Range Searching using Kd Tree.

Hemant M. Kakde.

25-08-2005

1 Introduction to Range Searching.

At first sight it seems that database has little to do with geometry. The queries about data in database can be interpreted geometrically.In this case the records in the database are transformed into points in multidimensional space and the queries about records are transformed into the queries over this set of points. Every point in the space will have some information of person associated with it.

Consider the example of the database of personal administration where the general information of each employee is stored. Consider an example of query where we want to report all employees born between 1950 and 1955, who earns between Rs.3000 and Rs.4000 per month. The query will report all the points that whose frost co-ordinate lies between 1950 and 1955, and second co-ordinate lies between 3000 and 4000.

In general if we are interested in answering queries on d - fields of the records in our database, we transforms the records to points in d-dimensional space. Such a query is called rectangular range query, or an orthogonal range query.

In Range Search problems, the collection of points in space and the query is some standard geometric shape translatable in space. Range search consist either of retrieving (report problems) or of counting (count problems) all points contained within the query domain.

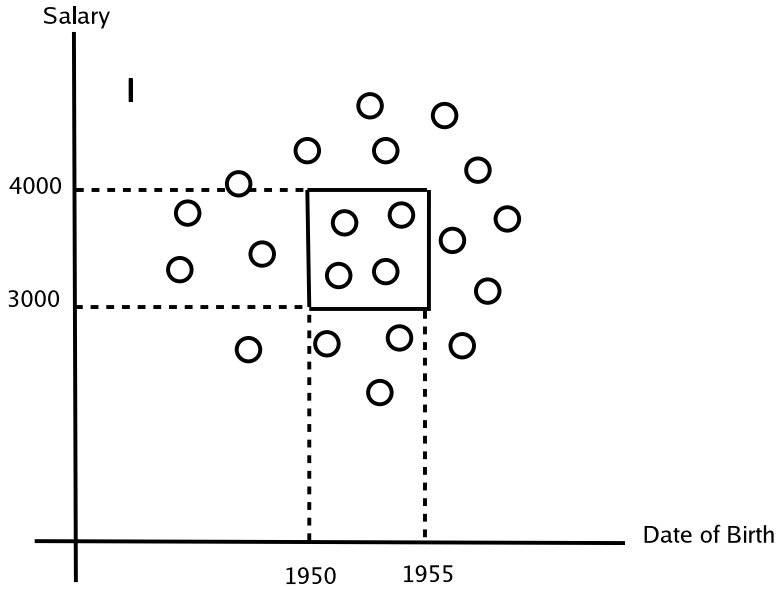

Fig : Interpreting a database Query Geometrically

Figure 1:

2 1 - Dimensional Range Searching.

Let us consider the set of points P=p_1,p_2.....p_n. We have to search the range [x,x'] and we have to report which points lies in that range. To solve the problem of range searching we use the data structure known as balanced binary search tree T. The leaves of the T store the points of P and the internal nodes of T will store the splitting values to guide the search. Let x_v denote the value stored at each split node v. The left subtree of the node v contains all points smaller than or equal to x_v, and the right subtree contains all the points strictly greater than x_v.

Let we search with x and x' in T. μ and μ' be the two leaves where the searches end, respectively. Then the points in the interval [x:x'] are stored in the leaves in between μ and μ' including μ and μ'. To find the leaves between μ and μ', we select the tree rooted at nodes v in between the two search paths whose parent are on the search path. To find these nodes we first find the node v_{split} where the paths to x and x' splits.

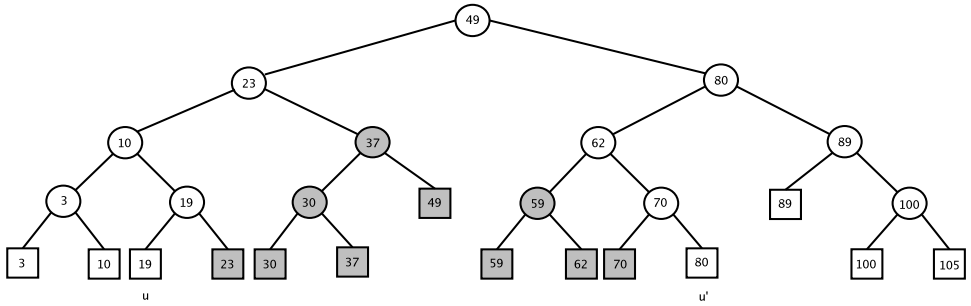

Fig : A 1 - dimensional range query in a binary search tree

Figure 2:

2.1 Algorithm

To find the split node let us consider lc(v) and rc(v) denote the left and right child, respectively, of the node v.

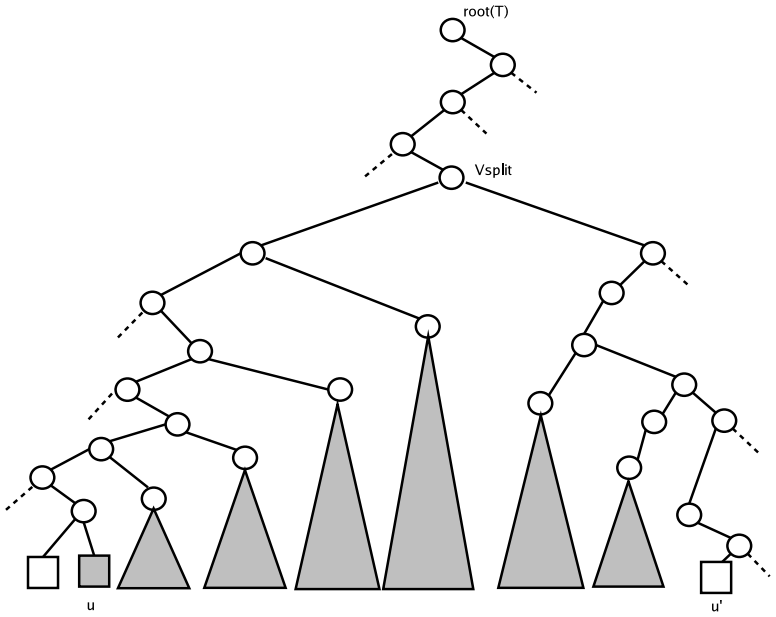

Fig : The Selected Subtrees

Figure 3:

Procedure name FINDSPLITNODE
Input: A Tree T and two values x and x' with x≤x'.
Output: The node v where the paths to x and x' split, or the leaf where both path ends.

1: v←root(T).
2: **while** v is not a leaf **and** (x'≤x_v **or** x>x_v).
3: **do if** x' ≤ x_v
4: **then** v← lc(v)
5: **else** v← rc(v)
6: return v

Starting from v_{split} we then follow the search path of x. At each node where the path goes left, we report all the leaves in the right subtree, because this subtree is in between the the two search paths. Similarly, we follow the path of x' and we report the leaves in the left subtree of the node where the path goes right. Finally we check the points stored at the leaves whether they lies in the range [x , x'] or not.

Now we see the query algorithm which uses the subroutine REPORTSUBTREE, which traverses the subtree rooted at the node and reports all the stored at its leaves.This subroutine takes takes the amount of time linear in the number of reported points, this is because the number of internal nodes of any binary tree is less than its internal node.

2.2 Algorithm

Procedure name 1DRANGEQUERY(T,[X:X'])
Input: A binary search tree T and a range [x,x'].
Output: All points stored in T that lie in the range.

1: v_{split}←FINDSPLITNODE(T,x,x').
2: **if** v_{split} is a leaf .
3: **then** check if the point stored at v_{split} must be reported.
4: **else** (* Follow the path to x and report the points in subtree right of the path*).
5: v← lc(v_{split})
6: **while** v is not a leaf.
7: **do if** x' ≤ x_v
8: **then** REPORTSUBTREE(rc(v))

9: v← lc(v)

10: **else** v← rc(v)

11: check if the point stored at the leaf v must be reported.

12: Similarly, follow the path to x', reports the points subtree left of the path, and check if the point stored at the leaf where the path ends must be reported.

2.2.1 Complexity analysis

The data structure binary search tree uses **O(n)** storage and it can be built in **O(n log n)** time. In worst case all the points could be in query range, so the query time will be **Θ(n)**. The query time of **Θ(n)** cannot be avoided when we have to report all the points. Therefore we shall give more refined analysis of query time. The refined analysis takes not only n, the number of points in the set P, into account but also k, the number of reported points.

As we know that the REPORTSUBTREE is linear in the number of reported points, then the total time spent in all such calls is **O(k)**. The remaining nodes that are visited are the nodes of the search path of x and x'. Because T is balanced, these paths have a length **O(log n)**. The time we spent at each node is **O(1)**, so the total time spent in these nodes is **O(log n)**, which gives a query time of **O(logn + k)**.

2.2.2 Theorem

Theorem 1 *Let P be the set of n points in one 1-dimensional space. The set P can be stored in balanced binary search tree, which uses* **O(n)** *storage and has* **O(n logn)** *construction time, such that the points in the query range can be reported in time* **O(k + logn)**, *where k is the number of reported points.*

3 Kd - Trees.

Consider the 2-dimensional rectangular range searching problem. Let P be the set of n points in the plane. The basic assumption is no two point have same x-coordinate, and no two points have same y-coordinate.

Definition 1 *A 2-dimensional rectangular range query on P asks for the points from P lying inside the query rectangle [x:x']*[y,y']. A point p :=(p_x,p_y) lies inside this rectangle if and only if,*

$$p_x \epsilon [x,x'] \text{ and } p_y \epsilon [y,y']$$

Let's consider the following recursive definition of the binary search tree : the set of (1-dimensional) points is split into two subsets of roughly equal size, one subset contains the point smaller than or equal to splitting value, the other contains the points larger than splitting value. The splitting value is stored at the root and the two subsets are stored recursively in two subtrees.

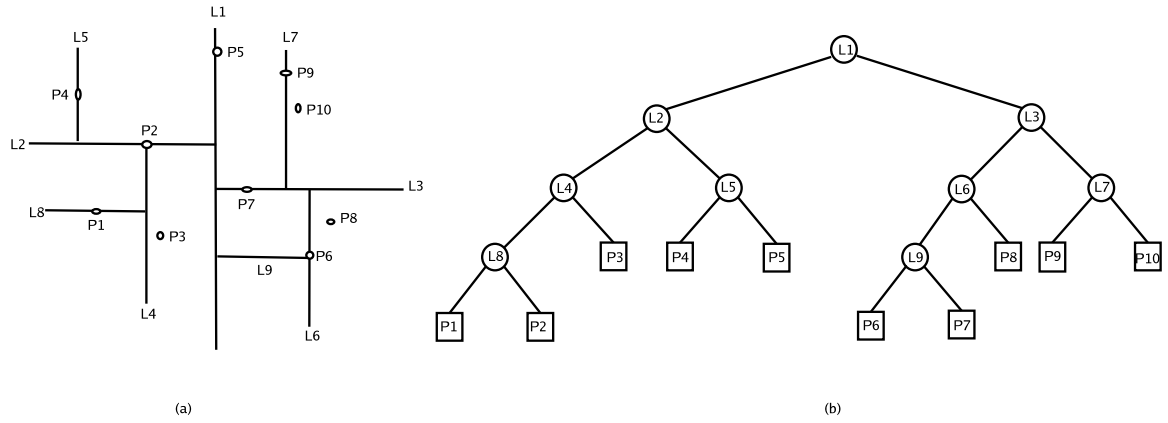

Fig : A Kd Tree : (a) The way the plane is divided, (b) Corresponing Binary Tree

Figure 4:

Each point has its x-coordinate and y-coordinate. Therefore we first split on x-coordinate and then on y-coordinate, then again on x-coordinate, and so on. At the root we split the set P with vertical line l into two subsets of roughly equal size. This is done by finding the median x- coordinate of the points and drawing the vertical line through it. The splitting line is stored at the root. P_{left}, the subset of points to left is stored in the left subtree,

and P_{right}, the subset of points to right is stored in the right subtree. At the left child of the root we split the P_{left} into two subsets with a horizontal line. This is done by finding the median y-coordinate if the points in P_{left}. The points below or on it are stored in the left subtree, and the points above are stored in right subtree. The left child itself store the splitting line. Similarly P_{right} is split with a horizontal line, which are stored in the left and right subtree of the right child. At the grandchildren of the root, we split again with a vertical line. In general, we split with a vertical line at nodes whose depth is even, and we split with horizontal line whose depth is odd.

3.1 Algorithm

Let us consider the procedure for constructing the kd-tree. It has two parameters, a set if points and an integer. The first parameter is set for which we want to build kd-tree, initially this the set P. The second parameter is the depth of the root of the subtree that the recursive call constructs. Initially the depth parameter is zero. The procedure returns the root of the kd-tree.

Procedure name BUILDKDTREE(P,depth)
Input: A set of points P and the current depth depth.
Output: The root of the kd-tree storing P.

1: **if** P contains only one point
2: **then return** a leaf storing this point
3: **else if** depth is even
4: **then** Split P into two subsets with a vertical line l through the median x-coordinate of the points in P.Let P_1 be the set of points to the left of l or on l, and let P_2 be the set of points to the right of l.
5: **else** Split P into two subsets with a horizontal line l through the median y-coordinate of the points in P.Let P_1 be the set of points to the below of l or on l, and let P_2 be the set of points above l.
6: v_{left}←BUILDKDTREE(P_1, depth +1).
7: v_{right}←BUILDKDTREE(P_2, depth +1).
8:Create a node v storing l, make v_{left} the left child of v, and make v_{right} the right child of v.

9:**return** v.

3.1.1 Construction time of 2-dimensional kd-tree

The most expensive step is to find the median. The median can be find in linear time, but linear time median finding algorithms are rather complicated. So first presort the set of points on x-coordinate and then on y-coordinate. The parameter set P is now passed to the procedure in the form of two sorted list. Given the two sorted list it is easy to find the median in linear time. Hence the building time satisfies the recurrence,

$$T(n)=O(1), \text{ if } n=1,$$
$$O(n)+2T(\lceil n/2 \rceil), \text{if } n>1$$

which solves to **O(n log n)**.
Because the kd-tree is the binary tree, and every leaf and internal node uses **O(1)** storage, therefore the total storage is **O(n)**.

Lemma 1 *A kd-tree for a set of n-points uses* **O(n)** *storage and and can be constructed in* **O(n logn)**.

The splitting line stored at the root partition the plane in two half-planes. The left child of the root corresponds to the left half-plane, and right child corresponds to right half-plane. The left child of the left child of the root, corresponds to the region bounded to the right by the splitting line stored at the root and bounded from above by the line stored at the left child of the root. In general the region corresponding to the node v is the rectangle, which can be unbounded on one or more sides. It is bounded by splitting lines stored at the ancestors of v.

We denote the region corresponding to the node v by region(v). The region of the root of the kd-tree is whole plane.Observe that a point is stored in subtree rooted at node v if and only if it lies in region(v). We have to search the subtree rooted at v only if the query rectangle intersects the region(v).

Observation 1 *We traverse the kd-tree, but visit only nodes whose region is intersected by query rectangle.*

When a region is fully contained in the query rectangle then report all the points stored at its subtree.When traverse reaches a leaf, we check whether the point is contained in the query region and, if so, report it.
let us consider the diagram given below.

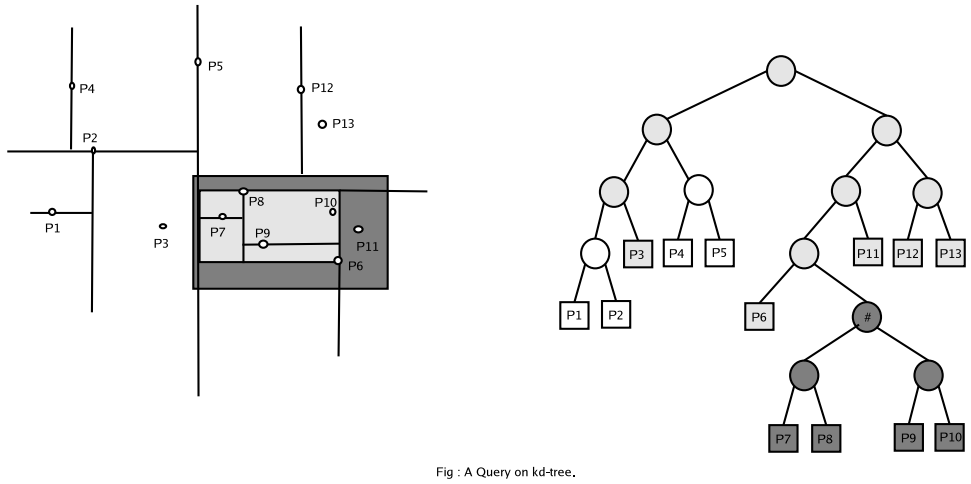

Figure 5:

The grey nodes are visited when when we query with grey rectangle. The node marked with star corresponds to a region that is completely contained in the query rectangle, which is shown by darker rectangle in fig. Hence, the dark grey subtree rooted at this node is traversed and all points stored in it are reported. The other leaves that are visited corresponds to region that are only partially inside the query rectangle. Hence, the points stored in them must be tested for inclusion in the query range; this results in point P_6 and P_{11} being reported, and points P_3, P_{12} and P_{13} not being reported.

3.2 Algorithm

Let us consider the procedure for searching the kd-tree. It has two parameters, the root of the kd-tree and the query range R.

Procedure name SEARCHKDTREE(v,R)
Input: The root of (a subtree of) a kd-tree, and a range R.

Output: All points at leaves below v that lies in range.

```
1: if v is a leaf
2: then  Report the stored at v if it lies in R
3: else if region(lv(c)) is fully contained in R
4:         then REPORTSUBTREE(lc(v))
5:         else if region(lc(v)) intersects R
6:         then SEARCHKDTREE(lc(v),R)
7:     if region(rv(c)) is fully contained in R
8:         then REPORTSUBTREE(rc(v))
9:         else if region(lc(v)) intersects R
10:        then SEARCHKDTREE(lc(v),R)
```

The region corresponding to the left child of a node v at even depth can be computed from region(v) as follows :

$$\text{region(lc(v))} = \text{region(v)} \bigcap l(v)^{left}$$

where l(v) is the splitting line stored at v, and $l(v)^{left}$ is the half plane to the left of and including l(v).

Lemma 2 *A query with an axis-parallel rectangle in a kd-tree storing n points can be performed in* $\mathbf{O(\sqrt{n}+k)}$ *time, where k is the number of reported points.*

First of all note that the time to traverse a subtree and report the points stored in its leaves is linear in number of reported points. Therefore the total time is $\mathbf{O(k)}$. It remains to bound the number of nodes visited by the query algorithm that are not in one of the traversed subtrees.For each such node v, the region(v) is intersected by, but not fully contained in the range.To analyze the number of such nodes, we shall bound the number of regions intersected by any vertical line. This will give us an upper bound on the number of regions intersected by the left and right edge of query rectangle. Similarly, we can find the number regions intersected by the top and bottom lines of query rectangle.

3.2.1 Complexity analysis

Let l be the vertical line, and T be a kd-tree. Let l(root(T)) be the splitting line stored at the root of the kd-tree. The line l intersects either the region to the left of l(root(T)) or the region to the right of l(root(T)), but not both.This observation seems to imply that Q(n), the number of intersected regions in the kd-tree storing a set of n points, satisfies the recurrence Q(n)=1+Q(n/2). But this is not true because the splitting lines are horizontal at the children of the root. This means that if the line l intersects the region(lc(root(T))), then it will always intersect the regions corresponding to both children of lc(root(T)). Hence the recurrence we get is incorrect. To write the correct recurrence for Q(n) we go down two steps in tree. Each of the four nodes at depth two in the tree corresponds to a region containing n/4 points. Two of the four nodes correspond to intersected regions, so we have to count the number of intersected regions in these subtrees recursively. Moreover, l intersects the regions of the root and of the root and of one of its children. Hence, Q(n) satisfies the recurrence

$$\begin{aligned} &\text{Q(n)=O(1),if n=1,} \\ &\text{2+2Q(n/4),if n>1.} \end{aligned}$$

This recurrence solves to $\mathbf{Q(n)=O(\sqrt{n})}$. In other words, any vertical line intersects $\mathbf{O(\sqrt{n})}$ regions in kd-tree.Similarly, horizontal line intersects $\mathbf{O(\sqrt{n})}$ regions. The total number of regions intersected by the boundary of a rectangular range query is bounded by $\mathbf{O(\sqrt{n})}$.

Theorem 2 *A kd-tree for a set P of n points can be build in* $\mathbf{O(n\ logn)}$ *time. A rectangular range query on the kd-tree takes* $\mathbf{O(\sqrt{n}+k)}$ *time, where k is the number of reported points.*

Kd-trees can be also be used for higher-dimensional spaces. The construction algorithm is very similar to the planar case. At the root, we split the set of points into two subsets of roughly the same size by a hyperplane perpendicular to the x1-axis. In other words, at the root the point set is partitioned based on the first coordinate of the points. At the children of the root the partition is based on second coordinate, at nodes at depth two on the third coordinate, and so on, until depth of d-1 we partition on last coordinate. Ai depth d we start all over again, partitioning on first coordinate. The recursion stops only

when one point is left, which is then stored at the leaf. Because a d-dimensional kd-tree for a set of n points is a binary tree with n leaves, it uses $\mathbf{O(n)}$ storage. The construction time is $\mathbf{O(n\ logn)}$.

Nodes in a d-dimensional kd-tree corresponds to regions, as in the plane. The query algorithm visits those nodes whose regions are properly intersected by the query range, and traverses subtree that are rooted at nodes whose region is fully contained in the query range. It can be shown that the query time is bounded by $\mathbf{O}(n^{1-1/d}+\mathbf{k})$.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, Prentice Hall India, New Delhi.

[2] Franco P. Preparata and Micheal Ian Shamos, *Computational Geometry - An Introduction*, Springer-Verlag,USA.