Data Structures for Range Searching

A Report on the 1979 Paper by J.L.Bentley and J.H.Friedman

By:
Team No. 74
V.Gautham - 2014B4A7637P
Vikram Nitin - 2015A7PS042P
Abhay Koushik - 2015A7PS056P

Submitted in Partial Fulfillment of:
CS F211 - Data Structures and Algorithms
Second Semester 2016-17

Abstract

This report describes several data structures and algorithms used for the problem of range searching. Given a collection of records, where each record can have multiple attributes, the task is to find those records for which each of the attributes falls within specified bounds. These bounds can be specific to each attribute. The main algorithms/structures presented are Sequential Scanning, Projection, Cells, k-d Trees, Range Trees and k-Ranges. The paper describes the theory and implementation details of these various approaches and compares their performance based on three metrics: the preprocessing, storage and query cost (all asymptotic).

1 Preliminaries

We present a discussion of the six different types of algorithms or structures. The meanings of some common notation used are as follows :

- N - number of records
- k - number of attributes
- F - number of records found as the result of a range-query

For N records with k attributes each,

- $P(N, k)$ - Time Complexity of Preprocessing
- $S(N, k)$ - Space Complexity of Storage
- $Q(N, k)$ - Time Complexity of Querying

2 Descriptions

2.1 Sequential Scanning

- **Data Structure**: The different records are stored as rows of a matrix, while the columns represent their attributes.
- **Query Algorithm**: The records are processed one by one, and each attribute is scanned to check if it falls within the bounds specified for that attribute. If every attribute of a record falls inside these bounds, it is added to the list of matches.
- **Complexity**:
  $P(N, k) = O(Nk)$ since k attributes of N records have to be converted to the matrix representation.
  $S(N, k) = O(Nk)$ since there are N records with k attributes each, giving a total of N times k values.
  $Q(N, k) = O(Nk)$ since in the worst case each of the k attributes will have to be scanned for the N records.

2.2 Projection

- **Data Structure**: k lists of pointers to records are stored. The nth list is sorted by the nth attribute.
- **Query Algorithm**: The records are processed one by one, and each attribute is scanned to check if it falls within the bounds specified for that attribute. If every attribute of a record falls inside these bounds, it is added to the list of matches.
- **Complexity**: $P(N, k) = O(kNlogN)$ since there are k lists and each can be sorted in $O(NlogN)$ time.
  $S(N, k) = O(kN)$ since the k lists dont store entire records, just pointers. Only N values need to be stored per list, in addition to the N records present in memory.
  $Q(N, k) = O(N^{1-1/k})$ If all ranges are of similar width, we can form the recurrence relation $Q(N) = 2^{k-1} * Q(N/2^k) + 1$. Applying the Master Theorem to this yields the result.

2.3 Cells

- **Data Structure**: A cell is a pointer to a collection of records, which lie within a region in k-dimensional space. For example, for a 2-D case, a cell may store all records whose x values lie between 5 and 10, and whose y values lie between 0 and 5. A directory is a collection of these cells. If the distribution of values within each attribute is uniform, a k-dimensional grid for a directory is feasible. Otherwise, it makes sense to store only those cells which point to a non-zero number of records.

- **Querying Algorithm**: When a range query is posed, it has to be translated to a set of candidate cells to look within. This is done by comparing a cells range with the query range and checking for overlaps. Then all the points within candidate cells are analysed, attribute-by-attribute, to check for a match.

- **Complexity**: $P(N,k) = O(Nk)$ because k attributes of N records have to be checked in order to group the records into their respective cells.
  $S(N,k) = O(Nk)$. Storing the records in memory will take $O(Nk)$ since there are N records with k attributes each. The directory size is usually far smaller than N, so storing the pointers of the directory takes only $O(N)$ which is less than $O(Nk)$.
  $Q(N,k)$: The choice of the shape of the cells makes a significant difference to the efficiency of the querying algorithm. If the ranges in the query have similar shape, then the optimal shape of the cell is the same as the ranges in the query.
  In this case, along just one dimension, the range will overlap with 2 cells (unless the range is exactly equal to a cell range, in which case it will overlap with only 1 cell). For the k dimensions together, there will thus be 2^k candidate cells on average. Each of these will have to be scanned for matches, giving an average complexity that is $O(F * 2^k)$.

2.4 k-d trees

- **Data Structure**: A non-leaf node in $k-d$ tree divides the space into two parts, known as half-spaces. Points to left of this space are shown by the left subtree of that node and points to right of the space are shown by the right subtree.
  - Discriminator value: at every level, this is one of the k attributes from one of the records.
  - Inserting into the right and left subtrees happen recursively until there are few (six or less) nodes in the set which are stored as a linked list.
  - Records are stored in the leaves of this tree and not in the internal nodes.

- **Querying Algorithm**: The j^{th}-range of the query is compared to the j^{th}-discriminator of (i.e. j^{th} attribute of the record denoted by) the current node.If the query range is totally above or below that value search the right subtree or left subtree of that node respectively. Otherwise, search the range of both children recursively by stack implementations.

- **Complexity**: $P(N,k) = O(NlogN)$, since $T(N) = N + 2T(N/2)$.
  $S(N,k) = O(Nk)$, as a $k-d$-tree for N points is a Binary Search Tree (BST) of N leaves with k attributes each.
  $Q(N,k) = O(N^{1-1/k} + F)$.
  After induction and generalization:
  We have the recurrence: $Q(N) = 2^{k-1} * Q(N/2^k) + 1$ which solves to $Q(n) = O(N^{1-1/k})$.
  $Q(N,k) = O(logN + F)$, for small answers if the query range is almost cubical.

$Q(N,k) = O(F)$ for large answers, as large fraction of the file satisfies the query and its thus linear with respect to F.

2.5 Range Trees

- **Data Structure**: The records of the file are represented as nodes of a Binary Search Tree (BST), with one attribute used as the discriminator value.
  A range tree is defined recursively: a range tree of dimension k is constructed from a range tree of dimension $k-1$ and an unsorted k^{th} attribute. Every node in a range tree of dimension k contains a range tree of dimension $k-1$. A range tree of dimension 1 is equivalent to a sorted array.

- **Querying Algorithm**:For a given range in k dimensions, one traverses recursively along the range tree of dimension k, using the discriminator value to find the required range of the k^{th} attribute. Upon reaching the appropriate node, the range tree of dimension $k-1$ present at the node is queried for the required range, and so on.

- **Complexity**: $P(N,k) = O(Nlog^{k-1}N)$
  $S(N,k) = O(Nlog^{k-1}N)$. Each of the N records will occupy one level in the $k-1$ BSTs and the remaining attribute is part of a sorted array.
  $Q(N,k) = O(log^kN+F)$ because we are traversing $k-1$ BSTs one after the other, performing a binary search, and finally printing all the valid points.

2.5.1 k-ranges

A k-range is a structure developed from range trees specifically for the purpose of range-searching. Its implementation is complicated, and it is primarily of theoretical interest.
Introduced as an "efficient worst-case structure" by J. Bentley in 1980, it involves sets of lists of points sorted by different coordinates, with additional dimensions added by recursion.
There are two types of k-ranges:
Overlapping k-ranges :

- $P(N,k) = O(N^{1+\epsilon})$
- $S(N,k) = O(N^{1+\epsilon})$
- $Q(N,k) = O(logN + F)$

Overlapping k-ranges can be made to have this performance for any $\epsilon > 0$.
Non-Overlapping k-ranges:

- $P(N,k) = O(NlogN)$
- $S(N,k) = O(N)$
- $Q(N,k) = O(N)$

2.6 Quad Trees

- **Data Structure**: Primarily described for k=2, but can be extended to any value. Each node contains a discriminator value (which need not correspond to record attributes) and points to a number of records (leaf) or to other nodes (interior) . If a leaf node points to more than

a minimum number of records, it is converted to an interior node, and its children point to the records instead. If the records have k attributes, each interior node of the tree will have 2^k children, and each child corresponding to changes in one or more of the k attributes.

- **Querying Algorithm**: Given a range query of k dimensions, the discriminator of the node is compared with the range, and depending on which of the k attributes differ, the corresponding *children* are queried recursively to obtain the required points.

- **Complexity**: Here, h represents the height of the Quad Tree.
  $P(N, h, k) = O(N)$ since in the worst-case the depth of the tree formed is N.
  $S(N, h, k) = O(Nk)$ since the worst-case number of nodes of the tree is $N+N/2^k+N/2^{2k}+ = O(N * 2^k/(2^k - 1))$ which is less than $O(Nk)$ (storing N records) for $k > 2$.
  $Q(N, h, k) = O(N+h)$ since we have to check every record (N) and perhaps all the way down to the bottom of the tree (h).

3 Critique

In this section we shall discuss the merits and demerits of each of these data structures, and which of them is most suitable for a given scenario. Also, we shall talk about overcoming the inherent limitations of static data structures by dynamization.

3.1 Algorithm Critiques

Method	Advantages	Disadvantages	Miscellaneous
Sequential Scan	Simple to implement If $N > k$ or F is large, it is competitive	Greater query cost compared to other approaches	Can handle batched queries effectively.
Projection	Superior query cost compared to a sequential scan.	Ineffective unless a query range attribute cuts down the search space drastically.	
Cells	When the sizes of query ranges are approximately the same and known, cell sizes can be chosen for good performance.	When query ranges vary in shape, choosing an optimal cell shape is difficult and performance suffers.	A variation of cells, recursive cells, can yield better query costs.
$k-d$ Trees	Most effective for range searching when the query bounds and sizes vary. Irrelevant portions of the space are quickly pruned.	Efficiency goes down for larger dimensions ($k > 20$).	Narrow search space by half, irrespective of attribute chosen. Thus performs one of the fastest range searches.

Range Trees	Best worst-case time complexity for querying among all the algorithms described so far.	Has the highest preprocessing and storage costs in all the algorithms described so far, useful only for small k.	Narrow down search space with respect to one attribute at a time.
Quad Trees	Queries are efficient since the tree needs to be traversed only once.	Can lead to several levels of partitioning if points are densely clustered.	

3.2 Dynamization

The data structures weve described so far are are static, meaning that all records are processed once initially, and then insertion and deletion is not natively supported. Dynamization is the process of converting these static data structures into dynamic structures which support insertion and deletion.

- **Insertion**:It is normally achieved by the splitting the available structure into smaller static data structures or blocks. Each insertion query is processed separately and then these blocks are merged to obtain the result. Instead of building a new structure from the old one, the work is spread under a number of insertions and thus the old structures are still valid for query answering.
- **Deletion**: The essential idea behind deletion is to transfer all the points to be retained into a new static structure, and use this structure as the output of deletion. This involves the creation of a new Ghost structure to buffer deletions. A Stay structure is constructed during the processing of the deletion queries. When the Stay is complete, the Ghost is discarded.

4 References

1. Jon Louis Bentley and Jerome H. Friedman. 1979. Data Structures for Range Searching. *ACM Comput. Surv. 11, 4 (December 1979).*

2. Friedman, J H., Baskett, F., and Shustek, L. J "An algorithm for finding nearest neighbors," *IEEE Trans Compute. C-24, 10 (Oct. 1975), 1000-1006.*

3. Knuth, D.E. The art of computer programming, vol. 3- sorting and searching, *Addison-Wesley, Reading, Mass., 1973*

4. Lee, D. T.; Wong, C. K. (1977). "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees". *Acta Informatica. 9. doi:10.1007/BF00263763.*

5. Lueker, G. "A data structure for orthogonal range queries," *Imp Proc 19th Symp Foundations of Computer Science, IEEE, Oct. 1978, pp. 28-34*

6. Lee, D. T., and Tong, C.K. "Quintary trees : a file structure for multidimensional database systems," to appear in *ACM Trans. Database Syst.*

7. Bentley, J. L, and Maurer, H. A. "Efficient worst-case data structures for range searching," in *Acta Inf.*

8. Overmars, M.H.; Leeuwen, J.V. (1980). Dynamizations of Decomposable Searching Problems Yielding Good Worst-case Bounds.