Spatial Data Handling in PostGIS

Degree Project in Computer Science, First Level

Dorothea Andersson
doa@kth.se

Supervisor: Michael Minock
Royal Institute of Technology (KTH), Sweden

Abstract. This thesis discusses the concept of spatial data, and focuses on a theoretical example to demonstrate the need for unique ways of modeling and representing such data. PostGIS is used to extend PostgreSQL to solve the presented problem and demonstrate methods for optimizing the solution queries. The behavior of PostGIS is studied and commented on, indicating there is still a long way to go before this kind of data can be handled in an equally straightforward manner as more traditional data.

1 Introduction

Information collection has become a cornerstone in today's society, and we are now heavily depending on information systems for communication and data storage. Huge spatial databases required for the development of, for example, robotics and geographical information systems, as well as medical imaging, containing terabytes of data have put query optimization in a whole new light [1]. Although extensive research on the topic has been done over the past two decades, complex spatial query processing and optimization remains a highly active field that presents us with many nontrivial issues. This thesis aims to explore different aspects of optimizing complex queries with focus on spatial data retrieval using PostGIS.

1.1 Overview of Article

Section 2 introduces the unique issues spatial data handling presents and what has been done to deal with these. Section 3 introduces the problem studied throughout this thesis, including 3.1 explaining how the spatial data types point and polyline can be represented in a traditional relational database, further motivating the need for spatial data types and functions. Section 4 explains the experimental setup and how the tests were executed, before section 5 shows the test results and section 6 discusses said results. Finally, some conclusions about what is currently happening within this field of research are drawn in section 7.

2 Background

To understand the issues that data retrieval from spatial databases pose, we must first understand the nature of spatial data, and why traditional database systems in their current form are not well suited for this kind of heavy duty, multidimensional data handling.

It is first and foremost the nature of spatial data that renders traditional database systems useless. To be properly represented, it requires more complex data types than traditional one dimensional types such as strings and integers. Spatial database applications must handle data types such as points, polylines and polygons - often in three dimensions. The sheer volume of data required to represent a geometric object is also significantly larger - a lake boundary might need a thousand vertices for sufficiently accurate representation, and one low resolution satellite image of the US can consume as much as 30 MB of disk space. [1] The complexity of functions involved in a spatial application is comparable to those in programming language applications, and the storage requirements are generally more severe.

A unique feature of spatial data is that the natural medium of interaction with the user is visual rather than textual. This means more constructs are needed to provide a representation that is closer to our perception of space. Also, because of the natural lack of order in multidimensional space, both traditional clustering techniques and indices need to be evolved and refined to meet the needs of this kind of data handling [1]. There is much demand for effective algorithms that can discover useful patterns from large and complex spatial databases in order to reduce search spaces, minimize unnecessary view materialization, and speed up query processing.

Over the past twenty years, research has led to the development of spatial relations that model topological relationships between objects in space, spatial extensions to SQL, as well as methods for spatial storage and indexing. The common denominator for all these is that they take into account the geometric aspects of spatial data, and they are all a response to the ever increasing need in industry to efficiently handle such data in commercial database environments.

3 Approach

Let us assume we have the database of a distribution company containing all the usual information such as addresses, names, customer numbers, order information, employees, and so on. This seems straightforward enough, and is in fact just a simple, relational database with some basic data types such as strings and numbers. With this setup we can easily answer any questions we might have about customer information, order status and employees. The trouble comes when we want to know which distribution office the customer belongs to. Even a seemingly simple query such as *"list all customers who reside within thirty miles of this and this office"* will confound our database. Listing all employees that live a certain distance from the office for the purpose of determining who is eligible for a travel expense refund would present us with the exact same problem.

To process this query, the database system will have to transform the customer addresses, as well as the addresses of the offices, into a suitable reference system such as longitude and latitude, in which distances can be computed and compared. Then, it will have to scan through the entire customer list, compute the distance between customer and office, and then compare this to the requested distance of 30 miles. Since traditional indices are incapable of ordering multidimensional coordinate data, a regular index cannot be used to narrow down the search [1].

Thus it only takes a simple legitimate business query to send a traditional database management system (DBMS) into a hopeless tailspin, and in this example we are still only dealing with a two dimensional space. The need for databases tailored for handling spatial queries is obvious, and then we haven't even considered the implications of modeling the three dimensional world we are living in.

The problems actually start emerging already when we consider how to store the geometric points. There is no data type *point* in a traditional relational database system, and there is no natural way of mapping spatial relations onto such a database. The problems caused by the lack of geometric data types could potentially be solved by creating a collection of tables with overlapping attributes, but this is far from straightforward and not at all computationally efficient.

3.1 Geometric Data Representation

The two dimensional point (x,y) is the most simple geometric data type, and forms the basis for all other more complex geometric types such as polygons and polylines. Points could be represented in a traditional relational database by simply adding a table *point* with the three columns point-id, x-coordinate and y-coordinate. Any point in use would have to be stored as a separate tuple in this table and be referenced with its own id.

What if we consider the implications of representing a simple polyline instead ? A polyline is a series of connected line segments, typically used to approximate the shape of a river or the boundary of a country. Let us assume we have a simple polyline forming a square with corner points (0,0), (1,0), (1,1) and (0,1) as shown in *Figure 1* below. How would we represent this in a non-spatial database ? We would definitely need the previously suggested point table. We would also need a means to represent a line with two endpoints, and a means to connect individual lines to form a boundary. This could be done using the three tables in *Fi-*

gure 2 [1]. A polyline can then be referenced by its unique boundary-ids. It should be obvious that this is not a very efficient or straightforward approach to storing and handling spatial data - just imagine the implications of extending this approach to three dimensions!

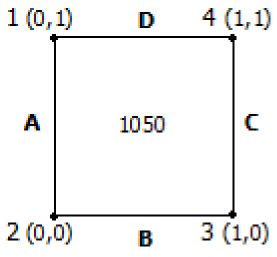
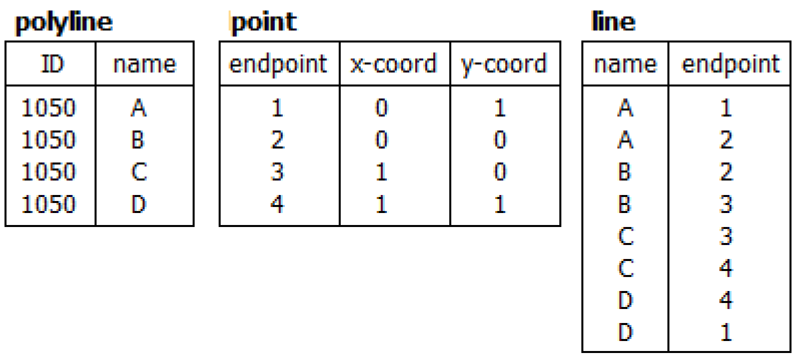


FIGURE 1 – A simple polyline.



polyline

ID	name
1050	A
1050	B
1050	C
1050	D

point

endpoint	x-coord	y-coord
1	0	1
2	0	0
3	1	0
4	1	1

line

name	endpoint
A	1
A	2
B	2
B	3
C	3
C	4
D	4
D	1

FIGURE 2 – Polyline representation in a traditional DBMS.

For simplicity, we will assume that the distribution company determines office memberships in a point a to point b manner. Basing this judgment on the actual street wise distance is not within the scope of this thesis, but it should be obvious that the computational complexity added by such an approach would be immense in comparison to the approach used here. Therefore, what we mean by the range query *"within thirty miles"* is to list all customers that live within a *radius* of thirty miles from the specified office. In reality, wanting to know the actual driving distance from an office to a customer could of course be an equally valid query.

Consider then the convenience of being able to, with one or several queries to the same database, get a list of all the company's customers, complete with their nearest office. Add to that being able to rely completely on this result and that no customer has been left out or has been matched with the wrong office or counted twice. From a business point of view this would be great and save a lot of time and money, but to implement such a query would be far from trivial. In the context of what we can imagine useful in a business environment, the example here is very simple and yet effectively shows the complexity of and the challenges this area presents us with.

Furthermore, we will omit the transformation of address strings to longitude-latitude pairs, since this computation simply cannot be avoided.

4 Implementation

For testing purposes, PostGIS has been used to “spatially enable” PostgreSQL. PostGIS is an open source extender for PostgreSQL which adds support for a number of spatial functions such as distance and area, as well as geometry data types like points, polygons and polylines [2].

The aim of the following experiments has been to show examples of what can be done to optimize spatial queries, further motivating the need of the functionality offered by the different solution approaches available. Also, a goal has been to build up enough knowledge to draw some conclusions on what is currently happening within this active field of research.

To solve our initial problem of the thirty mile radius, a test database was set up containing a thousand customers and three offices in accordance with the relational diagram shown in *Figure 3* below. This database forms the basis for both our radius based and the following region based range queries, as will be shown later on.

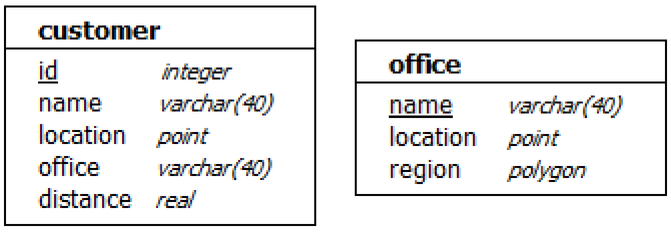


FIGURE 3 – Test database part 1.

Since we are omitting the address transformation, our initial queries will simply be based on the distances between the customer locations and the office location in a very straightforward manner. In reality one might have to consider what happens to the customers who live precisely between two offices, and if the radius approach is even realistic.

Let us say the company has three offices, and they deliver to all of Sweden. Then it would probably divide the whole of Sweden into three regions, each one being handled by its own office. Considering the shape of any country, the circle approach wouldn’t make much sense - rather these regions would most likely be represented by irregularly shaped geometric objects such as polygons. Such an example will also be shown later on.

The above mentioned examples are rather simple ones, but what if we wanted to know what customer was closest to a particular office instead ? Perhaps the office wants to start their delivery round with the closest customer and calculate the shortest delivery round based on that. How do we find that first customer ? Based on our first problem, the naive approach would be to ask for all customers within a small radius from the office, and then decrease the size of that radius until we are left with a limited number of customers among which we can determine the closest one. This is known as a Nearest Neighbor query, and is a common problem with many applications [3].

For testing purposes, a random point generator was used to generate 3000 points within a bounding box of size 50 by 50 units [4]. These were added to the test database along with a table of polygons representing bounding boxes of sizes ranging from 0 by 0 units up to the whole point range of 50 by 50 units. The problem here is to, given the set of random points, see what happens in PostGIS when we want all points within each one of these bounding boxes, sorted by distance from the given point. Does it use an index ? If so, what kind, and can we improve it somehow ? If not, what can we do to add one ? How long does it take to process this query depending on the size of the bounding box ?

The entire test database contained the tables shown in Figure 3 above, as well as the two tables in Figure 4 below.

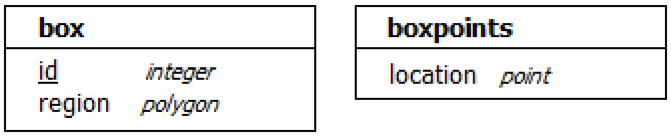


FIGURE 4 – Test database part 2.

5 Results

5.1 Radius Based Range

First, we simply query our test database for the id, name and location of all customers within a radius of 8 units from the Stockholm office. That is, we ask for all customers where the distance between the customer point and the office point is less than 8. To do this, we use the distance operator <->, which calculates the distance between two points in space :

```
SELECT *
FROM public.customer, public.office
WHERE office.name = 'Stockholm'
     AND (customer.location <-> office.location) < 8;
```

The average runtime based on ten consecutive runs for this query was 36 ms, and when we take a look at the query plan given by EXPLAIN ANALYZE, we find that it performed as expected :

```
1. Nested Loop
2.      Join Filter: (customer.location <-> office.location) < 8
3.      -> Index Scan using office pkey on office
4.           Index Cond: (name = 'Stockholm')
5.      -> Seq Scan on customer
```

A sequential scan is used to go through the entire customer list, and since office.name is the public key of the office table, an index scan is used to find the Stockholm office tuple. The two resulting relations are then combined using a nested loop join with our specified distance condition.

This time result might not seem very impressive, but keep in mind that we only have a thousand customers, and that this query actually only returns 400 of these. In fact, if we turn this into a non-spatial query and use the pre-calculated office memberships we already have in our database (and by doing so omitting the distance calculation altogether), the average runtime becomes 21.3 ms. Clearly, the distance calculation in the spatial query adds considerable complexity, and it should be safe to assume that this time difference would be quite noticeable given more realistic data sets.

5.2 Region Based Range

Next, we will leave the radius approach and instead assume that the three different office regions are represented by polygons, and that a customer belongs to a certain office if he or she lives within the boundaries of that office's region polygon. We still have a thousand customers in the database of whom 400 belong to Stockholm. The Stockholm region used here is now a polygon with 50 vertices rather than the radius of 8 units used above.

```
SELECT *
FROM public.customer, public.office
WHERE office.name = 'Stockholm'
     AND customer.location <@ office.region;
```

The average runtime based on ten consecutive runs for this query was 15 ms, and using EXPLAIN ANALYZE, we find that it is executed in the same way as the radius based query. A sequential scan is used on customer and an index scan on office, and the resulting relations are combined using a nested loop join with the containment condition location <@ region [5].

```
1. Nested Loop
2.      Join Filter (location <@ region)
3.      -> Index Scan using office pkey on office
```

```
4.          Index Cond: (name = 'Stockholm')
5.      -> Seq Scan on customer
```

We can clearly see that this approach is faster than the radius approach, enabling us to conclude that the containment function <@ is computationally cheaper than the distance function <->. We can also see that PostGIS doesn't differentiate between these two functions - the query is executed the same way no matter which function we use.

5.3 Constrained Nearest Neighbor

5.3.1 Default PostGIS

Now, let us consider the nearest neighbor problem - finding the customer who is closest to a given office location. What we get is in fact a constrained nearest neighbor (CNN) query, since we have an outer boundary within which all our customer points are located [3]. We are now using the second part of our test database (see *Figure 4*); the *boxpoints* table which contains 3000 customer points, and the *box* table with 51 different sized bounding boxes, the largest one containing all 3000 points.

Since these points were randomly generated within a bounding box of 50 times 50 units, we will assume that they are evenly distributed, and study the query runtime effects depending on the number of points in each bounding box query as the box size decreases. The query looks as follows, and has been run ten times for each bounding box size :

```
SELECT public.boxpoints.location
FROM public.boxpoints, public.box
WHERE box.id = 1 AND boxpoints.location <@ box.region
ORDER BY boxpoints.location <-> '(0,0)';
```

The average runtime based on ten consecutive runs for this query on the outermost bounding box (50 times 50 units) was 54.3 ms, and the effects of the decreasing bounding box sizes can be seen in *Figure 5.*

The graph above is a quadratic approximation of the resulting runtimes, and has been plotted against the number of points contained in each box, ranging from 54.3 ms for the outermost box containing 3000 points, to 11.6 ms for the innermost box containing zero points. Clearly, the average runtime is approximately linear with respect to the number of points in the query bounding box.

Similar to the previously run queries, EXPLAIN ANALYZE reveals that a sequential scan is used on boxpoints, and an index scan on box using the primary key box.id. These are joined together using a nested loop join on the region containment condition. The resulting points are then sorted using quicksort based on their distance to the given point. What is most interesting to note here is that even though we have quite a few points, PostGIS does not add an index to them to speed up the search. Clearly, the behaviour of the query optimizer is rather naïve in this case, and the generated query plan is far from ideal.

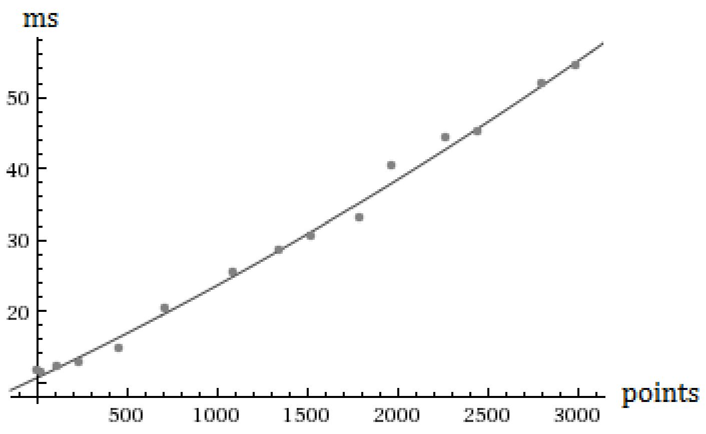


FIGURE 5 – Average runtime (ms) for increasing number of points in bounding box.

5.3.2 GiST on Location Points

Let us add an index on the *boxpoints* table and see what happens. Since this table only contains points, a spatial index must be used. A Generalized Search Tree (GiST) index was created on *boxpoints.location*, and then the query presented in section 5.3.1 was run again [5]. The average runtime based on ten consecutive runs this time was 44.7 ms (as opposed to 54.3 ms without the index). There is a difference, but what has happened ? Not very much, as it turns out. The only difference is that a bitmap heap scan is used on *boxpoints* using our newly created index, rather than the previously used sequential scan. Clearly, this simple change of search algorithm makes a big difference.

Repeating the same test as before with decreasing bounding box sizes, we find that the result is still approximately linear, and consistently faster than without the index.

6 Discussion

Particularly interesting to note is that PostGIS only seems to use nested loop joins as a default, which hardly can be considered ideal [6]. Like traditional joins, spatial joins are expensive and notoriously difficult to optimize, and there are numerous pieces of work out there discussing the use of different join algorithms - many of them promoting the spatial semijoin. The focus has been on developing methods for reducing the size of the relations involved, and by using a semijoin, approximations can be used to reduce the search space [7].

The query optimizer is ultimately responsible for choosing the most efficient execution plan from all logically equivalent expressions available for a certain query [8]. As we have seen, nothing beyond the most naïve approach , and no index beyond those manually created, are used in this case. It seems the only way of optimizing query performance here is by manual indexing and rewriting. Clearly, a lot could be done to increase the intelligence of the PostGIS query optimizer.

7 Conclusion

The development of optimizing techniques for spatial query processing has been heavily influenced by existing solutions for traditional relational databases, resulting in extensions to already existing systems and query languages. Relational databases were developed for efficient storage and retrieval of massive amounts of data in its most basic forms - not for handling spatial data.

Since the relational model and SQL are so widely used, this is what has been adopted to the needs of spatial data handling, resulting in extensions such as PostGIS. As we have seen, there is a lot of work to be done before industry can start letting go of their go to method of using EXPLAIN ANALYZE and manually rewriting queries. Everything is a work in progress, and no real standards are in place even though a possible route for future development is emerging in the form of field and object based approaches to modeling spatial data, as well as suggested standards for spatially enabling SQL. Along with the approximation-refine paradigm, this is leading the way into the future of spatial information handling.

References

1. Shekar, S., Chawla, S. : Spatial Databases : A Tour. Prentice Hall, ISBN 0130174807.
2. PostGIS. http ://postgis.refractions.net/ (2012-04-01).
3. Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., El Abbadi, A. : Constrained Nearest Neighbor Queries. Springer-Verlag Berlin, 2001.
4. Random Point Generator. http ://www.geomidpoint.com/random/ (2012-04-03).
5. PostgreSQL 9.1.3 Documentation.
6. Silberschatz, A., Korth, H.F., Sudarshan, S. : Database System Concepts. McGraw-Hill, 6th edition.
7. Tan, K-L., Ooi, B.C., Abel, D.J. : Exploiting Spatial Indexes for Semijoin-Based Join Processing in Distributed Spatial Databases. IEEE Transactions on Knowledge and Data Engineering, vol. 12, No. 6, November/December 2000.
8. Chaudhuri, S. : An Overview of Query Optimization in Relational Systems. Microsoft Research.