PERFORMANCE COMPARISON OF SPATIAL INDEXING STRUCTURES FOR DIFFERENT QUERY TYPES

Conference Paper · June 2016

CITATION
1

READS
153

4 authors:

Neelabh Pant
University of Texas at Arlington
3 PUBLICATIONS **2** CITATIONS
SEE PROFILE

Mohammadhani Fouladgar
University of Texas at Arlington
7 PUBLICATIONS **15** CITATIONS
SEE PROFILE

Ramez Elmasri
University of Texas at Arlington
211 PUBLICATIONS **6,279** CITATIONS
SEE PROFILE

Kulsawasd Jitkajornwanich
King Mongkut's Institute of Technology Ladkrabang
18 PUBLICATIONS **29** CITATIONS
SEE PROFILE

PERFORMANCE COMPARISON OF SPATIAL INDEXING STRUCTURES FOR DIFFERENT QUERY TYPES

[1]NEELABH PANT, [2]MOHAMMADHANI FOULADGAR, [3]RAMEZ ELMASRI, [4]KULSAWASD JITKAJORNWANICH

[1,2,3]Computer Science Department, University of Texas at Arlington, Arlington, Texas, USA
[4]Geo-Informatics and Space Technology Development Agency,
Ministry of Science and Technology of Thailand, Bangkok, Thailand 10210
E-mail: [1]neelabh.pant@mavs.uta.edu, [2]mohammadhani.fouladgar@mavs.uta.edu, [3]elmasri@cse.uta.edu,
[4]kulsawasdj@gistda.or.th

Abstract— Several techniques have been proposed to improve the performance of spatial indexes [1, 8, 9], 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), R-Tree implementation using GiST (Generalized Search Tree) [http//www.postgresql.org/docs/9.3/static/gist-intro.html] 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. We perform extensive experiments in all these five categories and compare the performance for each category on the index structures.
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.

Keywords— Spatial Indexing, Spatial Queries, Performance Comparison.

I. INTRODUCTION

Information about many real world data objects are represented spatially in the form of polygons, lines and points, which are stored and retrieved from a database. Such real world objects have spatial attributes, which are difficult to store in a traditional database (RDBMS) because these objects are complex and multi-dimensional in nature. Spatial databases are used to store spatial objects, both fixed location objects such a roads and buildings, and moving objects such as vehicles and cell phones for location based services. The latter is known as moving objects databases, because their spatial location changes over time. Other applications that utilize spatial databases include Geographic Information Systems (GIS), environmental modeling and impact assessment, resource management, among many other applications.
All this information is accessed from the Spatial Database Management System (SDBMS). We need an index to store database objects for efficient retrieval. Spatial database management systems make use of spatial indexing structures for fast and efficient access to the spatial data. There are several spatial indexing structures proposed and each of them are good for certain purposes. In this paper three spatial index structures are compared based upon their performance in different conditions. Our comparison focusses on spatial data, not moving objects data.
We compare R-tree (Rectangle Tree), GiST (Generalized Search Tree) and R*-tree (variant of R-tree) on two different spatial database management systems, PostgreSQL and SQLite. We make use of geographic data in the form of shapefiles. The shapefiles are the benchmark data of New York city, of 58,505 records that consists of points, lines, polygons and multi-polygons represented by x-y geometry.
The major objective of this paper is to compare the index performance based upon different *categories of queries* such as, simple SQL, spatial relationship, nearest neighborhood, spatial joins and geometry. The organization of this paper is as follows. We give a brief background on indexing structures, spatial databases, dataset used for experiments and the amount of time consumed in the construction of indexes in II. We perform experiments on each type of index for different categories of query in III. The experimental results are discussed in IV. Finally, we conclude and discuss future work in V.

II. BACKGROUND

A. Indexing Structures

1) R-trees: R-trees are a spatial extension of B-trees. They are height balanced and store the object information so that spatial queries, such as range queries, point queries, nearest neighbor searches, etc. can be executed efficiently. Spatial objects are stored in such a way that queries such as e.g. *Find all the restaurants within 20 miles from the current position* are executed within a fraction of a second.

2) GiST: GiST is a customizable balanced search tree which encapsulates basic access methods and supports wide range of data-types. GiST can emulate a variety of the tree-structure access methods through a template algorithm, which requires only a small set of functions to be configured [9]. Some examples of the implemented operations include delete and node split operations that modify the structure of the tree

[10]. The GiST provides the luxuries of basic search tree logic for a database system. GiST code can be adapted so that different tree structures (e.g. B+-tree and R-tree) can easily be implemented.

3) R-Tree:* R*-tree was proposed by Beckmann et al. in 1990 [1]. It is a variant of R-tree which is also used to index spatial data objects such as points, lines and polygons. It has a slightly higher cost of materialization than the standard R-tree, some data in R*-tree may need to be reinserted. However R*-tree gives a better overall search performance compared to the original R-tree.

B. Spatial Database

1) PostgreSQL: PostgreSQL is an open source Object-Relational Database Management System (ORDBMS). PostgreSQL supports the SQL standards as well as other key DBMS features including complex queries, triggers, updatable views, transactional integrity, and multi-version concurrency control [15]

2) PostGIS: PostGIS is an extension of the PostgreSQL. PostGIS enhances the database capabilities by allowing spatial objects to be represented, stored, indexed and queried in the database. Additional supported data types include geometry, geography, raster, and other spatial data objects. These spatial data-types are added along with their functions, operators and indexing structures, to optimize the performance of the database. We index our data using R-trees and GiST on a PostGIS/PostgreSQL database

3) SQLite: SQLite is an all-in-one RDBMS embedding all necessary components including database engine to a single disk file. The single disk file contains an entire database and communicates directly with a program whenever a read or write is performed. Other SQLite key features include cross-platform data file, compact, manifest typing, variable-length records and is an open source system. [https://www.sqlite.org/different.html]

4) SpatiaLite: One can think of SpatiaLite as an added spatial technology for SQLite similar to what PostGIS does for the PostgreSQL. SpatiaLite provides vector-geodatabase functionalities which are equivalent to PostGIS, Oracle Spatial and Microsoft SQL-Server with spatial extensions. Instead of implementing a client-server architecture, SpatiaLite adopts a simpler personal architecture in which both the application and the database are located together. As mentioned earlier, the complete database is an ordinary file, which can be transferred and moved between different computers and operating systems. We use R*-tree to index our data in a SpatiaLite database.

C. Dataset

The spatial data that are used in our experiments is the set of a benchmark data of New York City that includes point data: subway stations, line data: streets and subway lines, polygon data: boroughs and neighborhoods plus non-spatial data such as population data.

There are four tables: *nyc_census_blocks, nyc_neighborhoods, nyc_streets and nyc_subway_stations*. *nyc_census_blocks* table contains population information (denoted by *popn* in the attribute names) for each census block (denoted by *blk*) borough (*boro*) categorized by race. *nyc_neighborhoods* contains neighbourhood information for all boroughs. Street names, types, and other properties (e.g., whether the given street is *one-way*) are stored in *nyc_streets* table. The last table, *nyc_subway_stations*, provides station information as to which subway line run through (*routes*) and/or can *transfer* to via this station. The *nyc_subway_stations* table also tells whether the station is an *express* train stop. All four tables have a spatial column storing a geometry object (*geom*). The numbers of records for each table are 38,794; 129; 19,091; 491 respectively.

D. Index Construction Time

Creating index on the geometry columns takes time depending on the type of index. Some indexes take a short time, whereas, others can take more time. We record the time taken (vertical axis, in Millisecond, ms.) for constructing the indexing trees, namely, R-tree, GiST and R*-tree on our benchmark data set.

All of our four tables are indexed on the geometry column. Figure 1 shows the average execution time for creating each index on each of the four tables:

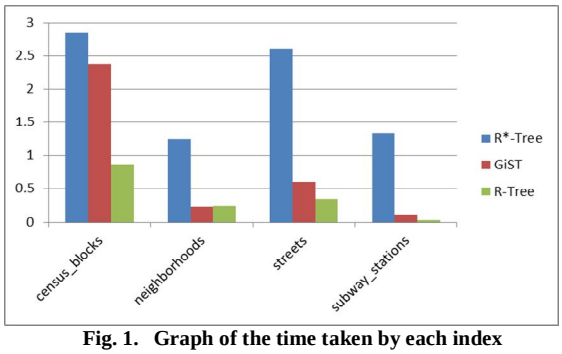

Fig. 1. Graph of the time taken by each index

As seen in Figure 1, R*-tree takes the most time to create and generally, more than twice the time, compared to the other indexing techniques.

III. SPATIAL QUERIES

The spatial queries we perform experiments with can be divided into five categories according to [12]. We denote each query by a certain variable (Q1, Q2,…, Qn) so that later in this paper, we can refer to each query just by its variable.

A. Simple SQL (Non-Spatial):
SimpleSQL are the queries that do not have spatial conditions. We use these to compare whether spatial indexes create overhead for processing non-spatial queries. Queries Q1 through Q6 in Appendix A are the non-spatial queries we used.

B. Geometry (Metric Queries):
Queries Q7 through Q13 include spatial measurements, such as length and area, and are listed in Appendix B.

C. Spatial Relationship Topological):
Queries Q14 through Q17 involve spatial conditions based on topological spatial relationships, and are listed in Appendix C.

D. Spatial Join:
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 locations. Queries Q18 through Q20 include Spatial joins, and are given in Appendix D.

E. Nearest neighborhoods:
Nearest neighbor queries find object(s) that are closest to a given object. We tested two queries in this category: Q21 and Q22, which are listed in Appendix E.

IV. EXPERIMENTAL RESULTS

In the this section we will see the execution time in milliseconds for each category of query, with/without an index of the three indexing structures.

A. Simple SQL:
The performance results for the queries Q1-Q6 are displayed in Figure 2 through Figure 5. Figure 2 shows the results for no index, and Figure 3 to 5 show the results for R-tree, GiST, and R*-tree, respectively.

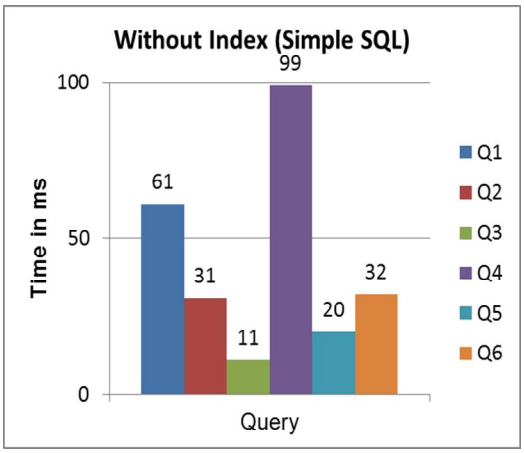

Fig. 2. Time taken without index (Simple SQL)

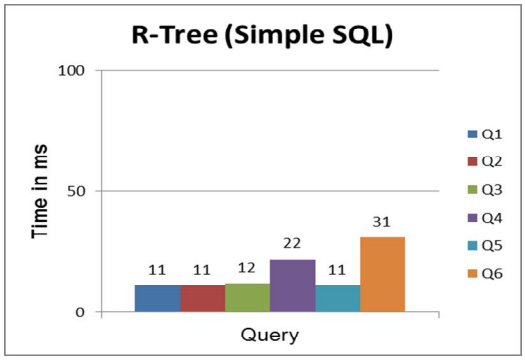

Fig. 3. Time taken by R-tree Index (Simple SQL)

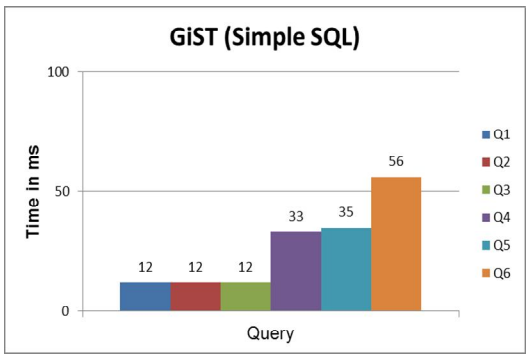

Fig. 4. Time taken by GiST index (Simple SQL)

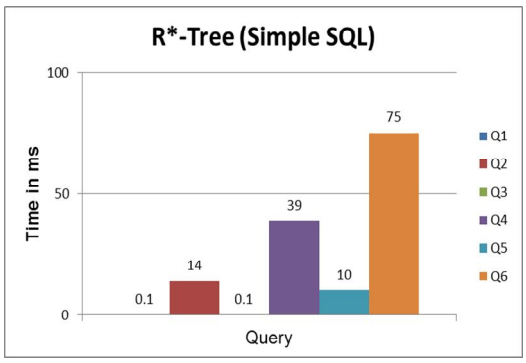

Fig. 5. Time taken by R*-tree index (Simple SQL)

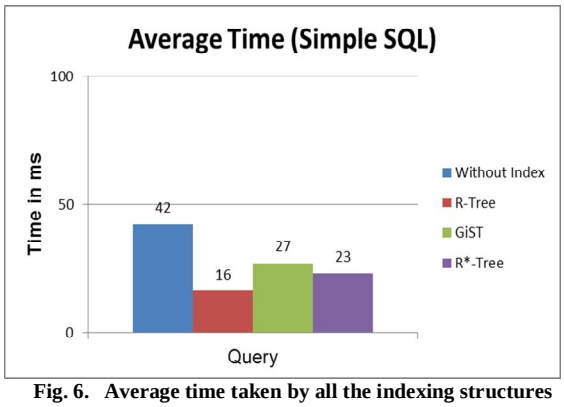

Fig. 6. Average time taken by all the indexing structures (Simple SQl)

Referring to all the histograms presented above, there is no single best method to follow when executing the Simple SQL queries. By calculating the average time of all queries Q1 - Q6, it is easier to determine the most efficient indexing structure for Simple SQL. This is shown in Figure 6.

According to the histograms, R-tree is the least time consuming indexing structure. The operations in Simple SQL were more about arithmetic operations like sum and division. Since, R*-tree was implemented on SpatiaLite and based on our additional experiments, SpatiaLite showed poor performance in computing arithmetic operations when compared with PostGIS.

B. Geometry (Meteric Queries):

Figure 7 displays the time taken by the metric queries (Q7-Q13) without index. Figure 8 through Figure 10 shows the time taken when indexed using R-tree, GiST and R*-tree respectively.

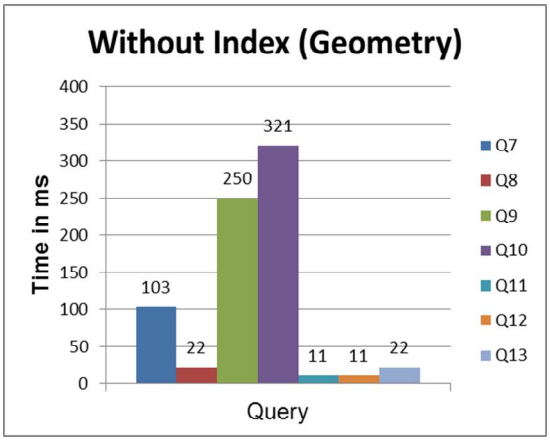

Fig. 7. Time taken without index (Geometry)

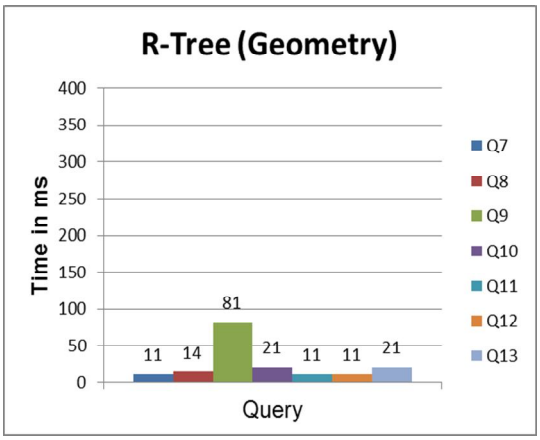

Fig. 8. Time taken by R-tree index (Geometry)

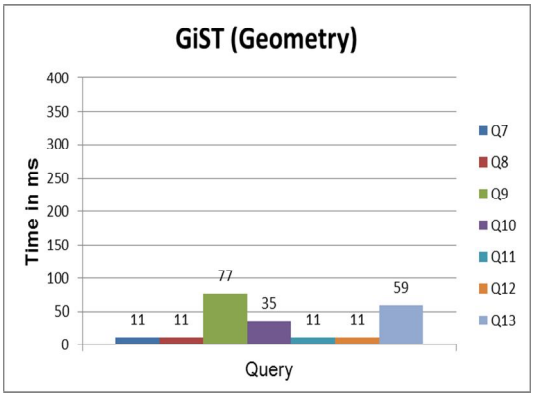

Fig. 9. Time taken by GiST index (Geometry)

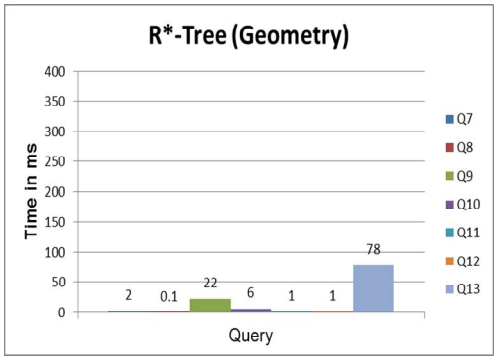

Fig. 10. Time taken by R*-tree index (Geometry)

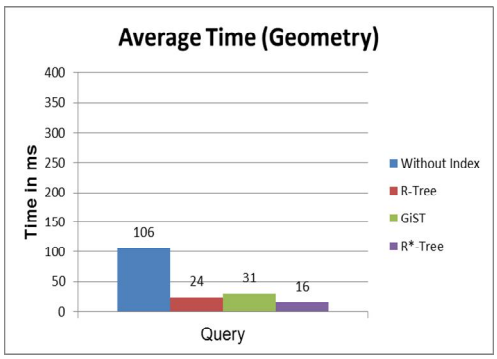

Fig. 11. Average time taken by all the indexes (Geometry)

According to the average time we can conclude that R*-tree is the best indexing structure for spatial measurement queries. 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 for queries Q9 and Q13 which include extensive arithmetic computation the average time goes up to 50 milliseconds. This is because since R*-tree is implemented on SpatiaLite and based on our additional experiments, computing arithmetic operations in SpatiaLite takes comparatively more time than PostGIS.

C. Spatial Relationship

The time taken without index is shown in Figure 12 for queries Q14-Q16. Figure 13-15 show the time executed by different queries (Q14-Q16) when indexed using R-tree, GiST and R*-tree respectively.

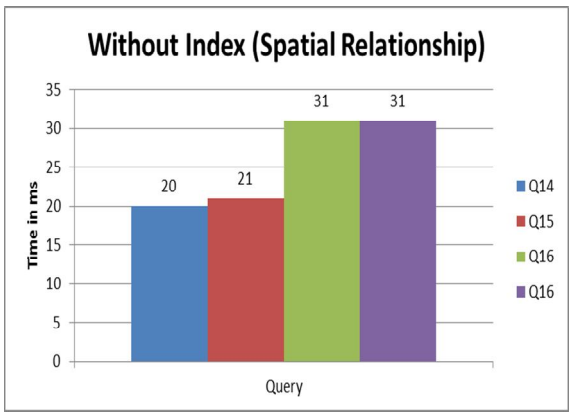

Fig. 12. Time taken without index (Spatial Relationship)

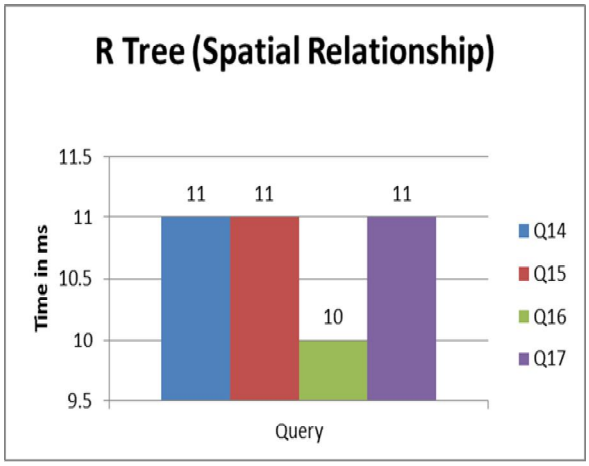

Fig. 13. Time taken by R-tree index (Spatial Relationship)

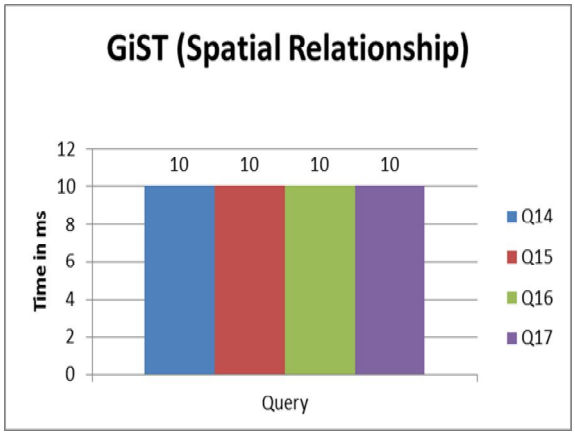

Fig. 14. Time taken by GiST index (Spatial Relationship)

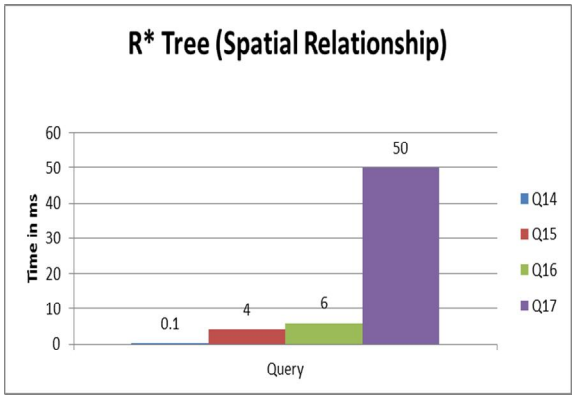

Fig. 15. Time taken by R*-tree (Spatial Relationship)

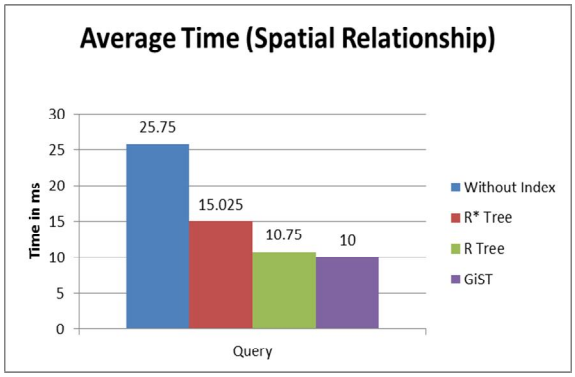

Fig. 16. Average time taken by all the indexes (Spatial Relationship)

Figure 16 also shows that, the R*-tree performs better than the other methods when executing Spatial Relationship queries, with an average time of 4 milliseconds.

D. Spatial Joins:

Total execution time without an index is shown in Figure 17 and Figures 18-20 demonstrate the total time taken to execute the queries (Q18-Q20) by the three different indexing structures.

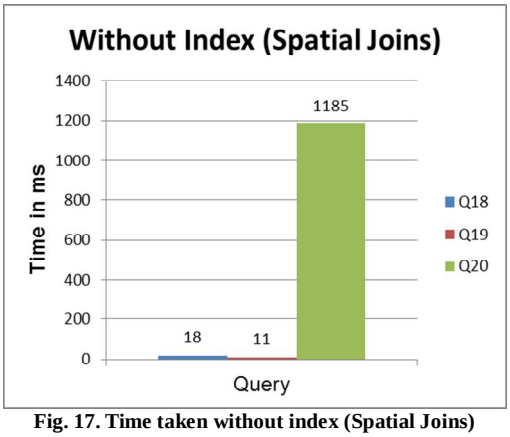

Fig. 17. Time taken without index (Spatial Joins)

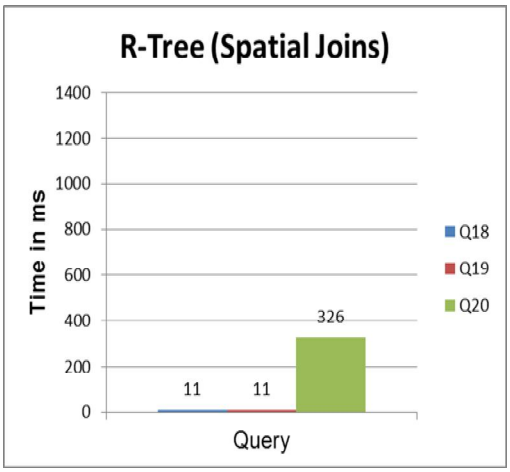

Fig. 18. Time taken by R-tree index (Spatial Joins)

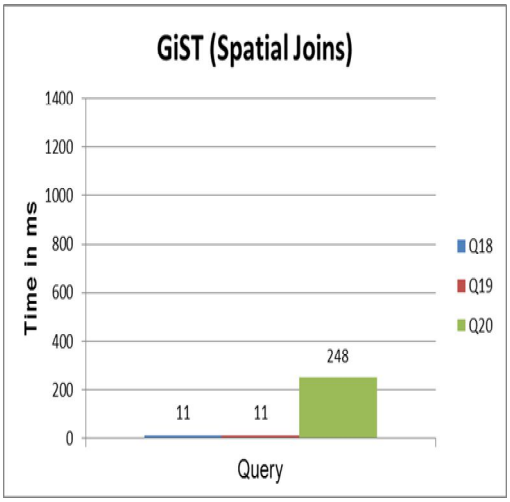

Fig. 19. Time taken by GiST index (Spatial Joins)

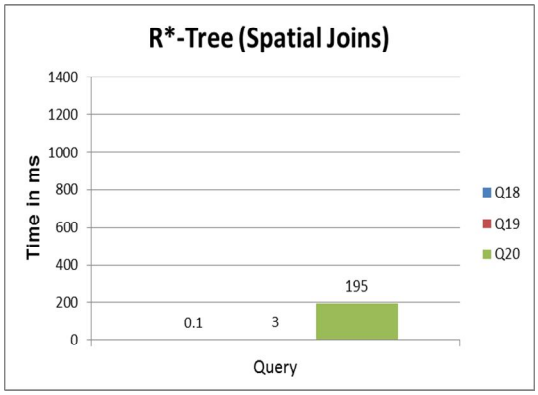

Fig. 20. Time taken by R*-tree index (Spatial Joins)

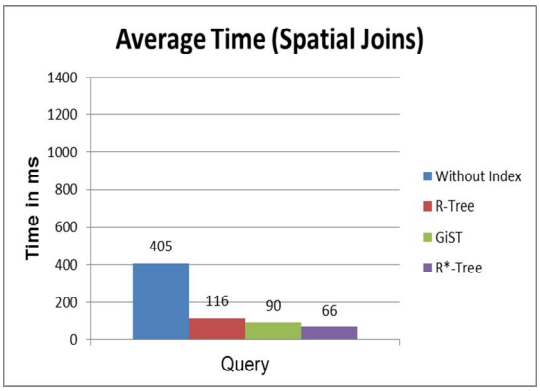

Fig. 21. Average time taken by all the indexes (Spatial Joins)

In the spatial joins queries we can notice that, executing query Q20 on an average took a lot of time without or with index. Without index it took 405 ms, with R-tree it took 116 ms, with GiST it took 90 ms and with R*-tree it took 66 ms The query was about finding the total population and racial makeup of all the 28 neighborhoods of Manhattan borough so it was natural for the database to take some time to compute the query. In this query, R*-tree still gave the best performance.

E. Nearest Neighbor

The time taken by all the queries (Q21-Q22) to execute without any index is displayed in Figure 22 and the Figures 23-25 show the execution time when indexed using R-Tree, GiST and R*-Tree respectively.

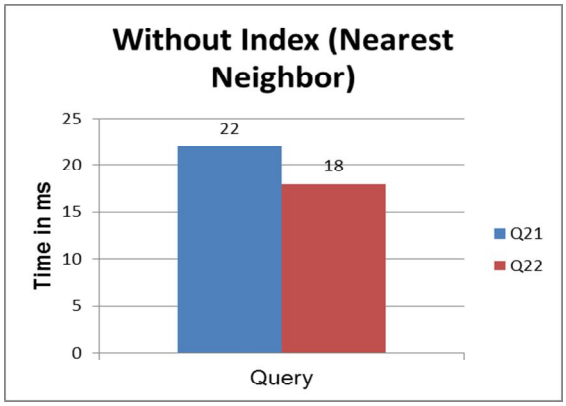

Fig. 22. Time taken without index (Nearest Neighbor)

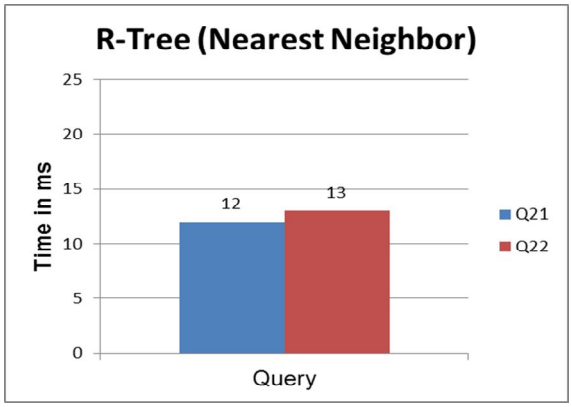

Fig. 23. Time taken by R-tree index (Nearest Neighbor)

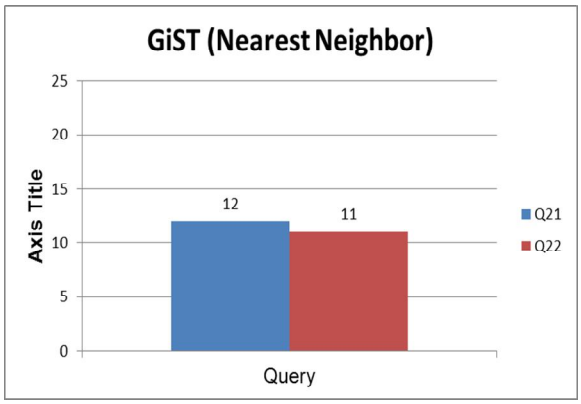

Fig. 24. Time taken by GiST index (Nearest Neighbor)

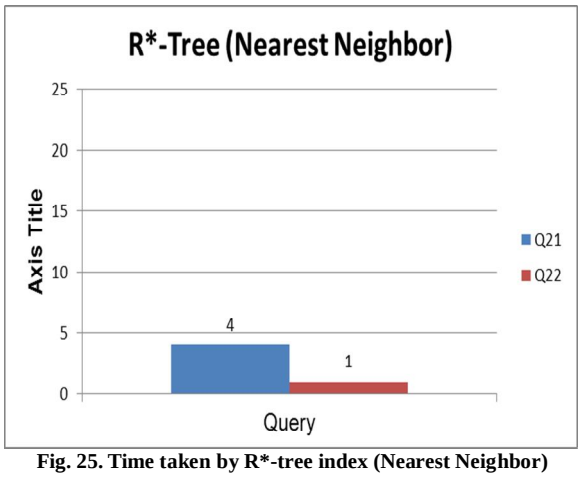

Fig. 25. Time taken by R*-tree index (Nearest Neighbor)

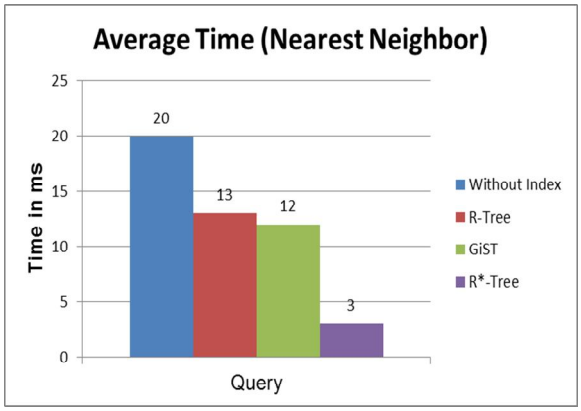

Fig. 26. Average time taken by all the indexes (Nearest neighbor)

As seen in Figure 26, it is evident that R*-tree has shown the best performance in order to execute the nearest neighbor queries with a large improvement in time from the other cases.

CONCLUSION

In this paper, we compared the performance of three different spatial indexing structures for five different categories of queries. The spatial indexing structures we implemented were R-tree implemented using GiSTs and R*-tree on two different SDBMS, PostgreSQL and SQLite using their spatial extensions, 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 many user 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 query results in least possible time.

REFERENCES

[1] Beckmann, N., Kriegal, H., Schneider R., and Seeger B. 1990. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data.*

[2] Borrman A. 2005. From GIS to BIM and back again – a spatial query language for 3d building models and 3d city models.

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

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

[5] Gandhi, V., Kang, J.M., and Shekhar, S. 2007. Technical Report TR07-020, Dept. of Computer Sci., U. of Minnesota.

[6] Gui-jun, Y. and Ji-xian, Z. 2005. A DYNAMIC INDEX STRUCTURE FOR SPATIAL DATABASE QUERYING BASED ON R-TREES. In *Proceedings of International Symposium on Spatio-temporal Modeling, Spatial Reasoning, Analysis, Data Mining and Data Fusion*, (Aug 2005, Beijing, China), 27-29.

[7] Guting R.H. 1994. An Introduction to Spatial Database Systems. *VLDB Journal*, 357-399.

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

[9] Hellerstein, J.M., Naughton, J.F., and Pfeffer, A. 1995. Generalized Search Trees for database Systems. *In Proceeding Proceeding VLDB '95 Proceedings of the 21th International Conference on Very Large Data Bases*.

[10] Kornacker, M. 2000. *Access Methods for Next-Generation Database Systems.* Doctoral Dissertation. University of California at Berkeley.

[11] Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., and Theodoridis, Y. 2006. *R-Trees Theory and Applications*. Springer Science & Business Media.

[12] Shekhar S. and Chawla S. 2003. *Spatial Databases- A Tour, Book*. ISBN 0-13-017480-7.

[13] Shekhar, S. , Chawla, S., and Ravada S., Fetterer, A., Liu, X., Lu, C.T. 2002. *Spatial Database: Accomplishments and Research Needs.* Knowledge and Data Engineering, IEEE Transactions on, 11.1, 45-55.

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

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

APPENDIX A

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;
```

APPENDIX B

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;
```

APPENDIX C

Q14: For the street named ‘W Lake Dr find the geometry value.

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

Q15: Find the neighborhood and borough that has the biggest section of ‘W Lake Dr’.

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

Q16: Find the street closest to ‘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.)

Q17: 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
);
```

APPENDIX D

Q18: 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.

Q19: 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';
```

Q20: 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;
```

APPENDIX E

Q21: 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';
```

Q22: 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;
```