Indexing PostGIS Databases and Spatial Query Performance Evaluations

Nguyen, T. T.
Department of Geography Education, College of Education, Kangwon National University
Chuncheon City, 200-701, Republic of Korea, E-mail: thi.nguyenthanh@gmail.com

Abstract
Geodatabases (also commonly known as geospatial databases) are central elements in spatial data infrastructures. The primary advantage over file-based data storage of spatial databases (access via GIS), is that they are structured to encompass existing capabilities of relational database management systems, including support for SQL (Structured Query Language) and the ability to generate complex geospatial queries. PostGIS is an extension to the PostgreSQL object-relational database system which allows GIS objects to be stored in the database. PostGIS comprises functions for basic analysis of GIS objects and more importantly, it also supports the spatial indexing schemes. Indexes are extremely important for large spatial tables, because they allow for quick retrieval of records during query. PostGIS is frequently used during analysis of large data sets if examination of spatial indexes is a particularly essential task. Reported here are results of indexing the PostGIS databases by adopting an R-Tree-over-GiST (Generalized Search Tree) scheme and evaluation of the performance of indexed and un-indexed spatial queries with respect to database size. Experiments were carried out with a huge amount of spatial data obtained from the ESRI website, and using the pgAdmin III tool, which is a comprehensive PostgreSQL database design and management system. Experimental results demonstrate approximately linear increase in spatial index building time as the size of tables increases, but, as the database size increases, processing time is very much greater if the spatial queries are not indexed. However, regardless of the size of the geodatabase, performance of spatial queries is highly sensitive to choice of geometric parameters that the queries refer to.

1. Introduction

A spatial query, a special type of database query, is supported by geodatabase structures. The queries differ from SQL queries in several important ways. Two of the most important are that they allow for the use of geometry data types such as points, lines and polygons and that these queries consider the spatial relationship between these geometries. Indexes of typical database systems supporting SQL support rapid data retrieval but this approach is not optimal for spatial queries. Instead, spatial databases use a spatial index to speed up database operations.

Spatial indexes are what make feasible the query of large data sets in spatial databases. Without indexing, any search for a feature would require a "sequential scan" of every record in the database. Indexing speeds up searching by organizing the data into a search tree which can be quickly traversed to find a particular record.

By default, PostgreSQL DBMS (Database Management System) supports three kinds of indexes: B-Tree indexes, R-Tree indexes, and GiST indexes. B-Trees are used for data which can be sorted along one axis; for example, numbers, letters, dates. GIS data cannot be rationally sorted along one axis so that B-Tree indexing is not recommended for geodatabases. R-Trees (Guttman, 1984 and Gavrila, 1994) break up data into rectangles, and sub-rectangles, and sub-sub rectangles, and so on. R-Trees are used by some spatial databases to index GIS data, but the PostgreSQL R-Tree implementation is not as robust as the GiST implementation (Ramsey, 2008). GiST stands for "Generalized Search Tree" and is a generic form of indexing. In addition to GIS indexing, GiST is used to speed up searches on all kinds of irregular data structures (integer arrays, spectral data, and so on) which are not amenable to normal B-Tree indexing. PostGIS (Ramsey, 2004), which is a spatial database add-on for the PostgreSQL relational database server, uses an R-Tree index implemented on top of GiST to index GIS data. R-Trees are able to handle well compression because spatial data is organized into nesting rectangles in order to facilitate fast searching. Early versions of PostGIS used the PostgreSQL R-Tree indexes. However, PostgreSQL R-Trees have been completely discarded since version 0.6, and spatial indexing is provided with an R-Tree-over-GiST scheme. The nature of this type

of indexing for PostGIS databases, not yet fully explored, is reported in the following sections.

2. Indexing PostGIS Databases

2.1. PostGIS R-Tree-over-GiST Scheme

The Generalized Search Tree was originally described by Hellerstein, *et al.* (1995). It is an index structure offering a balanced, tree-structured access method. It is easily extensible, both in the data types it can index, and in the queries it can support. Extensibility of queries is particularly important, because it allows new data types to be indexed in a manner that supports the queries natural to the types. In addition to providing extensibility for new data types, the GiST unifies previously disparate structures used for currently common data types. GiST acts as a template for implementing other indexing mechanisms such as B+-trees and R-trees, resulting in a single code base for indexing multiple dissimilar applications.

GiST indexes have two advantages over R-Tree indexes in PostgreSQL. First, GiST indexes are "null safe", meaning they can index columns which include null values. The original R-Tree indexes in PostgreSQL do not support this, so building an index on a geometry column which contains null geometries will fail. Secondly, GiST indexes support the concept of "lossiness" which is important when dealing with GIS objects larger than the PostgreSQL 8Kb page size. Lossiness allows PostgreSQL to store only the "important" part of an object in an index, e.g. in the case of GIS objects, just the bounding box. GIS objects larger than 8Kb will cause R-Tree indexes to fail in the process of being built.

Taking full advantage of GiST and R-Tree, PostGIS has deployed a GiST-related index that is more accurately referred to "R-Tree over GiST". With the data partitioning scheme reliant on an R-Tree, the GiST is utilized as the framework on which the index is implemented. Moreover, the GiST-based R-Tree has recently been applied the new linear node splitting algorithm proposed by Ang and Tan (1997). This algorithm has shown in many aspects to be more efficient than the previously existed algorithms, in terms of performance.

2.2. Timing GiST Index Building

To create a spatial index (GiST) in PostGIS, the following standard SQL syntax is conducted:

The larger the spatial table, the more efficiency the spatial indexes can bring utility to users. However, building a spatial index in the large table format is a computationally intensive procedure (Ramsey, 2008). A series of experiments was carried out to measure the index building time using a spatial dataset downloaded from the ESRI website. The so-collected spatial data is comprised of line features: roads of the thirteen southern states of the USA (Figure 1). The computer configuration used for the experiments reported here is: Intel Pentium 4, CPU 2.93GHz, and 1GB of RAM.

From the tabulation (Table 1) and plotting (Figure 2) of index building time versus the size of spatial tables is tabulated it is seen that the time of index building is virtually in direct proportion to the number of table rows. Notable, is the considerable time needed in index building of the largest table: as a 'rule of thumb', 20 minutes for a table of 10 million rows.

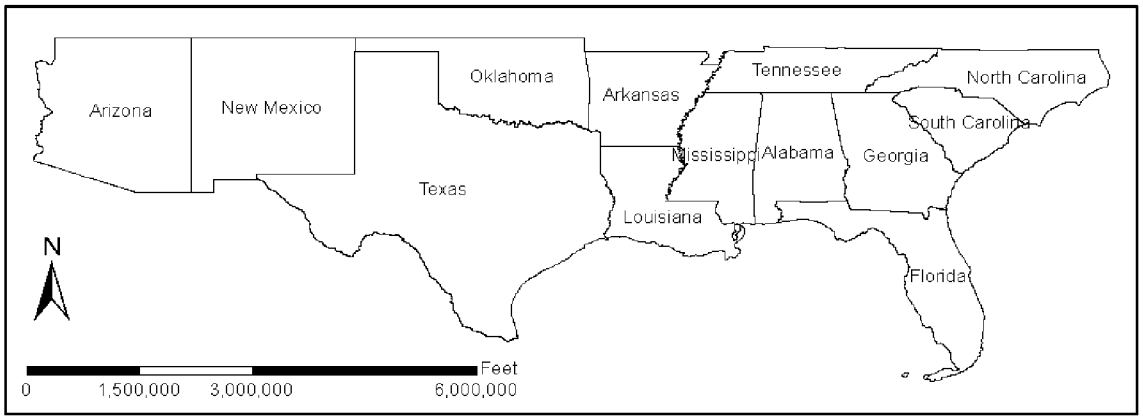

Figure 1: Road shape files of the thirteen southern states of the USA are used for experiments about the PostGIS database operations

Table 1: Index building time versus the size of spatial tables

Number of rows in spatial tables	Index building time	
	In milliseconds (ms)	In minutes
417	31	0.0005
204,114	11,875	0.1979
715,660	48,344	0.8057
2,792,865	200,062	3.3344
5,268,408	468,563	7.8094
8,481,083	965,016	16.0836
11,905,132	1,335,016	22.2503

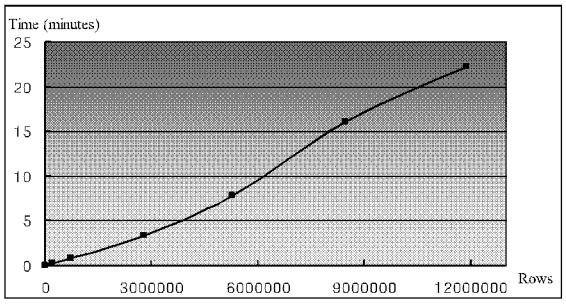

Figure 2: Index building time versus number of rows in spatial tables

3. Spatial Query Performance

3.1. Bounding Box Terminology

The efficiency of an index varies greatly according to the kind of data that is being stored, and so the scope for indexing spatial data, compared with common relational data is slightly different: for instance, different spatial data indexing methods are used for point, line and polygon data. In the case of collections of polygons, instead of indexing the object geometries themselves, whose shapes might be complex, an approximation of the geometry is deployed to index it. The approximation most commonly used is the so-called minimum bounding box.

For the sake of pedagogy, the PostGIS ST_Envelope (geometry) function, which returns a polygon representing the minimum bounding box of the supplied geometry, can be used to comprehend the bounding box term. The polygon is defined by the corner points of the bounding box ((MINX, MINY), (MINX, MAXY), (MAXX, MAXY), (MAXX, MINY), (MINX, MINY)). Degenerate cases will return a geometry of lower dimension than POLYGON, i.e. POINT or LINESTRING. In more specific terms, if a bounding box is a point then a POINT is returned, and if a bounding box is 1-dimensional, a LINESTRING is returned. Otherwise a POLYGON is returned. The following figures depict the bounding boxes of specific geometrical types (mostly polygons). Some degenerate cases are presented in the last ones.

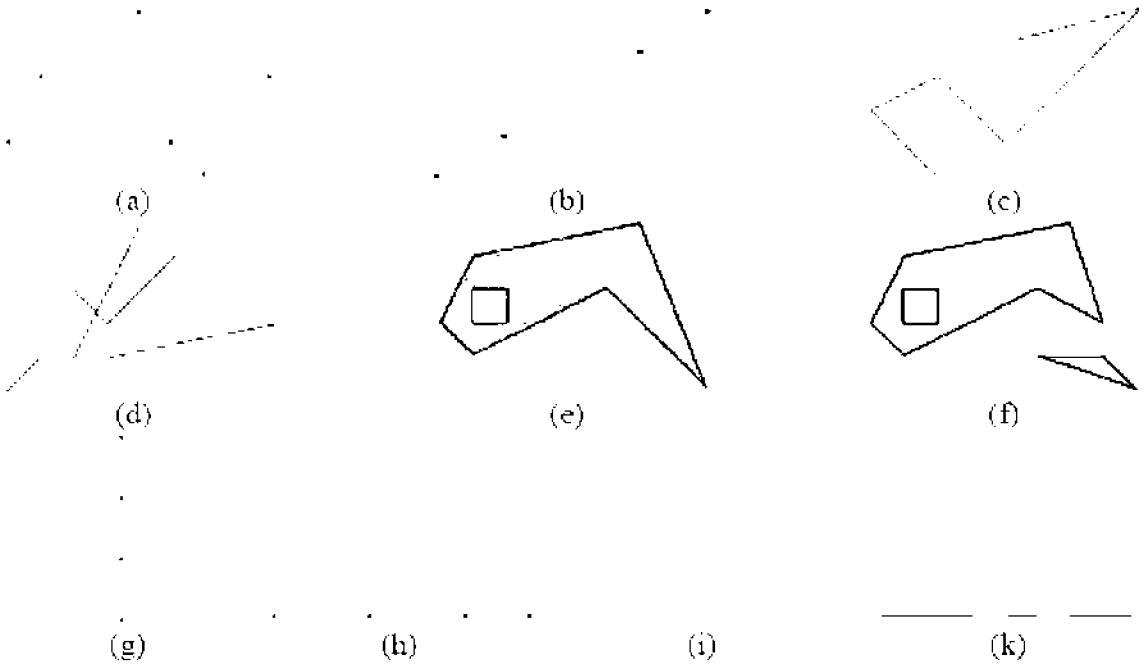

Figure 3: Bounding boxes corresponding to geometry types. Note some degenerate cases: (g)-(k)

The bounding box of a geometry in PostGIS is represented internally using float4s instead of float8s that are used to store geometries. The bounding box coordinates are floored, guaranteeing that the geometry is contained entirely within its bounds. Thus a geometry's bounding box is half the size as the minimum bounding rectangle, which brings significantly faster index building and general performance. But it also means that the bounding box is not the same as the minimum bounding rectangle that bounds the geometry. Noting that geometric testing against a bounding box is constant, it is clear that use of the bounding box as the geometric key for building the spatial index can save the cost of evaluating expensive geometric predicates during index traversal.

3.2. Indexed and Un-Indexed Query Evaluations

It is important to note that spatial indexes are not used automatically for every spatial comparison or operator. In fact, because of the "rectangular" nature of the R-Tree index, spatial indexes are only good for bounding box comparisons. Thus, all spatial databases implement a "two phase" form of spatial processing (Figure 4). The first phase is the indexed bounding box search, which runs on the whole table. The second phase is the accurate spatial processing test, which just runs on the subset returned by the first phase. In PostGIS, the first phase indexed search is activated by using the "&&" operator. This "&&"symbol has a particular meaning: "bounding boxes overlap". Actually, the definitions of indexed functions are SQL expansion functions that re-write the query using indexed operations (&&) and un-indexed functions. Indexed function calls will automatically include a bounding box comparison that will make use of any indexes that are available on the geometries. Un-indexed function calls will be performed conventionally by sequentially scanning all rows of tables.

It is however important to be aware that the implementation of this "two phase" spatial query model will not always bring advantage. The problem, which could be solved by applying the so-called The Oversized-Attribute Storage Technique (TOAST), appears if a table stores rather large geometries (over the commonly fixed page size: 8Kb), but not too many rows of them. Dealing with this issue, the PostGIS users are trying to compose the query estimation TOAST-aware designed to execute a process of two workarounds by adding a column in the table that contains the bounding box of geometry (Ramsey, 2008).

In order to compare performances (in terms of processing time) of indexed and un-indexed queries, experiments using popular spatial relationship functions were conducted on the spatial datasets that were mentioned in section 2.2. Four typical geometry relationship functions selected herein are: (spatially) intersect, cross, within, and cover. The pgAdmin III tool is used to measure the running time of queries. The average time of three tests on the same query performed on the specified table is recorded as the final processing time assigned for that query.

The PostgreSQL database configuration used for experiments is written out as follows.

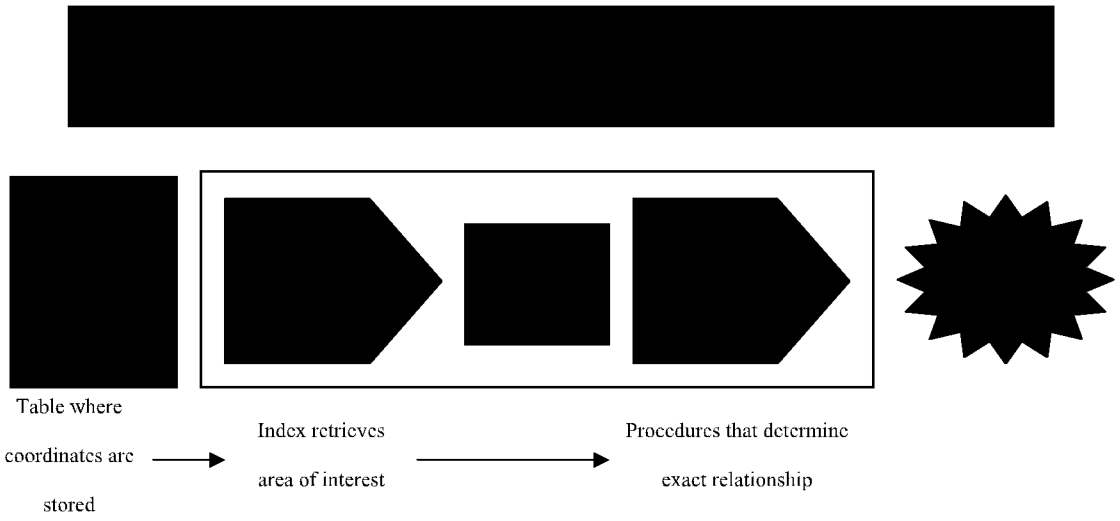

Figure 4: The two-step spatial query model: the first step benefits by spatial index and then spatial functions actually are involved only in the second step operation as conducted on the reduced dataset sub-setting by the first step

Optimal parameter configuration in terms of application is reflected in significantly faster processing time (Ramsey, 2007). Our tests however were all run in the same computer configuration as was the case for experiments described in section 2.2.

Five PostGIS-database tables with different sizes are created by "loading" ESRI road shape files. Table size depends on the number of roads that the shape file presents. The descriptions of shape files in terms of spatial coverage areas are presented case-by-case in Table 2.

Table 2: Spatial coverage of shape files used to create the PostGIS database

Databases having number of rows	**Shape files containing road network of areas, which are indicated by**
204,114	1 rectangle inside the Texas state's boundary (Figure 5)
2,792,865	1 state: Texas
5,268,408	5 states: New Mexico, Texas, Oklahoma, Arkansas, and Louisiana
8,481,083	8 states: Arizona, New Mexico, Texas, Oklahoma, Arkansas, Louisiana, Mississippi, and Tennessee
11,905,132	13 states: All states appearing in Figure 5

3.2.1. Intersects and Crosses Functions Experiments

Figure 5 represents the location of the second spatial parameter (across long line) of the "intersects" and "crosses" functions applied to experimental queries.

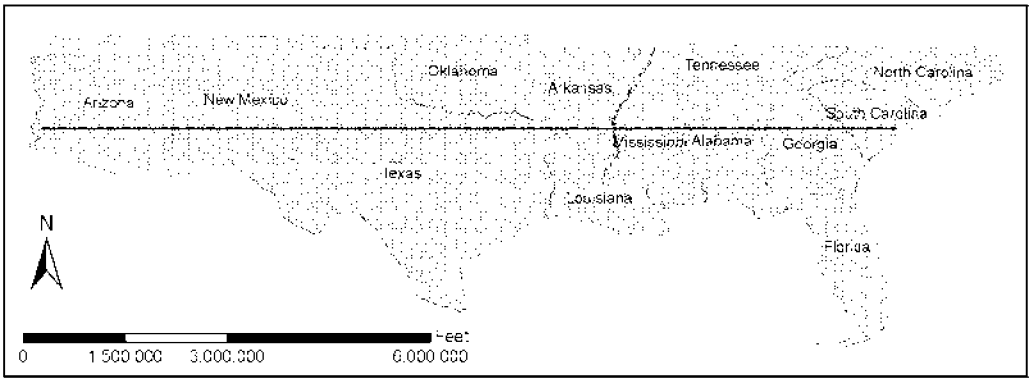

Figure 5: Spatial position of the across long line used for queries with the "Intersects" and "Crosses" functions

[redacted]

Table 3: Performing time of "intersects" indexed and un-indexed queries

Row number	**ST_Intersects vs _ST_Intersects**	
	Indexed timing (ms)	**Un-indexed timing (ms)**
204,114	109	1,500
2,792,865	282	23,359
5,268,408	485	42,469
8,481,083	875	69,453
11,905,132	1,016	126,063

Table 4: Performing time of "crosses" indexed and un-indexed queries

Row number	ST_Crosses vs _ST_Crosses	
	Indexed timing (ms)	Un-indexed timing (ms)
204,114	110	1,578
2,792,865	281	24,156
5,268,408	485	42,703
8,481,083	906	67,891
11,905,132	1,031	118,734

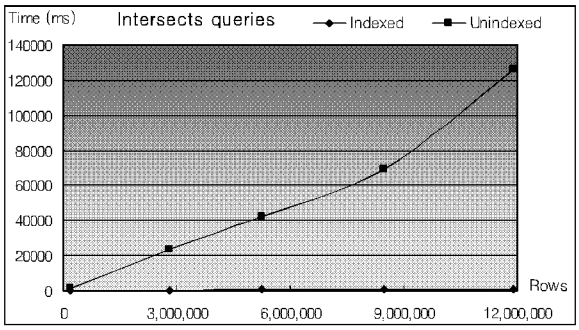

Figure 6: Diagram illustrated performing time of "intersects" queries

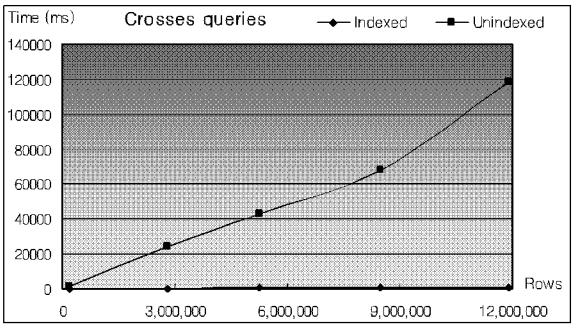

Figure 7: Diagram illustrated performing time of "crosses" queries

The larger the size of database tables, the more remarkably the difference between indexed and un-indexed queries occurs. For instance, in case of the "intersects" function, the ratio of performing time between indexed and un-indexed queries in the 200 thousand-row table is approximately 14 whereas this number is about 115 in the 11 million 900 thousand-row table. The same situation is recorded in the "crosses" queries. With regard to the tendency of performing time towards the size of tables, both indexed and un-indexed queries offer nearly linear dependences. Nevertheless, the variance against table sizes of un-indexed queries is much higher than that of indexed queries.

3.2.2. Within and Covers Functions Experiments

Two scenarios are investigated in order to evaluate the performance of queries that take different spatial (geometry) parameters. The first scenario is tested with a large rectangle polygon (Figure 8) that spatially exceeds the coverage of some small tables.

In contrast, the second scenario uses a small triangle polygon (Figure 11), which is completely inside the coverage of spatial tables, as a spatial parameter for "within" and "covers" functions.

Scenario 1

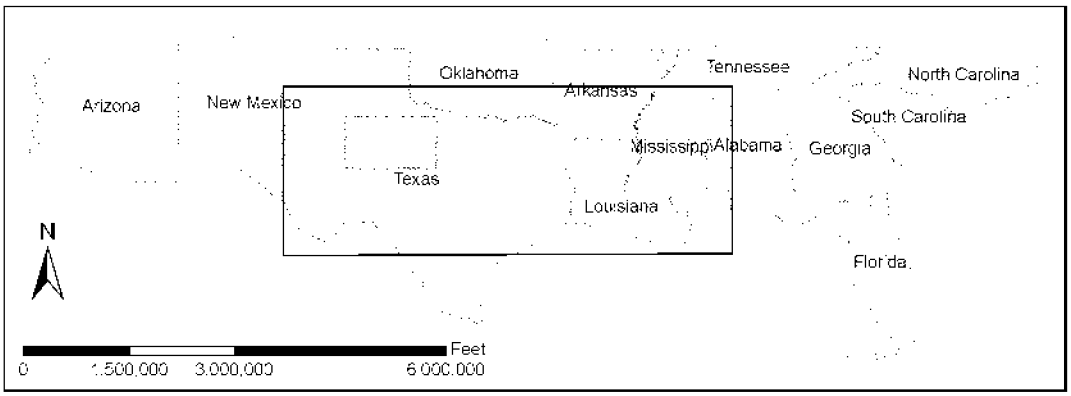

Figure 8: The large rectangle polygon demonstrates the geometry used for queries with the "Within" and "Covers" functions

Table 5: Performing time of "within" indexed and un-indexed queries in scenario 1

Row number	ST_Within vs _ST_Within	
	Indexed timing (ms)	Un-indexed timing (ms)
204,114	63109	62593
2,792,865	760500	750953
5,268,408	1040031	1076922
8,481,083	1379156	1503391
11,905,132	1514312	1685281

Table 6: Performing time of "covers" indexed and un-indexed queries in scenario 1

Row number	ST_Covers vs _ST_Covers	
	Indexed timing (ms)	Un-indexed timing (ms)
204,114	61937	61000
2,792,865	721344	715172
5,268,408	1173390	1101359
8,481,083	1395969	1411172
11,905,132	1451735	1618891

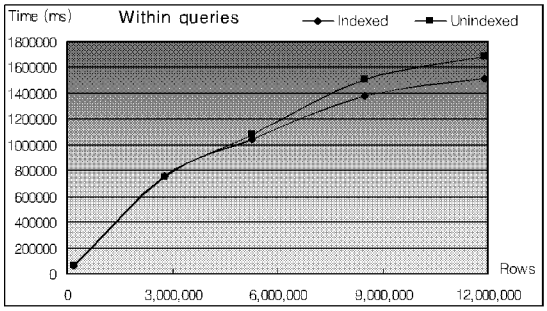

Figure 9: Diagram illustrated performing time of "within" queries in scenario 1

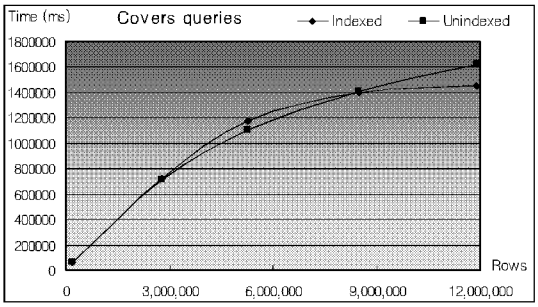

Figure 10: Diagram illustrated performing time of "covers" queries in scenario 1

Clearly shown, is the inefficiency of indexed queries in the first scenario when the container parameters (rectangle) in "within" and "covers" functions are spatially larger than the coverage of experimental road shape files. Take, for example, the first two cases of both "within" and "covers" experiments in scenario 1.

From the outcomes of queries, time consumed by indexed queries is higher than time consumed by un-indexed queries. In these cases, the resultant sets of queries are all records of tables so that the two-phase indexed queries take a longer time than the un-indexed queries, which simply use the sequential scanning method.

Scenario 2

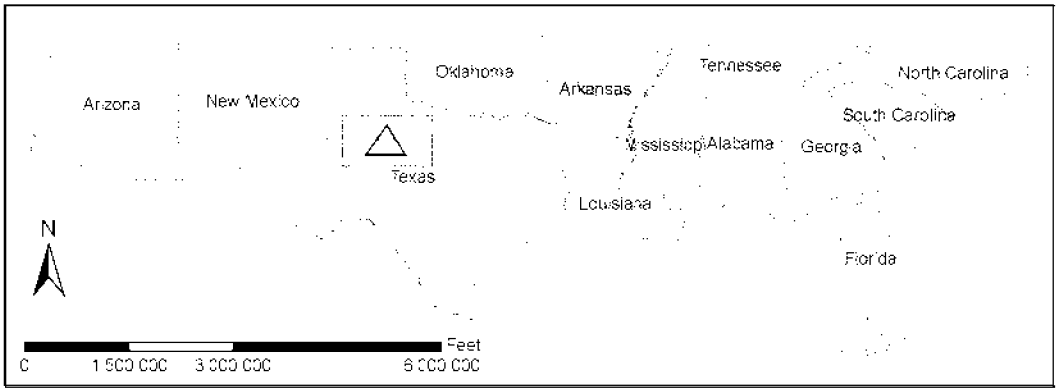

Figure 11: Spatial position of the small triangle polygon used for queries with the "Within" and "Covers" functions

[redacted]

Table 7: Performing time of "within" indexed and un-indexed queries in scenario 2

Row number	ST_Within vs _ST_Within	
	Indexed timing (ms)	Un-indexed timing (ms)
204,114	14797	16765
2,792,865	15671	29938
5,268,408	15812	47468
8,481,083	15954	71094
11,905,132	16235	118844

Table 8: Performing time of "covers" indexed and un-indexed queries in scenario 2

Row number	ST_Covers vs _ST_Covers	
	Indexed timing (ms)	Un-indexed timing (ms)
204,114	14734	15609
2,792,865	15265	29156
5,268,408	17547	39078
8,481,083	18125	53080
11,905,132	19547	115782

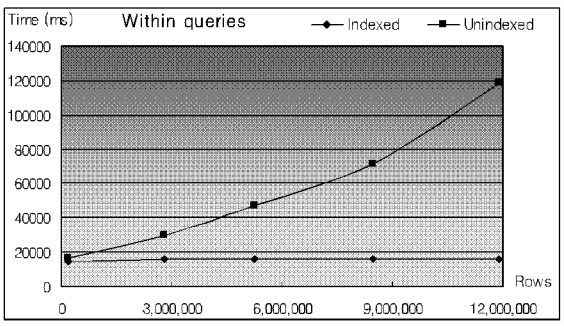

Figure 12: Diagram illustrated performing time of "within" queries in scenario 2

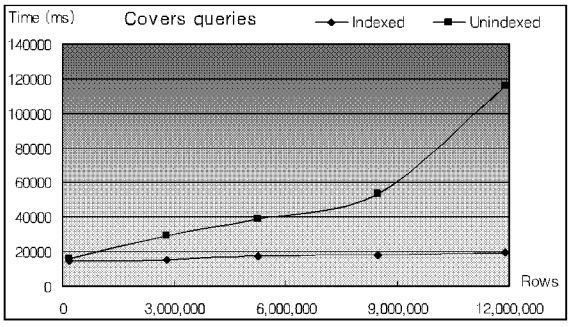

Figure 13: Diagram illustrated performing time of "covers" queries in scenario 2

Scenario 2 typifies the kind of spatial query advantaged by taking the indexed approach: obvious/significant advantages accrue when only the relevant records are called for. As the table size rises, the difference between the indexed and un-indexed approach to queries becomes wider. The indexed query performing time varies little with table size: 14 thousand milliseconds for the 200 thousand-row table whereas 16 thousand milliseconds are for the 11 million 900 thousand-row table. On the contrary, the performing time of un-indexed queries tends to be exponentially increasing with the size of tables.

Examining both scenarios simultaneously, with the same database and the same spatial-relation function in queries, but the different geometric parameters in functions, can lead to differences in performance time. For example, with the same 200 thousand-row table and using the "covers" query, the scenario 1 takes around 62 thousand milliseconds for the indexed query and 61 thousand milliseconds for the un-indexed query whereas in the scenario 2, these numbers are only 15 thousand milliseconds and 16 thousand milliseconds respectively. This is due to the difference in spatial parameters of functions: the scenario 1 uses the large rectangle whereas the scenario 2 uses the very small triangle.

4. Conclusions

PostGIS is an extension to the PostgreSQL relational database system which provides spatial database functionalities: spatial objects, spatial indexing, standard input/output representations, spatial functions and spatial operators. Building spatial indexes in PostGIS is a relatively time-consuming task. The implementation of GiST indexes in PostGIS has included a near linear-time algorithm for building R-Trees, which improved substantially, the scalability and performance of database operations.

The results reported here also emphasize the indispensable role of the GiST indexing scheme and indexed spatial queries in PostGIS databases. Without them, PostGIS spatial databases would simply serve no practical purpose because size of GIS objects is usually rather large. The R-tree-over-GiST spatial index has been implemented for high speed spatial querying. It is very important when the gap between indexed and un-indexed performances, is significant, as is shown in this study.

References

Ang, C. H., and Tan, T. C., 1997, New Linear Node Splitting Algorithm for R-trees, *Proceedings of the 5th International Symposium on Advances in Spatial Databases*, 339-349.

Gavrila, D. M., 1994. R-tree Index Optimization, *Proceedings of the 6th International Symposium on Spatial Data Handling*, 771-791.

Guttman, A., 1984, R-Trees: a Dynamic Index Structure for Spatial Searching, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 47-57.

Hellerstein, J. M., Naughton, J. F., and Pfeffer, A., 1995, Generalized Search Trees for Database Systems, *Proceedings of the 21st VLDB Conference Zurich, Switzerland*, 562-573.

Ramsey, P., 2004, *PostGIS Introduction and Evaluation*, Refractions Research Inc.

Ramsey, P., 2007, *Introduction to PostGIS: Installation, Tutorial, Exercises*. Refractions Research Inc.

Ramsey, P., 2008, *PostGIS Manual*. Refractions Research Inc.