SOLVING DATABASE QUERIES USING ORTHOGONAL RANGE SEARCHING

CPS234: Computational Geometry I Prof. Pankaj Agarwal

Prepared by: Khalid Al Issa (khalid@cs.duke.edu)
Fall 2005

BACKGROUND : RANGE SEARCHING, a database-oriented view

Range searching is the side of geometry science in which a set of items is analyzed to determine the subset that belong to a given "range". In other words, we are given as an input a set of elements with certain properties, and we are also given a "range" of properties. So, our output is expected to be "the set of elements whose properties exist in the given range". In mathematical notation, we have:

INPUT:
P : *{P0, P1, P2,.. , Pn} a set of elements with given properties (ex. weights..etc)*
Q : *a range of properties (ex. Weights between x and y)*

OUTPUT:
The set of elements that belong in the "intersection" of P and Q.

Database queries happen to be a major application in which theories of range searching are applied. Let's take an example of a database query, and interpret it in geometrical form:

Assume we have a database for students' records, in which we keep every student's ID, name, number of completed hours, and the cumulative GPA. We now would like to apply the following SQL query to the database:

SQL> **SELECT** ID,NAME **FROM** STUDENTS **WHERE** HOURS **BETWEEN** 75 **AND** 90 **AND** GPA **BETWEEN** 2.5 **AND** 3.0

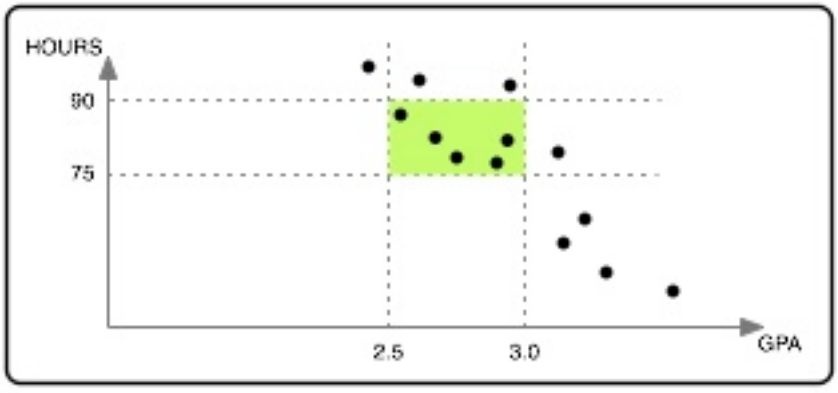

Figure.1: Range searching for a 2-D query

So this query above will give us the ID and name of every student whose GPA lies in the range [2.5,3.0] AND who has successfully completed hours in the range [75,90].

We notice that the query above is based on two elements: GPA and HOURS. So when we try to interpret the query in geometrical form, it will have two dimensions: the GPA would represent one dimension of the problem, and the HOURS will form the other dimension. The solution will be simply the area formed by the intersection of the two ranges generated from each dimension.

Similarly, database queries with a single element, like:

SQL> **SELECT** ID,NAME **FROM** STUDENTS **WHERE** GPA **BETWEEN** 2.7 **AND** 3.3

can be represented in a one-dimensional geometrical form. Also, database queries with more elements (3, 4 and more) will be interpreted in higher-dimension geometrical form. I will go over some of the data structures used to solve range searching queries of different dimensions.

1. SOLVING 1-D RANGE SEARCHING QUERIES

In order to solve a given query against some input, the set of data has to be represented in some form of data structure. We have mentioned that database queries with a single element can be interpreted in a one-dimensional geometrical form, and that will be represented on a **balanced binary search tree**.

1.1 USING BINARY SEARCH TREES

Let's look at the example mentioned above once again:

SQL> **SELECT** ID,NAME **FROM** STUDENTS **WHERE** GPA **BETWEEN** 2.7 **AND** 3.3

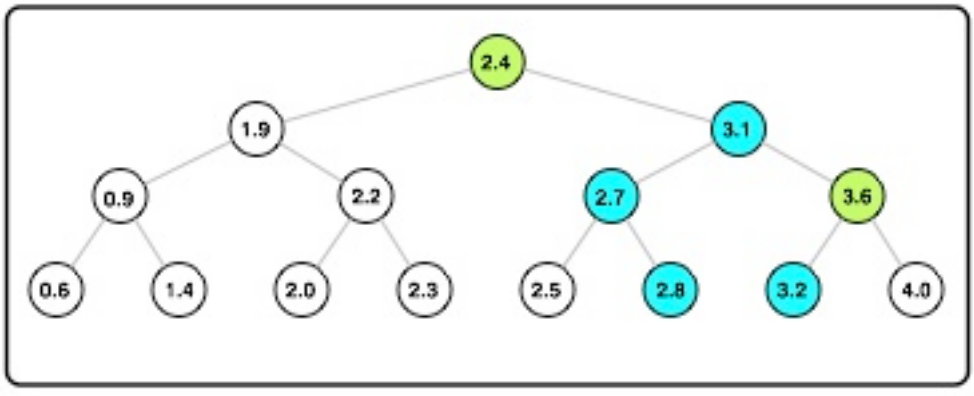

Figure.2: Range searching using binary search tree

Let's assume that the STUDENTS table has a number of students whose GPA's range between [0.0, 4.0]. So we will take these GPA values and arrange them in a balanced binary tree, which will be used later on to determine the location of both ends of the query range, and report all points in between them. The following diagram shows the entire tree of GPA's in the students' database:

The algorithm for determining both ends of the query (2.7 and 3.3 in this case) runs as follows:

1. Start from the root of the tree, and look for a node that has a value in between both ends, call it V_{split}.
2. If the value in the root is NOT in the range, it can not be used as V_{split}, so either move right – in case the root's value is less than the smaller end of the range, or move left – in case it's greater than the larger end of the range. We move until we encounter a value that lies in the range.
3. Whenever we encounter the node that is suitable for V_{split}, then its right sub-tree will contain the greater end of the range. Its left sub-tree will as well have the smaller end of the range.

So, by applying the algorithm to the tree shown in Figure.2, we start with the root, which has the value 2.4. Because 2.4 does not lie in the range of [2.7, 3.3], then we move to the right since 2.7 is greater than 2.4. Now, with 3.1 in hand, we know 3.1 lies in the range [2.7, 3.3], so we take it as V_{split}. We investigate both sub-trees of 3.1 to find both ends. From the left side, we find 2.7, which is in the range – but it's the lower end, so we stop moving to the left at this point, we, instead, check the right sub-tree of 2.7. Actually, we can blindly include the whole right sub-tree of 2.7 since it will only have elements whose values will be greater than 2.7 and less than 3.1. Now we are done with looking up the lower end of the range, we proceed with the higher end. We go to the right child of 3.1, it's 3.6, so we can't proceed with its right sub-tree. As a result, we go to its left sub-tree, which is only 3.2, and that is our higher end. The white nodes in the graph represent nodes our algorithm never visits. Light-green nodes are nodes we visit but we don't include in our result because they don't lie in the range. Finally, nodes in aqua color are the visited nodes that are in the range.

1.1.1 SPACE AND TIME ANALYSIS OF BST

Because every single element in the database gets represented as a node in the balanced binary search tree, our algorithm has space requirement of O(n).

On the other hand, the time needed to construct the tree – and any balanced BST - is going to be O(n log(n)). The time needed to run the query is going to be a function of the number of reported items and (log n) since we go around the reported items in a tree of height (log n). So the query time is O(k + log n), where k is the number of reported elements.

2. SOLVING 2-D RANGE SEARCHING QUERIES

We have seen, in the previous section, how to store a given database into a data structure to answer certain queries with a single element – which forms 1-D range searching. In this section, we will look at data structures used to answer queries with two elements, which makes the 2-D range searching.

Database queries that tackle two elements, total number of credit hours and the cumulative GPA, for example, cannot use the 1-D binary search tree we went over in the previous section, simply because we need to come up with a data structure that organizes points with respect to both query elements.

2.1 KD-TREES

KD-trees is a simple and convenient algorithm for arranging points with respect to two distinguished properties into a single tree, which makes searching much simpler. It, however, is built on the assumption that no two points share the same property. In simple words, if we plot these points on a graph with two axes x and y – representing the two properties -, then no two points have the same x or y coordinates. When dealing with databases, this limitation is a bit un-realistic. As an example, in a students' record database, having more than one student with 75 of completed credit hours is expected.

The KD-tree algorithm works as follows: We start with a set of points, each has its x and y coordinates. We first draw a vertical line L_1 that splits our set of points into two sets. The cutting line passes through a point we choose, and the number of points in every set is preferred to be equal as much as possible. In the next step, each set gets split into two sets with equal number of points, using a horizontal line this time, L_2 and L_3. After that, we split the four sets using vertical lines, and then horizontal lines, and so on until all points get passed through.

Now having covered all points, we can start building the KD-tree. In the first step, there was only a single line we drew, L_1 – so that line gets represented in a node that will be the root of the tree. In the second step we drew two lines, L_2 and L_3 – and they will be the children of the root node. We continue the same way with lines that were drawn in the step, get put in the same level of the tree. As you can see, odd levels will have vertical lines and even levels will have horizontal ones.

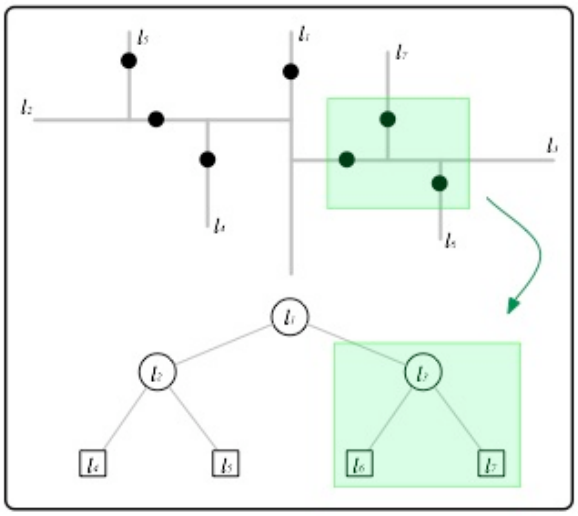

Figure.3: Constructing a KD-Tree

So let us assume that after building the KD-tree, we were given the query shown in green box in Figure 3. Since the query asks for points that are located to the right of L_1 , the answer for the query will be located in the right sub-tree of the root L_1 .

2.1.1 SPACE AND TIME ANALYSIS OF KD-TREES

Every point gets a line passing through it, which means every point will be represented with a node in the tree. So the space is going to be O(n). Constructing the tree, as expected, is going to be of order O(n log(n)).

To answer the query using KD-tree, it will take us time in the order of O(sqrt(n) + k), where k is the output.

2.2 RANGE TREES

"Range trees" is another data structure used for 2-D range searching. It basically works as follows: the first element of the query is considered at this time, and for that a binary search tree gets implemented exactly the same way it is done for the 1-D binary search tree algorithm. Now we have a binary search tree for the first element of the query, so when we are given a query with two elements, we start doing our search using only the first element. We come up with the sub-tree that has the nodes in the range of the first element of the query, rooted by V_{split}. From this point, we implement the associated tree, T_{assoc} , which holds the properties of the second element.

Let's use our original example here, the GPA and HOURS:

SQL> **SELECT** ID,NAME **FROM** STUDENTS **WHERE** HOURS **BETWEEN** 75
AND 90 **AND** GPA **BETWEEN** 2.5 **AND** 3.0

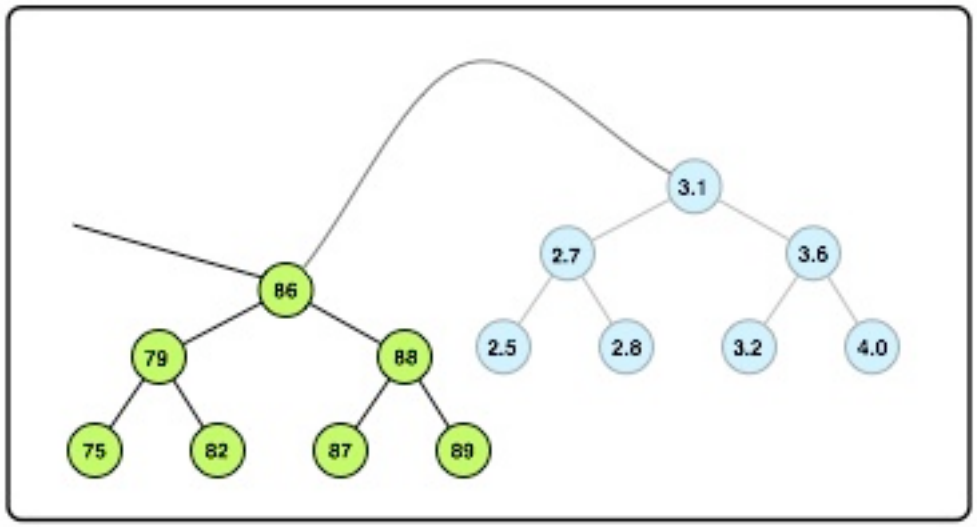

Figure.4: associating range search trees

To solve this query using Range Trees, a binary search tree that holds the values of all students' HOURS will be implemented first. A search will be applied to it exactly the same way we did our 1-D search in section 1, and V_{split} will be identified. Now we have a sub-tree under which we have all the students who has completed hours between 75 and 90. The next step will be building a second binary search tree that is associated with this tree, and it will hold values of GPA's associated with every node in our current sub-tree. After that all we need to do is to report the points whose properties match the elements of the query.

2.2.1 SPACE AND TIME ANALYSIS OF RANGE TREES

As you have noticed, the nodes that are in the sub-tree of V_{split} will be represented "again" in the other, associated tree. That means the storage will be more than O(n) but less than O(2n) since not necessarily all nodes will be there on the associated tree. The space requirement for a Range Tree is O(n log(n)). The time needed to construct the tree is also O(n log(n)).

The time it takes to answer a query using Range Tree as a data structure is going to be the summation of multiple O(log(n)), which turns out to be in the order of $O(\log^2(n) + k)$, where k is the number of reported points.

3. SOLVING RANGE SEARCHING QUERIES IN HIGHER DIMENSION

Let's look at this query:

SQL> **SELECT** ID,NAME **FROM** STUDENTS **WHERE** HOURS **BETWEEN** 75
AND 90 **AND** GPA **BETWEEN** 2.5 **AND** 3.0 **AND** MAJOR **BETWEEN** 23
AND 25

Let's assume that the values 23, 24, and 25 for the field MAJOR represent Computer Science, Computer Engineering, and Systems Engineering. So our query is looking for CS, COE and SE students who have completed between 75 and 90 credit hours, and their GPA's are between 2.5 and 3.0. This query is obviously composed of three elements, making it a range searching problem in 3-D.

3.1 APPLYING RANGE TREES IN 3-D AND HIGHER

Let's think about using range trees as a data structure for solving this 3-D query. We can first implement a binary search tree for the first element of the query, the HOURS, and narrow it down to the sub-tree that has our needed

range. From there we can come up with the associated tree that represents the second element, the GPA, and we may also need to narrow it down to the needed ranged. Finally, we can also come up with the last associated tree for the last element, the MAJOR, and also narrow it down to match the range our query is looking for.

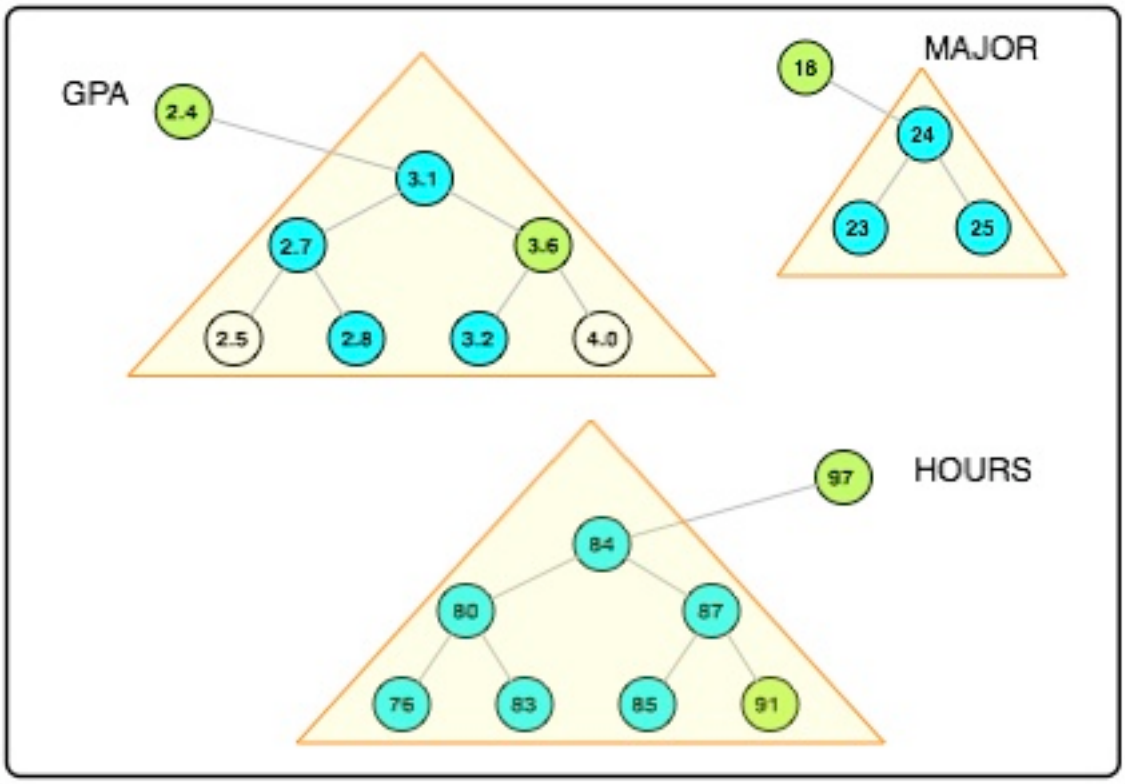

Figure.5: Using range trees in 3-D queries

Figure.5 shows the three range trees used for solving this example. So starting with the lower tree, which shows the HOURS values, we extract our sub-tree that has the range of hours we look for. Then we associate it with the GPA tree, and extract the values within the range of GPA we want. In the last step, we associate the GPA sub-tree we extracted with the MAJOR sub-tree. At the end, as you can see from the figure, we will have 3 students whose properties match the properties asked for by the query.

As a result, we could come to the conclusion that the Range Trees data structure is very much applicable for solving queries of 3-D and higher.

3.1.2 SPACE AND TIME ANALYSIS OF RANGE TREES IN 3-D AND HIGHER

The intuition is the same for range trees in higher dimension as it is in 2-D. Range trees in 3-D and higher dimensions require $O(n \log^{n-1}(n))$ storage. They can as well be constructed in time proportional to $O(n \log^{n-1}(n))$, and the running time required by a query is $O(\log^{d} n + k)$, where k is the number of points reported.

CONCLUSION

Database queries form a very interesting range searching problem. In this short paper, we have gone through the intuition of looking at these queries as 1-D, 2-D and higher dimensional graphs. This technique has simplified the process of narrowing down the input set of items, to the point at which we get the right answer for the given query.

References:

M. De Berg. Computation Geometry. Ch. 5. Berlin: Springer-Verlag, 1998

Pankaj K. Agarwal. Range Searching .*CRC Handbook of Discrete and Computational Geometry*, CRC Press, New York, 2004