4. HCQT FRAME WORK

4.1 INTRODUCTION

Searching techniques for nearest neighbour can be grouped into two categories; organize the data to be stored and those that organize the embedding space from which the data are drawn. In other words it can be called as object-space based hierarchies and image-space based. The binary search tree is an example of the former since the boundaries of different regions in the search space are determined by the data being stored. Address computation methods such as radix searching are examples of the latter. Since region boundaries are chosen from among locations that are fixed regardless of the content of the file.

4.2 NON HIERARCHICAL DATA STRUCTURE

The simplest way to store the point data is in a sequential file. Given N records and k-attributes to search on, searching such a data structure is $O(N*k)$ process since each record must be examined. Consider the set of 8 cities and their locations as shown in Table 4.1.

<table>
<thead>
<tr>
<th>Name</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Srinagar</td>
<td>34</td>
<td>74</td>
</tr>
<tr>
<td>Mumbai</td>
<td>18</td>
<td>72</td>
</tr>
<tr>
<td>Trivandrum</td>
<td>8</td>
<td>76</td>
</tr>
<tr>
<td>Bhubaneswar</td>
<td>20</td>
<td>85</td>
</tr>
<tr>
<td>Delhi</td>
<td>28</td>
<td>77</td>
</tr>
<tr>
<td>Raipur</td>
<td>21</td>
<td>81</td>
</tr>
<tr>
<td>Kolkata</td>
<td>22</td>
<td>88</td>
</tr>
<tr>
<td>Chennai</td>
<td>13</td>
<td>80</td>
</tr>
</tbody>
</table>
Another common technique is inverted list method, in which each for each key a sorted list method is maintained of the records in a file. Inverted list representation is stored of Table 4.1 shown in Table 4.2. There are two sorted list, one for x-coordinate and another for y-coordinate. Data stored in the list are pointers to into Table 4.1. This enables the pruning the search with respect to one key. The end points of the desired range for one key can be located efficiently by the corresponding sorted list. The resulting list searched by brute force. The average searching time is $O(N^{1-1/k})$.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trivandrum</td>
<td>Mumbai</td>
</tr>
<tr>
<td>Chennai</td>
<td>Srinagar</td>
</tr>
<tr>
<td>Mumbai</td>
<td>Trivandrum</td>
</tr>
<tr>
<td>Bhubaneswar</td>
<td>Delhi</td>
</tr>
<tr>
<td>Raipur</td>
<td>Chennai</td>
</tr>
<tr>
<td>Kolkata</td>
<td>Raipur</td>
</tr>
<tr>
<td>Delhi</td>
<td>Bhubaneswar</td>
</tr>
<tr>
<td>Srinagar</td>
<td>Kolkata</td>
</tr>
</tbody>
</table>

A popular method used by the cartographers is called fixed grid method. It divides the space into equal sized cell having width twice the search radius for a rectangular query. These often referred as buckets. The data structure is essentially a directory in the form of k-dimensional array with entry per cell. Each cell may be implemented as a link list to represent the points in it.

Figure 4.3 is a grid representation for each the data in figure 4.1 for a search region consisting of square of size 4 X 8. Assuming a 40 X 20 coordinate space, we have 25 rectangles of equal size. We adopt a convention that each rectangle is open with respect to its upper right boundaries and closed with respect to lower left boundaries. Therefore Trivandrum is located at (8,76) is found in the rectangle centered at (4, 76). The average search time for a range query using the grid is $O(F.2^k)$ where the F is the number of records found. The factor $2^k$ is the
maximum number of cells that must be accessed when the search rectangle is permitted to overlap more than one cell. If the range queries are made using only a fixed radius, the fixed grid is an efficient representation. For a non uniform distribution, it is less efficient because buckets may be unevenly filled leading to nearly empty pages as well as long overflow chains.

Figure 4.1: Grid representation for the data in Table 4.1

Number deficiencies are associated with the above mentioned structures. The fixed grid is characterized as organizing the embedding space of data into regions that contain records. Fixed grid method is not satisfactory, when the data is not distributed uniformly. This leads to bucket overflow. The problem with bucket overflow is that it leads to more number of disk accesses.
4.3 INDEXING STRUCTURE

The typical notion of a database is that of a system in which one stores data about a fixed enterprise that one has modeled in some way. Quite often, the modeling is constrained in ways that help to reduce the complexity to a manageable level. The usual kind of uses to which that data is put includes fairly straightforward applications like inventory management, shipping systems, and payroll systems. However, research in databases has advanced the state-of-the-art sufficiently that we are now moving into non-traditional applications that could not have been foreseen even ten years ago. This move has been largely demand-driven. For example, the emergence of spatial databases was initially driven by the demand for managing large volume of spatial data used in the geosciences and computer-aided design (CAD). The thrust was made more vigorous by newfound applications in computer vision and robotics, computer visualization, geographical information processing, automated mapping and facilities management. The kind of data modeling required is far more ambitious and complex than that required in the more traditional application areas.

Spatial databases of real-world interest are very large, with sizes ranging from tens of thousands to millions objects. These spatial objects may intersect, be adjacent to others, or contain other objects. Efficient processing of queries manipulating spatial relationships relies upon auxiliary indexing structures. Spatial relationships are materialized dynamically during query processing. In order to find spatial objects efficiently based on proximity, it is essential to have an index over spatial locations. The underlying data structure must support efficient spatial operations, such as locating the neighbours of an object and identifying objects in a defined query region.

Most indexes are based on the principle of divide and conquer [AHU74]. Indexing structures following this approach are typically hierarchical. The approach is naturally suitable for a database system where the memory space is limited, and hence the pruning of a search must be performed such that the more detail to be examined, the smaller number of objects are being examined. An advantage of hierarchical structures is that they are efficient in range searching. Indexing in a spatial database (SD) is different from indexing in a conventional database in that
Data in an SDS are multi-dimensional objects and are associated with spatial coordinates. The search is based not on the attribute values but on the spatial properties of objects.

The efficiency of operations is dictated by how the data is represented, and how fast the relevant data for a particular computation can be retrieved. The representation scheme which maps the original data objects into a set of objects in the storage medium is crucial to the computation of the operators. The sheer volume of the data being retrieved means that one must minimize the number of disk pages being accessed and the amount of redundant data being fetched and manipulated.

Non-spatial data are different from spatial ones in that spatial data are associated with spatial locations, and many-to-many spatial relationships exist among spatial objects. It is possible to represent some frequently referenced relationships using specially
constructed relations or fragments of existing relations, or by maintaining conventional links. However, it is not pragmatic to pre-materialize all such relationships.

Consequently, dynamic evaluation of spatial relationships is necessary. The basic concept of indexing spatial data in a space is to partition the space into a manageable number of smaller subspaces, which are in turn further partitioned into even smaller subspaces, and so on. The process repeats until each unpartitioned subspace contains a small number of objects which can be stored in a data page. Different space partitioning strategies may be designed to reduce retrieval time and storage space. By doing so, a hierarchy of spaces is formed, which should be organized using an appropriate data structure to reduce retrieval time. Similar to other databases, the two basic issues that need to be addressed in designing an indexing structure for spatial database are the efficient use of storage and the ease of information retrieval. The techniques used for adapting existing indexes into spatial indexes are dealt in the next section.

4.3.1 TRANSFORMATION APPROACH

Parameter Space Indexing: Objects with \( n \) vertices in a \( k \)-dimensional space are mapped into points in an \( nk \)-dimensional space. For example, a two-dimensional rectangle described by the bottom left corner \((x_1, y_1)\) and the top right corner \((x_2, y_2)\) is represented as a point in a four-dimensional space, where each attribute is taken as from a different dimension. After the transformation, points can be stored directly in existing point indexes. An advantage of such an approach is that there is no major alteration of the multi-dimensional base structure. The problem with the mapping scheme is that the \( k \)-dimensional objects that are spatially close in a \( k \)-dimensional space may be far apart when they are represented as points in an \( nk \) dimensional space. As a consequence, intersection search can be inefficient. Also, the complexity of insertion operation typically increases with higher dimensionality.

Mapping to Single Attribute Space: The data space is partitioned into grid cells of the same size, which are then numbered according to some curve-filling methods. A spatial object is then represented by a set of numbers or one-dimensional objects. These one-dimensional objects can be indexed using conventional indexes such as B+-trees. A B+-tree can be used directly to index objects that have been mapped from a \( k \)-dimensional space into points in a one-dimensional space.
4.3.2 NON-OVERLAPPING NATIVE SPACE INDEXING APPROACH

Object Duplication: A k-dimensional data space is partitioned into pairwise disjoint subspaces. These subspaces are then indexed. An object identifier is duplicated and stored in all of the subspaces it intersects; that is, an object identifier may be stored in multiple pages.

Object Clipping: A technique similar to object duplication object clipping. Instead of duplicating the identifier, an object is decomposed into several disjoint smaller objects so that each smaller sub-object is totally included in a subspace.

The most important property of object duplication or clipping is that the data structures used are straightforward extensions of the underlying point indexing structures. Also, both points and multi-dimensional non-zero sized objects can be stored together in one file without having to modify the structure. However, an obvious drawback is the duplication of objects which requires extra storage and hence more expensive insertion and deletion procedures. Another limitation is that the density (i.e., the number of objects that contain a point) in a map space must be less than the page capacity (i.e., the maximum number of objects that can be stored in a page).

4.3.3 OVERLAPPING NATIVE SPACE INDEXING APPROACH

The basic idea to indexing spatial database is to hierarchically partition its data space into a manageable number of smaller subspaces. While a point object is totally included in an unpartitioned subspace, a non-zero sized object may extend over more than one subspace. Rather than supporting disjoint subspaces as in the non-overlapping space indexing approach, the overlapping native space indexing approach allows overlapping subspaces such that objects are totally included in only one of the subspaces. These subspaces are organized as a hierarchical index and spatial objects are indexed in their native space.

A major design criterion for indexes using such an approach is the minimization of both the overlap between bounding subspaces and the coverage of subspaces. A poorly
designed partitioning strategy may lead to unnecessary traversal of multiple paths. Further, dynamic maintenance of effective bounding subspaces incurs high overhead during updates.

A number of indexing structures use more than one extending technique. Since each extending method has its own weaknesses, the combination of two or more methods may help to compensate the weaknesses of each other. However, an often overlooked fact is that the use of more than one extending method may also produce a counter effect: inheriting the weaknesses from each method.

The main purpose spatial access method is to support efficient spatial selection, for example range queries or nearest neighbour queries, of spatial objects. Further, spatial access methods are also used to implement efficient spatial analysis such as map overlay, and other types of spatial joins. A characteristic of spatial data sets is that they are usually large and that the data is quite often distributed in an irregular manner. A spatial access method needs to take into account both spatial indexing and clustering techniques. Without a spatial index, every object in the database has to be checked whether it meets the spatial selection criterion; a 'full table scan' in a relational database. As spatial data sets are usually very large, this is unacceptable in practice for interactive use and most other applications. Therefore, a spatial index is required, which enables to find the required object addresses efficiently without looking at every object. In case the whole spatial data set resides in main memory it is sufficient to know the addresses of the requested objects, as main memory storage allows random access and does not introduce significant delays. However, most spatial data sets are so large they cannot reside in the main memory of the computer and must be stored on secondary memory. Otherwise, many different disk pages will have to be fetched, resulting in slow response. For spatial selecting the clustering implies storing objects which are close together in reality also close together in the computer memory (instead of scattered over the whole memory). Solutions for this are storing the objects in the order implied by the index traversal (of the whole domain) or try to achieve some kind of spatial clustering based on a space filling curve.
4.4 QUADTREES

Quadtrees are hierarchical spatial tree data structures that are based on the principle of recursive decomposition of space. The term *quadtree* originated from representation of two dimensional data by recursive decomposition of space using separators parallel to the coordinate axis. The resulting split of a region into four regions corresponding to southwest, northwest, southeast and northeast quadrants is represented as four children of the node corresponding to the region, hence the term “quad”tree. In a three dimensional analogue, a region is split into eight regions using planes parallel to the coordinate planes. As each internal node can have eight children corresponding to the 8-way split of the region associated with it, the term *octree* is used to describe the resulting tree structure. Analogous data structures for representing spatial data in higher than three dimensions are called *hyperoctrees*. It is also common practice to use the term *quadtrees* in a generic way irrespective of the dimensionality of the spatial data. This is especially useful when describing algorithms that are applicable regardless of the specific dimensionality of the underlying data.

Several related spatial data structures are described under the common rubric of quadtrees. Common to these data structures is the representation of spatial data at various levels of granularity using a hierarchy of regular, geometrically similar regions (such as cubes, hyper rectangles etc.). The tree structure allows quick focusing on regions of interest, which facilitates the design of fast algorithms. As an example, consider the problem of finding all points in a data set that lie within a given distance from a query point, commonly known as the spherical region query. In the absence of any data organization, this requires checking the distance from the query point to each point in the data set. If a quadtree of the data is available, large regions that lie outside the spherical region of interest can be quickly discarded from consideration, resulting in great savings in execution time. Furthermore, the unit aspect ratio employed in most quadtree data structures allows geometric arguments useful in designing fast algorithms for certain classes of applications.

Another important aspect of the decomposition process is the termination condition to stop the subdivision process. This identifies regions that will not be subdivided further, which will be represented by leaves in the quadtree. Quadtrees have been used as fixed
resolution data structures, where the decomposition stops when a preset resolution is reached, or as variable resolution data structures, where the decomposition stops when a property based on input data present in the region is satisfied. They are also used in a hybrid manner, where the decomposition is stopped when either a resolution level is reached or when a property is satisfied.

4.5 QUADTREES FOR POINT DATA

We first explore quadtrees in the context of the simplest type of spatial data – multidimensional points. Consider a set of n points in d dimensional space. The principal reason a spatial data structure is used to organize multidimensional data is to facilitate queries requiring spatial information. A number of such queries such as range query, spherical region query and nearest neighbour query can be identified for point data.

While quadtrees are used for efficient execution of such spatial queries, one must also design algorithms for the operations required of almost any data structure such as constructing the data structure itself, and accommodating searches, insertions and deletions. Though such algorithms will be covered first, it should be kept in mind that the motivation behind the data structure is its use in spatial queries. If all that were required was search, insertion and deletion operations, any one dimensional organization of the data using a data structure such as a binary search tree would be sufficient.

The point quadtree is a natural generalization of the binary search tree data structure to multiple dimensions. For convenience, first consider the two dimensional case. Start with a square region that contains all of the input points. Each node in the point quadtree corresponds to an input point. To construct the tree, pick an arbitrary point and make it the root of the tree. Using lines parallel to the coordinate axis that intersect at the southwest, northwest, southeast and northeast quadrants, respectively. Each of the subregions is recursively decomposed in a similar manner to yield the point quadtree.

For points that lie at the boundary of two adjacent regions, a convention can be adopted to treat the points as belonging to one of the regions. For instance, points lying on the
left and bottom edges of a region may be considered included in the region, while points lying on the top and right edges are not. When a region corresponding to a node in the tree contains a single point, it is considered a leaf node. Note that point quadtrees are not unique and their structure depends on the selection of points used in region subdivisions. Irrespective of the choices made, the resulting tree will have \( n \) nodes, one corresponding to each input point.

![Fig. 4.3: Two dimensional set of points and corresponding point quadtree.](image)

In the quadtree representation, the root node corresponds to the entire array. Each son of a node represents a quadrant labeled in order NW, NE, SW, and SE of the region represented by that node. The leaf node the node corresponds to those blocks for which no further division is necessary.

The point quadtree is implemented as the generalization of a binary search tree. In dimensions each data is represented as a node containing seven fields as shown in Figure 4.4. The first four fields contain pointers to the four corresponding to the directions NW, NE, SW and SE. If \( P \) is a pointer to a node and \( I \) is a quadrant, then fields are referenced as \( \text{SON}(P, I) \). We can determine the specific quadrant in which a node say \( P \), lies relative to its father by the use of \( \text{SONTYPE}(P) \), which has a value of \( I \) if \( \text{SON} \left( \text{FATHER}(P), I \right) = P \). \( \text{XCOORD} \) and \( \text{YCOORD} \) contain the values of \( x \) and \( y \) coordinates respectively of the data point. The name field contains
If all the input points are known in advance, it is easy to choose the points for subdivision so as to construct a height balanced tree. A simple way to do this is to sort the points with one of the coordinates, say x, as the primary key and the other coordinate, say y, as the secondary key. The first subdivision point is chosen to be the median of this sorted data. This will ensure that none of the children of the root node receives more than half the points.

In \(O(n)\) time, such a sorted list can be created for each of the four resulting sub regions. As the total work at every level of the tree is bounded by \(O(n)\), and there are at most \(O(\log n)\) levels in the tree, a height balanced point quadtree can be built in \(O(n \log n)\) time. Generalization to d dimensions is immediate, with \(O(dn \log n)\) run time.

### 4.5.1 INSERTION

To insert a new point not already in the tree, first conduct a search for it which ends in a leaf node. The leaf node now corresponds to a region containing two points. One of them is chosen for subdividing the region and the other is inserted as a child of the node corresponding to the sub region it falls in. The run time for point insertion is also bounded by \(O(DH)\), where \(h\) is the height of the tree after insertion. One can also construct the tree itself by repeated insertions using this procedure. Similar to binary search trees, the run time under a random sequence of insertions is expected to be \(O(n \log n)\) [6]. Overmars and van Leeuwen [OL82] resent algorithms for constructing and maintaining optimized point quadtrees irrespective of the order of insertions.
The recursive structure of a point quadtree immediately suggests an algorithm for searching. To search for a point, compare it with the point stored at the root. If they are different, the comparison immediately suggests the subregion containing the point. The search is directed to the corresponding child of the root node. Thus, search follows a path in the quadtree until either the point is discovered, or a leaf node is reached. The run time is bounded by $O(dh)$, where $h$ is the height of the tree.

```
COMPARE (P,R)

// return the quadrant of the quadtree rooted at node R in which node P belongs.
If x-coordinate of P < x-coordinate of R
    If y-coordinate of P < y-coordinate of R then cname=‗sw‘ else cname=‗nw‘
Else
    If y-coordinate of P < y-coordinate of R then cname=‗se‘ else cname=‗ne‘
End.
```

**Figure 4.5: Algorithm for finding the appropriate region**

Deletion in point quadtrees is much more complex. The point to be deleted is easily identified by a search for it. The difficulty lies in identifying a point in its subtree to take the place of the deleted point. This may require nontrivial readjustments in the subtree underneath. The deletion of a node in a point quadtree is discussed in detail [FB74]. An analysis of the expected cost of various types of searches in point quadtrees is presented by Flajolet et al [FGP90].
Insert (P, R)

//to insert the node P in the quadtree rooted at node R
If R=Nil then R=P
Else
Begin
T=R
While T <> Nil
Begin
F=T
Q=COMPARE (P, T)
T=son (T, Q)
End
End

Figure 4.6: Algorithm for inserting a point

The amount of work done in building the quadtree is equal to the maximum path
length of the tree and it is equal to $O(\log_4 N)$, which gives average cost of inserting as well as
searching a node of $O(\log_4 N)$. This is dependent on the node to be inserted. The worst case
occurs when the node to be inserted is the son of deepest node in the tree. So to reduce the time
one of the following techniques can be used.

One approach proposed by Finkel and Bently assumes all the nodes are known in
advance. Building the optimized quadtree requires the data to be sorted by one key and
secondarily by another key. The root of the tree is said to the median of the data and the
remaining are grouped into four subsections that will form sub trees of the root. Another
approach discussed by Overmars and van Leeuwen is the dynamic formulation of the above
method. The optimized quadtree is built as data points are inserted into it.
4.5.2 DELETION

The algorithm is similar to the method used in binary search trees. When a node is deleted in binary search tree, it is replaced with a node which is very nearer to it. In other words it is replaced by its either the in order successor or predecessor of the node to be deleted. But in a quadtree we need to find in which direction the in order successor or predecessor has to found. In a quadtree there are four candidates for replacement. The algorithm Find_candidate will find them. Now it is required the find best among them for replacement.

\[
\text{Find\_candidate}(P,Q)
\]

\[
// P is a pointer to the son in quadrant Q of the node to be deleted
\]

1. If \( P = \text{nil} \)
   \( \text{return}(\text{INF}(Q)) \)
2. While \( \text{SON}(P, \text{OPQUAD}(Q))\neq\text{nil} \)
   \( P= \text{SON}(P, \text{OPQUAD}(Q)) \)
3. Return(P)

\[\text{Figure 4.7: Algorithms Finding the Node For Replacement}\]

\[\text{Figure 4.8: Quadtree before deletion}\]
Figure 4.9: Quadtree After deleting the node A

Once the set of candidates nodes found, an attempt is made to find the best candidate, which becomes the replacement node. There are two criteria for searching the best candidate. Criterion 1 stipulates the choice of the candidate that is closer to the each of its bordering axes than any other candidate on the same side of axes if such candidate exists. Note that the situation may arise that no candidate satisfies the criterion 1 or many candidates satisfy it. In such a situation the candidate with minimum Mahattan metric or city block metric is chosen. The Mahattan metric is sum of the displacements from the bordering x and y axes.

Deletion algorithm make use of the properties of the space obtained by new partition to reduce the number of nodes for reinsertion. The algorithm consists of two procedures new_root and Adj_quad. Let A be the node to be deleted and I be the quadrant of the tree containing B, the replacement node for A.

```
Delete(P,R)
//delete P from the the node point quadtree rooted at node R.
1. If P has no SON or only one SON
2. J= Father of P
3. If j<> nil
4. Attach P’s child to J
5. Else
6. Find the best candidate for replacement using Find_candidate and rearrange quadrant.
```

Figure 4.10: Deleting a Node in Point Quadtree
4.6 REGION QUADTREE

Quadtree data structures that use equal subdivision of the underlying space, called region quadtrees. This is because we regard Bentley’s multidimensional binary search trees, also called k-d trees, to be superior to point quadtrees. The k-d tree is a binary tree where a region is subdivided into two based only on one of the dimensions. If the dimension used for subdivision is cyclically rotated at consecutive levels in the tree, and the subdivision is chosen to be consistent with the point quadtree, then the resulting tree would be equivalent to the point quadtree but without the drawback of large degree. Thus, it can be argued that point quadtrees are contained in k-d trees. Furthermore, recent results on compressed region quadtrees indicate that it is possible to simultaneously achieve the advantages of both region and point quadtrees. In fact, region quadtrees are the most widely used form of quadtrees despite their dependence on the spatial distribution of the underlying data. While their use posed theoretical inconvenience, it is possible to create as large a worst-case tree as desired with as little as three points; they are widely acknowledged as the data structure of choice for practical applications.

Figure 4.11: A two dimensional set of points and the corresponding region quadtree
The region quadtree for n points in d dimensions is defined as follows: Consider a hypercube large enough to enclose all the points. This region is represented by the root of the d-dimensional quadtree. The region is subdivided into $2^d$ subregions of equal size by bisecting along each dimension. Each of these regions containing at least one point is represented as a child of the root node. The same procedure is recursively applied to each child of the root node. The process is terminated when a region contains only a single point. This data structure is also known as the point region quadtree, or PR-quadtree for short [SRS84]. At times, we will simply use the term quadtree when the tree implied is clear from the context. The region quadtree corresponding to a two dimensional set of points is shown in Figure 4.11. Once the enclosing cube is specified, the region quadtree is unique. The manner in which a region is subdivided is independent of the specific location of the points within the region. This makes the size of the quadtree sensitive to the spatial distribution of the points.

Figure 4.12: Illustration of hierarchy of cells in two dimensions. Cells D, E, F and G are immediate subcells of C. Cell H is an immediate subcell of D, and is a subcell of C.
Before proceeding further, it is useful to establish a terminology to describe the type of regions that corresponds to nodes in the quadtree (see Figure 4.12 for an illustration of the cell hierarchy in two dimensions). Call a hypercubic region containing all the points the root cell. Define a hierarchy of cells by the following: The root cell is in the hierarchy. If a cell is in the hierarchy, then the 2\(^d\) equal-sized cubic subregions obtained by bisecting along each dimension of the cell are also called cells and belong to

We use the term subcell to describe a cell that is completely contained in another. A cell containing the subcell is called a supercell. The subcells obtained by bisecting a cell along each dimension are called the immediate subcells with respect to the bisected cell. Also, a cell is the immediate supercell of any of its immediate subcells. We can treat a cell as a set of all points in space contained in the cell. Thus, we use \( C \subseteq D \) to indicate that the cell \( C \) is a subcell of the cell \( D \) and \( C \subset D \) to indicate that \( C \) is a subcell of \( D \) but \( C \neq D \). Define the length of a cell \( C \), denoted \( \text{length}(C) \), to be the span of \( C \) along any dimension.

An important property of the cell hierarchy is that, given two arbitrary cells, either one is completely contained in the other or they are disjoint. Cells are considered disjoint if they are adjacent to each other and share a boundary. Each node in a quadtree corresponds to a subcell of the root cell. Leaf nodes correspond to largest cells that contain a single point. There are as many leaf nodes as the number of points, \( n \). The size of the quadtree cannot be bounded as a function of \( n \), as it depends on the spatial distribution. For example, consider a data set of 3 points consisting of two points very close to each other and a faraway point located such that the first subdivision of the root cell will separate the faraway point from the other two. Then, depending on the specific location and proximity of the other two points, a number of subdivisions may be necessary to separate them. In principle, the location and proximity of the two points can be adjusted to create as large a worst-case tree as desired. In practice, this is an unlikely scenario due to limits imposed by computer precision.

Let \( s \) be the smallest distance between any pair of points and \( D \) is the length of the root cell. An upper bound on the height of the quadtree is obtained by considering the worst-case path needed to separate a pair of points which have the smallest pair wise distance. The length of
the smallest cell that can contain two points $s$ apart in $d$ dimension is $\frac{s}{\sqrt{d}}$. This is shown in Figure 4.13

Figure 4.13: Smallest cells that could possibly contain two points that are a distance $s$ apart in two and three dimensions.

The paths separating the closest points may contain recursive subdivisions until a cell of length smaller than $\frac{s}{\sqrt{d}}$ is reached. Since each subdivision halves the length of the cells, the maximum path length is given by the smallest $k$ for which $\frac{D}{2^k} < \frac{s}{\sqrt{d}}$ or $k = \lceil \log \frac{\sqrt{d}D}{s} \rceil$. For a fixed number of dimensions, the worst-case path length is $O(\log \frac{D}{s})$. Since the tree has $n$ leaves, the number of nodes in the tree is bounded by $O(\log \frac{D}{s})$. In the worst case, $D$ is proportional to the largest distance between any pair of points. Thus, the height of a quadtree is bounded by the logarithm of the ratio of the largest pair wise distance to the smallest pair wise distance. This ratio is a measure of the degree of non-uniformity of the distribution.

Search, insert and delete operations in region quadtrees are rather straightforward. To search for a point, traverse a path from root to a leaf such that each cell on the path encloses the point. If the leaf contains the point, it is in the quadtree. Otherwise, it is not.
**PR_INSERTION (root, A)**

//Suppose a point A is to be inserted.
1. If root = null then
   set Root as A
   go to Step 9
End if

2. If current is not a leaf node
   sqdt = sub-quadrant in which A lies w.r.t the quadrant represented by the current node.
   if child node of current corresponding to the sqdt sub-quadrant is empty then go to step 8
End if
   set parent=current
   set current=child node of current corresponding to sqdt sub-quadrant.
   Go to step 3
End if

3. Temp=current
4. sqdt=subquadrant of parent in which current lies
5. Create a node (n1) referring the sqdt subquadrant.
   Make n1 the child node of parent corresponding to sqt subquadrant
   Set current=n1
   Partition the quadrant corresponding to current node equally into 4 subquadrants
   set sqdt = subquadrant in which A lies after partition
6. If sqdt is same as subquadrant corresponding to temp node ’s data then
   parent=current
   go to step6.
End if

7. Insert temp as the NE or SE or SW or SE child of current node based on the sub-quadrant
   current node in which temp’s data lies.
8. Insert A as the NE or SE or SW or SE(based on sqdt) child of current node.
9: End

**Figure 4.14:** Algorithm for inserting a node in PR-Quad Tree
4.6.1 INSERTION

To insert a point not already in the tree, search for the point which terminates in a leaf. The leaf node corresponds to a region which originally had one point. To insert a new point which also falls within the region, the region is subdivided as many times as necessary until the two points are separated. This may create a chain of zero or more length below the leaf node followed by a branching to separate the two points. To delete a point present in the tree, conduct a search for the point which terminates in a leaf. Delete the leaf node. If deleting the node leaves its parent with a single child, traverse the path from the leaf to the root until a node with at least two children is encountered. Delete the path below the level of the child of this node. Each of the search, insert and delete operations takes O(h) time, where h is the height of the tree. Construction of a quadtree can be done either through repeated insertions, or by constructing the tree level by level starting from the root. In either case, the worst case run time is O(n log D_s).

4.6.2 DELETION

```plaintext
PR_Deletion (root,P)
// P is the Node to be deleted
1. Search for the node P. Record its parent as R if it exists
2. If P has sibling then
3. delete P
4. return
5. Else
6. While R does not have sibling
7. R=father(R)
8. If R can accommodate P as son
9. Attach P as son R
10. Return
11. else
12. Find the best replacement for P and replace it
```

**Figure 4.15: Algorithm for deleting a node in PR-Quad Tree**
4.7 STRUCTURE OF HCQT FRAMEWORK

The main idea in the HCQT (Hilbert Curve Quad Tree) method is to use two consecutive transformations, to map spatial objects into 1-dimensional points. Thus, we can use ANY primary key access method (e.g. B-tree). Good distance-preserving mappings are essential for the performance of the method. Space filling curves, also known as "fractals" are examined in Chapter 3. These curves, such as the Peano curve and the Hilbert curve, define a path that traverses the points in a \( N \times N \) square grid. These curves can be generalized for higher dimensionality spaces.

The HCQT framework method suggests pipelining the transformations of quad tree and Hilbert curve. The resulting framework method enjoys the best of both worlds, avoiding the drawbacks of its individual transformation. It works as follows.

**Step 1.** Divide the region into four quadrants. Sub divide each quadrant into four sub quadrants. Continue this process until each quadrant contains maximum until maximum allowed points in a page. Let us call each quadrant as cell. This transformation is called first transformation. In

**Step 2.** A distance preserving mapping is used to map the points present in each cell to a point in a 1-dimensional space. This transformation is called second transformation.

These two transformations have been applied in Phase I and Phase II which explained in detail in the next section.

4.7.1 HCQT ARCHITECTURE

**Phase I:** The nearest neighbour problem can be solved for transportation network in two steps. In the first step the dataset is preprocessed and stored for querying purposes. The region is divided into smaller region. Most of the methods of division are based on the data points present in the region. Divide the entire region 4 equal squares. Keep on dividing this till each subsquare contains required number of points that can be is approximately equal to \( N = 2^{32} \times 2^{32} \) or number of points that can be stored in a page' Each node in the quadtree contains the data of the form(x,y, width).
Figure 4.16: Flowchart for HCQT framework

PHASE I

START

Translate the region coordinates to the origin (0,0) and set its

GET the number of points in a
Page and set it to P

Divide the current region into 4
equal quadrants. For each i^{th}
region do the following

No Points in i^{th}
quadrant $\leq P$

Translate the each point location
to its H-value and store it

Store the page address and its
length and width of the region
as a leaf node. Increment i

STOP

PHASE II

START

Set the query point

Find the nearest region in which the
query point lies.

Get the address of the page in which
the region is stored

Load the page

Calculate the distance between the
query point and points in the current
Page

Choose the point having the
minimum distance.

STOP
Transform the region current location to the origin to \((0,0)\). Get width and length of the region. Let number of points a page can contain be \(P\). Divide the region in to four equal quadrants. Determine number of points in the first quadrant is less than or equal to \(P\). If so stop further division in this quadrant. Convert the points in the quadrant to its H-value using computational algorithm explained in paper 4. Store the points in the secondary storage device using the H-Value as key. Otherwise divide the quadrant further until the quadrant contain at the maximum \(P\) points.

**Phase II:** Initially the query point and the number of nearest point to be found are taken as input. It can be either inputted or using any pointing device in the map can be given. For example in Google map the point of interest is identified using mouse click on the map. The data set is preprocessed and stored using quad tree and Hilbert value in the secondary storage device. By traversing the quad tree in breadth first search, the region in which the query point lies is found. The region node that is identified has the page address of the region in which the query point is found. Load the page with the help of the address.

The data points are stored according to the Hilbert curve order. The Hilbert curve has the property of locality preserving over Peano or Z-curve. Convert the query point to its H-Value using the calculation mentioned in chapter 3 (reducing from two dimension to one dimension). Find the position of the nearest record. From that the points in its nearest point can be easily calculated and can be retrieved. Repeat the process till required number of points is found. If the page does not contain enough points, identify the current region siblings repeat the process until the desired numbers of points are found.

| Pointer from the quadtree leaf | Position of the quadrant in the entire space | Hilbert labeling of the objects |

Figure 4.18 Hebert node index structure
### 4.7.2 ALGORITHM FOR SEARCHING NEAREST NEIGHBOUR

The algorithm finding the nearest neighbour uses a procedure called LOCATE which finds the node in which the query point lies. In the Quad tree each node contains \(<x,y, \text{width}, \text{NE}, \text{NW}, \text{SE}, \text{SW}>\) where \(x,y\) is the position of top left corner of the square \(\text{width}\) is the length of the side. \(\text{NE}, \text{NW}, \text{SE}, \text{SW}\) point to subsquare of the parent square.

#### LOCATE(Q)

//\(Q(x, y)\) is the query point

1. Convert the point to the region coordinate called \((p,q)\)
   
   Current node = root
   
   Found=false

2. Repeat
   
   If \(p < x+\text{width}\) and \(q < y+\text{width}\)
   
   If current node is a leaf then Found = true
   
   Else
   
   If \((p < x->\text{NW}+\text{width} \text{ and } q < y+N+\text{width})\)
   
   Current node = current node->NW
   
   Else
   
   If \((p < x->\text{NE}+\text{width} \text{ and } q < y+\text{NE}+\text{width})\)
   
   Current node = current node->NE
   
   ELSE
   
   If \((p < x->\text{SW}+\text{width} \text{ and } q < y+\text{SW}+\text{width})\)
   
   Current node = current node->SW
   
   ELSE
   
   Current node = current node->SE
   
   ENDIF

   Until found

3. Return Current node

---

**Figure 4.17: Algorithm for finding the node in which the query point lies**
After identifying the node in which the point Q lies, the nearest neighbour algorithm finds the corresponding page using H-index. Calculate the distance between the points in the page and the query point. Choose the points with the minimum distance. This can be easily extended to find K-nearest neighbour also.

```
Nearest Neighbour (Q, current node)
// Q is the query point
// Current node is the node in which Q lies.
Current node = LOCATE(Q)
Load the page corresponding to the current node.
Find the distance between the query point Q and all the points in the current page.
Choose the point which has minimum distance and return
```

Figure 4.18: Algorithm for finding the nearest neighbour

4.8 CONCLUSION

Time required to search for accessing the region is equal to the height of the tree. Number regions created step 1 will be N/M where N is the total number of points. M is the highest Hilbert number that can be handled easily for encoding and decoding. Inserting a point in Hilbert curve tree structure is equal to the order of the curve called H. This framework is implemented and compared with existing methods which is explained in Chapter 5.