Quad Trees: A Data Structure for Retrieval on Composite Keys.

2 authors, including:

Raphael Finkel
University of Kentucky
124 PUBLICATIONS   **6,199** CITATIONS

Some of the authors of this publication are also working on these related projects:

Project   Create new project "Answer-set programming" View project

Project   Morphological tools View project

Quad Trees
A Data Structure for Retrieval on Composite Keys

R. A. Finkel and J. L. Bentley

Summary. The quad tree is a data structure appropriate for storing information to be retrieved on composite keys. We discuss the specific case of two-dimensional retrieval, although the structure is easily generalised to arbitrary dimensions. Algorithms are given both for staightforward insertion and for a type of balanced insertion into quad trees. Empirical analyses show that the average time for insertion is logarithmic with the tree size. An algorithm for retrieval within regions is presented along with data from empirical studies which imply that searching is reasonably efficient. We define an optimized tree and present an algorithm to accomplish optimization in *n* log *n* time. Searching is guaranteed to be fast in optimized trees. Remaining problems include those of deletion from quad trees and merging of quad trees, which seem to be inherently difficult operations.

Introduction

One way to attack the problem of retrieval on composite keys is to consider records arranged in a several-dimensional space, with one dimension for every attribute. Then a query concerning the presence or absence of records satisfying given criteria becomes a specification of some (possibly disconnected) subset of that space. All records which lie in that subset are to be returned as the response to the query.

The retrieval of information on only one key has been well studied. Experience has shown binary trees serve as a good data structure for representing linearly ordered data, and that balanced binary trees provide a guaranteed fast structure (Knuth, 6.2.3).

This paper will discuss a generalization of the binary tree for the treatment of data with inherently two-dimensional structure. One clear example of such records is that of cities on a map. A sample query might be: "Find all the cities which are within 300 miles of Chicago or north of Seattle." The data structure we propose to handle such queries is called a quad tree. It will be obvious that the basic concepts involved are easily generalized to records of any dimensionality.

Definitions and Notation

The location of records with two-dimensional keys will be stored in a tree with out-degree four at each node. Each node will store one record and will have up to four sons, each a node. The root of the tree divides the universe into four quadrants, namely NE, NW, SW, and SE (using the map analogy). Let us call these quadrants one, two, three and four, respectively. Fig. 1 shows the correspondence between a simple tree and the records it represents.

The convention we use for points which lie directly on one of the quadrant lines emanating from a node is as follows: Quadrants one and three are closed,

and quadrants two and four are open. Thus a point on the line due east of a node is in quadrant one of that node. Actually, the convention assumed is of little import to the basic idea behind the tree; the one we have chosen is nice for the purpose of deciding quickly in which quadrant of the root a given node lies.

Notice that we do not take into account the possibility of nonunique records (collisions). If collisions are indeed valid in a particular application, an extra pointer could be placed in each node for a linear list of similar records, or some more sophisticated structure could be used to store the collisions.

Given two records, say A and B, we define COMPARE (A, B) to be an integer representing which quadrant of A holds B. We will define COMPARE (A, A) to be 0.

It is occasionally convenient to be able to discuss the direction directly opposite to a given direction. We will use the notation CONJUGATE (DIREC-TION) to represent the direction in question. For example, conjugate $(3) = 1$. A simple formula for conjugate is conjugate $(N) = ((N + 1) \bmod 4) + 1$.

It is not necessary to specify any exact implementation for nodes; many alternatives are possible, and the choice made will almost certainly be language-dependent. However, we need some notation for the purpose of discussing the algorithms. Therefore, let us agree to these conventions: There is a datatype NODE; the entire tree is a node. To denote a subtree of a node, say the third subtree of the node called ELM, we will write ELM [3]. NULL is the empty node. By convention, the zeroth subtree (say ELM [0]) is always NULL.

The algorithms in this paper will be presented in an ad-hoc version of ALGOL.

Insertion

Insertion of new records into quad trees is based on the same philosophy that governs insertions into binary trees; at each node, a comparison is made and the correct subtree is chosen for the next test; upon falling out of the tree, the algorithm knows where to insert the new record.

Here is an algorithmic description of the process:

```
PROCEDURE INSERT (NODE VALUE K, R);
    BEGIN
    COMMENT: Inserts record K in the tree whose root is R;
    INTEGER DIRECTION; COMMENT: Direction from father to
        son, i.e. 1, 2, 3, or 4;
    DIRECTION ←COMPARE (R, K);
    WHILE R [DIRECTION] ≠ NULL DO
        BEGIN
        COMMENT: Each iteration dives one level deeper;
        R ← R [DIRECTION];
        DIRECTION ← COMPARE (R, K);
        END;
    IF DIRECTION = 0 THEN RETURN; COMMENT: Node
        already exists;
    R [DIRECTION] ← K;
    END
```
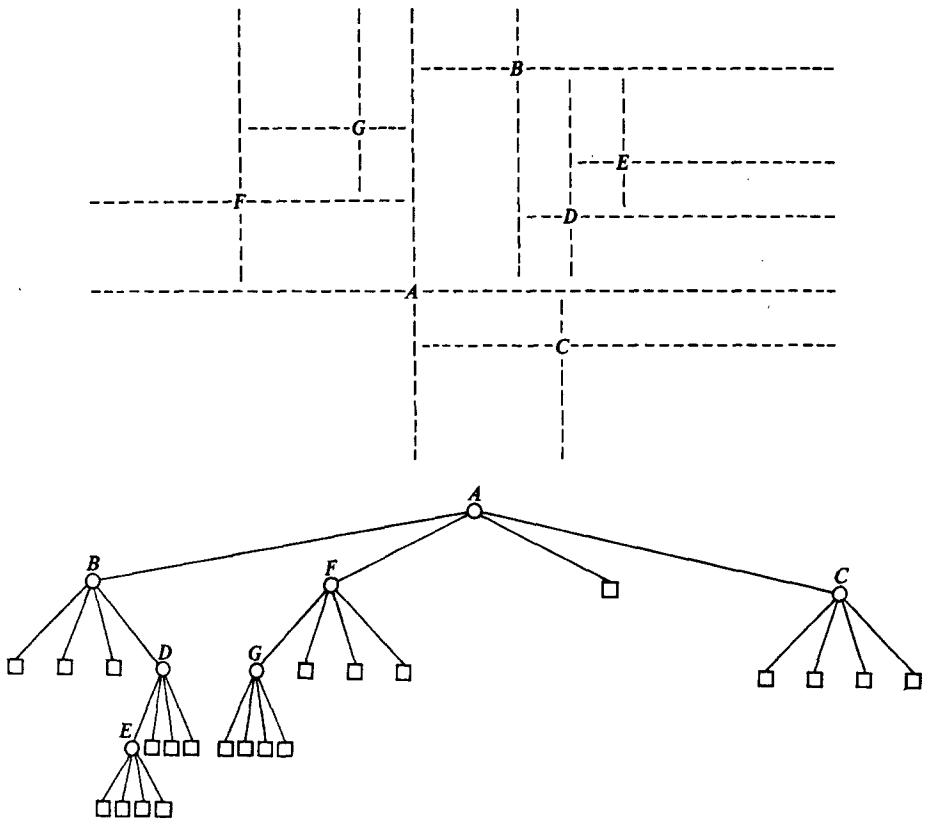
Fig. 1. Correspondence of a quad tree to the records it represents. RECORDS A, B, C, D, E, F, G. Null subtrees are indicated by boxes, but they do not appear explicitly in computer memory

This algorithm has been used to construct many trees of various sizes in order to collect statistics on its performance. The nodes were uniformly distributed in both coordinates, with key values ranging from 0 to $2^{31}-1$. Table 1 is a summary of the results.

Here X is the quantity (AVERAGE TPL)$/(n \log n)$, where n is the number of nodes, and the natural logarithm is used. The Low and High X are one standard

Table 1. Data on straightforward insertion

Number of nodes	Total path length		X	Low X	High X	Number of trees
	AVG	STD DEV				
25	67.21	8.699	0.8352	0.7271	0.9433	300
50	168.4	17.34	0.8608	0.7722	0.9495	300
100	403.5	30.68	0.8763	0.8096	0.9429	150
1000	6288	325.6	0.9103	0.8632	0.9575	30
10000	84705	2882	0.9197	0.8884	0.9510	10

1*

deviation on either side of X. These are of mild interest; they show that for random nodes the algorithm seldom gets much worse than the average.

The most interesting aspect of Table 1 is the column marked X; its slowly growing nature implies that the TPL of a quad tree under random insertion is roughly proportional to $N \log N$, where N is the number of nodes. Therefore, point searching in a quad tree can be expected to take about $\log N$ probes.

It should be noted, however, that the extreme case is much worse. If each successive node gets placed as a son of the currently lowest node in the tree, then the resulting tree will have TPL of $n(n-1)/2$. Thus worst-case insertion and searching are order n^2 operations.

More Sophisticated Insertion

In an effort to lessen the TPL and to find some analog to balancing of binary trees, a simple balancing algorithm was developed. It does not aim to keep the height of sons of a given node within some fixed distance of each other, but rather takes note of the fact that some simple situations can be represented in two ways, one of which has lower TPL. Two examples are shown in Fig. 2 and 3; the former demonstrates what is termed a single balance; the latter a double balance (performed if COMPARE (B, C) is the conjugate of COMPARE (A, B)).
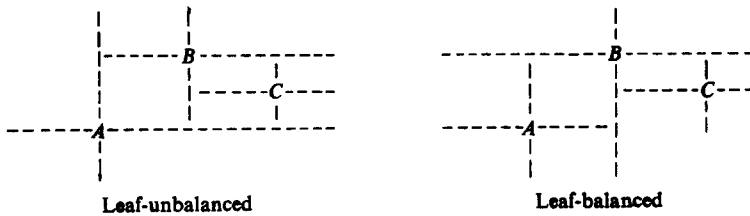

Fig. 2. Single balance
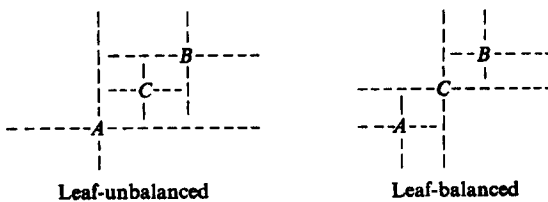
Fig. 3. Double balance

Table 2. Data on leaf-balanced insertion

Number of nodes	Total path length		X	Low X	High X	Number of trees
	AVG	STD DEV				
25	60.57	5.703	0.7526	0.6818	0.8235	300
50	152.9	12.37	0.7818	0.7185	0.8450	300
100	367.6	24.13	0.7982	0.7458	0.8506	150
1000	5812	247.0	0.8414	0.8057	0.8772	30
10000	78020	1960	0.8471	0.8258	0.8684	10

Table 2 gives statistics for the algorithm which performs these leaf balances during insertion. The nodes used were identical to those used in the tests of straightforward insertion. Of interest are the about 10% lower values of X and the slightly lower standard deviations. It should be remarked, however, that at times a tree built by the leaf-balancing algorithm has higher TPL than a tree built by the straightforward insertion algorithm out of the same nodes, arriving in the same order.

Searching

There are two general classes of searches facilitated by quad trees. The first is a point search for a single record, invoked by a query like "Is the point (37, 18) in the data structure, and what is the address of the node representing it?" The second is a region search, invoked by something like "What are all the points within a circle of center (15, 31) and radius 7?" or (if the quad tree's two dimensions represent age and annual income) "Who are all the people between the ages of 21 and 28 who earn between \$ 12000 and \$ 14000 per year?" Our definition of point search corresponds to Knuth's "simple query" (Knuth, 6.5) and our region search encompasses both a "range query" and a "boolean query" in Knuth's terminology.

Point searching is accomplished by an algorithm much like that used for insertion. The average time required to do a point search for a node in the tree is proportional to

$$\text{TPL}/(\text{number of nodes in tree}).$$

This is the algorithm used to perform a region search:

```
PROCEDURE REGIONSEARCH (NODE VALUE P; REAL VALUE L, R, B, T);
    BEGIN
        COMMENT: Recursively searches all subtrees of P to find all nodes within
            window bounded by L (left), R (right), B (bottom), and T (top) which
            are in region in question;
        REAL XC, YC; COMMENT: The X and Y coordinates of P;
        XC ← X (P);
        YC ← Y (P);
        IF IN_REGION (XC, YC) THEN FOUND (P);
        IF P [1] ≠ NULL AND RECTANGLE_OVERLAPS_REGION
            (XC, R, YC, T)
            THEN REGIONSEARCH (P [1], XC, R, YC, T);
        IF P [2] ≠ NULL AND RECTANGLE_OVERLAPS_REGION
            (L, XC, YC, T)
            THEN REGIONSEARCH (P [2], L, XC, YC, T);
        IF P [3] ≠ NULL AND RECTANGLE_OVERLAPS_REGION
            (L, XC, B, YC)
            THEN REGIONSEARCH (P [3], L, XC, B, YC);
        IF P [4] ≠ NULL AND RECTANGLE_OVERLAPS_REGION
            (XC, R, B, YC)
            THEN REGIONSEARCH (P [4], XC, R, B, YC);
    END
```

This algorithm must be supplied with two procedures to be used during searching and one procedure that is informed of each node found in the region. Procedure IN_REGION is a boolean procedure which is passed the x and y coordinates of a node and returns TRUE if an only if the node is in the region. Procedure RECTANGLE_OVERLAPS_REGION is passed the left, right, bottom, and top extreme values of a rectilinearly oriented rectangle. (For example, the parameters $(4, 6, 1, 3)$ represent the rectangle bounded by the lines $x = 4$, $x = 6$, $y = 1$, $y = 3$). RECTANGLE_OVERLAPS_REGION returns TRUE if the region being searched overlaps the described rectangle and FALSE otherwise. Procedure FOUND is given each node found in the region to allow processing.

The algorithm itself is recursive and is initially invoked by the statement

REGIONSEARCH (ROOT, MIN_X, MAX_X, MIN_Y, MAX_Y)

where ROOT is the root node of the tree, and the rest of the parameters are defined such that for all x and y values represented in the quad tree,

$$(\text{MIN\_}X \leq x \leq \text{MAX\_}X) \quad \text{and} \quad (\text{MIN\_}Y \leq y \leq \text{MAX\_}Y).$$

Here are the auxiliary procedures IN_REGION and RECTANGLE_OVER-LAPS_REGION necessary to search for all points in the rectilinearly oriented rectangle defined by $(BP \leq y \leq TP)$ and $(LP \leq x \leq RP)$:

```
BOOLEAN PROCEDURE IN_REGION (REAL VALUE X, Y);
        COMMENT: Returns TRUE if (X, Y) is in region;
        RETURN ((LP ≤ X) ∧ (X ≤ RP) ∧ (BP ≤ Y) ∧ (Y ≤ TP));
BOOLEAN PROCEDURE RECTANGLE_OVERLAPS_REGION
           (REAL VALUE L, R, B, T);
        COMMENT: Returns TRUE if region overlaps rectangle
           bounded by L, R, B, and T;
        RETURN ((L ≤ RP) ∧ (R ≥ LP) ∧ (B ≤ TP) ∧ (T ≥ BP));
```

Similar procedures can be defined to search for points in any connected geometric figure. After these basic procedures have been defined, the logical operators AND, OR and NOT can be used to search within unions, intersections, and complements of regions. Thus, the specification of the region of interest is highly flexible.

The versatility of this algorithm makes a formal analysis very difficult, so we have conducted some empirical tests to investigate its efficiency. We let the x and y coordinates of all points in the quad tree be real-valued numbers in $[0, 1]$. The regions of search were randomly located, rectilinearly oriented squares having a given edge size. For each of the tree sizes for which we gathered data, we randomly generated four quad trees. For all of the edge sizes presented we searched each of the four trees 25 times; a total of 100 searches per edge size. In Table 3 we show for every combination of tree and edge size the number of nodes visited in the 100 searches (Visited), of those visited the number that were actually in the region (Found), and significant ratios of these data. For example, there were four 250-node quad trees tested. On each of these trees, 25 searches were made for all the points within squares of edge 0.125 located at random

positions within the unit square. In these 100 searches a total of 1820 nodes were visited. Of the 1820 visited, 387 were in the particular squares being searched. The ratio of those visited to the number of searches was 18.20, the ratio of the number of nodes found to the number of searches was 3.87, and the ratio of those visited to those found was 4.70.

The two most significant figures in the table are the values Visited/Found and Visited/Search. Visited/Found is indicative of the amount of total work the algorithm has to do to find a particular node in the region it is searching. Visited/Search shows the amount of work the algorithm does to find all nodes in a certain region. It is pleasant to note in Table 3 that as Visited/Search increases, Visited/Found decreases, implying one is going to be low as the other is high. It is likely that the lower measure will be of more concern to the user; when one is gathering a large amount of data from a certain region one is more concerned about the cost per collected node, and when one is making a large number of small searches one is more concerned about the cost of each search.

Table 3. Region-search data

Number of nodes	Edge Size	Visited	Found	Number of Searches	Visited/ Search	Found/ Search	Visited/ Found
125	0.03125	598	13	100	5.98	0.13	46.00
	0.0625	789	41	100	7.89	0.41	19.24
	0.125	1218	206	100	12.18	2.06	5.91
	0.25	2195	799	100	21.95	7.99	2.75
	0.5	5188	3130	100	51.88	31.30	1.66
250	0.03125	777	22	100	7.77	0.22	35.32
	0.0625	1074	103	100	10.74	1.03	10.43
	0.125	1820	387	100	18.20	3.87	4.70
	0.25	3562	1536	100	35.62	15.36	2.32
	0.5	9550	6390	100	95.50	63.90	1.50
500	0.03125	975	55	100	9.75	0.55	17.73
	0.0625	1493	205	100	14.93	2.05	7.28
	0.125	2641	754	100	26.41	7.54	3.50
	0.25	6248	3163	100	62.48	31.63	1.97
	0.5	17453	12860	100	174.53	128.60	1.36
1000	0.03125	1316	99	100	13.16	0.99	13.29
	0.0625	2144	376	100	21.44	3.76	5.70
	0.125	4246	1611	100	42.46	16.11	2.64
	0.25	10100	6172	100	101.00	61.72	1.64
	0.5	31845	24880	100	318.45	248.80	1.28
2000	0.03125	1619	183	100	16.19	1.83	8.85
	0.0625	2906	786	100	29.06	7.86	3.70
	0.125	6803	3106	100	68.03	31.06	2.19
	0.25	18347	12627	100	183.47	126.27	1.45
	0.5	60581	49943	100	605.81	499.43	1.21
4000	0.03125	2407	397	100	24.07	3.97	6.06
	0.0625	4369	1504	100	43.69	15.04	2.90
	0.125	11096	6218	100	110.96	62.18	1.78
	0.25	33133	24610	100	331.33	246.10	1.35
	0.5	114767	99875	100	1147.67	998.75	1.15

Deletion

It turns out to be very difficult to perform deletions from quad trees. The difficulty lies in deciding what to do with the subtrees that were attached to the deleted node. It is necessary to merge them into the rest of the tree, but merging is not an easy process. In fact, it seems that one cannot do better than to reinsert all of the stranded nodes, one by one, into the new tree. This answer is not very satisfactory, and it is a matter of some interest whether there exists any merging algorithm that works faster than $n \log n$, where n is the total number of nodes in the two trees to be merged.

Some attempt has been made to take advantage of the hope that not all subtrees of a newly-merged node need be disjointed, that some of them (which ones depending on in what direction the newly-merged son lies with respect to its father) can be left intact. This has so far not been found to be of much help.

An Optimization Algorithm

At times the simple balancing mentioned earlier is not sufficient; the nodes might be presented in a very lopsided order, or the tree might be needed frequently for point searches and never updated. These conditions make it worthwhile to consider expending extra effort at the outset to optimize the tree.

By an optimized tree we will mean a quad tree such that every node K has this property: No subtree of K accounts for more than one half of the nodes in the tree whose root is K. The first step in building an optimized quad tree from a collection of records is to order the nodes lexicographically primarily by the x coordinate and secondarily by the y coordinate. A simple recursive algorithm to complete optimization is this: Given a collection of lexicographically ordered records, we will first find one, R, which is to serve as the root of the collection, and then we will regroup the nodes into four subcollections which will be the four subtrees of R. The process will be called recursively on each subcollection. The root that is then found for each subcollection is linked in as an appropriate son of R. To find the root R, select the median element of the ordered list. The reason this works is that all records falling before R in the ordered list will lie in quadrants 3 and 4; those falling after it will lie in quadrants 1 and 2. Thus the condition is met: No subtree can possibly contain more than half the total number of nodes. The maximum path length in an optimized tree of n nodes is therefore $\lfloor \log_2(n) \rfloor$ and the maximum TPL is

$$\sum_{1 \leq i \leq n} \lfloor \log_2(i) \rfloor.$$

The running time for this algorithm is on the order of $n \log n$: The ordering step on n elements will take $n \log n$. On each level of recursion the selection of the median requires n, the regrouping of the records takes n and can be done in such a way that each group is still ordered. The depth of recursion will be the maximum path length which is bounded by $\log n$, so the time for optimization will also be $n \log n$.

Empirical studies with optimized trees of random nodes have shown that the TPL of trees after the treatment is roughly 15% lower than that obtained by

straightforward insertion. The measures of searching effectiveness (nodes visited per nodes found and nodes visited per search) remain roughly the same.

Conclusions

The quad tree seems to be an efficient means of storage for two-dimensional data. The most straightforward insertion algorithm yields $n \log n$ performance when given random keys. Region searching is quite efficient.

Unfortunately, deletion from quad trees and merging of two quad trees is not easy.

The basic concepts involved can easily be generalized to an arbitrary number of dimensions. (In m dimensions, each node has 2^m sons.)

References

Knuth, D. E.: The art of computer programming, vol. 3: Sorting and Searching. Reading (Mass.): Addison-Wesley 1973

R. Finkel
Computer Science Department
Stanford University
Stanford, Calif. 94305/U.S.A.

J. Bentley
Computer Science Department
University of North Carolina
Chapel Hill, North Carolina 27514/U.S.A.