in this experiment stroke_type, race, age, sex, diabetes, systolic_blood_pressure, and diastolic_blood_pressure Primary key structures typically support only "GET" and "GET_NEXT" retrieval functions Associative file structures, on the other hand, support a richer set "EXACT" match, "PARTIAL" match, "RANGE" search, "NEAREST_NEIGHBOR", and "BEST" match queries are examples These functions give the illusion of retrieving data based on the content Sample queries were generated by sources independent of the experiment designers An interesting observation was that no record was retrieved for any of the exact match queries, thus leading one to conclude that exact match queries may be of little interest for this application A control set of exact match queries were generated to include successful responses for this query class in the comparison

EXACT MATCH In an exact match query, a single key value is specified for each key, and each key value of a retrieved record exactly matches the corresponding query key value To list any embolism case of a white, 70-year old female without diabetes, with systolic blood pressure of 140 and diastolic blood pressure of 80

```
SELECT ( STROKE_TYPE  = E  ,
         RACE         = W  ,
         AGE          = 070,
         SEX          = F  ,
         DIABETES     = N  ,
         SYSTOLIC_BP  = 140,
         DIASTOLIC_BP = 080 ),
```

PARTIAL MATCH In a partial match query, a single key value is specified for one or more or the keys, but not all of the keys For each of the keys that are specified, the corresponding key values of a record retrieved must match exactly To list any intracerebral hemorrhage case for a white male

```
SELECT ( STROKE_TYPE  = I  ,
         RACE         = W  ,
         SEX          = M  ),
```

RANGE SEARCH A range search query may involve a full or partial subset of the retrieval keys, but both a lower and upper bound value is given for each key specified To list any case of a 75 to 80 year old patient with systolic blood pressure between 141 and 169, and diastolic blood pressure between 90 and 121

```
SELECT ( 141 <= SYSTOLIC_BP  <= 169,
         090 <= DIASTOLIC_BP <= 121,
         075 <= AGE          <= 080 ),
```

NEAREST NEIGHBOR The nearest neighbor query uses a distance function to quantify the nearness of one record to another The function used in this application is the square root of the sum of normalized distance values The absolute value of the difference in AGE values is divided by 150 to obtain the normalized AGE value, the absolute value of the difference in SYSTOLIC blood pressures and the absolute value of the difference in DIASTOLIC blood pressures are both divided by 250 to obtain the respective normalized values The distance between two records is defined as the square root of the sum of the squares of these normalized values Only the numeric keys are used in nearest neighbor queries To list the nearest case(s) with age around 60, systolic blood pressure around 140, and diastolic blood pressure around 90)

```
SELECT ( NEAREST (AGE          = 060 ),
         NEAREST (SYSTOLIC_BP  = 140 ),
         NEAREST (DIASTOLIC_BP = 090 ) ),
```

BEST MATCH Best match queries do not have a distance function The best match query assumes all keys are of character type or coded values For each key in the query, an ordered list of "preference" values is specified from the most desired value to the least acceptable The dominance among keys is determined by the ordering of the preference lists in the query The search begins with the most preferred key value in the preference list of the most dominant key, preference lists of less dominant keys will then be used successively to identify the best record(s) To list the case(s) with the highest ranking for stroke type and diabetes

```
                                  1 1 1 1 1 1
Ranking        1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
Stroke_type    T T T T E E E E L L L L x x x
Diabetes       I O D x I O D x I O D x I O D )
```

```
SELECT ( PREF ( STROKE_TYPE  = T , E , L  ),
         PREF ( DIABETES     = I , O , D  ) ),
```

Tree Implementation

The implementation of K-d trees used in these experiments is described in detail in [2,3] Quad-trees have been used in many pictorial and image processing applications [17,20,24,26] Most applications have been only 2-dimensional and are concerned only with numeric coordinate data, but quad-trees can be extended to any dimension Overmars and Van Leeuwen [22] describe a generalized implementation For some number of keys, k, there are 2**k subspaces

Since this study uses seven keys, generalized quad-tree algorithms are necessary In the Michael Reese application with 7 keys, pointers for 2**7 or 128 subspaces emanate from each node The following table identifies the relationships of any record in a subspace to the current node for this application

subspace	k1	k2	k3	k4	k5	k6	k7
000	>=	>=	>=	>=	>=	>=	>=
001	>=	>=	>=	>=	>=	>=	<
002	>=	>=	>=	>=	>=	<	>=
003	>=	>=	>=	>=	>=	<	<
004	>=	>=	>=	>=	<	>=	>=
127	<	<	<	<	<	<	<

A record will reside in only one subspace of any node The following options exemplify three possible node implementations

Option 1 Fixed length node, bit-mapped pointers

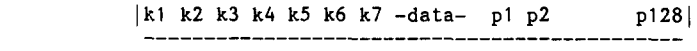
k1	k2	k3	k4	k5	k6	k7	-data-	-seq-	-bit map-

The structure of this option has the smallest node size but could potentially be very wasteful in overall space utilization The "seq" data is an internal field that uniquely identifies each record, this field is used to index a block of records reserved for the subspaces of the current node If a key or data field has this property, the sequence field is not necessary For the Michael Reese application, "case_number" is a non-key data field with this property The first index of the block of records reserved for subspaces of the "ith" case is

$$(2^{**}\text{max\_keys}) * (i-1) + 1$$

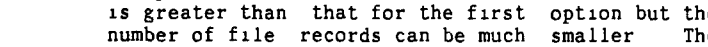
The first record is assumed to reside at index 0 of the file, any subspaces of the first record have indexes of 1 through 128 The waste occurs when few of the subspaces of a record have entries For each subspace, there is a corresponding flag in the bit map The bit map flags tag all subspaces of the current node that have at least one record Adding the index of the bit map flag to the start of the reserved block of records gives the file index of the next node of the corresponding subspace

Option 2 Fixed length node, index pointers



The node structure of the second option uses file index pointers instead of a sequence number and bit map flags Pointer space must be reserved in each node record for each possible subspace However file indexes do not have to be reserved for subspaces that have no records Thus the node size is greater than that for the first option but the number of file records can be much smaller The first record resides at file index one and a zero pointer indicates that there are no records in the corresponding subspace

Option 3 Variable length node, (subspace,index) pointers



The third option is a variable length structure Each subspace of the current node that has a record in it has a corresponding tuple The first entry of a tuple indicates a subspace, the second entry of the tuple is a file index of the next node of the subspace

Pointer space, number of records required, number of keys, and node structure simplicity are variables that must be weighed when chosing among the candidate node structure implementations For this experiment, Option 1 was chosen Record length was a controlled variable that was the same for all candidate structures With the first implementation option, quad-trees, K-d trees, and flat files could all be implemented with the given record length

Tree Algorithms

K-d tree associative query algorithms were published by Bentley, Friedman, and Finkel [4,6,7,8,9,19], but quad-tree algorithms were not available so they are discussed here Using the first node structure option, the following algorithms profile the retrieval requests for the quad-tree structure for the five multikey query classes described These algorithms, which were developed by the first author, assume implementation on a direct access file organization The following definitions are for variables and functions used in all the quad-tree query algorithms

Variables and Functions
---- for Quad-tree Algorithms ----

MAX_KEYS	Number of retrieval keys
QUERY_STR	Query node structure (key values)
REC_INDEX	File index of node
REC_STR	Record node structure (key values, data, sequence number, and bit mapped subspace flags)

EXACT MATCH The exact match retrieval descends only one path from the root of the quad-tree to a node without a candidate subtree At each intermediate node, the key values of the search query are compared to the node key values If all key values are equal, an exact match has been found In either case, the index for the root for the next candidate subtree is computed If there is at least one node in that subtree, the search continues with the subtree root, otherwise the search is complete

Variables and Functions
------ for Exact Match Only ------

BIT_INDEX	Index to a subspace bit flag

```
------ Quad-tree Exact Match ------
PROC EXACT_QD (REC_INDEX) RECURSIVE,
  READ KEY(REC_INDEX) INTO(REC_STR),
  BIT_INDEX = 1,
  IF REC_STR = QUERY_STR  THEN
    WRITE FROM(REC_STR),
  DO I = 1 TO MAX_KEYS,
    IF REC_STR KEY(I)  <  QUERY_STR KEY(I)  THEN
      BIT_INDEX = BIT_INDEX + 2**(MAX_KEYS - 1),
  END,
  IF REC_STR BIT_MAP(BIT_INDEX)  =  TRUE  THEN
    CALL EXACT_QD((REC_STR SEQ - 1)
                  (2**MAX_KEYS) + BIT_INDEX ),
END EXACT_QD,
```

PARTIAL MATCH The partial match retrieval algorithm traverses all paths from the root until a node with no candidate subtree is reached At each node, the key values of the search query are compared to the corresponding key values of the node If all of the query key values match the node key values, a partial match has been found The bit map is then checked for all candidate subtrees If there is at least one node in a candidate subtree, a recursive search will begin from the root of that subtree When all candidate subtrees have been scanned, the search is complete

Variables and Functions
----- for Partial Match Only -----

FIRST_BLOCK_INDEX	First index of flag in block of flags to be reset (do not search subspace)
PARTIAL_MATCH	Flag signifying the REC_STR node to be a candidate match for QUERY_STR query
PATTERN_STRING	Bit flags indicating subspaces to search
TABLE_BLOCK_SIZE	Number of flags to be reset

```
----- Quad-tree Partial Match -----
PROC PARTIAL_QD (REC_INDEX) RECURSIVE,
  READ KEY(REC_INDEX) INTO(REC_STR),
  PARTIAL_MATCH = TRUE,
  PATTERN_STRING = string(TRUE),
  DO I = 1 to MAX_KEYS,
    IF REC_STR KEY(I) used in query
      IF REC_STR KEY(I) ¬= QUERY_STR KEY(I) THEN
        PARTIAL_MATCH = FALSE,
      TABLE_BLOCK_SIZE = 2**(MAX_KEYS - I),
      IF QUERY_STR KEY(I) = REC_STR KEY(I) THEN
        FIRST_BLOCK_INDEX = 1,
      ELSE
        FIRST_BLOCK_INDEX = 2**(MAX_KEYS - I) + 1,
      DO J = 1 TO 2**MAX_KEYS
               BY 2*TABLE_BLOCK_SIZE,
        DO K = FIRST_BLOCK_INDEX
               TO  FIRST_BLOCK_INDEX +
                   TABLE_BLOCK_SIZE - 1,
          PATTERN_STRING(K) = FALSE,
        END,
        FIRST_BLOCK_INDEX = FIRST_BLOCK_INDEX +
                            2*TABLE_BLOCK_SIZE;
      END,
  END,
```

```
  IF PARTIAL_MATCH THEN
    WRITE FROM(REC_STR),
  DO I = 1 to 2**MAX_KEYS,
    IF PATTERN_STRING(I)   = TRUE   &
       REC_STR_BIT_MAP(I)  = TRUE  THEN
      CALL PARTIAL_QD((REC_STR SEQ - 1) *
                      (2**MAX_KEYS) + I ),
  END,
END PARTIAL_QD,
```

RANGE SEARCH The range search algorithm is very similar to the partial match All paths are searched from the root until a node with no candidate subtree is reached At each node, the upper and lower bounds of the keys of the search query are compared to the corresponding key values of the node If all of the node key values are within the upper and lower bound ranges, a range match has been found The bit map is then checked for all candidate subtrees If there is at least one node in a candidate subtree, a recursive search will begin from the root of that subtree When all candidate subtrees have been scanned, the search is complete

Variables and Functions
------ for Range Search Only ------

FIRST_BLOCK_INDEX	First index of flag in block of flags to be reset (do not search subspace)
RANGE_MATCH	Flag signifying the REC_STR node to be a candidate match for REC_STR query
PATTERN_STRING	Bit flags indicating subspaces to search
TABLE_BLOCK_SIZE	Number of flags to be reset

```
    ----- Quad-tree  Range Search -----
PROC RANGE_QD (REC_INDEX) RECURSIVE,
  READ KEY(REC_INDEX) INTO(REC_STR),
  RANGE_MATCH = TRUE,
  PATTERN_STRING = string(TRUE),
  DO I = 1 to MAX_KEYS,
    IF REC_STR KEY(I) in query
      IF QUERY_STR KEY(I,LO) > REC_STR KEY(I) |
         QUERY_STR KEY(I,HI) < REC_STR KEY(I) THEN
        RANGE_MATCH = FALSE,
      TABLE_BLOCK_SIZE = 2 **(MAX_KEYS - I),
      IF QUERY_STR KEY(I,HI) < REC_STR KEY(I) THEN
        FIRST_BLOCK_INDEX = 1,
      IF QUERY_STR KEY(I,LO) > REC_STR KEY(I) THEN
        FIRST_BLOCK_INDEX = 2**(MAX_KEYS - I) + 1,
      DO J = 1 TO 2**MAX_KEYS
              BY 2*TABLE_BLOCK_SIZE,
        DO K = FIRST_BLOCK_INDEX
              TO FIRST_BLOCK_INDEX +
                 TABLE_BLOCK_SIZE  - 1,
          PATTERN_STRING(K) = FALSE,
        END,
        FIRST_BLOCK_INDEX = FIRST_BLOCK_INDEX +
                            2*TABLE_BLOCK_SIZE,
      END,
  END,
  IF RANGE_MATCH THEN
    WRITE FROM(REC_STR),
  DO I = 1 to 2**MAX_KEYS,
    IF PATTERN_STRING(I) = TRUE THEN
      CALL RANGE_QD((REC_STR SEQ - 1) *
                    (2**MAX_KEYS) + I  ),
  END,
END RANGE_QD,
```

NEAREST NEIGHBOR The nearest neighbor algorithm begins at the root node and progresses down any subtree that might have a node with a smaller distance to the query keys than the nearest node found so far At each node, the distance from the node to the query is computed If the distance is smaller than the distance for the nearest so far, all saved nodes are released and the current node is saved as the nearest so far, the current node is added to the saved list Bounds arrays keep the upper and lower region bounds for each coordinate of any node that is in the current subtree For each subtree of a node, the bounds array are updated For any coordinate of the query that is outside of the bounds range, coordinate distance function of the query to the nearest bounds limit for that coordinate is added to a partial distance sum If the metric distance of the partial sum for all coordinates is greater than the distance to the nearest node so far, any node in the subtree will be further from the query than the nearest so far, thus the subtree will not be searched, a recursive search is performed on any candidate subtree

Variables and Functions
---- for Nearest Neighbor Only ----

B_LO	Lower bound for key
B_HI	Upper bound for key
cdist(i,j)	Coordinate distance function
COORD_DIST	Distance along one coordinate
DISTANCE	Distance between two tuples
mdist(z)	Metric distance function
NEAREST_SO_FAR	List of nodes that are NEAREST_SO_FAR_DIST from query
NEAREST_SO_FAR_DIST	Distance between NEAREST_SO_FAR and query
P_SUM	Partial distance sum
relation(i,j)	Function that returns '<' if relation of jth key of ith subspace pointer is '<' '>=' if relation of jth key of ith subspace pointer is '>='
sdist(i,j)	Sum of all coordinate distance functions

```
    --- Quad-tree  Nearest Neighbor ---
PROC NEAREST_QD (REC_INDEX) RECURSIVE,
  READ KEY(REC_INDEX) INTO(REC_STR),
  DISTANCE = mdist(sdist(REC_STR,QUERY_STR)),
  SELECT,
    WHEN (DISTANCE < NEAREST_SO_FAR_DIST),
      release all saved records
      NEAREST_SO_FAR(1) = REC_STR,
      NEAREST_SO_FAR_DIST = DISTANCE,
    WHEN (DISTANCE = NEAREST_SO_FAR_DIST),
      add REC_STR to list of NEAREST_SO_FAR
  END,
  DO I = 1 TO 2**MAX_KEYS,
    IF REC_STR BIT_MAP(I) = TRUE  THEN
      save B_LO's in TEMP_LO's
      save B_HI's in TEMP_HI's
      DO J = 1 TO MAX_KEYS,
        IF QUERY_STR KEY(J) numeric THEN
          IF relation(I,J) = '<'  THEN
            B_HI(J) = REC_STR KEY(J),
          ELSE
            B_LO(J) = REC_STR KEY(J),
      END,
      IF ¬OUT_OF_BOUNDS THEN
        CALL NEAREST_QD((REC_STR SEQ - 1) *
                        (2**MAX_KEYS) + I  ),
      restore B_LO's from TEMP_LO's
      restore B_HI's from TEMP_HI's
  END,
END NEAREST_QD,
```

```
PROC OUT_OF_BOUNDS RETURNS(FLAG),
  FLAG = FALSE,
  DISTANCE = mdist(sdist(NEAREST_SO_FAR,
                         QUERY_STR) ),
  DO I = 1 TO MAX_KEYS,
    IF REC_STR KEY(I) in query
      COORD_DIST   = 0,
      IF QUERY_STR KEY(I) < B_LO(I)  THEN
        COORD_DIST = cdist(B_LO(I),
                           QUERY_STR KEY(I) ),
      IF QUERY_STR KEY(I) > B_HI(I)  THEN
        COORD_DIST = cdist(B_HI(I),
                           QUERY_STR KEY(I) ),
      P_SUM = P_SUM + COORD_DIST,
  END,
  IF mdist(P_SUM) > DISTANCE THEN
    FLAG = TRUE,
  RETURN(FLAG),
```

BEST MATCH The best match algorithm begins at the root node and progresses down any subtree that might have a node with a better match to the query keys than the best node found so far At each node, the query keys are compared to the keys of the current node If the current node is a better match than the best so far, all saved nodes are released and the current node is saved as the best so far If the keys of the current node and the keys of the best so far match the query keys with the same preference levels, the current node is added to the saved list For each subtree, the preference lists are checked in order of key dominance to see if a better match could exist in the subtree If a better match is possible, a recursive search is performed on the subtree

Variables and Functions
------- for Best Match Only -------

BEST_SO_FAR	List of nodes that are best found yet
BETTER	Flag used in two ways (1) Current node better than BEST_SO_FAR (2) Subtree could have a better record
dominant_key_index(i)	Function that returns index of ith dominant key in query
KEY_MATCH	Key in record matches key in query
relation(i,j)	Function that returns '<' if relation of jth key of ith subspace pointer is '<' '>=' if relation of jth key of ith subspace pointer is '>='
WORSE	Flag used in two ways (1) Current node worse than BEST_SO_FAR (2) Subtree can not have a better record

```
    ------ Quad-tree  Best Match ------
PROC BEST_QD (REC_INDEX) RECURSIVE,
  READ KEY(REC_INDEX) INTO(REC_STR),
  DO I = 1 TO MAX_KEYS WHILE(¬BETTER & ¬WORSE),
    IX = dominant_key_index(I),
    KEY_MATCH = FALSE,
    DO J = 1 TO key_pref_count(IX)
             WHILE(¬BETTER & ¬WORSE & ¬KEY_MATCH),
      IF REC_STR KEY(IX)  = QUERY_STR KEY(IX,J)  &
         BEST_SO_FAR KEY(IX) ¬=QUERY_STR KEY(IX,J)
         THEN
        BETTER = TRUE,
      IF REC_STR KEY(IX) ¬= QUERY_STR KEY(IX,J)  &
         BEST_SO_FAR KEY(IX) = QUERY_STR KEY(IX,J)
         THEN
        WORSE  = TRUE,
      IF REC_STR KEY(IX)  = QUERY_STR KEY(IX,J)
         THEN
        KEY_MATCH = TRUE,
        CANDIDATE = TRUE,
    END,
  END,
  SELECT,
    WHEN (BETTER),
      release all saved records
      BEST_SO_FAR = REC_STR,
    WHEN (WORSE)
      ,
    WHEN (CANDIDATE),
      add REC_STR to list of BEST_SO_FAR
  END,
  DO I = 1 TO 2**MAX_KEYS,
    IF REC_STR BIT_MAP(I) = TRUE  THEN
      BETTER = FALSE,
      WORSE  = FALSE,
      DO J =1 TO MAX_KEYS WHILE(¬BETTER & ¬WORSE),
        JX = dominant_key_index(J),
        KEY_MATCH = FALSE;
        DO K = 1 TO pref_list_count(JX)
                 WHILE(¬BETTER & ¬KEY_MATCH),
          IF relation(I,JX) = '<'  THEN
            IF QUERY_STR KEY(JX,K)  <
               REC_STR KEY(JX)  THEN
              BETTER = TRUE,
          ELSE
            IF QUERY_STR KEY(JX,K) >=
               REC_STR KEY(JX) THEN
              BETTER = TRUE,
          IF QUERY_STR KEY(JX,K) =
             BEST_SO_FAR KEY(JX) THEN
            KEY_MATCH = TRUE,
        END,
        IF KEY_MATCH & ¬BETTER  THEN
          WORSE = TRUE,
      END,
      IF BETTER THEN
        CALL BEST_QD((REC_STR SEQ - 1) *
                     (2**MAX_KEYS) + I  ),
  END,
END BEST_QD,
```

Comparison to Flat Files

The same medical application used to exemplify the query classes was used to compare the performance of the quad-tree structure and the K-d structure with a flat file [2] Wall clock time, CPU time, and I/O counts were the observed factors of the experiment The first test was the "test of statistical significance" to determine if there is any significant difference [28] in performance between the two structures An analysis of variance on the session means was used to accept or reject the null hypotheses in Table 1 Each of the 18 combinations of observable factors with a query class gives a null hypothesis to be tested

	observable factors		query classes
The mean	CPU time wall clock time I/O counts	for	exact match * partial match range search nearest neighbor best match

queries are equal for the flat file, the K-d trees, and the quad-tree

* -Includes both the sample and control exact match queries

Null Hypotheses
Table 1

Since most analyses with K-d trees assume balanced K-d trees, there was interest in including both balanced K-d trees (BKD) and unbalanced K-d trees (UKD) in the experiment Table 2 summarizes the means observed for this experiment, Table 3 gives the standard deviations An "F" value (Table 4) of the analysis of variance is the ratio of the difference between the actual and predicted responses that is due to the file structure over the difference between the actual and predicted response that is due to random error The very large F variance ratios suggest that all null hypotheses can be rejected The PR values of Table 4 give the probability of observing a value of F larger than that observed if the hypotheses hold Clearly the choice of file structures are significant However, the analysis of variance gives no preference ranking for the file structures To examine more closely the relative performance of the file structures, the Student-Newman-Keuls (SNK) method is used to rank the file structures This test groups file structures into groups with significant differences between the means These rankings are also shown in Table 5 The group rankings "A", "B", and "C" are the ordered results of the SNK test When two or more file structures are in the same SNK group for a given query class, there is no significant difference between them The quad-tree and K-d trees consume less CPU time for exact match than the flat file, but for the other four query classes, the K-d trees used less than the flat file while the quad-tree used the most CPU resources For wall clock time, the quad-tree was faster than the K-d trees which in turn were faster than retrieval from the flat file, except for the exact match, the balanced K-d tree was faster than the unbalanced K-d tree Finally, the I/O overhead incurred by flat file was the greatest, and except for the nearest neighbor search, the quad-tree had less I/O than the K-d trees

	QUERY CLASS	FLAT ----FILE----	UKD ----TREE----	BKD ----TREE----	QUAD ----TREE----
CPU	EXAC*	235242 987	6004 426	4556 341	2145 302
	EXAC	234411 475	5155 953	4463 728	1626 787
	PART	248597 109	80479 466	65239 951	272773 781
	RANG	258995 029	172066 678	171227 311	1013088 252
	NEAR	279199 689	222990 773	251071 432	1070745 967
	BEST	239550 823	183214 101	189793 357	711583 912
Wall	EXAC*	9443984 710	167681 612	131131 921	44083 422
	EXAC	9063587 458	139475 778	132174 059	17326 885
	PART	9206685 641	1755740 003	2143803 596	691825 532
	RANG	9209356 403	3524287 155	5309605 192	2865013 498
	NEAR	8890100 953	3335738 412	5599895 268	3135874 467
	BEST	8998028 585	4018518 333	6245380 843	3693794 517
I/Os	EXAC*	567 000	12 320	9 280	3 960
	EXAC	567 000	10 760	9 220	3 040
	PART	567 000	160 320	125 640	51 400
	RANG	567 000	322 520	313 620	218 140
	NEAR	567 000	303 180	335 700	416 860
	BEST	567 000	370 800	371 980	301 960

* --> Control query set

Means
Table 2

	QUERY CLASS	FLAT ----FILE----	UKD ----TREE----	BKD ----TREE----	QUAD ----TREE----
CPU	EXAC*	5778 073	1236 799		636 518
	EXAC	6801 802	1140 000	636 518	469 579
	PART	11250 889	36904 977	469 579	206616 036
	RANG	17339 745	77855 711	206616 036	559486 317
	NEAR	10445 615	118405 812	559486 317	554907 282
	BEST	6794 660	42770 615	252443 834	252443 834
Wall	EXAC*	985228 531	41492 480	40433 798	40433 798
	EXAC	1976609 515	37799 410	14711 188	14711 188
	PART	1777080 575	750877 311	955150 701	955150 701
	RANG	1714526 587	1489451 182	1994255 918	1994255 918
	NEAR	2244266 953	1954117 855	2048963 635	2048963 635
	BEST	2076297 676	931497 733	2581662 327	2581662 327
I/Os	EXAC*	0 000	2 438	1 149	1 149
	EXAC	0 000	2 338	0 938	0 938
	PART	0 000	68 540	44 029	44 029
	RANG	0 000	128 066	123 009	123 009
	NEAR	0 000	177 483	217 381	217 381
	BEST	0 000	83 605	107 510	107 510

--> Control query set

Standard Deviations
Table 3

	Exact*	Exact	Partial	Range	Nearest	Best
CPU						
F	99,999 99	99,999 99	1,574 59	3,089 07	3,055 36	5,784 79
PR	0 0000	0 0000	0 0001	0 0001	0 0001	0 0001
	A QD	A QD	A BKD	A BKD	A UKD	A UKD
	B BKD	B BKD	B UKD	A UKD	B BKD	A BKD
	C UKD	C UKD	C FL	B FL	C FL	B FL
	D FL	D FL	D QD	C QD	D QD	C QD
Wall						
F	99,999 99	30,857 61	15,624 08	4,040 94	2,309 42	2,586 52
PR	0 0000	0 0001	0 0001	0 0001	0 0001	0 0001
	A QD	A QD	A QD	A QD	A QD	A QD
	B BKD	B BKD	B UKD	B UKD	B UKD	B UKD
	C UKD	B UKD	C BKD	C BKD	C BKD	C BKD
	D FL	C FL	D FL	D FL	D FL	D FL
I/Os						
F	99,999 99	99,999 99	31,507 31	3,252 74	841 90	3,179 51
PR	0 0000	0 0000	0 0001	0 0001	0 0001	0 0001
	A QD	A QD	A QD	A QD	A UKD	A QD
	B BKD	B BKD	B BKD	B BKD	B BKD	B UKD
	C UKD	C UKD	C UKD	C UKD	C QD	B BKD
	D FL	D FL	D FL	D FL	D FL	C FL

Student-Newman-Keuls Rankings
Table 4

Summary

There are a number of performance factors that data base designers must consider, including CPU time, response time, and I/O counts But the performance of no file structure is dominant for all associative query classes The SNK rankings are not sufficient to suggest an "appropriate" file structure

The means of each SNK group were computed and are shown in Table 5

Using the group means of Table 5, normalized SNK group coefficients were computed For the best group, the normalized coefficient is 1 0 while for the last group, the normalized coefficient is 0 0 For the intermediate groups, difference between the intermediate group mean and the best group mean is divided by the difference between the last group mean and the best group mean, this factor is then subtracted from 1 0 A significant difference was observed between each group, but the absolute differences between consecutive group means are not equal The normalized coefficients rank groups relative the best and worst SNK groups These normalized coefficients are given in Table 6

CPU

SNK Group	Exact*	Exact	Partial	Range	Nearest	Best
A	2145 30	1626 79	65239 95	171646 99	222990.77	186503 73
B	4556 34	4463 73	80479 47	258995 03	251071.43	239550 82
C	6004 43	5155 95	248597 11	1013088 25	279199.69	711583 91
D	235242 99	234411 48	272773 78		1070745 97	

Wall Clock Time

SNK Group	Exact*	Exact	Partial	Range	Nearest	Best
A	44083 42	17326 89	691825 53	2865013 50	3135874 47	3693794.52
B	131131.92	135824 92	1755740 00	3524287 15	3335738 41	4018518 33
C	167681 61	9063587 46	2143803 60	5309605 19	5599895.27	6245380 84
D	9443984 71		9206685 64	9209356 40	8890100 95	8998028 59

I/O Counts

SNK Group	Exact*	Exact	Partial	Range	Nearest	Best
A	3 96	3 04	51 40	218 14	303 18	301 96
B	9 28	9 22	125 64	313 62	335 70	371 39
C	12 32	10 76	160 32	322 52	416 86	567.00
D	567 00	567 00	567 00	567 00	567 00	

Means for SNK Groups
Table 5

CPU

SNK Group	Exact*	Exact	Partial	Range	Nearest	Best
A	1 000	1 000	1 000	1 000	1 000	1 000
B	990	988	927	896	967	899
C	983	985	116	000	934	000
D	000	000	000		000	

Wall Clock Time

SNK Group	Exact*	Exact	Partial	Range	Nearest	Best
A	1 000	1 000	1 000	1 000	1 000	1 000
B	991	987	875	896	965	939
C	987	000	829	615	572	519
D	000		000	000	000	000

I/O Counts

SNK Group	Exact*	Exact	Partial	Range	Nearest	Best
A	1 000	1 000	1 000	1 000	1 000	1 000
B	991	989	856	726	877	942
C	985	986	789	701	569	000
D	000	000	000	000	000	

Normalized SNK Coefficients
Table 6

The data base designer must assign relative weights of importance to each query class level For this experiment, the following weights were assumed

exact* -----	05
exact ------	05
partial ----	20
range ------	50
nearest ----	10
best -------	10

These parameters are purely application dependent and typically reflect some measure of frequency of query class use The next step is to calculate a sum of products for each file structure that

CPU

BKD	990(05) +	988(05) + 1 000(20) +		
	1 000(50) +	967(10) + 1 000(10)	=	996
UKD	983(05) +	985(05) + 927(20) +		
	1 000(50) + 1 000(10) + 1 000(10)		=	984
FL	000(05) +	000(05) + 116(20) +		
	896(50) +	934(10) + 899(10)	=	655
QD	1 000(05) + 1 000(05) +	000(20) +		
	000(50) +	000(10) + 000(10)	=	100

Wall Clock Time

QD	1.000(05) + 1 000(05) + 1 000(20) +			
	1.000(50) + 1 000(10) + 1 000(10)		=	1 000
UKD	.987(05) +	987(05) + 875(20) +		
	.896(50) +	965(10) + 939(10)	=	912
BKD	.991(05) +	987(05) + 829(20) +		
	.615(50) +	572(10) + 519(10)	=	681
FL	000(05) +	000(05) + 000(20) +		
	.000(50) +	000(10) + 000(10)	=	000

I/O Counts

QD	1 000(05) + 1 000(05) + 1 000(20) +			
	1 000(50) +	569(10) + 1 000(10)	=	957
BKD	991(05) +	989(05) + 856(20) +		
	726(50) +	877(10) + 942(10)	=	815
UKD	985(05) +	986(05) + 789(20) +		
	701(50) + 1 000(10) +	942(10)	=	801
FL	000(05) +	000(05) + 000(20) +		
	000(50) +	000(10) + 000(10)	=	000

File Structure Rankings by Metric
Table 7

reflects its performance for each metric For a given file structure, the product of each query class weight is multiplied by its corresponding query group SNK normalized factor, the sum of these products is the measure used to rank the file structures for the given metric Table 7 summarizes the calculations for this experiment The file structures are listed in order of preference for each metric

The rankings for each metric are different, in fact the first ranked file structure for the wall clock time and I/O counts is the last ranked file structure for the CPU time The next step is to propose the most appropriate file structure for the application Again the data base designer must supply relative measures of importance for the metrics For this experiment, the following weights were assumed

CPU time ----------	3
Wall clock time ----	5
I/O counts ---------	2

These weights are also application dependent As indicated in the experiment design, these metrics represent tradeoffs between cost (CPU and I/O) and response (wall clock time) The sum of products for each file structure in Table 7 is used to rank the file structures by metrics Multiplying each of these factors for a given file structure by the given metric weight and summing these partial products gives an overall measure for each file structure that incorporates both query class and the metrics Table 8 gives the results of these final calculations

UKD	984(	3) +	912(	5) +	801(	2) =	911
BKD	996(	3) +	681(	5) +	815(	2) =	802
QD	100(	3) + 1	000(	5) +	957(	2) =	721
FL	655(	3) +	000(	5) +	000(	2) =	197

Most Appropriate File Structure
Table 8

The file structures are ranked in order of preference The K-d trees were ranked over the quad-tree which was ranked over the flat file Of the two K-d tree implementations, the unbalanced K-d tree ranked over the balanced K-d tree This ranking reflects considerations about the data the queries, and the environment during the process of suggesting the most appropriate file structure

Acknowledgments

The data for this experiment was made available through the kind efforts of Dr Daniel B Hier, M D of the Neurology Department at Michael Reese Hospital In addition, the helpful comments of Jim Vandendorpe and Jon Bentley are greatly appreciated Finally, we are grateful to AT&T Bell Laboratories for use of the computing resources at the Indian Hill Computation Center for this experiment

References

[1] Banks, G , Caplan, L R , and Hier, D B 1983 The Michael Reese Hospital Stroke Registry - A Microcomputer-Implemented Data Base 7th Annual Symposium on Computer Applications in Medical Care IEEE Computer Society 724-727

[2] Beckley, D A , Evens, M W , and Raman, V K 1984 An Experiment With Balanced and Unbalanced K-D Trees for Associative Retrieval 8th Int Computer Software and Applications Conf Proceedings IEEE Computer Society IEEE Catalog No 84CH2096-6 256-262

[3] Beckley, D A , Evers, M W , and Raman, V K 1985 Empirical Comparison of Associative File Structures International Conference on Foundations of Data Organization Kyoto, Japan May, 1985 (to be published)

[4] Bentley, J L 1975 Multidimensional Binary Search Trees Used for Associative Searching Communications of the ACM Vol 18, No 9 509-517

[5] Bentley, J L 1979 Multidimensional Binary Search Trees Used in Database Applications IEEE Transactions on Software Engineering Vol SE-5, No 4 333-340

[6] Bentley, J L and Friedman, J H 1978 Fast Algorithms for Constructing Minimal Spanning Trees in Coordinate Spaces IEEE Transactions on Computers Vol C-27, No 2 97-105

[7] Bentley, J L 1979 Decomposable Searching Problems Information Processing Letters Vol 8, No 5 244-251

[8] Bentley, J L and Friedman, J H 1979 Data Structures for Range Searching Computing Surveys Vol 11, No 4 397-409

[9] Bentley, J L 1980 Multidimensional Divide-and-Conquer Communications of the ACM Vol 23, No 4 214-229

[10] Cardenas, A F 1973 Evaluation and Selection of File Organization - A Model and System Communications of the ACM Vol 16, No 9 540-548

[11] Cardenas, A F 1975 Analysis and Performance of Inverted Data Base Structures Communications of the ACM Vol 18, No 5 253-263

[12] Cardenas, A F , Sagamang, J P 1977 Doubly-Chained Tree Data Base Organization - Analysis and Design Strategies The Computer Journal Vol 20, No 1 15-26

[13] Chang, J M and Fu, K S 1978 Dynamic Clustering Techniques for Physical Database Design TR-EE 78-49, Purdue University

[14] Chang, J M and Fu, K S 1979 Extended K-D Tree Data Base Organization A Dynamic Multi-Attribute Clustering Method 3rd Int Computer Software and Applications Conf Proceedings IEEE Computer Society IEEE Catalog No 79CH1515-6C 39-43

[15] Chang, J M and Fu, K S 1980 A Dynamic Clustering Technique for Physical Database Design ACM-SIGMOD 1980 International Conference on Management of Data Association for Computing Machinery ACM Order No 472800

[16] Comer, D 1979 The Ubiquitous B-tree Computing Surveys Vol 11, No 2 121-137

[17] Dyer, C R , Rosenfeld, A , and Samet, H 1980 Region Representation Boundary Codes from Quadtrees Communications of the ACM Vol 23, No 3 171-179

[18] Finkel, R A and Bentley, J L 1974 A Data Structure for Retrieval on Composite Keys Acta Informatica Vol 4 1-9

[19] Friedman, J H , Bentley, J L , and Finkel, R A 1977 An Algorithm for Finding Best Matches in Logarithmic Expected Time ACM Transactions on Mathematical Software Vol 3, No 3 209-226

[20] Gargantini, I 1982 An Effective Way to Represent Quadtrees Communications of the ACM Vol 25, No 12 905-910

[21] Lum, V Y , Ling, H , and Senko, M E 1970 Analysis of a Complex Data Management Access Method by Simulation Modeling AFIPS Proceedings of Fall Joint Computer Conference AFIPS Press, Arlington, Va Vol 37 211-222

[22] Overmars, M H and Van Leeuwen, Jan 1982 Dynamic Multi-Dimensional Data Structures Based on Quad- and K-d Trees Acta Informatica Vol 17 267-285

[23] Robinson, J T 1981 The K-D-B-Tree A Search Structure for Large Multidimensional Dynamic Indexes ACM-SIGMOD 1981 International Conference on Management of Data Association for Computing Machinery

[24] Samet, H 1980 Region Representation Quadtrees from Boundary Codes Communications of the ACM Vol 23, No 3 163-170

[25] Samet, H 1980 Deletion in Two-Dimensional Quad Trees Communications of the ACM Vol 23, No 12 703-710

[26] Samet, H 1984 The Quadtree and Related Hierarchical Data Structures Computing Surveys Vol 16, No 2 187-260

[27] Saxe, J B and Bentley, J L 1979 Transforming Static Data Structures to Dynamic Structures Proc 20th Annual IEEE Symposium on Foundations of Computer Science IEEE Catalog No 79CH1471-2 148-168

[28] Snedecor, G W and Cochran, W G 1980 Statistical Methods 7th ed The Iowa State University Press Ames, Iowa

[29] Vandendorpe, J 1980 A Crash Tolerant B-tree Data Structure for Data Base Retrieval Systems Ph D Thesis, Illinois Institute of Technology

[30] Van Leeuwen, J and Overmars, M H 1983 Stratified Balanced Search Trees Acta Informatica Vol 18 345-359