Data Structures for Dynamic Queries: An Analytical and Experimental Evaluation

Vinit Jain and Ben Shneiderman+
Human-Computer Interaction Laboratory &
Department of Computer Science
Institute for Systems Research+
University of Maryland
College Park, MD 20742

Index Terms : Dynamic queries, data structures, main memory database, search overhead quantification, storage overhead quantification.

Abstract

Dynamic Queries is a querying technique for doing range search on multi-key data sets. It is a direct manipulation mechanism where the query is formulated using graphical widgets and the results are displayed graphically in real time.

This paper evaluates four data structures, the multilist, the grid file, k-d tree and the quad tree used to organize data in high speed storage for dynamic queries. The effect of factors like size, distribution and dimensionality of data on the storage overhead and the speed of search is explored. A way of estimating the storage and the search overheads using analytical models is presented. These models are verified to be correct by empirical data.

Results indicate that multilists are suitable for small (few thousand points) data sets irrespective of the data distribution. For large data sets the grid files are excellent for uniformly distributed data, and trees are good for skewed data distributions. There was no significant difference in performance between the tree structures.

1 Introduction

Most users of database systems must learn a querying language which they use to select and retrieve information. A query language is a special purpose language for constructing queries to retrieve information from a database of information stored in the computer [21].

1.1 Dynamic Queries

Dynamic queries [1] is a novel way to explore information. This mechanism is well suited for multi-key data sets where the results of the search fit completely on a single screen. Figure 1 shows an application of dynamic queries in searching a real estate database. The query is

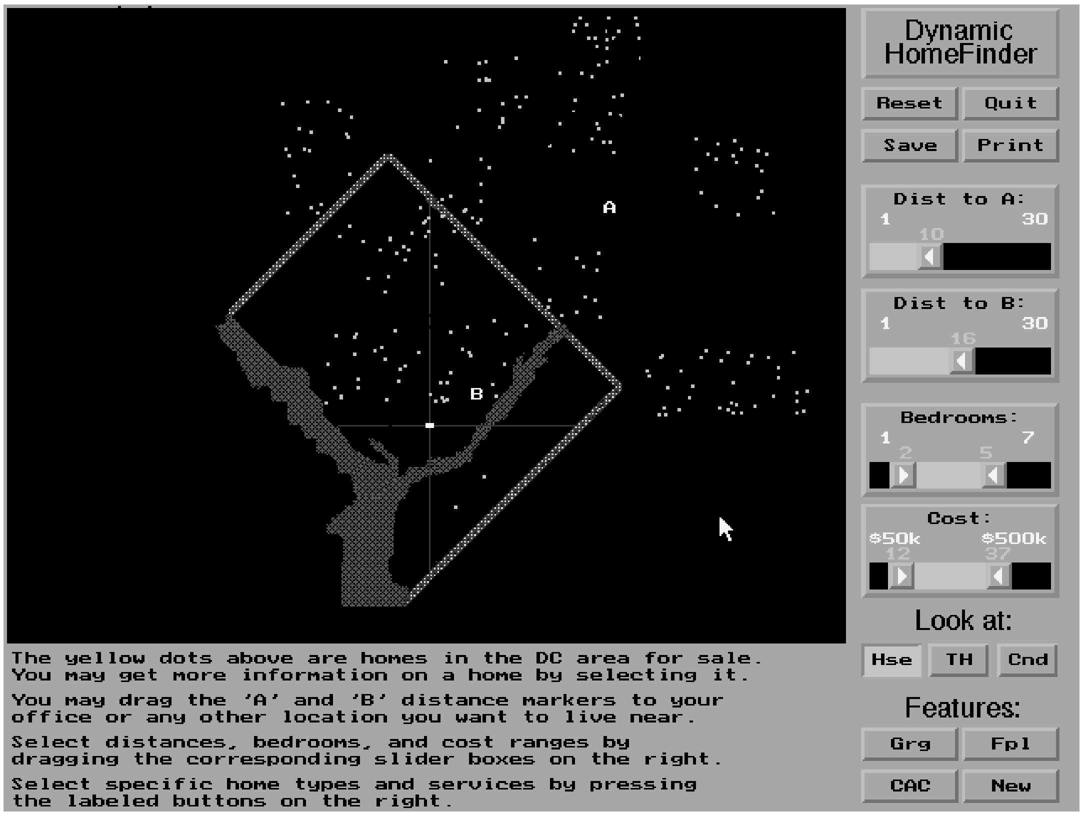

Figure 1: The Dynamic Home Finder

formulated using widgets such as buttons and sliders, one widget being used for every key. A study [26] was conducted which compared dynamic queries (DQ) to a natural language system known as "Q & A" and a traditional paper listing sorted by several fields. There was statistically significant difference in the performance of the DQ interface compared to the other two interfaces. The DQ interface enabled users to perform faster and was rated higher than the other two in the terms of user satisfaction. The DQ interface was very useful in spotting trends and exception to trends as compared to the other two interfaces.

One of the important features of a DQ interface is the immediate display of the results of the query. In fact, users should be able to perform tens of queries in a span of a few seconds so that the mechanism remains dynamic. Using larger data sets slows down the mechanism so that there is a noticeable time interval between the movement of sliders and the display of results.

There are two main facets to the issue of speed. Since the mechanism of dynamic query is a Graphical User Interface (GUI) the speed depends considerably on the graphical capabilities of the machine on which it runs. The effect of this factor largely depends on how the results are displayed. The other factor on which the speed is dependent is the time it takes to compute the results of queries. Even though the search time depends to a great extent on the hardware of the machine used, query computation can be optimized to a great extent

Data Structure	Storage Cost $S(N,k)$	Search Cost $Q(N,k)$
Sequential List	$O(Nk)$	$O(Nk)$
Multilist	$O(Nk)$	$O(Nk)$
Cells	$O(Nk)$	$O(2^k F)$
k-d Tree	$O(Nk)$	$O(N^{1-1/k} + F)$
Quad Tree	$O(Nk)$	$O({N_1}^{1-1/k} + F)$
Range Tree	$O(Nlog^{k-1}N)$	$O(log^k N + F)$
k-Ranges	$O(N^{2k-1})$	$O(klogN + F)$

Figure 2: Storage and Search time overheads for various data structures

by using suitable data structures.

In this paper data structures for main memory are examined. We assumed that the data sets remains frozen i.e. there are no insertions, deletions or updates. The time taken to load the data into memory i.e. the preprocessing time is ignored as it is done only once. Only simple rectangular queries are considered i.e. queries will be a simple conjunct of the ranges specified by the sliders.

1.2 Multi Attribute Range Search Methods

The problem of range search on multi attribute data sets can be defined as:
For a given multi-attribute data set, and a query which specifies a range for each attribute, find all records whose attributes lie in the given ranges.
The cost functions of various data structures are provided where N is the number of records, k is the number of attributes and F is the number of records found. For the quad tree N_1 is the number of nodes in the tree. Further details about these data structures can be obtained from the references.

- $S(N,k)$ is the cost of storage required by the data structure.
- $Q(N,k)$ is the search time or query cost.

Figure 2 shows cost functions for structures that are suitable for rectangular queries. Many other complex structures exist, but they are mainly of theoretical interest only because of their high memory overhead. It can be seen that range trees and k-ranges have relatively high storage overheads and are thus eliminated from consideration.

2 Data Structures

We assume the following characteristics of dynamic queries. The parameters of search are specified using sliders with one slider being used for each dimension. There are a limited number of positions the dragbox of a slider can take. This results in the ranges getting broken into discrete intervals. If every slider is assumed to break a range into G intervals, and if the data set has D dimensions then the search space can be split into G^D buckets. A bucket is the smallest unit of search and it is not possible to differentiate between points in a bucket. During search all or none of the data points of a bucket get included in the solution set. It may happen that in certain data distributions some buckets are empty. Unlike the case in bucket methods for storage on disks, there is no limit on the number of points in a bucket when main memory is used.

Four data structures are described in this section. The data points are stored in a simple array. It is assumed that points belonging to the same bucket will be stored consecutively. The data structures will be used to maintain an index on the array so that the search time is reduced. To maintain these indices, memory overhead is incurred which needs to be kept low. These structures can be classified in two categories i.e. bucket and non-bucket methods. In bucket methods an index is maintained on buckets and in non-bucket methods it is maintained on data points. The linked array which is a non-bucket method is described first. Later the bucket methods are described.

2.1 Linked Array

Figure 3 shows a part of the linked array when the data is two dimensional (i.e. $D = 2$). Also shown in the figure is a data array. The data array is an array which holds the data points. With every interval in the slider range is associated a linked list. Every point in the data set will lie in one and only one linked list of every slider. Also with every record is associated a flag (not shown in the figure). This flag keeps count of the number of fields of the record which satisfy the region of interest. When this count becomes equal to the number of dimensions then the record is displayed.

2.2 Grid Array

Figure 4 shows a part of the grid array used to index the data array for the two dimensional case. This is a bucket method and a bucket is essentially a pair of index numbers or pointers which point to the first and last record in the data array that belong to the bucket. These buckets form a part of the D dimensional search space. Therefore, to index them a D dimensional array is used.

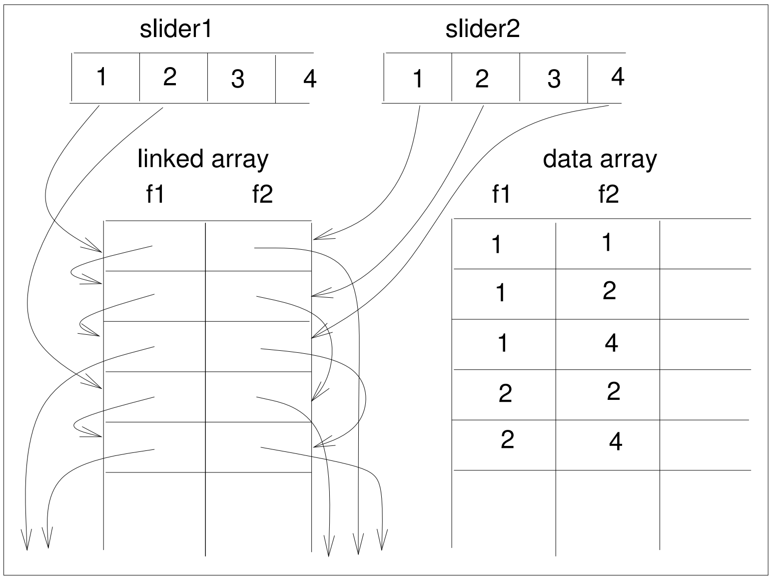

Figure 3: Linked Array used to index the Data Array

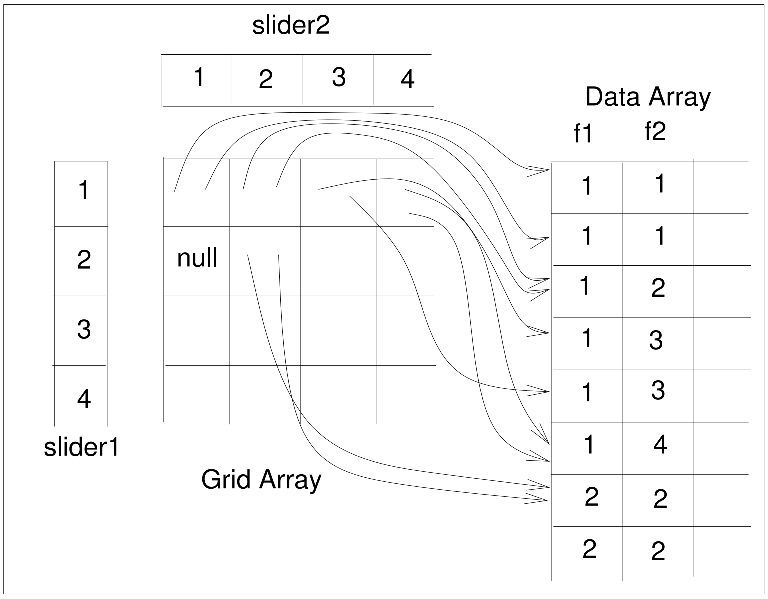

Figure 4: Grid Array used to index the Data Array

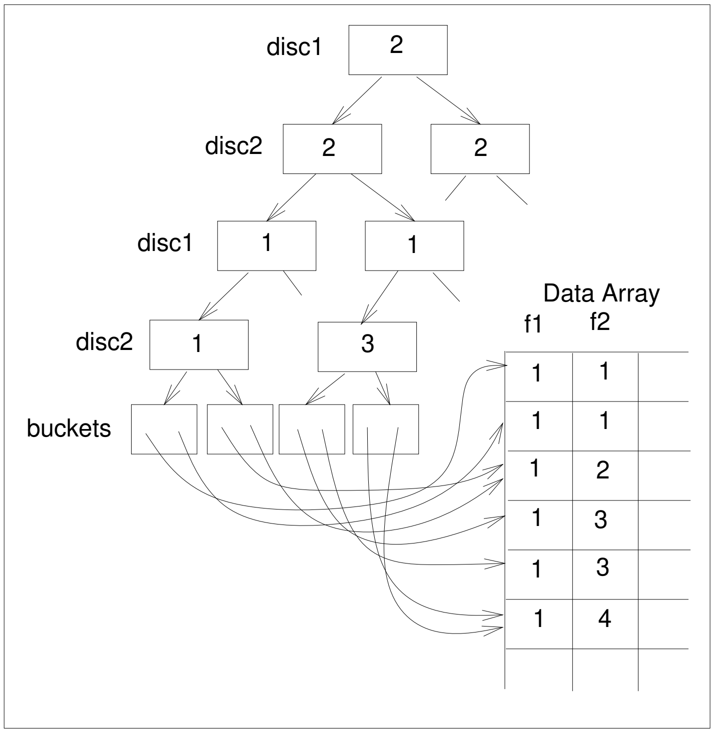

Figure 5: k-d Tree used to index the Data Array

2.3 k-d Tree

Figure 5 shows a part of a k-d tree and the data array associated with it. In k-d trees, the concept of the buckets is again the same as the grid array. For the k-d tree being used not all nodes at the same level in the tree have the same discriminator key. Nor are all the leaves at the same level. Therefore, in each node, besides the discriminator key value, the type of the discriminator key and a flag (not shown in the figure) which indicates the type of children (node or leaf) is also stored. This is done so that if the number of non-empty buckets is few, then the tree size can be reduced. In some cases, it is possible that after optimization some leaves may move so that they are no longer at the level they were in the non-optimized tree. In such cases an additional check needs to be done to ensure that the bucket reached is valid. A flag (not shown in the figure) is kept in the leaves which indicates whether this check needs to done.

2.4 Quad Tree

Just as k-d trees could be used to index the buckets, quad trees can also be used to index the buckets. Figure 6 shows how a quad tree can be used to maintain an index. The nodes of the quad tree depend on the dimensionality of the data set. Each node has D discriminator keys and 2^D pointers for the children. Just as in the case of a k-d tree, the children of a non-leaf node may be a mix of leaves and (non-leaf) nodes which is determined by a flag.

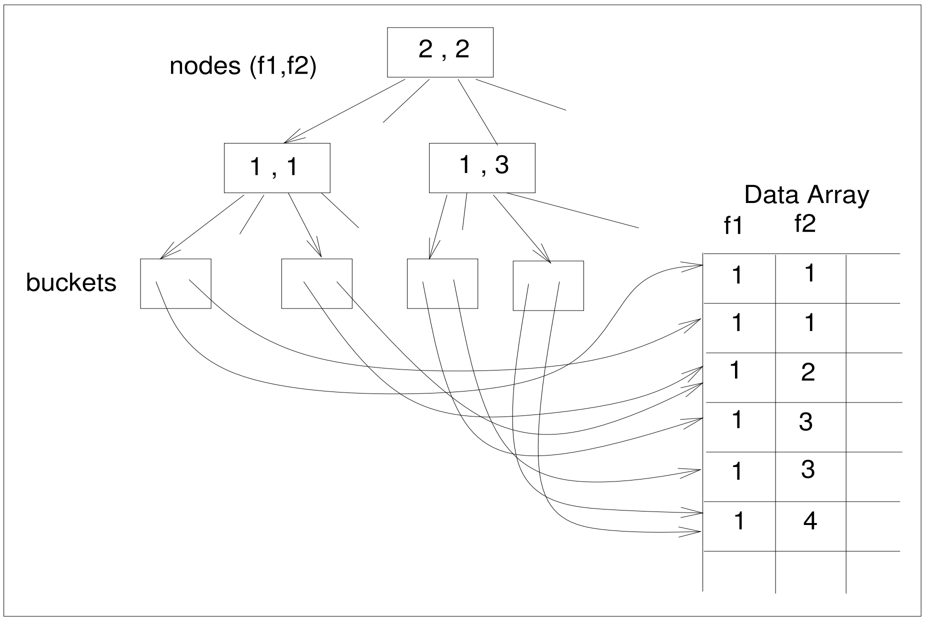

Figure 6: Quad Tree used to index the Data Array

One bit is required to maintain this information about each child. It is assumed that D is at most 5, hence the number of children is at most 32 and one word (32 bits or 4 bytes) suffices to store leaf/non-leaf information for all the children of a node. In some cases, after optimization, some leaves may move, (as in the k-d tree) and a check may be required to ensure correctness. A flag is maintained in the leaf nodes for this purpose.

3 Analytical Model

We analyze the storage and search overheads of data structures used for dynamic queries in this section. Storage overhead refers to the additional storage requirements for the data structure used. Search overhead is the number of operations required to compute the query result when the slider is moved. These two metrics will be used throughout this section to evaluate the data structures for dynamic queries. Search overhead will depend on how large the space being searched is. At any given moment the sliders define something called the *region of interest*, which is the portion of the search space being displayed. Every movement of a slider is a query which increases or decreases the region of interest. It is assumed that at any time only one dragbox of a slider will move in steps of one discrete interval. As a result of this move, points have to be removed or added to the display. In the case of the search overhead, it is assumed that the case where the region of interest is increasing will apply as a general case. For the worst case the increment in the region of interest should be the greatest. This happens when $D-1$ sliders have their left and right dragboxes in the extreme left and right respectively. In this case with every move of the D^{th} slider G^{D-1} buckets are added to the region of interest.

The effectiveness of a method will be studied with respect to factors such as

- The distribution of the data points in the search space.
- Dimensionality of the data set D
- Size of the data set N

3.1 Comparing Bucket Methods

In this section a comparison of bucket methods is presented. Only the number of non-empty buckets will be considered in the analysis. The number of tuples does not effect the performance in any way. As mentioned earlier, two performance metrics used in the analysis are the storage overhead for the index on buckets and the worst case search time.

The following symbols will be used:

N	:	Number of points in the data set.
D	:	Number of dimensions.
G	:	Number of intervals in each slider range.
No_i	:	Number of nodes in the tree (no. of dimensions is i).
B_i	:	Number of non-empty buckets (leaves , no. of dimensions is i).
Nv_i	:	Number of (non-leaf) nodes visited in the worst-case (no. of dimensions is i).
Bv_i	:	Number of leaves visited (non-empty buckets for the grid array) in the worst case (no. of dimensions is i).

The storage overhead is the cost of maintaining an index on the data points. In computing the storage overheads, it is assumed that each integer is 4 bytes, each character is 1 byte and each pointer is 4 bytes. If the number of dimensions is i, then the following holds true:

Grid Array: In grid arrays, 2 integer indices (first and last) are maintained for each bucket, irrespective of data distribution. So, 8 bytes are required for each bucket. Therefore,
Total storage overhead $= 8G^i$ bytes .

k-d Tree: In k-d trees, each non-leaf node has an integer discriminator key (4 bytes), a character discriminator key type (1 byte), a character flag for type of children (1 byte) and two pointers for left and right children (4 bytes each), resulting in a total of 14 bytes. Each leaf (non-empty bucket) has 2 integer indices (4 bytes each) and a character flag (1 byte), resulting in a total of 9 bytes. So,
Total storage overhead $= 14No_i + 9B_i$ bytes.

Quad Tree: In quad trees, each non-leaf node has 2^i pointers for children of the node (4 bytes each), i integer discriminator keys (4 bytes each) and one flag for maintaining types of children (4 bytes), resulting in a total of $4(2^i+i+1)$ bytes. Each leaf (non-empty bucket) requires 9 bytes as in the case of k-d trees. So,
Total storage overhead $= 4(2^i + i + 1)No_i + 9B_i$ bytes.

The following assumptions have been made in calculating the search time overhead. i is used to indicate that the terms are for i dimensional case.

- For every non-empty bucket visited it is assumed that one operation is done to report that the bucket is non-empty.
- For the grid array one operation is required to visit a bucket and if it is non-empty then another operation is performed.
- For the k-d tree every non-leaf node visited has to put into the stack and then later retrieved from the stack. This requires 2 operations. In processing every node two comparisons have to be made. This makes it a total of 4 operations for every non-leaf node visited. In visiting leaves, 2 operations will always be required: 1 operation for reporting that the bucket is non-empty (as discussed earlier) and 1 operation to get to the leaf. In some cases, because of the optimizations on the tree that were discussed earlier, an additional check is required to ensure that the bucket reached is the correct one. This may require up to $2i$ operations.
- For the quad tree every non-leaf node visited has to be put into the stack and later retrieved from the stack. This requires 2 operations. $2i$ comparisons are required to determine which children of the node to search. 2 operations are required for checking the flags to determine the type of children. So, in all $2i + 4$ operations are done for every non-leaf node visited. In visiting leaves, 2 operations will always be required: 1 operation for reporting that the bucket is non-empty (as discussed earlier) and one operation to get to the leaf. As in the case of k-d trees, an additional check may be required to ensure correctness of the leaf reached which requires $2i$ operations.
- In the worst case, for both k-d tree and quad tree, it is assumed that the number of nodes visited, when the data is i dimensional is No_{i-1}. This is because in the worst case $i-1$ sliders do not restrict the search in any way. The slider restricting the search has only one of its discrete intervals to be searched for. It is like taking a slice of thickness 1 from the i dimensional search space.

3.1.1 Uniform Data Distribution

In this subsubsection, the analysis for computing storage overheads and search time is presented, for the case when the data distribution is uniform. An important factor effecting these performance metrics is the percentage of buckets which are non-empty. Two extreme cases will be considered in this subsection, when all the buckets are non-empty and when only 25% of the buckets are non-empty. The method of analysis presented here can be used to derive exact expressions for other cases, but details of those are not presented here.

Case 1 : When all buckets are non-empty
In this case, the number of non-empty buckets (B_D) is G^D. The storage overheads for each of the data structures are as follows:

Grid Array: As discussed earlier, the storage overhead for the grid array is $8G^D$ bytes, irrespective of the data distribution.

k-d Tree: For optimal k-d trees, number of non-leaf nodes is the same as the number of leaf nodes (non-empty buckets), i.e. $No_i = B_i$. Therefore, if the number of dimensions is D, $No_D = B_D$. The storage overhead for k-d trees is $14No_D + 9B_D$. Since, $No_D = B_D = G^D$,
Total storage overhead $= 23G^D$ bytes.

Quad Tree: In the quad trees, the number of nodes No_D is $G^D/(2^D - 1)$. As discussed earlier, the storage overheads for quad trees, when the number of dimensions is D is $4(2^D + D + 1)No_D + 9B_D$. Since $B_D = G^D$,
Total storage overhead $= 4(2^D + D + 1)\frac{G^D}{2^D - 1} + 9G^D$ bytes.

Next, the second performance metric, search time is considered. Search time means the time (operations) required when the exactly one slider is advanced by one region. Search time overheads in the worst case are as follows:

Grid Array: For the grid array, the number of buckets visited (Bv_D) is G^{D-1}. As discussed earlier, one operation is required for each bucket visited and one additional operation is required if the bucket is non-empty. Since all the buckets are non-empty,
Total number of operations $= 2G^{D-1}$.

k-d Tree: For the k-d tree, the number of (non-leaf) nodes visited (Nv_D) is No_{D-1}. The number of leaves (non-empty buckets) visited (Bv_D) is B_D/G. For an optimal k-d tree, the number of non-leaf nodes is the same as the number of leaf nodes, i.e. $No_i = B_i$. Therefore, $Nv_D = No_{D-1} = B_{D-1}$. Since all the buckets are full, average height of a leaf node from the maximum depth is 0. Consequently, additional checks for finding if the bucket reached is the correct one will never be required. As discussed earlier, accessing each non-leaf node requires 4 operations and accessing each leaf node requires 2 operations.
Total number of operations $= 4Nv_D + 2Bv_D = 6G^{D-1}$.

Quad Tree: For the quad tree, the number of (non-leaf) nodes visited (Nv_D) is No_{D-1}. The number of leaves (non-empty buckets) visited (Bv_D) is B_D/G. Also, for a quad tree, the number of (non-leaf) nodes is $G^D/(2^D - 1)$. Since all the buckets are full, average height of a leaf node from the maximum depth is 0. Consequently, additional checks for finding if the bucket reached is the correct one will never be required. As discussed earlier, accessing each non-leaf node requires $2D+4$ operations and accessing each leaf node requires 2 operations.
Total number of operations $= (2D + 4)\frac{G^{D-1}}{2^{D-1}-1} + 2G^{D-1}$.

Case 2 : When 25% of all buckets are non-empty
In this case, the number of non-empty buckets $B_D = \frac{G^D}{4}$
The storage overheads for data structures are:

Grid Array: For a grid array, the storage overhead is independent of data distribution. 8 bytes are required for each bucket, so
Total storage overhead $= 8G^D$ bytes.

k-d Tree: For the case of the k-d tree, it can be assumed that $No = B$ for optimal trees.

Therefore $No_D = B_D$. The storage overhead, as discussed earlier is, $14No_D + 9B_D$. Since $B_D = G^D/4$,
The storage overhead $= 23B_D = \frac{23G^D}{4}$ bytes.

Quad Tree: For quad trees, number of nodes $No_D = \frac{G^D}{2^D-1}$. Storage overhead, as discussed earlier, for the quad trees is $4(2^D + D + 1)No_D + 9B_D$. Since $B_D = G^D/4$,
Total storage overhead $= 4(2^D + D + 1)\frac{G^D}{2^D-1} + 9\frac{G^D}{4}$ bytes.

Search time overheads in the worst case are as follows:

Grid Array: For the grid array, the total number of buckets visited is G^{D-1}. One operation is required for visiting each bucket. On an average, we will expect that 25% of all the visited buckets are non-empty. One additional operation is required for each non-empty bucket visited. So,
Total number of operations $= G^{D-1} + \frac{G^{D-1}}{4} = 5\frac{G^{D-1}}{4}$.

k-d Tree: As discussed earlier, for the k-d tree, $Nv_D = No_{D-1}$. Also, the total number of non-empty buckets (leaf nodes) visited (Bv_D) is B_D/G For an optimal k-d tree, $No_i = B_i$. Therefore, $Nv_D = B_{D-1}$. Since only 1/4 of all the buckets are non-empty, average height of a leaf node from the maximum depth of the tree is 2. Therefore, 4 additional operations are required for every bucket visited. In all, 4 operations are required for each non-leaf node visited and 6 operations are required for each bucket visited. Since $Nv_D = B_{D-1} = G^{D-1}/4$ and $Bv_D = B_D/G$,
Total number of operations $= \frac{4G^{D-1}}{4} + \frac{6G^{D-1}}{4} = 5\frac{G^{D-1}}{2}$.

Quad Tree: For the quad tree, number of non-leaf nodes visited (Nv_D) is No_{D-1}. The number of buckets visited is (Bv_D) is B_D/G. Since $No_i = G^i/(2^i-1)$, $Nv_D = G^{D-1}/(2^{D-1}-1)$. Average height of a leaf node from the maximum depth is 0 (except in the case of $D = 2$, which is ignored). $2D + 4$ operations will be required for each non-leaf node visited and 2 operations will be required for each bucket visited.
Total number of operations $= (2D + 4)\frac{G^{D-1}}{2^{D-1}-1} + \frac{G^{D-1}}{2}$.

Figures 7 and 8 show how the storage and search overheads vary as the fraction of non-empty buckets changes for uniformly distributed data. The value used were, $G = 16$ and $D = 4$. On the basis of this analysis, the grid array is a significantly better structure to use when data is uniformly distributed and most buckets are non-empty. It has a lower memory and search time overhead than both the tree structures. However as the number of empty buckets rises the difference in the memory overhead reduces and the trees get better. When comparing the search overheads of the structures for the case where most buckets are non-empty the quad tree has a lower search overhead. The dimensionality of the data only increases the differences with the differences in performance becoming greater as dimensionality rises.

3.1.2 Skewed Data Distribution

So far only the cases when the data distribution was uniform were examined. In this subsubsection, the performance of data structures is examined when the data distribution

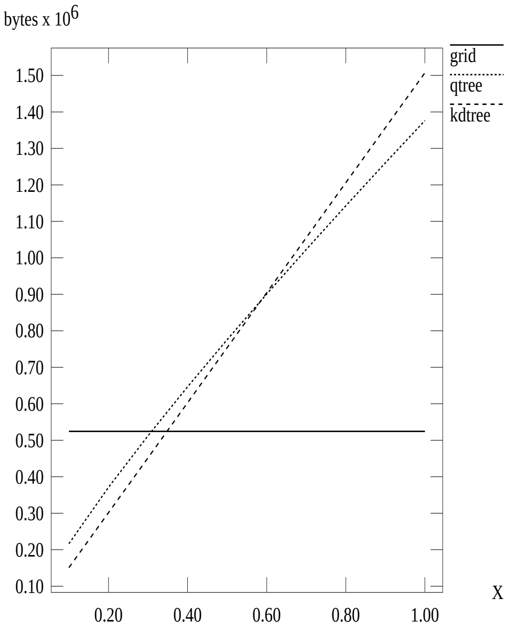

Figure 7: Memory overhead for Uniformly Distributed Data Vs. the fraction of non-empty buckets

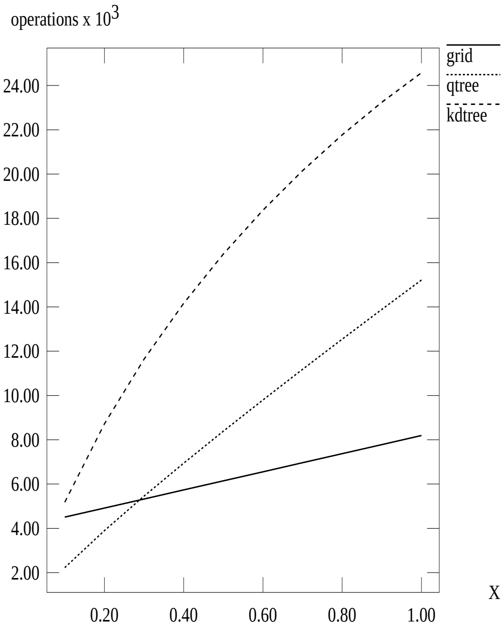

Figure 8: Search overhead for Uniformly Distributed Data Vs. the fraction of non-empty buckets

is skewed. Here two cases will be examined. First the case where all the non-empty buckets are only along the diagonal of the search space. Later the case where all the non-empty buckets are within a distance of $G/4$ from the diagonal is examined.

Case 1 : When all non-empty buckets are along the diagonal
In this case, the number of non-empty buckets B_D is G. The storage overheads for the data structures will be as follows:

Grid Array: For the grid array, 8 bytes will be required for each bucket. Total storage overhead $= 8G^D$ bytes.

k-d Tree: For optimal k-d trees, $No_i = B_i$. Therefore, $No_D = B_D = G$. Since 14 bytes are required for storing each non-leaf node and 9 bytes are required for storing each bucket, Total storage overhead $= 14No_D + 9B_D = 23G$ bytes.

Quad Tree: Since there are only G non-empty buckets, all of them along the diagonal, every node of the quad tree will have only two non-empty children. Therefore, $No_D = B_D$. Total storage overhead $= 4(2^D + D + 1)No_D + 9B_D = 4(2^D + D + 1)G + 9G$ bytes.

Search time overheads in the worst case are as follows:

Grid Array: For the grid array, number of buckets visited (Nv_D) is G^{D-1}. The number of non-empty buckets visited is $B_D/G = 1$.
Total number of operations $= G^{D-1} + 1 \approx G^{D-1}$.

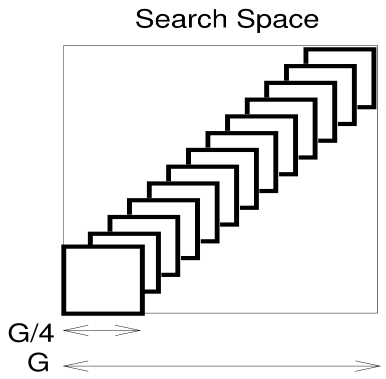

Figure 9: Skewed Distribution: All points within $G/4$ of Diagonal

k-d Tree: For the k-d tree, the number of non-leaf nodes which need to be visited (Nv_D) is equal to the height of the tree which is $log_2 G$. The number of buckets to be visited (Bv_D) is $B_D/G = 1$. Average height of a leaf node from the maximum depth is greater than D. Therefore, $2D+2$ operation will be required for the bucket visited. So,
Total number of operations $= 4log_2 G + 2D + 2$.

Quad Tree: For the k-d tree, the number of non-leaf nodes which need to be visited (Nv_D) is equal to the height of the tree which is $log_2 G$. The number of buckets to be visited (Bv_D) is $B_D/G = 1$. Average height of a leaf node from the maximum depth is 0 (no optimization). So,
Total number of operations $= (2D+4)log_2 G + 2$.

Case 2 : When all non-empty buckets are within a distance of $G/4$ of the diagonal

In this case, the number of non-empty buckets (B_D)is

$$B_D = \frac{G^D}{4} + \frac{3G}{4}((\frac{G}{4})^D - (\frac{G}{4} - 1)^D)$$

Figure 9 shows the two dimensional case for this distribution. The storage overheads for data structures are as follows:

Grid Array: 8 bytes are required for each bucket, irrespective of its being empty or not. So,
Total storage overhead $= 8G^D$ bytes.

k-d Tree: For optimal k-d trees, $No_i = B_i$. Therefore, $No_D = B_D$. Since 14 bytes are required for each non-leaf node and 9 bytes are required for each leaf node (bucket),
Total storage overhead $= 23((\frac{G}{4})^D + \frac{3G}{4}((\frac{G}{4})^D - (\frac{G}{4} - 1)^D))$ bytes.

Quad Tree: For quad trees estimating No_D for this case is a little complex. Out of the B_D leaves in the tree $b_1 = 4(G/4)^D$ leaves are set aside. Let $b_2 = B_D - b_1$. A quad tree of b_2 leaves and height $log_2 G$ will have $bf = {b_2}^{1/log_2 G}$ children for a node. Number of nodes is $\frac{b_2}{bf-1}$. The b_1 nodes that were kept aside form 4 dense subtrees each with approximately

$(G/4)^D/(2^D-1)$ nodes. Total number of nodes (No_D) is

$$No_D = \frac{b_1}{2^D-1} + \frac{b_2}{bf-1}$$

Total storage overhead $= 4(2^D + D + 1)No_D + 9B_D$ bytes.

Worst case search time overheads are as follows:

Grid Array: For the grid array, number of buckets visited is G^{D-1}. On an average, B_D/G non-empty buckets will be visited. Since one operation is required for visiting each bucket and one additional operation is required if the node is non-empty,
Total number of operations $= \frac{G^{D-1}}{4} + \frac{B_D}{G}$.

k-d Tree: For the k-d tree, Nv_D is No_{D-1} and Bv_D is B_D/G. For an optimal k-d tree, $No = B$. Therefore, $Nv_D = B_{D-1}$. Average height of a leaf node from the maximum depth is greater than D. Therefore, $2D+2$ operation will be required for the bucket visited. So,
Total number of operations $= 4B_{D-1} + (2D+2)\frac{B_D}{G}$.

Quad Tree: For the quad tree, Nv_D is No_{D-1} and Bv_D is B_D/G. No_{D-1} can be calculated as was done for computing the storage overhead earlier. Average height of a leaf node from the maximum depth is 0 (no optimization).
Total number of operations is $= (2D+4)No_{D-1} + 2\frac{B_D}{G}$.

For skewed distributions there is a significant difference between the performance of trees and the grid array with the trees being superior. This is reflected both in memory and search overheads. For the case where non-empty buckets lie only along the diagonal of the search space the difference in the trees and the grid is phenomenal. In the second case also the trees are significantly better. Amongst the trees it can be said that the k-d tree has a marginally lower memory overhead and a marginally higher search overhead than the quad tree. As in the case of uniformly distributed data higher dimensionality of data makes the differences more pronounced.

3.2 Bucket Vs Non-Bucket Methods

In the previous subsection bucket methods were compared. In bucket methods the number of points does not effect the search overhead if the number of points is sufficiently large to make most buckets non-empty. However when the number of points is small or the number of dimensions is low it may be worth trying to use the linked array. This is because for linked arrays the storage overhead is directly proportional to the number of points in the data set and the dimension of the data set.

Using the same symbols the storage and search time overheads of the linked array are discussed. In addition to the value of $G = 16$ the value of $D = 4$ has been used in the graphs comparing the two methods.

- Every tuple of the linked array has to be kept on D lists. For this D additional pointers (4 bytes) are needed. In addition a flag (1 byte) is required as discussed in section 2.1 earlier. Therefore, storage overhead is $(4D+1)N$.

- Generally each linked list associated with a slider will have N/G tuples in it. Therefore, search overhead (worst case and average case) is $\frac{N}{G}$.

The linked array was compared to the grid array for uniform data distributions. Only the grid array was chosen because it has a superior performance compared to the trees for uniform distribution. With a value of $G = 16$ and $D = 4$, it was seen that the linked array performed much better as far as the search overhead is concerned. However the storage overhead for this structure gets very high.

The linked array was compared with the tree structures for the skewed distribution. The grid array was dropped from consideration here because trees perform better under skewed data distributions. In this case the performance of the tree structures, specially the quad tree is much better both when storage overhead and search overheads are compared. One reason could be that in skewed distributions the bucket occupancy rises very steeply when compared to the uniform distributions.

The analytical models developed in this section give an insight as to how the data structures can be analyzed for cases other than the ones already discussed. However the models developed in this section need to be confirmed by empirical evidence, specially for the case of k-d trees and quad trees where the differences are not clear. The next section is devoted to the empirical results from the implementation of these models.

4 Experimental Results

The analytical models of section 3 were verified by implementing the cases discussed. The implementation was done on a dedicated SUN 4/50 with 16 MB of memory and running SunOS. The memory overhead was calculated by counting the nodes and leaves for the bucket methods, and the number of points for the linked array. Clock time in microseconds was used to measure the speed of search instead of the number of operations as in section 3. The process switching overhead was ignored as the machine had negligible load.

4.1 Comparing Bucket Methods

The analytical models of subsection 3.1 were implemented and the results are presented in this subsection. In the calculation of memory overhead, only the extra memory required to maintain the index was considered. As mentioned before in calculating the search time, the time for display of records was ignored. The value of $G = 16$ was used in implementations.

4.1.1 Uniform Data Distribution

As discussed in the subsubsection 3.1.1 there are two cases to be considered. First the case where all the buckets are non-empty is discussed. Figures 10 and 11 show the results of the memory and search time overhead respectively. For this case the grid array is significantly better than the tree structures both in terms of memory overhead and search time overhead.

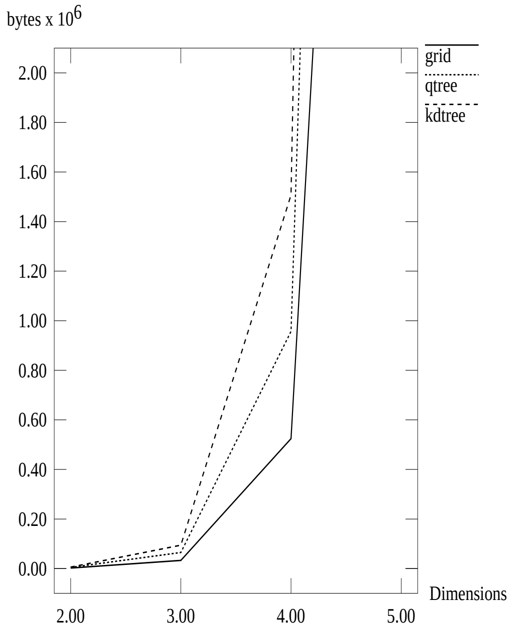

Figure 10: Memory overhead for Uniformly Distributed Data (100% buckets non-empty)

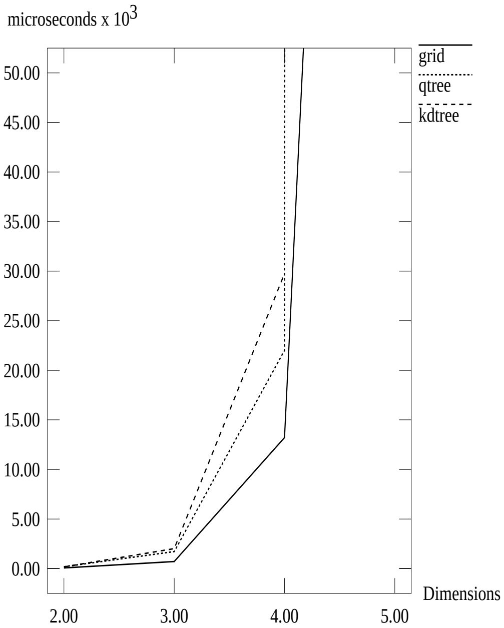

Figure 11: Search overhead for Uniformly Distributed Data (100% buckets non-empty)

In the case where 25% of the buckets are empty, figure 13 shows the grid array to be a significantly better structure as far as the search time overhead is considered. However figure 12 shows that the k-d tree is marginally better when memory overhead is considered but has a higher search overhead. All results in this subsubsecton closely match previous analytical models.

4.1.2 Skewed Data Distribution

As discussed in the subsection 3.1.2 there are two cases to be considered. First the case where all the non-empty buckets are along the diagonal is discussed. Figures 14 and 15 show the results of the memory and search time overhead respectively. For this case both the k-d tree and the quad tree give a performance far superior to the grid array both for memory overhead and search time overhead. However there is no significant difference when the performance of trees is compared with each other.

When all points lie in buckets within a distance $G/4$ of the diagonal the tree structures turn out to be excellent performers compared to the grid array. This can be seen clearly in figures 16 and 17. However the difference between the tree structures themselves in not large. It should be noted that as in the previous cases the results of the implementations do not differ from the analytical models.

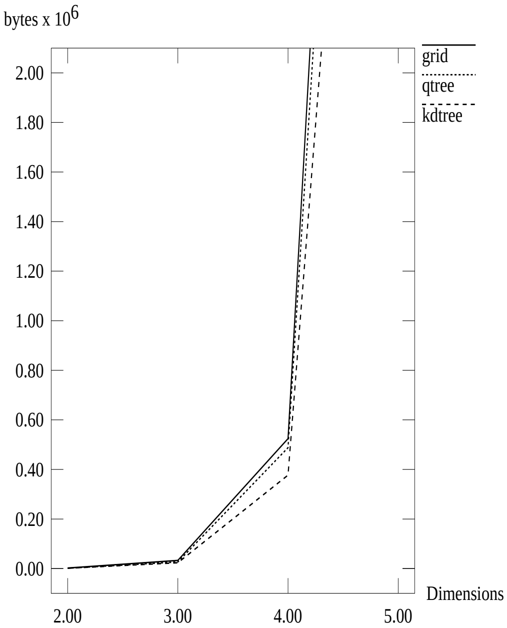

Figure 12: Memory overhead for Uniformly Distributed Data (25% buckets non-empty)

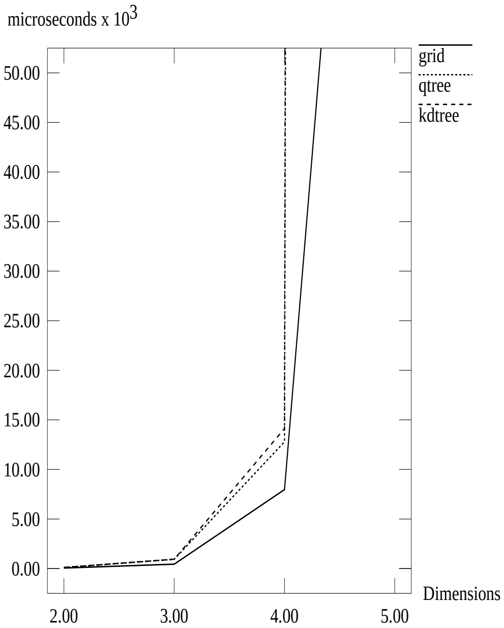

Figure 13: Search overhead for Uniformly Distributed Data (25% buckets non-empty)

4.2 Bucket Vs Non-Bucket Methods

In this subsection the results from the implementation of the analytical models of section 3.2 are presented. In the calculation of memory overhead for linked array, only the extra memory required to maintain the linked list was considered. As mentioned before in calculating the search time, the time for display of records found was ignored. The values of $G = 16$ and $D = 4$ were used in the implementations.

4.2.1 Uniform Data Distribution

Figures 18 and 19 show the comparison between the linked array and the grid array. As mentioned before in subsection 4.2 only the grid array was chosen among bucket methods as it has the best performance for uniformly distributed data. As far as search time overhead is considered the linked array performed better than the grid for up to approximately $100,000$ points. However the drawback is that the memory overhead for this structure keeps increasing as the size of the data set increases unlike the case for the grid array where it remains a constant. So users may be constrained in using linked array because of its high memory requirements.

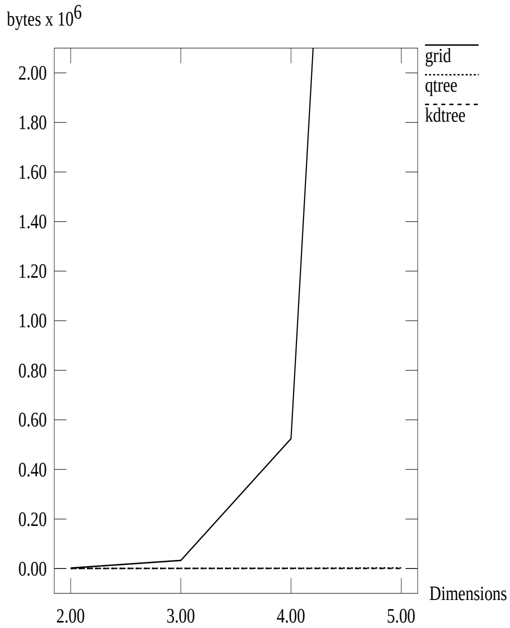

Figure 14: Memory overhead for Skewed Data Distribution (non-empty buckets along diagonal only)

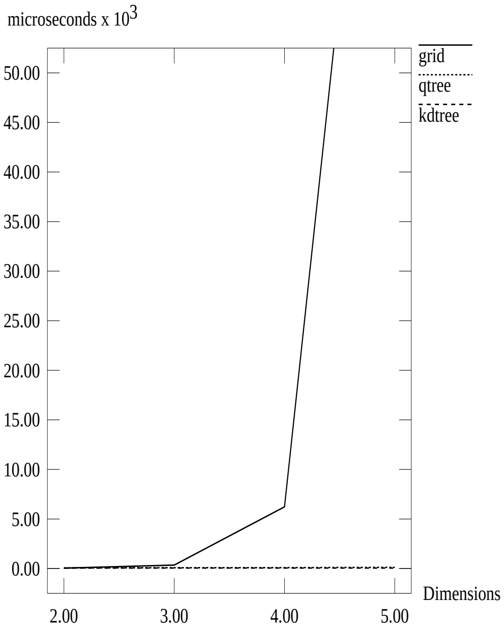

Figure 15: Search overhead for Skewed Data Distribution (non-empty buckets along diagonal only)

4.2.2 Skewed Data Distribution

Figures 20 and 21 show the comparison between the linked array and the tree structures for skewed distributions. As mentioned before in subsection 4.2 only the trees were chosen among bucket methods as they have significantly better performance for skewed data distribution. When compared to the linked array the tree structures get significantly better than the linked array both in terms of search time and memory overhead. However when the number of tuples is small (about $10,000$) it is better to use a linked array because of its simplicity.

In this section the results of implementations have been discussed. All the results were as predicted by the analytical models. It can be noticed that there is a steep rise in the gradient of the curves showing the search time overhead for uniformly distributed data. This is due to the memory requirements getting so high that a very high number of page faults starts occuring.

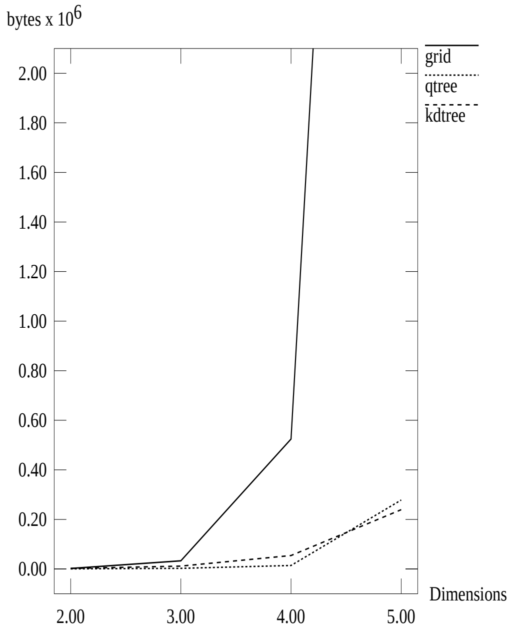

Figure 16: Memory overhead for Skewed Data Distribution (all non-empty buckets within a distance of $G/4$ of diagonal)

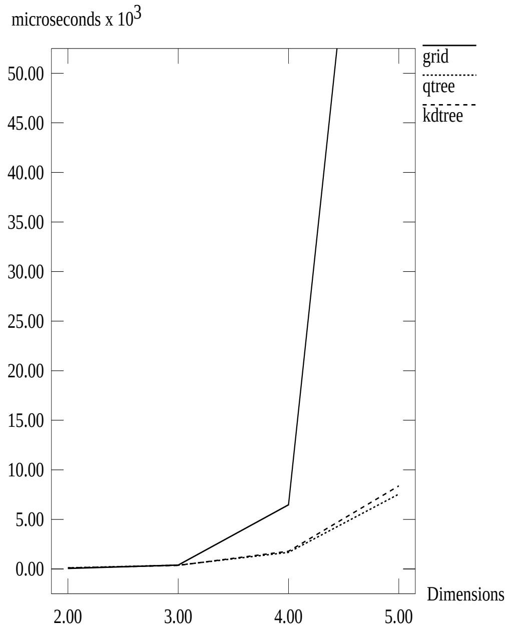

Figure 17: Search overhead for Skewed Data Distribution (all non-empty buckets within a distance of $G/4$ of diagonal)

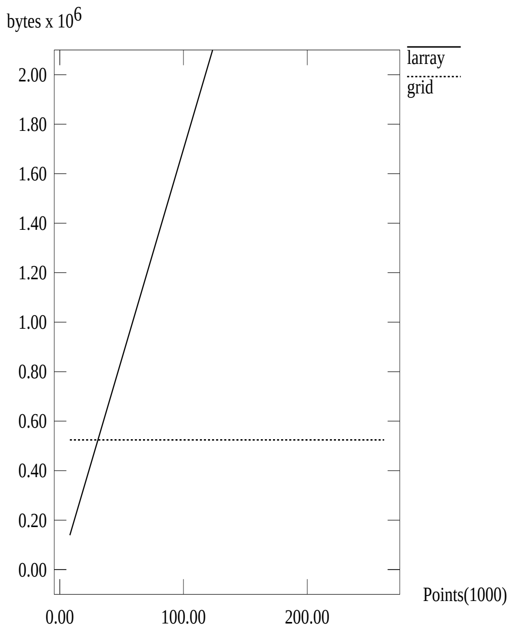

Figure 18: Memory overhead for Uniformly Distributed Data

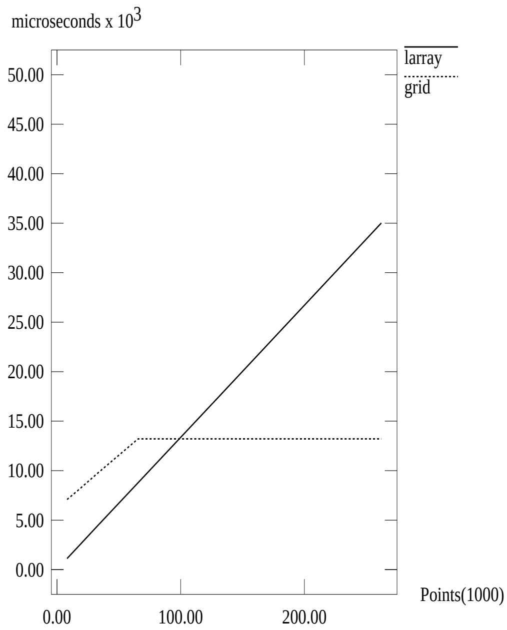

Figure 19: Search overhead for Uniformly Distributed Data

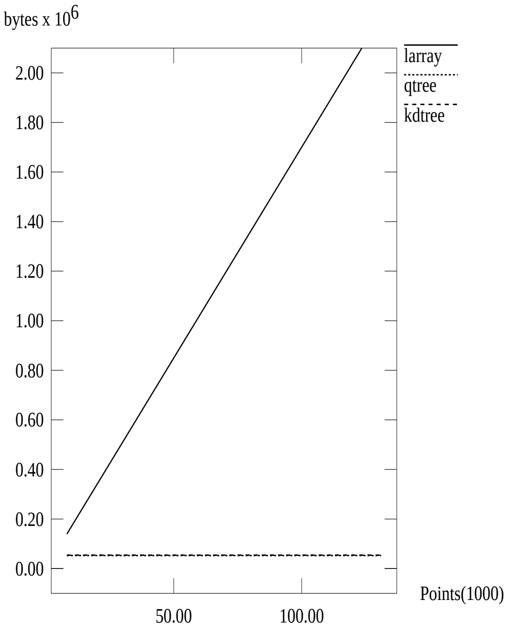

Figure 20: Memory overhead for Skewed Data Distribution (all non-empty buckets within a distance of $G/4$ of diagonal)

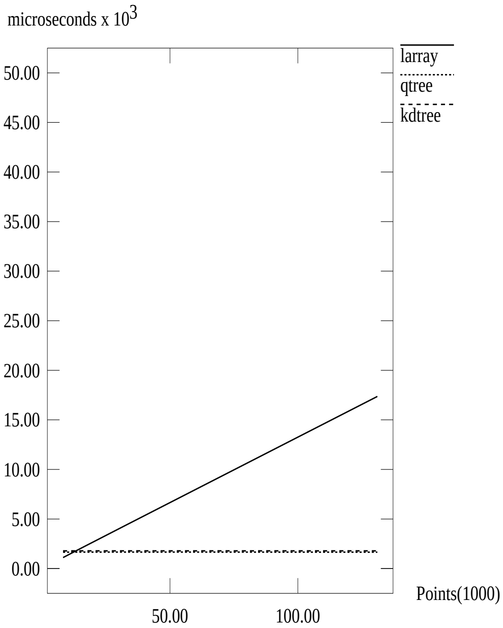

Figure 21: Search overhead for Skewed Data Distribution (all non-empty buckets within a distance of $G/4$ of diagonal)

5 Conclusions

5.1 Contributions

We have presented a way of analyzing data structures for dynamic queries applications. Analytical models were constructed and their usefulness was shown by empirical data. In almost all cases the empirical results confirmed the analytical models. The results can be divided into two on the basis of the data distributions used.

In the case of uniformly distributed data the linked array structure performed quite well but the drawback in this structure is that its memory overhead is very high and therefore it should be used only for small data sets. For larger data sets it is recommended that a grid array be used. The advantage in the grid array is that the memory overhead does not depend on the number of points in the data set but only on the number of buckets in the data set.

The second case investigated was for skewed data distributions where most of the buckets are empty. The performance of the tree structures, k-d tree, and quad tree was much better than the grid array. Among tree structures the k-d tree used marginally less memory but had a marginally higher search overhead. Compared to the linked array again the trees were much better except for the cases where the number of data points were just a few thousand. However there is a temptation to use the linked array because of its simplicity.

It is recommended that the tree structures be used for skewed data distributions if the number of points exceed a few thousand.

In cases where knowledge of the data distribution is lacking we recommend using the k-d tree as the it is highly likely that the distribution is non-uniform. The k-d tree is also much easier to construct compared to the quad tree when the ranges of the sliders are not equal. It was noticed that the performance of a data structure does not change with the dimensionality of the data set. The only effect of increasing dimensions is that the number of buckets increases, which results in the differences in the performance becoming more pronounced.

The data structures discussed in this paper are practical and make it possible to implement dynamic queries on standard machines in common use without major special requirements. This is essential, specially because in addition to experts, novice users with inexpensive machines also find DQ very appealing.

Dynamic query applications can be integrated with standard DBMS to rapidly browse through parts of a database, as it is usually not possible to fit entire databases in main memory. One way of doing this is to select a part of the database which is to be browsed using dynamic queries. The selected parts could then be loaded in memory and browsed through rapidly. This technique was used in "Dynamic Trend Maps", a project developed for the National Center for Health Statistics, where trends in mortality data for 38 types of cancer and their correlation to 15 demographic variables was studied using a dynamic query interface.

5.2 Future Directions

While analyzing the data structures an assumption was made that the data set remains frozen. However data sets usually change over time. It would be an interesting problem to investigate the effect of updates on these data structures. Another assumption made was about the nature of queries, which were assumed to be a simple conjunct of ranges. In many applications this assumption does not hold. The suitability of these structures on the relaxation of this assumption can also be studied.

Data structures that were investigated were for main memory only. However for very large data sets it would be impossible to do with main memory only. In such cases disk accesses become a necessity. Even though seeing the current state of technology it would not be possible to maintain real time performance it would be worth while to relax this requirement and study applications where data is organized on disks.

The segregation of data into buckets can also lead to interesting methods for compression. Values of some of the fields in the data set can be determined just by knowing the bucket the record resides in. If dynamic queries are to run in main memory then the issues of compression become very important.

For very large data sets the querying could be done on multiple machines i.e. the query could be distributed. In these cases data sets have to be fragmented and replicated at the local sites of the machines involved in the computation. Good schemes which reduce communication overhead and distribute query computation load in a fair manner become essential.

Making dynamic query applications run on parallel machines will give us the capability of handling very large data sets while maintaining real time performance. Using parallel machines will require solving of problems like, efficient distribution of data among nodes and ways of reducing messages between nodes.

One of the reasons dynamic query applications are effective is because they present query results in a way to help users visualize the data set. Therefore effective ways of visualizing data, specially multi-dimensional data are important for the success of dynamic queries.

Acknowledgements : We would like to thank The National Center for Health Statistics for supporting the development of "Dynamic Trend Maps" which in part inspired this work. We also thank Catherine Plaisant for her leadership in developing "Dynamic Trend Maps".

References

[1] C. Ahlberg, C. Williamson, and B. Shneiderman, "Dynamic Queries for Information Exploration: An Implementation and Evaluation", *Proc. CHI'92, ACM*, New York, 1992, pp. 619-626.

[2] N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-Tree: an Efficient and Robust Access Method for Points and Rectangles", *ACM SIGMOD*, Atlantic City NJ, 1990, pp. 322-331.

[3] D.A. Beckley, M.W. Evans and V.K. Raman, "Multikey Retrieval from K-d Trees and Quad-Trees", *Proc. ACM SIGMOD International Conference on the Management of Data*, Austin, 1985, pp. 291-301.

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

[5] J. L. Bentley and D. F. Stanat, "Analysis of Range Searches in Quad Trees", *Information Processing Letters*, Vol. 3, No. 6, 1975, pp. 170-173.

[6] J. L. Bentley, D. F. Stanat and E. H. Williams Jr., "The Complexity of Fixed-Radius Near Neighbor Searching", *Information Processing Letters*, Vol. 6, No. 6, December 1977, pp. 209-212.

[7] J. L. Bentley, "Decomposable Searching Problems", *Information Processing Letters*, Vol. 8, No. 5, 1979, pp. 133-136.

[8] J. Bentley and J. Friedman, "Data Structures for Range Searching", *Computing Surveys*, Vol. 11, No. 4, December 1979, pp. 397-409.

[9] J. Bentley and H. Maurer, "Efficient Worst-Case Data Structures for Range Searching", *Acta Informatica*, Vol. 13, No. 2, 1980, pp. 155-168.

[10] S. Dandamudi and P. Sorenson, "Algorithms for BD Trees", *Software-Practice and Experience*, Vol. 16, No. 12, December 1986, pp. 1077-1096.

[11] C. Faloutsos and P. Bhagwat, "Declustering Using Fractals", *2nd International Conference on Parallel and Distributed Information Sysems*, San Diego CA, 1993, pp. 18-25.

[12] R. A. Finkel and J. L. Bentley, "Quad Trees , A Data Structure for Retrieval on Composite Keys", *Acta Informatica*, Vol. 4, 1974, pp. 1-9.

[13] H. Garcia-Molina and K. Salem, "Main Memory Database Systems: An Overview", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, 1992, pp. 509-516.

[14] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proc. ACM SIGMOD Conference*, Boston, 1984, pp. 47-57.

[15] D. E. Knuth, "The Art of Computer Programming, Vol. 3: Sorting and Searching", Addison-Wesley, 1973.

[16] D. T. Lee and C. K. Wong, "Worst-Case Analysis for Region and Partial Region Searches in Multidimensional Binary Search Trees and Balanced Quad Trees", *Acta Informatica*, Vol. 9, 1977, pp. 23-29.

[17] D. Lomet, "A Review of Recent Work on Multi-attribute Access Methods", *SIGMOD RECORD*, Vol. 21, No. 3, September 1992, pp. 56-63.

[18] V. Y. Lum, "Multi-attribute Retrieval with Combined Indexes", *Communications of the ACM*, Vol. 13, No. 11, 1970, pp. 660-665.

[19] J. Nievergelt and H. Hinterberger, "The Grid File: An Adaptable, Symmetric Multikey File Structure", *ACM Transactions on Database Systems*, Vol. 9, No. 1, March 1984, pp. 38-71.

[20] M. Regnier, "Analysis of Grid File Algorithms", *BIT*, Vol. 25, 1985, pp. 335-357.

[21] P. Reisner, "Human Factors Studies of Database Query Languages: A Survey and Assessment", *Computing Surveys*, Vol. 13, No. 1, 1981, pp. 13-31.

[22] H. Samet, "The Design and Analysis of Spatial Data Structures", Chapter 2, Addison Wesley 1989.

[23] P. Scheuermann and M. Ouksel, "Multidimensional B-Trees for Associative Searching in Database Systems", *Information Systems*, Vol. 7, No. 2, 1982, pp. 123-137.

[24] B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages", *IEEE Computer*, Vol. 16, No. 8, August 1983, pp. 57-69.

[25] B. Shneiderman, "Designing the User Interface: Strategies for effective Human-Computer Interaction", Second Edition, Chapter 5, Addison-Wesley 1992.

[26] C. Williamson and B. Shneiderman, "The Dynamic Homefinder: Evaluating Dynamic Queries in a Real-Estate Information Exploration System", *Proc. ACM SIGIR Conference on Information Retrieval*, Copenhagen Denmark, 1992.