Manage spatial data with IBM DB2 Spatial Extender, Part 2: Implementing typical spatial use cases

Tips and techniques for developing efficient spatial applications

David Adler
Consultant, Adtech Geospatial

March 2015
(First published 13 December 2012)

This series of tutorials takes you through common tasks that work with spatial data in the DB2® Spatial Extender. These tasks include importing and creating spatial data, constructing and executing spatial queries, working with IBM spatial tools, working with third-party and open source software, improving application performance, and taking special considerations in a data warehouse environment.

View more content in this series

Before you start

About this series

For more than 10 years, the DB2 Spatial Extender has provided the capability of managing and analyzing spatial data stored in DB2 relational tables. Table columns can be defined with spatial types for points, lines, and polygons, which can represent objects such as the following:

- Point locations, such as customers, retail stores, electrical transformers, cell towers
- Lines, such as highways, coastline, delivery routes, electrical transmission lines
- Polygons, such as sales or service territories, flood or fire risk, states, provinces, counties

SQL queries can incorporate spatial functions to analyze spatial relationships like finding customers within a flood zone. Listing 1 shows an example of code that incorporates spatial functions.

Query that incorporates spatial functions

```
SELECT cust_name, cust_addr
FROM customers, floodzones
WHERE ST_Within(cust_loc, flood_loc) = 1
```

The result of this query would produce a set of customers names and addresses whose houses could be at risk of flooding. This information could be useful in calculating insurance rates or sending out an offer to purchase flood insurance.
This "Manage spatial data with IBM DB2 Spatial Extender" series of tutorials takes you through common tasks that work with spatial data in the DB2 Spatial Extender. These tasks include importing and creating spatial data, constructing and executing spatial queries, working with IBM spatial tools, working with IBM third-party and open source software, improving application performance, and considering special circumstances in a data warehouse environment.

Although the focus of this series is DB2 Spatial Extender on Linux®, UNIX®, and Windows®, many concepts are applicable to other IBM database offerings with spatial capability, including:

- Spatial support for DB2 for z/OS®
- Informix® Spatial DataBlade®
- Netezza Spatial

This series includes:

- **Part 1: Acquiring spatial data and developing applications**
  This tutorial introduces you to the technology provided by DB2 Spatial Extender and approaches for populating DB2 tables with spatial data, a critical first step before spatial queries can be implemented. This tutorial also addresses environments for executing spatial queries, including the DB2 command-line processor, IBM Data Studio and embedded in an application using JDBC, CLI, and other interfaces.

- **Part 2: Implementing typical spatial use cases**
  There are a number of typical use cases in spatial application that include spatial joins, distance queries, geofencing, and nearest-n queries. This tutorial provides examples for implementing these use cases efficiently.

- **Part 3: Spatial application tuning**
  Application performance is always an important consideration. This tutorial looks at factors in spatial application performance and tools for understanding query behavior. Some of the factors considered include inline data, data clustering, compression, and generalization.

- **Part 4: Spatial analysis in a data warehouse environment**
  Achieving good scalability in an InfoSphere Warehouse nothing-shared environment requires additional considerations. This tutorial addresses the techniques to get good spatial query co-location. It also provides special considerations when working with Esri middleware.

- **Part 5: Spatial visualization tools from IBM, third parties, and open sources**
  Visualization of spatial data is often an important aspect of a spatial application. This tutorial introduces some of the tools available, including the free IBM geobrowser, Esri software, web-mapping services, such as Google maps, and open source software that uses the GeoTools driver for DB2.

**About this tutorial**

There are a number of typical use cases in spatial application that include distance, nearest-n, point in polygon, spatial joins, and window queries. This tutorial provides examples for implementing these use cases efficiently.
Prerequisites and system requirements
This tutorial is for application architects and developers who have a working knowledge of DB2 and SQL.

In order to work through the steps of the tutorial, you need a working DB2 environment with Spatial Extender installed and configured. Information on setting up this environment can be found in the Resources section.

Getting started
Tutorial environment
IBM DB2 Server with Spatial Extender
In order to run the examples, you will need a working installation of DB2 with Spatial Extender. If you already have it, you can skip the rest of this section.

Starting with DB2 10.1, Spatial Extender is included in all DB2 server installation images. The free DB2 Express-C works well for this tutorial and can be downloaded via the link in the Resources section.

In order to include the spatial functionality, select the Custom option during the DB2 installation and Spatial Extender under both the Server and Client selections.

NOTE: If you use the 10.1 DB2 Express-C installation image, it must be at Fixpack 1 or greater.

IBM Data Studio
The examples were developed with IBM Data Studio 3.1.1. IBM Data Studio is an Eclipse-based environment you can use to develop and test SQL queries, highlight SQL syntax, and show the results of the query execution in tabular form. There is no charge for Data Studio. For more information on downloading and installing Data Studio, see the Resources section. For details about how to set up a development project, see Data Studio overview.

In order to simplify the syntax of the spatial functions and predicates, the schema DB2GSE has not been explicitly prefixed to the spatial function names. The schema DB2GSE can be implicitly included by specifying DB2GSE in the default function path. You can specify this in the dialog when the Data Development project is created in Data Studio with a list like SYSIBM, SYSFUN, SYSPROC, DB2GSE. The default function path can also be specified in the project properties in the Database Connection tab. It is important to set the default function path before creating the queries for it to be applied to the query when executed.

Other query execution environments
You can run the queries in any environment that allows you to execute SQL queries, whether as scripts executed from a command line or in another GUI tool.

To run the queries without specifying the DB2GSE schema, you should include in your script the command to set the function path with a statement like SET CURRENT FUNCTION PATH = CURRENT FUNCTION_PATH, DB2GSE;.
Tutorial sample scripts and data

You can download the sample SQL scripts and data from the Downloads section. Download and unzip these files to a convenient directory to continue.

This tutorial uses the same data files as Part 1: point data for hospitals and and polygon data for counties. Note that this is sample data and is not necessarily current or accurate. In your applications, the point data might represent customer, branch office, ATM, or other locations. Polygon data might represent service or marketing territories, fire or flood risk regions, or lakes. The sample SQL scripts include the explicit schema DB2GSE in the spatial function references. To install the sample data used in the tutorials, run the setup.sql script from a DB2 command window with the command db2 -tvf setup.sql.

If you are going to use a database other than the SAMPLE database, modify the setup.sql script to specify the name of the database you want to use.

Point(s) in polygon queries

Two of the most common types of spatial queries are to find what polygon contains a particular point location and to find what points are contained within a particular polygon. With DB2 Spatial Extender you can issue SQL queries from your application for queries like the following examples:

- Find the service territory containing a specified location
- Find the risk (fire, flood, hurricane, earthquake) polygon containing a customer location
- Find retail outlets contained within a specified polygon

Typically, spatial functions have one parameter for the spatial column in a table and another parameter for a spatial constant or a spatial column in a table. Here are some examples of spatial functions you can use for point(s) in polygon queries:

- **ST_Contains(geom1, geom2)**
  Returns '1' for geom2 values are completely contained by the polygon(s) identified by geom1. This does not include geom2 values, which touch or intersect the boundary of the polygon.
- **ST_Within(geom1, geom2)**
  This is the inverse of ST_Contains, returning '1' for geom1 values are within the polygon(s) identified by geom2.
- **ST_Intersects(geom1, geom2)**
  Returns '1' if geom1 and geom2 spatially interact in any way, touching, crossing or containing each other. For point and polygon values, this is generally the same as ST_Contains and ST_Within, but will also include points which fall exactly on the boundary of the polygon. It is important to note this difference if you get unexpected results.

The examples in this section will use a table with a spatial column for one parameter and a spatial constant for the other parameter. The following section on spatial joins will provide examples where both parameters are spatial columns.
All spatial functions are defined in the schema DB2GSE, which must be prefixed to the function name or specified in the DB2 CURRENT FUNCTION PATH. In the examples that follow, the CURRENT FUNCTION PATH has been set in Data Studio order to make the queries more readable.

**Which polygon contains this point?**

You can use the following queries to determine the county for a specified location. These queries return equivalent results except for the potential boundary condition mentioned above. If a point happens to fall exactly on a boundary that forms adjacent polygons, ST_Intersects will return polygons and ST_Contains and ST_Within will return neither.

The location is defined by a longitude and latitude value, which in a typical application might have been derived by a user clicking on a location on a map in a web browser. The screen location would need to be converted by application software to the longitude and latitude values.

**Contents of pip_query1.sql**

```sql
SELECT name 
FROM test.counties 
WHERE 
  ST_Contains(shape, ST_Point(-74.237, 42.037, 1)) = 1 
SELECTIVITY 0.0001
;

SELECT name 
FROM test.counties 
WHERE 
  ST_Within(ST_Point(-74.237, 42.037, 1), shape) = 1 
SELECTIVITY 0.0001
;

SELECT name 
FROM test.counties 
WHERE 
  ST_Intersects(shape, ST_Point(-74.237, 42.037, 1)) = 1 
SELECTIVITY 0.0001
;
```

These all result in the single county with the name Ulster being returned.

**Query results for pip_query1.sql**

<table>
<thead>
<tr>
<th>Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td></td>
</tr>
<tr>
<td>Ulster</td>
<td></td>
</tr>
</tbody>
</table>

**What is in this polygon?**

You can use the following queries to find the hospitals that are located within the specified polygon. The polygon is defined as a constant using the well-known text (WKT) representation. The polygon definition consists of the character string `POLYGON` followed by pairs of `x,y` coordinates separated by a comma. The individual `x` and `y` values are separated by a space. The entire list of coordinate pairs must be in parentheses.
The \( x \) and \( y \) values in this example are longitude and latitude values describing a rectangle one degree on each side for the sake of simplicity. A polygon definition might contain any number of coordinate pairs (well, less than about 500,000) and can represent a complex shape such as a lake or county boundary.

### Contents of pip_query2.sql

```sql
SELECT name
FROM test.hospitals
WHERE
  ST_Contains(ST_Polygon(
    'POLYGON ((-74.0 42.0, -73.0 42.0, -73.0 43.0, -74.0 43.0, -74.0 42.0))', 1)
  , location) = 1 SELECTIVITY 0.0001
;

SELECT name
FROM test.hospitals
WHERE
  ST_Within(location, ST_Polygon(
    'POLYGON ((-74.0 42.0, -73.0 42.0, -73.0 43.0, -74.0 43.0, -74.0 42.0))'
    , 1)) = 1 SELECTIVITY 0.0001
;

SELECT name
FROM test.hospitals
WHERE
  ST_Intersects(location, ST_Polygon(
    'POLYGON ((-74.0 42.0, -73.0 42.0, -73.0 43.0, -74.0 43.0, -74.0 42.0))'
    , 1)) = 1 SELECTIVITY 0.0001
;
```

These queries return 21 hospitals, only nine of which are shown in the result window.

### Query results for pip_query2.sql

<table>
<thead>
<tr>
<th>Status</th>
<th>Result 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Marshall Hospital</td>
</tr>
<tr>
<td>2</td>
<td>Southwestern Medical Center</td>
</tr>
<tr>
<td>3</td>
<td>Hillcrest Hospital</td>
</tr>
<tr>
<td>4</td>
<td>Saint Lukes Hospital</td>
</tr>
<tr>
<td>5</td>
<td>Adirondack Home</td>
</tr>
<tr>
<td>6</td>
<td>Albany Hospital</td>
</tr>
<tr>
<td>7</td>
<td>Albany Hospital For Incurables</td>
</tr>
<tr>
<td>8</td>
<td>Albany Hospital Sanatorium</td>
</tr>
<tr>
<td>9</td>
<td>Bellevue Hospital</td>
</tr>
</tbody>
</table>

### Spatial join queries

Just as queries can join two tables based on the relationship of values in character and numeric columns, spatial functions can be used to join tables based on the relationship of spatial values in table columns. This capability is very important for business intelligence and spatial analytic applications such as risk analysis and demographic analysis. The following examples use the customers and hospitals tables based on common customer cases.

### What's in a polygon that's defined in another spatial table?

You can use the following query to find the hospitals located within a specified county. The polygon representing the county is selected by the query expression `c.name = 'Dutchess'`. 
Contents of join_query1.sql

```
SELECT h.name, c.name
FROM test.hospitals AS h, test.counties AS c
WHERE ST_Intersects(h.location, c.shape) = 1 SELECTIVITY 0.0001
AND c.name = 'Dutchess'
```

This query returns eight hospitals.

### Query results for join_query1.sql

<table>
<thead>
<tr>
<th>Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>NAME</td>
</tr>
<tr>
<td>1</td>
<td>Hudson River State Hospital</td>
</tr>
<tr>
<td>2</td>
<td>Vassar Brothers Hospital</td>
</tr>
<tr>
<td>3</td>
<td>Beacon Hospital</td>
</tr>
<tr>
<td>4</td>
<td>Harlem Valley State Hospital</td>
</tr>
<tr>
<td>5</td>
<td>Marcellin State Hospital</td>
</tr>
<tr>
<td>6</td>
<td>New York State Hospital</td>
</tr>
<tr>
<td>7</td>
<td>Saint Francis Hospital</td>
</tr>
<tr>
<td>8</td>
<td>United States Veterans Hospital</td>
</tr>
</tbody>
</table>

The application developer also can use SQL `JOIN ... ON` notation, which is equivalent to a spatial predicate in the `WHERE` clause.

Contents of join_query2.sql

```
SELECT h.name, c.name
FROM test.hospitals AS h
JOIN test.counties AS c
ON ST_Intersects(h.location, c.shape) = 1 SELECTIVITY 0.0001
AND c.name = 'Dutchess'
```

Note that specifying the `JOIN` is equivalent to specifying the `INNER JOIN` clause. `OUTER JOIN` operations are not supported for spatial predicates.

If you have tables that do not contain a spatial column but that contain existing longitude and latitude columns, you can use these values to dynamically create point values in the query, although it might not be as efficient as having the point spatial values stored in the table.

In this query, the longitude and latitude columns in the hospitals table are used to dynamically create point values, which can be tested against the county polygon.

Contents of join_query3.sql

```
SELECT h.name, c.name
FROM test.hospitals AS h, test.counties AS c
WHERE ST_Intersects(ST_Point(longitude, latitude,1), c.shape) = 1 SELECTIVITY 0.0001
AND c.name = 'Dutchess'
```

Conversely, an application can determine which county a particular hospital is in by slightly modifying the query to identify the hospital and return the name of the county it is in as follows.
Contents of join_query4.sql

```sql
SELECT h.name, c.name
FROM test.hospitals AS h, test.counties AS c
WHERE ST_Intersection(h.location, c.shape) = 1 SELECTIVITY 0.0001
AND h.name = 'Vassar Brothers Hospital';
```

Query results for join_query4.sql

<table>
<thead>
<tr>
<th>Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>NAME</td>
</tr>
<tr>
<td>1</td>
<td>Vassar Brothers Hospital</td>
</tr>
</tbody>
</table>

If this type of query is required on a regular basis and is relatively static, it might be more efficient to add a column to the hospitals table and populate it with the corresponding county name. The following SQL statements alter the hospitals table to add a county name column and update each row with the county name.

Contents of update_hospitals.sql

```sql
ALTER TABLE test.hospitals ADD COLUMN county_name VARCHAR(32);

UPDATE test.hospitals AS h
SET h.county_name = (SELECT c.name FROM test.counties AS c
WHERE ST_Intersection(h.location, c.shape) = 1 SELECTIVITY 0.0001)
;

SELECT h.name, h.county_name
FROM test.hospitals AS h
WHERE h.name = 'Vassar Brothers Hospital';
```

Results of update_hospitals.sql

<table>
<thead>
<tr>
<th>Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>COUNTY_NAME</td>
</tr>
<tr>
<td>1</td>
<td>Vassar Brothers Hospital</td>
</tr>
</tbody>
</table>

Analysis with spatial joins

This section provides examples of queries that use spatial joins in conjunction with additional predicates and aggregation, which can address business problems. These examples continue to
use the hospitals and counties tables, but you should be able to use your imagination to consider how they could be applied to different tables of business data.

This example queries the hospitals within each county in New York state, qualifying by the state name in the counties table.

**Contents of join_query5.sql**

```
SELECT h.name AS hospital_name, c.name AS county_name
FROM test.hospitals AS h, test.counties AS c
WHERE ST_Intersects(h.location, c.shape) = 1 SELECTIVITY 0.0001
AND c.state_name = 'New York'
ORDER BY c.name, h.name
```

The result shown is the first nine of the 230 hospitals in New York.

**Results of join_query5.sql**

<table>
<thead>
<tr>
<th>Status</th>
<th>Hospital Name</th>
<th>County Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Albany Hospital</td>
<td>Albany</td>
</tr>
<tr>
<td>2</td>
<td>Albany Hospital For Incurables</td>
<td>Albany</td>
</tr>
<tr>
<td>3</td>
<td>Albany Hospital Saratoga</td>
<td>Albany</td>
</tr>
<tr>
<td>4</td>
<td>Cohoes Memorial Hospital</td>
<td>Albany</td>
</tr>
<tr>
<td>5</td>
<td>Saint Peters Hospital</td>
<td>Albany</td>
</tr>
<tr>
<td>6</td>
<td>Bronx State Hospital</td>
<td>Bronx</td>
</tr>
<tr>
<td>7</td>
<td>Daughters of Jacob Hospital</td>
<td>Bronx</td>
</tr>
<tr>
<td>8</td>
<td>Fordham Hospital</td>
<td>Bronx</td>
</tr>
<tr>
<td>9</td>
<td>Hebrew Hospital</td>
<td>Bronx</td>
</tr>
</tbody>
</table>

Total 230 records shown

You can also use the state information in the hospitals table and obtain the same results.

**Contents of join_query6.sql**

```
SELECT h.name AS hospital_name, c.name AS county_name
FROM test.hospitals AS h, test.counties AS c
WHERE ST_Intersects(h.location, c.shape) = 1 SELECTIVITY 0.0001
AND h.state = 'NY'
ORDER BY c.name, h.name
```

This example summarizes the number of hospitals in each county in New York.

**Contents of join_query7.sql**

```
SELECT c.name AS county_name, count(h.name) AS hospital_count
FROM test.hospitals AS h, test.counties AS c
WHERE ST_Intersects(h.location, c.shape) = 1 SELECTIVITY 0.0001
AND c.state_name = 'New York'
GROUP BY c.name
```
Results of join_query7.sql

<table>
<thead>
<tr>
<th>Status</th>
<th>COUNTY_NAME</th>
<th>HOSPITAL_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Albany</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Bronx</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>Broome</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Cayuga</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>Chautauqua</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Chenango</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>Chenango</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Columbia</td>
<td>3</td>
</tr>
</tbody>
</table>

Total 45 records shown.

If you are interested in knowing which counties don’t have any hospitals, use a `NOT EXISTS` expression, which checks for the hospitals in each county. The previous query can only identify the counties that have one or more hospitals because the query will not return any rows to count if a county does not contain any hospitals.

Contents of join_query8.sql

```sql
SELECT c.name AS county_name
FROM test.counties AS c
WHERE c.state_name = 'New York'
AND NOT EXISTS
    ( SELECT h.id
      FROM test.hospitals AS h
      WHERE ST_Intersects(h.location, c.shape) = 1
      ORDER BY c.name
    )
ORDER BY c.name
```

The result shows that 32 New York counties have no hospitals.

Results of join_query8.sql

To identify counties where the population is underserved by hospitals, an interesting metric might be the number of people per hospital in each county. Using the population for each county in the year 2000, you can calculate this number.

Contents of join_query9.sql

```sql
SELECT c.name AS county_name, count(h.name) AS hospital_count, c.pop2000 AS "2000 population", c.pop2000 / count(h.name) AS people_per_hospital
FROM test.hospitals AS h, test.counties AS c
WHERE ST_Intersects(h.location, c.shape) = 1
AND c.state_name = 'New York'
GROUP BY c.name, c.pop2000
ORDER BY people_per_hospital DESC
```
Results of join_query9.sql

With additional details about the number of beds or the number of doctors in each hospital, you can determine a better measure of health care capability.

**Window queries**

A common use case for mapping applications and in particular for web mapping applications is to select objects that fall within a rectangular region. This can be done by creating a polygon to represent the rectangle and the `ST_Intersects` spatial predicate. However, there is another spatial predicate, `EnvelopesIntersect`, which can be used to select objects whose envelope intersect a rectangular region. This spatial predicate is simpler to use and is more efficient in most cases.

One factor to consider is that since this predicate only compares the envelope of a geometry, it might select line and polygon geometries that are outside the specified window. This is generally not a problem with mapping applications, which will discard geometries outside the display window. This predicate is accurate for point data.

`EnvelopesIntersect` takes as parameters the name of a spatial column, four `DOUBLE` values representing the lower-left and upper-right corners of a rectangle, and an `INTEGER` value of the spatial reference system. The following query shows an example that uses `ST_Intersects`.

**Contents of join_query9.sql**

```sql
SELECT name
FROM test.hospitals
WHERE ST_Intersects(location, ST_Polygon('POLYGON ((-74.0 42.0, -73.0 42.0, -73.0 43.0, -74.0 43.0, -74.0 42.0))', 1)) = 1 SELECTIVITY 0.0001;
```

This query can be rewritten as follows.
Contents of window_query1.sql

```
SELECT name
FROM test.hospitals
WHERE
  EnvelopesIntersect(location, -74.0, 42.0, -73.0, 43.0, 1) = 1
SELECTIVITY 0.0001
```

When building web-mapping applications with widely used web-mapping APIs from providers such as Google Maps, Yahoo! Maps, Bing Maps, and others, it is necessary to provide the longitude and latitude values to be used in placing custom markers on the map. Obtain this information with a query like the following.

Contents of window_query2.sql

```
SELECT name, location..ST_X AS longitude, location..ST_Y AS latitude
FROM test.hospitals
WHERE
  EnvelopesIntersect(location, -74.0, 42.0, -73.0, 43.0, 1) = 1
SELECTIVITY 0.0001
```

Results of window_query2.sql

<table>
<thead>
<tr>
<th>Status</th>
<th>Result</th>
<th>LONGITUDE</th>
<th>LATITUDE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Marshall Hospital</td>
<td>-73.678177</td>
<td>42.71897100000001</td>
</tr>
<tr>
<td>2</td>
<td>Southwestern Medical Center</td>
<td>-73.206772</td>
<td>42.87424809999999</td>
</tr>
<tr>
<td>3</td>
<td>Hillcrest Hospital</td>
<td>-73.280113</td>
<td>42.457863</td>
</tr>
<tr>
<td>4</td>
<td>Saint Lukes Hospital</td>
<td>-73.245377</td>
<td>42.447304</td>
</tr>
<tr>
<td>5</td>
<td>Adventist Home</td>
<td>-73.7854</td>
<td>42.13052599999999</td>
</tr>
<tr>
<td>6</td>
<td>Albany Hospital</td>
<td>-73.776505</td>
<td>42.653141000000005</td>
</tr>
<tr>
<td>7</td>
<td>Albany Hospital For Incurables</td>
<td>-73.768173</td>
<td>42.63366800000001</td>
</tr>
<tr>
<td>8</td>
<td>Albany Hospital Sanatorium</td>
<td>-73.817345</td>
<td>42.68175099999999</td>
</tr>
<tr>
<td>9</td>
<td>Bellevue Hospital</td>
<td>-73.397077</td>
<td>42.79201439999999</td>
</tr>
</tbody>
</table>

There are 21 records shown.

Distance queries

Another common type of spatial query is to find things within a specified distance of a particular location. You have probably used web mapping applications to get this kind of information. With DB2 Spatial Extender, you can issue SQL queries from your application for queries like:

- Finding customers within 10 miles of a store
- Finding ATMs within 500 meters of the current location
- Finding competitive stores within 10 kilometers of a proposed store location

There are a number of ways to do this with different performance characteristics.

The primary spatial function is `ST_Distance`, which computes the distance between spatial values and which support an optional `units` parameter. The default units returned by `ST_Distance` are the units of the coordinate system. This means that if you are using the `NAD83` or `WGS84` coordinate...
systems, which use degrees of longitude and latitude, the distances are returned in degrees which are not the most user-friendly. The example use case is to find the hospitals within 30 miles of the specified location (Woodstock, NY).

**Don't do this: Contents of distance_query1.sql**

```sql
SELECT name
FROM test.hospitals
WHERE ST_Distance(location, ST_Point(-74.237, 42.037, 1)) < 30.0
ORDER BY name;
```

This query has a surprising result, with 500 rows returned when there are only eight hospitals within 30 miles. The query is actually finding all the hospitals within a distance of 30 degrees, which are the coordinate units of the data; 30 degrees is equivalent to more than 1,000 miles.

**Query results for distance_query1.sql**

<table>
<thead>
<tr>
<th>Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>46th Street Mental Health Center</td>
</tr>
<tr>
<td>2</td>
<td>A B Adams Convalescent Center</td>
</tr>
<tr>
<td>3</td>
<td>A D Williams Memorial Clinic</td>
</tr>
<tr>
<td>4</td>
<td>A G Hillen State Hospital</td>
</tr>
<tr>
<td>5</td>
<td>A M I Kendall Regional Medical Center</td>
</tr>
<tr>
<td>6</td>
<td>Abbeville County Memorial Hospital</td>
</tr>
<tr>
<td>7</td>
<td>Abbeville General Hospital</td>
</tr>
<tr>
<td>8</td>
<td>Abbott Hospital</td>
</tr>
<tr>
<td>9</td>
<td>Aberdeen Hospital (historical)</td>
</tr>
<tr>
<td>10</td>
<td>Aberdeen-Monroe County Hospital</td>
</tr>
</tbody>
</table>

Total 500 records shown

**Correct result but slow**

Changing the query to specify a unit parameter of **STATUTE MILE** returns the correct results, but takes a surprisingly long time — more than 1 second to return just three rows. The reason for this is that the spatial index can not be exploited if the units parameter is specified on the `ST_Distance` predicate. If the spatial index is not exploited, the distance is calculated from the specified point to every location in the table, which is computationally intensive.

**Contents of distance_query2.sql**

```sql
SELECT name
FROM test.hospitals
WHERE ST_Distance(location, ST_Point(-74.237, 42.037, 1), 'STATUTE MILE') < 30.0
ORDER BY name;
```
Query results for distance_query2.sql

<table>
<thead>
<tr>
<th>Status</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Adventist Home</td>
</tr>
<tr>
<td>2</td>
<td>Bowman Hospital</td>
</tr>
<tr>
<td>3</td>
<td>Columbia Memorial Hospital</td>
</tr>
<tr>
<td>4</td>
<td>Firemen Home</td>
</tr>
<tr>
<td>5</td>
<td>Greene County Memorial Hospital</td>
</tr>
<tr>
<td>6</td>
<td>Hudson River State Hospital</td>
</tr>
<tr>
<td>7</td>
<td>Saint Francis Hospital</td>
</tr>
<tr>
<td>8</td>
<td>Vassar Brothers Hospital</td>
</tr>
</tbody>
</table>

Correct result but fast

The spatial index can be exploited by including an additional predicate on the query, which approximates the linear distance in degrees. We calculate this by dividing the desired distance in miles by the number of miles in a degree. Unfortunately, the number of miles in a degree decreases as one increases the latitude. At the equator, a degree of longitude is about 69.2 miles, while at 50 degrees latitude, a degree of longitude is about 44.6 miles.

The length of a degree of longitude at a particular latitude can be computed with a query like

VALUES (st_distance(st_point(0.0, 50.0, 1), st_point(1.0, 50.0, 1), 'STATUTE MILE'));

where 50.0 is the latitude.

Fifty degrees is the northern boundary of the continental United States, which makes 44.6 a useful factor for queries against this region. For other regions, you need to consider the northern-most or southern-most (below the equator) latitude boundary of your data.

You can influence the DB2 query optimizer to exploit the spatial index by including the SELECTIVITY clause, resulting in the following query.

Contents of distance_query3.sql

```
SELECT name
FROM test.hospitals
WHERE  
  ST_Distance(location, ST_Point(-74.237, 42.037, 1), 'STATUTE MILE') < 30.0
AND
  DB2GSE.ST_Distance(location, DB2GSE.ST_Point(-74.237, 42.037, 1)) < (30.0 / 44.6)
  SELECTIVITY 0.0001
ORDER BY name
```

This query provides the correct results in a fraction of a second.

A simpler approach

In the previous query, the constructor for the point using longitude and latitude was duplicated in the query. It is preferable to create the point only once both in terms of performance and query complexity. A common table expression can be used to construct the point once and reference it in both predicates where it is needed.
Contents of distance_query4.sql

WITH current_location(point) AS (  
  VALUES(ST_Point(-74.237, 42.037, 1))  
)  
SELECT name  
FROM test.hospitals, current_location  
WHERE  
  ST_Distance(location, point, 'STATUTE MILE') < 30.0  
AND  
  ST_Distance(location, point) < 30.0 / 44.6  
SELECTIVITY 0.0001  
ORDER BY name  
;

An even-simpler approach

If you use DB2 9.7 fixpack 5 or later, or DB2 10.1 and are working with point data, you can use a different approach. Finding objects within a specified distance is equivalent to finding the objects whose location intersects a circular buffer of the specified distance around the desired point. The \texttt{ST_Buffer} function takes as parameters a spatial value and a distance to buffer around the spatial value. An optional third parameter specifies a unit value. In the following query an accurate buffer in degrees is constructed around the specified point which corresponds to a linear distance of 30 statute miles.

Contents of distance_query5.sql

SELECT name  
FROM test.hospitals  
WHERE  
  ST_Intersects(location,  
    ST_Buffer(ST_Point(-74.237, 42.037, 1), 30.0, 'STATUTE MILE')) = 1  
SELECTIVITY 0.0001  
ORDER BY NAME  
;

Returning the distances to each object

Continuing the previous example using \texttt{ST_Distance}, the distance from the specified point to each object within 30 miles can be found with this query. Since the application also needs the actual distance returned as well as testing within the distance specified, we use another common table expression, which does the approximate distance selection in degrees, selecting the name and distance. The final \texttt{SELECT} qualifies by the distance in miles. The \texttt{DECIMAL} function is applied to the distance to make the output more readable. The result set is then ordered by the distance value.
Contents of distance_query6.sql

WITH
current_location(point) AS (
  VALUES(ST_Point(-74.237, 42.037, 1))
),
work_tab(name, distance) AS (
  SELECT  name
  ,ST_Distance(location, point , 'STATUTE MILE') AS distance
  FROM test.hospitals, current_location
  WHERE ST_Distance(location, point) < (30.0 / 44.6)
  SELECTIVITY 0.0001
)
SELECT name, DECIMAL(distance, 8, 2) AS distance
FROM work_tab
WHERE distance < 30.0
ORDER BY distance
;

Query results for distance_query6.sql

<table>
<thead>
<tr>
<th>NAME</th>
<th>DISTANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greene Country Memorial Hospital</td>
<td>22.76</td>
</tr>
<tr>
<td>Adirondack Home</td>
<td>24.27</td>
</tr>
<tr>
<td>Hudson River State Hospital</td>
<td>25.02</td>
</tr>
<tr>
<td>Saint Francis Hospital</td>
<td>27.18</td>
</tr>
<tr>
<td>Bowman Hospital</td>
<td>27.51</td>
</tr>
<tr>
<td>Columbia Memorial Hospital</td>
<td>27.60</td>
</tr>
<tr>
<td>Firemen's Home</td>
<td>28.01</td>
</tr>
<tr>
<td>Vassar Brothers Hospital</td>
<td>28.26</td>
</tr>
</tbody>
</table>

Similarly, continuing on the previous example using `ST_Buffer`, the distance from the specified point to each object within 30 miles can be found with the following query.

Calculate distance from specified point to each object within 30 miles

WITH

current_location(point) AS (
  VALUES(ST_Point(-74.237, 42.037, 1))
)
SELECT
  name
,DECIMAL(ST_Distance(location, point , 'STATUTE MILE'), 8, 2) AS distance
FROM test.hospitals, current_location
WHERE
  ST_Intersects(location,
    ST_Buffer(point, 30.0, 'STATUTE MILE')) = 1
  SELECTIVITY 0.0001
ORDER BY distance
;

Since the examples using `ST_Buffer` are simpler, you might wonder why the more complex examples using `ST_Distance` were shown previously. There are two main reasons:

1. If you want to find the distance from a spatial type other than a point, for example a line or a polygon, the `ST_Buffer` function can not be used in this case.
2. Depending on your specific application, one approach might provide better performance than the other. The computation of the buffer is computationally intensive. If this is being done
only a few times and the table being queried is fairly large, this computation time might not be significant. However, if the application needs to perform this query for a large number of points, the time to construct the buffer for each point might become a significant portion of the processing time.

**Nearest and distance ranking queries**

Although an application is often interested in all the objects within a specified location, other common use cases related to distance are:

- Return the nearest object
- Return the nearest $n$ objects
- Return the nearest $n$ objects with rank order

**Returning the nearest $n$ objects**

A common use case is to find the nearest or nearest $n$ objects to a specified location. The number of rows returned can be limited by adding the `FETCH FIRST` expression to the SQL query. In the example below, the first five rows are being returned. If you only wanted the first row, you could specify `FETCH FIRST ROW ONLY` instead.

**Contents of nearest_query1.sql**

```sql
WITH current_location(point) AS ( 
    VALUES(ST_Point(-74.237, 42.037, 1))
) SELECT name, DECIMAL(ST_Distance(location, point, 'STATUTE MILE'), 8, 2) AS distance FROM test.hospitals, current_location WHERE ST_Intersects(location, ST_Buffer(point, 30.0, 'STATUTE MILE')) = 1 SELECTIVITY 0.0001 ORDER BY distance FETCH FIRST 5 ROWS ONLY;
```

**Query results for nearest_query1.sql**

<table>
<thead>
<tr>
<th>Status</th>
<th>Result</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Greene County Memorial Hospital</td>
<td>23.76</td>
</tr>
<tr>
<td>2</td>
<td>Adventist Home</td>
<td>24.27</td>
</tr>
<tr>
<td>3</td>
<td>Hudson River State Hospital</td>
<td>26.92</td>
</tr>
<tr>
<td>4</td>
<td>Saint Francis Hospital</td>
<td>27.18</td>
</tr>
<tr>
<td>5</td>
<td>Boone Hospital</td>
<td>27.51</td>
</tr>
</tbody>
</table>

Although the query works well in this case, there are two important considerations:
1. With the buffer distance specified as 30 miles, there might not be enough rows within this distance to satisfy the application need for a particular number of rows, if for example, the specified location is in a remote area.

2. Within the specified distance, there might be a large number of rows within this distance, for example, if the specified location is in a metropolitan area like New York City.

In the first case, it might be adequate to have a fixed radius to search. If there are fewer objects than the limit specified, this is an acceptable response.

An approach to ensure that the application will always get the number of rows desired is to iteratively reissue the query with larger distance values until the desired number of rows are returned. This presumes that the SQL query is being issued by application logic written in a language such as Java™, C, Perl, etc. Or it can be implemented as a DB2 stored procedure using the DB2 SQL PL language, which is shown in the next example.

**Returning the nearest $n$ objects with a stored procedure**

In this example, a stored procedure incorporates the logic required to find the nearest objects to a given location specified in latitude and longitude. The stored procedure starts with a search distance specified as an input parameter. If insufficient rows are returned, the distance value is doubled and the query reissued. This process is continued until the desired number of rows are returned. Although there is obviously additional cost in issuing the query multiple times, if you pick a reasonable starting distance, the desired results will come back on the first query. If they are not, the first query will have executed quickly to process a small number of rows and subsequent queries will re-use data cached in the DB2 buffer pool.

**Contents of nearest_sp1.sql**

```sql
-- #######################################################################
-- # Leaves cursor open to return nearest row(s)
-- # Input parameters are specified location in longitude, latitude
-- # and a starting target distance in miles for nearest search.
-- # Although not necessary, this stored procedure returns as values
-- # the final distance to satisfy the nearest requirement and
-- # the number of rows within that distance.
-- #######################################################################

CREATE PROCEDURE TEST.HOSPITAL_NEAREST (IN P_LONGITUDE DOUBLE, IN P_LATITUDE DOUBLE, IN P_STARTDISTANCE DOUBLE, OUT FINALDISTANCE DOUBLE, OUT FOUNDCOUNT INT)
DYNAMIC RESULT SETS 1
P1: BEGIN

DECLARE V_DISTANCE DOUBLE;
DECLARE V_FOUNDCOUNT INT DEFAULT 0;
DECLARE V_ROWSNEEDED INT DEFAULT 5;

-- Query to return data rows to application
DECLARE RETURN_CURSOR CURSOR WITH RETURN FOR
WITH CURRENT_LOCATION(POINT) AS (
    VALUES(ST_POINT(P_LONGITUDE, P_LATITUDE, 1))
)
SELECT
    NAME,
    DECIMAL(ST_DISTANCE(LOCATION, POINT, 'STATUTE MILE'), 8, 2) AS DISTANCE
FROM TEST.HOSPITALS, CURRENT_LOCATION
```
WHERE
    ST_INTERSECTS(LOCATION,
    ST_BUFFER(POINT, V_DISTANCE, 'STATUTE MILE')) = 1
    SELECTIVITY 0.0001
ORDER BY DISTANCE
FETCH FIRST 5 ROWS ONLY
;
-- Query to check number of rows within current distance
DECLARE COUNT_CURSOR CURSOR FOR
    WITH CURRENT_LOCATION(POINT) AS (
        VALUES(ST_POINT(P_LONGITUDE, P_LATITUDE, 1))
    )
    SELECT COUNT(*)
FROM TEST.HOSPITALS, CURRENT_LOCATION
WHERE
    ST_INTERSECTS(LOCATION,
    ST_BUFFER(POINT, V_DISTANCE, 'STATUTE MILE')) = 1
    SELECTIVITY 0.0001;
SET V_DISTANCE = P_STARTDISTANCE;

FETCH_LOOP: LOOP
    OPEN COUNT_CURSOR;
    FETCH COUNT_CURSOR INTO V_FOUNDCOUNT;
    CLOSE COUNT_CURSOR WITH RELEASE;
    IF V_FOUNDCOUNT >= V_ROWSNEEDED THEN LEAVE FETCH_LOOP;
    END IF;
    SET V_DISTANCE = V_DISTANCE * 2;
    END LOOP FETCH_LOOP;

SET FINALDISTANCE = V_DISTANCE;
SET FOUNDCOUNT = V_FOUNDCOUNT;
-- Open cursor to fetch rows for client application
OPEN RETURN_CURSOR;
END P1

Testing in Data Studio

You can easily create the stored procedure in the Data studio Data Project Explorer by right-clicking on the Stored Procedures folder under the project name and selecting New. Copy and paste the above code and save the stored procedure. In order to use the stored procedure, you must first deploy the stored procedure to the database by right-clicking on the stored procedure name and selecting Deploy. You can then execute the stored procedure by right-clicking on the name and selecting Run, which will allow you to specify the input parameters before running the stored procedure.

Testing in a command-line environment

You can also deploy the stored procedure by executing the sample SQL with the command db2 -td@-vf nearest_sp1.sql. Since the stored procedure uses ; as the statement end character, the @ needs to be at the end of the file to mark the end of the stored procedure, and the -td@ option must be specified when executing the SQL.

You can then execute the stored procedure with db2 call test.hospital_nearest(-74.237, 42.037, 5.0, ?, ?).
Sample executions

In this sample run, the location is in a rural area (Woodstock, N.Y.) and starting with a starting search radius of 5 miles, the radius needs to be doubled three times to 40 miles in order to find at least five objects. It actually finds 14 objects within 40 miles.

Sample execution 1

<table>
<thead>
<tr>
<th>Status</th>
<th>Parameters</th>
<th>Result1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Type</td>
<td>Data type</td>
</tr>
<tr>
<td>P_LONGITUDE</td>
<td>INPUT</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>P_LATITUDE</td>
<td>INPUT</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>P_STARTDISTANCE</td>
<td>INPUT</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FINDCOUNT</td>
<td>OUTPUT</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>FOUNDCOUNT</td>
<td>OUTPUT</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

Sample execution 2

In this sample run, the location is in a major city (Manhattan) and starting at the same starting radius of 5 miles, 36 objects are found at this radius, satisfying the requirement to find the five nearest.

Sample execution 3

<table>
<thead>
<tr>
<th>Status</th>
<th>Parameters</th>
<th>Result1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td>DISTANCE</td>
</tr>
<tr>
<td>1</td>
<td>Greene County Memorial Hospital</td>
<td>22.76</td>
</tr>
<tr>
<td>2</td>
<td>Adventist Home</td>
<td>24.27</td>
</tr>
<tr>
<td>3</td>
<td>Hudson River State Hospital</td>
<td>26.92</td>
</tr>
<tr>
<td>4</td>
<td>Saint Francis Hospital</td>
<td>27.10</td>
</tr>
<tr>
<td>5</td>
<td>Brune Hospital</td>
<td>27.81</td>
</tr>
</tbody>
</table>

Sample execution 4

<table>
<thead>
<tr>
<th>Status</th>
<th>Parameters</th>
<th>Result1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td></td>
<td>DISTANCE</td>
</tr>
<tr>
<td>1</td>
<td>Bellevue Medical Center</td>
<td>0.02</td>
</tr>
<tr>
<td>2</td>
<td>Beth Israel Hospital</td>
<td>0.58</td>
</tr>
<tr>
<td>3</td>
<td>Saint Vincent's Hospital</td>
<td>1.27</td>
</tr>
<tr>
<td>4</td>
<td>Goldin Memorial Hospital</td>
<td>1.41</td>
</tr>
<tr>
<td>5</td>
<td>Polytech Hospital</td>
<td>1.72</td>
</tr>
</tbody>
</table>

Returning the nearest \( n \) objects with ranking

Another common use case is to not only find the nearest objects but to rank them numerically with an integer value starting with 1. This example finds the three nearest hospitals to the specified location. Other typical use cases would be to find the nearest fire stations for insurance purposes or to find the nearest stores.
## Contents of rank_query1.sql

```sql
WITH current_location(point) AS (  
  VALUES(ST_Point(-73.976, 40.740, 1))
)  
SELECT * from (  
  SELECT  
    name,  
    DECIMAL(ST_Distance(point, location, 'STATUTE MILE'), 8, 3)  
    , ROW_NUMBER() OVER (ORDER BY ST_Distance(point, location, 'STATUTE MILE') ASC)  
  FROM test.hospitals, current_location  
  WHERE  
    ST_Intersects(location,  
    ST_Buffer(point, 10.0, 'STATUTE MILE')) = 1  
) temp (name, miles, rownumber)  
WHERE rownumber <= 3;
```

This query is similar to previous distance queries, but instead of ordering by distance and select the first $n$, it uses the `ROW_NUMBER()` OVER expression to assign row numbers based on the distance ordering.

## Query results from rank_query1.sql

<table>
<thead>
<tr>
<th>Name</th>
<th>Miles</th>
<th>Rownumber</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellevue Medical Center</td>
<td>0.020</td>
<td>1</td>
</tr>
<tr>
<td>Beth Israel Hospital</td>
<td>0.590</td>
<td>2</td>
</tr>
<tr>
<td>Saint Vincent's Hospital</td>
<td>1.272</td>
<td>3</td>
</tr>
</tbody>
</table>

## Conclusion

This tutorial has shown examples of typical spatial use cases and how to implement them efficiently in DB2 SQL queries using spatial functions. Subsequent tutorials will cover tuning for the best performance, working in a data warehouse environment and using visualization tools.

## Acknowledgement

I thank Amyris Rada for her review and suggestions for improving this tutorial.
## Downloads

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample SQL scripts</td>
<td>sqlscripts.zip</td>
<td>9KB</td>
</tr>
<tr>
<td>Sample spatial data</td>
<td>sampledata.zip</td>
<td>1400KB</td>
</tr>
</tbody>
</table>
Resources

Learn

- Find out more about all the IBM Spatial Offerings.
- Check out all the DB2 Spatial Extender documentation in the DB2 Knowledge Center.
- Check out the Esri shapefile specification.
- Learn more about Information Management at the developerWorks Information Management zone. Find technical documentation, how-to articles, education, downloads, product information, and more.
- Stay current with developerWorks technical events and webcasts.
- Follow developerWorks on Twitter.

Get products and technologies

- Download DB2 Express-C, which now includes DB2 Spatial Extender. This is a fully functional no-charge version of DB2.
- Download sample spatial data in shapefile format for use with Spatial Extender is available.
- Download the Data Management Geobrowser for DB2, Informix, and Netezza 1.1. This is free for you to use with any version of DB2 or Informix with the spatial feature installed.
- Download IBM Data Studio. Data Studio provides developers and DBAs with basic capabilities for database management and development at no charge for DB2 and Informix.
- Build your next development project with IBM trial software, available for download directly from developerWorks.
- Now you can use DB2 for free. Download DB2 Express-C, a no-charge version of DB2 Express Edition for the community that offers the same core data features as DB2 Express Edition and provides a solid base to build and deploy applications.

Discuss

- Participate in the discussion forum for this content.
- Check out the developerWorks blogs and get involved in the developerWorks community.
About the author

David Adler

David Adler has been responsible for the development of spatial database technology in IBM for more than 20 years, and 10 years in DB2 Spatial Extender Development. Since retiring from IBM, he provides DB2 Spatial and GIS services.

© Copyright IBM Corporation 2012, 2015

Trademark
(www.ibm.com/developerworks/ibm/trademarks/)