PERFORMANCE COMPARISON OF SPATIAL INDEXING STRUCTURES FOR DIFFERENT QUERY TYPES

by

NEELABH PANT

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2015

To my father, mother and sister

whose support and love have helped me all along.

Acknowledgements

I would like to thank my supervisor and mentor Dr. Ramez Elmasri without whom this thesis wouldn't have ever got completed. His guidance and constant support have helped me to understand the Spatial Database and various indexing techniques in depth. I also thank Dr. Leonidas Fegaras and Dr. Bahram Khalili for their interest in my research and taking time to serve on my dissertation committee.

I would also like to extend my appreciation to the Computer Science and Engineering Department to support me financially in my Master's program.

I would like to thank all the teachers who taught me at the University of Texas at Arlington including Mr. Saravanan Thirumuruganathan for believing in me and for encouraging me to pursue higher education. I would also like to thank Dr. Kulsawasd Jitkajornwanich for his support and encouragement.

I extend my gratitude to all my research mates including Mr. Mohammadhani Fouladgar, Mr. Vivek Sharma, Mr. Surya Swaminathan and everyone else whose support, encouragement and motivation helped me to complete my goals.

Finally, I would like to express my deep gratitude to my father and mother who have inspired and motivated me all the times to achieve my goals. I am extremely fortunate to be so blessed. I am also thankful to my sister for her sacrifice and patience. I extend my love and gratitude to my friend who has always believed in me and made me realize how powerful I am.

April 10, 2015

Abstract

PERFORMANCE COMPARISION OF SPATIAL INDEXING STRUCTURES FOR DIFFERENT QUERY TYPES

Neelabh Pant, M.S.

The University of Texas at Arlington, 2015

Supervising Professor: Ramez Elmasri

R-Trees are among the most popular multidimensional access methods suitable for indexing two dimensional spatial data and point data. R-Trees are found in most of the spatial database systems for indexing the spatial data. The data include points, lines and polygons which are retrieved and stored efficiently. There are many Spatial Database Systems which have incorporated R-Trees, for example, IBM Informix, Oracle Spatial, PostgreSQL and many others.

Another version of R-Tree is R*-Tree which is also used for the same purpose i.e. indexing spatial data. R*-Tree has also been incorporated in an open source software SQLite with an extension of Spatialite.

Several techniques have been proposed to improve the performance of spatial indexes, but none showed the comparative studies in their performance with the different categories of spatial and non-spatial queries.

In this work, we compare the performance of three spatial indexing techniques: R-Tree (Rectangle Tree), GiST (Generalized Search Tree) and R*-Tree (A variant of R-Tree).

We have five categories of spatial and non-spatial queries, namely, Simple SQL, Geometry, Spatial Relationship, Spatial Join and Nearest Neighbor search. We perform extensive experiments in all these five categories and record the execution time.

The spatial data that are used for the experiments is the set of a benchmark data of New York City that include Point data: Subway stations, Line data: Streets and Subway lines, Polygon data: Boroughs and Neighborhoods plus non-spatial data such as Population data: Racially categorized.

The comparison done in the experiments will give the reader performance criteria for selecting the most suitable index structure depending on the types of queries in the application.

Table of Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

The most utilized application in today's time is the Global Positioning System or GPS. A user through such a device views spatial objects like roads, gas stations, cities or continents. Information about the real world data objects are represented spatially in the form of polygons, lines and points which are stored and retrieved from a database. The real world data objects have spatial attributes which are nearly impossible to store in a traditional database, like RDBMS, which are complex and inefficient as the objects are multi-dimensional in nature. Spatial databases are used in location based services, Geographic Information Systems (GIS), environmental modeling and impact assessment, resource management, decision support, data quality and integrity enforcement.

All this information is accessed from the Spatial Database Management System (SDBMS). We need an index to store any data in a database for efficient retrieval. Spatial Database Management Systems make use of spatial indexing structures for fast and efficient access to the data. There are several spatial indexing structures proposed and each of them are good for certain purposes. We in this thesis present three spatial index structures which are compared based upon their performance in different conditions.

We make use of R-Trees (Rectangle Trees), GiST (Generalized Search Tree) and R*-Tree (variant of R-Tree) on two different Spatial Database Management Systems, PostgreSQL and SQLite, which are extended with PostGIS and SQLite respectively which allow them to perform location queries on geographic data. We make use of geographic data in form of shapefile. The shapefile is benchmark data of New York City, of 58,505 records that consists of points, lines, polygons and multipolygons represented by x-y geometry.

The major objective of this thesis is to compare the index performance based upon different categories of queries like simple SQL, spatial relationship, nearest neighbor search, spatial joins and geometry queries. Till now none of the research has shown comparison of the indexes based upon different categories of queries.

The thesis is organized as follows: chapter 2 presents the spatial indexes which include R-Tree, GiST and R*-Tree. Chapter 3 discusses the Spatial Database Management Systems and gives an overview of SDBMS like PostgreSQL with PostGIS and SQLite with SpatiaLite. Chapter 4 explains the spatial queries. We categorize this research in several areas. Section 4.1 presents areas of spatial-models, section 4.2 discusses spatial query language, section 4.3 defines the conceptual design of the database, section 4.4 defines the queries used in performance evaluation along with the spatial functions used. Chapter 5 presents the experimental results and finally chapter 6 concludes the research.

Chapter 2

OVERVIEW OF THE INDEXING STRUCTURES

2.1 Spatial Indexing

With the advancement in computer science and especially in the field of GIS there has been a vast necessity for the abstraction of the spatial data in the form of spatial objects (points, lines and polygons etc.). The efficient storage of and access to this data is the highest priority and is done by using sophisticated data structures. However, traditional indexing methods are not efficient for such purpose, so we will study about R-Tree (Rectangle Tree) which is capable of handling and managing multi-dimensional objects, GiST (Generalized Search Tree) a data structure and an API which is used to build varieties of indexing structures and R*-Tree which is the advanced version and variant of R-Tree which works in the same manner as an R - Tree but has some differences in the method of indexing.

2.2 R-Tree

R-Trees are the extension of B-Trees. They are height balanced and store the objects so that the main intentions, like the rage queries, point queries, nearest neighbor, etc. can be executed efficiently. Spatial objects are stored in such a way that the queries such as e.g. Find all the restaurants within 20 miles from current position are executed within fraction of seconds.

Data sets are often too large to fit in the primary memory of the computer and secondary storage access time is several orders of magnitude slower than the main memory. This is a performance bottleneck because data has to be shipped back and forth from the primary memory to the secondary storage. Thus the goal of good physical database design is to keep this amount of data transfer to an absolute minimum.[1]

Indexes are important for the multidimensional objects to store. The classical indexing structures cannot be used to store them. B-Trees and ISAM cannot be used as the search space is multi-dimensional. Hash table (exact matching) cannot also be used because a range search is required.

2.2.1 R-Tree Index Structure

R-Tree is a height balanced tree just like the B-Tree. It consists of nodes connected to each other where the leaf nodes consist the index records containing pointers to the data objects and the node corresponds to the disk pages.

It stores multi-dimensional rectangles and doesn't transform them to higher dimensional points and also not performs clipping.

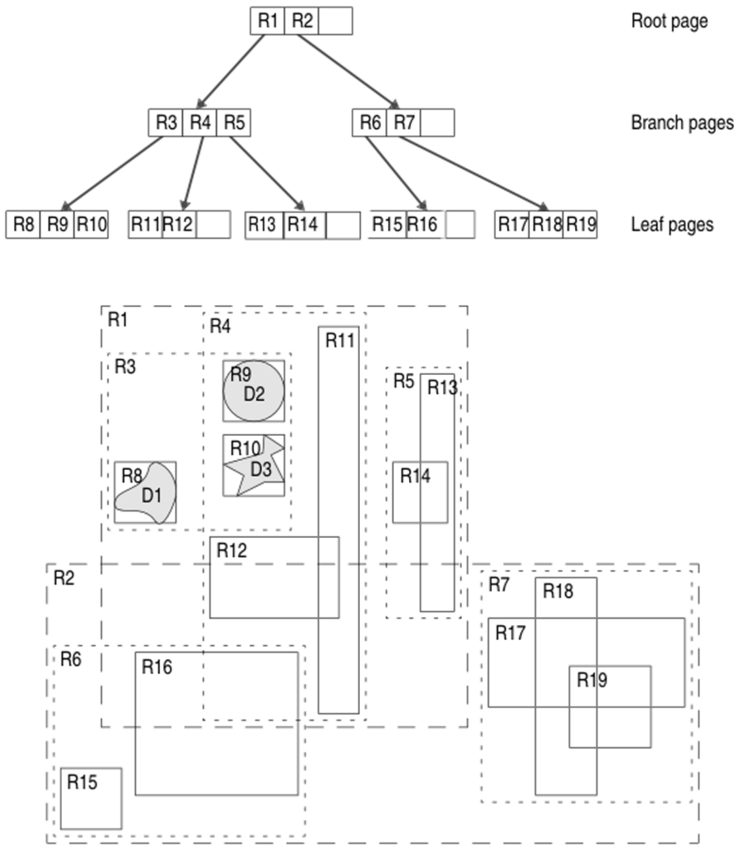


Figure 1: The R-Tree Structure

Spatial Databases have tuples which represent different spatial objects which are identified by a unique tuple-identifier. The purpose of the unique-identifier is to uniquely identify the spatial object stored in there and can be retrieved easily. The leaf nodes of the R-tree contain index record entries of the form (I, tuple-identifier) where I is the Minimum Bounding Rectangle (MBR) which is the bounding box of the spatial object indexed and tuple-identifier is the unique identity of the tuple stored in the database.[2]

2.2.2 R-Tree Properties

Lets consider M to be the maximum number of objects that can be stored in a node and let *m* <= *M/2* be a parameter specifying the minimum number of entries in a node. An R-Tree satisfies the following properties.

1. Every leaf node contains between m and M index records unless it is the root.
2. For each index record *(I, tuple-identifier)* in a leaf node, *I* is the smallest rectangle that spatially contains the n-dimensional data object represented by the indicated tuple.
3. Every non-leaf node has between m and M children, unless it is the root.
4. For each entry *(I, child-pointer)* in a non-leaf node, *I* is the smallest rectangle that spatially contains the rectangles in the child node.
5. The root nodes have at least two children unless it is a leaf.
6. All leaves appear on the same level [1]

2.2.3 Operations on R-Tree

An R-Tree has multiple operations that include Insertion, Search, Deletion. The following will discuss the algorithm for each of the mentioned operations.

Insert: Unlike B-Trees where the new element is to be inserted into the last node of the tree here in R-Tree the new data (object) is inserted in any leaf which has the room to install it. If the node exceeds the given order limit, then the node splits and the tree height and node size are adjusted accordingly.

Algorithm:

```
Choose subtree
    a.  Set N to be the root
    b.  If N is a leaf
                return N
        else
                Choose the entry in N which requires the least area enlargement to
                include the new data. If ties occur, then resolve ties by selecting a
                rectangle of the smallest area.
        end
    c.  The chosen entry, then points to the N (childnode) by the child pointer.
```

Search: Search in R-Tree is unlike B-Tree. Minimum Bounding Rectangles of the internal nodes intersecting with the search rectangle are visited during a search. Consider an example where the user of GIS wants to know the sushi restaurant in a range of 5 miles to his current location. In such queries the R-Tree looks for the person's current location (parent, node) and covers its area within 5miles of radius and looks for the sushi restaurant (object).

Algorithm:

```
Search Subtree
        a.  Set N to be the root
```

```
b. If N is a leaf
        do if entry E in N overlaps the search rectangle
                return E
else
        Each entry E in N,
                if E overlaps the search rectangle then search subtree until the
                entry found which overlaps the search rectangle.
                return E
end
```

Delete: Deletion is a simple and straightforward operation where it removes the index record from an R-Tree. The algorithm looks for the node carrying the object which needs to be deleted. Once found, it removes the element and if required condenses the tree.

Algorithm:

```
a. Find the leaf
        If leaf is null
        return
b. If leaf found with entry to be deleted
        remove entry from the leaf
c. Condense tree
d. If root node has only one child
        Make the child the new root
```

2.3 GiST (Generalized Search Tree)

GiST is a balanced, tree structure template access method which encapsulates basic access methods search and update functions. GiST is used to emulate varieties of tree-structure access methods by providing a small set of custom functions. It is used to navigate the tree by providing a template algorithm. Various functions like delete and node splits are performed in order to modify the structure of the tree [3]. The GiST provides the luxuries of basic search tree logic for a database system. In a single piece of code it encapsulates different tree structures like B+-Tree and R-Tree.

GiST are the third direction for extending the search tree technology. It allows the new data types to be indexed in a manner that supports the queries natural to the type [4]. B+-Trees and R-Trees can be implemented as an extension of the GiST. Single code base yet indexes multiple dissimilar application.

The leaf entry in GiST is of the format (key, RID) where RID refers to the corresponding records on the data pages. For a non-leaf node, (p, ptr) where p is true if it returns the desired values of the tuples reachable from ptr. It is as similar to B+-Trees where it refers to the values representing ranges which bound values of keys in the leaves of the respective subtrees. Another example is the R-Tree which represents MBRs (Minimum Bounding Rectangle) as predicates in the internal nodes.[3]

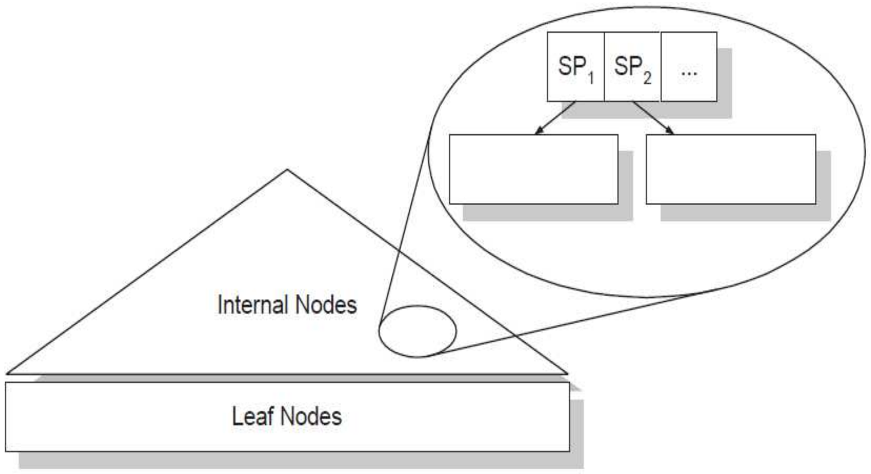


Figure 2: GiST structure.

2.3.1 GiST structure

It is a balanced tree of variable fanout between *kM* and *M* where *k* is the minimum fill factor of the tree, *2/M* <= *k* <= *½*. The exception is the root which may have a fanout between *2* and *M*. The non-leaf nodes are represented as *(p, ptr)* where *p* is the predicate that is used as a search key and *ptr* is a pointer to another tree node. The leaf nodes *(p, ptr)* where key is used as a search key and *ptr* as an identifier of some tuple (indexed datum) in the database.

2.3.2 GiST properties

The following are the properties of a GiST:

1. Unless the node is the root, every node contains between kM and M entries.
2. The root if not a leaf has at least two children.
3. Entries in the leaf node represented by *(p, ptr)*, *p* is true when instantiated with the values from the indicated tuple. In other words key holds for the tuple if pointed by the childpointer of the indicating parent node.

4. Entries in the non-leaf node *(p, ptr) p* is true if the values of any tuple is reachable from *ptr*.
5. All leaves are on the same level.

2.3.3 GiST key methods

1. Search

   Consistent (E, q): Consider an entry *E = (p, ptr)* and a query *q*. The consistent key method asks *E.p^q* whether it satisfies or not, false if unsatisfied otherwise true.

2. Characterization

   Union (P): It returns a new predicate that holds for all tuples in *P*. For example, given a set *P* of entries *(p', ptr'), … ,(p'', ptr'')*, returns a predicate *r* that holds for all the tuples from *ptr'* through *ptr''.*

3. Categorization

   Penalty (E1, E2): Describes the penalty for inserting entry *E2* into entry *E1*. A penalty is basically described as the difference in the enlargement of the area. Thus, *area(Union({E1, E2}) - area(E1.p1) = penalty.*

4. Compression

   Compress (E): If given an entry *E = (p, ptr)* it returns an entry *(π, ptr)* where *π* is the compressed representation of *p*.

5. Decompress (E):

   If a compressed representation is given *E = (π, ptr),* it will return as decompressed entry *(r, ptr)* such that *p→ r*. This is a "lossy" compression. It *is because each document is represented in the index by a fixed signature.* When two words hash to the same bit position there will be a false match.

2.3.4 Operations on GiST

In this section we will describe the different operations GiST method is capable of. Operations like Search, Insert, Delete are almost similar to the R-Tree Index operations with some differences.

1. Search: The search technique is analogous to R-Tree where it takes a query predicate to search any dataset. It traverses the tree until it statisfies the query.

   Algorithm:

   Input: A query predicate *q,* and GiST with the root node N.

   Output: Set of tuples satisfying the query *q*.

   Sketch: Traverse the tree and follow the path until all the tuples satisfying the predicate are found.

   Step 1: [Subtree] If *R* is not leaf, check all the entries *E* in *R* to find whether it is consistent with *q*, i.e. *Consistent (E, q)*. Consistent entries are invoked Search on the subtree whose root node is referenced by *E.ptr*.

   If *R* is leaf node, check each entry *E* in *R* if *Consistent (E, q)*. If consistent return the entry and the *ptr* fetches the actual tuple from the database.

2. Insert: Insertion is very similar to the operation insert on R-Trees. In GiST the operation Insert makes sure that it doesn't violate the balanced nature of the GiST tree. The insertion in GiST allows specification of the level at which to insert. The level number increases as on ascends the tree. Thus, the leaf nodes are at level 0. The new entries will be inserted at level *l = 0*.

Algorithm:

Input: Root node *N*, Level *l*, Entry *E = (p, ptr)*

Output: New GiST after entry *E* is inserted at level *l*

Sketch: Look for an appropriate place for the entry *E* and insert it. If splitting necessary, invoke split.

Step 1: [ChooseSubtree] Choose subtree where the entry *E* is to be inserted. Let *CS = ChooseSubtree (N, E, l)*

Step 2: If space available for *E* on *CS*, insert *E* on *CS*. If not, invoke Split *(N, E, l)*.

Step 3: Propagate changes upwards. *AdjustKeys (N,l).*

3. Delete: The deletion algorithm maintains the balance of the tree. It simply looks for the entry which needs to be deleted by searching for the subtree, it is currently residing at and removes it. It then adjusts the tree if the deletes makes changes to the structure of the tree.

   Algorithm:

   Objective: Remove E from it's leaf node. If this causes underflow, adjust tree accordingly.

   Step 1: [Find node containing entry] Invoke *Search (N, E, p)* and find leaf node *L* containing *E*. Stop if *E* not found.

   Step 2: [Delete entry] Remove *E* from *CS* (subtree).

   Step 3: [Propogate changes] Invoke *CondenseTree (N, CS).*

   Step 4: [Shorten Tree] If the root node has only one child after the tree has been adjusted, make the child as new root.

2.3.5 GiST over R-Trees

In this section we describe the implementation of the key classes to make GiST emulate like R-Tree. In R-Tree the keys are the 4-tuples of reals, which represent the upper-left and lower-right of the Minimum Bounding Rectangle of the 2 dimensional polygons.

The key is defined as *(Xul, Yul, Xlr, Ylr)* where *Xul* = points defining upper left corner of *X* axis, *Yul* = points defining upper left corner of *Y* axis, *Xlr* = points defining upper left corner of *X* axis, *Ylr* = points defining lower left corner of Y axis. The query predicate that is supported are *Contains (box, v)*, *Overlap (box, v)* and *Equal (box, v)*, where box is a 4-tuple as above.

The following is the implementation of the query predicates:

- *Contains* *((Xul1, Yul1, Xlr1, Ylr1), (Xul2, Yul2, Xlr2, Ylr2))*

  Returns true if

  (Xul1<=Xul2) ^ (Yul1 >= Yul2) ^ (Xlr1 >= Xlr2) ^ (Ylr1 <= Ylr2)

- *Overlaps* *((Xul1, Yul1, Xlr1, Ylr1), (Xul2, Yul2, Xlr2, Ylr2))*

  Returns true if

  (Xul1 <= Xlr2) ^ (Yul1 >= Ylr2) ^ (Xul2 <= Xlr1) ^ (Ylr1 <= Yul2)

- *Equals* *((Xul1, Yul1, Xlr1, Ylr1), (Xul2, Yul2, Xlr2, Ylr2))*

  Returns true if

  (Xul1 = Xul2) ^ (Yul1 = Yul2) ^ (Xlr1 = Xlr2) ^ (Ylr1 = Ylr2)

Now, we present the implementation of GiST over R-Trees

- *Consistent (E, q):*

  E = (p, ptr) where *p = (Xul1, Yul1, Xlr1, Ylr1)* and query *q* is either *Contains, Overlaps* or *Equals (Xul2, Yul2, Xlr2, Ylr2)*. For any of these queries the consistent function will return true if *Overlap ((Xul1, Yul1, Xlr1, Ylr1), (Xul2, Yul2, Xlr2, Ylr2)).*

- *Union ({E1, … , En}):*

  The union function returns a new predicate that holds for all tuples. In other words in R-Tree *E1 = ((Xul1, Yul1, Xlr1, Ylr1), ptr1), …, En = ((Xuln, Yuln, Xlrn, Ylrn), ptr)* thus union function returns the maximum bounding rectangles of all the rectangles.

- *Penalty (E1, E2):*

  The penalty function returns the metric value of the change in the area after *E2* is installed in *E1*. Given *E1 = (p1, ptr1)* and *E2 = (p2, ptr2),* compute *q = Union({E1, E2})* and return *area (q) – area (E1.p)*.

- *Compress (E = (p, ptr))*:

  The compress function returns the bounding box of a polygon entry E.p.

- *Decompress (E1 = (Xul1, Yul1, Xlr1, Ylr1))*: The identity function, i.e., return *E*. [4]

2.4 R*-Tree

R*-Tree was proposed by Norbert Beckmann, Hnas-Peter Kriegal, Ralf Schneider and Bernard Seeger in 1990 [5]. It is a variant of R-Tree which is also used to store spatial data such as points, lines and polygons. It has a little higher cost of implementation than the standard R-Tree. The data in R*-Tree may need to be reinserted, but R*-Tree gives a much better performance than the R-Tree.

The major objective of R-Trees is to minimize the area of MBRs but R*-Trees go beyond that. There are more than one criteria that R*-Tree follow such as:

1. **Minimizes of the area covered by each MBR**. The aim is to minimize the area of the dead space, i.e. the area covered by MBR, but not the data rectangles. It is because it tries to reduce the number of paths to travel during query processing.
2. **Minimizes the overlap of the MBRs**. It tries to reduce the overlapping between MBRs, since a large overlapping will cause a large number of paths followed for a query.
3. **Minimizes the margins of the MBR**. Margin is defined as the sum of all sides of an MBR which is also known as perimeter. It aims at shaping more quadratic rectangles to improve the larger quadratic shape query performance. MBRs at the upper levels are smaller as the quadratic objects are packed more easily.
4. **Minimizes the storage utilization**. There is an increase in the number of nodes which are invoked when the utilization is low. Especially for a larger query, where a significant portion of the entries satisfy the queries. Also, whenever the utilization of the node decreases the height of the tree increases.

The above mentioned criteria is followed with an engineering approach as they can become contradictory. For an example, if we keep the area and overlap low, then the entries in a node decreases. Hence, storage utilization may be impacted. Also, by keeping the margin minimized the node overlapping may be increased.

The R*-Tree differs from R-Tree precisely in insertion technique and does not use any specialized deletion algorithm. The deletion in R*-Tree is similar to the original R-Tree deletion algorithm.

2.4.1 Insertion

- ChooseSubtree [Root Node]

  Choose the entry at the root level whose MBR needs least area enlargement to insert the new entry.

- ChooseSubtree [leaf node]

  Here it follows the minimizing the overlap of the MBR criteria where it chooses the entry whose MBR enlargement needs least overlap increase out of all the entries in the node.

2.4.2 Reinsert

In the case where ChooseSubtree cannot find the node with enough space to insert the new entry, it looks for an entry in the node whose centroid distance from the node distance are among the largest 30%. It then reinserts those node(s).

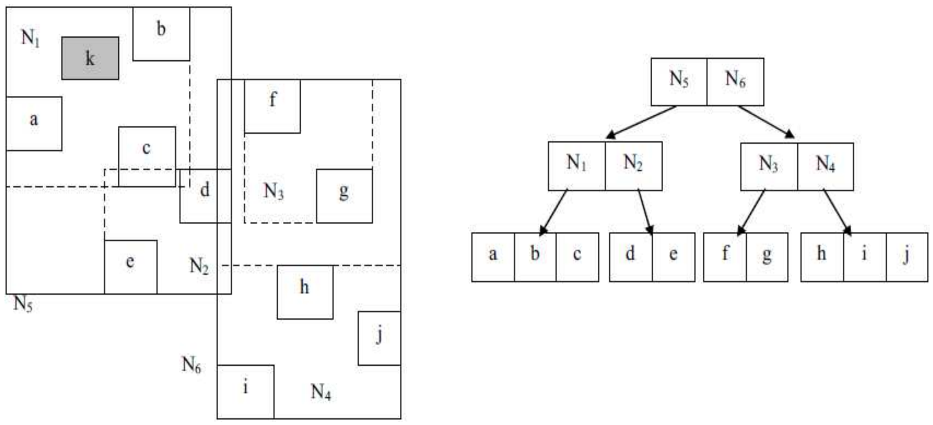


Figure 3: An R*-Tree

If we look at the example given above, we assume that the node N1 is overflowing and the centroid of entry b is the farthest from the centroid of N1 and b is considered for reinsertion. Reinsertion improves the performance during the query processing as it tries to rebalance the tree.

Since, reinsertion is a very costly process, thus it limits to just one reinsertion application per level.

If overflow is unable to be handled by reinsertion then the splitting process takes place. [6]

R*-Tree uses topological split when splitting needs to be done. This method chooses a split axis (smallest overall perimeter and works by sorting all the entries by the coordinates of their left boundaries) based on the perimeter and then minimizes overlap.

Chapter 3

OVERVIEW OF PostGIS AND SpatialLite

3.1 Introduction

The Spatial Database Management System is a collection of spatially referenced data that acts as a model of reality. In its most basic form, the spatial database system is used to store, compute and retrieve spatial objects such as points, lines and polygons. In many applications these days, we require and manage 2D geographic, geometric or spatial data. In many other high level fields we need a storage and retrieval facility for 3D data, such as the human brain or the arrangement of molecular proteins in a human body [12]. Until the advent of SDBMS, relational database systems were used to manage such data in the database system. The purpose of this emerging technology is to manage a large collection of relatively simple geometric objects, for example, 100,000 polygons. This is different from CAD which deals with the geometric entities composed hierarchically into complex structures.[8]

The two kinds of systems “Spatial Database Systems” and “Image Database Systems” are differentiated by the fact that an Image database store, manipulates and retrieves pictures and raster images where as a Spatial database contains objects in space rather than pictures or images of space. These objects have a definite identity, well-defined locations and relationships. There are three requirements for a Spatial Database System:

The first requirement is that Spatial or Geometric information is always connected to non-spatial (alphanumeric) data.

The second requirement is the Spatial Data Types (Points, lines and polygons) provide a fundamental abstraction for modeling the structure of geometric entities in space, as well as their relationships and their operations.

The third requirement is of the Spatial Indexing is that a system is at least able to retrieve data from a large collection of objects without scanning all the objects.

The first requirement sounds of less importance, but emphasizes the fact that a spatial database cannot exist without non-spatial data. In general, the combination of Relational Query Language and Spatial Relationship gives us the Spatial Query. The second point describes the different SDTs and the operations that could be performed. Points, Lines and Polygons are basic geometric structures which provide fundamental abstraction for modeling the geometric structure in space. Finally, the third point explains about the indexing used to store these geometries and how (without scanning the entire index) we can retrieve the data by saving a lot of time [8]. In today's time where we generate trillions of data every hour, we also need some method to store and index them so that we could access them in fractions of seconds.

In this chapter, we will study about two different kinds of spatial database systems that have been used as tools to carry out spatial queries. The spatial databases, we have used are:

1. PostgreSQL Database Management System with an extension of PostGIS, which is an open source, an OGC (Open Geospatial Consortium) complaint spatial database extender. It provides spatial functions like geometry data types, area, distance, intersection, union and much more. [9]

2. SQLite Database Management System which is a lightweight open source database system. We used SpatiaLite, an extender for SQLite database engine with added spatial functions.

3.2 PostgreSQL

PostgreSQL is an Object-Relational Database System (ORDBMS). It was developed at The University of California at Berkeley Computer Science Department. It is an open source which supports the SQL standards and has a lot of modern features to be offered such as:

- Complex Queries
- Foreign Keys
- Triggers
- Updatable Views
- Transactional Integrity
- Multiversion concurrency control [10]

The implementation of POSTGRES began in 1986 under Professor Michael Stonebraker, who was sponsored by Defense Advanced Research Project Agency (DARPA), Army Research Office (ARO), the National Science Foundation (NSF), and ESL Inc.

There have been many different research and production applications which are implemented in POSTGRES. Some of them being:

- Jet engine performance monitoring package
- Financial data analysis system
- Asteroid tracking database
- Medical Information database, and
- Several Geographic Information Systems

3.2.1 PostGIS

PostGIS is a spatial database extender for PostgreSQL object-relational database system. PostGIS allows geographic objects to run location queries in SQL.

```
(SELECT nyc_subway_stations.name
FROM nyc_subway_stations
JOI nyc_neighborhoods
ON ST_Contains(nyc_neighborhoods.geom, nyc_subway_stations.geom)
WHERE nyc_neighborhoods.name = 'Little Italy';)
```

Extra types such as geometry, geography, raster and others are added by PostGIS to PostgreSQL database. These spatial types are added with functions, operators and indexing structures which makes PostgreSQL Database Management System fast, feature-plenty and robust spatial DBMS.

3.2.2 The dataset:

For using PostGIS we needed a spatial dataset preferably a shapefile. The spatial data that are used for the experiments is the set of a benchmark data of New York City that include Point data: Subway stations, Line data: Streets and Subway lines, Polygon data: Boroughs and Neighborhoods plus non-spatial data such as Population data: Racially categorized. We downloaded the shapefile dataset provided to us by workshops.boundlessgeo.com/postgis-intro [11]

There are four tables, namely:

1. *nyc_census_blocks*, which contained 38794 non-spatial population records.

blkid	Unique identity for every census block
popn_total	Total population in the census block
popn_white	Total white population
popn_blck	Total black population
popn_asian	Total Asian population
popn_native	Total native population.
popn_other	Total other population.
boroname	Names of the borough in New York.
geom	Polygon boundary of the block.

2. *nyc_neighborhoods*, which contained 129 polygon records

name	Name of the neighborhoods.
Boroname	Name of all the boroughs in New York.
geom	Polygon boundary of the neighborhood.

3. *nyc_streets*, which contained 19091 line records

name	Street names.
oneway	Is street a oneway? “yes” = yes, “” = no.
type	Type of the street, either primary, secondary, residential or motorway.
geom	Geometry of the line street.

4. nyc_subway_stations

name	Station name

borough	Name of the borough in New York
routes	Subway lines that run through this station
transfers	Lines you can transfer to, via, this station
express	Stations where express trains stop, “express” = yes, “” = no
geom	Point geometry of the station

After creating the database in PostgreSQL with PostGIS extension, we started populating it with the data into the tables specifically.

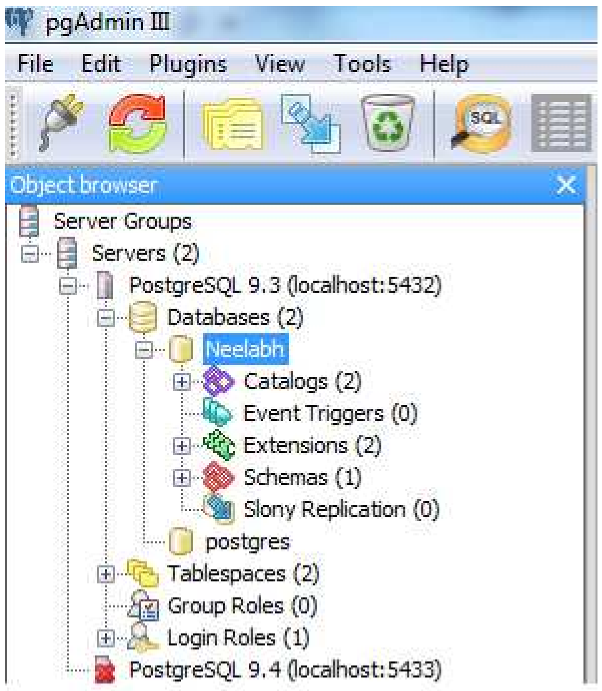


Figure 4: PostgreSQL Database

After creating the database and populating it with the tables mentioned above, it was the time to index the geometry of all the tables.

PostGIS is fully capable of indexing the geometries with various indexing structures like B+-Trees, R-Trees and GiST. We now show the geometries indexing method using R-Tree in PostGIS and the syntax used to do so.

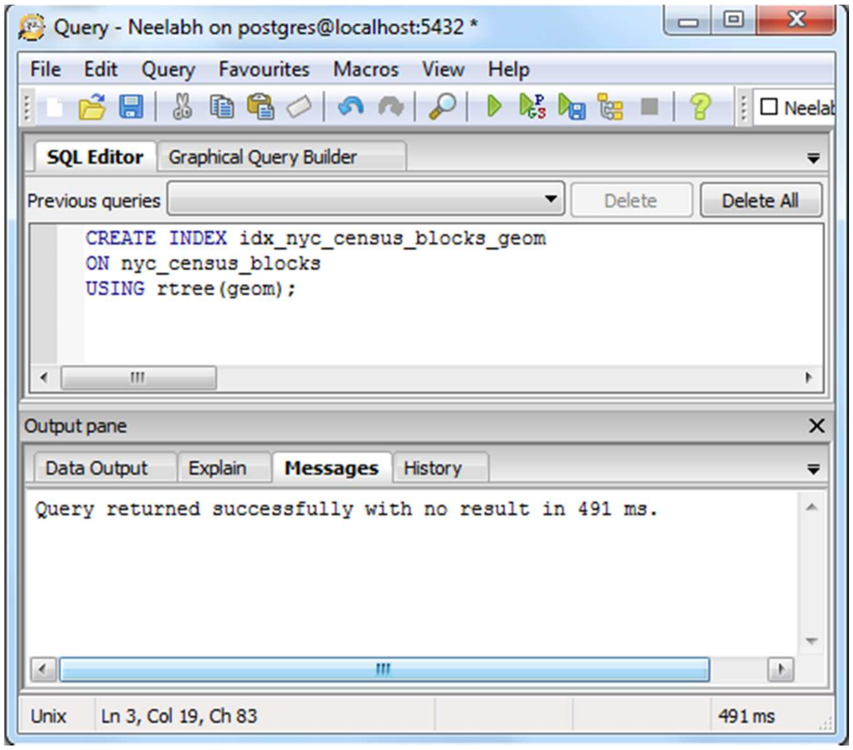
3.2.3 Creating Index

We start indexing the geometries of different tables in the database using R-Tree indexing structure. The PostGIS syntax for creating the index is

CREATE INDEX (“index_name”)

On (“table_name”)

Using rtree (“column_name”);



```
CREATE INDEX idx_nyc_census_blocks_geom
ON nyc_census_blocks
USING rtree(geom);
```

```
Query returned successfully with no result in 491 ms.
```

Figure 5: R-Tree index on geometry column of census blocks

In this way we created an R-Tree index of the geometry column of the rest of the tables.

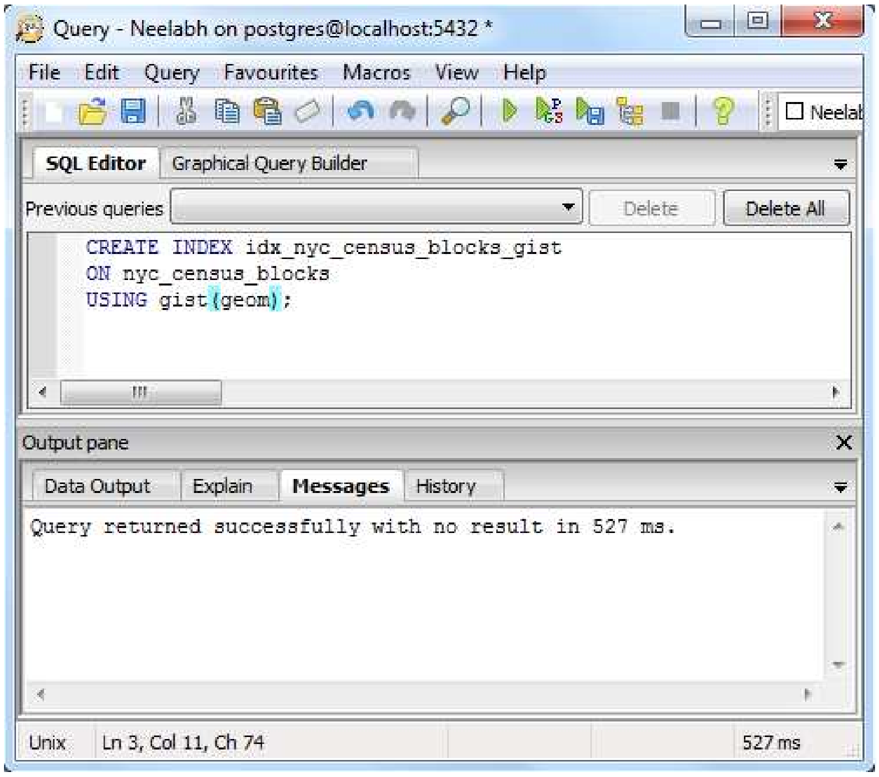
Further, after doing the experiments by running various spatial queries on the tables indexed with R-Tree structure we recorded execution time and dropped the indexes. The spatial queries that we executed are explained in details in the next chapter (Chapter 4). We then indexed the geometries on GiST structure and ran the same set of different spatial queries and recorded the execution time.

The syntax to create the GiST index on the geometry columns in GiST is

CREATE INDEX ("index_name")

ON ("table_name")

USING gist ("geometry_column");



```
CREATE INDEX idx_nyc_census_blocks_gist
ON nyc_census_blocks
USING gist (geom);
```

```
Query returned successfully with no result in 527 ms.
```

Unix Ln 3, Col 11, Ch 74 527 ms

Figure 6: GiST index on geometry column of census blocks

3.3 SQLite

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. It is an open source and is available for free for any purpose. There are numerous applications where SQLite is being used and some of the high-profile projects include

1. Apple – It uses SQLite in many functions like Apple Mail, Safari, Aperture which is deployed on their iPhones, iPods and iTunes software.
2. Google – People are suspicious that Google uses SQLite in a lot of things, including Android cell phones, Google Gears etc.
3. Python – SQLite is bundled up in the python programming language since python 2.5.
4. Firefox – Firefox uses SQLite as its primary metadata storage format.

And other companies like, Microsoft, Adobe, Dropbox use SQLite in many ways like Skype, Photoshop etc.

It was designed by D. Richard Hipp in 2000 [13] while working for General Dynamics on contract with the United States Navy. Hipp was designing a Guided Missile Destroyer software for which he also designed SQLite, which allowed programs to be operated without installing a database management system or requiring a database administrator.

SQLite is an embedded SQL database engine which doesn’t have a separate server process. It reads and writes directly to ordinary disk file and creates an entire SQL database with multiple tables, indices, triggers and views contained in a single disk file. Since, the format of the database file is cross-platform, one can freely copy a database in different architectures. In contrast to other databases, like Oracle, MySQL, PostgreSQL it is not a client-server database engine. Rather, it is embedded into the end program.

3.3.1 SpatiaLite

For our experiments, it was important for us to extend the capabilities of SQLite with SpatiaLite. One can think of SpatiaLite as an added Spatial technology for SQLite similar to what PostGIS does for the PostgreSQL. SpatiaLite is an SQLite database engine with added spatial functions.

Spatialite provides vector geodatabase functionality which is similar to PostGIS, Oracle Spatial and SQL Server with spatial extentions. It is not a client-server architecture, but rather adopts a simpler personal architecture where the entire SQL engine is directly embedded within the application itself. As already mentioned the complete database is an ordinary file which can be transferred and moved between different computers and operating system.

It gives spatial support to SQLite by covering Open Geospatial Consortium (OGC) and Simple Features specifications. Spatialite also adds R*-Tree index support in SQLite and also allows to do advanced spatial queries. It also supports multiple map projections and can also be used as a GIS vector format to exchange geospatial data.

There are various softwares that support spatialite such as:

1. ESRI’s ArcGIS
2. QGIS
3. Autocad MAP
4. Global Mapper

Web servers like

1. GeoDjango
2. Web2py

And tools like

1. GeoTools

2. pyspatialite, which is a python library

3.3.2 Insertion and Indexing

For using Spatialite we needed a spatial dataset preferably a shapefile. The spatial data that are used for the experiments is the same set of a benchmark data of New York City which was also used in PostGIS which include Point data: Subway stations, Line data: Streets and Subway lines, Polygon data: Boroughs and Neighborhoods plus non-spatial data such as Population data: Racially categorized.

All the operations were taken place at the command line interface which is described below:

1. **Insertion**

To insert the data in Spatialite we first execute a SQL script by writing *.read init_spatialite-2.3.sql*. This command initializes the Spatial Metadata where the *.read* macro command executes an SQL script. Next, to insert the tables Spatialite provides *.loadshp* command which is used to import the shapefiles. The command follows with the shapefile's name without *.shp* or *.dbf* extention and allows the user to put desired name of the table. For example, the shapefile of the *nyc_neighborhoods* was entered with the following command:

.loadshp nyc_neighborhoods neighborhoods CP1252 32632 ASCII

Here *nyc_neighborhoods* is the shapefile which is stored in the same directory where Spatialite is stored and neighborhoods is the table name for the shapefile. *CP1252* is the charset name for Windows Latin-1 and *32632* is the SRID. Spatial Reference System Identifier or simply SRID is a unique value which is used to identify projected, unprojected, and local spatial coordinate system definitions.

Thus, this way we inserted all the four shapefiles into four separate tables and named them accordingly. *nyc_neighborhoods* as neighborhoods, *nyc_census_blocks* as censusblocks, *nyc_streets as streets* and *nyc_subway_stations* as subwaystations

2. **Indexing**

After we got all the four tables in the database (*nyc_db.sqlite*) it was the time to index their geometries on R*-Tree index. As already discussed Spatialite adds R*-Tree index feature as an additional support to SQLite so that one can index the geometries on R*-Tree index. The only problem was that most of the database systems have direct queries like:

CREATE INDEX "index_name" ON ("table_name", "geometry_column");

This is because the *CREATE INDEX* represents an implementation of B-Tree. In this case B-Tree is not the choice of index as B-Tree is only capable to doing the comparative search like lesser than <, greater than > or equals = but not for geometries. For geometries we needed R*-Tree index. Thus, the R*-Tree module in Spatialite first creates a virtual table which has an odd number of columns between 3 and 11. The first column is always a 64 bit signed integer primary key and the other columns are represented as dimension pairs, where each pair is one dimension. The pairs describe the minimum and maximum value for that dimension. The SQL statement that creates the R*-Tree index is

SELECT CreatSpatialIndex ('neighborhoods', 'geom');

Where neighborhoods is the table name and *geom* is the geometry column. The *geometry_column* in the database stores the spatial information about all the tables in the database. If we take a closer look in the *geometry_column* by typing *SELECT * FROM geometry_column;* we get

Table 1: geometry_column

name	*geometry*	*Type*	*Coord*	*Srid*	*Spatial_index_enabled*
neighborhoods	*geom*	*MULTIPOLYGON*	*2*	*32632*	*1*
census_blocks	*geom*	*MULTIPOLYGON*	*2*	*32632*	*1*
streets	*geom*	*MULTIPOLYGON*	*2*	*32632*	*1*
subwaystations	*geom*	*MULTIPOLYGON*	*2*	*32632*	*1*

Hence, the geometry_column describes the spatial features of all the tables in the database. The most important of all the columns in *geometry_column* table is the *Spatial_index_enabled* column which lets the user know whether the spatial index is enabled (1) or not (0).

3.4 Enabling index execution time

Enabling index on the geometry columns takes time depending on the type of index. Some indexes take a short time, whereas, others take more than twice the standard times.

We now record the time taken by the indexing trees, namely, R-Tree, GiST and R*-Tree to implement in the geometry of the benchmark data set. Since, the shapefile we used has been kept the same for all the three indexing structures, thus then comparison gives a clear picture to the reader.

We started indexing the geometries of the different tables in the database. Below is the table with the average execution time for the different indexes implementation on the geometry columns of *nyc_census_blocks*, *nyc_neighborhoods*, *nyc_streets* and *nyc_subway_stations*

Table 2: Time taken by each index

Indexes	*Nyc_Census_blocks*	*Nyc_Neighborhoods*	*Nyc_Streets*	*Nyc_Subway_stations*
R-Tree*	*2.842*	*1.245*	*2.606*	*1.332*
GiST	*2.37*	*0.231*	*0.594*	*0.11*
R-Tree	*0.862*	*0.25*	*0.345*	*0.032*

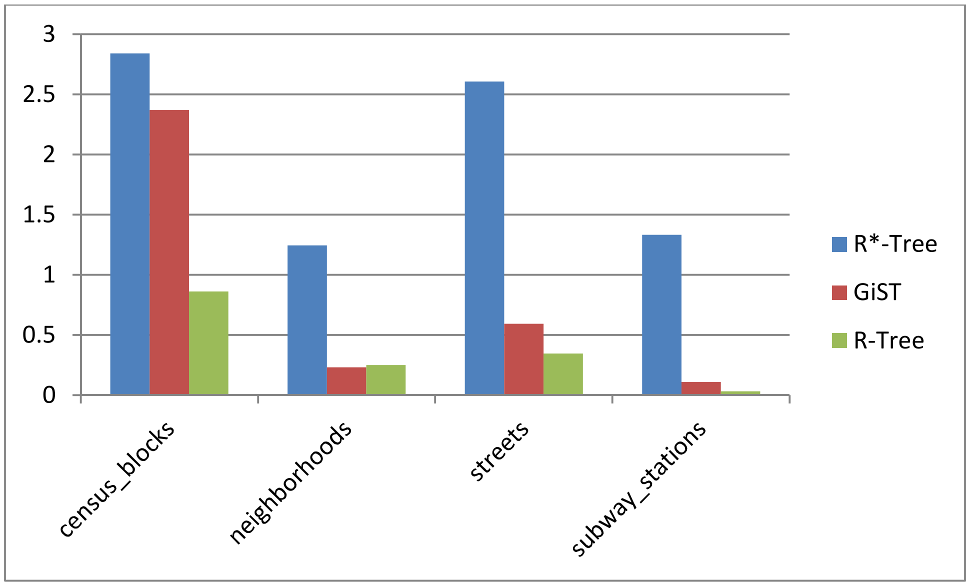


Figure 7: Graph of the time taken by each index

So, clearly R*-Tree takes more than twice of the time as compared to other indexing techniques to implement on the geometry.

In Chapter 2, it is already mentioned that R*-Tree has a little higher cost of implementation than the R-Trees and GiSTs but it gives a much better performance than the other two. We will testify the performance of the indexing techniques based on different criteria in later chapters.

Chapter 4

SPATIAL QUERIES

4.1 Areas of Spatial Models

The only reason for the invention of and so much research based on SDBMS is to have a deeper analysis of space and the objects of which it is made. Spatial data is a term used to describe data that pertain to the space occupied by objects in a database. It is the geometric data like points, lines, rectangles, polygons and time, that have non-spatial attributes, e.g. names of all the rivers, coordinates of a particular city, etc.

There are two types of spatial models: Object model and field model.

1. **Object model**: Object-modeling abstracts the spatial information in distinct, identifiable entities called objects. These objects have specific area and are represented by coordinates. Each object has a set of attributes such as name, address, coordinates, shape, etc., which are stored non-spatially in the database.

2. **Field model**: Data that is spread over a region and which is defined by its continuity is known as field model. It is continuous in nature and has function values. Field models do not have a specific value, but change with respect to time or space. It sees the world as a continuous surface over which features vary, using object-based spatial database. [7]

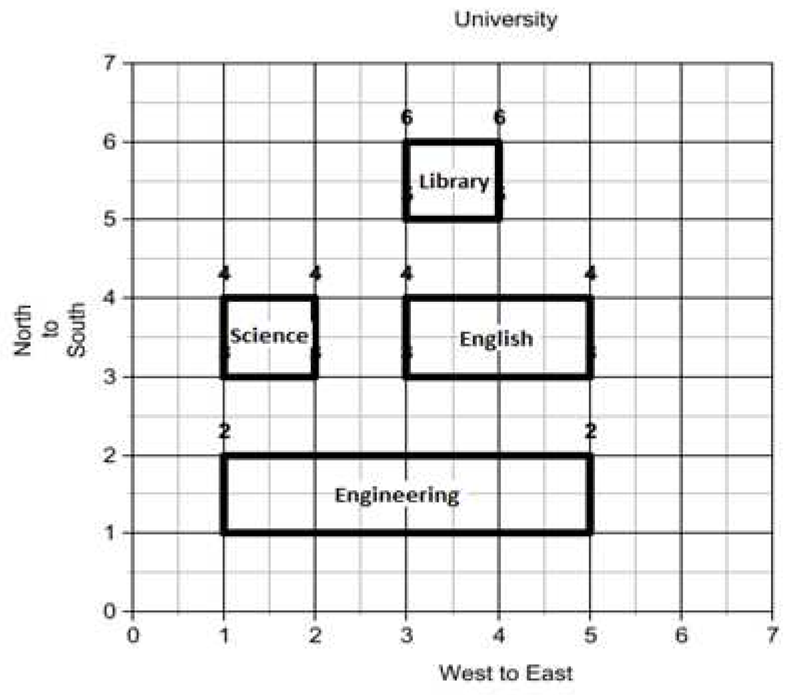


Figure 8: Department Location

Table 3: Object Viewpoint of University

Area-ID	*Department*	*Area/Boundary*
UN1	*Engineering*	*[(1,1),(5,1),(5,2),(2,2)]*
UN2	*English*	*[(3,3),(2,3),(5,4),(4,4)]*
UN3	*Science*	*[(1,3),(2,3),(2,4),(1,4)]*
UN4	*Library*	*[(3,5),(4,5),(4,6),(3,6)]*

Table 4: Field Viewpoint of University

f(x,y)	*"Library," $3 \le x \le 4$; $5 \le y \le 6$*
f(x,y)	*"English," $3 \le x \le 5$; $3 \le y \le 4$*
f(x,y)	*"Science," $1 \le x \le 2$; $1 \le y \le 4$*
f(x,y)	*"Engineering," $1 \le x \le 5$; $1 \le y \le 2$*

In the functional viewpoint, the University is modeled as a function where the domain is the underlying geographic space of the University and the range is a set consisting of four elements-the names of the different buildings in the campus. The functional values are shown by a range of values of the x-y coordinates. Field model is also represented by a piecewise function or by the use of grids. The grids make use of the cells or pixels and have their precise coordinates. It is independent of the longitudes and latitudes, but, takes its center as the reference.

In the object model, the buildings shown in the example have a definite starting and ending point in the x-y coordinate structure. The objects are represented as independent entities with some definite points of access such as longitude and latitude. We get the demarcation clearly defined in the University model shown above, thus get a specific boundary of the polygons. Each polygon has a unique identifier and non-spatial attributes. [12]

The use of field models and object models depends on the requirement of a map developer. For example, in a map where the developer is supposed to define the region of interest in a city, the roads and building shown would be the lines and polygons (object model), whereas the density of people or vehicles stopping by to a specific part of the city would be shown by different colors (field model).

4.2 Spatial Query Language

Spatial Query Language is a database language which is developed to query spatial features using the traditional Structured Query Language (SQL). It is a normal extension to the SQL where the traditional relational query language is packed with the spatial relationships which in turn gives spatial query language. Spatial Query Language helps the user to retrieve and display the queries [13] by the use of query language to retrieve data and by the presentation language i.e Graphical Presentation Language (GPL). Basically, we do not develop an exclusive spatial system, but integrate spatial query attributes and operations to Structured Query Language [14]

We discussed the definition and the differences between field based models (raster data) and object based models (vector data). Table 4 describes the different operations that can take place over the data models based upon different queries. [15]

Table 5: Data model and operation

Data Model	*Operator Group*	*Operation*
Vector Object	*Set-oriented*	*equals, is a member of, is empty, is a subset of, is disjoint, from, intersection, union, diffrence, cardinality*
	Topological	*boundary, interior, closure, meets, overlaps, is inside, covers, connected, components, extremes, is within*
	Metric	*distance, bearing/angle, length, area, perimeter*
	Direction	*east, west north, south*
	Network	*successors, ancestors, connected, shortest-path*
	Dynamic	*translate, rotate, sclae, shear, split, merge*
Ratser Field	*Local*	*point-wise sums, differences, maximums, means, etc*
	Focal	*slope, aspect, weighted average of neighborhoods*
	Zonal	*sum or mean or maximum of field values in each zones*

Different Spatial Query Operations can be classified into following major groups

- Update operation: Includes standard database operations such as create, modify and update.
- Spatial selection: Contains the following operations:
    - Point query: Find all rectangles containing given point.
    - Range query: Find all points within a query rectangle.

- Nearest neighbor: Find all lines which intersect a query rectangle.
- Distance scan: Enumerate points in increasing distance from a query point.
- Intersection query: Find all the rectangles and polygons intersecting a query rectangle.
- Containment query: Find all the rectangles or polygons within a query rectangle. [16]

For the object data models, important spatial functions are those which determine the dimension of an object such as boundary and interior. Such operations come under the topological section and much future research is needed in the other fields of operator group. There are various queries that we performed in our experiment over the benchmark dataset using PostGIS and Spatialite. Both the database performed the same set of queries, but with a slight difference in terms of using the functions. There are various spatial functions that an SDBMS support. Table 5 describes different sets of functions and their use in a spatial database. Each function falls under a certain category which has a definite role while querying a database.

Table 6: Operations listed in the OGC standard for SQL

Basic Functions	
SpatialReference()	Returns the underlying coordinate geometry
Envelope()	Returns the minimum orthogonal bounding rectangle of the geometry
Export()	Returns the geometry in a different representation
IsEmpty()	Returns true if the geometry is an empty set
IsSimple	Returns true if the geometry is simple (no self-intersection)

Table 6- continued

Boundary()	Returns the boundary of the geometry
Topological/ Set Operators	
Equal	Returns true if the interior and the boundary of the two geometries are spatially equal
Disjoint	Returns true if the boundaries and interior do not intersect
Intersect	Returns true if the interiors of the geometries intersect
Touch	Returns true if the boundaries intersect but the interiors do not
Cross	Returns true if the interiors of the geometries intersect but the boundaries do not
Within	Returns true if the interior of the given geometry does not intersect with the exterior of another geometry
Contains	Tests if the given geometry contains another geometry
Overlaps	Returns true if the interiors of two geometries have non-empty intersection
Spatial Analysis	
Distance	Returns the shortest distance between two geometries
Buffer	Returns a geometry that consists of all points whose distance from the given geometry is less than or equal to the distance
ConvexHull	Returns the smallest convex set enclosing the geometry
Intersection	Returns the geometric intersection of two geometries
Union	Returns the geometric union of two geometries
Difference	Returns the portion of a geometry which does not intersect with another given geometry
SymmDiff	Returns the portion of two geometries which do not intersect with each other

4.3 Conceptual Design of the Database

The pictogram-enhanced ER diagram is shown in Figure 9. The Borough, Neighborhoods and Census_Blocks are encoded with polygon pictograms, Streets with line pictogram and subway_stations with point pictogram. The different relations between different entities are also shown. It is clear from the figure that the pictograms enhance the spatial semantics conveyed by the ER diagram.

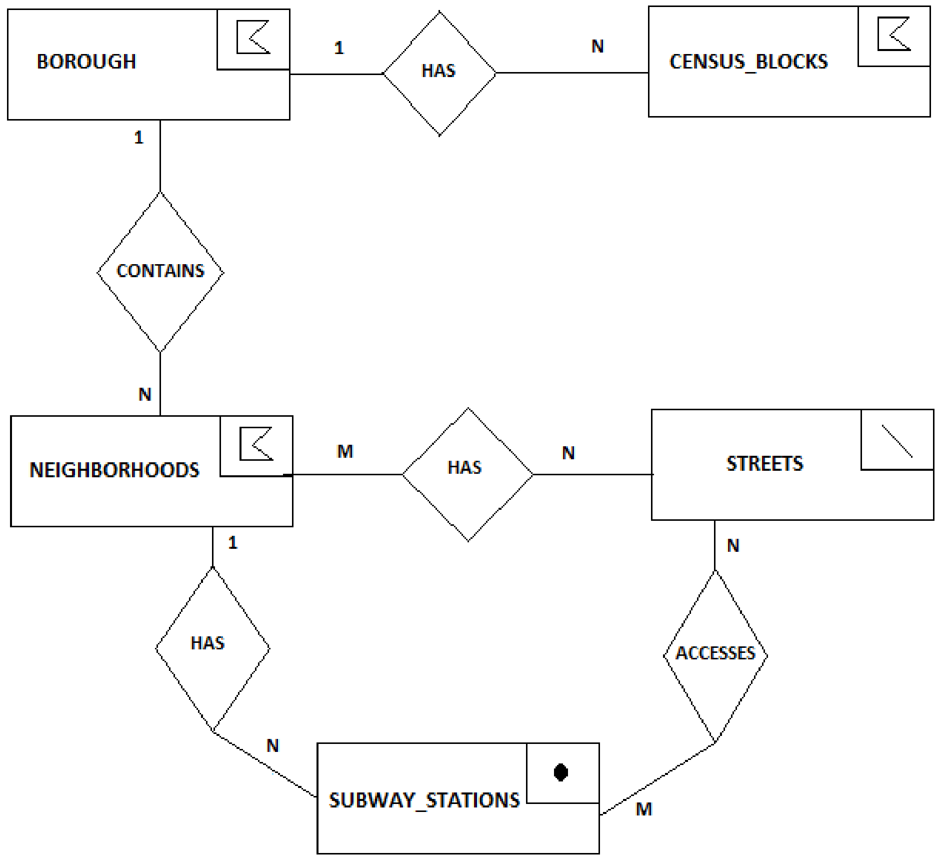


Figure 9: ER diagram for the nyc_dataset, with pictograms

4.4 Queries Used in Performance Evaluation

As already discussed in section 4.2, in order to execute spatial queries on our database we used many spatial functions. In this section we will go through the different queries and will discuss the spatial function we used in order to execute them

The spatial queries we will present will be divided according to the categories they fall into.[11]. We will denote each query with a certain variable (Q1, Q2, …, Qn) so that later in this thesis we can refer to each query just by the variable.

Simple SQL:

Q1: Select name from nyc_neighborhoods

```
SELECT name
FROM nyc_neighborhoods;
```

Q2: Select all the neighborhood names which are under ‘Manhattan’ borough.

```
SELECT name
FROM nyc_neighborhoods
WHERE boroname = ‘Manhattan’;
```

Q3: Find number of letters in all the neighborhood names in Brooklyn.

```
SELECT char_length(name)
FROM nyc_neighborhoods
WHERE boroname = 'Brooklyn';
```

Q4: What is the population of the city of New York?

```
SELECT Sum(popn_total) AS population
FROM nyc_census_blocks;
```

Q5: Find the total population of the borough The Bronx.

```
SELECT Sum (popn_total) AS population
FROM nyc_census_blocks
```

```
WHERE boroname = 'The Bronx';
```

Q6: Find the percentage of white people for each borough.

```
SELECT boroname
,100*Sum(popn_white)/Sum(popn_total) AS white_pct
FROM nyc_census_blocks
GROUP BY boroname;
```

Functions used in the simple SQL exercise

F1: *Average( )*

The function *Average( )* in PostGIS returns the average value of the numeric column.

F2: *Char_length( )*

The function *char_length( )* in PostGIS counts the length of the characters in the column.

F3: *Sum( )*

The *sum( )* function in PostGIS returns the sum of records in a set of records.

Geometry:

Q7: Compute the area of the 'West Village' neighborhood.

```
SELECT ST_Area(geom)
FROM nyc_neighborhoods
WHERE name = 'West Village';
```

Q8: Compute the area of 'Manhattan' in acres. (The unit given to us in the data is in meters

```
SELECT Sum(ST_Area(geom)) / 4047
FROM nyc_neighborhoods
WHERE boroname = 'Manhattan';
```

Q9: Compute the number of the census blocks with hole in New York City

```
SELECT Count(*)
FROM nyc_census_blocks
WHERE ST_NumInteriorRings(ST_GeometryN(geom,1)) > 0;
```

Q10: Find the total length of all the streets in New York City in Kilometers.

```
SELECT Sum (ST_Length(geom)) / 1000
FROM nyc_streets;
```

Q11: Find the length of the street 'Columbus Cir'.

```
SELECT ST_Length(geom)
FROM nyc_streets
WHERE name = 'Columbus Cir';
```

Q12: What is the JSON representation of the boundary of 'West Village'?

```
SELECT ST_AsGeoJSON(geom)
FROM nyc_neighborhoods
WHERE name = 'West Village';
```

Q13: Summarized by the type, calculate the length of the streets in New York.

SELECT type, Sum(ST_Length(geom)) AS length

FROM nyc_streets

GROUP BY type

ORDER BY length DESC;

Functions used in the Geometry exercise

F4: *ST_Area*

The function *ST_Area* in PostGIS returns the area of the surface if it is a polygon or a multipolygon.

F5: *ST_AsGeoJSON()*

The function *ST_AsGeoJSON* in PostGIS returns the geometry as a GeoJSON element.

F6: *ST_GeometryN*

The *ST_GeometryN* function in PostGIS returns the 1-based Nth geometry if the geometry is a GEOMETRYCOLLECTION, MULTIPOINT, MULTILINESTRING, MULTICURVE OR MULTIPOLYGON. Otherwise, return NULL.

Spatial relationship:

Q14: What is the neighborhood of the ‘Broad St Subway Station’?

SELECT name, ST_AsText(geom)

FROM nyc_subway_stations

WHERE name = 'Broad St';

(Which returns POINT (583571 4506714))

SELECT name, boroname

FROM nyc_neighborhoods

```
WHERE ST_Intersects(geom, ST_GeomFromText ('POINT(583571
4506714)',26918));
```

Q15: For the street named 'W Lake Dr find the geometry value.

```
SELECT ST_AsText(geom)
FROM nyc_streets
WHERE name = 'W Lake Dr';
```

Q16: Find the neighborhood and borough of 'W Lake Dr'.

```
SELECT name, boroname
FROM nyc_neighborhoods
WHERE ST_Intersects(
geom,
ST_GeomFromText('LINESTRING(586812 4501262,586811 4501142)',
26918));
```

Q17: Find the street which joins 'W Lake Dr' .

```
SELECT name
FROM nyc_streets
WHERE ST_DWithin(
geom,
ST_GeomFromText('LINESTRING(586782 4504202,586864 4504216)',
26918),
 0.1
);
```

(Here, 0.1 at the end is the distance in meters which says, find the street which is within distance 0.1 meters from W Lake Dr.)

Q18: Find the total number of people who live within 50 meters of ‘W Lake Dr’

```
SELECT Sum(popn_total)
FROM nyc_census_blocks
WHERE ST_DWithin(
geom,
ST_GeomFromText('LINESTRING(586782 4504202,586864 4504216)',
26918),
50
);
```

Functions used in the Spatial Relation exercise

F7: *ST_AsText( )*

The function ST_AsText( ) in PostGIS returns the Well-Known Text (WKT) representation of the geometry/geography without SRID metadata.

F8: *ST_GeomFromText( )*

The function ST_GeomFromText( ) in PostGIS returns a specified Geometry value from Well-Known Text representation (WKT).

F9: *ST_DWithin(geometry A, geometry B, radius)*

The *ST_DWithin(geometry A, geometry B, radius)* function in PostGIS returns true if the geometries are within the specified distance (radius) of one another.

F10: *ST_Intersects(geometry A, geometry B)*

Returns TRUE if the geometries/geography “spatially intersect” – (share any portion of space) and FALSE if they don’t (they are disjoint).

Spatial Joins: Spatial Joins allow a user to combine information from different tables by using spatial relationships. It matches rows from the join features to the target feature based on their spatial relative location.

Q19: Find the distance between ‘Columbus Cir’ and ‘Fulton Ave’.

```
SELECT ST_Distance(
ST_GeomFromText(
(SELECT ST_AsText(geom)
FROM nyc_streets
WHERE name = 'Columbus Cir'), 26918),
ST_GeomFromText(
(SELECT ST_AsText(geom)
FROM nyc_streets
WHERE name = 'Fulton Ave'), 26918)
)/1000 as Distance_in_Kms;
```

If we look carefully, in this query the user first tries to find WKT representation of Columbus Cir and Fulton Ave, then, by using the function ST_Distance calculates the distance between them.

Q20: Find the neighborhood of ‘South Ferry’ subway station.

```
SELECT
nyc_subway_stations.name,
nyc_neighborhoods.name,
nyc_neighborhoods.boroname
FROM nyc_neighborhoods
JOIN nyc_subway_stations
ON ST_Contains(nyc_neighborhoods.geom,
nyc_subway_stations.geom)
WHERE nyc_subway_stations.name = 'South Ferry';
```

Q21: What is the population and racial make-up of the neighborhoods of Manhattan?

```
SELECT
nyc_neighborhoods.name,
Sum (nyc_census_blocks.popn_total),
100.0 * Sum(nyc_census_blocks.popn_white) /
Sum(nyc_census_blocks.popn_total),
100.0 * Sum(nyc_census_blocks.popn_black) /
Sum(nyc_census_blocks.popn_total)
FROM nyc_neighborhoods
JOIN nyc_census_blocks
ON
ST_Intersects(nyc_neighborhoods.geom, nyc_census_blocks.geom)
WHERE nyc_neighborhoods.boroname = 'Manhattan'
GROUP BY nyc_neighborhoods.name
ORDER BY white_pct DESC;
```

Functions used in the Spatial Join exercise

F11: *ST_Contains(geometry A, geometry B)*

The function *ST_Contains( )* in PostGIS returns true if and only if no points of B lie in the exterior of A, and at least one point of the interior of B lie in the interior of A.

F12: *ST_Distance(geometry A, geometry B)*

The function *ST_Distance(geometry A, geometry B)* in PostGIS returns 2-dimensional Cartesian minimum distance between two geometries.

Nearest Neighborhoods:

Q22: What subway station is in ‘Bensonhurst’?

```
SELECT s.name, s.routes
FROM nyc_subway_stations AS s
JOIN nyc_neighborhoods AS n
ON ST_Contains(n.geom, s.geom)
WHERE n.name = 'Bensonhurst';
```

Q23: What is the closest street to ‘Cortlandt’ subway station?

```
SELECT streets.gid, streets.name
FROM
nyc_streets streets,
nyc_subway_stations subways
WHERE subways.name = 'Cortlandt'
ORDER BY ST_Distance(streets.geom, subways.geom)
ASC
LIMIT 1;
```

List of some more important functions in Spatial Database Management System

F13: *ST_Length*

The function *ST_Length* in PostGIS returns the 2d length of the geometry if it is a linestring or multilinestring.

F8: *ST_Perimeter*

The function *ST_Perimeter* returns the total length of the boundary of the polygon or multipolygon.

F9: *ST_X*

It returns the X coordinate of the point.

F10: *ST_Y*

It returns the Y coordinate of the point.

F11: *ST_Crosses (geometry A, geometry B)*

The function *ST_Crosses (geometry A, geometry B)* returns TRUE if geometry A and geometry B have some interior points in common.

F12: *ST_Disjoint (geometry A, geometry B)*

The function *ST_Disjoint (geometry A, geometry B)* returns TRUE if the geometries do not spatially intersect.

F13: *ST_Equals (geometry A, geometry B)*

If both the geometry A and B represent the same geometry regardless of their direction then the function *ST_Equals (geometry A, geometry B)* returns TRUE.

F14: *ST_Overlaps (Geometry A, Geometry B)*

The function *ST_Overlaps (Geometry A, Geometry B)* returns TRUE if both the geometries have same dimension, and share space but are not completely contained by each other.

F15: *ST_Touches (Geometry A, Geometry B)*

The function *ST_Touches (Geometry A, Geometry B)* returns TRUE if the interiors of the geometries do not intersect but have at least one point in common.

F16: *ST_Within (Geometry A, Geometry B)*

The function *ST_Within (Geometry A, Geometry B)* returns TRUE if geometry A is completely inside geometry B.

Chapter 5

EXPERIMENTAL RESULTS

5.1 Platform

In our experimental evaluation, we present the methodologies for our experiments. We used Intel core i7 2.8 GHz CPU with 4GB memory on Windows 7 64-bit operating system. We performed the experiments on PostgreSQL with an extension of PostGIS with R-Tree and GiST indexing and SQLite with an extension of Spatialite which helped us to index the geometries on R*-Tree indexing structure. As, already discussed in Chapter 4, the data we used was a benchmark dataset of New York City which had a total of 58505 records consisting of 19091 lines, 129 polygons, 491 points and 38794 non-spatial data with multipolygon geometries.

5.2 Methodology

We ran a set of 23 queries of different categories. In the last chapter, we divided the queries according to their categories and here we will identify them by their variable which are already defined in section 4.4. In the experiment we ran each set of queries on the database without index and on R-Tree, GiST and R*_tree indexed database and finally recorded their execution time. Here we will describe the execution time for each query by histograms which will help us evaluate the performance of each index under different categories.

In the next section we will see the execution time in milliseconds for each category without index and indexed in all three indexing structures.

5.3 Time taken

In this section we will describe the execution time of each category with the help of histograms. Each bar of the histogram will represent the time taken (in ms) by each query. We will start from the first category i.e. Simple SQL where we will show the execution time of the queries without and with indexes.

5.3.1 Simple SQL

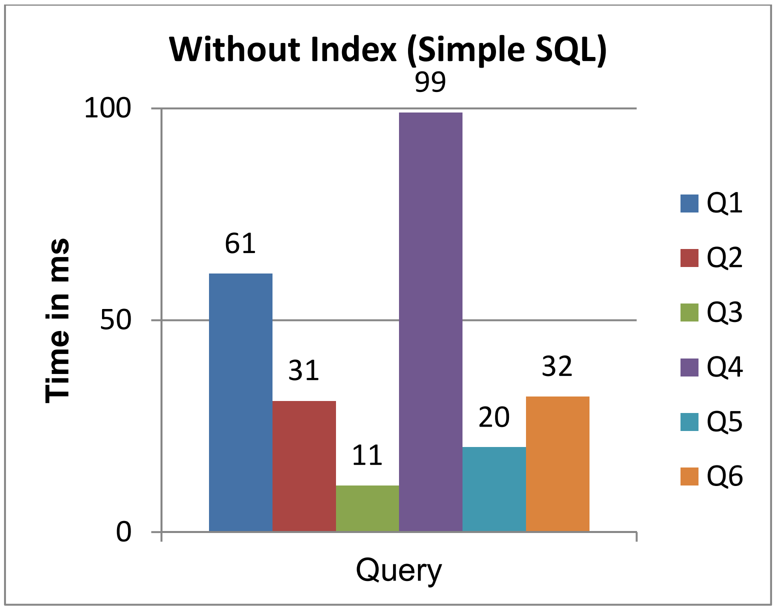


Figure 10: Time taken without Index (Simple SQL)

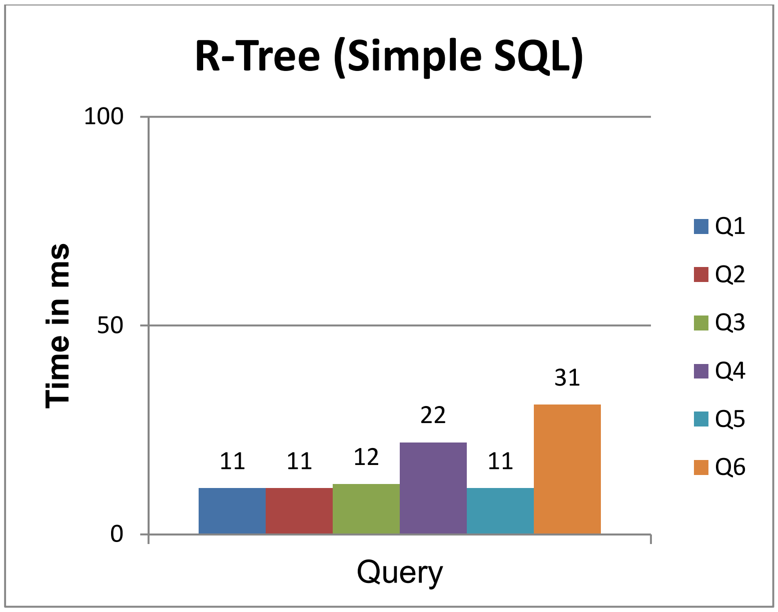


Figure 11: Time taken by R-Tree Index (Simple SQL)

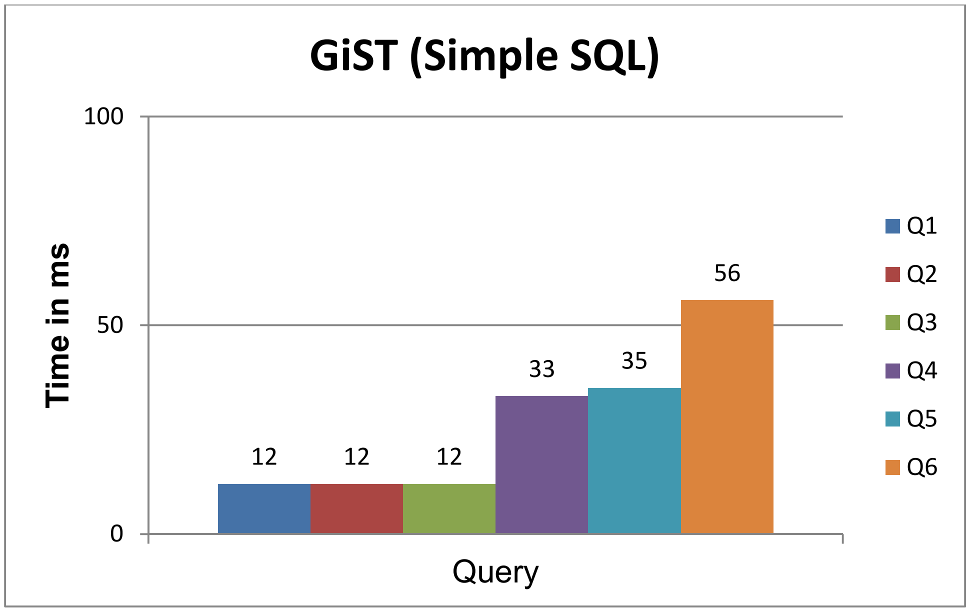


Figure 12: Time taken by GiST index (Simple SQL)

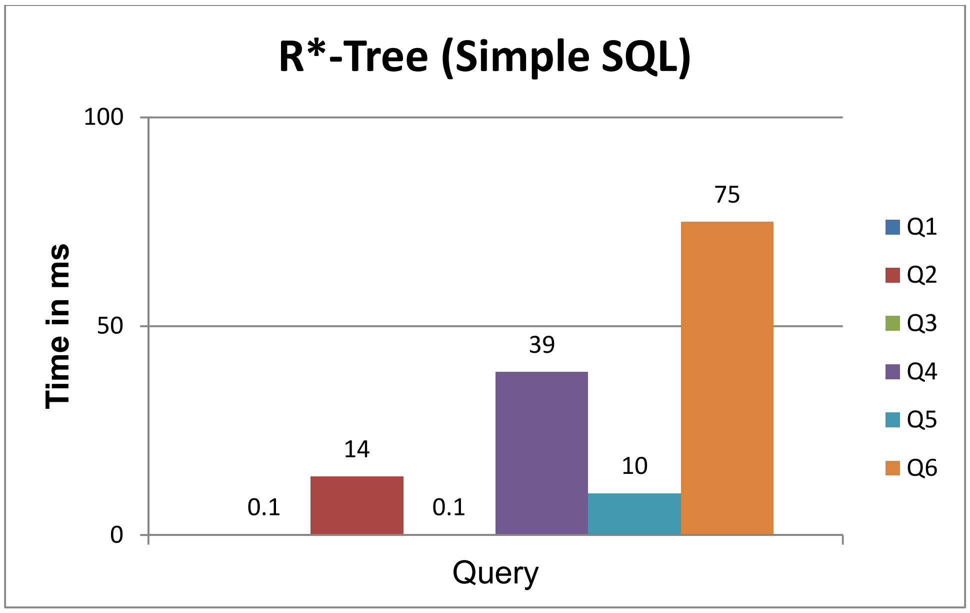


Figure 13: Time taken by R*-Tree index (Simple SQL)

By all the histograms presented above, we are not able to distinguish clearly about the best method to follow when executing the Simple SQL queries. Lets, try putting the average time in a histogram so that it is easier to find the most efficient indexing structure for Simple SQL.

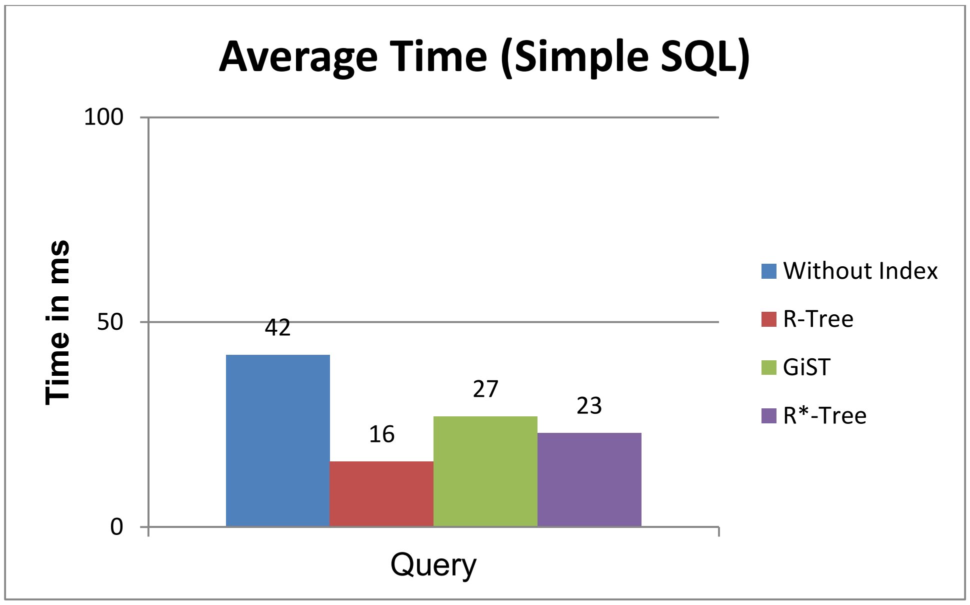


Figure 14: Average time taken by all the indexing structures (Simple SQL)

Clearly, R-Tree is the least time consuming indexing structure. R*-Tree is another good option after R-Tree but since R*-Tree in our experiments didn’t give good performance in executing arithmetic operations like sum/ multiplication/ division, thus, we conclude that R*-Tree is not a good choice of indexing whenever we have to perform arithmetic operations in the queries.

5.3.2 Geometry

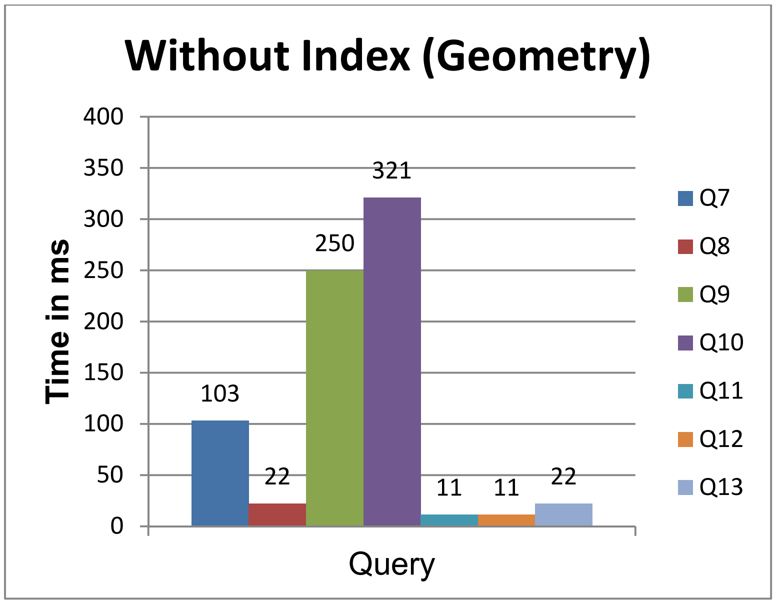


Figure 15: Time taken without Index (Geometry)

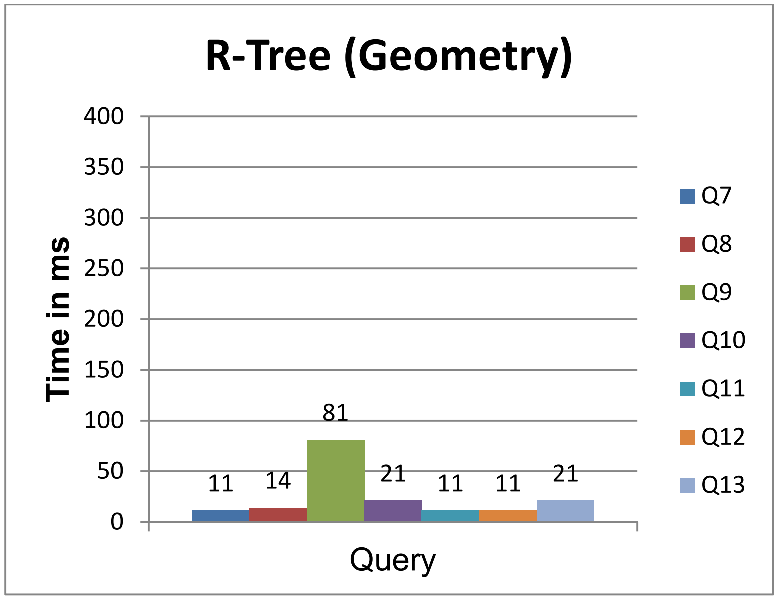


Figure 16: Time taken by R-Tree Index (Geometry)

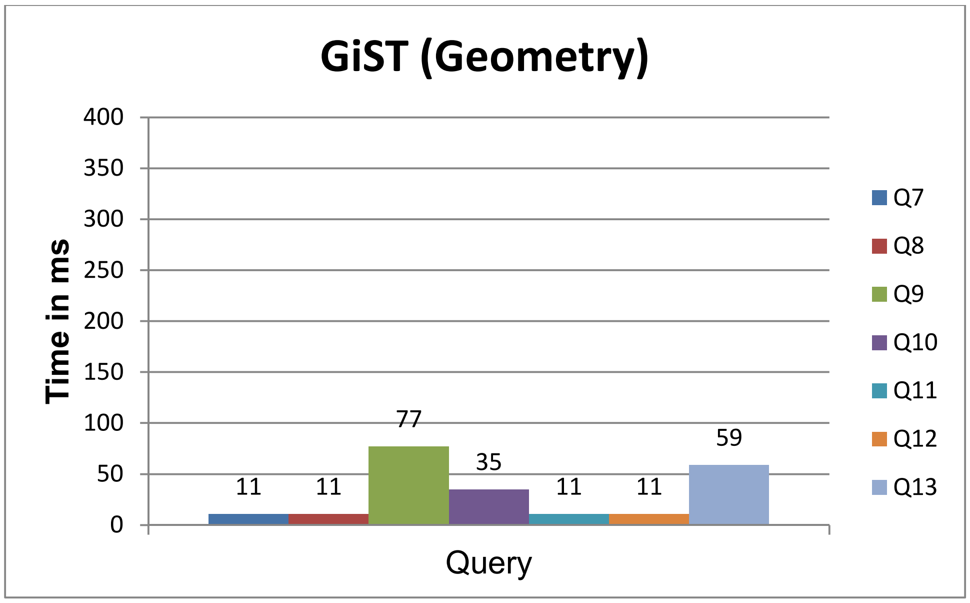


Figure 17: Time taken by GiST Index (Geometry)

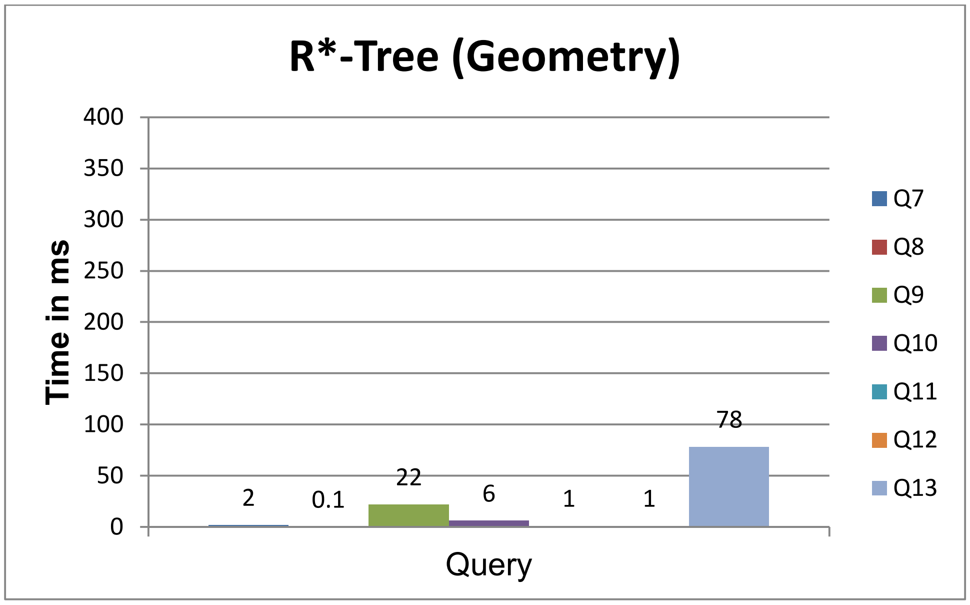


Figure 18: Time taken by R*-Tree Index (Geometry)

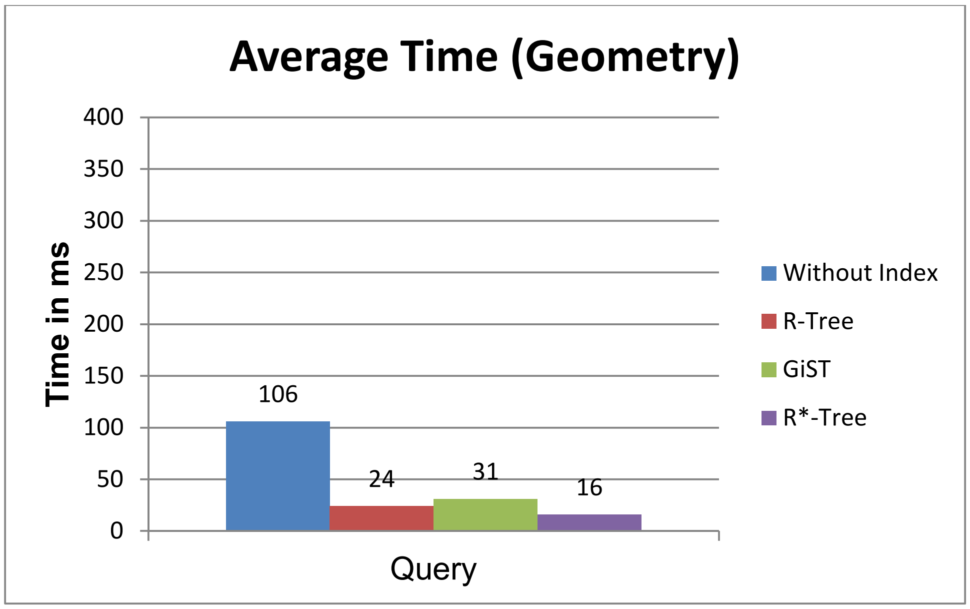


Figure 19: Average Time taken by all the indexing structures (Geometry)

According to the average time we can conclude that R*-Tree is the best indexing structure. The average time taken by R*-Tree is 16 milliseconds. If we look closely, R*-Tree performs very well in executing queries Q7, Q8, Q10, Q11 and Q12 with an average time of 2.02 milliseconds, but due to queries Q9 and Q13 which include extensive arithmetic computation the average time goes up to 50 milliseconds. Thus again it proves that R*-Tree performs bad whenever the query includes arithmetic operations.

5.3.3 Spatial Relationship

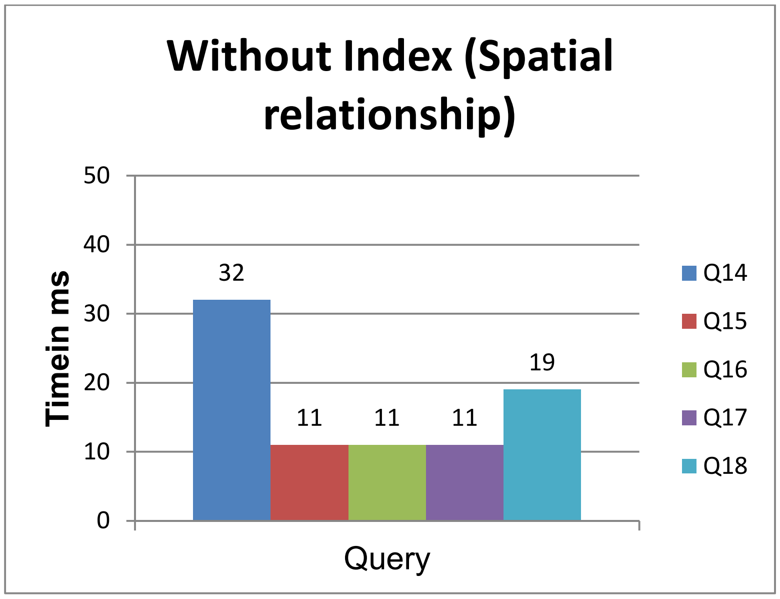


Figure 20: Time taken without Index (Spatial Relationship)

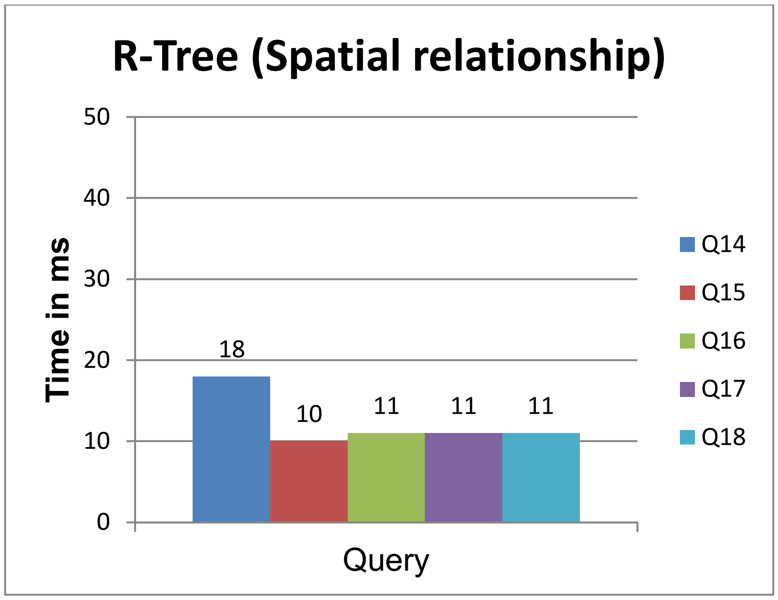


Figure 21: Time taken by R-Tree Index (Spatial Relationship)

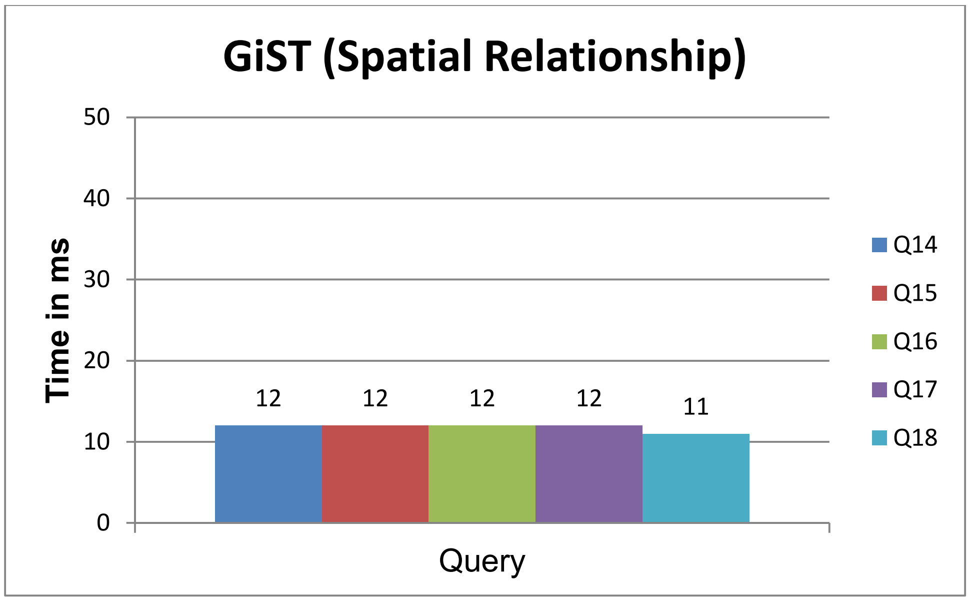


Figure 22: Time taken by GiST Index (Spatial Relationship)

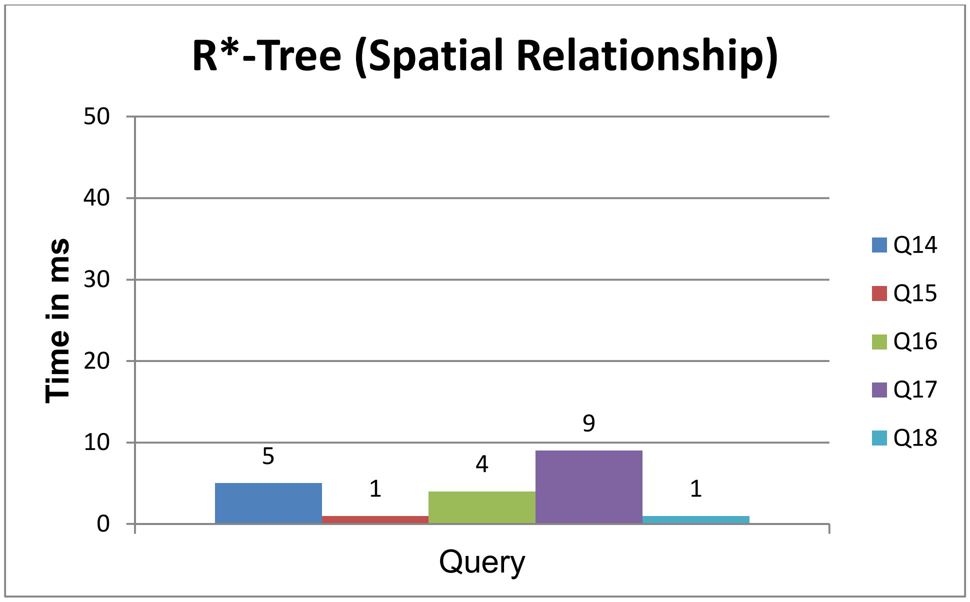


Figure 23: Time taken by R*-Tree Index (Spatial Relationship)

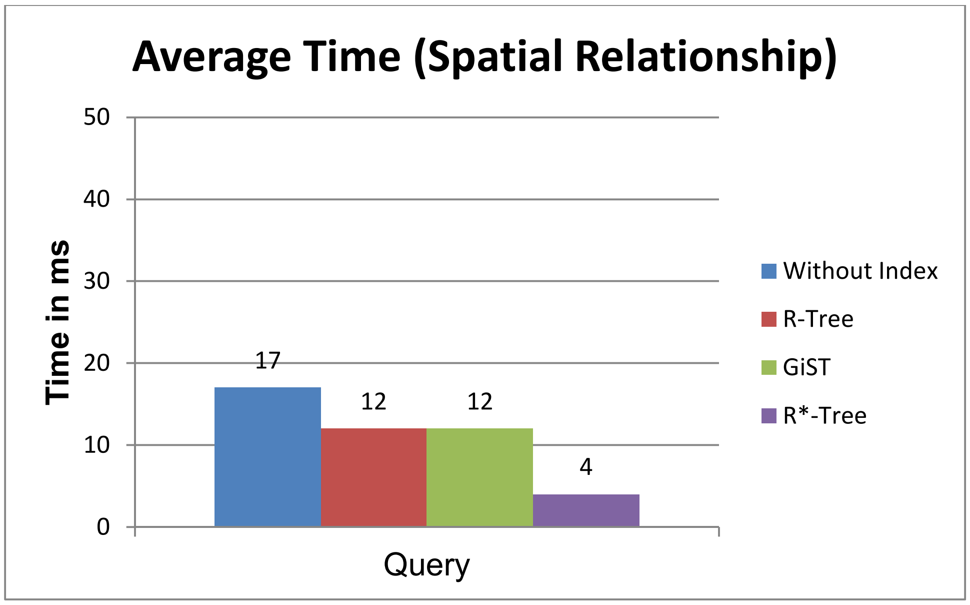


Figure 24: Average time taken by all the indexes (Spatial Relationship)

Thus again, R*-Tree wins by executing Spatial Relationship queries with an average time of 4 milliseconds.

5.3.4 Spatial Joins

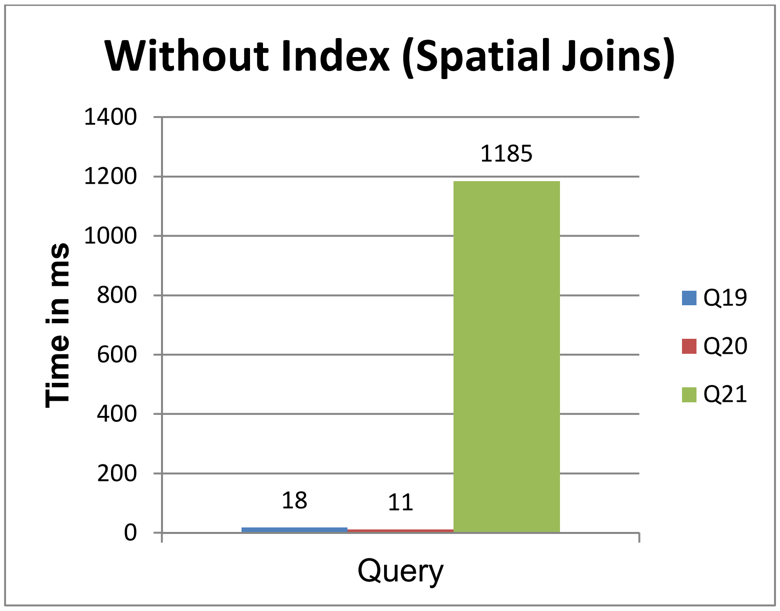


Figure 25: Time taken without Index (Spatial Joins)

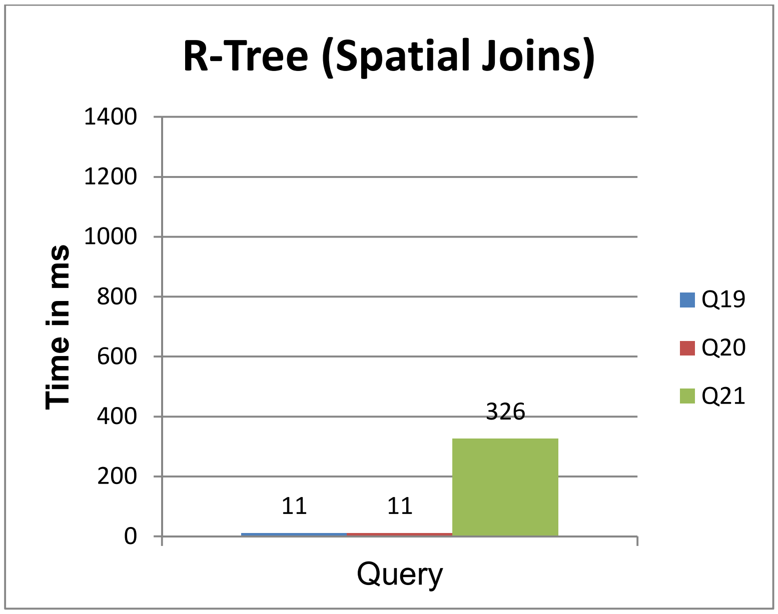


Figure 26: Time taken by R-Tree Index (Spatial Joins)

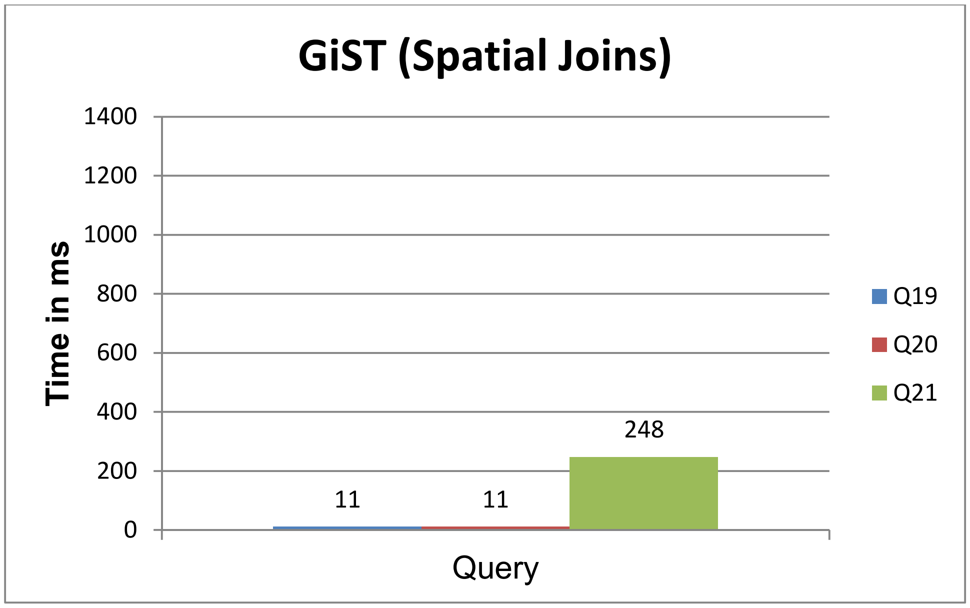


Figure 27: Time taken by GiST Index (Spatial Joins)

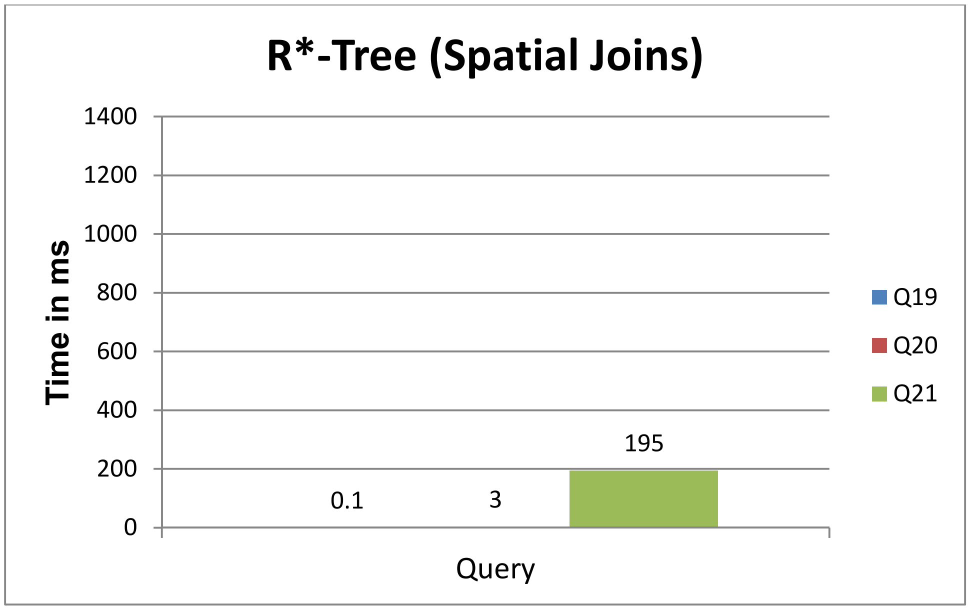


Figure 28: Time taken by R*-Tree Index (Spatial Joins)

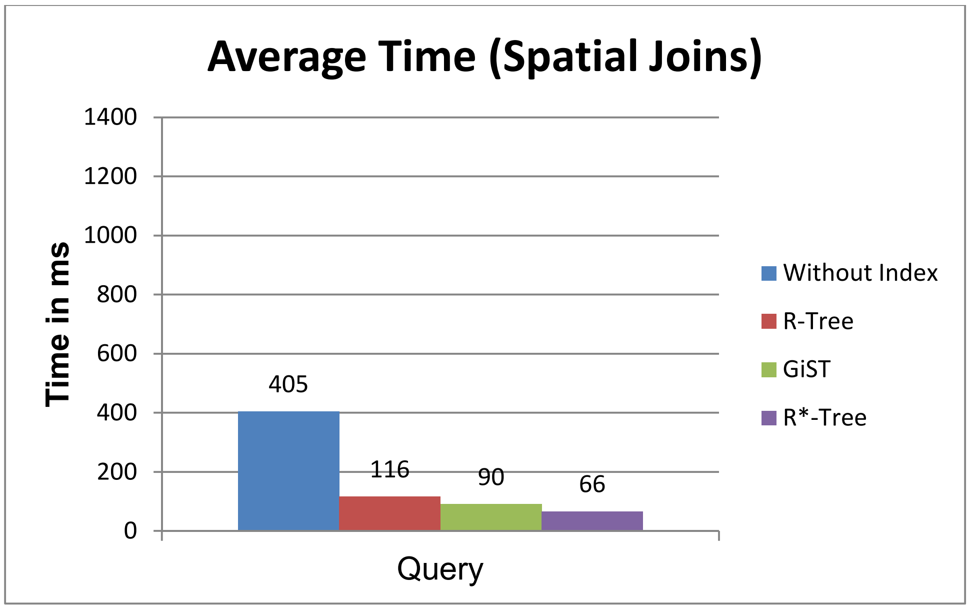


Figure 29: Average time taken by all the indexes (Spatial Joins)

In the spatial joins queries if we look closely, executing query Q21 on an average took a lot of time without or with index. Without index it took 1185 ms, with R-Tree it took 326 ms, with GiST it took 248 ms and with R*-Tree it took 195 ms. The query was about finding the total population and racial make-up of all the 28 neighborhoods of Manhattan borough so it was natural for the database to take some time to compute the query.

5.3.5 Nearest Neighbor

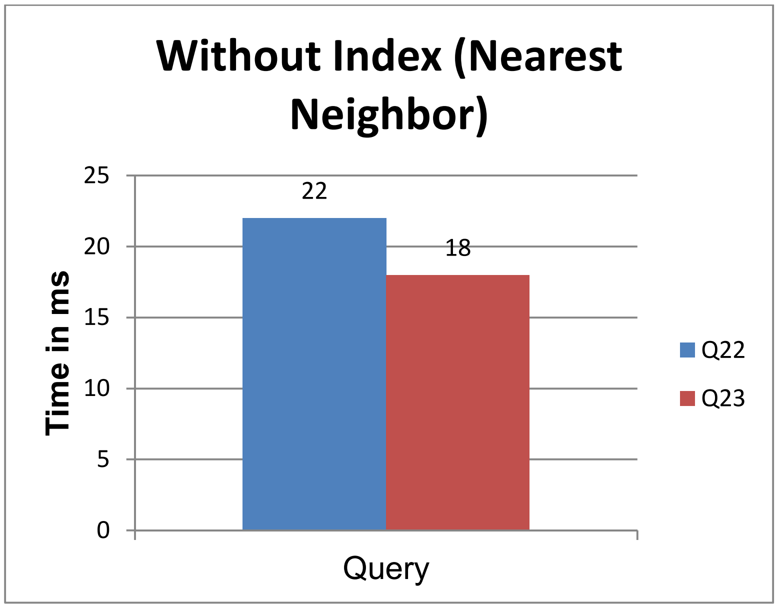


Figure 30: Time taken without Index (Nearest Neighbor)

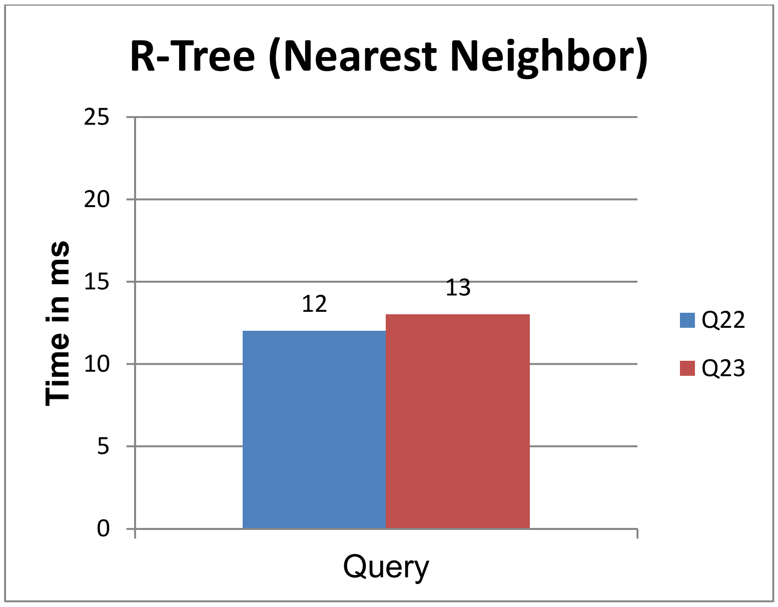


Figure 31: Time taken by R-Tree Index (Nearest Neighbor)

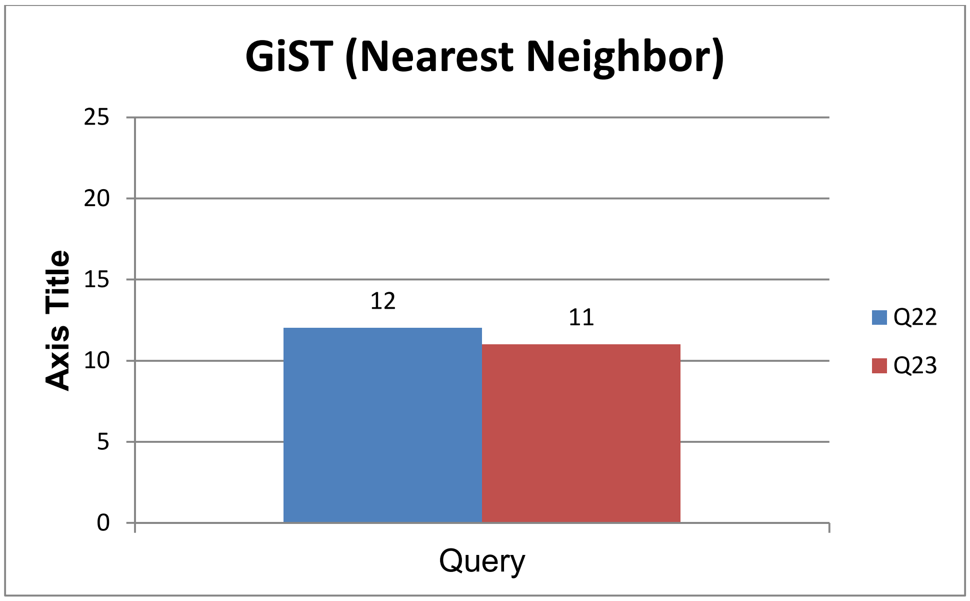


Figure 32: Time taken by GiST Index (Nearest Neighbor)

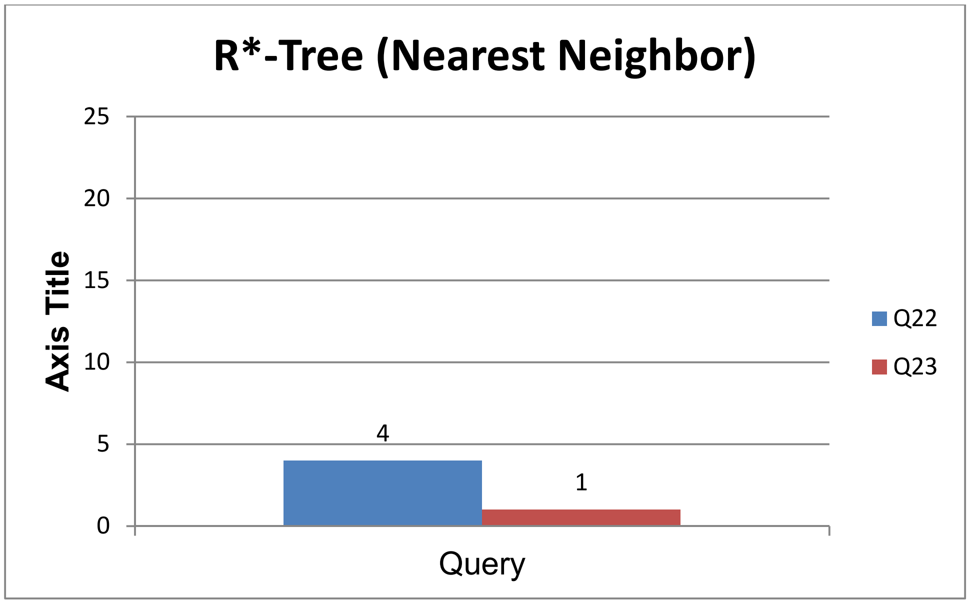


Figure 33: Time taken by R*-Tree Index (Nearest Neighbor)

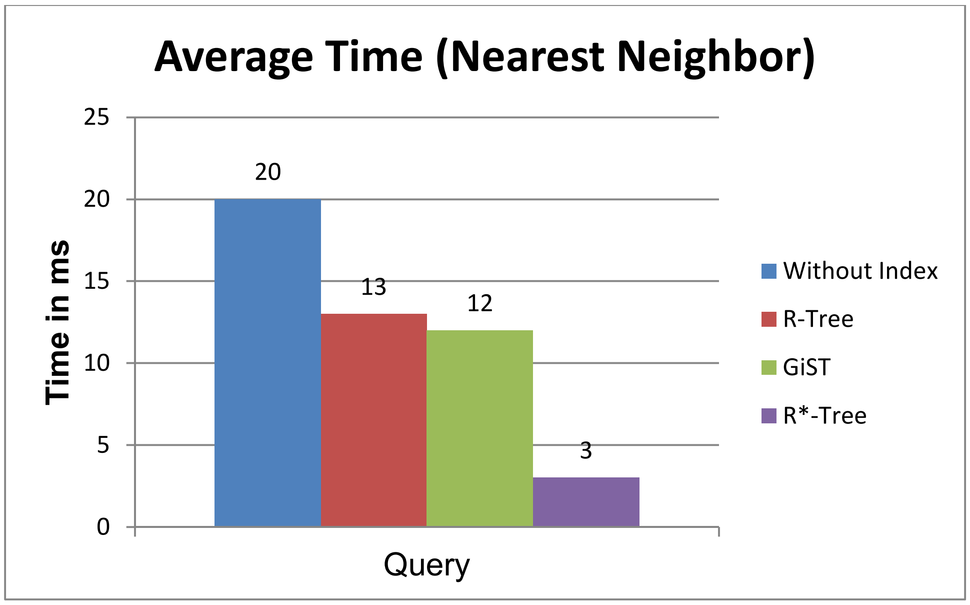


Figure 34: Average time taken by all the indexes (Nearest Neighbor)

From the above graphs it is evident that R*-Tree has shown the best performance in order to execute the nearest neighbor queries with a large difference in time.

Chapter 6
CONCLUSION AND FUTURE WORK

In this thesis, we compared the performance of three different spatial indexing structures for five different categories of queries. The spatial indexing structures we implemented were R-Trees, GiSTs and R*-Trees on two different Spatial Database Management Systems, namely PostgreSQL and SQLite with an extension of PostGIS and SpatiaLite respectively.

After executing various extensive queries, R*-Trees gave us the results in the least time for all the categories except for Simple SQL. R-Trees are the best indexing structure for executing Simple SQL queries and GiST indexing can be considered for Spatial Relationships, Spatial Joins, and Nearest Neighbor search queries after R*-Trees.

We now plan to build a Spatio-Temporal indexing structure which can efficiently index dynamic data with an additional dimension of time. Our further works will include the implementation of a new spatio-temporal indexing structure on dynamic geographical datasets. Dynamic datasets have time as another dimension which makes it more complex than the static data. Dynamic datasets consist of million users location data. Our objective is to develop and implement an indexing structure that could index the dynamic data and allow a user to retrieve the data in least possible time.

REFERENCES

[1] Antonin Guttman: R-Trees: A dynamic index structure for spatial searching, Proceeding SIGMOD '84 Proceedings of the 1984 ACM SIGMOD international conference on Management of data Pages 47 – 57

[2] Yang Gui-jun, Zhang Ji-xian: A DYNAMIC INDEX STRUCTURE FOR SPATIAL DATABASE QUERYING BASED ON R-TREES, Proceedings of International Symposium on Spatio-temporal Modeling, Spatial Reasoning, Analysis, Data Mining and Data Fusion, 27-29 Aug 2005, Beijing, China

[3] Marcel Kornacker: Access Methods for Next-Generation Database Systems, Doctoral Dissertation by Marcel Kornacker, University of California at Berkeley

[4] Joseph M. Hellerstein Jeffrey F. naughton, Avi Pfeffer: Generalized Search Trees for database Systems Proceeding Proceeding VLDB '95 Proceedings of the 21th International Conference on Very Large Data Bases

[5] Norbert Beckmann, Hans-Peter Kriegal, Ralf Schneider and Bernhard Seeger: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles, Proceeding SIGMOD '90 Proceedings of the 1990 ACM SIGMOD international conference on Management of data

[6] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos and Yannis Theodoridis: R-Trees Theory and Applications book chapter 2 Dynamic Versions of R-Trees

[7] S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, X. Liu, C. T. Lu: Spatial Database: Accomplishments and Research Needs, 2002

[8] Ralf Hartmut Guting: An Introduction to Spatial Database Systems, VLDB Journal, 3, 357-399, 1994

[9] Tutorials on PostGIS by BostonGIS: Boston Geographic Information Systems, www.bostongis.com/?content_name=postgis_tut01#304

[10] workshops.boundlessgeo.com: PostgreSQL workshops and the dataset source

[11] Shashi Shekhar, Sanjay Chawla: Spatial Databases- A Tour, Book, ISBN 0-13-017480-7, 2003

[12] Max J. Egenhofer: Spatial SQL: A Query and Presentation Language, IEEE Transactions on Knowledge and Data Engineering 6 (1): 86-95, 1994

[13] A. Borrman: from GIS to BIM and back again – a spatial query language for 3d building models and 3d city models, 2005

[14] V. Gandhi, J. M. Kang, S. Shekhar: Encyclopedia of Computer Science and Engineering, Wiley, Cassie Craig (Eds.), 2009

[15] V. Gandhi, J. M. Kang, S. Shekhar: Technical Report TR07-020, Dept. of Compter Sci., U. of Minnesota, 2007

BIOGRAPHICAL INFORMATION

Neelabh Pant was born in Nainital, India, in 1993. He earned his B. Tech degree from Birla Institute of Applied Sciences, Bhimtal, India and M.S. degree from The University of Texas at Arlington in 2015 all in Computer Science. His current research interest is Spatial Indexing for the large geographical dataset. He will start his PhD in the fall of 2015 at the University of Texas at Arlington, where his primary focus will be on temporal aspect in spatial indexing.