

IBM[®] DB2[®] Spatial Extender



User's Guide and Reference

Version 7

IBM[®] DB2[®] Spatial Extender



User's Guide and Reference

Version 7

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 311.

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

Order publications through your IBM representative or the IBM branch office serving your locality or by calling 1-800-879-2755 in the United States or 1-800-IBM-4YOU in Canada.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1998, 2000. All rights reserved.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures vii

Tables ix

About this book xi

Who should read this book xi

Conventions xi

How to send your comments xi

Part 1. Using DB2 Spatial Extender 1

Chapter 1. About DB2 Spatial Extender . . . 3

The purpose of DB2 Spatial Extender 3

Data that represents geographic features 4

 How data represents geographic features. 4

 The nature of spatial data 5

 Where spatial data comes from 6

How to create and use a DB2 Spatial Extender
GIS 8

 Interfaces to DB2 Spatial Extender and
 associated functionality 8

 Tasks you perform to create and use a DB2
 Spatial Extender GIS 9

 Scenario: An insurance company updates
 its GIS 11

Chapter 2. Installing DB2 Spatial Extender 15

DB2 Spatial Extender configuration 15

System requirements 15

 Supported operating systems 16

 Required database software 16

 Disk space requirements 16

Installing DB2 Spatial Extender 17

 Before you begin 17

 Installing DB2 Spatial Extender on
 Windows NT systems 17

 Installing DB2 Spatial Extender on AIX
 systems 17

Verifying the installation 18

Post-installation considerations. 19

 Downloading ArcExplorer 19

 Running the DB2 instance update utility
 (db2iupdt) 19

What's next? 20

Chapter 3. Setting up resources 21

Inventory of resources 21

 Reference data 21

 Resources that enable a database for
 spatial operations 21

Enabling a database for spatial operations 22

Creating a spatial reference system 23

 About coordinate and spatial reference
 systems 23

 Creating a spatial reference system from
 the Control Center 26

Chapter 4. Defining spatial columns, registering them as layers, and enabling a geocoder to maintain them 31

About spatial data types 31

 Data types for single-unit features 32

 Data types for multi-unit features 33

 A data type for all features 33

Defining a spatial column for a table,
registering this column as a layer, and
enabling a geocoder to maintain it 34

Registering a view column as a layer 36

Chapter 5. Populating spatial columns . . . 37

Using geocoders 37

 About geocoding 37

 Running the geocoder in batch mode 39

Importing and exporting data 41

 About importing and exporting 41

 Importing data to a new or existing table 41

 Importing data to an existing table 43

 Exporting data to a shape file 45

Chapter 6. Creating spatial indexes 47

Using the Control Center to create a spatial
index 47

Determining grid cell sizes 48

Chapter 7. Retrieving and analyzing spatial information 49

Methods of performing spatial analysis 49

Building a spatial query 49

 Spatial functions and SQL 49

 Spatial predicates and SQL 50

Chapter 8. Writing applications for DB2

Spatial Extender	53
Using the sample program	53
The sample program steps	53

Part 2. Reference material 61

Chapter 9. Stored procedures 63

db2gse.gse_disable_autogc	66
db2gse.gse_disable_db	68
db2gse.gse_disable_sref	69
db2gse.gse_enable_autogc	70
db2gse.gse_enable_idx	73
db2gse.gse_enable_idx	74
db2gse.gse_enable_sref	76
db2gse.gse_export_shape	78
db2gse.gse_import_sde	80
db2gse.gse_import_shape	82
db2gse.gse_register_gc	84
db2gse.gse_register_layer	86
db2gse.gse_run_gc	92
db2gse.gse_unregister_gc	94
db2gse.gse_unregister_layer	95

Chapter 10. Messages 97

Chapter 11. Catalog views 105

DB2GSE.COORD_REF_SYS	105
DB2GSE.GEOMETRY_COLUMNS	106
DB2GSE.SPATIAL_GEOCODER	106
DB2GSE.SPATIAL_REF_SYS	107

Chapter 12. Spatial indexes 109

A sample program fragment	109
B tree indexes	110
Ways to create a spatial index	110
How a spatial index is generated	111
Guidelines on using a spatial index	115
Selecting the grid cell size	116
Selecting the number of levels	116

Chapter 13. Geometries and associated spatial functions. 119

About geometries	119
Properties of geometries and associated functions.	121
Class	122
X and Y coordinates	122
Z coordinates	122

Measures	122
Interior, boundary, and exterior	123
Simple or non-simple	123
Empty or not empty	123
Envelope.	123
Dimension	124
Spatial reference system identifier	124
Instantiable geometries and associated functions.	125
Points.	125
Linestrings	126
Polygons.	127
Multipoints	129
Multilinestrings	129
Multipolygons	130
Functions that show relationships and comparisons, generate geometries, and convert values' formats	131
Functions that show relationships or comparisons between geographic features.	132
Functions that generate new geometries from existing ones	144
Functions that convert the format of a geometry's values.	149

Chapter 14. Spatial functions for SQL queries 155

AsBinaryShape.	156
GeometryFromShape.	157
EnvelopesIntersect	158
Is3d	160
IsMeasured	161
LineFromShape	162
LocateAlong	164
LocateBetween.	166
M	168
MLine FromShape	169
MPointFromShape	171
MPolyFromShape.	172
PointFromShape	173
PolyFromShape	174
ShapeToSQL	176
ST_Area	178
ST_AsBinary	180
ST_AsText	181
ST_Boundary	182
ST_Buffer	184
ST_Centroid	186
ST_Contains	187
ST_ConvexHull	189

ST_CoordDim	191
ST_Crosses	193
ST_Difference	195
ST_Dimension	196
ST_Disjoint	198
ST_Distance	200
ST_Endpoint	201
ST_Envelope	202
ST_Equals	204
ST_ExteriorRing	205
ST_GeometryFromText	207
ST_GeomFromWKB	209
ST_GeometryN	211
ST_GeometryType	212
ST_InteriorRingN	214
ST_Intersection	219
ST_Intersects	221
ST_IsClosed	222
ST_IsEmpty	224
ST_IsRing	226
ST_IsSimple	227
ST_IsValid	228
ST_Length	230
ST_LineFromText	232
ST_LineFromWKB	233
ST_MLineFromText	235
ST_MLineFromWKB	236
ST_MPointFromText	238
ST_MPointFromWKB	239
ST_MPolyFromText	240
ST_MPolyFromWKB	241
ST_NumGeometries	242
ST_NumInteriorRing	243
ST_NumPoints	244
ST_OrderingEquals	245
ST_Overlaps	246
ST_Perimeter	248
ST_PointFromText	249
ST_PointFromWKB	250
ST_Point	251
ST_PointN	252
ST_PointOnSurface	253
ST_PolyFromText	254
ST_PolyFromWKB	255
ST_Polygon	257
ST_Relate	258
ST_SRID	260
ST_StartPoint	261
ST_SymmetricDiff	262
ST_Touches	264

ST_Transform	265
ST_Union	266
ST_Within	267
ST_WKBToSQL	268
ST_WKTToSQL	270
ST_X	271
ST_Y	272
Z	273

Chapter 15. Coordinate systems 275

Overview of coordinate systems	275
Supported linear units	277
Supported angular units	277
Supported spheroids	278
Supported geodetic datums	279
Supported prime meridians	281
Supported map projections	282
Conic projections	282
Azimuthal or planar projections	283
Map projection parameters	283

Chapter 16. File formats for spatial data 285

The OGC well-known text representations	285
The OGC well-known binary (WKB) representations	290
Numeric type definitions	291
XDR (Big Endian) encoding of numeric types	291
NDR (Little Endian) encoding of numeric types	291
Conversion between NDR and XDR	291
Description of WKBGeometry byte streams	292
Assertions for the WKB representation	293
The ESRI shape representations	294
Shape types in XY space	295
Measured shape types in XY space	299
Shape types in XYZ space	303

Part 3. Appendixes 309

Notices	311
Trademarks	314

Index 317

Contacting IBM	323
Product Information	323

Figures

1. Table row that represents a geographic feature; table row whose address data represents a geographic feature 4
2. Tables with spatial columns added 5
3. Tables that include spatial data derived from source data 7
4. Table that includes new spatial data derived from existing spatial data 7
5. Client-server setup 15
6. Hierarchy of spatial data types 32
7. Application of a 10.0e0 grid level 112
8. Effect of adding grid levels 30.0e0 and 60.0e0 114
9. Hierarchy of geometries supported by DB2 Spatial Extender. 120
10. Linestring objects 127
11. Polygons 128
12. Multilinestrings 130
13. Multipolygons 131
14. ST_Equals 134
15. ST_Disjoint 135
16. ST_Touches 137
17. ST_Overlaps. 138
18. Within. 141
19. ST_Contains. 142
20. Minimum distance between two cities 143
21. ST_Intersection. 145
22. ST_Difference 146
23. ST_Union. 146
24. ST_Buffer. 147
25. LocateAlong. 148
26. LocateBetween 149
27. ST_ConvexHull. 149
28. Using area to find a building footprint 179
29. A buffer with a five-mile radius is applied to a point 185
30. Using ST_Contains to ensure that all buildings are contained within their lots. 188
31. Using ST_Crosses to find the waterways that pass through a hazardous waste area. 194
32. Using ST_Disjoint to find the buildings that do not lie within (intersect) any hazardous waste area. 199
33. Using ST_ExteriorRing to determine the length of an island shore line. 206
34. Using ST_InteriorRingN to determine the length of the lakeshores within each island 214
35. Using ST_Intersection to determine how large an area in each of the buildings might be affected by hazardous waste 220
36. Using ST_Length to determine the total length of the waterways in a county. 231
37. Using ST_Overlaps to determine the buildings that are at least partially within of a hazardous waste area. 247
38. Using ST_SymmetricDiff to determine the hazardous waste areas that do not contain sensitive areas (inhabited buildings) 263
39. Representation in NDR format 293
40. A polygon with a hole and eight vertices 298
41. Contents of the polygon byte stream 298

Tables

1. Minimum software requirements	16	20. Input parameters for the db2gse.gse_import_sde stored procedure..	80
2. Disk space requirements	16	21. Output parameters for the db2gse.gse_import_sde stored procedure..	81
3. Spatial functions and operations	49	22. Input parameters for the db2gse.gse_import_shape stored procedure..	82
4. Rules for index exploitation	51	23. Output parameters for the db2gse.gse_import_shape stored procedure..	83
5. DB2 Spatial Extender sample program	54	24. Input parameters for the db2gse.gse_register_gc stored procedure.	84
6. Input parameters for the db2gse.gse_disable_autogc stored procedure..	66	25. Output parameters for the db2gse.gse_register_gc stored procedure.	85
7. Output parameters for the db2gse.gse_disable_autogc stored procedure..	67	26. Input parameters for the db2gse.gse_register_layer stored procedure..	86
8. Output parameters for the db2gse.gse_disable_db stored procedure.	68	27. Output parameters for the db2gse.gse_register_layer stored procedure..	91
9. Input parameter for the db2gse.gse_disable_sref stored procedure..	69	28. Input parameters for the db2gse.gse_run_gc stored procedure..	92
10. Output parameters for the db2gse.gse_disable_sref stored procedure..	69	29. Output parameters for the db2gse.gse_run_gc stored procedure..	93
11. Input parameters for the db2gse.gse_enable_autogc stored procedure..	70	30. Input parameter for the db2gse.gse_unregist_gc stored procedure..	94
12. Output parameters for the db2gse.gse_enable_autogc stored procedure..	72	31. Output parameters for the db2gse.gse_unregist_gc stored procedure..	94
13. Output parameters for the db2gse.gse_enable_db stored procedure..	73	32. Input parameters for the db2gse.gse_unregist_layer stored procedure..	95
14. Input parameters for the db2gse.gse_enable_idx stored procedure.	74	33. Output parameters for the db2gse.gse_unregist_layer stored procedure..	96
15. Output parameters for the db2gse.gse_enable_idx stored procedure.	75	34. Columns in the DB2GSE.COORD_REF_SYS catalog view	105
16. Input parameters for the db2gse.gse_enable_sref stored procedure..	76	35. Columns in the DB2GSE.GEOMETRY_COLUMNS catalog view.	106
17. Output parameters for the db2gse.gse_enable_sref stored procedure..	77		
18. Input parameters for the db2gse.gse_export_shape stored procedure..	78		
19. Output parameters for the db2gse.gse_export_shape stored procedure..	79		

36.	Columns in the DB2GSE.SPATIAL_GEOCODER catalog view	106	55.	Matrix for ST_Contains	143
37.	Columns in the DB2GSE.SPATIAL_REF_SYS catalog view	107	56.	Equals pattern matrix	258
38.	The 10.0e0 grid cell entries for the example geometries	112	57.	Supported linear units	277
39.	The intersections of the geometries in the three-tiered index.	114	58.	Supported angular units.	277
40.	Matrix for ST_Within.	133	59.	Supported spheroids	278
41.	Matrix for equality	134	60.	Supported geodetic datums	279
42.	Matrix for ST_Disjoint	136	61.	Supported prime meridians	281
43.	Matrix for ST_Intersects (1).	136	62.	Supported map projections	282
44.	Matrix for ST_Intersects (2).	136	63.	Conic projections	282
45.	Matrix for ST_Intersects (3).	136	64.	Map projection parameters	283
46.	Matrix for ST_Intersects (4).	137	65.	Geometry types and their text representations	288
47.	Matrix for ST_Touches (1)	137	66.	Point byte stream contents	295
48.	Matrix for ST_Touches (2)	138	67.	MultiPoint byte stream contents	295
49.	Matrix for ST_Touches (3)	138	68.	PolyLine byte stream contents	296
50.	Matrix for ST_Overlaps (1)	138	69.	Polygon byte stream contents	298
51.	Matrix for ST_Overlaps (2)	139	70.	PointM byte stream contents	299
52.	Matrix for ST_Crosses (1)	140	71.	MultiPointM byte stream contents	300
53.	Matrix for ST_Crosses (2)	140	72.	PolyLineM byte stream contents	301
54.	Matrix for ST_Within.	141	73.	PolygonM byte stream contents	302
			74.	PointZ byte stream contents	303
			75.	MultiPointZ byte stream contents	303
			76.	PolyLineZ byte stream contents	305
			77.	PolygonZ byte stream contents	306

About this book

This book is divided into two parts. The first part contains conceptual information about DB2 Spatial Extender and explains how to install, configure, administer, and program for DB2 Spatial Extender on Windows NT and AIX systems. The second part consists of reference information about stored procedures, geometries, functions, messages, and catalog views that you use with DB2 Spatial Extender.

Who should read this book

This book is for administrators setting up the spatial environment and for application programmers developing applications with spatial data.

Conventions

This book uses these highlighting conventions:

Boldface type

Indicates commands and graphical user interface (GUI) controls (for example, names of fields, names of folders, menu choices).

`Monospace type`

Indicates examples of coding or of text that you type.

Italic type

Indicates variables that you should replace with a value. Italic type also indicates book titles and emphasizes words.

UPPERCASE TYPE

Indicates SQL keywords and names of objects (for example, tables, views, and servers).

How to send your comments

Your feedback helps IBM to provide quality information. Please send any comments that you have about this book or other DB2 documentation. You can use any of the following methods to provide comments:

- Send your comments from the Web. You can access the IBM Data Management online readers' comment form at <http://www.ibm.com/software/data/rcf>
- Send your comments by e-mail to comments@vnet.ibm.com. Be sure to include the name of the product, the version number of the product, and the name and part number of the book (if applicable). If you are

commenting on specific text, please include the location of the text (for example, a chapter and section title, a table number, a page number, or a help topic title).

Part 1. Using DB2 Spatial Extender

Chapter 1. About DB2 Spatial Extender

This chapter introduces DB2 Spatial Extender by explaining its purpose, discussing the data that it processes, and illustrating how to use it. The chapter concludes with a quick guide to the rest of this book.

The purpose of DB2 Spatial Extender

You use DB2 Spatial Extender to create a *geographic information system* (GIS): a complex of objects, data, and applications that allows you to generate and analyze spatial information about geographic features. *Geographic features* include the objects that comprise the earth's surface and the objects that occupy it. They make up both the natural environment (examples are rivers, forests, hills, and deserts) and the cultural environment (cities, residences, office buildings, landmarks, and so on).

Spatial information includes facts such as:

- The location of geographic features with respect to their surroundings (for example, points within a city where hospitals and clinics are located, or the proximity of the city's residences to local earthquake zones)
- Ways in which geographic features are related to each other (for example, information that a certain river system is enclosed within a specific region, or that certain bridges in that region cross over the river system's tributaries)
- Measurements that apply to one or more geographic features (for example, the distance between an office building and its lot line, or the length of a bird preserve's perimeter)

Spatial information, either by itself or in combination with traditional relational database management system (RDBMS) output, can help you to design projects and make business and policy decisions. For example, suppose that the manager of a county welfare district needs to verify which welfare applicants and recipients actually live within the area that the district services. DB2 Spatial Extender can derive this information from the serviced area's location and from the addresses of the applicants and recipients.

Or suppose that the owner of a restaurant chain wants to do business in nearby cities. To determine where to open new restaurants, the owner needs answers to such questions as: Where in these cities are concentrations of clientele who typically frequent my restaurants? Where are the major highways? Where is the crime rate lowest? Where are the competition's restaurants located? DB2 Spatial Extender can produce spatial information in

visual displays to answer these questions, and the underlying RDBMS can generate labels and text to explain the displays.

Several other examples of the uses of DB2 Spatial Extender appear in this book, especially in “Chapter 7. Retrieving and analyzing spatial information” on page 49, “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53, and “Chapter 14. Spatial functions for SQL queries” on page 155.

Data that represents geographic features

This section provides an overview of the data that you generate, store, and manipulate to obtain spatial information. The topics covered are:

- How data represents geographic features
- The nature of spatial data
- Ways to produce spatial data

How data represents geographic features

In DB2 Spatial Extender, a geographic feature can be represented by a row in a table or view, or by a portion of such a row. For example, consider two of the geographic features mentioned in “The purpose of DB2 Spatial Extender” on page 3, office buildings and residences. In Figure 1, each row of the BRANCHES table represents a branch office of a bank. As a variation, each row of the CUSTOMERS table in Figure 1, taken as a whole, represents a customer of the bank. However, part of each row—specifically, the cells that contain a customer’s address—can be regarded as representing the customer’s residence.

BRANCHES

ID	NAME	ADDRESS	CITY	STATE	ZIP
937	Airzone-Multern	92467 Airzone Blvd	San Jose	CA	95141

CUSTOMERS

ID	LAST NAME	FIRST NAME	ADDRESS	CITY	STATE	ZIP	CHECKING	SAVINGS
59-6396	Kriner	Endela	9 Concourt Circle	San Jose	CA	95141	A	A

Figure 1. Table row that represents a geographic feature; table row whose address data represents a geographic feature. The row of data in the BRANCHES table represents a branch office of a bank. The cells for address data in the CUSTOMERS table represent the residence of a customer. The names and addresses in both tables are fictional.

The tables in Figure 1 contain data that identifies and describes the bank’s branches and customers. Such data is called *attribute data*.

A subset of the attribute data—the values that denote the branches’ and customers’ addresses—can be translated into values that yield spatial information. For example, as shown in Figure 1 on page 4, one branch office’s address is 92467 Airzone Blvd., San Jose CA 95141. A customer’s address is 9 Concourt Circle, San Jose CA 95141. DB2 Spatial Extender can translate these addresses into values that indicate where the branch and the customer’s home are situated with respect to their surroundings. Figure 2 shows the BRANCHES and CUSTOMERS tables with new columns that are designated to contain such values.

BRANCHES

ID	NAME	ADDRESS	CITY	STATE	ZIP	LOCATION
937	Airzone-Multern	92467 Airzone Blvd	San Jose	CA	95141	

CUSTOMERS

ID	LAST NAME	FIRST NAME	ADDRESS	CITY	STATE	ZIP	LOCATION	CHECKING	SAVINGS
59-6396	Kriner	Endela	9 Concourt Circle	San Jose	CA	95141		A	A

Figure 2. Tables with spatial columns added. In each table, the LOCATION column will contain coordinates that correspond to the addresses.

When addresses and similar identifiers are used as the starting point for spatial information, they are called *source data*. Because the values derived from them yield spatial information, these derived values are called *spatial data*. The next section describes spatial data and introduces its associated data types.

The nature of spatial data

Much spatial data is made up of coordinates. A *coordinate* is a number that denotes a position that is relative to a point of reference. For example, latitudes are coordinates that denote positions relative to the equator. Longitudes are coordinates that denote positions relative to the Greenwich meridian. Thus, the position of Yellowstone National Park is defined by its latitude (44.45 degrees north of the equator) and its longitude (110.40 degrees west of the Greenwich meridian).

Latitudes, longitudes, their points of reference, and other associated parameters are referred to collectively as a *coordinate system*. Coordinate systems based on values other than latitude and longitude also exist. These coordinate systems have their own measures of position, points of reference, and additional distinguishing parameters.

The simplest spatial data item consists of two coordinates that define the position of a single geographic feature. (A *data item* is the value or values that occupy a cell of a relational table.) A more extensive spatial data item consists of several coordinates that define a linear path such as a road or river might form. A third kind consists of coordinates that define the perimeter of an area; for example, the rim of a land parcel or flood plain. These and other kinds of spatial data items that DB2 Spatial Extender supports are described more fully in “Chapter 13. Geometries and associated spatial functions” on page 119.

Each spatial data item is an instance of a spatial data type. The data type for two coordinates that mark a location is `ST_Point`; the data type for coordinates that define linear paths is `ST_LineString`; and the data type for coordinates that define perimeters is `ST_Polygon`. These types, together with the other data types for spatial data, are structured types that belong to a single hierarchy. For an overview of the hierarchy, see “About spatial data types” on page 31.

Where spatial data comes from

You can obtain spatial data by:

- Deriving it from attribute data
- Deriving it from other spatial data
- Importing it

Using attribute data as source data

DB2 Spatial Extender can derive spatial data from attribute data, such as addresses (as mentioned in “How data represents geographic features” on page 4). This process is called *geocoding*. To see the sequence involved, consider Figure 2 on page 5 as a “before” picture and Figure 3 on page 7 as an “after” picture. Figure 2 on page 5 shows that the `BRANCHES` table and the `CUSTOMERS` table both have an empty column designated for spatial data. Suppose that DB2 Spatial Extender geocodes the addresses in these tables to obtain coordinates that correspond to the addresses, and places the coordinates into the columns. Figure 3 on page 7 illustrates this result.

BRANCHES

ID	NAME	ADDRESS	CITY	STATE	ZIP	LOCATION
937	Airzone-Multern	92467 Airzone Blvd	San Jose	CA	95141	1653 3094

CUSTOMERS

ID	LAST NAME	FIRST NAME	ADDRESS	CITY	STATE	ZIP	LOCATION	CHECKING	SAVINGS
59-6396	Kriner	Endela	9 Concourt Circle	San Jose	CA	95141	953 1527	A	A

Figure 3. Tables that include spatial data derived from source data. The LOCATION column in the CUSTOMERS table contains coordinates that a geocoder derived from the address in the ADDRESS, CITY, STATE, and ZIP columns. Similarly, the LOCATION column in the BRANCHES table contains coordinates that the geocoder derived from the address in this table's ADDRESS, CITY, STATE, and ZIP columns. This example is fictional; simulated coordinates, not actual ones, are shown.

DB2 Spatial Extender uses a function, called a *geocoder*, to translate attribute data into spatial data and to place this spatial data into table columns. For more information about geocoders, see “About geocoding” on page 37.

Using other spatial data as source data

Spatial data can be generated not only from attribute data, but also from other spatial data. For example, suppose that the bank whose branches are defined in the BRANCHES table wants to know how many customers are located within five miles of each branch. Before the bank can obtain this information from the database, it must supply the database with the definition of the zone that lies within a five-mile radius around each branch. A DB2 Spatial Extender function, ST_Buffer, can create such a definition. Using the coordinates of each branch as input, ST_Buffer can generate the coordinates that demarcate the perimeters of the desired zones. Figure 4 shows the BRANCHES table with information that is supplied by ST_Buffer.

BRANCHES

ID	NAME	ADDRESS	CITY	STATE	ZIP	LOCATION	SALES_AREA
937	Airzone-Multern	92467 Airzone Blvd	San Jose	CA	95141	1653 3094	1002 2001, 1192 3564, 2502 3415, 1915 3394, 1002 2001

Figure 4. Table that includes new spatial data derived from existing spatial data. The coordinates in the SALES_AREA column were derived by the ST_Buffer function from the coordinates in the LOCATION column. Like the coordinates in the LOCATION column, those in the SALES_AREA column are simulated; they are not actual.

In addition to ST_Buffer, DB2 Spatial Extender provides several other functions that derive new spatial data from existing spatial data. For

descriptions of ST_Buffer and these other functions, see “Functions that generate new geometries from existing ones” on page 144.

Importing spatial data

A third way to obtain spatial data is to import it from files that are in one of the formats that DB2 Spatial Extender supports. For descriptions of these formats, see “Chapter 16. File formats for spatial data” on page 285. These files contain data that is usually applied to maps: census tracks, flood plains, earthquake faults, and so on. By using such data in combination with spatial data that you produce, you can augment the geographic information available to you. For example, if a public works department needs to determine what hazards a residential community is vulnerable to, it could use ST_Buffer to define a zone around the community. The public works department could then import data on flood plains and earthquake faults to see which of these problem areas overlap the zone.

How to create and use a DB2 Spatial Extender GIS

You create a DB2 Spatial Extender GIS by setting up DB2 Spatial Extender and developing GIS projects within the combined environments of DB2 Spatial Extender and its underlying DB2 RDBMS. You use the GIS by implementing these projects; that is, by generating and analyzing the information—both spatial and traditional—that they are designed to provide. The entire effort involves performing several sets of tasks. This section introduces the interfaces with which you can perform these tasks, provides an overview of the tasks, and presents a scenario to illustrate them.

Interfaces to DB2 Spatial Extender and associated functionality

This section surveys the interfaces by means of which you can create a DB2 Spatial Extender GIS (that is, set up resources for it, obtain spatial data, and so on) and use it (that is, generate and analyze information about geographic features).

You can create a DB2 Spatial Extender GIS by:

- Using the DB2 Control Center’s DB2 Spatial Extender windows and menu choices. For instructions, see:
 - “Chapter 3. Setting up resources” on page 21
 - “Chapter 4. Defining spatial columns, registering them as layers, and enabling a geocoder to maintain them” on page 31
 - “Chapter 5. Populating spatial columns” on page 37
 - “Chapter 6. Creating spatial indexes” on page 47
- Running an application program that calls DB2 Spatial Extender stored procedures. For guidelines on developing such a program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

- Using both the Control Center and an application program. For example, you can use the Control Center to invoke the default geocoder. If, in addition, you want to use another geocoder, you must first register it to DB2 Spatial Extender by invoking the `db2gse.gse_register_gc` stored procedure in an application program. (For information about non-default geocoders, see “About geocoding” on page 37. For information about the `db2gse.gse_register_gc` stored procedure, see “`db2gse.gse_register_gc`” on page 84.)
- Using the Control Center, an application program, or both, in combination with other interfaces. For example, to create a table to hold data that is to be generated by a spatial function, such as a geocoder, you could use either the Command Line Processor or the Control Center interfaces.

You can use a DB2 Spatial Extender GIS by:

- Rendering information graphically with a geobrowser; for example, ArcExplorer, which is offered by the Environmental Systems Research Institute (ESRI)
- Submitting SQL queries explicitly from the DB2 Control Center or Command Line Processor
- Submitting SQL queries from an application program

Tasks you perform to create and use a DB2 Spatial Extender GIS

This section provides an overview of the tasks through which you create and use a DB2 Spatial Extender GIS. The tasks through which you create the GIS involve setting up DB2 Spatial Extender and developing GIS projects. The tasks through which you use the GIS involve implementing the projects. This overview begins with setting up DB2 Spatial Extender, and then moves on to developing and implementing a GIS project. The section concludes by indicating how the tasks described in the overview can vary in actual practice.

Setting up DB2 Spatial Extender

To set up DB2 Spatial Extender:

1. Plan and make preparations (decide what GIS projects to develop, decide what database to enable for DB2 Spatial Extender, select personnel to administer DB2 Spatial Extender and develop the projects, and so on).
2. Install DB2 Spatial Extender.
3. Put resources in place to support GIS projects; for example:

Resources supplied by DB2 Spatial Extender

These include a system catalog, spatial data types, spatial functions (including a default geocoder), and so on. The task of setting up these resources is referred to as *enabling the database for spatial operations*.

Geocoders developed by users, vendors, or both.

The default geocoder translates United States addresses into

spatial data. Your organization and others can provide geocoders that translate foreign addresses and other kinds of attribute data into spatial data.

For instructions on installing DB2 Spatial Extender, see “Chapter 2. Installing DB2 Spatial Extender” on page 15. For instructions on using the Control Center to put resources in place, see “Chapter 3. Setting up resources” on page 21. For guidelines on using an application program for this purpose, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53. For a scenario that illustrates the overall effort of setting up DB2 Spatial Extender, see “A system to integrate spatial and traditional data” on page 12.

Developing and implementing a GIS project

To develop and implement a GIS project:

1. Plan and make preparations (set goals for the project, decide what tables and data you need, determine what coordinate system or systems to use, and so on).
2. Decide what spatial reference system or systems to use. Coordinate values typically include positive integers, negative numbers, and decimal numbers. DB2 Spatial Extender, however, must store all coordinate values in the form of positive integers. A *spatial reference system* is a set of parameters that defines how negative and decimal numbers in a specific coordinate system are to be converted into positive integers, so that DB2 Spatial Extender can store them. After you decide what coordinate system to use for a spatial column, you need to specify the spatial reference system by which the necessary conversion can take place for that column. If an existing spatial reference system meets your requirements, you can use it; otherwise, you can create one.
3. Define one or more columns to contain spatial data, register them to DB2 Spatial Extender, and enable a geocoder to maintain them automatically. Registering a spatial column involves recording it in the DB2 Spatial Extender catalog. From the time that you register it, it is called a *layer*, because information generated from it will add a stratum, or layer, to the virtual geographic landscape that your GIS creates for you. After you register it, you can perform spatial operations on it; for example, you can populate it and define a spatial index on it.
4. Populate spatial columns:
 - For a project that requires a geocoder, set parameters for the geocoder. Then, run it so that, in a single operation, it geocodes all available source data and loads the resulting coordinates into a layer.
 - For a project that requires spatial data to be imported, import the data.
5. Facilitate access to spatial columns. Specifically, this involves defining indexes that enable DB2 to access spatial data quickly, and defining views that enable users to retrieve interrelated data efficiently. After you define such a view, you need to register its spatial columns as layers.

6. Generate and analyze spatial information and related business information. This involves querying spatial columns and related attribute columns. In such queries, you can include DB2 Spatial Extender functions that return a wide variety of information; for example, the minimum distance between two geographic features, or coordinates that define an area that surrounds a geographic feature. For information about the function that returns such coordinates, `ST_Buffer`, see “Using other spatial data as source data” on page 7 and “`ST_Buffer`” on page 184. For examples of queries that use spatial functions, see “Chapter 7. Retrieving and analyzing spatial information” on page 49 and “Chapter 14. Spatial functions for SQL queries” on page 155.

For instructions on using the Control Center to perform the tasks involved in developing a GIS project, see:

- “Chapter 3. Setting up resources” on page 21
- “Chapter 4. Defining spatial columns, registering them as layers, and enabling a geocoder to maintain them” on page 31
- “Chapter 5. Populating spatial columns” on page 37
- “Chapter 6. Creating spatial indexes” on page 47

For guidelines on using the Control Center to implement a GIS project, see “Chapter 7. Retrieving and analyzing spatial information” on page 49.

For guidelines on using an application program to develop and implement a GIS project, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

For a scenario that illustrates the overall effort, see “A project to establish offices and adjust premiums” on page 12.

How the sets of tasks can vary

The sets of tasks that you perform to create and use a DB2 Spatial Extender GIS can vary in content and sequence, depending on your requirements and on the interfaces that you use. For example, consider the tasks of defining columns to contain spatial data, registering them as layers, and enabling a geocoder to maintain them automatically. With the Control Center, you can perform these tasks together, from a single window. If you are invoking stored procedures from a program, however, you can perform these tasks separately, and you can time them at your discretion.

Scenario: An insurance company updates its GIS

This section presents a scenario to illustrate the sets of tasks that are described in the preceding section.

The Safe Harbor Real Estate Insurance Company’s information systems environment includes a DB2 Universal Database system and a separate GIS

database management system. To an extent, queries can retrieve combinations of data from the two systems. For example, a DB2 table stores information about revenue, and a GIS table stores the locations of the company's branch offices. Therefore, it is possible to find out the locations of offices that bring in revenues of specified amounts. But data from the two systems cannot be integrated (for example, users cannot join DB2 columns with GIS columns) and DB2 services such as query optimization are unavailable to the GIS. To overcome these disadvantages, Safe Harbor acquires DB2 Spatial Extender and establishes a new GIS development department. The following sections describe how the department sets up DB2 Spatial Extender and carries out its first project.

A system to integrate spatial and traditional data

To set up DB2 Spatial Extender, Safe Harbor's GIS development department proceeds as follows:

1. The department prepares to include DB2 Spatial Extender in its DB2 environment. For example:
 - a. The department's management team appoints a spatial administration team to install and implement DB2 Spatial Extender, and a spatial analysis team to generate and analyze spatial information.
 - b. Because Safe Harbor's business decisions are driven primarily by customers' requirements, the management team decides to install DB2 Spatial Extender in the database that contains information about its customers. Most of this information is stored in a table called CUSTOMERS.

As a convenient way to refer to the selected database, the members of the GIS development department call it a *GIS database*. They are aware, however, that it is not reserved for GIS projects only; non-spatial applications can continue to use it, as before.

2. The spatial administration team installs DB2 Spatial Extender.
3. The spatial administration team sets up resources that GIS projects will require:
 - The team uses the Control Center to supply the resources that enable the GIS database for spatial operations. These resources include the DB2 Spatial Extender catalog, spatial data types, spatial functions, and so on.
 - Because Safe Harbor is starting to extend its business into Canada, the spatial administration team begins soliciting Canadian vendors for geocoders that translate Canadian addresses into spatial data.

A project to establish offices and adjust premiums

To carry out its first GIS project under DB2 Spatial Extender, the GIS development department proceeds as follows:

1. The department prepares to develop the project; for example:
 - The management team sets goals for the project:

- To determine where to establish new branch offices.
 - To adjust premiums on the basis of customers' proximity to hazardous areas (areas with high rates of traffic accidents, areas with high rates of crime, flood zones, earthquake faults, and so on).
 - The GIS project will be concerned with customers and offices in the United States. Therefore, the spatial administration team decides to:
 - Use coordinate systems that accurately define locations in the parts of the United States in which Safe Harbor does business.
 - Use the default geocoder, because it is designed to geocode United States addresses.
 - The spatial administration team decides what data is needed to meet the project's goals and what tables will contain this data.
2. Using the Control Center, the spatial administration team creates two spatial reference systems. One determines how coordinates that define offices' locations are to be converted to data items that DB2 Spatial Extender can store. The other determines how coordinates that define locations of customers' residences are to be converted to data items that DB2 Spatial Extender can store.
 3. Using the Control Center, the spatial administration team defines columns to contain spatial data, registers them as layers, and enables a geocoder to maintain them automatically:
 - The team adds a LOCATION column to the CUSTOMERS table. The table already contains customers' addresses. The default geocoder will translate them into spatial data and load this data into the LOCATION column.
 - The team creates an OFFICES table to contain the data that is now stored in the separate GIS. This data includes the addresses of Safe Harbor's branch offices, spatial data that was derived from these addresses by a geocoder, and spatial data that defines a zone within a five-mile radius around each office. The data generated by the geocoder will go into a LOCATION column. The data that defines the zones will go into a SALES_AREA column.
 - The team registers the two LOCATION columns and the SALES_AREA columns as layers.
 - The team enables the default geocoder to automatically maintain the two LOCATION columns.
 4. The spatial administration team populates the CUSTOMER table's LOCATION column, the entire OFFICES table, and a new HAZARD_ZONES table:
 - The team uses the Control Center to populate the CUSTOMER table's LOCATION column:

- a. The team instructs the geocoder to insert spatial data for an address into the LOCATION column only under the following condition: a match between the address and its counterpart in the United States Census Bureau's records must be 100 percent accurate. (A file of addresses that are supplied by the Census Bureau is shipped with DB2 Spatial Extender. Before the geocoder can translate an address in the source data into spatial data, the geocoder must try to match this address with a counterpart in the file. Users specify what percentage of the match must be accurate in order for the spatial data to be placed in a table. This percentage is called a *precision*.)
 - b. The team runs the geocoder in batch mode, so that it can geocode all the addresses in the table in one operation. To the team's dismay, the geocoder rejects about one out of every ten addresses!
 - c. The team surmises that the rejects must be new addresses that have no exact matches in the Census Bureau's records. To resolve the problem, the team reduces the precision to 85.
 - d. The team runs the geocoder in batch mode again. The rate at which addresses are rejected falls to an acceptable level.
- Using a utility that is provided by the separate GIS, the team loads the office data into a file. Then the team uses the Control Center to import this data from the file to the new OFFICES table.
 - Using the Control Center, the team creates a HAZARD ZONES table, registers its spatial columns as layers, and imports data to it. The data comes from a file supplied by a map vendor.
5. Using the Control Center, the spatial administration team facilitates access to the new layers:
 - The team creates indexes for them.
 - The team creates a view that joins columns from the CUSTOMERS and HAZARD ZONES tables. The team then registers the views' spatial columns as layers.
 6. The spatial analysis team runs queries to obtain information that will help it meet the original objectives: to determine where to establish new branch offices, and to adjust premiums on the basis of customers' proximity to hazard areas.

Chapter 2. Installing DB2 Spatial Extender

This chapter provides instructions for installing DB2 Spatial Extender. The following topics are discussed:

- “DB2 Spatial Extender configuration”
- “System requirements”
- “Installing DB2 Spatial Extender” on page 17
- “Verifying the installation” on page 18
- “Post-installation considerations” on page 19
- “What’s next?” on page 20

DB2 Spatial Extender configuration

A DB2 Spatial Extender system consists of DB2 Universal Database, DB2 Spatial Extender, and a geobrowser (for example, ArcExplorer). Typically, a database that is enabled for spatial operations is located on the server. You use client applications to access spatial data through the DB2 Spatial Extender stored procedures and spatial queries. In addition, you can view spatial data with a geobrowser.

Figure 5 illustrates the architecture of DB2 Spatial Extender.

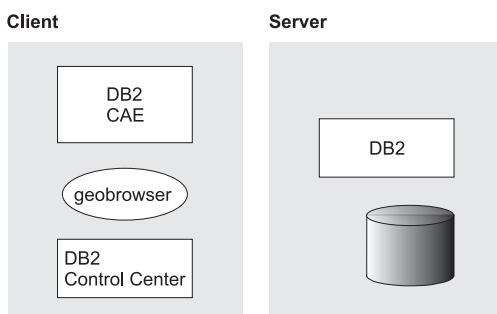


Figure 5. Client-server setup

System requirements

This section explains the software and hardware requirements for DB2 Spatial Extender.

Supported operating systems

DB2 Spatial Extender can be installed on the following operating systems:

- AIX 4.2 or later
- Windows NT 4.0 or later with Service Pack 5

Required database software

Before you install DB2 Spatial Extender, you must have DB2 software installed and configured on your system. Table 1 lists the database software requirements for both the DB2 Spatial Extender client component and the DB2 Spatial Extender server component.

Table 1. Minimum software requirements

Component	Software
Client	DB2 Administration Client, Version 7.1 ¹
Server	Either: <ul style="list-style-type: none">• DB2 Universal Database Enterprise Edition, Version 7.1• DB2 Universal Database Enterprise – Extended Edition, Version 7.1²

Notes:

1. If you are *not* planning to use the DB2 Control Center, a geobrowser to access spatial data, or the DB2 Spatial Extender sample program, you can use a down-level version of DB2 Administration Client.
2. Although you can use DB2 Spatial Extender with the DB2 Universal Database Enterprise - Extended Edition, the spatial index cannot be partitioned across multiple nodes as in the massive parallel processing (MPP) environment.

Disk space requirements

Table 2 lists the recommended disk space requirements for DB2 Spatial Extender.

Table 2. Disk space requirements

DB2 Spatial Extender component	Disk space
DB2 Spatial Extender server library (includes DB2 Spatial Extender server library, geocoder reference data, and documentation)	600 MB
DB2 Spatial Extender client support (includes the sample program data)	15 MB

Installing DB2 Spatial Extender

This section provides the information you need to install DB2 Spatial Extender on Windows NT and AIX operating systems.

Before you begin

If you did not already do so, install DB2 Administration Client (administration tools including the Control Center and the run-time client) on the client workstation, and install DB2 Universal Database Enterprise Edition or DB2 Universal Database Enterprise - Extended Edition. Instructions for doing so are in the appropriate *Quick Beginnings* book.

Installing DB2 Spatial Extender on Windows NT systems

To install DB2 Spatial Extender on a Windows NT system:

1. Log on to the system with a user name that has the required administration permissions.
2. Shut down any other programs.
3. Insert the CD-ROM into the drive. The installation launchpad opens.
4. Optional: Click **Release Notes** to check the DB2 Spatial Extender Release Notes for any changes to the installation process, then return to the DB2 Spatial Extender launchpad.
5. Click **Install**.
6. Respond to the setup program's prompts. Online help is available to guide you through the remaining steps. To invoke online help, click **Help** or press the F1 key.

When the installation is complete, DB2 Spatial Extender is installed under the directory %DB2PATH% (for example, c:\sqlib).

Installing DB2 Spatial Extender on AIX systems

To install DB2 Spatial Extender on an AIX system:

1. Log in as root.
2. Insert the CD-ROM into the drive.
3. Mount the CD-ROM on your AIX system. For information on mounting a CD-ROM refer to the *IBM DB2 Universal Database for UNIX Quick Beginnings*.
4. Change to the directory where the CD-ROM is mounted by entering the following command:

```
cd /cdrom
```

where *cdrom* is the mount point of the CD-ROM drive on AIX.

5. Enter the **db2setup** command to start the DB2 installer program. The Install DB2 Spatial Extender window opens.

Note: The DB2 installer program will start slowly because it scans your system for information.

6. From the product list of the Install DB2 Spatial Extender window, select the products that you want to install and click **OK**.

For more information or assistance when installing DB2 Spatial Extender, click **Help**.

When the installation is complete, DB2 Spatial Extender is installed in the /usr/lpp/db2_07_01 directory.

Verifying the installation

After you install DB2 Spatial Extender, you can verify its installation by using the DB2 Spatial Extender sample program. Before you can run the sample program, you must create a SAMPLE database and make the sample program executable.

Note: Be sure to use the compiler specified in the DB2 Spatial Extender makefile.

To compile and run the sample program for Windows NT:

1. Log in with the user ID that has administrator privileges.
2. From a command line prompt, enter **db2sampl** to create the DB2 SAMPLE database.
3. From a command line prompt, enter the following command:
`cd %DB2PATH%\samples\spatial`

Note: In order to perform step 3 and continue verifying the installation, you must own the default DB2 instance (DB2-DB2).

4. Enter **make rungsedemo**.
5. Enter **rungsedemo.exe**.
6. Check the error and completion messages that are displayed as the program is running.

To compile and run the sample program for AIX:

1. Log in as root.
2. Create or update a DB2 instance.
3. From a command line prompt, enter **db2sampl** to create the DB2 SAMPLE database.
4. From a command line prompt, enter the following command:
`cd $DB2INSTANCE/sql1lib/samples/spatial`

Note: In order to perform step 4 and continue verifying the installation, you must own the DB2 instance that you created or updated.

5. Enter **make rungsedemo**.
6. Enter **rungsedemo**.
7. Check the error and completion messages that appear as the program is running.

For information about the sample programs, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Post-installation considerations

After you successfully install DB2 Spatial Extender, you need to consider:

- Downloading ArcExplorer
- Running the DB2 instance update utility

Downloading ArcExplorer

IBM distributes ArcExplorer Java 3.0 as a sample program or you can obtain it from the ESRI Web site at <http://www.esri.com>.

For more information on installing and using ArcExplorer, refer to the *Using ArcExplorer* book, which is also available on the ESRI Web site.

ArcExplorer requires Java[®] 2 Runtime Environment (Standard Edition or Enterprise Edition), V1.2.2 available at no charge from the Sun Web site at <http://java.sun.com>.

Important: DB2 Universal Database V7.1 is shipped with IBM JDK 1.1.8. When you install JRE 1.2.2 for ArcExplorer, place it in a separate directory from DB2. Remember to set the CLASSPATH environment variable appropriately.

Running the DB2 instance update utility (db2iupdt)

The db2iupdt utility updates a specified DB2 instance to:

- Enable the instance to acquire a new system configuration.
- Enable the instance to access a function associated with installing or removing certain product options.

On AIX, this utility is located in /usr/lpp/db2_07_01. If you need help, type db2iupdt -h on the command line to open a help menu. On the Windows NT operating system, db2iupdt is located in the \sqllib\bin directory. Change to that directory to type the command. For a complete description of this command, refer to the *IBM DB2 Universal Database Command Reference*.

What's next?

After DB2 Spatial Extender is installed, you can use the DB2 Control Center to set-up the GIS environment and start to work with spatial information.

To invoke DB2 Spatial Extender from the Control Center:

1. From the Control Center window, expand the object tree until you find the **Databases** folder under the server where you want DB2 Spatial Extender to run.
2. Click the **Databases** folder. The databases are displayed in the contents pane on the right side of the window.
3. Right-click the database that you want to work with, and click the spatial operation in the pop-up menu that you want to perform.

For more information about using DB2 Spatial Extender from the Control Center, refer to:

- “Chapter 3. Setting up resources” on page 21
- “Chapter 4. Defining spatial columns, registering them as layers, and enabling a geocoder to maintain them” on page 31
- “Chapter 5. Populating spatial columns” on page 37
- “Chapter 6. Creating spatial indexes” on page 47

Chapter 3. Setting up resources

After you install DB2 Spatial Extender, you are ready to supply your database with resources that you need when you create spatial columns and manipulate spatial data. This chapter summarizes these resources and describes two of the tasks through which you make them available: enabling your database for spatial operations and creating spatial reference systems.

Inventory of resources

The resources that you draw on to create spatial columns and manipulate spatial data include:

- *Reference data*: addresses that DB2 Spatial Extender checks to verify addresses that you want to geocode
- Resources that enable a database for spatial operations: stored procedures, spatial functions, and others
- Non-default geocoders that are provided by users and vendors
- Spatial reference systems

This section discusses reference data and resources that enable a database for spatial operations. For information about non-default geocoders, see “About geocoding” on page 37. For information about spatial reference systems, see “About coordinate and spatial reference systems” on page 23.

Reference data

Reference data consists of the most recent addresses in the United States that the United States Census Bureau has collected. Before the default geocoder can translate an address in your database into coordinates, it must first match part or all of the address to an address in the reference data.

Reference data becomes available to you when you install DB2 Spatial Extender. For the amount of disk space that this data requires, see “Disk space requirements” on page 16. To verify on AIX that the data was loaded properly, look for it in the `$DB2INSTANCE/sqlib/gse/refdata/` directory. To verify on Windows NT that the data was loaded properly, look for it in the `%DB2PATH%\gse\refdata\` directory.

Resources that enable a database for spatial operations

The first task you perform after installing DB2 Spatial Extender is to enable your database for spatial operations. This involves initiating an action that causes DB2 Spatial Extender to load the database with the following resources:

- Stored procedures. When you request an action from the Control Center, DB2 Spatial Extender invokes one of these stored procedures to execute the action.
- Spatial data types. You must assign a spatial data type to each table or view column in which spatial data is to be stored. For more information, see “About spatial data types” on page 31.
- DB2 Spatial Extender’s catalog tables and views. Certain operations depend on the DB2 Spatial Extender catalog. For example, before a column with a spatial data type can be populated, it must be registered in the catalog as a layer. For information about layers, see “Developing and implementing a GIS project” on page 10.
- A spatial index type. It allows you to define indexes for layers.
- Spatial functions. You use these to work with spatial data in a number of ways; for example, to determine relationships between geographic features and to generate more spatial data. One of these functions is a default geocoder. It translates addresses in the United States into coordinates, and then inserts these coordinates into spatial columns. For more information about spatial functions, see “Chapter 13. Geometries and associated spatial functions” on page 119 and “Chapter 14. Spatial functions for SQL queries” on page 155. For more information about the default geocoder, see “About geocoding” on page 37.
- A schema, called DB2GSE, that contains the objects just listed.

For instructions on how to use the Control Center to initiate the loading of these resources, see “Enabling a database for spatial operations”. For guidelines on using a routine in an application program to perform the same task, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Enabling a database for spatial operations

To find out what authorization is required to enable a database for spatial operations, see “Authorization” on page 73.

To enable a database for spatial operations from the Control Center:

1. From the Control Center window, expand the object tree until you find the **Databases** folder under the server where you want DB2 Spatial Extender to run.
2. Click the **Databases** folder. The databases are displayed in the contents pane on the right side of the window.
3. Right-click the database that you want, and click **Spatial Extender** —> **Enable** in the pop-up menu. DB2 Spatial Extender supplies the database with the resources that allow you to create and work with spatial columns and data.

Reminder: Before you can enable a database for spatial operations, DB2 Spatial Extender must be installed on the server where the database resides.

Creating a spatial reference system

This section describes the relationship between spatial reference systems and coordinate systems, and explains how to create a spatial reference system from the Control Center.

About coordinate and spatial reference systems

This section continues the discussion of coordinate systems that was begun in “The nature of spatial data” on page 5. Then it expands on the definition of spatial reference systems that is provided in “Developing and implementing a GIS project” on page 10. It also provides guidelines for determining what values to assign to a spatial reference system’s parameters.

Coordinate systems, coordinates, and measures

You can think of a coordinate system in terms of an imaginary grid that covers a specific geographic area. Examples include a grid that covers the earth, a grid that covers a nation, or a grid that covers a region in a state. Each geographic feature in the area is situated at the intersection of an east-west gridline and a north-south gridline. A value, called an *X coordinate*, indicates where the location lies on the east-west gridline. Another value, a *Y coordinate*, indicates where the location lies on the north-south gridline. Both values reference the location to the grid’s center, or *origin*.

The X and Y coordinates at the origin are both zero. From the origin eastward, X coordinates are positive; from the origin westward, they are negative. Similarly, from the origin northward, Y coordinates are positive; from the origin southward, they are negative. For an illustration of this distribution, consider the following generalized example: Coordinate system A includes a grid that covers a large metropolitan area. An X coordinate of 7 would denote a position that is seven units of measurement eastward from the origin of this grid. An X coordinate of -9.5 would denote a position that is nine and a half units of measurement westward from the origin.

Each data item in a spatial column includes either (1) an X coordinate and a Y coordinate that define the location of a geographic feature or (2) multiple X and Y coordinates that define the locations of the parts of a feature, or that define the area that a feature covers. Two other kinds of values—a *Z coordinate* and a *measure*—can also be included. Unlike X and Y coordinates, Z coordinates and measures are not used in DB2 Spatial Extender to define locations or areas. Rather, they simply convey information required by a GIS application. A Z coordinate typically indicates the height or depth of a geographic feature. Z coordinates above the origin are positive; Z coordinates

below it are negative. A measure is numeric; it can convey any sort of information. For example, suppose that you are representing oil wells in your GIS. If you require your applications to process values that denote shot point IDs for seismic data, you could store these values as measures.

Spatial reference systems, offsets, and scale factors

As indicated in “Coordinate systems, coordinates, and measures” on page 23, coordinates can be negative and expressed in decimals. The same is true for measures. However, to reduce storage overhead, DB2 Spatial Extender stores each coordinate and measure as a non-negative integer (that is, as a positive integer or as zero). Therefore, actual negative and decimal coordinates and measures must be converted to non-negative integers, so that DB2 Spatial Extender can store them. Furthermore, you need to tell DB2 Spatial Extender how to make the conversion. You do this by setting certain parameters. Parameter settings that are to be used to convert coordinates and measures within a specific geographic area are collectively called a *spatial reference system*.

You can create a spatial reference system by:

- Determining the lowest negative coordinates and measures for the features that you are representing. (The further from zero a negative value is, the lower it is. An X coordinate of -10 is lower than an X coordinate of -5; a measure of -100 is lower than a measure of -50.)
- Specifying *offset factors* (or *offsets*, for short): values that, when subtracted from negative coordinates and measures, leave non-negative numbers.
- Specifying *scale factors*: values that, when multiplied by decimal coordinates and measures, yield integers whose precision is at least the same as that of the coordinates or measures. For example, consider a coordinate with a precision of four: 92.77. You could multiply it by a scale factor of 100 to obtain an integer with a precision of four: 9277.

Determining the lowest negative coordinates and measures

Before you set parameters for a spatial reference system, you need to determine the lowest negative X coordinate, Y coordinate, Z coordinate, and measure in the geographic area that contains the features that you want information about. You can find out what these values are by answering the following questions:

- Of the features that you are representing, do any lie west of the origin of the coordinate system that you are using? If so, what X coordinate indicates the location or western edge of the westernmost feature? (The answer will be the lowest of the negative X coordinates that you are dealing with.) For example, if you are representing oil wells, and some of them lie to the west of the origin, what X coordinate indicates the location of the oil well that is furthest west?

- Do any features lie south of the origin? If so, what Y coordinate indicates the location or southern edge of the southernmost feature? (The answer will be the lowest of the negative Y coordinates that you are dealing with.) For example, if you are representing oil wells, and some of them lie to the south of the origin, what Y coordinate indicates the location of the oil well that is furthest south?
- If you are going to use Z coordinates to define depth, which feature is the deepest, and which Z coordinate represents this feature's lowest point? (The answer will be the lowest of the negative Z coordinates that you are dealing with.)
- If you are going to include measures in your spatial data, will any be negative? If so, what is the lowest of the negative measures?

Having ascertained the lowest negative coordinates and measures, add to each one an amount equal to five to ten percent of its value. For example, if the lowest negative X coordinate is -100, you could add -5 to it. This book calls the resulting figure an *augmented value*.

Specifying offset factors

Next, specify what offset factors DB2 Spatial Extender should use to convert negative coordinates and measures to non-negative ones:

- After you decide what you want your augmented X value to be, specify an offset that, when subtracted from this value, leaves zero. DB2 Spatial Extender will then subtract this number from all negative X coordinates to arrive at a positive value. DB2 Spatial Extender will subtract this number from all other X coordinates as well.

For example, if the augmented X value is -105, you need to subtract -105 from it to get 0. DB2 Spatial Extender will then subtract -105 from all X coordinates that are associated with the features that you are representing. Because none of these coordinates is greater than -100, all the values that result from the subtraction will be positive.

- Similarly, specify offsets that leave 0 when subtracted from the augmented Y value, the augmented Z value, and the augmented measure.

The offset subtracted from X coordinates is called a *false X*. The offsets subtracted from Y coordinates, Z coordinates, and measures are called *false Y*, *false Z*, and *false M*, respectively. For instructions on specifying these parameters from the Control Center, see "Creating a spatial reference system from the Control Center" on page 26.

Specifying scale factors

Next, specify what scale factors DB2 Spatial Extender should use to convert decimal coordinates and measures to integers:

- Specify a scale factor that, when multiplied by a decimal X coordinate or a decimal Y coordinate, yields a 32-bit integer. It is advisable to make this scale factor a factor of 10: 10 to the first power (10), 10 to the second power

(100), 10 to the third power (1000), or, if necessary, a larger factor. To decide what factor of 10 the scale factor should be:

1. Determine which X and Y coordinates are, or are likely to be, decimal numbers. For example, suppose that of the various X and Y coordinates that you will be dealing with, you determine that three of them are decimal numbers: 1.23, 5.1235, and 6.789.
2. Take the decimal coordinate that has the longest decimal precision. Then determine by what factor of ten this coordinate can be multiplied in order to yield an integer of equal precision. To illustrate: of the three decimal coordinates in the current example, 5.1235 has the longest decimal precision. Multiplying it by ten to the fourth power (10000) would yield the integer, 51235.
3. Determine whether the integer produced by the multiplication that was just described is too long to store as a 32-bit data item. 51235 is not too long. But suppose that in addition to 1.23, 5.11235, and 6.789, your range of X and Y coordinates includes a fourth decimal value, 10006.789876. Because this coordinate's decimal precision is longer than that of the other three, you would multiply *this* coordinate—not 5.1235—by a factor of 10. To convert it to an integer, you could multiply it by 10 to the sixth power (1000000). But the resulting value, 10006789876, is too long to store as a 32-bit data item. If DB2 Spatial Extender tried to store it, the results would be unpredictable.

To avoid this problem, select a factor of 10 that, when multiplied by the original coordinate, yields a decimal number that DB2 Spatial Extender can truncate to a storable integer, with minimum loss of precision. In this case, you could select 10 to the fourth power (10000). Multiplying 10000 by 10006.789876 yields 100067898.76. DB2 Spatial Extender would truncate this number to 100067898, reducing its accuracy by a virtually insignificant amount.

- If the features that you are representing have decimal Z coordinates, follow the foregoing procedure to ascertain a scale factor for these coordinates. If the features are associated with decimal measures, follow this same procedure to ascertain a scale factor for these measures.

The scale factor for X and Y coordinates is called an *XY unit*. The scale factors for Z coordinates and measures are called *Z units* and *M units*, respectively. For instructions on specifying these parameters from the Control Center, see "Creating a spatial reference system from the Control Center".

Creating a spatial reference system from the Control Center

This section gives an overview of the steps to create a spatial reference system from the Control Center. The overview is followed by details of how to complete each step.

No authorization is required to perform these steps.

Overview of steps to create a spatial reference system from the Control Center:

1. Open the Create Spatial Reference window.
2. Indicate which coordinate system you want to use.
3. Specify identifiers for the spatial reference system that you want to create.
4. Determine what ranges of coordinates and measures apply to the geographic features that you want information about.
5. Specify values that can be used to convert negative or decimal coordinates and measures into data items that DB2 Spatial Extender can store.
6. Tell DB2 Spatial Extender to create the spatial reference system that you want.

Detailed steps to create a spatial reference system from the Control Center:

1. Open the Create Spatial Reference window.
 - a. From the Control Center window, expand the object tree until you find the **Databases** folder under the server where you want DB2 Spatial Extender to run.
 - b. Click the **Databases** folder. The databases are displayed in the contents pane on the right side of the window.
 - c. Right-click the database that you enabled for spatial data, and click **Spatial Extender** → **Spatial References** in the pop-up menu. The Spatial References window opens.
 - d. From the Spatial References window, click **Create**. The Create Spatial Reference window opens.
2. From the Create Spatial Reference window, use the **Coordinate system** field to indicate what coordinate system you want to use.
3. Specify identifiers for the spatial reference system that you want to create.
 - In the **Name** field, type a 1- to 64-character name for the system.

Restriction: Do not specify the name of another spatial reference system. No two spatial reference systems in the database can have the same name.

- In the ID field, type a numerical identifier. It must be an integer.

Restriction: Do not specify the ID of another spatial reference system. No two spatial reference systems in the database can have the same ID.

4. Using a medium outside the Control Center—for example, paper or a white board—determine the lowest negative coordinates and measures that apply to the geographic features that you are representing. For guidelines on how to do this, see “Determining the lowest negative coordinates and measures” on page 24.

5. From the Create Spatial Reference window, specify values to convert negative or decimal coordinates and measures into data items that DB2 Spatial Extender supports—that is, into 32-bit non-negative integers.
 - a. Specify values to convert negative or decimal X coordinates into non-negative integers:
 - In the **Offset** column, in the field nearest to the **X**, specify a false X:
 - If any values within the range of X coordinates that you identified in step 4 on page 27 are negative, type a false X that, when subtracted from the lowest negative coordinate, leaves a positive number. For guidelines, see “Specifying offset factors” on page 25.
 - If all the X coordinates are non-negative, type a false X of 0.
 - In the **Scale factor** column, specify an XY unit in the field to the far right of the **X**. This XY unit should be one that, when multiplied by any decimal X coordinate or decimal Y coordinate, yields a whole number that can be stored as a 32-bit data item, with minimum loss of precision. For guidelines, see “Specifying scale factors” on page 25. After you specify the XY unit in the field to the far right of the **X**, it will appear also in the field to the far right of the **Y**.
 - b. Specify a false Y that will allow DB2 Spatial Extender to convert negative Y coordinates into positive values. You do this in the **Offset** column, in the field nearest to the **Y**:
 - If any values within the range of Y coordinates that you identified in step 4 on page 27 are negative, type a false Y that, when subtracted from the lowest negative coordinate, leaves a positive number. For guidelines, see “Specifying offset factors” on page 25.
 - If all the Y coordinates are positive, type a false Y of 0.
 - c. If you are going to include Z coordinates in your spatial data, specify values to convert negative or decimal Z coordinates into non-negative integers:
 - In the **Offset** column, in the field nearest to the **Z**, type a false Z:
 - If any values within the range of Z coordinates that you identified in step 4 on page 27 are negative, type a false Z that, when subtracted from the lowest negative coordinate, leaves a positive number. For guidelines, see “Specifying offset factors” on page 25.
 - If all the Z coordinates are non-negative, type a false Z of 0.
 - In the **Scale factor** column, specify a Z unit in the field to the far right of the **Z**. This Z unit should be one that, when multiplied by any decimal Z coordinate, yields a whole number that can be stored as 32-bit data item, with minimum loss of precision. For guidelines, see “Specifying scale factors” on page 25.
 - d. If you are going to include measures in your spatial data, specify values to convert negative or decimal measures into positive integers:

- In the **Offset** column, in the field nearest to the **Linear** label, type a false M:
 - If any values within the range of measures that you identified in step 4 on page 27 are negative, type a false M that, when subtracted from the lowest negative measure, leaves a positive number. For guidelines, see “Specifying offset factors” on page 25.
 - If all the measures are positive, type a false M of 0.
 - In the **Scale factor** column, specify an M unit in the field to the far right of the **Linear** label. This M unit should be one that, when multiplied by any decimal measure, yields a whole number that can be stored as 32-bit data item, with minimum loss of precision. For guidelines, see “Specifying scale factors” on page 25.
6. Click **OK** to create the spatial reference system that you want.

Chapter 4. Defining spatial columns, registering them as layers, and enabling a geocoder to maintain them

After you set up resources for your DB2 Spatial Extender GIS, you are ready to create objects that will contain spatial data. For example, if you need new tables to contain spatial data, you can define them, assigning spatial data types to the columns that you want the data to go into. If you need to add spatial columns to existing tables, you can do that also.

When you provide a new or existing table with a spatial column, you need to register this column as a layer. In addition, if you plan to have a geocoder populate the column, you can, at the time that you register the column as a layer, enable the geocoder to maintain it automatically. This enablement occurs in the following way: DB2 Spatial Extender defines triggers that are coded to invoke the geocoder whenever the spatial column's corresponding attribute column (or columns) receive new or updated data. When invoked, the geocoder translates the new or updated data into spatial data, and places this spatial data into the spatial column.

After you define a spatial column for a table, you can, if you choose, create a view column over this table column. You must register the view column as a layer after you register the table column as a layer.

This chapter discusses the nature and use of the data types that you can assign to spatial column. Next, the chapter explains how to use the Control Center to define a spatial column for a table, to register this column as a layer, and to enable a geocoder to maintain it. Finally, the chapter explains how to use the Control Center to register a view column as a layer.

About spatial data types

This section introduces the data types that are required for spatial columns and provides guidelines for choosing what a spatial column's data type should be.

When you enable a database for spatial operations, DB2 Spatial Extender supplies the database with a hierarchy of structured data types. Figure 6 on page 32 presents this hierarchy. In this figure, the instantiable types have a white background; the uninstantiable types have a shaded background.

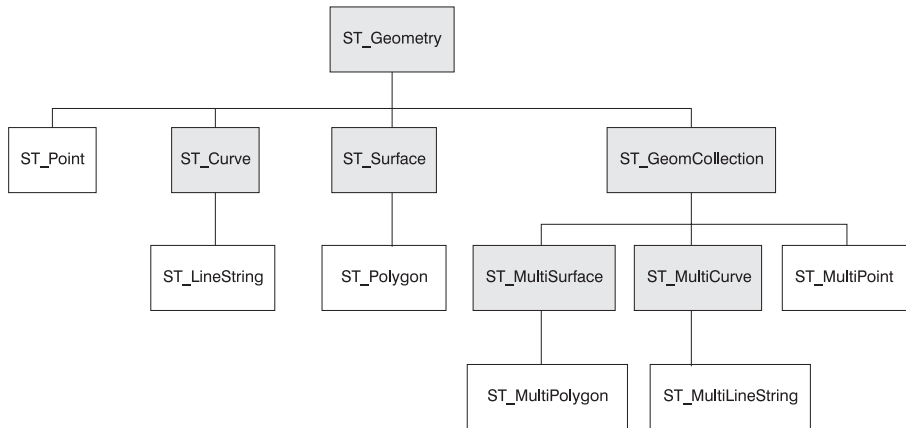


Figure 6. Hierarchy of spatial data types. Data types named in white boxes are instantiable. Data types named in shaded boxes are uninstantiable.

The hierarchy in Figure 6 includes:

- Data types for geographic features that can be perceived as forming a single unit; for example, individual residences and isolated lakes.
- Data types for geographic features that are made up of multiple units or components; for example, highway systems and mountain ranges.
- A data type for geographic features of all kinds.

Data types for single-unit features

Use `ST_Point`, `ST_LineString`, and `ST_Polygon` to store coordinates that define the space occupied by features that can be perceived as forming a single unit:

- Use `ST_Point` when you want to indicate the point in space that is occupied by a discrete geographic feature. The feature might be a very small one, such as a water well; a very large one, such as a city; or one of intermediate size, such as a building complex or park. In each case, the point in space can be located at the intersection of an east-west coordinate line (for example, a parallel) and a north-south coordinate line (for example, a meridian). An `ST_Point` data item includes values—an X coordinate and a Y coordinate—that define such an intersection. The X coordinate indicates where the intersection lies on the east-west line; the Y coordinate indicates where the intersection lies on the north-south line.
- Use `ST_LineString` for coordinates that define the space that is occupied by linear features; for example, streets, canals, and pipelines.
- Use `ST_Polygon` when you want to indicate the extent of space covered by a multi-sided feature; for example, a welfare district, a forest, or a wildlife habitat. An `ST_Polygon` data item consists of the coordinates that define the perimeter of such a feature.

In some cases, `ST_Polygon` and `ST_Point` can be used for the same feature. For example, suppose that you need spatial information about several apartment complexes. If you want to represent the point in space where each complex is located, you would use `ST_Point` to store the X and Y coordinates that define each such point. On the other hand, if you want to represent the area that each complex covers, you would use `ST_Polygon` to store the coordinates that define the perimeter of each such area.

Data types for multi-unit features

Use `ST_MultiPoint`, `ST_MultiLineString`, and `ST_MultiPolygon` to store coordinates that define spaces occupied by features that are made up of multiple units:

- Use `ST_MultiPoint` when you want to represent features made up of discrete units, and you want to indicate the point in space occupied by each component. An `ST_MultiPoint` data item includes the pairs of X and Y coordinates that define the location of each component of such a feature. For example, consider a table whose rows represent island chains and whose columns include an `ST_MultiPoint` column. Each data item in this column includes the pairs of X and Y coordinates that define the locations of the islands in each chain.
- Use `ST_MultiLineString` when you want to represent features made up of linear units, and you want information about the space occupied by each unit. An `ST_MultiLineString` data item consists of the coordinates that define such spaces. For example, consider a table whose rows represent river systems and whose columns include an `ST_MultiLineString` column. Each data item in this column includes the sets of coordinates that define the paths of the rivers in each system.
- Use `ST_MultiPolygon` when you want to represent features made up of multi-sided units, and you want information about the space occupied by each unit. For example, consider a table whose rows represent midwestern counties and whose columns include an `ST_MultiPolygon` column. This column contains information about farmlands. Specifically, each data item in the column includes the sets of coordinates that define the perimeters of the farmlands in a particular county.

A data type for all features

You can use `ST_Geometry` when you are not sure which of the other data types to use. Because `ST_Geometry` is the root of the hierarchy to which the other data types belong, an `ST_Geometry` column can store any or all of the values that can be stored in columns to which the other data types are assigned.

Attention: The default geocoder can translate addresses into `ST_Point` or `ST_Geometry` data items. Therefore, if you plan to use this geocoder to populate a spatial column, you must assign either the `ST_Point` or the `ST_Geometry` data type to this column.

Defining a spatial column for a table, registering this column as a layer, and enabling a geocoder to maintain it

This section gives an overview of the steps to define a spatial column for a table, register this column as a layer, and enable a geocoder to maintain it. The overview is followed by details of how to complete each step.

To find out what authorization you need to register a table column as a layer, see “Authorization” on page 86. To find out what authorization you need to enable a geocoder to maintain this column, see “Authorization” on page 70.

Overview of steps to define a spatial column for a table, register this column as a layer, and enable a geocoder to maintain it:

1. If the spatial column is to be part of a new table, create this table.
2. Open the Create Spatial Layer window.
3. Either add a spatial column to a table, and indicate that you want to register this column as a layer; or indicate that you want to register an existing column as a layer.
4. Indicate which spatial reference system is to be used for the layer.
5. If the layer is to contain imported data, or data that is generated from another spatial column, tell DB2 Spatial Extender to create the layer.
6. If the layer is to contain data derived from attribute data:
 - a. Specify which column or columns contain this attribute data.
 - b. Indicate that you want to enable a geocoder to maintain the layer.
 - c. Tell DB2 Spatial Extender to create the layer.

Detailed steps to define a spatial column for a table, register this column as a layer, and enable a geocoder to maintain it:

1. If the spatial column is to be part of a new table, create this table:
 - Use an interface of your choice (for example, the Control Center or the Command Line Processor) to create the table.
 - If you plan to use a geocoder, include one to ten columns for the geocoder to operate on. A geocoder cannot take more than ten columns of data as input.
 - Either include the spatial column that you will be registering as a layer, or define this column in step 3 on page 35.

If you want to use an existing table, go on to the next step.

2. Open the Create Spatial Layer window.
 - a. From the Control Center window, expand the object tree until you find the **Tables** folder for the tables in the database that you use for spatial operations.

- b. Click the **Tables** folder. The tables are displayed in the contents pane on the right side of the window.
 - c. Right-click the table that you want and click **Spatial Extender** → **Spatial Layers** in the pop-up menu. The Spatial Layers window opens.
 - d. From the Spatial Layers window, click **Create**. The Create Spatial Layer window opens.
3. From the Create Spatial Layer window, either add a spatial column to a table, and indicate that you want to register this column as a layer; or indicate that you want to register an existing column as a layer.
 - If you want to add a spatial column to a table and define this column as a layer:
 - a. In the **Layer column** field, type a name for the column.
 - b. In the **Column type** field, select or type the data type that you want the column to have. For a discussion of allowable data types, see “About spatial data types” on page 31.
 - If you want to define an existing column as a layer, select it in the **Layer column** field.

Restriction: Do not select a column that has already been defined as a layer.

4. In the **Spatial reference name** field, specify the name of the spatial reference system to be used for the layer.
5. If you want the layer to contain imported data, or data that is generated from another spatial column, click **OK** to register it.
6. If you want the layer to contain data that is derived from attribute data:
 - a. Specify which column or columns contain this attribute data:
 - 1) Select the column or columns in the **Available columns** box. You can select up to ten columns.
 - 2) Click the > push button, the >> push button, or both, to list the selected column or columns in the **Selected columns** box.
 - b. If you want to enable a geocoder to maintain the layer:
 - 1) Select the **Enable automatic geocoder** check box.
 - 2) In the **Name** field, select the name of the geocoder that you want to use.
 - 3) In the **Precision level** field, specify, in terms of percentage, the degree to which input records must match corresponding records in the reference data in order to be processed. This percentage is called a *precision*. For example, suppose that the geocoder reads an input record that contains the address, 557 Bailey, San Jose 94120. If the precision is 100, and if the match between this address and its counterpart in the reference data is not 100 percent accurate, the geocoder will reject it. If the precision is 75, and the match between

- the record and its reference data counterpart is at least 75 percent accurate, the geocoder will process it.
- 4) If the geocoder was supplied by a vendor, use the **Properties** box to specify any vendor-supplied geocoding parameters that you want to use.
 - c. Click **OK** to register the selected column as a layer and, if you requested it, to enable the geocoder to maintain the column.

Registering a view column as a layer

To find out what authorization you need to register a view column as a layer, see “Authorization” on page 86.

To register a view column as a layer:

1. Open the Create Spatial Layer window.
 - a. From the Control Center window, expand the object tree until you find the **Views** folder for the views in the database that you use for spatial operations.
 - b. Click the **Views** folder. The views are displayed in the contents pane on the right side of the window.
 - c. Right-click the view that you want and click **Spatial Extender** → **Spatial Layers** in the pop-up menu. The Spatial Layers window opens.
 - d. From the Spatial Layers window, click **Create**. The Create Spatial Layer window opens.
2. Use the **Layer column** box to specify the column that you want to register as a layer.
3. In the **Underlying spatial layer field**, specify the name of the table column on which the selected view column is based. This table column must already be registered as a layer.
4. Click **OK** to register the specified view column as a layer.

Chapter 5. Populating spatial columns

After you register spatial columns as layers, you are ready to supply them with spatial data. As noted in “Where spatial data comes from” on page 6, there are three ways to supply this data: use a function, called a geocoder, to derive it from attribute data; use other functions to derive it from other spatial data; or import it from files. This chapter:

- Talks about geocoding and explains how to use the Control Center to geocode attribute data in batch mode
- Discusses importing and exporting data and explains how to use the Control Center to import data to your GIS and export it from your GIS

For information about functions that can derive new spatial data from existing spatial data, see “Functions that generate new geometries from existing ones” on page 144.

Using geocoders

This section describes the process of geocoding and explains how to run a geocoder in batch mode from the Control Center.

About geocoding

This section distinguishes basic differences between geocoders and their sources. It also describes the two modes in which a geocoder can operate, and introduces factors to consider when you plan to use a geocoder.

With DB2 Spatial Extender, you can:

- Use the default geocoder that is supplied with DB2 Spatial Extender.
- Plug in geocoders that are developed by third-party vendors.
- Plug in your own geocoders.

The default geocoder geocodes United States addresses, and can translate them into either ST_Point data or ST_Geometry data. If you need to store data of other spatial data types, you could plug in a geocoder to generate such data. If you need spatial data that represents sites outside the United States, or sites that have no addresses—for example, farmlands that vary in soil content—you could plug in a geocoder to meet that need as well.

Before a plug-in geocoder can be used, it must be registered. Users and vendors can register it with the `db2gse.gse_register_gc` stored procedure. It cannot be registered from the Control Center. For information about `db2gse.gse_register_gc`, see “`db2gse.gse_register_gc`” on page 84. For general

information about using the DB2 Spatial Extender stored procedures, see “Chapter 9. Stored procedures” on page 63.

A geocoder operates in two modes:

- In *batch mode*, it attempts, in a single operation, to translate all existing source data for a spatial column into spatial data, and to populate the column with that data. You can initiate this operation from the Run Geocoder window. Alternatively, you can initiate it in an application program, by coding the program to call the `db2gse.gse_run_gc` stored procedure.
- In *incremental mode*, a geocoder translates data when it is inserted or updated in a table, placing the resulting spatial values in a column in order to keep the column current. It is activated by insert and update triggers that you can request from the Create Spatial Layer window. Alternatively, you can request them in an application program, by coding the program to call the `db2gse.gse_enable_autogc` stored procedure.

Incremental geocoding is referred to also as *automatic geocoding*.

When planning to use a geocoder, you might consider the following factors:

1. When you use the Control Center, you typically use the Create Spatial Layers window before you use the Run Geocoder window. This means that you can have DB2 Spatial Extender set up triggers for incremental geocoding before you initiate batch geocoding. Therefore, it is possible for incremental geocoding to precede batch geocoding. Processing all source data in batch mode, the geocoder will geocode the same data that it operated on in incremental mode. This redundancy will not cause duplications (when spatial data is produced twice, the second yield of data will override the first). However, it can degrade performance. One way to avoid it is to defer setting up the triggers until after batch geocoding is done.
2. If the triggers are in place when you are ready to geocode in batch mode, it is advisable to deactivate them until the batch geocoding is over. You can deactivate them either from the Run Geocoder window or in an application program, by coding the program to call the `db2gse.gse_disable_autogc` stored procedure. If you use the Run Geocoder window, DB2 Spatial Extender reactivates them automatically when the geocoding is over. If you use the `db2gse.gse_disable_autogc` stored procedure, you can reactivate them by calling the `db2gse.gse_enable_autogc` stored procedure.
3. If you want to run a geocoder in batch mode to populate a spatial column that has an index, disable or drop the index first. Otherwise, if the index remains operable while the geocoder runs, performance will be degraded severely. If you are using the Control Center, you can disable the index from the Run Geocoder window. DB2 Spatial Extender re-enables the index automatically when the geocoding is over. If you are using an

application program, you can drop the index with the SQL DROP statement. If you do this, be sure to keep a note of the index's parameters, so that you can recreate it after the batch geocoding is over.

4. When the geocoder reads a record of source data, it tries to match that record with a counterpart in the reference data. The match must be accurate to a certain degree (called a *precision*) in order for the geocoder to process the record. For example, a precision of 85 means that the match between a source record and its counterpart in the reference data must be at least 85 percent accurate in order for the source record to be processed. You specify what the precision should be. Be aware that you might need to adjust it. For example, suppose that the precision is 100. If many source records contain addresses that are more recent than the reference data, matches of 100 percent accuracy between these records and the reference data will be impossible. As a result, the geocoder will reject these records. On the whole, if a geocoder produces spatial data that seems insufficient or largely inaccurate, you might be able to resolve this problem by changing the precision and running the geocoder again.

Running the geocoder in batch mode

This section gives an overview of the steps to run a geocoder in batch mode from the Control Center. The overview is followed by details of how to complete each step.

To find out what authorization you need to run a geocoder in batch mode, see "Authorization" on page 92.

Overview of steps to run a geocoder in batch mode:

1. Open the Run Geocoder window.
2. Indicate which geocoder you want to use.
3. Disable objects that could impede the performance of the geocoder.
4. Specify how many records to geocode before DB2 issues a commit.
5. Indicate how you want the geocoder to operate.
6. Tell DB2 Spatial Extender to run the geocoder.

Detailed steps to run a geocoder in batch mode:

1. Open the Run Geocoder window.
 - a. From the Control Center window, expand the object tree until you find the **Tables** folder in your spatially-enabled database.
 - b. Click the **Tables** folder. The tables are displayed in the contents pane on the right side of the window.
 - c. Right-click the table that you want in the contents pane and click **Spatial layers** in the pop-up menu. The Spatial Layers window opens.
 - d. From the Spatial Layers window:

- 1) Select the layer that is defined on the column that you want to populate.
 - 2) Click the **Run Geocoder** push button. The Run Geocoder window opens.
2. If you want to use the default geocoder, leave the **Name** box, which displays the name of this default, as is. Otherwise, use the box to select the geocoder that you want.
 3. Disable objects that could impede the performance of the geocoder:
 - If the column that you want to populate has an index, select the **Temporarily disable spatial indexes during geocoding process** check box.
 - If triggers have been set to activate incremental geocoding for this column, select the **Temporarily disable spatial triggers during geocoding process** check box.

The index and triggers will be re-enabled automatically when you click **OK** on the Run Geocoder window.
 4. Use the **Commit scope** spin button to specify how many records to geocode before DB2 issues a commit. For example, if you want DB2 to commit 100 geocoded records at a time, specify the number, 100.
- Tip:** If you want DB2 to issue a commit only after all records are processed, specify zero.
5. Use the fields in the **Geocoder parameters** group box to indicate how you want the geocoder to operate:
 - Use the **Precision level** spin button to specify, in terms of a percentage, how accurate the match between source records and their reference data counterparts should be. For more information about precision, see “About geocoding” on page 37.
 - If you are using a vendor-supplied geocoder, and want to use properties that it supports, use the **Properties** box to set these properties.
 - If you want to geocode only a subset of rows in the table that you selected, use the **WHERE clause** box to code a SELECT WHERE clause that will specify the criteria for the rows that you want. This clause can reference any columns in the table.

Type the criteria only. Omit the keyword WHERE. For example, if the table has a STATE column, and you want to geocode only those rows that contain the value MA in this column, type:

```
STATE='MA'
```
 6. Click **OK** to run the geocoder.

Importing and exporting data

This section describes the processes of importing and exporting data, and explains how to use the Control Center to:

- Import data from a data exchange file to a new or existing table
- Import data from a data exchange file to an existing table
- Export data from a table to a data exchange file

About importing and exporting

This section lists reasons for importing and exporting spatial data. It also discusses the data exchange files that serve as interfaces between sources of the export and targets of the import.

You can use DB2 Spatial Extender to import spatial data from, and export it to, data exchange files. Consider these scenarios:

- Your GIS contains spatial data that represents your offices, customers, and other business concerns. You want to supplement this data with spatial data that represents your organization's cultural environment—cities, streets, points of interest, and so on. The data that you want is available from a map vendor. You can use DB2 Spatial Extender to import it from a data exchange file that the vendor supplies.
- You want to migrate spatial data from an Oracle system to your DB2 Spatial Extender GIS. You proceed by using an Oracle utility to load the data into a data exchange file. You then use DB2 Spatial Extender to import the data from this file to the database that you have enabled for spatial operations.
- You want to use a GIS browser to show visual presentations of spatial information to customers. The browser needs only files to work from; it does not need to be connected to a database. You could use DB2 Spatial Extender to export the data to a data exchange file, and then use a browser utility to load the data into the browser.

The Control Center supports two kinds of data exchange files for DB2 Spatial Extender: shape files and ESRI_SDE transfer files. Shape files are often used for importing data that originates in file systems and for exporting data to files that are to be loaded into file systems. ESRI_SDE transfer files are often used for importing data that originates in ESRI databases.

Importing data to a new or existing table

This section gives an overview of the steps to import data from a shape or ESRI_SDE transfer file to a new or existing table. The overview is followed by details of how to complete each step.

To find out what authorization is required for importing shape data, see "Authorization" on page 82. To find out what authorization is required for importing ESRI_SDE data, see "Authorization" on page 80.

Overview of steps to import data to a new or existing table:

1. Open the Import Spatial Data window.
2. Specify the path, name, and format of the file that contains the data to be imported.
3. Specify how many records to import before each commit.
4. If you want to import spatial data to a table that is to be created, supply a name for this table and a name for the column that the data is intended for. If importing spatial data to an existing table, indicate which column the data is intended for.
5. Specify which spatial reference system is to be associated with the data.
6. Designate a file to collect the records that fail the import.
7. Tell DB2 Spatial Extender to import the data and, if you defined a table from this window, to create the table and to register the column for which the data is intended as a layer.

Details of steps to import data to a new or existing table:

1. Open the Import Spatial Data window.
 - a. From the Control Center window, expand the object tree until you find the **Databases** folder under the server where you are running DB2 Spatial Extender.
 - b. Click **Databases** folder. The databases are displayed in the contents pane on the right side of the window.
 - c. Right-click the database to which you want to import data and click **Spatial Extender** —> **Import Spatial Data** in the pop-up menu. The Import Spatial Data window opens.
2. Specify the path, name, and format of the file that contains the data to be imported:
 - a. Use the **File name** field to specify the path and name.
 - b. Use the **File format** box to specify the format. The format can be:
 - Shape** This is the default.
 - ESRI_SDE**
If you specify this format, the **Spatial reference name** field defaults to the name of the spatial reference system that is associated with this format.
3. Use the **Commit scope** field to specify the number of records that you want imported before each commit. For example, to require DB2 to commit 100 records at a time, specify the number 100.

Tip: If you want DB2 to issue a commit only after all records are processed, specify zero.

4. Specify the table and column that the data is intended for.

- a. Use the **Layer schema** box to specify the schema for the table to which data is to be imported.
- b. Specify the table and the column:
 - If the table does not yet exist:
 - 1) In the **Layer table** field, type a name for the table.
 - 2) In the **Layer column** field, type a name for the column that is to contain the imported data. DB2 Spatial Extender will automatically register this column as a layer.
 - If the table already exists:
 - 1) In the **Layer table** field, specify the table. It must already contain the column that you want the imported data to go into. In addition, this column must already be registered as a layer.
 - 2) In the **Layer column** field, specify the name of the column that the imported data is intended for.
5. In the **Spatial reference name** field, type or select the spatial reference system that is to be associated with this data. (If the data is to come from an ESRI_SDE transfer file, the name of the associated spatial reference system is displayed in the field automatically.)
6. In the **Exception file** field, specify the path and name for a new file into which records that fail the import can be collected. Later, you can fix these records and import them from this file.
DB2 Spatial Extender will create this file; do not specify one that already exists.
7. Click **OK** to import the data. Also, if you supplied a name for a table that does not exist yet, this table will be created and the column for which the data is intended will be registered as a layer. In addition, the exception file that you specified will be created.

Importing data to an existing table

This section gives an overview of the steps to import data from a shape file or ESRI_SDE transfer file to an existing table. The overview is followed by details of how to complete each step.

To find out what authorization is required for importing shape data, see “Authorization” on page 82. To find out what authorization is required for importing ESRI_SDE data, see “Authorization” on page 80.

Overview of steps to import data to an existing table:

1. Open the Import Spatial Data window.
2. Specify the path and name of the file that contains the data to be imported.
3. Specify how many records to import before each commit.

4. Specify the column that is to contain the spatial data that you are importing.
5. Specify which spatial reference system is to be associated with this data.
6. Designate a file to collect records that fail the import.
7. Tell DB2 Spatial Extender to import the data and, if you specified a column that has not been created yet, to create this column and to register it as a layer.

Details of steps to import data to an existing table:

1. Open the Import Spatial Data window.
 - a. From the Control Center window, expand the object tree until you find the **Tables** folder for the database that you want to import data to.
 - b. Click the **Tables** folder. The tables are displayed in the contents pane on the right side of the window.
 - c. Right-click the table to which you are importing data and click **Spatial Extender** → **Import Spatial Data** in the pop-up menu. The Import Spatial Data window opens.
2. In the **File name** box, specify the path and name of the file that contains the data to be imported.
3. Use the **Commit scope** box to specify the number of records that you want imported before each commit. For example, to require DB2 to commit 100 records at a time, specify the number 100.

Tip: If you want DB2 to issue a commit only after all records are processed, specify zero.

4. Specify the column that is to contain the spatial data that you are importing.
 - If the column does not yet exist in the table, use the **Layer column** box to type a name for the column.
 - If the column already exists, use the **Layer column** box to select or type the name of the column.
5. Use the **Spatial reference name** box to specify which spatial reference system is to be associated with the imported data.
 - If you are adding a column to a table, type or select the name of the spatial reference system.
 - If the imported data is intended for an existing column, leave the **Spatial reference name** box as is. It displays the name of the default spatial reference system.
6. In the **Exception file** field, specify the path and name for a new file into which records that fail the import can be collected. Later, you can fix these records and import them from this file.

DB2 Spatial Extender will create this file; do not specify one that already exists.

7. Click **OK** to import the data. Also, if you specified a column that does not exist yet, this column will be created and registered as a layer. In addition, the exception file that you specified will be created.

Exporting data to a shape file

This section gives an overview of the steps to export data to a shape file. The overview is followed by details of how to complete each step.

To find out what authorization is required for performing these steps, see “Authorization” on page 78.

Overview of steps to export data from a shape file:

1. Open the Export Spatial Data window.
2. Specify the column that contains the spatial data to be exported.
3. If you want to export a subset of rows of data, identify this subset to DB2 Spatial Extender.
4. Specify the path and name of the file that you are exporting data to.
5. Tell DB2 Spatial Extender to export the data.

Detail of steps to export data to a shape file:

1. Open the Export Spatial Data window.
 - a. From the Control Center window, expand the object tree until you find the **Tables** or **Views** folder in the database that contains spatial data:
 - b. Click the **Tables** or **Views** folder. Tables or views are displayed in the contents pane on the right side of the window.
 - c. Right-click the table or view that contains the data to be exported and click **Spatial Extender** → **Export Spatial Data** in the pop-up menu. The Export Spatial Data window opens.
2. In the **Layer column** field, specify the name of the column that contains the spatial data to be exported.
3. If you want to export a subset of table rows, use the **WHERE clause** box to type a WHERE clause that specifies the criteria for the rows that you want. In this clause, you can reference only columns in the table or view that you are exporting data from.

Type the criteria only. Omit the keyword WHERE. For example, if the table or view has a STATE column, and you want to geocode only those rows that contain the value MA in this column, type:

```
STATE='MA'
```

4. In the **File name** field, specify the path and name of the file that you are exporting data to.
5. Click **OK** to export the data.

Chapter 6. Creating spatial indexes

This chapter explains how to use the Control Center to create an index for your spatial data.

After you populate spatial columns with your data, you are ready to create a spatial index. Typical indexing structures, such as a B-tree, perform linear, one-dimensional sorts on table data. Table data that has been enabled for spatial operations is not stored as a single entry, but is two-dimensional. For example, spatial geometries such as a polygon consists of several coordinate values in one spatial column or layer. Because a B-tree index cannot handle spatial data types, DB2 Spatial Extender created a proprietary indexing technology known as a *grid index*. The grid index is based on the B-tree index, which was enhanced to handle two-dimensional data and perform indexing on spatial columns. The grid index supports three layers and is designed to provide good performance over a wide range of objects, sizes, and distributions of data. For more information about spatial indexes, see “Chapter 12. Spatial indexes” on page 109.

To find out what authorization is required for creating a spatial index, see “Authorization” on page 74.

Using the Control Center to create a spatial index

To create a spatial index using the Control Center:

1. In the object tree, select the **Tables** folder. All existing tables are displayed in the contents pane.
2. From the contents pane, right-click the table that you want to create an index for, and click **Spatial Extender** → **Spatial Indexes** in the pop-up menu. The Spatial Indexes window opens.
3. From the Spatial Indexes window, click **Create**. The Create Spatial Index window opens.
4. In the **Name** field, type the name of the new spatial index that you want to create.

Note: You do not need to specify a schema. DB2 Spatial Extender will automatically add the schema and create a fully qualified name for you.

5. In the **Layer column** field, select the layer that you are creating an index for.

A layer is a spatial column defined or registered to DB2 Spatial Extender.

6. In the **Grid size** fields, type the grid size value that you want to assign to each field.

The grid levels, **Finest**, **Middle**, and **Coarsest**, are entered by increasing the cell size. Thus, the second level must have a larger cell size than the first, and the third larger than the second.

Determining grid cell sizes

Determining the correct grid size is done through a process of trial and error. It is recommended that you set your grid size in relationship to the approximate size of the object that you are indexing. Grid sizes that are set too small or too large can result in slowed performance. Sizes that are set too small affect the key/object ratio during an index search. In this case, too many keys are created and a large number of candidates is returned. For grid sizes that are set too large, the initial index search returns a small number of candidates, but the performance might be slowed during the final table scan.

For more information about selecting grid cell sizes and the number of grid levels, see “Selecting the grid cell size” on page 116.

Chapter 7. Retrieving and analyzing spatial information

After you construct the spatial indexes, the spatial tables are ready for use. This chapter discusses issues related to retrieving and analyzing spatial data. It contains an overview of various retrieval methods and provides examples of table queries that utilize spatial functions.

Methods of performing spatial analysis

You can perform spatial analysis by using SQL and spatial functions with any of the following programming environments:

- A geobrowser (for example, ESRI's ArcExplorer).
For more information about using the ArcExplorer, refer to *Using ArcExplorer*, which is available on the ESRI Web site at <http://www.esri.com>.
- Interactive SQL statements.
- User-developed applications (for example, ODBC, JDBC, and embedded SQL).

Applications can be launched from the DB2 Command Center, the DB2 Command Window, or the command line processor.

Building a spatial query

This section discusses building spatial queries that utilize spatial functions and predicates.

Spatial functions and SQL

DB2 Spatial Extender includes functions that perform various operations on spatial data. The examples in this section show you how to use spatial functions to build your own spatial queries.

Table 3 provides a list of spatial functions and the types of operations they can perform.

Table 3. Spatial functions and operations

Function type	Operation example
Calculation	Calculate the distance between two points
Comparison	Find all customers located within a flood zone
Data exchange	Convert data into supported formats
Transformation	Add a five-mile radius to a point

For more information about spatial functions, see “Chapter 13. Geometries and associated spatial functions” on page 119 and “Chapter 14. Spatial functions for SQL queries” on page 155.

Example 1: Comparison

The following query finds the average customer distance from each department store. The spatial functions used in this example are ST_Distance and ST_Within.

```
SELECT s.id, AVG(db2gse.ST_Distance(c.location,s.location))
FROM customers c, stores s
WHERE db2gse.ST_Within(c.location,s.zone)=1
GROUP BY s.id
```

Example 2: Data exchange

The following query finds the customer locations for those who live in the San Francisco Bay Area. The spatial functions used in this example are ST_AsText (data exchange) and ST_Within. ST_AsText converts the spatial data in the c.location column into the OGC TEXT format.

```
SELECT db2gse.ST_AsText(c.location, cordref(1))
FROM customers c
WHERE db2gse.ST_Within(c.location, :BayArea)=1
```

Example 3: Calculation

The following query finds all streets longer than 10.5 miles. The spatial function used in this example is ST_Length.

```
SELECT s.name,s.id
FROM street s
WHERE db2gse.ST_Length(s.path) > 10.5
```

Example 4: Transformation

This query finds the customers who live within the flood zone or within 2 miles from the boundary of the flood zone. The spatial functions used in this example are ST_Buffer (transformation) and ST_Within.

```
SELECT c.name,c.phoneNo,c.address
FROM customers c
WHERE db2gse.ST_Within(c.location,ST_Buffer(:floodzone,2))=1
```

Spatial predicates and SQL

A specialized group of spatial functions that are called spatial predicates can improve query performance. Spatial predicates, such as ST_Overlaps, which compares two polygons to see if they overlap, can be expensive to execute for both time and memory requirements. Therefore, optimization techniques to minimize execution cost are very important. The DB2 query optimizer uses the spatial index to improve query performance when you use the spatial predicates according to the rules described later in this section. For more information about spatial predicates, see “Predicate functions” on page 132. The spatial predicates used to exploit the spatial index are:

- ST_Contains
- ST_Crosses
- ST_Disjoint
- ST_Distance
- ST_Envelope
- ST_Equals
- ST_Intersects
- ST_Overlaps
- ST_Touches
- ST_Within

For a complete list of all spatial functions and predicates, see “Chapter 14. Spatial functions for SQL queries” on page 155.

Rules for index exploitation

The following rules apply if you want to optimize spatial queries using spatial predicates:

- The predicate must be used in the WHERE clause.
- The predicate must be on left-hand side of the comparison. For example:
WHERE db2gse.ST_Within(c.location,:BayArea)=1
- Equality comparisons must use the integer constant 1.
WHERE db2gse.ST_Within(c.location,:BayArea)=1
- There must be a spatial column used in the predicate as the search target, and there must be a spatial index created on that column.

Examples of index exploitation

Table 4 shows the correct and incorrect ways of creating spatial queries to exploit the spatial index.

Table 4. Rules for index exploitation

Spatial query	Rule violated
SELECT * FROM customers c WHERE db2gse.ST_Within(c.location,:BayArea)=1	No condition is violated in this example.
SELECT * FROM customers c WHERE db2gse.ST_Distance(c.location,:SanJose)<10	No condition is violated in this example.
SELECT * FROM customers c WHERE db2gse.ST_Length(c.location)>10	The predicate must be used in the WHERE clause. (ST_Length is a spatial function, but <i>not</i> a predicate.)

Table 4. Rules for index exploitation (continued)

Spatial query	Rule violated
<pre>SELECT * FROM customers c WHERE 1=db2gse.ST_Within(c.location,:BayArea)</pre>	<p>The predicate must be on the left-hand side of the comparison.</p>
<pre>SELECT * FROM customers c WHERE db2gse.ST_Within(c.location,:BayArea)=2</pre>	<p>Equality comparisons must use the integer constant 1.</p>
<pre>SELECT * FROM customers c WHERE db2gse.ST_Within(:SanJose,:BayArea)=1</pre>	<p>There must be a spatial column used in the predicate as the search target, and there must be a spatial index created on that column. (SanJose and BayArea are not spatial columns and therefore, cannot have a spatial index associated with them.)</p>

Chapter 8. Writing applications for DB2 Spatial Extender

This chapter explains how to use the DB2 Spatial Extender sample program to write applications to work with and customize spatial information. The following topics are included:

- Using the sample program
- The sample program steps

Using the sample program

The DB2 Spatial Extender sample program makes application programming easier. With the sample program, you can:

- Automate routine spatial procedures
- Cut and paste sample code into your own applications
- Understand the steps typically required to create and maintain a spatially enabled database

Use the sample program to code complex tasks for DB2 Spatial Extender, for example to write an application that uses the database interface to call DB2 Spatial Extender stored procedures. From the sample program, you can copy and customize your applications. If you are unfamiliar with the programming steps for DB2 Spatial Extender, you can run the sample program, which shows you each step in detail. First, however, you must create the sample program. You do this with the sample makefile. For instructions on creating and running the sample program, see “Verifying the installation” on page 18.

The sample program steps

Table 5 on page 54 shows the sample program steps, the associated stored procedures, and a description of each step. The C functions for invoking the stored procedures is displayed in the Action column of Table 5 on page 54 and is enclosed in parentheses. For more information about the stored procedures, see “Chapter 9. Stored procedures” on page 63. The sample program is based on scenarios that are introduced in “Scenario: An insurance company updates its GIS” on page 11.

Table 5. DB2 Spatial Extender sample program

Sample program steps	Action	Description
Enable/disable spatial database	<ol style="list-style-type: none"> 1. Enable the spatial database (gseEnableDB) 2. Disable the spatial database (gseDisableDB) 3. Enable the spatial database (gseEnableDB) 	<ol style="list-style-type: none"> 1. This is the first step needed to use DB2 Spatial Extender. A database that has been enabled for spatial operations has a set of spatial types, a set of spatial functions, a set of spatial predicates, a new index type, and a set of administration tables and views. 2. This step is usually performed when you have enabled spatial capabilities for the wrong database. When you disable a spatial database, you remove a set of spatial types, a set of spatial functions, a set of spatial predicates, a new index type, and a set of administration tables and views. Note: The disable database will fail if there are objects created that depend on the objects created by the enable database procedure. For example, creating a table with a spatial column of the type ST_Point will cause the disable database to fail. This occurs because the table depends on the type ST_Point which is intended to be dropped by the disable database procedure. 3. Same as 1.

Table 5. DB2 Spatial Extender sample program (continued)

Sample program steps	Action	Description
Register spatial reference systems	<ol style="list-style-type: none"> 1. Register the spatial reference system for the LOCATION column of the CUSTOMERS table (gseEnableSref) 2. Register the spatial reference system for the LOCATION column of OFFICES table (gseEnableSref) 3. Unregister the spatial reference system for the LOCATION column of OFFICES table (gseDisableSref) 4. Re-register the spatial reference system for the ZONE columns of the OFFICES table (gseEnableSref) 	<ol style="list-style-type: none"> 1. This step defines a new spatial reference system (SRS) intended to be used to interpret the spatial data of the CUSTOMERS table. A spatial reference system includes geometry data in a form that can be stored in a column of a spatially enabled database. After the SRS is registered to a specific layer, the coordinates applicable to that layer can be stored in the associated CUSTOMERS table column. 2. This step defines a new spatial reference system (SRS) intended to be used to interpret the spatial data of the OFFICES layer. Each table layer must have an SRS defined to it. The OFFICES table layers might require a different associated SRS than the CUSTOMERS table layer. 3. This step is performed if you specify the wrong SRS parameters to the layer or spatial column. When you unregister an SRS for the OFFICES table layer, you remove the definition with its associated parameters. 4. This step defines a new spatial reference system (SRS) intended to be used to interpret the spatial data of the OFFICES layer.

Table 5. DB2 Spatial Extender sample program (continued)

Sample program steps	Action	Description
Create the spatial tables	<ol style="list-style-type: none"> 1. Alter the CUSTOMERS table by adding the LOCATION column (gseSetupTables) 2. Create the OFFICES table (gseSetupTables) 	<ol style="list-style-type: none"> 1. The CUSTOMERS table represents business data that has been stored in the database for several years. The ALTER TABLE statement adds a new column (LOCATION) of type ST_Point. This column will be populated by geocoding the address columns in a subsequent step. 2. The OFFICES table represents, among other data, the sales zone for each office of an insurance company. The entire table will be populated with the attribute data from a non-DB2 database in a subsequent step. This step involves importing attribute data into the OFFICES table from a SHAPE file.
Register the spatial layers	<ol style="list-style-type: none"> 1. Register the LOCATION column in the CUSTOMERS table as a layer (gseRegisterLayer) 2. Register the ZONE column of the OFFICES table as a layer (gseRegisterLayer) 	<p>These steps register the LOCATION and ZONE columns as layers to DB2 Spatial Extender. Before a spatial column can be populated or accessed by the DB2 Spatial Extender utilities (for example, the geocoder), you need to register it as a layer.</p>
Populate the spatial layers	<ol style="list-style-type: none"> 1. Geocode the addresses data for the LOCATION column of CUSTOMERS table (gseRunGC) 2. Load the OFFICES table using append mode (gseImportShape) 3. Load the HAZARD_ZONE table using create mode (gseImportShape) 	<ol style="list-style-type: none"> 1. This step performs batch geocoding by invoking the geocoder utility. Batch geocoding is usually performed when a significant portion of the table needs to be geocoded or re-geocoded. 2. This step loads the OFFICES table with spatial data existing in the form of a SHAPE file. Because the OFFICES table exists and the OFFICES/ZONE layer is registered, the load utility will append the new records to an existing table. 3. This step loads the HAZARD_ZONE layer with spatial data existing in the form of a SHAPE file. Because the table and layer do not exist, the load utility will create the table and register the layer before the data is loaded.

Table 5. DB2 Spatial Extender sample program (continued)

Sample program steps	Action	Description
Enable spatial indexes	<ol style="list-style-type: none"> 1. Enable the spatial index for the LOCATION column of the CUSTOMERS table (gseEnableIdx) 2. Enable the spatial index for the ZONE column of the OFFICES table (gseEnableIdx) 3. Enable the spatial index for the LOCATION column of the OFFICES table (gseEnableIdx) 4. Enable the spatial index for the BOUNDARY column of the HAZARD_ZONE table (gseEnableIdx) 	These steps enable the spatial index for the CUSTOMERS, OFFICES and HAZARD_ZONE table.
Enable automatic geocoding	<ol style="list-style-type: none"> 1. Enable automatic geocoding for the LOCATION and ADDRESS columns of CUSTOMERS table (gseEnableAutoGC) 	This step turns on the automatic invocation of the geocoder. Using automatic geocoding causes the LOCATION and ADDRESS columns of the CUSTOMERS table to be synchronized with each other for subsequent insert and update operations.
Insert/update the CUSTOMERS table	<ol style="list-style-type: none"> 1. Insert some records with a different street (gseInsDelUpd) 2. Update some records with a new address (gseInsDelUpd) 	These steps demonstrate an insert and update on the LOCATION column of the CUSTOMERS table. Once the automatic geocoding is enabled, information from the ADDRESS column is automatically geocoded when it is inserted or updated in the LOCATION column. This process was enabled in the previous step.
Disable the automatic geocoding	<ol style="list-style-type: none"> 1. Disable the automatic geocoding for the CUSTOMERS layer (gseDisableAutoGC) 2. Disable the spatial index for the CUSTOMERS layer (gseDisableIdxCustomersLayer) 	These steps disable the automatic invocation of the geocoder and the spatial index in preparation for the next step (the next step involves the re-geocoding of the entire CUSTOMERS table). If you are loading a large amount of geodata, it is recommend that you disable the spatial index before you load the data and then enable it after the load is complete.

Table 5. DB2 Spatial Extender sample program (continued)

Sample program steps	Action	Description
Re-geocode the CUSTOMERS table	<ol style="list-style-type: none"> 1. Geocode the CUSTOMERS layer again with a lower precision level – 90% instead of 100% (gseRunGC) 2. Re-enable the spatial index for the CUSTOMERS layer (gseEnableIdx) 3. Re-enable the automatic geocoding with a lower precision level – 90% instead of 100% (gseEnableAutoGC) 	<p>These steps runs the geocoder in batch mode again, re-enable the automatic geocoding with a new precision level, and re-enable the spatial index and the automatic geocoding. This action is recommended when the spatial administrator notices a high failure rate in the geocoding process. If the precision level is set to 100%, it may fail to geocode an address because it cannot find a matching address in the reference data. By reducing the precision level, the geocoder has a better chance of finding matching data. After the table is re-geocoded in batch mode, both the automatic geocoding and the spatial index are enabled again to facilitate the incremental maintenance of the spatial index and the spatial column for subsequent inserts and updates.</p>
Create a view and register its spatial columns as view layers	<ol style="list-style-type: none"> 1. Create a view, HIGHRISK_CUSTOMERS, based on the join of the CUSTOMERS table and the HAZARD_ZONE table (gseCreateView) 2. Register the view's spatial columns as view layers (gseRegisterLayer) 	<p>These steps create a view and register its spatial columns as view layers.</p>

Table 5. DB2 Spatial Extender sample program (continued)

Sample program steps	Action	Description
Perform spatial analysis	<ol style="list-style-type: none"> 1. Find the average customer distance from each office (ST_Within, ST_Distance) 2. Find the average customer income and premium for each office (ST_Within) 3. Find customers who are not covered by any existing office (ST_Within) 4. Find the number of hazards zones that each office zone overlaps (ST_Overlaps) 5. Find the nearest office from a particular customer location assuming that the office is located in the centroid of the office zone (ST_Distance, ST_Centroid) 6. Find the customers whose location is close to the boundary of a particular hazard zone (ST_Buffer, ST_Overlaps) 7. Find those high risk customers who are covered by a particular office 	<p>These steps perform spatial analysis using the spatial predicates and functions in DB2 SQL language. The DB2 query optimizer exploits the spatial index on the spatial columns to improve the query performance whenever possible.</p>
<p>(All steps utilize gseRunSpatialQueries)</p>		
Export spatial layers into files	Export the highRiskCustomers layer (gseExportShape)	<p>The step shows an example of exporting the results of your query to a SHAPE file. Exporting query results to another file format allows the information to be used by a third party tool (for example, ESRI ArcInfo).</p>

Part 2. Reference material

Chapter 9. Stored procedures

This chapter documents the stored procedures that allow you to build a geographic information system with DB2 Spatial Extender. When you enable and use DB2 Spatial Extender from the Control Center, you invoke these stored procedures implicitly. For example, when you click **OK** from a DB2 Spatial Extender window, DB2 calls the stored procedure or stored procedures associated with that window for you. Alternatively, you can invoke the stored procedures explicitly in an application program. It is advisable to include the header file, `db2gse.h`, in such a program. This file contains the macro definitions for the constants that you assign to the stored procedures' parameters. On AIX, it is stored in the `$DB2INSTANCE/sqlllib/include/` directory. On Windows NT, it is stored in the `%DB2PATH%\include\` directory.

Attention:

All character string constants for stored procedures' input parameters are case-sensitive. To find out which parameters require these constants, see the tables in this chapter.

Before you can invoke a stored procedure, either implicitly or explicitly, you must be connected to the database in which DB2 Spatial Extender is installed. The first stored procedure that you use is `db2gse.gse_enable_db`. It enables the database for spatial operations. You can use the other stored procedures only after the database has been enabled.

The implementations of the stored procedures are archived in the `db2gse` library on the DB2 Spatial Extender server.

You can use the following lists to look up the stored procedures either by their names or by the tasks that they carry out. The first list presents the names:

- “`db2gse.gse_disable_autogc`” on page 66
- “`db2gse.gse_disable_db`” on page 68
- “`db2gse.gse_disable_sref`” on page 69
- “`db2gse.gse_enable_autogc`” on page 70
- “`db2gse.gse_enable_db`” on page 73
- “`db2gse.gse_enable_idx`” on page 74
- “`db2gse.gse_enable_sref`” on page 76

- “db2gse.gse_export_shape” on page 78
- “db2gse.gse_import_sde” on page 80
- “db2gse.gse_import_shape” on page 82
- “db2gse.gse_register_gc” on page 84
- “db2gse.gse_register_layer” on page 86
- “db2gse.gse_run_gc” on page 92
- “db2gse.gse_unregist_gc” on page 94
- “db2gse.gse_unregist_layer” on page 95

The next list presents the tasks that the stored procedures carry out.

- Creating an index for a spatial column (see “db2gse.gse_enable_idx” on page 74).
- Creating a spatial reference system (see “db2gse.gse_enable_sref” on page 76).
- Disabling a geocoder so that it cannot automatically keep spatial columns synchronized with their corresponding attribute columns (see “db2gse.gse_disable_autogc” on page 66).
- Disabling support for spatial operations in a database (see “db2gse.gse_disable_db” on page 68).
- Dropping a spatial reference system (see “db2gse.gse_disable_sref” on page 69).
- Enabling a database to support spatial operations (see “db2gse.gse_enable_db” on page 73).
- Enabling a geocoder to automatically keep spatial columns synchronized with their corresponding attribute columns (see “db2gse.gse_enable_autogc” on page 70).
- Exporting a layer and its associated table to a shape file (see “db2gse.gse_export_shape” on page 78).
- Importing a layer and its associated table from an ESRI_SDE transfer file (see “db2gse.gse_import_sde” on page 80).
- Importing a layer and its associated table from a shape file (see “db2gse.gse_import_shape” on page 82).
- Registering a geocoder other than the default geocoder (see “db2gse.gse_register_gc” on page 84).
- Registering a spatial column as a layer (see “db2gse.gse_register_layer” on page 86).
- Running a geocoder in batch mode (see “db2gse.gse_unregist_gc” on page 94).
- Unregistering a geocoder other than the default geocoder (see “db2gse.gse_unregist_layer” on page 95).
- Unregistering a layer (see “db2gse.gse_unregist_layer” on page 95).

For information about the sequences in which you can perform these tasks, see “Chapter 1. About DB2 Spatial Extender” on page 3 and “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

db2gse.gse_disable_autogc

Use this stored procedure to drop or temporarily disable triggers that keep a spatial column synchronized with its associated attribute column or columns. For example, it is advisable to disable the triggers while you geocode the values in the attribute column or columns in batch mode. For more on this, see “About geocoding” on page 37.

For an example of the code for invoking this stored procedure, see the C function `gseDisableAutoGc` in the sample program. For information about this program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Authorization

The user ID under which this stored procedure is invoked must have authorization in the form of an authority, privilege, or set of privileges; specifically:

- SYSADM or DBADM authority on the database the contains the table on which the triggers that are being dropped or temporarily disabled are defined.
- The CONTROL privilege on this table.
- The ALTER, SELECT, and UPDATE privileges on this table.

Input parameters

Table 6. Input parameters for the db2gse.gse_disable_autogc stored procedure.

Name	Data type	Description
operMode	SMALLINT	Indicates whether the triggers are to be dropped or temporarily disabled. This parameter is not nullable. Comment: To drop triggers, use the <code>GSE_AUTOGC_DROP</code> macro. To temporarily disable them, use the <code>GSE_AUTOGC_INVALIDATE</code> macro. To find out what values are associated with these macros, consult the <code>db2gse.h</code> file. On AIX, this file is stored in the <code>\$DB2INSTANCE/sqlib/include/</code> directory. On Windows NT, it is stored in the <code>%DB2PATH%\include\</code> directory.

Table 6. Input parameters for the `db2gse.gse_disable_autogc` stored procedure. (continued)

Name	Data type	Description
layerSchema	VARCHAR(30)	Name of the schema to which the table or view specified in the layerTable parameter belongs. This parameter is nullable. Comment: If you do not supply a value for the layerSchema parameter, it will default to the user ID under which the <code>db2gse.gse_disable_autogc</code> stored procedure is invoked.
layerTable	VARCHAR(128)	Name of the table on which the triggers you want dropped or temporarily disabled are defined. This parameter is not nullable.
layerColumn	VARCHAR(128)	Name of the spatially-enabled column that is maintained by the triggers that you want dropped or temporarily disabled. This parameter is not nullable.

Output parameters

Table 7. Output parameters for the `db2gse.gse_disable_autogc` stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

db2gse.gse_disable_db

Use this stored procedure to remove resources that allow DB2 Spatial Extender to store spatial data and to support operations performed on this data.

The purpose of this stored procedure is to help you resolve problems or issues that arise after you enable your database for spatial operations, but *before* you add any spatial table columns or data to it. For example, if, after you enable a database for spatial operations, it is decided to use DB2 Spatial Extender for another database instead. As long as you have not defined any spatial columns or imported any spatial data, you could invoke this stored procedure to remove all spatial resources from the first database.

For an example of the code for invoking this stored procedure, see the C function `gseDisableDB` in the sample program. For information about this program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Authorization

The user ID under which this stored procedure is invoked must have either SYSADM or DBADM authority on the database from which DB2 Spatial Extender resources are to be removed.

Output parameters

Table 8. Output parameters for the db2gse.gse_disable_db stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

db2gse.gse_disable_sref

Use this stored procedure to drop a spatial reference system. When this stored procedure is processed, information about the spatial reference system is removed from the DB2GSE.SPATIAL_REF_SYS catalog view. For information about this view, see “DB2GSE.SPATIAL_REF_SYS” on page 107.

For an example of the code for invoking this stored procedure, see the C function gseDisableSref in the sample program. For information about this program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Authorization

None required.

Input parameter

Table 9. Input parameter for the db2gse.gse_disable_sref stored procedure.

Name	Data type	Description
srId	INTEGER	Numeric identifier of the spatial reference system that is to be dropped.
This parameter is not nullable.		

Output parameters

Table 10. Output parameters for the db2gse.gse_disable_sref stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

Restriction

Before you can drop a spatial reference system, you must unregister any layers that use it. If such layers remain unregistered, the request to drop the spatial reference system will be rejected.

db2gse.gse_enable_autogc

Use this stored procedure to:

- Create triggers that will keep a spatial column synchronized with its associated attribute column or columns. Each time values are inserted into, or updated in, the attribute column or columns, a trigger will call a registered geocoder to geocode the inserted or updated values and place the resulting data in the spatial column.
- Reactivate the triggers after they have been temporarily disabled.
- Establish which function will be used to geocode the inserted and updated values.

For an example of the code for invoking this stored procedure, see the C function `gseEnableAutoGC` in the sample program. For information about this program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Authorization

The user ID under which this stored procedure is invoked must have authorization in the form of an authority, privilege, or set of privileges; specifically:

- SYSADM or DBADM authority on the database that contains the table on which the triggers created by this stored procedure are defined.
- The CONTROL privilege on this table.
- The ALTER, SELECT, and UPDATE privileges on this table.

Input parameters

Table 11. Input parameters for the db2gse.gse_enable_autogc stored procedure.

Name	Data type	Description
operMode	SMALLINT	Value that indicates whether the triggers that initiate the geocoding are to be created for the first time or to be reactivated after being temporarily disabled. This parameter is not nullable. Comment: To create the triggers, use the <code>GSE_AUTOGC_CREATE</code> macro. To reactivate them, use the <code>GSE_AUTOGC_RECREATE</code> macro. To find out what values are associated with these macros, consult the <code>db2gse.h</code> file. On AIX, this file is stored in the <code>\$DB2INSTANCE/sqlib/include/</code> directory. On Windows NT, it is stored in the <code>%DB2PATH%\include\</code> directory.

Table 11. Input parameters for the `db2gse.gse_enable_autogc` stored procedure. (continued)

Name	Data type	Description
<code>layerSchema</code>	<code>VARCHAR(30)</code>	<p>Name of the schema to which the table specified in the <code>layerTable</code> parameter belongs.</p> <p>This parameter is nullable.</p> <p>Comment: If you do not supply a value for the <code>layerSchema</code> parameter, it will default to the user ID under which the <code>db2gse.gse_enable_autogc</code> stored procedure is invoked.</p>
<code>layerTable</code>	<code>VARCHAR(128)</code>	<p>Name of the table that the triggers created or reactivated by this stored procedure are to operate on.</p> <p>This parameter is not nullable.</p>
<code>layerColumn</code>	<code>VARCHAR(128)</code>	<p>Name of the spatial column that is to be maintained by the triggers that this stored procedure creates or reactivates.</p> <p>This parameter is not nullable.</p>
<code>gcId</code>	<code>INTEGER</code>	<p>Identifier of the geocoder that will be invoked by the insert and update triggers that this stored procedure creates or reactivates.</p> <p>This parameter is not nullable if the <code>operMode</code> parameter is set to <code>GSE_AUTOGC_CREATE</code>. It is nullable if <code>operMode</code> is set to <code>GSE_AUTOGC_RECREATE</code>.</p>
<code>precisionLevel</code>	<code>INTEGER</code>	<p>The degree to which source data must match corresponding reference data in order for the geocoder to process the source data successfully.</p> <p>This parameter is not nullable if the <code>operMode</code> parameter is set to <code>GSE_AUTOGC_CREATE</code>. It is nullable if <code>operMode</code> is set to <code>GSE_AUTOGC_RECREATE</code>.</p> <p>Comment: The precision level can range from 1 to 100 percent.</p>

Table 11. Input parameters for the db2gse.gse_enable_autogc stored procedure. (continued)

Name	Data type	Description
vendorSpecific	VARCHAR(256)	Technical information provided by the vendor; for example, the path and name of a file that the vendor uses to set parameters. This parameter is not nullable if the operMode parameter is set to GSE_AUTOGC_CREATE. It is nullable if operMode is set to GSE_AUTOGC_RECREATE.

Output parameters

Table 12. Output parameters for the db2gse.gse_enable_autogc stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

Restrictions

- The layerColumn parameter must reference a column that has been registered as a table layer.
- If the operMode parameter is set to GSE_AUTOGC_CREATE, you must assign an identifier of a registered geocoder to the gcId parameter.

db2gse.gse_enable_db

Use this stored procedure to supply a database with the resources that it needs to store spatial data and to support operations. These resources include spatial data types, a spatial index type, catalog tables and views, supplied functions, and other stored procedures.

For an example of the code for invoking this stored procedure, see the C function `gseEnableDB` in the sample program. For information about this program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Authorization

The user ID under which the stored procedure is invoked must have either SYSADM or DBADM authority on the database that is being enabled.

Output parameters

Table 13. Output parameters for the db2gse.gse_enable_db stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

db2gse.gse_enable_idx

Use this stored procedure to create an index for a spatial column.

For an example of the code for invoking this stored procedure, see the C function `gseEnableIdx` in the sample program. For information about this program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Authorization

The user ID under which this stored procedure is invoked must hold one of the following authorities or privileges:

- SYSADM or DBADM authority on the database that contains the table for which the enabled index is to be used.
- The CONTROL or INDEX privilege on this table.

Input parameters

Table 14. Input parameters for the db2gse.gse_enable_idx stored procedure.

Name	Data type	Description
layerSchema	VARCHAR(30)	Name of the schema to which the table specified in the layerTable parameter belongs. This parameter is nullable. Comment: If you do not supply a value for the layerSchema parameter, it will default to the user ID under which the db2gse.gse_enable_idx stored procedure is invoked.
layerTable	VARCHAR(128)	Name of the table on which the index that you are creating is to be defined. This parameter is not nullable.
layerColumn	VARCHAR(128)	Name of the spatially enabled column that is to be searched with the aid of the index that you are creating. This parameter is not nullable.
indexName	VARCHAR(128)	Name of the index that is to be created. This parameter is not nullable. Comment: Do not specify a schema name. DB2 Spatial Extender automatically assigns the index to the schema referenced by the layerSchema parameter.

Table 14. Input parameters for the `db2gse.gse_enable_idx` stored procedure. (continued)

Name	Data type	Description
gridSize1	DOUBLE	Number that indicates what the granularity of the finest index grid should be. This parameter is not nullable.
gridSize2	DOUBLE	Number that denotes either (1) that there is to be no second grid for this index or (2) what the granularity of the second grid should be. This parameter is nullable. Comment: If there is to be no second grid, specify 0. If you want a second grid, it must be less granular than the grid denoted by gridSize1.
gridSize3	DOUBLE	Number that denotes either (1) that there is to be no third grid for this index or (2) what the granularity of the third grid should be. This parameter is nullable. Comment: If there is to be no third grid, specify 0. If you want a third grid, it must be less granular than the grid denoted by gridSize2.

Output parameters

Table 15. Output parameters for the `db2gse.gse_enable_idx` stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

db2gse.gse_enable_sref

Use this stored procedure to specify how negative and decimal numbers in a specific coordinate system are to be converted into positive integers, so that DB2 Spatial Extender can store them. Your specifications are collectively called a *spatial reference system*. When this stored procedure is processed, information about the spatial reference system is added to the DB2GSE.SPATIAL_REF_SYS catalog view. For information about this view, see “DB2GSE.SPATIAL_REF_SYS” on page 107.

For an example of the code for invoking this stored procedure, see the C function `gseEnableSref` in the sample program. For information about this program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Authorization

None required.

Input parameters

Table 16. Input parameters for the `db2gse.gse_enable_sref` stored procedure.

Name	Data type	Description
<code>srId</code>	INTEGER	A numeric identifier for the spatial reference system. This parameter is not nullable. Comment: This identifier must be unique within your spatially-enabled database.
<code>srName</code>	VARCHAR(64)	Short description of the spatial reference system. This parameter is not nullable. Comment: This description must be unique within your spatially-enabled database.
<code>falseX</code>	DOUBLE	A number that, when subtracted from a negative X coordinate value, leaves a non-negative number (that is, a positive number or a zero). This parameter is not nullable.
<code>falseY</code>	DOUBLE	A number that, when subtracted from a negative Y coordinate value, leaves a non-negative number (that is, a positive number or a zero). This parameter is not nullable.

Table 16. Input parameters for the *db2gse.gse_enable_sref* stored procedure. (continued)

Name	Data type	Description
xyunits	DOUBLE	A number that, when multiplied by a decimal X coordinate or a decimal Y coordinate, yields an integer that can be stored as a 32-bit data item. This parameter is not nullable.
falsez	DOUBLE	A number that, when subtracted from a negative Z coordinate value, leaves a non-negative number (that is, a positive number or a zero). This parameter is not nullable.
zunits	DOUBLE	A number that, when multiplied by a decimal Z coordinate, yields an integer that can be stored as a 32-bit data item. This parameter is not nullable.
falsem	DOUBLE	A number that, when subtracted from a negative measure, leaves a non-negative number (that is, a positive number or a zero). This parameter is not nullable.
munits	DOUBLE	A number that, when multiplied by a decimal measure, yields an integer that can be stored as a 32-bit data item. This parameter is not nullable.
scId	INTEGER	Numeric identifier of the coordinate system from which the spatial reference system is being derived. To find out what a coordinate system's numeric identifier is, consult the DB2GSE.COORD_REF_SYS catalog view "DB2GSE.COORD_REF_SYS" on page 105. This parameter is not nullable.

Output parameters

Table 17. Output parameters for the *db2gse.gse_enable_sref* stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

db2gse.gse_export_shape

Use this stored procedure to export a layer and its associated table to a shape file, or to create a new shape file and export a layer and its associated table to this new file.

For an example of the code for invoking this stored procedure, see the C function `gseExportShape` in the sample program. For information about this program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Authorization

The user ID under which this stored procedure is invoked must hold the SELECT privilege on the table that is to be exported.

Input parameters

Table 18. Input parameters for the `db2gse.gse_export_shape` stored procedure.

Name	Data type	Description
layerSchema	VARCHAR(30)	Name of the schema to which the table specified in the <code>layerTable</code> parameter belongs. This parameter is nullable. Comment: If you do not supply a value for the <code>layerSchema</code> parameter, it will default to the user ID under which the <code>db2gse.gse_export_shape</code> stored procedure is invoked.
layerTable	VARCHAR(128)	Name of the table that you are exporting. This parameter is not nullable.
layerColumn	VARCHAR(30)	Name of the column that has been registered as the layer that you are exporting. This parameter is not nullable.
fileName	VARCHAR(128)	Name of the shape file to which the specified layer is to be exported. This parameter is not nullable.
whereClause	VARCHAR(1024)	The body of the SQL WHERE clause. It defines a restriction on the set of records to be geocoded. The clause can reference any attribute column in the table that you are exporting. This parameter is nullable.

Output parameters

Table 19. Output parameters for the db2gse.gse_export_shape stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

Restriction

You can export only one layer at a time.

db2gse.gse_import_sde

Use this stored procedure to import an SDE transfer file to a database that has been enabled for spatial operations. The stored procedure can operate in either of two ways:

- If the SDE transfer file is targeted for an existing table that has a registered layer column, DB2 Spatial Extender will load the table with the file's data.
- Otherwise, DB2 Spatial Extender will create a table that has a spatial column, register this column as a layer, and load the layer and the table's other columns with the file's data.

The spatial reference system specified in the SDE transfer file will be compared with the spatial reference systems that are registered to DB2 Spatial Extender. If the specified system matches a registered system, the negative and decimal values in the transfer data will, when loaded, be modified in the way prescribed by the registered system. If the specified system matches none of the registered systems, DB2 Spatial Extender will create a new spatial reference system to prescribe the modifications.

Authorization

When you import data to an existing table, the user ID under which this stored procedure is invoked must hold one of the following authorities or privileges:

- SYSADM or DBADM authority on the database that contains the table to which data is to be imported.
- The CONTROL privilege on this table.

When the table to which you want to import data must be created, the user ID under which this stored procedure is invoked must hold one of the following authorities or privileges:

- SYSADM or DBADM authority on the database that contains the table that is to be created.

Input parameters

Table 20. Input parameters for the db2gse.gse_import_sde stored procedure.

Name	Data type	Description
layerSchema	VARCHAR(30)	Name of the schema to which the table or view specified in the layerTable parameter belongs. This parameter is nullable. Comment: If you do not supply a value for the layerSchema parameter, it will default to the user ID under which the db2gse.gse_import_sde stored procedure is invoked.

Table 20. Input parameters for the db2gse.gse_import_sde stored procedure. (continued)

Name	Data type	Description
layerTable	VARCHAR(128)	Name of the table into which the SDE transfer data is to be loaded. This parameter is not nullable.
layerColumn	VARCHAR(30)	Name of the column that has been registered as the layer into which the SDE transfer file's spatial data is to be loaded. This parameter is not nullable.
fileName	VARCHAR(128)	Name of the SDE transfer file that is to be imported. This parameter is not nullable.
commitScope	INTEGER	Number of records per checkpoint. This parameter is nullable.

Output parameters

Table 21. Output parameters for the db2gse.gse_import_sde stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

db2gse.gse_import_shape

Use this stored procedure to import a shape file to a database that has been enabled for spatial operations. The stored procedure can operate in either of two ways:

- If the shape file is targeted for an existing table that has a registered layer column, DB2 Spatial Extender will load the table with the file's data.
- Otherwise, DB2 Spatial Extender will create a table that has a spatial column, register this column as a layer, and load the layer and the table's other columns with the file's data.

For an example of the code for invoking this stored procedure, see the C function `gseImportShape` in the sample program. For information about this program, see "Chapter 8. Writing applications for DB2 Spatial Extender" on page 53.

Authorization

The user ID under which this stored procedure is invoked must hold one of the following authorities or privileges:

- SYSADM or DBADM authority on the database that contains the table into which imported shape data is to be loaded.
- The CONTROL privilege on this table.

Input parameters

Table 22. Input parameters for the `db2gse.gse_import_shape` stored procedure.

Name	Data type	Description
layerSchema	VARCHAR(30)	Name of the schema to which the table or view specified in the <code>layerTable</code> parameter belongs. This parameter is nullable. Comment: If you do not supply a value for the <code>layerSchema</code> parameter, it will default to the user ID under which the <code>db2gse.gse_import_shape</code> stored procedure is invoked.
layerTable	VARCHAR(128)	Name of the table into which the imported shape file is to be loaded. This parameter is not nullable.
layerColumn	VARCHAR(30)	Name of the column that has been registered as the layer into which shape data is to be loaded. This parameter is not nullable.

Table 22. Input parameters for the `db2gse.gse_import_shape` stored procedure. (continued)

Name	Data type	Description
fileName	VARCHAR(128)	Name of the shape file that is to be imported. This parameter is not nullable.
exceptionFile	VARCHAR(128)	Path and name of the file in which the shapes that could not be imported are stored. This is a new file that will be created when the <code>db2gse.gse_import_shape</code> stored procedure is run. This parameter is not nullable.
srId	INTEGER	Identifier of the spatial reference system to be used for the layer into which shape data is to be loaded. This parameter is nullable. Comment: If this identifier is not specified, the internal transformation will be set to the maximum resolution possible resolution for the shape file.
commitScope	INTEGER	Number of records per checkpoint. This parameter is nullable.

Output parameters

Table 23. Output parameters for the `db2gse.gse_import_shape` stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

db2gse.gse_register_gc

Use this stored procedure to register a geocoder other than the default. To find out whether a geocoder has already been registered, consult the DB2GSE.SPATIAL_GEOCODER catalog view (described in “DB2GSE.SPATIAL_GEOCODER” on page 106).

Authorization

The user ID under which this stored procedure is invoked must hold either SYSADM or DBADM authority on the database that contains the geocoder that this stored procedure registers.

Input parameters

Table 24. Input parameters for the db2gse.gse_register_gc stored procedure.

Name	Data type	Description
gcId	INTEGER	Numeric identifier of the geocoder that you are registering. This parameter is not nullable. Comment: This identifier must be unique within the database.
gcName	VARCHAR(64)	Short description of the geocoder that you are registering. This parameter is not nullable. Comment: This description must be a unique character string within the database.
vendorName	VARCHAR(64)	Name of vendor that supplied the geocoder that you are registering. This parameter is not nullable.
primaryUDF	VARCHAR(256)	Fully-qualified name of the geocoder that you are registering. This parameter is not nullable.
precisionLevel	INTEGER	The degree to which source data must match corresponding reference data in order for the geocoder to process the source data successfully. This parameter is not nullable. Comment: The precision level can range from 1 to 100 percent.

Table 24. Input parameters for the db2gse.gse_register_gc stored procedure. (continued)

Name	Data type	Description
vendorSpecific	VARCHAR(256)	Technical information provided by the vendor; for example, the path and name of a file that the vendor uses to set parameters. This parameter is nullable.
geoArea	VARCHAR(256)	Geographical area to be geocoded. This parameter is nullable.
description	VARCHAR(256)	Remarks provided by the vendor This parameter is nullable.

Output parameters

Table 25. Output parameters for the db2gse.gse_register_gc stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

db2gse.gse_register_layer

Use this stored procedure to register a spatial column as a layer. When this stored procedure is processed, information about the layer being registered is added to the DB2GSE.GEOMETRY_COLUMNS catalog view. For information about this view, see “DB2GSE.GEOMETRY_COLUMNS” on page 106.

For an example of the code for invoking this stored procedure, see the C function `gseRegisterLayer` in the sample program. For information about this program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Authorization

The user ID under which this stored procedure is invoked must hold one of the following authorities or privileges:

- For a table layer:
 - SYSADM or DBADM authority on the database that contains the table to which this layer belongs.
 - The CONTROL or ALTER privilege on this table.
- For a view layer:
 - The SELECT privilege on the base table or tables that contain (1) the address data that is to be geocoded for this layer and (2) the spatial data that results from the geocoding.

Input parameters

Table 26. Input parameters for the db2gse.gse_register_layer stored procedure.

Name	Data type	Description
layerSchema	INTEGER(30)	Name of the schema to which the table or view specified in the layerTable parameter belongs. This parameter is nullable. Comment: If you do not supply a value for the layerSchema parameter, it will default to the user ID under which the db2gse.gse_register_layer stored procedure is invoked.
layerTable	VARCHAR(128)	Name of the table or view that contains the column that is being registered as a layer. This parameter is not nullable.
layerColumn	VARCHAR(128)	Name of the column that is being registered as a layer. If the column does not exist, DB2 Spatial Extender will create it. This parameter is not nullable.

Table 26. Input parameters for the `db2gse.gse_register_layer` stored procedure. (continued)

Name	Data type	Description
<code>layerTypeName</code>	<code>VARCHAR(64)</code>	<p>Data type of the column that is being registered as a layer. You must specify the data type in uppercase; for example: <code>ST_POINT</code></p> <p>This parameter is not nullable if the column is a table column that is to be created when this stored procedure is processed. Otherwise, if the column is an existing column within a table or view, this parameter is nullable.</p>
<code>srId</code>	<code>INTEGER</code>	<p>Identifier of the spatial reference system used for this layer.</p> <p>This parameter is not nullable for a table layer. DB2 Spatial Extender ignores this parameter when you register a view layer.</p>
<code>geoSchema</code>	<code>VARCHAR(30)</code>	<p>Applies when you register a view column as a layer. The <code>geoSchema</code> parameter is the schema of the table that underlies the view to which the column belongs.</p> <p>This parameter is nullable when you register a view column as a layer. DB2 Spatial Extender ignores this parameter when you register a table column as a layer.</p> <p>Comment: If you do not supply a value for the <code>geoSchema</code> parameter, it will default to the value of the <code>layerSchema</code> parameter.</p>
<code>geoTable</code>	<code>VARCHAR(128)</code>	<p>Applies when you register a view column as a layer. The <code>geoTable</code> parameter is the name of the table that underlies the view to which the column belongs.</p> <p>This parameter is not nullable when you register a view column as a layer. DB2 Spatial Extender ignores this parameter when you register a table column as a layer.</p>

Table 26. Input parameters for the `db2gse.gse_register_layer` stored procedure. (continued)

Name	Data type	Description
<code>geoColumn</code>	<code>VARCHAR(128)</code>	<p>Applies when you register a view column as a layer. The <code>geoColumn</code> parameter is the name of the table column that underlies this view column.</p> <p>This parameter is not nullable when you register a view column as a layer. DB2 Spatial Extender ignores this parameter when you register a table column as a layer.</p>
<code>nAttributes</code>	<code>SMALLINT</code>	<p>Number of columns that contain the source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p>
<code>attr1Name</code>	<code>VARCHAR(128)</code>	<p>Name of the first column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>If you intend to use the default geocoder, you need to store street addresses in the <code>attr1Name</code> column.</p>
<code>attr2Name</code>	<code>VARCHAR(128)</code>	<p>Name of the second column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>If you intend to use the default geocoder, you need to store names of cities in the <code>attr2Name</code> column.</p>

Table 26. Input parameters for the `db2gse.gse_register_layer` stored procedure. (continued)

Name	Data type	Description
attr3Name	VARCHAR(128)	<p>Name of the third column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>If you intend to use the default geocoder, you need to store names or abbreviations of states in the attr3Name column.</p>
attr4Name	VARCHAR(128)	<p>Name of the fourth column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>If you intend to use the default geocoder, you need to store zip codes in the attr4Name column.</p>
attr5Name	VARCHAR(128)	<p>Name of the fifth column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>The default geocoder ignores the Attr5Name column.</p>
attr6Name	VARCHAR(128)	<p>Name of the sixth column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>The default geocoder ignores the Attr6Name column.</p>

Table 26. Input parameters for the `db2gse.gse_register_layer` stored procedure. (continued)

Name	Data type	Description
attr7Name	VARCHAR(128)	<p>Name of the seventh column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>The default gecoder ignores the Attr7Name column.</p>
attr8Name	VARCHAR(128)	<p>Name of the eighth column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>The default gecoder ignores the Attr8Name column.</p>
attr9Name	VARCHAR(128)	<p>Name of the ninth column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>The default gecoder ignores the Attr9Name column.</p>
attr10Name	VARCHAR(128)	<p>Name of the tenth column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>The default gecoder ignores the Attr10Name column.</p>

Output parameters

Table 27. Output parameters for the db2gse.gse_register_layer stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

Restrictions

- If you are registering a view column as a layer, it must be based on a table column that has already been registered as a layer.
- No more than ten attribute columns can contain the data that is to be geocoded for the layer that you are registering.

db2gse.gse_run_gc

Use this stored procedure to run a geocoder in batch mode. For information about this task, see “Running the geocoder in batch mode” on page 39.

For an example of the code for invoking this stored procedure, see the C function gseRunGC in the sample program. For information about this program, see “Chapter 8. Writing applications for DB2 Spatial Extender” on page 53.

Authorization

The user ID under which this stored procedure is invoked must hold one of the following authorities or privileges:

- SYSADM or DBADM authority on the database that contains the table on which the specified geocoder is to operate.
- The CONTROL or UPDATE privilege on this table.

Input parameters

Table 28. Input parameters for the db2gse.gse_run_gc stored procedure.

Name	Data type	Description
layerSchema	VARCHAR(30)	Name of the schema to which the table or view specified in the layerTable parameter belongs. This parameter is nullable. Comment: If you do not supply a value for the layerSchema parameter, it will default to the user ID under which the db2gse.gse_run_gc is invoked.
layerTable	VARCHAR(128)	Name of the table that contains the column into which the geocoded data is to be inserted. This parameter is not nullable.
layerColumn	VARCHAR(128)	Name of the column into which the geocoded data is to be inserted. This parameter is not nullable.
gcId	INTEGER	Identifier of the geocoder that you want to run. This parameter is nullable. To find out the identifiers of registered geocoders, consult the DB2GSE.SPATIAL_GEOCODER catalog view.

Table 28. Input parameters for the `db2gse.gse_run_gc` stored procedure. (continued)

Name	Data type	Description
precisionLevel	INTEGER	The degree to which source data must match corresponding reference data in order for the geocoder to process the source data successfully. This parameter is nullable. Comment: The precision level can range from 1 to 100 percent.
vendorSpecific	VARCHAR(256)	Technical information provided by the vendor; for example, the path and name of a file that the vendor uses to set parameters. This parameter is nullable.
whereClause	VARCHAR(256)	The body of the WHERE clause. It defines a restriction on the set of records to be geocoded. The clause can reference any attribute column in the table that the geocoder is to operate on. This parameter is nullable.
commitScope	INTEGER	Number of records per checkpoint. This parameter is nullable.

Output parameters

Table 29. Output parameters for the `db2gse.gse_run_gc` stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

db2gse.gse_unregist_gc

Use this stored procedure to unregister a geocoder other than the default geocoder.

To find information about the geocoder that you want to unregister, consult the DB2GSE.SPATIAL_GEOCODER catalog view; see “DB2GSE.SPATIAL_GEOCODER” on page 106.

Authorization

The user ID under which this stored procedure is invoked must hold either SYSADM or DBADM authority on the database that contains the geocoder that is to be unregistered.

Input parameter

Table 30. Input parameter for the db2gse.gse_unregist_gc stored procedure.

Name	Data type	Description
gcId	INTEGER	The identifier of the geocoder that is to be unregistered.
This parameter is not nullable.		

Output parameters

Table 31. Output parameters for the db2gse.gse_unregist_gc stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

db2gse.gse_unregist_layer

Use this stored procedure to unregister a layer. The stored procedure does this by:

- Removing the definition of the layer from DB2 Spatial Extender catalog tables.
- Deleting the check constraint that DB2 Spatial Extender placed on this layer's base table to ensure that the layer's spatial data conforms to the requirements of the layer's spatial reference system.
- Dropping the triggers that are used to update the spatial column whenever address data is added, changed, or removed.

When the stored procedure is processed, information about the layer is removed from the DB2GSE.GEOMETRY_COLUMNS meta view. For information about this view, see "DB2GSE.GEOMETRY_COLUMNS" on page 106.

Authorization

The user ID under which this stored procedure is invoked must hold one of the following authorities or privileges:

- For a table layer:
 - SYSADM or DBADM authority on the database that contains this layer's base table.
 - The CONTROL or ALTER privilege on this table.
- For a view layer:
 - The SELECT privilege on the base table or tables that contain (1) the address data that is geocoded for this layer and (2) the spatial data that results from the geocoding.

Input parameters

Table 32. Input parameters for the db2gse.gse_unregist_layer stored procedure.

Name	Data type	Description
layerSchema	VARCHAR(30)	Name of the schema to which the table specified in the layerTable parameter belongs. This parameter is nullable. Comment: If you do not supply a value for the layerSchema parameter, it will default to the user ID under which the db2gse.gse_unregister_layer stored procedure is invoked.
layerTable	VARCHAR(128)	Name of the table that contains the column specified in the layerColumn parameter. This parameter is not nullable.

Table 32. Input parameters for the db2gse.gse_unregister_layer stored procedure. (continued)

Name	Data type	Description
layerColumn	VARCHAR(128)	Name of the spatial column that has been defined as the layer that you want to unregister. This parameter is not nullable.

Output parameters

Table 33. Output parameters for the db2gse.gse_unregister_layer stored procedure.

Name	Data type	Description
msgCode	INTEGER	Code associated with the messages that the caller of this stored procedure can return.
Reserved	VARCHAR(1024)	Complete error message, as constructed at the DB2 Spatial Extender server.

Restriction

If a view column that has been defined as a view layer is based on a table column that has been defined as a table layer, you cannot unregister this table layer until you unregister the view layer.

Chapter 10. Messages

This chapter documents messages that DB2 Spatial Extender returns to users. Each message has an identifier. Identifiers that end in the letter E are for error messages; those that end in W are for warnings; and those that end in I are for general information.

DBA7200E More than 10 columns are selected as input to a geocoder

Explanation: Up to 10 columns can be selected as input to a geocoder.

User Reponse: Move column names from the Selected columns box to the Available columns box until the Selected Columns box lists ten names or fewer.

DBA7201E The database is not enabled for Spatial Extender operation.

Explanation: The database must be enabled for Spatial Extender before you can use Spatial Extender.

User Reponse: Right-click the database and select Spatial Extender —> Enable from the menus.

GSE0000I The operation is completed successfully.

GSE0001E Spatial Extender could not perform the requested operation ("**<operation-name>**") under user ID "**<user-id>**".

Explanation: You requested this operation under a user ID that does not hold the privilege or authority to perform the operation.

User Reponse: Consult the documentation to find out what the proper authorization is or obtain it from a Spatial Extender administrator.

GSE0002E "**<value>**" is not a valid value for the "**<argument-name>**" argument.

Explanation: The value that you entered was incorrect or misspelled.

User Reponse: Consult the documentation or a Spatial Extender administrator to find out what value or range of values you need to specify.

GSE0003E Spatial Extender could not perform the requested operation because argument "**<argument-name>**" was not specified.

Explanation: You did not specify an argument that is required for this operation.

User Reponse: Specify argument "**<argument-name>**" with the value that you want; then request the operation again.

GSE0004W The argument "**<argument-name>**" was not evaluated.

Explanation: The operation you requested does not use argument "**<argument-name>**".

User Reponse: None required.

GSE0005E Spatial Extender could not process your request to create an object named "**<object-name>**".

Explanation: Either object "**<object-name>**" already exists, or you do not have the proper permission to create it. It could be a table,

column, trigger, index, file, or other kind of object.

User Reponse: If “<object-name>” is the object that you want, do nothing. Otherwise, specify the name correctly and verify that you have the right permission to create the object.

GSE0006E Spatial Extender could not perform the requested operation on an enabled or registered object named “<object-name>”.

Explanation: Object “<object-name>” is already enabled or registered, or it already exists. It could be a layer, index, spatial reference system, coordinate system, geocoder, or other kind of object.

User Reponse: Make sure that object “<object-name>” exists and resubmit your request.

GSE0007E Spatial Extender could not perform the requested operation on “<object-name>”, an object that has not yet been enabled or registered.

Explanation: Object “<object-name>” has not been enabled or registered. It could be a layer, index, spatial reference system, spatial coordinate system, geocoder, or other kind of object.

User Reponse: Enable or register object “<object-name>”. Then resubmit your request.

GSE0008E An unexpected SQL error (“<sql-error-message>”) has occurred.

User Reponse: Look up detailed message associated with SQLCODE in the SQL error message “<sql-error-message>”. If necessary, contact your IBM service representative.

GSE0009E The requested operation could not be performed on an object named “<object-name>” that already exists.

Explanation: “<object-name>” already exists in the database or the operating system. It could be a file, table, view, column, index, trigger, or other kind of object.

User Reponse: Make sure that you specify the object correctly when you try to access it. If necessary, delete the object.

GSE0010E The requested operation could not be performed on an object named “<object-name>” that might not exist.

Explanation: “<object-name>” does not exist in the database or in the operating system. It could be a file, table, view, column, index, trigger, file, or other kind of object.

User Reponse: Make sure that you have the right permission to access the object. If you have this permission and the object does not exist, then you need to create it.

GSE0011E Spatial Extender could not disable or unregister object “<object-name>”.

Explanation: “<object-name>” is dependent on another object. “<object-name>” could be a spatial reference system, layer, geocoder, or other kind of object.

User Reponse: Consult the documentation to find what kinds of objects “<object-name>” can be dependent on. Then remove the specific object that “<object-name>” is dependent on.

GSE0012E Spatial Extender could not process your request because the fully qualified spatial column "`<layer-schema.layer-name.layer-column>`" is not registered as a table layer.

Explanation: The fully qualified spatial column "`<layer-schema.layer-name.layer-column>`" must be registered as a table layer before you can perform certain operations associated with it (for example, enabling its index, enabling a geocoder to populate it in batch mode or to update it automatically).

User Reponse: Make sure that the fully qualified spatial column "`<layer-schema.layer-name.layer-column>`" is registered as a table layer by checking the DB2GSE.GEOMETRY_COLUMNS view in the Spatial Extender catalog. Also make sure that the table that contains this column also includes valid corresponding attribute columns.

GSE0013E The database is not enabled for spatial operations.

Explanation: The database is not enabled for spatial operations. Therefore, the Spatial Extender catalog does not exist.

User Reponse: Enable the database for spatial operations.

GSE0014E The database has already been enabled for spatial operations.

Explanation: The database has already been enabled for spatial operations.

User Reponse: Verify that the database has been enabled as you expected. If necessary, disable the database.

GSE0498E The following error occurred: "`<error-message>`".

GSE0499W Spatial Extender issued the following warning: "`<warning-message>`".

GSE0500E The operation mode that you specified ("`<operation-mode>`") is invalid.

Explanation: The specified mode is not supported by the operation that you requested.

User Reponse: Consult the documentation to find out what modes are supported by the operation.

GSE1001E Spatial Extender was unable to register a view layer that is named "`<schema-name.view-name.column-name>`" and that is based on spatial column "`<schema-name.table-name.column-name>`".

Explanation: The spatial column that you specified ("`<schema-name.table-name.column-name>`") has not been registered as a table layer.

User Reponse: Register column "`<schema-name.table-name.column-name>`" as a table layer.

GSE1002E Spatial Extender was unable to register a view layer that is named "`<schema-name.view-name.column-name>`" and that is based on table "`<schema-name.table-name>`".

Explanation: The table that you specified ("`<schema-name.table-name>`") does not underlie view "`<schema-name.view-name.column-name>`", either directly or indirectly.

User Reponse: Find out what the base table for view "`<schema-name.view-name.column-name>`" is, and specify this table.

GSE1003E Spatial Extender was unable to access a column named "`<column-name>`" in a table or view named "`<schema-name.object-name>`".

Explanation: Table or view "`<schema-name.object-name>`" does not have a

column named “<column-name>”.

User Reponse: Check the definition of table or view “<schema-name.object-name>” to find out the proper name of the column that you want.

GSE1004E Spatial Extender was unable to register the fully qualified spatial column “<schema-name.table-name.column-name>” as a table layer.

Explanation: Column “<schema-name.table-name.column-name>” does not have a spatial data type, or is not associated with a base table.

User Reponse: Define a spatial data type for column “<schema-name.table-name.column-name>”, or make sure that this column is part of a local base table.

GSE1005E The spatial reference system (“<view-layer-spatial-reference-id>”) that you specified for a view layer differs from the spatial reference system (“<table-layer-spatial-reference-id>”) that is used for this layer’s underlying table layer.

Explanation: A view layer’s spatial reference system must be the same as the underlying table layer’s spatial reference system.

User Reponse: Specify the underlying table layer’s spatial reference system for the view layer.

GSE1006E Because “<spatial-reference-id>” is an invalid spatial reference system ID, Spatial Extender was unable to register the layer that you requested.

Explanation: The spatial reference system that you specified (“<spatial-reference-id>”) has not been enabled or registered.

User Reponse: Enable or register the spatial reference system. Then resubmit your request to register the layer.

GSE1007E An SQL error (SQLSTATE “<sqlstate>”) might have occurred when Spatial Extender tried unsuccessfully to add a spatial column (“<column-name>”) to table “<schema-name.table-name>”.

User Reponse: Look up the message associated with SQLSTATE “<sqlstate>”.

GSE1008E DB2 Spatial Extender was unable to register a view layer “<layer-schema.layer-name.layer-column>” because the spatial data type “<layer-column-type>” of the view layer does not match the spatial data type “<geo-column-type>” of the underlying table layer “<geo-schema.geo-name.geo-column>”.

Explanation: The spatial data type of a view layer “<layer-schema.layer-name.layer-column>” must match the spatial data type of the layer’s underlying table layer “<geo-schema.geo-name.geo-column>”. The inconsistency between these two data types causes ambiguity when spatial data is processed.

User Reponse: Make sure that the spatial data types of the view layer and its underlying table layer are the same.

GSE1020E “<spatial-reference-id>” is an invalid spatial reference system ID.

Explanation: A spatial reference system with an identifier of “<spatial-reference-id>” has not been enabled.

User Reponse: Make sure that the specified spatial reference has been enabled.

GSE1021E Spatial Extender could not enable spatial reference system “<spatial-reference-id>” because the corresponding spatial coordinate system ID “<spatial-coordinate-id>” is invalid.

Explanation: A coordinate system with an identifier of “<spatial-coordinate-id>” is not defined in the Spatial Extender catalog.

User Reponse: Verify the coordinate system identifier “<spatial-coordinate-id>” by checking the DB2GSE.COORD_REF_SYS view in the Spatial Extender catalog.

GSE1030E Because “<schema-name.table-name>” is not a base table, Spatial Extender could not enable a geocoder for it.

Explanation: The object that contains the source data that you want geocoded must be a base table.

User Reponse: Be sure that the columns that contain the source data that you want geocoded are part of a base table.

GSE1031E Spatial Extender could not enable geocoder “<geocoder-id>” to operate automatically in create mode for layer “<layer-schema.layer-name.layer-column>”.

Explanation: Possible explanations are:

- The geocoder is already enabled to update layer “<layer-schema.layer-name.layer-column>” automatically.
- The geocoder has been temporarily invalidated for this layer.
- No columns for source data have been defined for this layer.

User Reponse: If the geocoder has been temporarily invalidated, enable it to operate automatically in “Recreate” mode.

GSE1032E Spatial Extender could not enable geocoder “<geocoder-id>” to operate automatically in recreate mode for layer “<layer-schema.layer-name.layer-column>”.

Explanation: Possible explanations are:

- The geocoder is already enabled to update layer “<layer-schema.layer-name.layer-column>” automatically.
- The geocoder was not previously invalidated for this layer.
- No columns for source data have been defined for this layer.

User Reponse: If the geocoder was previously disabled in drop mode, or if it was never defined for this layer, enable it to operate automatically in “Create” mode.

GSE1033E An SQL error occurred when Spatial Extender tried to add triggers to a table that contains the column for layer “<layer-schema.layer-name.layer-column>” (SQLSTATE “<sqlstate>”).

Explanation: The purpose of the triggers is to maintain data integrity between the attribute columns that the geocoder’s input comes from and the spatial column that its output goes into. The SQL error occurred when DB2 tried to create these triggers.

User Reponse: Look up the message associated with SQLSTATE “<sqlstate>”.

GSE1034E Spatial Extender could not disable geocoder “<geocoder-id>” in drop mode for layer “<layer-schema.layer-name.layer-column>”.

Explanation: Possible explanations are:

- The geocoder has never been enabled to update layer “<layer-schema.layer-name.layer-column>” automatically.
- The geocoder has been disabled in drop mode.

User Reponse: Determine the state of the geocoder before you tried to disable it. For example, was it registered? was it enabled? Then decide whether it needs to be disabled in drop mode. For example, if it was never enabled, there would be no need to disable it at all.

GSE1035E Spatial Extender could not disable geocoder "<geocoder-id>" in invalidate mode for layer "<layer-schema.layer-name.layer-column>".

Explanation: Possible explanations are:

- The geocoder has never been enabled to update layer "<layer-schema.layer-name.layer-column>" automatically.
- The geocoder has been disabled in invalidated mode or in drop mode.

User Reponse: Determine the state of the geocoder before you tried to disable it. For example, was it registered? Was it enabled? Then decide whether it needs to be disabled in invalidate mode. For example, if it was already disabled in invalidate mode, there would be no need to disable it in this mode a second time.

GSE1036E An SQL error occurred when Spatial Extender tried to drop triggers from a table that contains the column for layer "<layer-schema.layer-name.layer-column>" (SQLSTATE "<sqlstate>").

Explanation: The triggers were created to maintain data integrity between the attribute columns that the geocoder's input comes from and the spatial column that its output goes into. The SQL error occurred when DB2 tried to drop these triggers.

User Reponse: Look up the message associated with SQLSTATE "<sqlstate>".

GSE1037E Spatial Extender could not geocode source data for table layer "<layer-schema.layer-name.layer-column>", possibly because an incorrect value "<number-of-attributes>" was assigned to the argument that specifies how many attribute columns are to provide source data for this layer.

Explanation: The number of attribute columns associated with this layer was specified incorrectly, or the name of one or more of these columns was specified incorrectly.

User Reponse: Make sure that this layer is registered with the correct number and names of associated attribute columns, or verify the correctness of input and output data for the geocoder.

GSE1038E An SQL error occurred when Spatial Extender tried to geocode source data for table layer "<layer-schema.layer-name.layer-column>" in batch mode (SQLSTATE "<sqlstate>").

User Reponse:

- Look up the message associated with SQLSTATE "<sqlstate>".
- Make sure that the content and the primaryUDF argument of this layer are defined correctly.

GSE1050E The grid size that you specified ("<grid-size>") is invalid for the first grid level.

Explanation: You specified zero or a negative number as the grid size for the first grid level.

User Reponse: Specify a positive number as the grid size.

GSE1051E The grid size that you specified ("**<grid-size>**") is invalid for the second and third grid levels.

Explanation: You specified a negative number as the grid size for the second or the third grid level.

User Response: Specify zero or a positive number as the grid size.

GSE1052E An SQL error occurred when the Spatial Extender tried to create spatial index "**<index-schema.index-column>**" for a table layer "**<layer-schema.layer-name.layer-column>**" (SQLSTATE "**<sqlstate>**").

User Response:

- Make sure that the spatial index is specified correctly and that the spatial column has no associated index.
- Look up the message that is associated with SQLSTATE "**<sqlstate>**".

GSE1500I Source record "**<record-number>**" was successfully geocoded.

Explanation: A record containing attribute data was successfully geocoded.

GSE1501W Source record "**<record-number>**" was not geocoded.

Explanation: The precision level was too high.

User Response: Geocode with a lower precision level.

GSE1502W Source record "**<record-number>**" was not found.

User Response: Determine whether the record exists in the database.

GSE2001E The specified transfer file ("**<filename>**") is not valid.

User Response: Verify that the specified file is an SDE transfer file, and that the pathname is properly specified.

GSE2002E The supplied SQL WHERE clause ("**<SQL-where-clause>**") is not valid.

User Response: Check the WHERE clause for proper SQL syntax, spelling errors, and invalid column names.

GSE2003E The supplied shape value is not legal.

User Response: Check to be sure that the supplied shape matches the specified type of the spatial column. If the types match, or are compatible, then the form of the geometry is illegal. Check for overlapping polygons, single point arcs, etc.

GSE2004E The transfer file schema is incompatible with the schema of the specified layer.

User Response: Check to be sure that the schema and layer names are properly specified. If the schemas do not match, load the data as a new table and resolve the schema differences.

GSE2005E The transfer file geometry type is incompatible with the geometry type of the specified layer.

User Response: Check to be sure that the schema and layer names are properly specified.

GSE2006E An I/O error for a file named "**<filename>**" has occurred.

User Response: Verify that the file exists, that you have the appropriate access to the file, and that the file is not in use by another user.

GSE2007E An attribute conversion error has occurred.

User Response: Check to be sure that all attribute types in the table are supported - for example, BLOB data is not supported in shape files. Also check for out-of-range data values, or illegal data values such as bad dates.

GSE2008E The import/export function has run out of memory.

User Response: Verify that you adequate memory available.

Chapter 11. Catalog views

DB2 Spatial Extender's catalog views contain metadata on:

- Coordinate systems that you can use. For information such as these systems' identifiers and annotation texts, see "DB2GSE.COORD_REF_SYS".
- Spatial columns that have been registered as layers. For information such as these columns' names, data types, and associated spatial reference systems, see "DB2GSE.GEOMETRY_COLUMNS" on page 106.
- Geocoders that you can use. For information such as these geocoders' identifiers and the regions that contain the locations that the geocoders process, see "DB2GSE.SPATIAL_GEOCODER" on page 106.
- Spatial reference systems that you can use. For information such as their identifiers and descriptions of them, see "DB2GSE.SPATIAL_REF_SYS" on page 107.

DB2GSE.COORD_REF_SYS

When you enable a database for spatial operations, DB2 Spatial Extender registers the coordinate systems that you can use in a catalog table. Selected columns from this table comprise the DB2GSE.COORD_REF_SYS catalog view, which is described in Table 34.

Table 34. Columns in the DB2GSE.COORD_REF_SYS catalog view

Name	Data Type	Nullable?	Content
SCID	INTEGER	No	Unique numeric identifier for this coordinate system.
SC_NAME	VARCHAR(64)	No	Name of this coordinate system.
AUTH_NAME	VARCHAR(256)	Yes	Name of the organization that compiled this coordinate system adheres to; for example, the European Petroleum Survey Group (EPSG).
AUTH_SRID	INTEGER	Yes	A numeric identifier assigned to this coordinate system by the organization specified in the AUTH_NAME column.
DESC	VARCHAR(256)	Yes	Description of this coordinate system.
SRTEXT	VARCHAR(2048)	No	Annotation text for this coordinate system.

DB2GSE.GEOMETRY_COLUMNS

When you create a layer, DB2 Spatial Extender registers it by recording its identifier and information relating to it in a catalog table. Selected columns from this table comprise the DB2GSE.GEOMETRY_COLUMNS catalog view, which is described in Table 35.

Table 35. Columns in the DB2GSE.GEOMETRY_COLUMNS catalog view

Name	Data Type	Nullable?	Content
LAYER_CATALOG	VARCHAR(30)	Yes	Fully-qualified name of this layer.
LAYER_SCHEMA	VARCHAR(30)	No	Schema of the table or view that contains the column that was registered as this layer.
LAYER_NAME	VARCHAR(128)	No	Name of the table or view that contains the column that was registered as this layer.
LAYER_COLUMN	VARCHAR(30)	No	Name of the column that was registered as this layer.
GEOMETRY_TYPE	INTEGER	No	Data type of the column that was registered as this layer.
SRID	INTEGER	No	Identifier of the spatial reference system used for the values in the column that was registered as this layer.
STORAGE_TYPE	INTEGER	Yes	Information as to how DB2 stores the values in the column that was registered as this layer. For example, data in STORAGE_TYPE might indicate that the values are stored as large objects (LOBs) or as instances of abstract data types (ADTs).

DB2GSE.SPATIAL_GEOCODER

Available geocoders are registered in a catalog table. Selected columns from this table comprise the DB2GSE.SPATIAL_GEOCODER catalog view, which is described in Table 36.

Table 36. Columns in the DB2GSE.SPATIAL_GEOCODER catalog view

Name	Data Type	Nullable?	Content
GCID	INTEGER	No	Numeric identifier of this geocoder.
GC_NAME	VARCHAR(64)	No	Short description of this geocoder.
VENDOR_NAME	VARCHAR(128)	No	Name of the vendor that provided this geocoder.
PRIMARY_UDF	VARCHAR(256)	No	Fully qualified name of this geocoder.

Table 36. Columns in the DB2GSE.SPATIAL_GEOCODER catalog view (continued)

Name	Data Type	Nullable?	Content
PRECISION_LEVEL	INTEGER	No	The degree to which source data must match corresponding reference data in order to be processed successfully by the geocoder.
VENDOR_SPECIFIC	VARCHAR(256)	Yes	Path to, and name of, a file that a vendor can use to set any special parameters that this geocoder supports.
GEO_AREA	VARCHAR(256)	Yes	Geographical area that contains the locations to be geocoded.
DESCRIPTION	VARCHAR(256)	Yes	Remarks provided by the vendor.

DB2GSE.SPATIAL_REF_SYS

When you create a spatial reference system, DB2 Spatial Extender registers it by recording its identifier and information related to it in a catalog table. Selected columns from this table comprise the DB2GSE.SPATIAL_REF_SYS catalog view, which is described in Table 37.

Table 37. Columns in the DB2GSE.SPATIAL_REF_SYS catalog view

Name	Data Type	Nullable?	Content
SRID	INTEGER	No	User-defined identifier for this spatial reference system.
SR_NAME	VARCHAR(64)	No	Name of this spatial reference system.
SCID	INTEGER	No	Numeric identifier for the coordinate system that underlies this spatial reference system.
SC_NAME	VARCHAR(64)	No	Name of the coordinate system that underlies this spatial reference system.
AUTH_NAME	VARCHAR(256)	Yes	Name of the organization that sets the standards for this spatial reference system.
AUTH_SRID	INTEGER	Yes	The identifier that the organization specified in the AUTH_NAME column assigns to this spatial reference system.
SRTEXT	VARCHAR(2048)	No	Annotation text for this spatial reference system.

Chapter 12. Spatial indexes

Because spatial columns contain two-dimensional geographic data, applications querying those columns require an index strategy that quickly identifies all geometries that lie within a given extent. For this reason, DB2 Spatial Extender provides the three-tiered spatial index based on a grid.

This chapter describes this kind of index and provides guidelines on using it. The topics covered are:

- “A sample program fragment”
- “B tree indexes” on page 110
- “Ways to create a spatial index” on page 110
- “How a spatial index is generated” on page 111
- “Guidelines on using a spatial index” on page 115

A sample program fragment

Consider the following example of how an index is created and used in SQL. You can refer to the *SQL Reference* for more information about the CREATE INDEX and CREATE INDEX EXTENSION commands. Notice that after the index is created, you can then issue standard DDL and DML statements that use the spatial functions and predicates.

```
create table customers (cid int, addr varchar(40), ..., loc db2gse.ST_Point)
create table stores (sid int, addr varchar(40), ..., loc db2gse.ST_Point,
    zone db2gse.ST_Polygon)

create index customersx1 on customers(loc) extend using spatial_index(10e0,
    100e0, 1000e0)
create index storesx1 on stores(loc) extend using spatial_index(10e0, 100e0,
    1000e0)
create index storesx2 on stores(zone) extend using spatial_index(10e0, 100e0,
    1000e0)

insert into customers (cid, addr, loc) values
(:cid, :addr, sdeFromBinary(:loc))
insert into customers (cid, addr, loc) values
(:cid, :addr, geocode(:addr))
insert into stores (sid, addr, loc) values
(:sid, :addr, sdeFromBinary(:loc))

update stores set zone = db2gse.ST_Buffer (loc, 2)

select cid, loc from customers
    where db2gse.ST_Within(loc, :polygon) = 1
```

```

select cid, loc from customers
  where db2gse.ST_Within(loc, :circle1) = 1 OR
         db2gse.ST_Within(loc, :circle2) = 1

select c.cid, loc from customers c, stores s
  where db2gse.ST_Contains(s.zone, c.loc) = 1 selectivity 0.01

select avg(c.income) from customers c
  where not exist (select * from stores s
                  where db2gse.ST_Distance(c.loc, s.loc) < 10)

```

B tree indexes

Spatial indexing technology is based on the traditional hierarchical B tree index, but is significantly different. The spatial index utilizes *grid indexing* which is designed to index two-dimensional spatial columns. The B tree index can only handle one-dimensional data and cannot be used with GIS information. This section describes how a B tree index is structured and used.

The top level of a B tree index, called the root node, contains one key for each node at the next level. The value of each key is the largest existing key value for the corresponding node at the next level. Depending on the number of values in the base table, several intermediate nodes may be needed. These nodes form a bridge between the root node and the leaf nodes that hold the actual base table row IDs.

The database manager searches a B tree index starting at the root node. It then continues through the intermediate nodes until it reaches the leaf node with the row ID of the base table.

The B tree index cannot be applied to a spatial column because the two-dimensional characteristic of the spatial column requires the structure of a spatial index. For the same reason, you cannot apply a spatial index to a non-spatial column. Further, a spatial index cannot be applied to a composite column of any kind.

Ways to create a spatial index

There are several ways to create a spatial index:

- Define one from the Create Spatial Index window. For instructions, see “Chapter 6. Creating spatial indexes” on page 47.
- Invoke the `db2gse.gse_enable_idx` stored procedure in an application program. For information about this stored procedure, see “Chapter 9. Stored procedures” on page 63.
- Issue the **db2 create index** command with the **spatial_index** function in the **USING** clause. For example:

```
create index storesx1 on customers (loc) using spatial_index(10e0,  
100e0, 1000e0)
```

The nature of spatial data requires that the database designer understand its relative size distribution. The designer must determine the optimum size and number of grid levels with which to create the spatial index.

The grid levels, <grid level 1>, <grid level 2>, and <grid level 3>, are entered by increasing the cell size. Thus, the second level must have a larger cell size than the first, and the third larger than the second. The first grid level is mandatory, but you can disable the second and third with a double precision zero value (0.0e0).

How a spatial index is generated

A spatial index is generated using *envelopes*. The envelope is a geometry itself and represents the minimum and maximum X and Y extent of a geometry. For most geometries, the envelope is a box, but for horizontal and vertical linestrings the envelope is a two-point linestring. For points, the envelope is the point itself. For more information about envelopes, see “Envelope” on page 123.

The spatial index is constructed on a spatial column by making one or more entries for the intersections of each geometry’s envelope with the grid. An intersection is recorded as the internal ID of the geometry and minimum X and Y coordinates of the grid cell intersected. For example, the polygon in Figure 7 on page 112 intersects the grid on coordinates (20,30), (30,30), (40,30), (20,40), (30,40), (40,40), (20,50), (30,50), and (40,50). See Table 38 on page 112 for minimum X and Y coordinates for all the geometries in Figure 7 on page 112.

If multiple grid levels exist, DB2 Spatial Extender attempts to use the lowest grid level possible. When a geometry has intersected four or more grid cells at a given level, it is promoted to the next higher level. Therefore, a spatial index that has the three grid levels of 10.0e0, 100.0e0, and 1000.0e0, DB2 Spatial Extender will first intersect each geometry with the level 10.0e0 grid. If a geometry intersects with four or more 10.0e0 grid cells, it is promoted and intersected with the level 100.0e0 grid. If four or more intersections result at the 100.0e0 level, the geometry is promoted to the 1000.0e0 level. At the 1000.0e0 level, the intersections must be entered into the spatial index because this is the highest possible level.

Figure 7 on page 112 illustrates how four different types of geometries intersect a 10.0e grid. All 23 intersections for the four geometries are recorded in the spatial index.

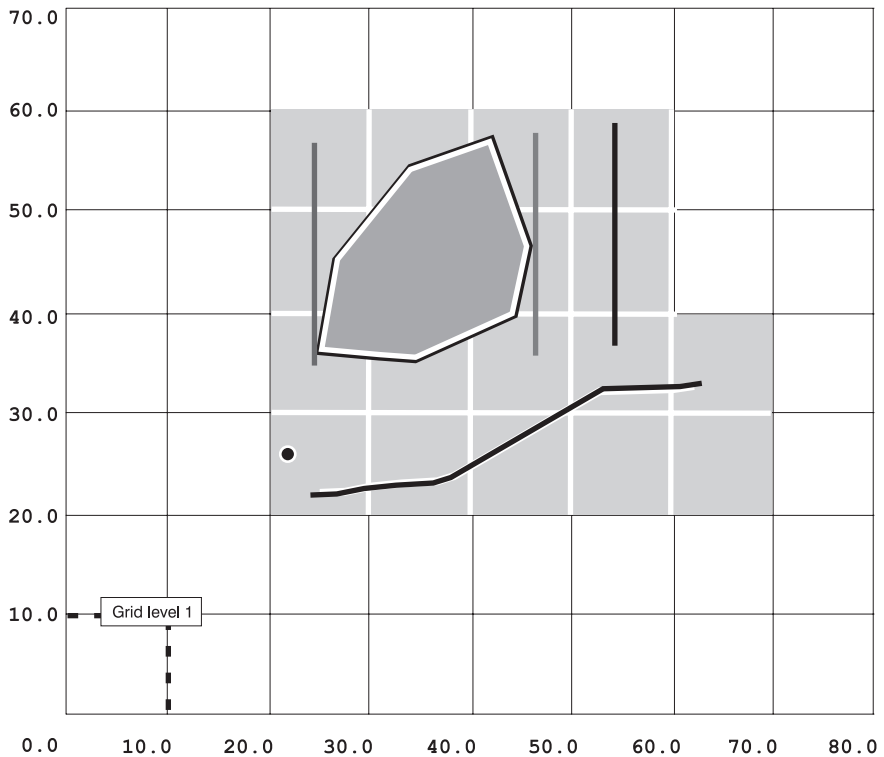


Figure 7. Application of a 10.0e0 grid level

Table 38 lists the geometries and their corresponding grid intersections. The envelopes of four different geometries types intersect the 10.0e grid. The minimum X and Y coordinate of each grid cell that it intersects are entered into the spatial index.

Table 38. The 10.0e0 grid cell entries for the example geometries

Geometry	Grid X	Grid Y
Polygon	20.0	30.0
Polygon	30.0	30.0
Polygon	40.0	30.0
Polygon	20.0	40.0
Polygon	30.0	40.0
Polygon	40.0	40.0
Polygon	20.0	50.0
Polygon	30.0	50.0
Polygon	40.0	50.0

Table 38. The 10.0e0 grid cell entries for the example geometries (continued)

Geometry	Grid X	Grid Y
Vertical linestring	50.0	30.0
Vertical linestring	50.0	40.0
Vertical linestring	50.0	50.0
Point	20.0	20.0
Horizontal linestring	20.0	20.0
Horizontal linestring	30.0	20.0
Horizontal linestring	40.0	20.0
Horizontal linestring	50.0	20.0
Horizontal linestring	60.0	20.0
Horizontal linestring	20.0	30.0
Horizontal linestring	30.0	30.0
Horizontal linestring	40.0	30.0
Horizontal linestring	50.0	30.0
Horizontal linestring	60.0	30.0

Figure 8 on page 114 displays how the number of intersections is greatly reduced to eight by the addition of grid levels 30.0e0 and 60.0e0. In this case, the polygon identified as geometry 1 is promoted to grid level 30.0e0 and the linestring identified as geometry 4 is promoted to grid level 60.0e0. Instead of the nine and ten intersections that geometries had at the 10.0e0 level, they have only two after promotion.

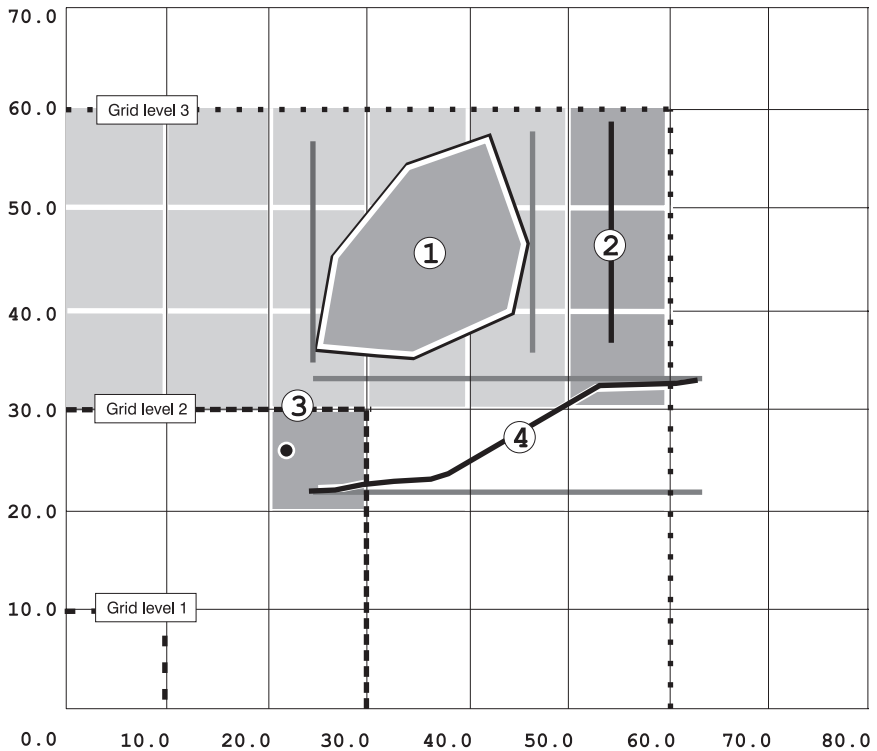


Figure 8. Effect of adding grid levels 30.0e0 and 60.0e0. The envelope of the polygon identified as geometry 1 intersects nine grid cells. The envelope of the vertical linestring identified as geometry 2 intersects three grid cells. The envelope of the point identified as geometry 3 intersects just one grid cell. The envelope of the linestring identified as geometry 4 intersects ten grid cells.

DB2 Spatial Extender takes the grid level parameters specified in the CREATE INDEX statement and checks each spatial object to determine the coordinates and number of grid blocks in which the object exists. In Figure 8, the grid levels 10.0e0, 30.0e0, and 60.0e0 are displayed with ever-increasing line weights and different shades of gray. The vertical linestring and the point envelope cell intersections are entered into the index at the 10.0e0 grid level, because both generate less than four intersections. The polygon intersects nine 10.0e0 grid cells, and is therefore promoted to the 30.0e0 grid level. At this level, the polygon intersects two grid cells, which are entered into the index. The linestring identified as geometry 4 intersects ten 10.0e0 grid cells, and is therefore promoted to the 30.0e0 grid level. Yet at this level, it intersects six grid cells, so it is again promoted to the 60.0e0 grid level, where it generates two intersections. The linestring 60.0e0 grid intersections are then entered into the index. If the linestring had generated four or more intersections at this level, they still would have been entered into the index because this is the highest level at which a geometry can be promoted.

Table 39. The intersections of the geometries in the three-tiered index

Geometry	Grid X	Grid Y
<i>The intersections between the vertical linestring and the point in the level 1 (10.0e0 grid size)</i>		
2	50.0	30.0
2	50.0	40.0
2	50.0	50.0
3	20.0	20.0
<i>The intersections of the polygon in the level 2 (30.0e0 grid size)</i>		
1	0.0	30.0
1	30.0	30.0
<i>The intersections of the linestring in the level 3 (60.0e0 grid size)</i>		
4	0.0	0.0
4	60.0	0.0

DB2 Spatial Extender does not actually create a polygon grid structure of any kind. DB2 Spatial Extender manifests each grid level parametrically by defining the origin at the X,Y offset of the columns' spatial reference system. It then extends the grid into positive coordinate space. Using a parametric grid, DB2 Spatial Extender generates the intersections mathematically.

Guidelines on using a spatial index

DB2 Spatial Extender works with a spatial index to improve the performance of a spatial query. Consider the most basic and probably most popular spatial query, the box query. This query asks DB2 Spatial Extender to return all geometries that are either fully or partially within a user-defined box. If an index does not exist, DB2 Spatial Extender must compare all of the geometries with the box. However, with an index, DB2 Spatial Extender can locate all the index entries that have a lower left-hand coordinate greater than or equal the box's and an upper right-hand coordinate less than or equal to the box's. Because the index is ordered by this coordinate system, DB2 Spatial Extender is able to quickly obtain a list of candidate geometries. The process just described is referred to as the *first pass*.

A *second pass* determines if each candidate's envelope intersects the box. A geometry that qualifies for first pass because its grid cells' envelope intersects the box may itself have an envelope that does not.

A *third pass* compares the actual coordinates of the candidate with the box to determine if any part of the geometry is actually within the box. This last and

rather complex process of comparison operates on a list of candidates composed of a subset of the total population, which is significantly reduced by the first two passes.

All spatial queries perform the three passes except for the `EnvelopesIntersect` function. It performs only the first two passes. The `EnvelopesIntersect` function was designed for display operations that often employ their own built-in clipping routines and don't require the granularity of the third pass.

Selecting the grid cell size

The irregular shape of the geometry envelopes complicates the selection of the grid cell size. Because of this irregularity, some geometry envelopes intersect several grids, while others fit inside a single grid cell. Conversely, depending on the spatial distribution of the data, some grid cells intersect many geometry envelopes.

For a spatial index to function well, it is essential that the correct number and size of grids are selected. Consider a spatial column containing uniformly sized geometry. In this case, a single grid level will suffice. Start with a grid cell size that encompasses the average geometry envelope. While testing your application, you might find that increasing the grid cell size improves the performance of your queries. This is because each grid cell contains more geometries, and the first pass is able to discard non-qualifying geometries faster. However, you will find that as you continue to increase the cell size, performance will begin to deteriorate. This is because eventually the second pass will have to contend with more candidates.

Selecting the number of levels

If the objects that you want to index are about the same relative size, you can use a single grid level. Although this is true, not all columns will contain geometry of the same relative size. Usually geometries of spatial columns can be grouped into several size intervals. For example, consider a road network in which the geometries are divided into streets, major roads, and highways. The streets are all about the same length and can be grouped in one size interval. This is also true for the major roads and highways. Therefore, the streets, representing one size interval, could be grouped into the first grid level, the road networks into the second, and the major highways into the third grid level. Another example includes a county-parcel column that contains clusters of small urban parcels surrounded by larger rural parcels. In this instance, there are two size intervals and two grid levels, one for the small urban parcels, and another for the larger rural parcels. These situations are very common and require the use of a multilevel grid.

To select the cell size of each grid level, select grid cell sizes that are slightly larger than each size interval. Test the index by performing queries against the spatial column.

Each additional level requires an extra index scan. Try adjusting the grid sizes up or down slightly to determine if an appreciable improvement in performance can be obtained.

Chapter 13. Geometries and associated spatial functions

This chapter discusses units of information, called *geometries*, that consist of coordinates and symbolize geographic features. The chapter also introduces spatial functions that take geometries as input and return results that help you to analyze geographic features and to move spatial data between geographic information systems. The topics covered are:

- The nature of geometries
- Geometries' properties; functions that return information related to these properties
- Instantiable geometries; functions that operate on them
- Functions that:
 - Show relationships and comparisons between geographic features
 - Generate geometries
 - Convert geometry values into importable and exportable formats

About geometries

The Oxford American Dictionary defines *geometry* as “the branch of mathematics dealing with the properties of and relations of lines, angles, surfaces and solids.” On August 11, 1997, the Open GIS Consortium Inc. (OGC) in its publication, *Open GIS Features for ODBC (SQL) Implementation Specification*, coined another definition for the term. The word *geometry* was selected to denote the geometric features that, for the past millennium or more, cartographers have used to map the world. A very abstract definition of this new meaning of geometry might be “a point or aggregate of points symbolizing a feature on the ground.”

In DB2 Spatial Extender, an *operational* definition of geometry might be “a model of a geographic feature.” The model can be expressed in terms of the feature's coordinates and also, in some cases, in terms of a visual symbol. The model conveys information; for example, the coordinates identify the position of the feature with respect to fixed points of reference, and the symbol outlines its form. Also, the model can be used to produce information; for example, the `ST_Overlaps` function can take the coordinates of two proximate regions as input and return information as to whether the regions overlap or not.

The coordinates of a feature that a geometry symbolizes are regarded as *properties* of the geometry. Several kinds of geometries have other properties as well; for example:

- An *interior* represents the content of the feature that the geometry symbolizes.
- An *exterior* represents the space around the feature.
- A *boundary* represents the demarcation where the content ends and the surrounding space begins.

These and additional properties are discussed in “Properties of geometries and associated functions” on page 121.

The geometries supported by DB2 Spatial Extender form a hierarchy, shown in Figure 9. Six members of the hierarchy are instantiable; they can be expressed as visual symbols, which are also shown in the figure.

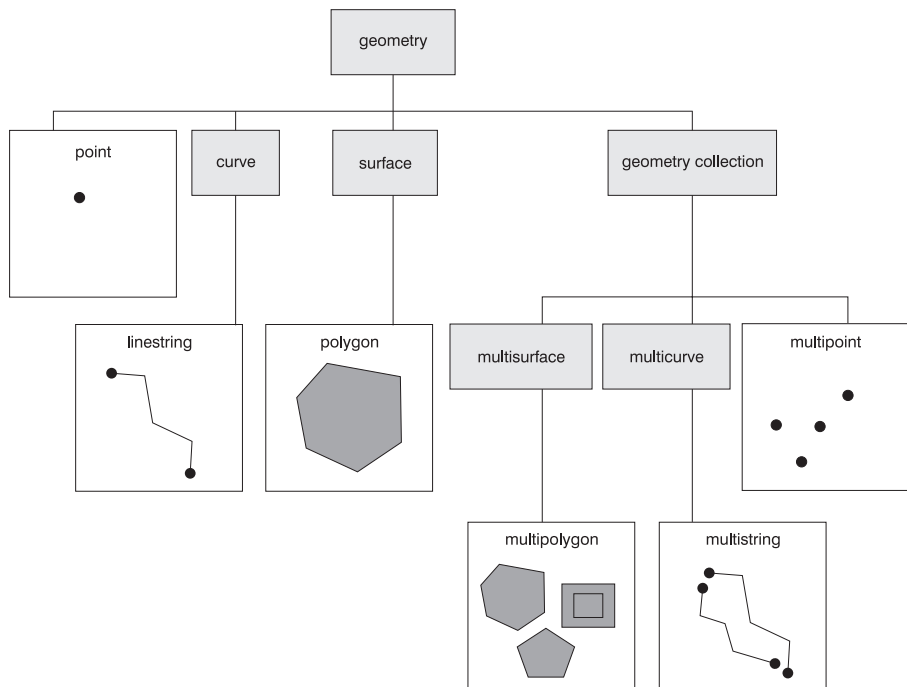


Figure 9. Hierarchy of geometries supported by DB2 Spatial Extender. Instantiable geometries can be expressed as visual symbols. These symbols are shown under the names of these geometries.

As Figure 9 indicates, a superclass called *geometry* is the root of the hierarchy. The subclasses are divided into two categories: the base geometry subclasses, and the homogeneous collection subclasses. The base geometries include:

- *Points*, which symbolize discrete features that are perceived as occupying the locus where an east-west coordinate line (such as a parallel) intersects a north-south coordinate line (such as a meridian). For example, suppose that

the notation on a large-scale map shows that each city on the map is located at the intersection of a parallel and a meridian. On this scale, each city could be symbolized by a point.

- *Linestrings*, which symbolize linear geographic features (for example, streets, canals, and pipelines).
- *Polygons*, which symbolize multisided geographic features (for example, welfare districts, forests, and wildlife habitats).

The homogeneous collections include:

- *Multipoints*, which symbolize multipart features whose components are each located at the intersection of an east-west coordinate line and a north-south coordinate line (for example, an island chain whose members are each situated at an intersection of a parallel and meridian).
- *Multilinestrings*, which symbolize multipart features made up of linear units or components (for example, river systems and highway systems).
- *Multipolygons*, which symbolize multipart features made up of multisided units or components (for example, the collective farmlands in a specific region, or a system of lakes).

As their names imply, the homogeneous collections are collections of base geometries. In addition to sharing base geometry properties, homogeneous collections have some of their own properties as well.

The spatial data types supported by DB2 Spatial Extender are implementations of the geometries shown in Figure 9 on page 120. For a description of these data types, see “About spatial data types” on page 31.

Properties of geometries and associated functions

This section describes geometries’ properties and the spatial functions that are associated with these properties. The section begins with core properties:

- What class a geometry belongs to
- X and Y coordinates

This section also explains:

- Z coordinates
- Measures
- A geometry’s interior, boundary, and exterior
- The quality of being simple or non-simple
- The quality of being empty or not empty
- A geometry’s envelope
- Dimension
- The identifier of a geometry’s associated spatial reference system

Class

Each geometry belongs to a class in the hierarchy shown in Figure 9 on page 120. As indicated in “About geometries” on page 119, six subclasses in the hierarchy—points, linestrings, polygons, multipoints, multilinestrings, and multipolygons—are instantiable. The superclass and other subclasses are not instantiable.

The `ST_GeometryType` function takes a geometry and returns the instantiable subclass in the form of a character string. For more information, see “`ST_GeometryType`” on page 212.

The `ST_IsValid` function takes a geometry that has been assigned to an `ST_Geometry` data type. The function returns 1 (TRUE) if the geometry is valid and 0 (FALSE) if the geometry is not valid. For more information, see “`ST_IsValid`” on page 228.

X and Y coordinates

An *X coordinate value* denotes a location that is relative to a point of reference to the east or west. A *Y coordinate value* denotes a location that is relative to a point of reference to the north or south. For further information, see “The nature of spatial data” on page 5 and “About coordinate and spatial reference systems” on page 23.

Z coordinates

Some geometries have an associated altitude or depth. Each of the points that form the geometry of a feature can include an optional Z coordinate that represents an altitude or depth normal to the earth’s surface.

The `Is3d` predicate function takes a geometry and returns 1 (TRUE) if the function has Z coordinates and 0 (FALSE) otherwise. For more information, see “`Is3d`” on page 160.

Measures

A measure is a value that conveys information about a geographic feature and that is stored together with the coordinates that define the feature’s location. For example, suppose that you are representing transportation systems in your GIS. If you want your application to process values that denote linear distances or mileposts, you can store these values along with the coordinates that define the locations of the systems. Measures are stored as double precision numbers.

The `IsMeasured` predicate takes a geometry and returns a 1 (TRUE) if it contains measures and 0 (FALSE) otherwise. For more information, see “`IsMeasured`” on page 161.

Interior, boundary, and exterior

All geometries occupy a position in space defined by their interior, boundary, and exterior. The exterior of a geometry is all space not occupied by the geometry. The boundary of a geometry serves as the interface between its interior and exterior. The interior is the space occupied by the geometry. Subclasses inherit the interior and exterior properties directly, yet the boundary property differs for each.

The `ST_Boundary` function takes a geometry and returns a geometry that represents the source geometry's boundary. For more information, see "`ST_Boundary`" on page 182.

Simple or non-simple

Some subclasses of geometry (linestrings, multipoints, and multilinestrings) are either simple or non-simple. A subclass is simple if it obeys all the topological rules imposed on the subclass, and non-simple if it doesn't. A linestring is simple if it does not intersect its interior. A multipoint is simple if none of its elements occupy the same coordinate space. A multilinestring is simple if none of its element's interiors are intersected by its own interior.

The `ST_IsSimple` predicate function takes a geometry and returns 1 (TRUE) if the geometry is simple and 0 (FALSE) otherwise. For more information, see "`ST_IsSimple`" on page 227.

Empty or not empty

A geometry is empty if it does not have any points. The envelope, boundary, interior, and exterior of an empty geometry are NULL. An empty geometry is always simple and can have Z coordinates or measures. Empty linestrings and multilinestrings have a 0 length. Empty polygons and multipolygons have a 0 area.

The `ST_IsEmpty` predicate function takes a geometry and returns 1 (TRUE) if the geometry is empty and 0 (FALSE) otherwise. For more information, see "`ST_IsEmpty`" on page 224.

Envelope

The envelope of a geometry is the bounding geometry formed by the minimum and maximum (X,Y) coordinates. With the following exceptions, the envelopes of most geometries form a boundary rectangle:

- The envelope of a point is the point itself, because its minimum and maximum coordinates are the same.
- The envelope of a horizontal or vertical linestring is a linestring represented by the boundary (the endpoints) of the source linestring.

The ST_Envelope function takes a geometry and returns a bounding geometry, which represents its envelope. For more information, see “ST_Envelope” on page 202.

Dimension

A geometry can have a dimension of 0, 1, or 2. The dimensions are listed as follows:

- 0 Has neither length nor area
- 1 Has a length
- 2 Contains area

The point and multipoint subclasses have a dimension of zero. Points represent dimensional features that can be modeled with a single coordinate, while multipoint subclasses represent data that must be modeled with a cluster of disconnected coordinates.

The subclasses linestring and multilinestring have a dimension of one. They store road segments, branching river systems and any other features that are linear in nature.

Polygon and multipolygon subclasses have a dimension of two. Features whose perimeter encloses a definable area, such as forests, parcels of land, and water bodies can be rendered by either the polygon or multipolygon data type.

Dimension is important not only as a property of the subclass, but it also plays a part in determining the spatial relationship of two features. The dimension of the resulting feature or features determines whether or not the operation was successful. DB2 Spatial Extender examines the dimension of the features to determine how they should be compared.

The ST_Dimension function takes a geometry and returns its dimension as an integer. For more information, see “ST_Dimension” on page 196.

Spatial reference system identifier

The spatial reference system identifies the coordinate transformation for each geometry.

All spatial reference systems known to the database can be accessed through the DB2GSE.SPATIAL_REF_SYS catalog view. For information about this view, see “Chapter 11. Catalog views” on page 105.

The ST_SRID function takes a geometry and returns its spatial reference identifier as an integer. For more information, see “ST_SRID” on page 260.

The ST_Transform function assigns a geometry to a spatial reference system other than the spatial reference system to which the geometry is currently assigned. For more information, see “ST_Transform” on page 265.

Instantiable geometries and associated functions

This section profiles the six subclasses of instantiable geometries and describes the functions that operate on them. The subclasses are:

- Points
- Linestrings
- Polygons
- Multipoints
- Multilinestrings
- Multipolygons

For illustrations of the hierarchy to which these subclasses belong and of the visual symbols associated with them, see Figure 9 on page 120.

Points

A point is a zero-dimensional geometry that occupies a single location in coordinate space. A point includes an X coordinate and a Y coordinate that define this location. It can also include a Z coordinate and a measure.

A point is simple and has a NULL boundary. Points are often used to define features such as oil wells, landmarks, and elevations.

Functions that operate solely on the point subclass:

ST_Point

Takes an X coordinate, its associated Y coordinate, and the identifier of the spatial reference system to which these coordinates belong, and returns the point that the coordinates define. For more information, see “ST_Point” on page 251.

ST_CoordDim

Returns a value that denotes what coordinates a point contains, and whether it also contains a measure. This value is called a *coordinate dimension*. Possible coordinate dimensions are:

- 2 The point consists of an X coordinate and a Y coordinate.
- 3 The point consists of an X coordinate, a Y coordinate, and a Z coordinate.
- 4 The point consists of an X coordinate, a Y coordinate, a Z coordinate, and a measure.

For more information, see “ST_CoordDim” on page 191.

ST_PointFromText

Takes an OGC well-known text (WKT) representation of a point and returns the point. For more information, see “ST_PointFromText” on page 249.

ST_X Returns an ST_Point data type’s X coordinate value as a double precision number. For more information, see “ST_X” on page 271.

ST_Y Returns an ST_Point data type’s Y coordinate value as a double precision number. For more information, see “ST_Y” on page 272.

Z Returns an ST_Point data type’s Z coordinate value as a double precision number. For more information, see “Z” on page 273.

M Returns an ST_Point data type’s measure as a double precision number. For more information, see “M” on page 168.

Linestrings

A linestring is a one-dimensional object stored as a sequence of points defining a linear interpolated path. The linestring is simple if it does not intersect its interior. The endpoints (the boundary) of a closed linestring occupy the same point in space. A linestring is a ring if it is closed and if its interior does not intersect itself. In addition to the other properties inherited from the superclass geometry, linestrings have length. Linestrings are often used to define linear features such as roads, rivers, and power lines.

A simple linestring whose startpoint and endpoint are the same is called a *ring*.

The endpoints normally form the boundary of a linestring unless the linestring is closed in which case the boundary is NULL. The interior of a linestring is the connected path that lies between the endpoints, unless it is closed in which case the interior is continuous.

Functions that operate on linestrings:

ST_StartPoint

Takes a linestring and returns its first point. For more information, see “ST_StartPoint” on page 261.

ST_EndPoint

Takes a linestring and returns its last point. For more information, see “ST_Endpoint” on page 201.

ST_PointN

Takes a linestring and an index to *n*th point and returns that point. For more information, see “ST_PointN” on page 252.

ST_Length

Takes a linestring and returns its length as a double precision number. For more information, see “ST_Length” on page 230.

ST_NumPoints

Takes a linestring and returns the number of points in its sequence as an integer. For more information, see “ST_NumPoints” on page 244.

ST_IsRing

Takes a linestring and returns 1 (TRUE) if the linestring is a ring and 0 (FALSE) otherwise. For more information, see “ST_IsRing” on page 226.

ST_IsClosed

Takes a linestring and returns 1 (TRUE) if the linestring is closed and 0 (FALSE) otherwise. For more information, see “ST_IsClosed” on page 222.

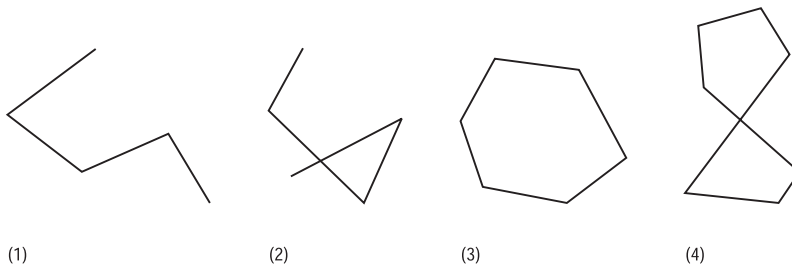


Figure 10. Linestring objects.

1. A simple non-closed linestring.
2. A non-simple non-closed linestring.
3. A closed simple linestring and therefore a ring.
4. A closed non-simple linestring. It is not a ring.

Polygons

A polygon is a two-dimensional surface stored as a sequence of points defining its exterior bounding ring and 0 or more interior rings. A polygon’s rings cannot overlap. Therefore, by definition, polygons are always simple. Most often they define parcels of land, water bodies, and other features that have a spatial extent.

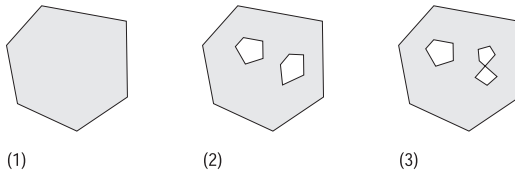


Figure 11. Polygons.

1. A polygon whose boundary is defined by an exterior ring.
2. A polygon whose boundary is defined by an exterior ring and two interior rings. The area inside the interior rings is part of the polygons exterior.
3. A legal polygon because the rings intersect at a single tangent point.

The exterior and any interior rings define the boundary of a polygon, and the space enclosed between the rings defines the polygon's interior. The rings of a polygon can intersect at a tangent point but never cross. In addition to the other properties inherited from the superclass geometry, polygons have area.

Functions that operate on polygons:

ST_Area

Takes a polygon and returns its area as a double precision number. For more information, see "ST_Area" on page 178.

ST_ExteriorRing

Takes a polygon and returns its exterior ring as a linestring. For more information, see "ST_ExteriorRing" on page 205.

ST_NumInteriorRing

Takes a polygon and returns the number of interior rings that it contains. For more information, see "ST_NumInteriorRing" on page 243.

ST_InteriorRingN

Takes a polygon and an index and returns the *n*th interior ring as a linestring. For more information, see "ST_InteriorRingN" on page 214.

ST_Centroid

Takes a polygon and returns a point that is the center of the polygon's extent. For more information, see "ST_Centroid" on page 186.

ST_PointOnSurface

Takes a polygon and returns a point that is guaranteed to be on the surface of the polygon. For more information, see "ST_PointOnSurface" on page 253.

ST_Perimeter

Takes a polygon and returns the perimeter of its surface. For more information, see "ST_Perimeter" on page 248.

Multipoints

A multipoint is a collection of points, and like its elements, it has a dimension of 0. A multipoint is simple if none of its elements occupy the same coordinate space. The boundary of a multipoint is NULL. Multipoints may be used to define phenomena such as aerial broadcast patterns and incidents of an epidemic outbreak.

Functions that operate on multipoints:

ST_NumGeometries

Takes a homogeneous collection and returns the number of base geometry elements it contains. For more information, see “ST_NumGeometries” on page 242.

ST_GeometryN

Takes a homogeneous collection and an index and returns the nth base geometry. For more information, see “ST_GeometryN” on page 211.

Multilinestrings

A multilinestring is a collection of linestrings. Multilinestrings are simple if they intersect only at the endpoints of the linestring elements. Multilinestrings are non-simple if the interiors of the linestring elements intersect.

The boundary of a multilinestring is the non-intersected endpoints of the linestring elements. The multilinestring is closed if all of its linestring elements are closed. The boundary of a multilinestring is NULL if all of the endpoints of all of the elements are intersected. In addition to the other properties inherited from the superclass geometry, multilinestrings have length. Multilinestrings are used to define streams or road networks.

Functions that operate on multilinestrings:

ST_Length

Takes a multilinestring and returns the cumulative length of all its linestring elements as a double precision number. For more information, see “ST_Length” on page 230.

ST_IsClosed

Takes a multilinestring and returns 1 (TRUE) if the multilinestring is closed and 0 (FALSE) otherwise. For more information, see “ST_IsClosed” on page 222.

ST_NumGeometries

Takes a homogeneous collection and returns the number of base geometry elements it contains. For more information, see “ST_NumGeometries” on page 242.

ST_GeometryN

Takes a homogeneous collection and an index and returns the nth base geometry. For more information, see “ST_GeometryN” on page 211.

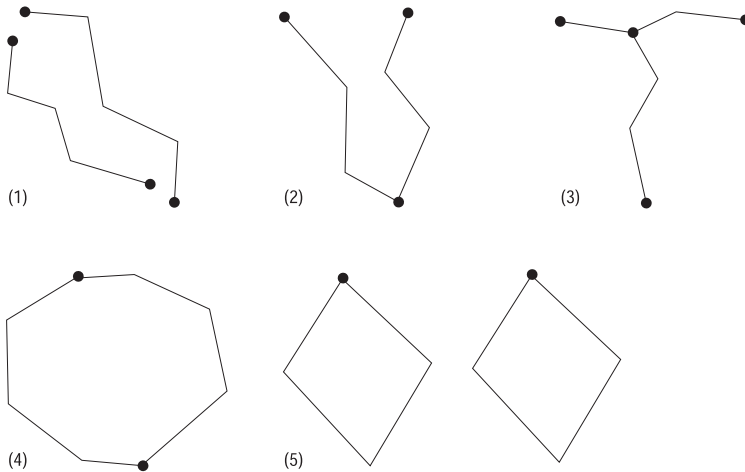


Figure 12. Multilinestrings.

1. A simple multilinestring whose boundary is defined by the four endpoints of its two linestring elements.
2. A simple multilinestring because only the endpoints of the linestring elements intersect. The boundary is defined by the two non-intersecting endpoints.
3. A non-simple linestring because the interior of one of its linestring elements is intersected. The boundary of this multilinestring is defined by the four endpoints, including the intersecting point.
4. A simple non-closed multilinestring. It is not closed because its element linestrings are not closed. It is simple because none of the interiors of any of the element linestrings are intersected.
5. A simple closed multilinestring. It is closed because all of its elements are closed. It is simple because none of its elements are intersected at the interiors.

Multipolygons

The boundary of a multipolygon is the cumulative length of its element's exterior and interior rings. The interior of a multipolygon is defined as the cumulative interiors of its element polygons. The boundary of a multipolygon's elements can intersect only at a tangent point. In addition to the other properties inherited from the superclass geometry, multipolygons have area. Multipolygons define features such as a forest stratum or a non-contiguous parcel of land such as an island chain.

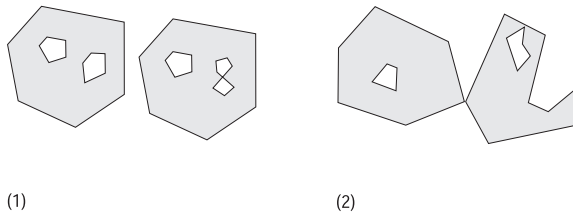


Figure 13. Multipolygons.

1. A multipolygon with two polygon elements. The boundary is defined by the two exterior rings and the three interior rings.
2. A multipolygon with two polygon elements. The boundary is defined by the two exterior rings and the two interior rings. The two polygon elements intersect at a tangent point.

Functions that operate on multipolygons:

ST_Area

Takes a multipolygon and returns the cumulative area of its polygon elements as a double precision number. For more information, see “ST_Area” on page 178.

ST_Centroid

Takes a multipolygon and returns a point that is its geometric-weighted center. For more information, see “ST_Centroid” on page 186.

ST_NumGeometries

Takes a homogeneous collection and returns the number of base geometry elements it contains. For more information, see “ST_NumGeometries” on page 242.

ST_GeometryN

Takes a homogeneous collection and an index and returns the nth base geometry. For more information, see “ST_GeometryN” on page 211.

Functions that show relationships and comparisons, generate geometries, and convert values’ formats

The preceding sections introduced three categories of spatial functions:

- Functions associated with geometries’ properties
- Functions associated with specific geometries

This section introduces three more categories:

- Functions that determine ways in which geographic features relate or compare
- Functions that generate new geometries

- Functions that convert the values of a geometry into a format that can be imported or exported

Functions that show relationships or comparisons between geographic features

Several spatial functions return information about ways in which geographic features relate to one another or compare with one another. Most of these functions, called *predicates*, are boolean functions. This section describes the predicates in general and then discusses each function individually.

Predicate functions

Predicate functions return 1 (TRUE) if a comparison meets the function's criteria, or 0 (FALSE) if the comparison fails. Predicates that test for a spatial relationship compare pairs of geometries that can be a different type or dimension.

Predicates compare the X and Y coordinates of the submitted geometries. The Z coordinates and the measure (if they exist) are ignored. This allows geometries that have Z coordinates or measure to be compared with those that do not.

The *Dimensionally Extended 9 Intersection Model (DE-9IM)*¹ is a mathematical approach that defines the pair-wise spatial relationship between geometries of different types and dimensions. This model expresses spatial relationships between all types of geometries as pair-wise intersections of their interior, boundary and exterior, with consideration for the dimension of the resulting intersections.

Given geometries a and b : $I(a)$, $B(a)$, and $E(a)$ represent the interior, boundary, and exterior of a . And, $I(b)$, $B(b)$, and $E(b)$ represent the interior, boundary, and exterior of b . The intersections of $I(a)$, $B(a)$, and $E(a)$ with $I(b)$, $B(b)$, and $E(b)$ produces a 3 by 3 matrix. Each intersection can result in geometries of different dimensions. For example, the intersection of the boundaries of two polygons consists of a point and a linestring, in which case the dim function would return the maximum dimension of 1.

The dim function returns a value of 1, 0, 1 or 2. The 1 corresponds to the null set or dim(null), which is returned when no intersection was found.

1. The DE-9IM was developed by Clementini and Felice, who dimensionally extended the 9 Intersection Model of Egenhofer and Herring. DE-9IM is collaboration of four authors, Clementini, Eliseo, Di Felice, and van Osstrom. They published the model in "A Small Set of Formal Topological Relationships Suitable for End-User Interaction," D. Abel and B.C. Ooi (Ed.), *Advances in Spatial Database—Third International Symposium. SSD '93*. LNCS 692. Pp. 277-295. The 9 Intersection model by Springer-Verlag Singapore (1993) Egenhofer M.J. and Herring, J., was published in "Categorizing binary topological relationships between regions, lines, and points in geographic databases," *Tech. Report, Department of Surveying Engineering*, University of Maine, Orono, ME 1991.

	Interior	Boundary	Exterior
Interior	$\dim(I(a) \cap I(b))$	$\dim(I(a) \cap B(b))$	$\dim(I(a) \cap E(b))$
Boundary	$\dim(B(a) \cap I(b))$	$\dim(B(a) \cap B(b))$	$\dim(B(a) \cap E(b))$
Exterior	$\dim(E(a) \cap I(b))$	$\dim(E(a) \cap B(b))$	$\dim(E(a) \cap E(b))$

The results of the spatial relationship predicates can be understood or verified by comparing the results of the predicate with a pattern matrix that represents the acceptable values for the DE-9IM.

The pattern matrix contains the acceptable values for each of the intersection matrix cells. The possible pattern values are:

- T** An intersection must exist, $\dim = 0, 1, \text{ or } 2$.
- F** An intersection must not exist, $\dim = -1$.
- *** It does not matter if an intersection exists, $\dim = -1, 0, 1, \text{ or } 2$.
- 0** An intersection must exist and its maximum dimension must be 0, $\dim = 0$.
- 1** An intersection must exist and its maximum dimension must be 1, $\dim = 1$.
- 2** An intersection must exist and its maximum dimension must be 2, $\dim = 2$.

For example, the following pattern matrix for the ST_Within predicate includes the values T, F, and *.

Table 40. Matrix for ST_Within. The pattern matrix of the ST_Within predicate for geometry combinations.

		b		
		Interior	Boundary	Exterior
a	Interior	T	*	F
	Boundary	*	*	F
	Exterior	*	*	*

The ST_Within predicate returns TRUE when the interiors of both geometries intersect and when the interior and boundary of *a* does not intersect the exterior of *b*. All other conditions do not matter.

Each predicate has at least one pattern matrix, but some require more than one to describe the relationships of various geometry type combinations.

ST_Equals

ST_Equals returns 1 (TRUE) if two geometries of the same type have identical X,Y coordinate values.


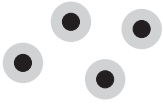



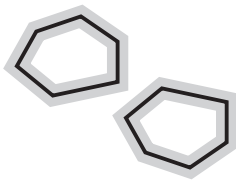
	
point / point	multipoint / multipoint
	
linestring /linestring	multistring /multistring
	
polygon / polygon	multipolygon / multipolygon

Figure 14. ST_Equals. Geometries are equal if they have matching X,Y coordinates.

Table 41. Matrix for equality. The DE-9IM pattern matrix for equality ensures that the interiors intersect and that no part interior or boundary of either geometry intersects the exterior of the other.

		b		
		Interior	Boundary	Exterior
a	Interior	T	*	F
	Boundary	*	*	F
	Exterior	F	F	*

For more information, see “ST_Equals” on page 204.

ST_OrderingEquals

ST_OrderingEquals compares two geometries and returns 1 (TRUE) if the geometries are equal and the coordinates are in the same order; otherwise, it returns 0 (FALSE). For more information, see “ST_OrderingEquals” on page 245.

ST_Disjoint

ST_Disjoint returns 1 (TRUE) if the intersection of the two geometries is an empty set.

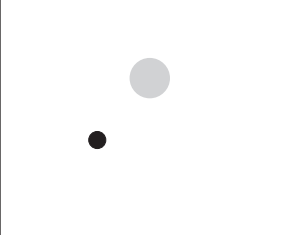
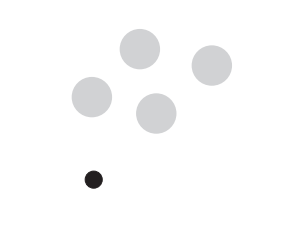




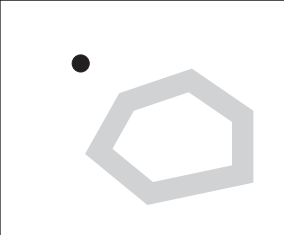
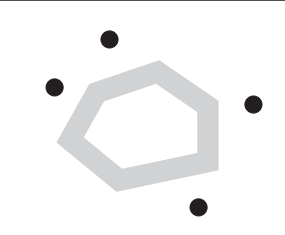
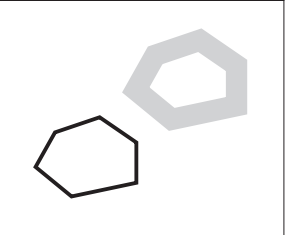
		
point / point	point / multipoint	multipoint / multipoint
		
point / linestring	multistring / linestring	polygon / linestring
		
point / polygon	multipoint / multipolygon	polygon / polygon

Figure 15. ST_Disjoint. Geometries are disjoint if they do not intersect one another in any way.

Table 42. Matrix for *ST_Disjoint*. The *ST_Disjoint* predicate's pattern matrix simple states that neither the interiors nor the boundaries of either geometry intersect.

		b		
		Interior	Boundary	Exterior
a	Interior	F	F	*
	Boundary	F	F	*
	Exterior	*	*	*

For more information, see “*ST_Disjoint*” on page 198.

ST_Intersects

ST_Intersects returns 1 (TRUE) if the intersection does not result in an empty set. *Intersects* returns the exact opposite result of *ST_Disjoint*.

The *ST_Intersects* predicate returns TRUE if the conditions of any of the following pattern matrices returns TRUE.

Table 43. Matrix for *ST_Intersects* (1). The *ST_Intersects* predicate returns TRUE if the interiors of both geometries intersect.

		b		
		Interior	Boundary	Exterior
a	Interior	T	*	*
	Boundary	*	*	*
	Exterior	*	*	*

Table 44. Matrix for *ST_Intersects* (2). The *ST_Intersects* predicate returns TRUE if the boundary of the first geometry intersects the boundary of the second geometry.

		b		
		Interior	Boundary	Exterior
a	Interior	*	T	*
	Boundary	*	*	*
	Exterior	*	*	*

Table 45. Matrix for *ST_Intersects* (3). The *ST_Intersects* predicate returns TRUE if the boundary of the first geometry intersects the interior of the second.

		b		
		Interior	Boundary	Exterior
a	Interior	*	*	*
	Boundary	T	*	*
	Exterior	*	*	*

Table 46. Matrix for *ST_Intersects* (4). The *ST_Intersects* predicate returns TRUE if the boundaries of either geometry intersect.

		b		
		Interior	Boundary	Exterior
a	Interior	*	*	*
	Boundary	*	T	*
	Exterior	*	*	*

For more information, see “*ST_Intersects*” on page 221.

EnvelopesIntersect

This function returns 1 (TRUE) if the envelopes of two geometries intersect. It is a convenience function that efficiently implements *ST_Intersects* (*ST_Envelope*(g1), *ST_Envelope*(g2)). For more information, see “*EnvelopesIntersect*” on page 158.

ST_Touches

ST_Touches returns 1 (TRUE) if none of the points common to both geometries intersect the interiors of both geometries. At least one geometry must be a linestring, polygon, multilinestring or multipolygon.

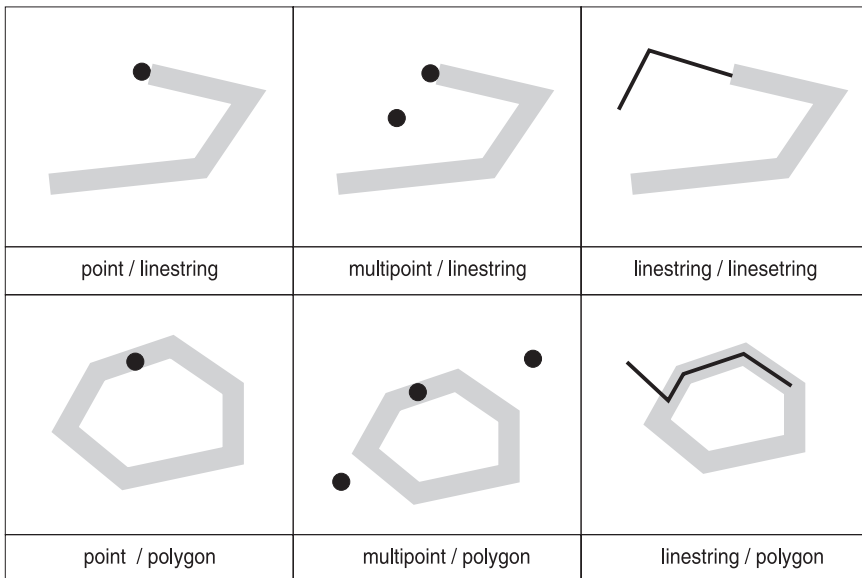


Figure 16. *ST_Touches*

The pattern matrices show that the *ST_Touches* predicate returns TRUE when the interiors of the geometry do not intersect, and the boundary of either geometry intersects the other’s interior or its boundary.

Table 47. Matrix for ST_Touches (1)

		b		
		Interior	Boundary	Exterior
a	Interior	F	T	*
	Boundary	*	*	*
	Exterior	*	*	*

Table 48. Matrix for ST_Touches (2)

		b		
		Interior	Boundary	Exterior
a	Interior	F	*	*
	Boundary	T	*	*
	Exterior	*	*	*

Table 49. Matrix for ST_Touches (3)

		b		
		Interior	Boundary	Exterior
a	Interior	F	*	*
	Boundary	*	T	*
	Exterior	*	*	*

For more information, see “ST_Touches” on page 264.

ST_Overlaps

ST_Overlaps compares two geometries of the same dimension. It returns 1 (TRUE) if their intersection set results in a geometry different from both, but that has the same dimension.

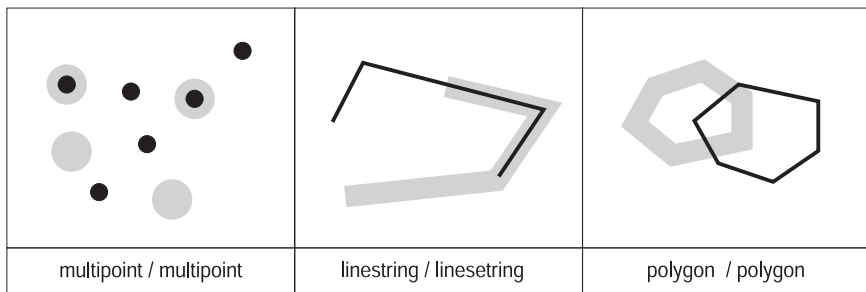


Figure 17. ST_Overlaps

The pattern matrix in Table 50 on page 139 applies to polygon/polygon, multipoint/multipoint and multipolygon/multipolygon overlays. For these combinations the overlay predicate returns TRUE if the interior of both geometries intersect the others interior and exterior.

Table 50. Matrix for ST_Overlaps (1)

		b		
		Interior	Boundary	Exterior
a	Interior	T	*	T
	Boundary	*	*	*
	Exterior	T	*	*

The pattern matrix in Table 51 applies to linestring/linestring and multilinestring/multilinestring overlays. In this case the intersection of the geometries must result in a geometry that has a dimension of 1 (another linestring). If the dimension of the intersection of the interiors is 1, the ST_Overlaps predicate would return FALSE, however the ST_Crosses predicate would return TRUE.


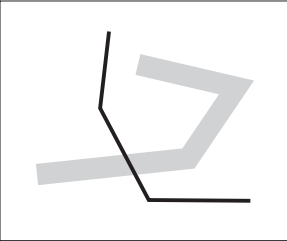
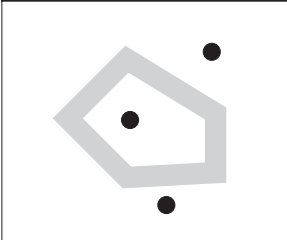
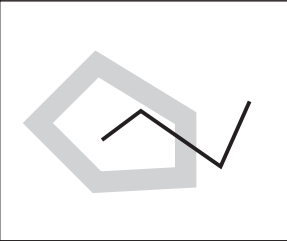
Table 51. Matrix for ST_Overlaps (2)

		b		
		Interior	Boundary	Exterior
a	Interior	1	*	T
	Boundary	*	*	*
	Exterior	T	*	*

For more information, see “ST_Overlaps” on page 246.

ST_Crosses

ST_Crosses returns 1 (TRUE) if the intersection results in a geometry whose dimension is one less than the maximum dimension of the two source geometries and the intersection set is interior to both source geometries. ST_Crosses returns 1 (TRUE) for only multipoint/polygon, multipoint/linestring, linestring/linestring, linestring/polygon, and linestring/multipolygon comparisons.

	
multipoint / linestring	Linesetring / Linestring
	
multipoint / polygon	linestring / polygon

The pattern matrix in Table 52 applies to multipoint/linestring, multipoint/multilinestring, multipoint/polygon, multipoint/multipolygon, linestring/polygon, linestring/multipolygon. The matrix states that the interiors must intersect and that the interior of the primary (geometry *a*) must intersect the exterior of the secondary (geometry *b*).

Table 52. Matrix for *ST_Crosses* (1)

		b		
		Interior	Boundary	Exterior
a	Interior	T	*	T
	Boundary	*	*	*
	Exterior	*	*	*

The pattern matrix in Table 53 applies to the linestring/linestring, linestring/multilinestring and multilinestring/multilinestring. The matrix states that the dimension of the intersection of the interiors must be 0 (intersect at a point). If the dimension of this intersection is 1 (intersect at a linestring), the *ST_Crosses* predicate returns FALSE; however, the *ST_Overlaps* predicate returns TRUE.

Table 53. Matrix for *ST_Crosses* (2)

		b		
		Interior	Boundary	Exterior
a	Interior	0	*	*
	Boundary	*	*	*
	Exterior	*	*	*

For more information, see “ST_Crosses” on page 193.

ST_Within

ST_Within returns 1 (TRUE) if the first geometry is completely within the second geometry. ST_Within returns the exact opposite result of ST_Contains.


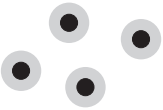


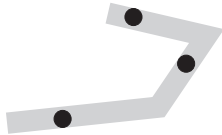




		
point / multipoint	multipoint / multipoint	multipoint / polygon
		
point / linestring	multipoint / linestring	linestring / linestring
		
point / polygon	linestring / polygon	polygon / polygon

Figure 18. Within

The ST_Within predicate pattern matrix states that the interiors of both geometries must intersect, and that the interior and boundary of the primary geometry (geometry *a*) must not intersect the exterior of the secondary (geometry *b*).

Table 54. Matrix for ST_Within

		b		
		Interior	Boundary	Exterior
a	Interior	T	*	F
	Boundary	*	*	F
	Exterior	*	*	*

For more information, see “ST_Within” on page 267.

ST_Contains

ST_Contains returns 1 (TRUE) if the second geometry is completely contained by the first geometry. The ST_Contains predicate returns the exact opposite result of the ST_Within predicate.

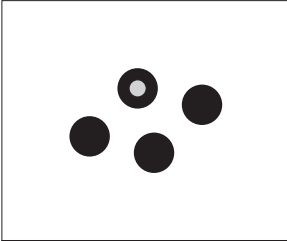
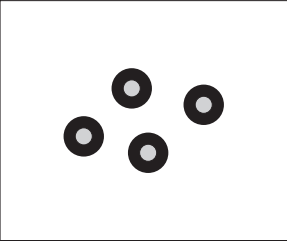
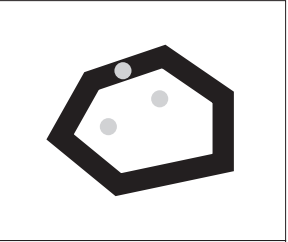
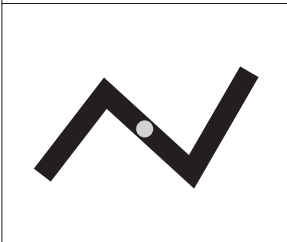
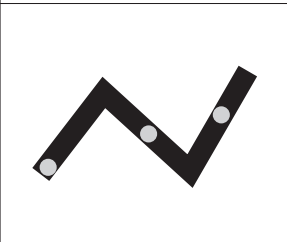
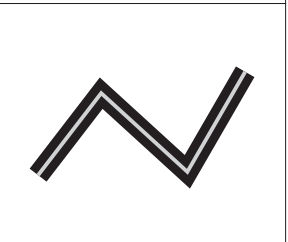
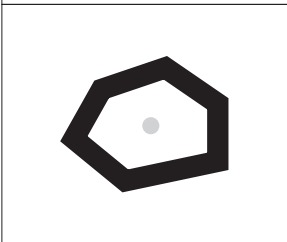


		
multipoint / point	multipoint / multipoint	polygon / multipoint
		
linestring / point	linestring / multipoint	linestring / linestring
		
polygon / point	polygon / linestring	polygon / polygon

Figure 19. ST_Contains

The pattern matrix of the ST_Contains predicate states that the interiors of both geometries must intersect and that the interior and boundary of the secondary (geometry *b*) must not intersect the exterior of the primary (geometry *a*).

Table 55. Matrix for ST_Contains

		b		
		Interior	Boundary	Exterior
a	Interior	T	*	*
	Boundary	*	*	*
	Exterior	F	F	*

For more information, see “ST_Contains” on page 187.

ST_Relate

The ST_Relate function compares two geometries and returns 1 (TRUE) if the geometries meet the conditions specified by the DE-91M pattern matrix string; otherwise, the function returns 0 (FALSE). For more information, see “ST_Relate” on page 258.

ST_Distance

The ST_Distance function reports the minimum distance separating two disjoint features. If the features are not disjoint, the function will report a 0 minimum distance.

For example, ST_Distance could report the shortest distance an aircraft must travel between two locations. Figure 20 illustrates this information.



Figure 20. Minimum distance between two cities. ST_Distance can take the coordinates for the locations of Los Angeles and Chicago as input, and return a value denoting the minimum distance between these locations.

For more information, see “ST_Distance” on page 200.

Functions that generate new geometries from existing ones

DB2 Spatial Extender provides predicates and transformation functions that generate new geometries from existing ones.

ST_Intersection

The `ST_Intersection` function returns the intersection set of two geometries. The intersection set is always returned as a collection that is the minimum dimension of the source geometries. For example, for a linestring that intersects a polygon, the intersection function returns a multilinestring comprised of that portion of the linestring common to the interior and boundary of the polygon. The multilinestring contains more than one linestring if the source linestring intersects the polygon with two or more discontinuous segments. If the geometries do not intersect or if the intersection results in a dimension less than both of the source geometries, an empty geometry is returned.

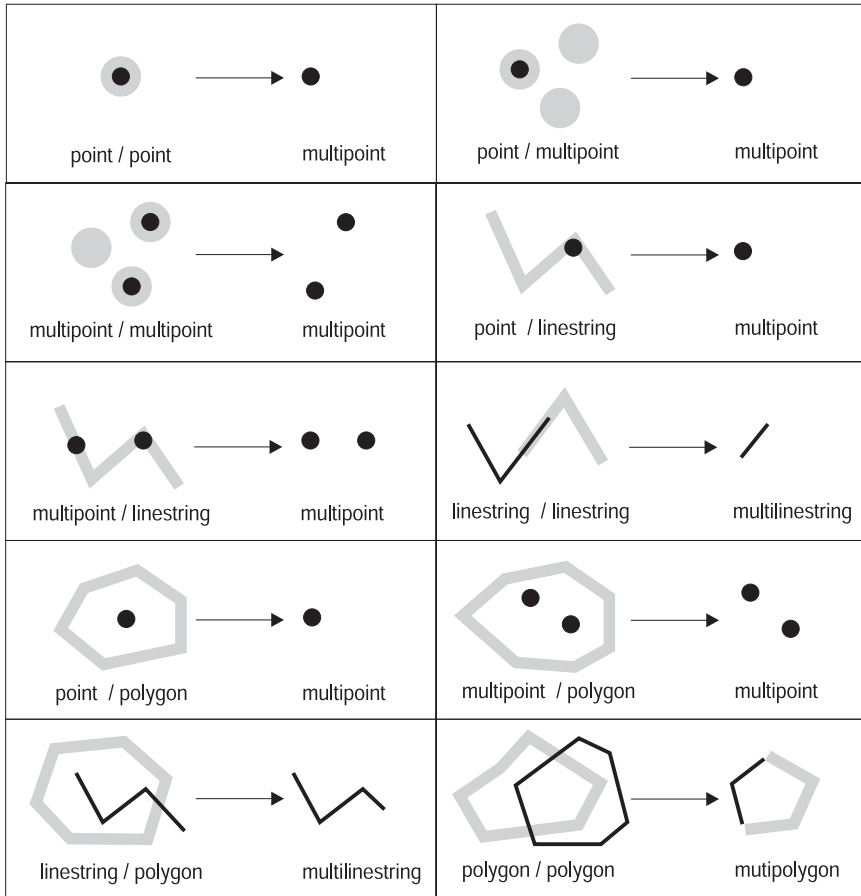


Figure 21. *ST_Intersection*. Examples of the *ST_Intersection* function.

For more information, see “*ST_Intersection*” on page 219.

ST_Difference

The *ST_Difference* function returns the portion of the primary geometry that is not intersected by the secondary geometry. This is the logical AND NOT of space. The *ST_Difference* function operates only on geometries of like dimension and returns a collection that has the same dimension as the source geometries. In the event that the source geometries are equal, an empty geometry is returned.

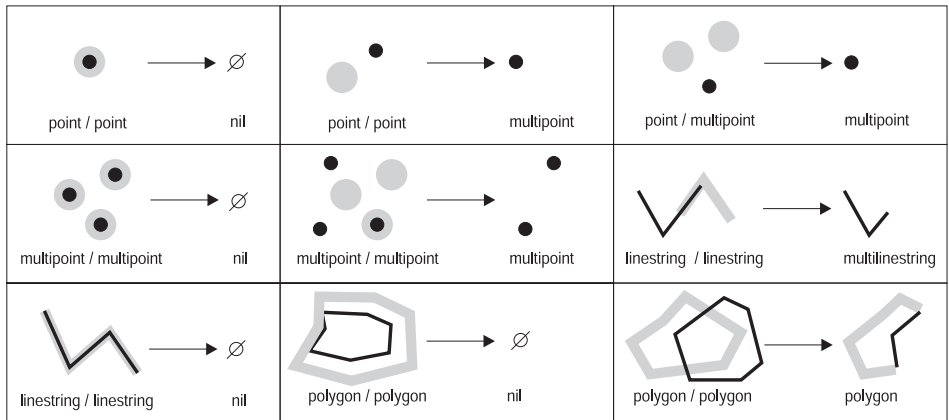


Figure 22. *ST_Difference*

For more information, see “*ST_Difference*” on page 195.

ST_Union

The *ST_Union* function returns the union set of two geometries. This is the logical OR of space. The source geometries must be of like dimension. *ST_Union* always returns the result as a collection.

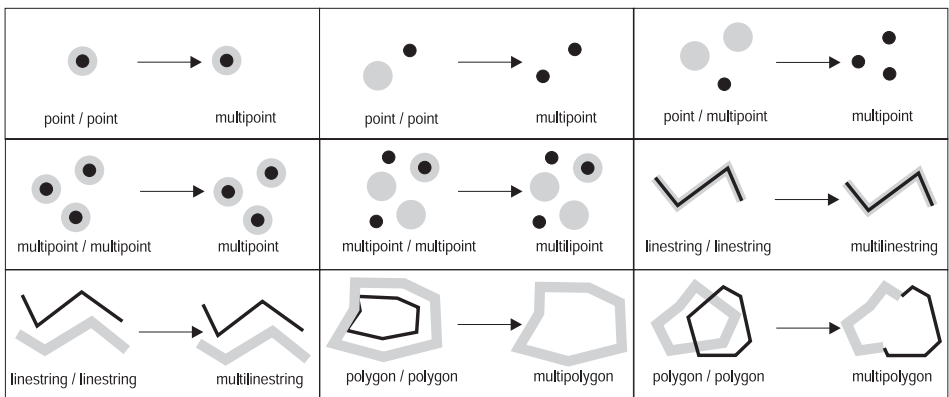


Figure 23. *ST_Union*

For more information, see “*ST_Union*” on page 266.

ST_Buffer

The *ST_Buffer* function generates a geometry by encircling a geometry at a specified distance. A polygon results when a primary geometry is buffered or whenever the elements of a collection are close enough such that all of the buffer polygons overlap. However, when there is enough separation between

the elements of a buffered collection, individual buffer polygons will result, in which case the `ST_Buffer` function returns a multipolygon.

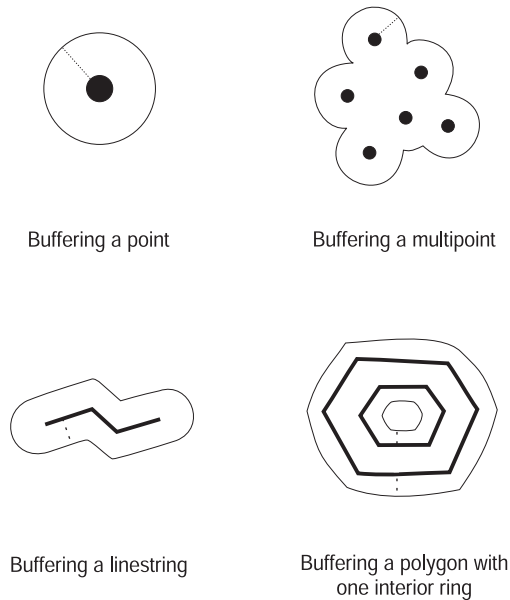


Figure 24. `ST_Buffer`

The `ST_Buffer` function accepts both positive and negative distance, however, only geometries with a dimension of two (polygons and multipolygons) apply a negative buffer. The absolute value of the buffer distance is used whenever the dimension of the source geometry is less than 2 (all geometries that are not polygon or multipolygon).

In general, for exterior rings, positive buffer distances generate polygon rings that are away from the center of the source geometry; negative buffer distances generate polygon or multipolygon rings toward the center. For interior rings of a polygon or multipolygon, a positive buffer distance generates a buffer ring toward the center, and a negative buffer distance generates a buffer ring away from the center.

The buffering process merges polygons that overlap. Negative distances greater than one half the maximum interior width of a polygon result in an empty geometry.

For more information, see “`ST_Buffer`” on page 184.

LocateAlong

For geometries that have measures, the location of a particular measure can be found with the `LocateAlong` function. `LocateAlong` returns the location as a multipoint. If the source geometry's dimension is 0 (for example, a point and a multipoint), an exact match is required, and those points having a matching measure value are returned as a multipoint. However, for source geometries whose dimension is greater than 0, the location is interpolated. For example, if the measure value entered is 5.5 and the measures on vertices of a linestring are a respective 3, 4, 5, 6, and 7, the interpolated point that falls exactly halfway between the vertices with measure values 5 and 6 is returned.

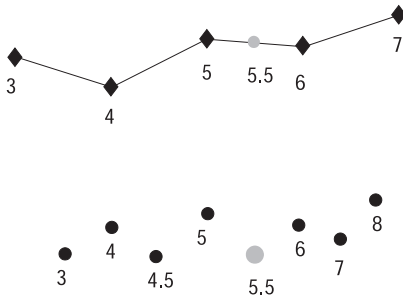


Figure 25. `LocateAlong`

For more information, see “`LocateAlong`” on page 164.

LocateBetween

The `LocateBetween` function returns either the set of paths or locations that lie between two measure values from a source geometry that has measures. If the source geometry's dimension is 0, `LocateBetween` returns a multipoint containing all points whose measures lie between the two source measures. For source geometries whose dimension is greater than 0, `LocateBetween` returns a multilinestring if a path can be interpolated; otherwise `LocateBetween` returns a multipoint containing the point locations. An empty point is returned whenever `LocateBetween` cannot interpolate a path or find a location between the measures. `LocateBetween` performs an inclusive search of the geometries; therefore the geometries' measures must be greater than or equal to the *from* measure and less than or equal to the *to* measure.

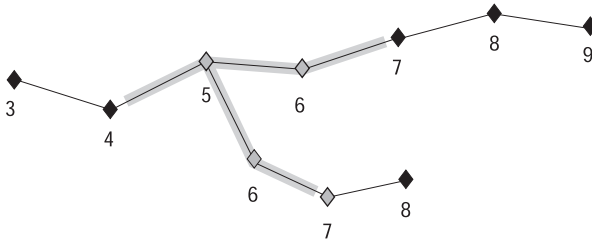


Figure 26. *LocateBetween*

For more information, see “LocateBetween” on page 166.

ST_ConvexHull

The `ST_ConvexHull` function returns the convex hull polygon of any geometry that has at least three vertices forming a convex. If vertices of the geometry do not form a convex, `ST_ConvexHull` returns a null.

`ST_ConvexHull` is often the first step in tessellation used to create a TIN network from a set of points.

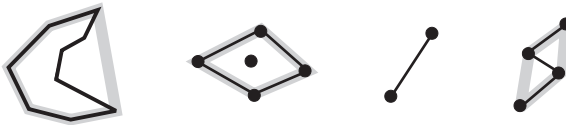


Figure 27. *ST_ConvexHull*

For more information, see “`ST_ConvexHull`” on page 189.

ST_Polygon

Generates a polygon from a linestring. For more information, see “`ST_Polygon`” on page 257.

Functions that convert the format of a geometry’s values

DB2 Spatial Extender supports three GIS data exchange formats:

- Well-known text representation
- Well-known binary representation
- ESRI binary shape representation

Well-known text representation

DB2 Spatial Extender has several functions that generate geometries from text descriptions.

ST_WKTTToSQL

Creates a geometry from a text representation of any geometry type. No spatial reference system identifier should be specified. For more information, see “`ST_WKTTToSQL`” on page 270.

ST_GeomFromText

Creates a geometry from a text representation of any geometry type. A spatial reference system identifier must be specified. For more information, see “ST_GeometryFromText” on page 207.

ST_PointFromText

Creates a point from a point text representation. For more information, see “ST_PointFromText” on page 249.

ST_LineFromText

Creates a linestring from a linestring text representation. For more information, see “ST_LineFromText” on page 232.

ST_PolyFromText

Creates a polygon from a polygon text representation. For more information, see “ST_PolyFromText” on page 254.

ST_MPointFromText

Creates a multipoint from a multipoint representation. For more information, see “ST_MPointFromText” on page 238.

ST_MLineFromText

Creates a multilinestring from a multilinestring representation. For more information, see “ST_MLineFromText” on page 235.

ST_MPolyFromText

Creates a multipolygon from a multipolygon representation. For more information, see “ST_MPolyFromText” on page 240.

The text representation is an ASCII string. It permits geometry to be exchanged in ASCII text form. These functions do not require the definition of any special program structures to map a binary representation. So, they can be used in either a 3GL or 4GL program.

The ST_AsText function converts an existing geometry value into text representation. For more information, see “ST_AsText” on page 181.

For a detailed description of well-known text representations, see “The OGC well-known text representations” on page 285.

Well-known binary representation

The DB2 Spatial Extender has several functions that generate geometries from well-known binary (WKB) representations.

ST_WKBTToSQL

Creates a geometry from a well-known binary representation of any geometry type. No spatial reference system identifier should be specified. For more information, see “ST_WKBTToSQL” on page 268.

ST_GeomFromWKB

Creates a geometry from a well-known binary representation of any geometry type. A spatial reference system identifier must be specified. For more information, see “ST_GeomFromWKB” on page 209.

ST_PointFromWKB

Creates a point from a well-known binary representation of a point. For more information, see “ST_PointFromWKB” on page 250.

ST_LineFromWKB

Creates a linestring from a well-known binary representation of a linestring. For more information, see “ST_LineFromWKB” on page 233.

ST_PolyFromWKB

Creates a polygon from a well-known binary representation of a polygon. For more information, see “ST_PolyFromWKB” on page 255.

ST_MPointFromWKB

Creates a multipoint from a well-known binary representation of a multipoint. For more information, see “ST_MPointFromWKB” on page 239.

ST_MLineFromWKB

Creates a multilinestring from a well-known binary representation of a multilinestring. For more information, see “ST_MLineFromWKB” on page 236.

ST_MPolyFromWKB

Creates a multipolygon from a well-known binary representation of a multipolygon. For more information, see “ST_MPolyFromWKB” on page 241.

The well-known binary representation is a contiguous stream of bytes. It permits geometry to be exchanged between an ODBC client and an SQL database in binary form. These geometry functions require the definition of C structures to map the binary representation. So, they are intended for use within a 3GL program, and are not suited to a 4GL environment.

The ST_AsBinary function converts an existing geometry value into well-known binary representation. For more information, see “ST_AsBinary” on page 180.

For a detailed description of well-known binary representations, see “The OGC well-known binary (WKB) representations” on page 290.

ESRI shape representation

DB2 Spatial Extender has several functions that generate geometries from an ESRI shape representation. The ESRI shape representation supports Z

coordinates and measures in addition to the two-dimensional representations supported by the text and well-known binary representations.

ShapeToSQL

Creates a geometry from a shape of any geometry type. No spatial reference system identifier should be specified. For more information, see “ShapeToSQL” on page 176.

GeometryFromShape

Creates a geometry from a shape of any geometry type. A spatial reference system identifier must be specified. For more information, see “GeometryFromShape” on page 157.

PointFromShape

Creates a point from a point shape. For more information, see “PointFromShape” on page 173.

LineFromShape

Creates a linestring from a polyline shape. For more information, see “LineFromShape” on page 162.

PolyFromShape

Creates a polygon from a polyline shape. For more information, see “PolyFromShape” on page 174.

MPointFromShape

Creates a multipoint from a multipoint shape. For more information, see “MPointFromShape” on page 171.

MLineFromShape

Creates a multilinestring from a multipart polyline shape. For more information, see “MLine FromShape” on page 169.

MPolyFromShape

Creates a multipolygon from a multipart polygon shape. For more information, see “MPolyFromShape” on page 172.

The general syntax of these functions is the same. The first argument is the shape representation that is entered as a BLOB data type. The second argument is the spatial reference identifier that will be assigned to the geometry. For example, the GeometryFromShape function has the following syntax:

```
GeometryFromShape(shapegeometry, SRID)
```

To map the binary representation, these shape functions require the definition of a C structures. So, they are intended for use within a 3GL program, and are not suited to a 4GL environment.

The AsBinaryShape function converts a geometry value into an ESRI shape representation. For more information, see “AsBinaryShape” on page 156.

For a detailed description of shape representations, see “The ESRI shape representations” on page 294.

Chapter 14. Spatial functions for SQL queries

This chapter lists the available functions that you can invoke when you query spatial data. Each function is described in a section that shows you the syntax, return type, and code examples. Some of the examples in this chapter include a CREATE TABLE statement in which one or more columns are defined as spatial columns.

The following considerations apply to spatial functions:

- The examples in this chapter are qualified with the library name db2gse. Instead of explicitly qualifying each spatial function and type with db2gse, you can set the function path to include db2gse.
- Before you can insert data into a spatial column:
 - You might need to increase the udf_mem_sz parameter. The suggested initial setting is 2048. If 2048 is inadequate, increase the udf_mem_sz parameter in increments of 256.
 - You must register it as a layer. For more information about registering a spatial column as a layer, see “Chapter 4. Defining spatial columns, registering them as layers, and enabling a geocoder to maintain them” on page 31.

AsBinaryShape

AsBinaryShape takes a geometry object and returns a BLOB.

Syntax

```
db2gse.AsBinaryShape(g db2gse.ST_Geometry)
```

Return type

BLOB(1m)

Examples

The following code fragment illustrates how the AsBinaryShape function converts the zone polygons of the SENSITIVE_AREAS table into shape polygons. These shape polygons are passed to the application's draw_polygon function for display.

```
/* Create the SQL expression. */
strcpy(sqlstmt, "select db2gse.AsBinaryShape (zone) from SENSITIVE_AREAS
where db2gse.EnvelopesIntersect(zone, db2gse.PolyFromShape(cast(? as blob(1m)),
db2gse.coordref(..srid(0)))");

/* Prepare the SQL statement. */
SQLPrepare(hstmt, (UCHAR *)sqlstmt, SQL_NTS);

/* Set the pcbvalue1 length of the shape. */
pcbvalue1 = blob_len;

/* Bind the shape parameter */
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_BLOB, blob_len,
0, shape, blob_len, &pcbvalue1);

/* Execute the query */
rc = SQLExecute(hstmt);

/* Assign the results of the query (the Zone polygons) to the
  fetched_binary variable. */
SQLBindCol (hstmt, 1, SQL_C_Binary, fetched_binary, 100000, &ind_blob);

/* Fetch each polygon within the display window and display it. */

while(SQL_SUCCESS == (rc = SQLFetch(hstmt)))
  draw_polygon(fetched_binary);
```

GeometryFromShape

GeometryFromShape takes a shape and a spatial reference system identity to return a geometry object.

Syntax

```
db2gse.GeometryFromShape(ShapeGeometry Blob(1M), cr db2gse.coordref)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The following C code fragment contains ODBC functions embedded with DB2 Spatial Extender SQL functions that insert data into the LOTS table.

The LOTS table was created with two columns: the LOT_ID column, which uniquely identifies each lot, and the LOT polygon column, which contains the geometry of each lot.

```
CREATE TABLE LOTS ( lot_id integer,
                    lot      db2gse.ST_MultiPolygon);
```

The GeometryFromShape function converts shapes into DB2 Spatial Extender geometry. The entire INSERT statement is copied into shp_sql. The INSERT statement contains parameter markers to accept the LOT_ID data and the LOT data dynamically.

```
/* Create the SQL insert statement to populate the lot id and the
   lot multipolygon. The question marks are parameter markers that
   indicate the lot_id and lot values that will be retrieved at
   runtime. */
strcpy (shp_sql,"insert into LOTS (lot_id, lot) values (?,
db2gse.GeometryFromShape (cast(? as blob(1m)), db2gse.coordref(..srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the integer key value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, 0, 0, &lot_id, 0, &pcbvalue1);

/* Bind the shape to the second parameter. */
pcbvalue2 = blob_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
SQL_BLOB, blob_len, 0, shape_blob, blob_len, &pcbvalue2);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

EnvelopesIntersect

EnvelopesIntersect returns 1 (TRUE) if the envelopes of two geometries intersect; otherwise it returns 0 (FALSE).

Syntax

```
db2gse.EnvelopesIntersect(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Integer

Examples

The `get_window` function retrieves the display window's coordinates from the application. The window parameter is actually a polygon shape structure containing a string of coordinates that represent the display polygon. The `PolyFromShape` function converts the display window shape into a DB2 Spatial Extender polygon that the `EnvelopesIntersect` function uses as its intersection envelope. All SENSITIVE_AREAS zone polygons that intersect the interior or boundary of the display window are returned. Each polygon is fetched from the result set and passed to the `draw_polygon` function.

```
/* Get the display window coordinates as a polygon shape.
get_window(&window)
```

```
/* Create the SQL expression. The db2gse.EnvelopesIntersect function
   will be used to limit the result set to only those zone polygons
   that intersect the envelope of the display window. */
strcpy(sqlstmt, "select db2gse.AsBinaryShape(zone) from SENSITIVE_AREAS where
db2gse.EnvelopesIntersect (zone, db2gse.PolyFromShape(cast(? as blob(1m)),
db2gse.coordref()..srid(0)))");
```

```
/* Set blob_len to the byte length of a 5 point shape polygon. */
blob_len = 128;
```

```
/* Prepare the SQL statement. */
SQLPrepare(hstmt, (UCHAR *)sqlstmt, SQL_NTS);
```

```
/* Set the pcbvalue1 to the window shape */
pcbvalue1 = blob_len;
```

```
/* Bind the shape parameter */
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_BLOB,
blob_len, 0, window, blob_len, &pcbvalue1);
```

```
/* Execute the query */
rc = SQLExecute(hstmt);
```

```
/* Assign the results of the query, (the Zone polygons) to the
   fetched_binary variable. */
SQLBindCol (hstmt, 1, SQL_C_Binary, fetched_binary, 100000, &ind_blob);
```

```
/* Fetch each polygon within the display window and display it. */  
while(SQL_SUCCESS == (rc = SQLFetch(hstmt))  
      draw_polygon(fetched_binary);
```

Is3d

Is3d takes a geometry object and returns 1 (TRUE) if the object has 3D coordinates; otherwise, it returns 0 (FALSE).

Syntax

```
db2gse.Is3d(g db2gse.ST_Geometry)
```

Return type

Integer

Examples

The following CREATE TABLE statement creates the THREED_TEST table, which has two columns: the GID column of type integer and the G1 geometry column.

```
CREATE TABLE THREED_TEST (gid smallint, g1 db2gse.ST_Geometry)
```

The INSERT statements insert two points into the THREED_TEST table. The first point does not contain Z coordinates, while the second does.

```
INSERT INTO THREED_TEST  
VALUES(1, db2gse.ST_PointFromText('point (10 10)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO THREED_TEST  
VALUES (2, db2gse.ST_PointFromText('point z (10.92 10.12 5)',  
db2gse.coordref()..srid(0)))
```

The following SELECT statement lists the contents of the GID column with the results of the Is3d function. The function returns a 0 for the first row, which does not have Z coordinates, and a 1 for the second row, which does have Z coordinates.

```
SELECT gid, db2gse.Is3d (g1) "Is it 3d?" FROM THREED_TEST
```

The following result set is returned.

gid	Is it 3d?
1	0
2	1

IsMeasured

IsMeasured takes a geometry object and returns 1 (TRUE) if the object has measures; otherwise it returns 0 (FALSE).

Syntax

```
db2gse.IsMeasured(g db2gse.ST_Geometry)
```

Return type

Integer

Examples

The following CREATE TABLE statement creates the MEASURE_TEST table, which has two columns. The GID column uniquely identifies the rows, and the G1 column stores the point geometries.

```
CREATE TABLE MEASURE_TEST (gid smallint, g1 db2gse.ST_Geometry)
```

The following INSERT statements insert two records into the MEASURE_TEST table. The first record stores a point that does not have a measure. The second record's point does have a measure.

```
INSERT INTO MEASURE_TEST  
VALUES(1, db2gse.ST_PointFromText('point (10 10)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO MEASURE_TEST  
VALUES (2, db2gse.ST_PointFromText('point m (10.92 10.12 5)',  
db2gse.coordref()..srid(0)))
```

The following SELECT statement and corresponding result set show the GID column along with the results of the IsMeasured function. The IsMeasured function returns a 0 for the first row because the point does not have a measure. It returns a 1 for the second row because the point does have measures.

```
SELECT gid, db2gse.IsMeasured (g1) "Has measures?" FROM MEASURE_TEST  
gid      Has measures
```

```
-----  
1          0  
2          1
```

LineFromShape

LineFromShape takes a shape of type point and a spatial reference system identity and returns a linestring.

Syntax

```
db2gse.Line FromShape(ShapeLineString Blob(1M), cr db2gse.coordref)
```

Return type

```
db2gse.ST_LineString
```

Examples

The following code fragment populates the SEWERLINES table with the unique id, size class, and geometry of each sewer line.

The CREATE TABLE statement creates the SEWERLINES table, which has three columns. The first column, SEWER_ID, uniquely identifies each sewer line. The second column, CLASS, of type integer identifies the type of sewer line, which is generally associated with a line's capacity. The third column, SEWER, of type linestring stores the sewer line's geometry.

```
CREATE TABLE SEWERLINES (sewer_id integer, class integer,  
                          sewer db2gse.ST_LineString);
```

```
/* Create the SQL insert statement to populate the sewer_id, size class and  
   the sewer linestring. The question marks are parameter markers that  
   indicate the sewer_id, class and sewer geometry values that will be  
   retrieved at runtime. */
```

```
strcpy (shp_sql,"insert into sewerlines (sewer_id,class,sewer)  
values (?,?, db2gse.Line FromShape (cast(? as blob(1m)),  
db2gse.coordref()..srid(0)))");
```

```
/* Allocate memory for the SQL statement handle and associate the  
statement handle with the connection handle. */
```

```
rc = SQLAllocStmt (handle, &hstmt);
```

```
/* Prepare the SQL statement for execution. */
```

```
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);
```

```
/* Bind the integer key value to the first parameter. */
```

```
pcbvalue1 = 0;
```

```
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,  
SQL_INTEGER, 0, 0, &sewer_id, 0, &pcbvalue1);
```

```
/* Bind the integer class value to the second parameter. */
```

```
pcbvalue2 = 0;
```

```
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_SLONG,  
SQL_INTEGER, 0, 0, &sewer_class, 0, &pcbvalue2);;
```

```
/* Bind the shape to the third parameter. */
```

```
pcbvalue3 = blob_len;
```

```
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
```



```
        SQL_BLOB, blob_len, 0, sewer_shape, blob_len, &pcbvalue3);  
/* Execute the insert statement. */  
rc = SQLExecute (hstmt);
```

LocateAlong

LocateAlong takes a geometry object and a measure to return as a multipoint the set of points found at the measure.

Syntax

```
db2gse.LocateAlong(g db2gse.ST_Geometry, a distance Double)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The following CREATE TABLE statement creates the LOCATEALONG_TEST table. LOCATEALONG_TEST has two columns: the GID column, which uniquely identifies each row, and the G1 geometry column, which stores sample geometry.

```
CREATE TABLE LOCATEALONG_TEST (gid integer, g1 db2gse.ST_Geometry)
```

The following INSERT statements insert two rows. The first is a multilinestring; the second is a multipoint.

```
INSERT INTO db2gse.LOCATEALONG_TEST VALUES(
1, db2gse.ST_MLineFromText('multilinestring m ((10.29 19.23 5,23.82 20.29 6,
30.19 18.47 7,
45.98 20.74 8), (23.82 20.29 6,30.98 23.98 7,42.92 25.98 8))',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO db2gse.LocateAlong_TEST VALUES(
2, db2gse.ST_MPointFromText('multipoint m (10.29 19.23 5,23.82 20.29 6,
30.19 18.47 7,45.98 20.74 8,23.82 20.29 6,30.98 23.98 7,42.92 25.98)',
db2gse.coordref()..srid(0)))
```

In the following SELECT statement and the corresponding result set, the LocateAlong function is directed to find points whose measure is 6.5. The first row returns a multipoint containing two points. However, the second row returned an empty point. For linear features (geometry with a dimension greater than 0), LocateAlong can interpolate the point; however, for multipoints, the target measure must match exactly.

```
SELECT gid, CAST(db2gse.ST_AsText(db2gse.LocateAlong (g1,6.5)) AS
varchar(96))
"Geometry"
FROM LOCATEALONG_TEST
```

```
GID          Geometry
-----
1 MULTIPOINT M ( 27.01000000 19.38000000 6.50000000, 27.40000000
22.14000000 6.50000000)
2 POINT EMPTY
```

```
2 record(s) selected.
```

In the following SELECT statement and the corresponding result set, the LocateAlong function returns multipoints for both rows. The target measure of 7 matches the measures in both the multilinestring and multipoint source data.

```
SELECT gid,CAST(db2gse.ST_AsText(db2gse.LocateAlong (g1,7)) AS varchar(96))
      "Geometry"
FROM LOCATEALONG_TEST
```

```
GID      Geometry
```

```
-----
      1 MULTIPOINT M ( 30.19000000 18.47000000 7.00000000, 30.98000000
23.98000000 7.00000000)
      2 MULTIPOINT M ( 30.19000000 18.47000000 7.00000000, 30.98000000
23.98000000 7.00000000)
```

```
2 record(s) selected.
```

LocateBetween

LocateBetween takes a geometry object and two measure locations and returns a geometry that represents the set of disconnected paths between the two measure locations.

Syntax

```
db2gse.LocateBetween(g db2gse.ST_Geometry, a distance Double,  
another distance Double)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The following CREATE TABLE statement creates the LOCATEBETWEEN_TEST table. LOCATEBETWEEN_TEST has two columns: the GID column, which uniquely identifies each row, and the G1 multilinestring column, which stores the sample geometry.

```
CREATE TABLE LOCATEBETWEEN_TEST (gid integer, g1 db2gse.ST_Geometry)
```

The following INSERT statements insert two rows into the LOCATEBETWEEN_TEST table. The first row is a multilinestring, and the second is a multipoint.

```
INSERT INTO db2gse.LOCATEBETWEEN_TEST  
VALUES(1,db2gse.ST_MLineFromText('multilinestring m ((10.29 19.23 5,  
23.82 20.29 6, 30.19 18.47 7,45.98 20.74 8),  
(23.82 20.29 6,30.98 23.98 7,  
42.92 25.98 8))',  
db2gse.coordref()..srid(0)))
```

```
INSERT INTO db2gse.LOCATEBETWEEN_TEST  
VALUES(2, db2gse.ST_MPointFromText('multipoint m (10.29 19.23 5,23.82 20.29 6,  
30.19 18.47 7,45.98 20.74 8,23.82 20.29 6,  
30.98 23.98 7,42.92 25.98 8)',  
db2gse.coordref()..srid(0)))
```

The following SELECT statement and corresponding result set show how the LocateBetween function locates measures lying between measures 6.5 and 7.5 inclusive. The first row returns a multilinestring containing several linestrings. The second row returns a multipoint because the source data was multipoint. When the source data has a dimension of 0 (point or multipoint), an exact match is required.

```
SELECT gid, CAST(db2gse.ST_AsText(db2gse.LocateBetween (g1,6.5,7.5))  
AS varchar(96)) "Geometry"  
FROM LOCATEBETWEEN_TEST
```

GID	Geometry
1	MULTILINESTRING M (27.01000000 19.38000000 6.50000000, 31.19000000 18.47000000 7.00000000,38.09000000 19.61000000 7.50000000),(27.40000000 22.1400

```
0000 6.50000000, 30.98000000 23.98000000 7.00000000,36.95000000 24.98000000 7.5  
00000000) 2 MULTIPOINT M ( 30.19000000 18.47000000 7.00000000, 30.98000000 23.9  
8000000 7.00000000)
```

2 record(s) selected.

M

M takes a point and returns its measure.

Syntax

```
db2gse.M(p db2gse.ST_Point)
```

Return type

Double

Examples

The following CREATE TABLE statement creates the M_TEST table. M_TEST has two columns: the GID integer column, which uniquely identifies the row, and the PT1 point column, which stores the sample geometry.

```
CREATE TABLE M_TEST (gid integer, pt1 db2gse.ST_Point)
```

The following INSERT statements insert a row that contains a point with measures and row that contains a point without measures.

```
INSERT INTO db2gse.M_TEST
VALUES(1, db2gse.ST_PointFromText('point (10.02 20.01)',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO db2gse.M_TEST
VALUES(2, db2gse.ST_PointFromText('point zm(10.02 20.01 5.0 7.0)',
db2gse.coordref()..srid(0)))
```

In the following SELECT statement and the corresponding result set, the M function lists the measure values of the points. Because the first point does not have measures, the M function returns a NULL.

```
SELECT gid, db2gse.M (pt1) "The measure" FROM M_TEST
```

GID	The measure
1	-
2	+7.000000000000000E+000

2 record(s) selected.

MLine FromShape

MLine FromShape takes a shape of type multilinestring and a spatial reference system identity and returns a multilinestring.

Syntax

```
db2gse.MLineFromShape(ShapeMultiLineString Blob(1M), cr db2gse.coordref)
```

Return type

```
db2gse.ST_MultiLineString
```

Examples

The following code fragment populates the WATERWAYS table with a unique id, a name, and a water multilinestring.

The WATERWAYS table is created with the ID and NAME columns that identify each stream and river system stored in the table. The WATER column is a multilinestring because the river and stream systems are often an aggregate of several linestrings.

```
CREATE TABLE WATERWAYS (id          integer,
                        name        varchar(128),
                        water       db2gse.ST_MultiLineString);

/* Create the SQL insert statement to populate the id, name and
   multilinestring. The question marks are parameter markers that
   indicate the id, name and water values that will be retrieved at
   runtime. */
strcpy (shp_sql,"insert into WATERWAYS (id,name,water)
values (?,?, db2gse.MLineFromShape (cast(? as blob(1m)),
db2gse.coordref(..srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the integer id value to the first parameter. */

pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, 0, 0, &id, 0, &pcbvalue1);
/* Bind the varchar name value to the second parameter. */

pcbvalue2 = name_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, name_len, 0, &name, name_len, &pcbvalue2);

/* Bind the shape to the third parameter. */
pcbvalue3 = blob_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
```

```
        SQL_BLOB, blob_len, 0, water_shape, blob_len, &pcbvalue3);  
/* Execute the insert statement. */  
rc = SQLExecute (hstmt);
```

MPointFromShape

MPointFromShape takes a shape of type multipoint and a spatial reference system identity to return a multipoint.

Syntax

```
db2gse.MPointFromShape(ShapeMultiPoint (1M), srs db2gse.coordref)
```

Return type

```
db2gse.ST_MultiPoint
```

Examples

This code fragment populates a biologist's SPECIES_SITINGS table.

The SPECIES_SITINGS table is created with three columns. The species and genus columns uniquely identify each row while the sitings multipoint stores the locations of the species sitings.

```
CREATE TABLE SPECIES_SITINGS (species varchar(32),
                               genus varchar(32),
                               sitings db2gse.ST_MultiPoint);

/* Create the SQL insert statement to populate the species, genus and
   sitings. The question marks are parameter markers that indicate the
   name and water values that will be retrieved at runtime. */
strcpy (shp_sql,"insert into SPECIES_SITINGS (species,genus,sitings)
values (?,?, db2gse.MPointFromShape (cast(? as blob(1m)),
db2gse.coordref()?.srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the varchar species value to the first parameter. */
pcbvalue1 = species_len;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, species_len, 0, species, species_len, &pcbvalue1);

/* Bind the varchar genus value to the second parameter. */
pcbvalue2 = genus_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, genus_len, 0, name, genus_len, &pcbvalue2);

/* Bind the shape to the third parameter. */
pcbvalue3 = blob_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
SQL_BLOB, sitings_len, 0, sitings_shape, sitings_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

MPolyFromShape

MPolyFromShape takes a shape of type multipolygon and a spatial reference system identity to return a multipolygon.

Syntax

```
db2gse.MPolyFromShape(ShapeMultiPolygon Blob(1m), srs db2gse.coordref)
```

Return type

```
db2gse.ST_MultiPolygon
```

Examples

This code fragment populates the LOTS table.

The LOTS table stores the lot_id which uniquely identifies each lot, and the lot multipolygon that contains the lot line geometry.

```
CREATE TABLE LOTS (lot_id integer, lot db2gse.ST_MultiPolygon);

/* Create the SQL insert statement to populate the lot_id and lot. The
   question marks are parameter markers that indicate the lot_id and lot
   values that will be retrieved at runtime. */
strcpy (shp_sql,"insert into LOTS (lot_id,lot)
values (?, db2gse.MPolyFromShape (cast(? as blob(1m)),
db2gse.coordref()..srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the lot_id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
SQL_INTEGER, 0, 0, &lot_id, 0, &pcbvalue1);

/* Bind the lot shape to the second parameter. */
pcbvalue2 = lot_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
SQL_BLOB, lot_len, 0, lot_shape, lot_len, &pcbvalue2);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

PointFromShape

PointFromShape takes a shape of type point and a spatial reference system identity to return a point.

Syntax

```
db2gse.PointFromShape(db2gse.ShapePoint blob(1M), srs db2gse.coordref)
```

Return type

```
db2gse.ST_Point
```

Examples

The program fragment populates the HAZARDOUS_SITES table.

The hazardous sites are stored in the HAZARDOUS_SITES table created with the CREATE TABLE statement that follows. The location column, defined as a point, stores a location that is the geographic center of each hazardous site.

```
CREATE TABLE HAZARDOUS_SITES (site_id integer,
                               name      varchar(128),
                               location  db2gse.ST_Point);

/* Create the SQL insert statement to populate the site_id, name and
   location. The question marks are parameter markers that indicate the
   site_id, name and location values that will be retrieved at runtime. */
strcpy (shp_sql,"insert into HAZARDOUS_SITES (site_id, name, location)
values (?,?, db2gse.PointFromShape (cast(? as blob(1m)),
db2gse.coordref(..srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the site_id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
SQL_INTEGER, 0, 0, &site_id, 0, &pcbvalue1);

/* Bind the name varchar value to the second parameter. */
pcbvalue2 = name_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, 0, 0, name, 0, &pcbvalue2);

/* Bind the location shape to the third parameter. */
pcbvalue3 = location_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
SQL_BLOB, location_len, 0, location_shape, location_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

PolyFromShape

PolyFromShape takes a shape of type polygon and a spatial reference system identity to return a polygon.

Syntax

db2gse.PolyFromShape (ShapePolygon Blob(1M), srs db2gse.coordref)

Return type

db2gse.ST_Polygon

Examples

The program fragment populates the SENSITIVE_AREAS table. The question marks represent parameter markers for the id, name, size, type and zone values that will be retrieved at runtime.

The SENSITIVE_AREAS table contains several columns that describe the threatened institutions in addition to the zone column which stores the institution's polygon geometry.

```
CREATE TABLE SENSITIVE_AREAS (id          integer,
                               name        varchar(128),
                               size        float,
                               type        varchar(10),
                               zone        db2gse.ST_Polygon);
```

```
/* Create the SQL insert statement to populate the id, name, size, type and
   zone. The question marks are parameter markers that indicate the
   id, name, size, type and zone values that will be retrieved at runtime. */
strcpy (shp_sql, "insert into SENSITIVE_AREAS (id, name, size, type, zone)
values (?, ?, ?, ?, db2gse.PolyFromShape (cast(? as blob(1m)),
db2gse.coordref()..srid(0)))");
```

```
/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);
```

```
/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);
```

```
/* Bind the id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
SQL_INTEGER, 0, 0, &site_id, 0, &pcbvalue1);
/* Bind the name varchar value to the second parameter. */
pcbvalue2 = name_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, 0, 0, name, 0, &pcbvalue2);
```

```
/* Bind the size float to the third parameter. */
pcbvalue3 = 0;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT,
SQL_REAL, 0, 0, &size, 0, &pcbvalue3);
```

```
/* Bind the type varchar to the fourth parameter. */
pcbvalue4 = type_len;
rc = SQLBindParameter (hstmt, 4, SQL_PARAM_INPUT, SQL_C_CHAR,
    SQL_VARCHAR, type_len, 0, type, type_len, &pcbvalue4);

/* Bind the zone polygon to the fifth parameter. */
pcbvalue5 = zone_len;
rc = SQLBindParameter (hstmt, 5, SQL_PARAM_INPUT, SQL_C_BINARY,
    SQL_BLOB, zone_len, 0, zone_shp, zone_len, &pcbvalue5);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

ShapeToSQL

ShapeToSQL constructs a db2gse.ST_Geometry value given its well-known binary representation. The SRID value of 0 is automatically used.

Syntax

```
db2gse.ST_ShapeToSQL(ShapeGeometry blob(1M))
```

Return type

```
db2gse.ST_Geometry
```

Examples

The following C code fragment contains ODBC functions embedded with DB2 Spatial Extender SQL functions that insert data into the LOTS table. The LOTS table was created with two columns: the lot_id, which uniquely identifies each lot, and the lot multipolygon column, which contains the geometry of each lot.

```
CREATE TABLE lots (lot_id integer,
                   lot      db2gse.ST_MultiPolygon);
```

The ShapeToSQL function converts shapes into DB2 Spatial Extender geometry. The entire INSERT statement is copied into shp_sql. The INSERT statement contains parameter markers to accept the LOT_id and the lot data, dynamically.

```
/* Create the SQL insert statement to populate the lot id and the
   lot multipolygon. The question marks are parameter markers that
   indicate the lot_id and lot values that will be retrieved at
   run time. */
```

```
strcpy (shp_sql,"insert into lots (lot_id, lot) values(?,
db2gse.ShapeToSQL(cast(? as blob(1m)))");
```

```
/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
```

```
rc = SQLAllocStmt (handle, &hstmt);
```

```
/* Prepare the SQL statement for execution. */
```

```
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);
```

```
/* Bind the integer key value to the first parameter. */
```

```
pcbvalue1 = 0;
```

```
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, 0, 0, &lot_id, 0, &pcbvalue1);
```

```
/* Bind the shape to the second parameter. */
```

```
pcbvalue2 = blob_len;
```

```
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,  
    SQL_BLOB, blob_len, 0, shape_blob, blob_len, &pcbvalue2);  
  
/* Execute the insert statement. */  
  
rc = SQLExecute (hstmt);
```

ST_Area

ST_Area takes a polygon or multipolygon and returns its area.

Syntax

```
db2gse.ST_Area(s db2gse.ST_Surface)
```

Return type

Double

Examples

The city engineer needs a list of building areas. To obtain the list, a GIS technician selects the building ID and area of each building's footprint.

The building footprints are stored in the BUILDINGFOOTPRINTS table that was created with the following CREATE TABLE statement:

```
CREATE TABLE BUILDINGFOOTPRINTS (
    building_id integer,
    lot_id      integer,
    footprint   db2gse.ST_MultiPolygon);
```

To satisfy the city engineer's request, the technician uses the following SELECT statement to select the unique key, the building_id, and the area of each building footprint from the BUILDINGFOOTPRINTS table:

```
SELECT building_id, db2gse.ST_Area (footprint) "Area"
FROM BUILDINGFOOTPRINTS;
```

The SELECT statement returns the following result set:

building_id	Area
506	+1.407680000000000E+003
1208	+2.557590000000000E+003
543	+1.807860000000000E+003
178	+2.086710000000000E+003
.	.
.	.
.	.

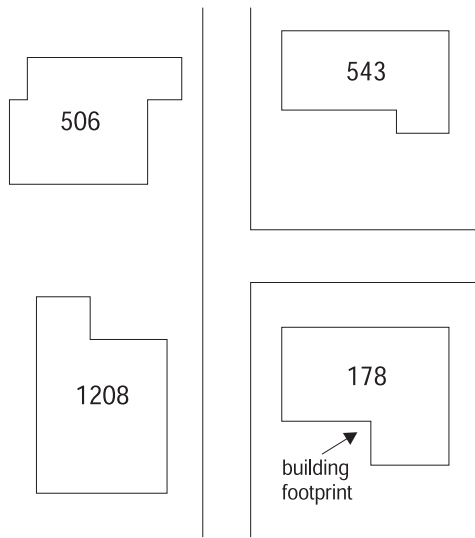


Figure 28. Using area to find a building footprint. Four of the building footprints labeled with their building ID numbers are displayed along side their adjacent street.

ST_AsBinary

ST_AsBinary takes a geometry object and returns its well-known binary representation.

Syntax

```
db2gse.ST_AsBinary(g db2gse.ST_Geometry)
```

Return type

BLOB(1m)

Examples

The following code fragment illustrates how the ST_AsBinary function converts the footprint multipolygons of the BUILDINGFOOTPRINTS table into WKB multipolygons. These multipolygons are passed to the application's draw_polygon function for display.

```
/* Create the SQL expression. */
strcpy(sqlstmt, "select db2gse.ST_AsBinary (footprint) from BUILDINGFOOTPRINTS
where db2gse.EnvelopesIntersect(footprint, db2gse.ST_PolyFromWKB
(cast(? as blob(1m)),
db2gse.coordref()..srid(0)))");

/* Prepare the SQL statement. */
SQLPrepare(hstmt, (UCHAR *)sqlstmt, SQL_NTS);

/* Set the pcbvalue1 length of the shape. */
pcbvalue1 = blob_len;

/* Bind the shape parameter */
SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_BINARY, SQL_BLOB, blob_len,
0, shape, blob_len, &pcbvalue1);

/* Execute the query */
rc = SQLExecute(hstmt);

/* Assign the results of the query (the Zone polygons) to the
   fetched_binary variable.
   */
SQLBindCol (hstmt, 1, SQL_C_Binary, fetched_binary, 100000, &ind_blob);

/* Fetch each polygon within the display window and display it. */
while(SQL_SUCCESS == (rc = SQLFetch(hstmt)))
    draw_polygon(fetched_binary);
```

ST_AsText

db2gse.ST_AsText takes a geometry object and returns its well-known text representation.

Syntax

```
db2gse.ST_AsText(g db2gse.ST_Geometry)
```

Return type

Varchar(4000)

Examples

In the scenario that follows, the db2gse.ST_AsText function converts the HAZARDOUS_SITES location point into its text description:

```
CREATE TABLE HAZARDOUS_SITES (site_id integer,
                                name     varchar(40),
                                location  db2gse.ST_Point);

INSERT INTO HAZARDOUS_SITES
VALUES (102,
       'W. H. Kleenare Chemical Repository',
       db2gse.ST_PointFromText('point (1020.12 324.02)',
                                db2gse.coordref().srid));

SELECT site_id, name, cast(db2gse.ST_AsText(location) as varchar(40))
       "Location"
FROM HAZARDOUS_SITES;
```

The SELECT statement returns the following result set:

SITE_ID	Name	Location
102	W. H. Kleenare Chemical Repository	POINT (1020.00000000 324.00000000)

ST_Boundary

ST_Boundary takes a geometry object and returns its combined boundary as a geometry object.

Syntax

```
db2gse.ST_Boundary(g db2gse.ST_Geometry)
```

Return type

```
db2gse.ST_Geometry
```

Examples

In the following code fragment, a table named BOUNDARY_TEST is created. BOUNDARY_TEST has two columns: GEOTYPE, which is defined as a varchar, and G1, which is defined as the superclass geometry. The INSERT statements that follow insert each one of the subclass geometries. The ST_Boundary function retrieves the boundary of each subclass that is stored in the G1 geometry column. Note that the dimension of the resulting geometry is always one less than the input geometry. Points and multipoints always result in a boundary that is an empty geometry, dimension 1. Linestrings and multilinestring return a multipoint boundary, dimension 0. A polygon or multipolygon always return a multilinestring boundary, dimension 1.

```
CREATE TABLE BOUNDARY_TEST (GEOTYPE varchar(20), G1 db2gse.ST_Geometry)

INSERT INTO BOUNDARY_TEST
VALUES('Point',
      db2gse.ST_PointFromText('point (10.02 20.01)',
                              db2gse.coordref()..srid(0)))

INSERT INTO BOUNDARY_TEST
VALUES('Linestring',
      db2gse.ST_LineFromText('linestring (10.02 20.01,10.32 23.98,
11.92 25.64)', db2gse.coordref()..srid(0)))

INSERT INTO BOUNDARY_TEST
VALUES('Polygon',
      db2gse.ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
19.15 33.94, 10.02 20.01))',
                              db2gse.coordref()..srid(0)))

INSERT INTO BOUNDARY_TEST
VALUES('Multipoint',
      db2gse.ST_MPointFromText('multipoint (10.02 20.01,10.32 23.98,
11.92 25.64)', db2gse.coordref()..srid(0)))

INSERT INTO BOUNDARY_TEST
VALUES('Multilinestring',
      db2gse.ST_MLineFromText('multilinestring ((10.02 20.01,10.32 23.98,
11.92 25.64), (9.55 23.75,15.36 30.11))',
                              db2gse.coordref()..srid(0)))

INSERT INTO BOUNDARY_TEST
```

```

VALUES('Multipolygon',
       db2gse.ST_MPolyFromText('multipolygon (((10.02 20.01,11.92 35.64,
                                               25.02 34.15, 19.15 33.94,10.02 20.01)),
                               ((51.71 21.73,73.36 27.04,71.52 32.87,
                               52.43 31.90,51.71 21.73)))',
       db2gse.coordref()..srid(0)))

SELECT GEOTYPE,
       CAST(db2gse.ST_AsText(db2gse.ST_Boundary (G1)) as varchar(280))
       "The boundary"
FROM BOUNDARY_TEST

```

GEOTYPE	The boundary
Point	POINT EMPTY
Linestring	MULTIPOINT (10.02000000 20.01000000, 11.92000000 25.64000000)
Polygon	MULTILINESTRING ((10.02000000 20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000, 11.92000000 35.64000000, 10.02000000 20.01000000))
Multipoint	POINT EMPTY
Multilinestring	MULTIPOINT (9.55000000 23.75000000, 10.02000000 20.01000000, 11.92000000 25.64000000, 15.36000000 30.11000000)
Multipolygon	MULTILINESTRING ((51.71000000 21.73000000, 73.36000000 27.04000000, 71.52000000 32.87000000, 52.43000000 31.90000000, 51.71000000 21.73000000),(10.02000000 20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000, 11.92000000 35.64000000, 10.02000000 20.01000000))

6 record(s) selected.

ST_Buffer

ST_Buffer takes a geometry object and distance and returns the geometry object that surrounds the source object.

Syntax

```
db2gse.ST_Buffer(g db2gse.ST_Geometry , adistance Double)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The County Supervisor needs a list of hazardous sites whose five-mile radius overlaps sensitive areas such as schools, hospitals, and nursing homes. The sensitive areas are stored in the table SENSITIVE_AREAS that is created with the following CREATE TABLE statement. The ZONE column is defined as a polygon, which is stored as the outline of each sensitive area.

```
CREATE TABLE SENSITIVE_AREAS (id      integer,
                               name    varchar(128),
                               size    float,
                               type    varchar(10),
                               zone    db2gse.ST_Polygon);
```

The hazardous sites are stored in the HAZARDOUS_SITES table that is created with the following CREATE TABLE statement. The LOCATION column, defined as a point, stores a location that is the geographic center of each hazardous site.

```
CREATE TABLE HAZARDOUS_SITES (site_id integer,
                               name     varchar(128),
                               location  db2gse.ST_Point);
```

The SENSITIVE_AREAS and HAZARDOUS_SITES tables are joined by the db2gse.ST_Overlaps function. The function returns 1 (TRUE) for all SENSITIVE_AREAS rows whose zone polygons overlap the buffered five-mile radius of the HAZARDOUS_SITES location point.

```
SELECT sa.name "Sensitive Areas", hs.name "Hazardous Sites"
FROM SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
WHERE db2gse.ST_Overlaps(sa.zone, db2gse.ST_Buffer (hs.location,(5 * 5280)))
= 1;
```

In Figure 29 on page 185, some of the sensitive areas in this administrative district lie within the five-mile buffer of the hazardous site locations. Both of the five-mile buffers intersect the hospital, and one of them intersects the school. However the nursing home lies safely outside both radii.

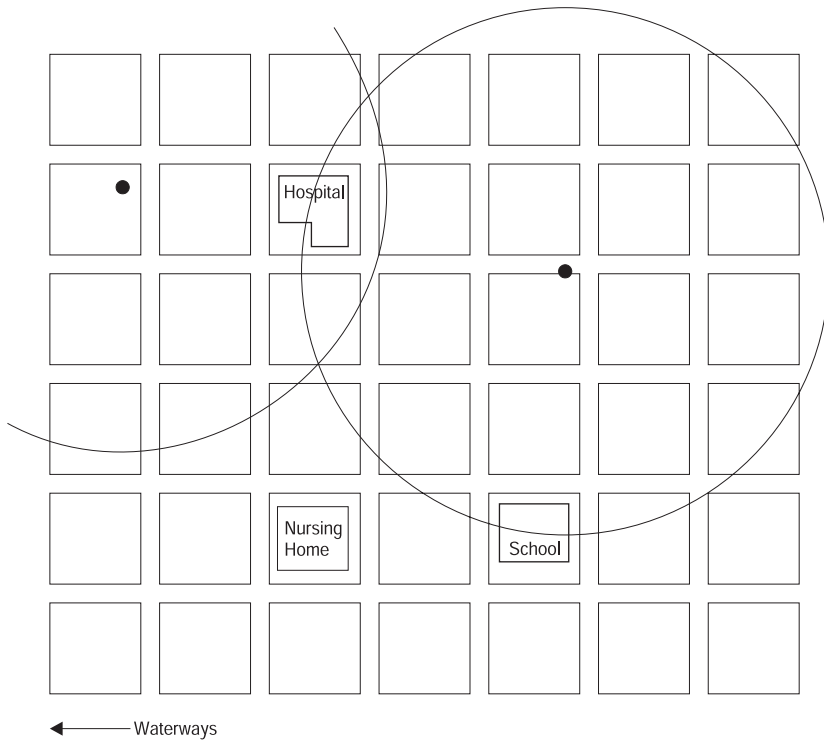


Figure 29. A buffer with a five-mile radius is applied to a point

ST_Centroid

ST_Centroid takes a polygon or multipolygon and returns its geometric center as a point.

Syntax

```
db2gse.ST_Centroid(s db2gse.ST_Surface)
db2gse.ST_Centroid(ms db2gse.ST_MultiSurface)
```

Return type

For surface: db2gse.ST_Point

Examples

The city GIS technician wants to display the multipolygons of the building footprints as single points in a building density graphic.

The building footprints are stored in the BUILDINGFOOTPRINTS table that was created with the following CREATE TABLE statement.

```
CREATE TABLE BUILDINGFOOTPRINTS (building_id integer,
                                   lot_id      integer,
                                   footprint   db2gse.ST_MultiPolygon);
```

The ST_Centroid function returns the centroid of each building footprint multipolygon. The AsBinaryShape function converts centroid point into a shape, the external representation that is recognized by the application.

```
SELECT building_id,
       CAST(db2gse.AsBinaryShape(db2gse.ST_Centroid (footprint)) as blob(1m))
  "Centroid"
FROM BUILDINGFOOTPRINTS;
```

ST_Contains

ST_Contains takes two geometry objects and returns 1 (TRUE) if the first object completely contains the second; otherwise it returns 0 (FALSE).

Syntax

```
db2gse.ST_Contains(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Integer

Examples

In the example below two tables are created. One table contains a city's building footprints, while the other table contains its lots. The city engineer wants to make sure that all the building footprints are completely inside their lots.

In both tables the multipolygon data type stores the geometry of the building footprints and the lots. The database designer selected multipolygons for both features. The designer realized that the lots can be disjointed by natural features, such as a river, and that the building footprints can often be made of several buildings.

```
CREATE TABLE BUILDINGFOOTPRINTS (building_id integer,  
                                lot_id        integer,  
                                footprint     db2gse.ST_MultiPolygon);
```

```
CREATE TABLE LOTS (lot_id integer, lot db2gse.ST_MultiPolygon);
```

The city engineer first selects the buildings that are not completely contained within one lot.

```
SELECT building_id  
FROM BUILDINGFOOTPRINTS, LOTS  
WHERE db2gse.ST_Contains(lot, footprint) = 0;
```

The city engineer realizes that the first query will return a list of all building IDs that have footprints outside of a lot polygon. But the city engineer also knows that this information will not indicate whether the other buildings have the correct lot ID assigned to them. This second query performs a data integrity check on the lot_id column of the BUILDINGFOOTPRINTS table.

```
SELECT bf.building_id "building id", bf.lot_id "buildings lot_id",  
       LOTS.lot_id "LOTS lot_id"  
FROM BUILDINGFOOTPRINTS bf, LOTS  
WHERE db2gse.ST_Contains(lot, footprint) = 1 AND LOTS.lot_id <> bf.lot_id;
```

In Figure 30 on page 188, the building footprints labeled with their building IDs lie inside their lots. The lot lines are illustrated with dotted lines.

Although not shown, these lines extend to the street centerline and completely encompass the lots and the building footprints within them.

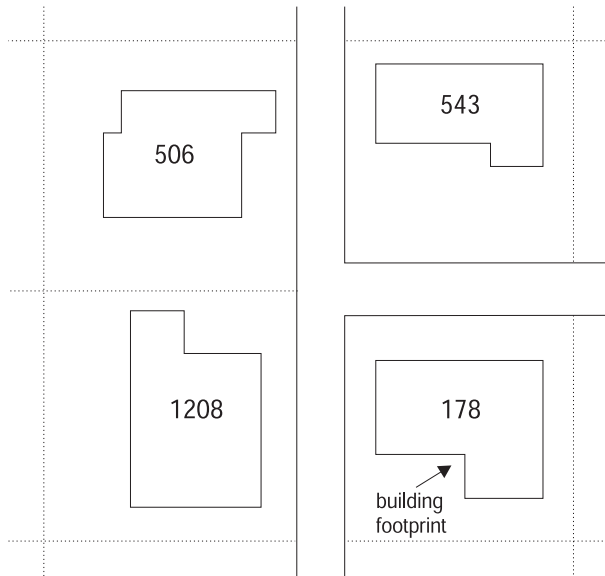


Figure 30. Using `ST_Contains` to ensure that all buildings are contained within their lots

ST_ConvexHull

ST_ConvexHull takes a geometry object and returns the convex hull.

Syntax

```
db2gse.ST_ConvexHull(g db2gse.ST_Geometry)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The example creates the CONVEXHULL_TEST table that has two columns: GEOTYPE and G1. The GEOTYPE column, a varchar(20), will store the name of the subclass of geometry that is stored in G1, which is defined as a geometry.

```
CREATE TABLE CONVEXHULL_TEST (geotype varchar(20), g1 db2gse.ST_Geometry)
```

Each INSERT statement inserts a geometry of each subclass type into the CONVEXHULL_TEST table.

```
INSERT INTO CONVEXHULL_TEST
VALUES('Point',
      db2gse.ST_PointFromText('point (10.02 20.01)',
      db2gse.coordref()..srid(0)))
```

```
INSERT INTO CONVEXHULL_TEST
VALUES('Linestring',
      db2gse.ST_LineFromText('linestring (10.02 20.01,10.32 23.98,
      11.92 25.64)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO CONVEXHULL_TEST
VALUES('Polygon',
      db2gse.ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
      19.15 33.94,10.02 20.01))',
      db2gse.coordref()..srid(0)))
```

```
INSERT INTO CONVEXHULL_TEST
VALUES('Multipoint',
      db2gse.ST_MPointFromText('multipoint (10.02 20.01,10.32 23.98,
      11.92 25.64)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO CONVEXHULL_TEST
VALUES('Multilinestring',
      db2gse.ST_MLineFromText('multilinestring ((10.02 20.01,10.32 23.98,
      11.92 25.64),(9.55 23.75,15.36 30.11))',
      db2gse.coordref()..srid(0)))
```

```
INSERT INTO CONVEXHULL_TEST
VALUES('Multipolygon',
      db2gse.ST_MPolyFromText('multipolygon (((10.02 20.01,11.92 35.64,
      25.02 34.15, 19.15 33.94,10.02 20.01)),
```

```
        ((51.71 21.73,73.36 27.04,71.52 32.87,  
          52.43 31.90,51.71 21.73)))',  
db2gse.coordref(..srid(0)))
```

The following SELECT statement lists the subclass name stored in the GEOTYPE column and the convex hull. The convexhull generated by the ST_ConvexHull function is converted to text by the ST_AsText function. It is then cast to a varchar(256) because the default output of ST_AsText is varchar(4000).

```
SELECT GEOTYPE, CAST(db2gse.ST_AsText(db2gse.ST_ConvexHull(G1)))  
as varchar(256) "The convexhull"  
FROM CONVEXHULL_TEST
```

ST_CoordDim

ST_CoordDim returns the coordinate dimensions of the ST_Geometry value. For an explanation of coordinate dimensions, see “Points” on page 125.

Syntax

```
db2gse.ST_CoordDim(g1 db2gse.ST_Geometry)
```

Return type

Integer

Examples

The `coorddim_test` table is created with the columns `geotype` and `g1`. The `geotype` column stores the name of the geometry subclass stored in the `g1` geometry column.

```
CREATE TABLE coorddim_test (geotype varchar(20), g1 db2gse.ST_Geometry)
```

The INSERT statements insert a sample subclass into the `coorddim_test` table.

```
INSERT INTO coorddim_test VALUES(
  'Point', db2gse.ST_PointFromText('point (10.02 20.01)',
  db2gse.coordref()..srid(0))
)
```

```
INSERT INTO coorddim_test VALUES(
  'Linestring',
  db2gse.ST_LineFromText('linestring (10.02 20.01,10.32 23.98,11.92 25.64)',
  db2gse.coordref()..srid(0))
)
```

```
INSERT INTO coorddim_test VALUES(
  'Polygon', db2gse.ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,
  25.02 34.15, 19.15 33.94,10.02 20.01))', db2gse.coordref()..srid(0))
)
```

```
INSERT INTO coorddim_test VALUES(
  'Multipoint', db2gse.ST_MPointFromText('multipoint (10.02 20.01,10.32 23.98,
  11.92 25.64)', db2gse.coordref()..srid(0))
)
```

```
INSERT INTO coorddim_test VALUES(
  'Multilinestring', db2gse.ST_MLineFromText('multilinestring ((10.02 20.01,
  10.32 23.98,11.92 25.64),(9.55 23.75,15.36 30.11))', db2gse.coordref()..srid(0))
)
```

```
INSERT INTO coorddim_test VALUES(
  'Multipolygon',
  MPolyFromText('multipolygon (((10.02 20.01,11.92 35.64,25.02 34.15,
  19.15 33.94,10.02 20.01)),((51.71 21.73,73.36 27.04,71.52 32.87,
  52.43 31.90,51.71 21.73)))', db2gse.coordref()..srid(0))
)
```

The SELECT statement lists the subclass name stored in the geotype column with the coordinate dimension of that geotype.

```
SELECT geotype, db2gse.ST_coordDim(g1)' coordinate_dimension'  
FROM coorddim_test
```

GEOTYPE	coordinate_dimension
ST_Point	2
ST_Linestring	2
ST_Polygon	2
ST_Multipoint	2
ST_Multilinestring	2
ST_Multipolygon	2

6 record(s) selected.

ST_Crosses

ST_Crosses takes two geometry objects and returns 1 (TRUE) if their intersection results in a geometry object whose dimension is one less than the maximum dimension of the source objects. The intersection object contains points that are interior to both source geometries and is not equal to either of the source objects. Otherwise it returns 0 (FALSE).

Syntax

```
db2gse.ST_Crosses(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Integer

Examples

The county government is considering a new regulation that states that all hazardous waste storage facilities within the county cannot be within five miles of any waterway. The county GIS manager has an accurate representation of rivers and streams, which are stored as multilinestrings in the WATERWAYS table. But, the GIS manager has only a single point location for each of the hazardous waste storage facilities.

```
CREATE TABLE WATERWAYS (id      integer,
                          name    varchar(128),
                          water   db2gse.ST_MultiLineString);
```

```
CREATE TABLE HAZARDOUS_SITES ( site_id  integer,
                                name      varchar(128),
                                location  db2gse.ST_Point);
```

To determine if the county supervisor needs to be alerted to facilities that violate the proposed regulation, the GIS manager should buffer the hazardous site locations and see if any rivers or streams cross the buffer polygon. The ST_Crosses predicate compares the buffered HAZARDOUS_SITES with WATERWAYS. So, only those records in which the waterway crosses over the county's proposed regulated radius are returned.

```
SELECT ww.name "River or stream", hs.name "Hazardous site"
FROM WATERWAYS ww, HAZARDOUS_SITES hs
WHERE db2gse.ST_Crosses(db2gse.ST_Buffer(hs.location,(5 * 5280)),ww.water)
= 1;
```

In Figure 31 on page 194, the five-mile buffer of the hazardous waste sites crosses the stream network that runs through the county's administrative district. The stream network has been defined as a multilinestring. So, the result set includes all linestring segments that are part of those segments that cross the radius.

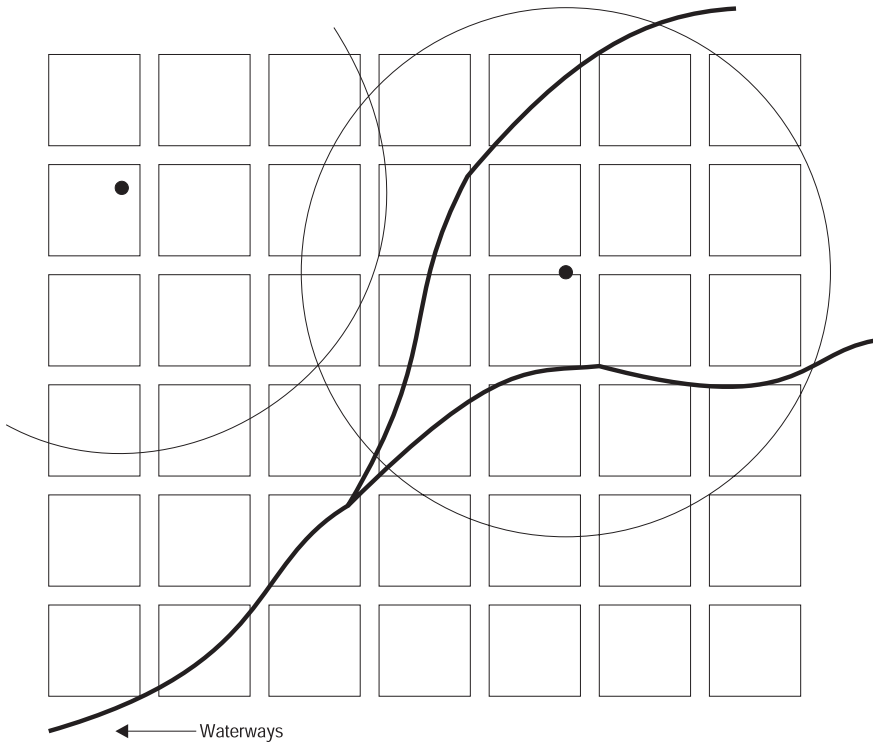


Figure 31. Using `ST_Crosses` to find the waterways that pass through a hazardous waste area

ST_Difference

ST_Difference takes two geometry objects and returns a geometry object that is the difference of the source objects.

Syntax

```
db2gse.ST_Difference(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The city engineer needs to know the total area of the city's lot area not covered by a building. That is, the city engineer wants the sum of the lot area after the building area has been removed.

```
CREATE TABLE BUILDINGFOOTPRINTS (building_id integer,  
                                lot_id       integer,  
                                footprint    db2gse.ST_MultiPolygon);  
  
CREATE TABLE LOTS (lot_id integer,  
                  lot     db2gse.ST_MultiPolygon);
```

The city engineer equijoins the BUILDINGFOOTPRINTS and LOTS table on the lot_id. The engineer then takes the sum of the area of the difference of the lots, minus the building footprints.

```
SELECT SUM(db2gse.ST_Area(db2gse.ST_Difference(lot,footprint)))  
FROM BUILDINGFOOTPRINTS bf, LOTS  
WHERE bf.lot_id = LOTS.lot_id;
```

ST_Dimension

ST_Dimension takes a geometry object and returns its dimension.

Syntax

```
db2gse.ST_Dimension(g1 db2gse.ST_Geometry)
```

Return type

Integer

Examples

The DIMENSION_TEST table is created with the columns GEOTYPE and G1. The GEOTYPE column stores the name of the geometry subclass that is stored in the G1 geometry column.

```
CREATE TABLE DIMENSION_TEST (geotype varchar(20), g1 db2gse.ST_Geometry)
```

The INSERT statements insert a sample subclass into the DIMENSION_TEST table.

```
INSERT INTO DIMENSION_TEST
VALUES('Point',
      db2gse.ST_PointFromText('point (10.02 20.01)',
                              db2gse.coordref()..srid(0)))
```

```
INSERT INTO DIMENSION_TEST
VALUES('Linestring',
      db2gse.ST_LineFromText('linestring (10.02 20.01,10.32 23.98,
                                          11.92 25.64)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO DIMENSION_TEST
VALUES('Polygon',
      db2gse.ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
                                       19.15 33.94,10.02 20.01))',
                              db2gse.coordref()..srid(0)))
```

```
INSERT INTO DIMENSION_TEST
VALUES('Multipoint',
      db2gse.ST_MPointFromText('multipoint (10.02 20.01,10.32 23.98,
                                           11.92 25.64)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO DIMENSION_TEST
VALUES('Multilinestring',
      db2gse.ST_MLineFromText('multilinestring ((10.02 20.01,10.32 23.98,
                                                  11.92 25.64),(9.55 23.75,15.36 30.11))',
                              db2gse.coordref()..srid(0)))
```

```
INSERT INTO DIMENSION_TEST
VALUES('Multipolygon',
      db2gse.ST_MPolyFromText('multipolygon (((10.02 20.01,11.92 35.64,
                                                25.02 34.15,19.15 33.94,10.02 20.01)),
                                ((51.71 21.73,73.36 27.04,71.52 32.87,
                                52.43 31.90,51.71 21.73)))',
                              db2gse.coordref()..srid(0)))
```

The following SELECT statement lists the subclass name stored in the GEOTYPE column with the dimension of that geotype.

```
SELECT geotype, db2gse.ST_Dimension(g1) "The dimension"  
FROM DIMENSION_TEST
```

The following result set is returned.

GEOTYPE	The dimension
ST_Point	0
ST_Linestring	1
ST_Polygon	2
ST_Multipoint	0
ST_Multilinestring	1
ST_Multipolygon	2

6 record(s) selected.

ST_Disjoint

ST_Disjoint takes two geometries and returns 1 (TRUE) if the intersection of two geometries produces an empty set; otherwise it returns 0 (FALSE).

Syntax

```
db2gse.ST_Disjoint(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Integer

Examples

An insurance company needs to assess the insurance coverage for a town's hospital, nursing homes, and schools. Part of this process includes determining the threat that the hazardous waste sites pose to each institution. The insurance company wants to consider only those institutions that are not at risk of contamination. The GIS consultant hired by the insurance company has been commissioned to locate all institutions that are not within a five-mile radius of a hazardous waste site.

The SENSITIVE_AREAS table contains several columns that describe the threatened institutions in addition to the ZONE column, which stores the institution's polygon geometry.

```
CREATE TABLE SENSITIVE_AREAS (id          integer,
                               name       varchar(128),
                               size       float,
                               type       varchar(10),
                               zone       db2gse.ST_Polygon);
```

The HAZARDOUS_SITES table stores the identity of the sites in the SITE_ID and NAME columns, while the actual geographic location of each site is stored in the LOCATION column.

```
CREATE TABLE HAZARDOUS_SITES (site_id  integer,
                               name     varchar(128),
                               location db2gse.ST_Point);
```

The following SELECT statement lists the names of all sensitive areas that are not within the 5-mile radius of a hazardous waste site. The ST_Intersects function could replace the ST_Disjoint function in this query if the result of the function is set equal to 0 instead of 1. This is because ST_Intersects and ST_Disjoint return the exact opposite result.

```
SELECT sa.name
FROM SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
WHERE db2gse.ST_Disjoint(db2gse.ST_Buffer(hs.location,(5 * 5280)),sa.zone) = 1;
```

In Figure 32 on page 199, sensitive are sites are compared to the five-mile radius of the hazardous waste sites. The nursing home is the only sensitive

area where the `ST_Disjoint` function will return 1 (TRUE). The `ST_Disjoint` function returns 1 whenever two geometries do not intersect in any way.

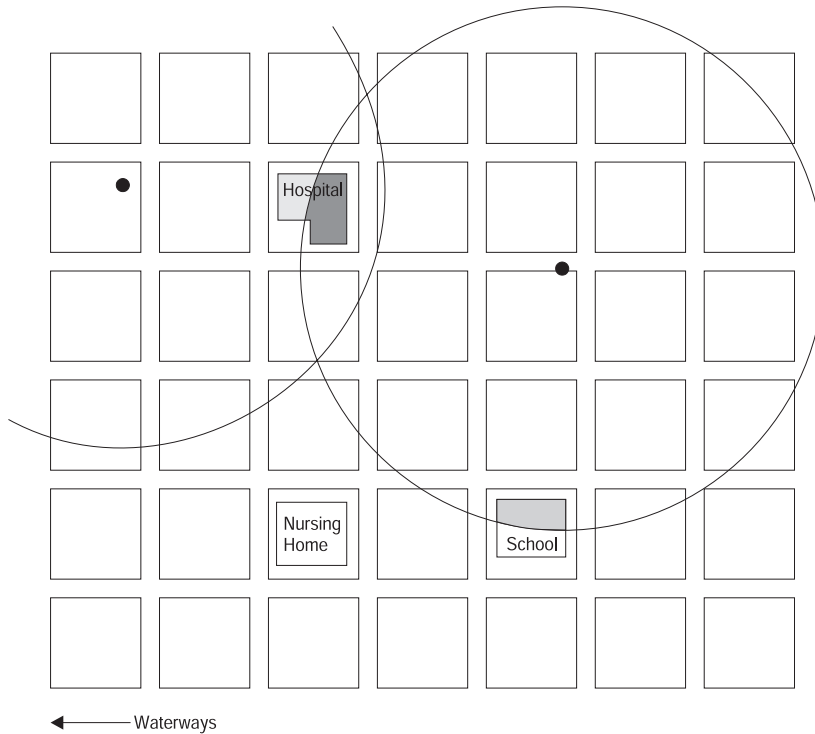


Figure 32. Using `ST_Disjoint` to find the buildings that do not lie within (intersect) any hazardous waste area

ST_Distance

ST_Distance takes two geometries and returns the closest distance separating them.

Syntax

```
db2gse.ST_Distance(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Double

Examples

The city engineer needs a list of all buildings that are within one foot of any lot line.

The BUILDING_ID column of the BUILDINGFOOTPRINTS table uniquely identifies each building. The LOT_ID column identifies the lot each building belongs to. The footprint multipolygon stores the geometry of each building's footprint.

```
CREATE TABLE BUILDINGFOOTPRINTS (  building_id integer,
                                   lot_id         integer,
                                   footprint       db2gse.ST_MultiPolygon);
```

The LOTS table stores the lot ID that uniquely identifies each lot, and the lot multipolygon that contains the lot line geometry.

```
CREATE TABLE LOTS (  lot_id   integer,
                    lot       db2gse.ST_MultiPolygon);
```

The query returns a list of building IDs that are within one foot of their lot lines. The ST_Distance function performs a spatial join between the footprints and the boundary of the lot multipolygons. However, the equijoin between the bf.lot_id and LOTS.lot_id ensures that only the multipolygons belonging to the same lot are compared by the ST_Distance function.

```
SELECT bf.building_id
FROM   BUILDINGFOOTPRINTS bf, LOTS
WHERE  bf.lot_id = LOTS.lot_id AND
       db2gse.ST_Distance(bf.footprint, db2gse.ST_Boundary(LOTS.lot)) <= 1.0;
```

ST_Endpoint

ST_Endpoint takes a linestring and returns a point that is the linestring's last point.

Syntax

```
db2gse.ST_Endpoint(c db2gse.ST_Curve)
```

Return type

```
db2gse.ST_Point
```

Examples

The ENDPOINT_TEST table stores the GID integer column that uniquely identifies each row and the LN1 linestring column that stores linestrings.

```
CREATE TABLE ENDPOINT_TEST (gid integer, ln1 db2gse.ST_LineString)
```

The INSERT statements insert linestrings into the ENDPOINT_TEST table. The first one does not have Z coordinates or measures; the second one does.

```
INSERT INTO ENDPOINT_TEST
VALUES( 1,
       db2gse.ST_LineFromText('linestring (10.02 20.01,23.73 21.92,
                               30.10 40.23)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO ENDPOINT_TEST
VALUES(2,
       db2gse.ST_LineFromText('linestring zm (10.02 20.01 5.0 7.0,
                               23.73 21.92 6.5 7.1, 30.10 40.23 6.9 7.2)',
                               db2gse.coordref()..srid(0)))
```

The following SELECT statement lists the GID column with the output of the ST_Endpoint function. The ST_Endpoint function generates a point geometry that is converted to text by the ST_AsText function. The CAST function is used to shorten the default varchar(4000) value of the ST_AsText function to a varchar(60).

```
SELECT gid, CAST(db2gse.ST_AsText(db2gse.ST_Endpoint(ln1)) AS varchar(60))
"Endpoint"
FROM ENDPOINT_TEST
```

The following result set is returned.

```
GID      Endpoint
-----
1 POINT ( 30.10000000 40.23000000)
2 POINT ZM ( 30.10000000 40.23000000 7.00000000 7.20000000)
```

2 record(s) selected.

ST_Envelope

ST_Envelope takes a geometry object and returns its bounding box as a geometry.

Syntax

```
db2gse.ST_Envelope(g db2gse.ST_Geometry)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The GEOTYPE column in the ENVELOPE_TEST table stores the name of the geometry subclass stored in the G1 geometry column.

```
CREATE TABLE ENVELOPE_TEST (geotype varchar(20), g1 db2gse.ST_Geometry)
```

The following INSERT statements insert each geometry subclass into the ENVELOPE_TEST table.

```
INSERT INTO ENVELOPE_TEST
VALUES('Point',
      db2gse.ST_PointFromText('point (10.02 20.01)',
      db2gse.coordref()..srid(0)))
```

```
INSERT INTO ENVELOPE_TEST
VALUES ('Linestring',
      db2gse.ST_LineFromText('linestring (10.01 20.01, 10.01 30.01,
      10.01 40.01)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO ENVELOPE_TEST
VALUES('Linestring',
      db2gse.ST_LineFromText('linestring (10.02 20.01,10.32 23.98,11.92 25.64)',
      db2gse.coordref()..srid(0)))
```

```
INSERT INTO ENVELOPE_TEST
VALUES('Polygon',
      db2gse.ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
      19.15 33.94,10.02 20.01))',
      db2gse.coordref()..srid(0)))
```

```
INSERT INTO ENVELOPE_TEST
VALUES('Multipoint',
      db2gse.ST_MPointFromText('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)',
      db2gse.coordref()..srid(0)))
```

```
INSERT INTO ENVELOPE_TEST
VALUES('Multilinestring',
      db2gse.ST_MLineFromText('multilinestring ((10.01 20.01,20.01 20.01,
      30.01 20.01), (30.01 20.01,40.01 20.01,50.01 20.01))',
      db2gse.coordref()..srid(0)))
```

```
INSERT INTO ENVELOPE_TEST
VALUES('Multilinestring',
```



```

db2gse.ST_MLineFromText('multilinestring ((10.02 20.01,10.32 23.98,
11.92 25.64), ( 9.55 23.75,15.36 30.11))',
db2gse.coordref()..srid(0))

```

```

INSERT INTO ENVELOPE_TEST
VALUES('Multipolygon',
db2gse.ST_MPolyFromText('multipolygon (((10.02 20.01,11.92 35.64,
25.02 34.15, 19.15 33.94,10.02 20.01)),
((51.71 21.73,73.36 27.04,71.52 32.87,
52.43 31.90,51.71 21.73)))',
db2gse.coordref()..srid(0))

```

The following SELECT statement lists the subclass name next to its envelope. Because the ST_Envelope function returns either a point, linestring, or polygon, its output is converted to text with the ST_AsText function. The CAST function converts the default varchar(4000) result of the ST_AsText function to a varchar(280).

```

SELECT GEOTYPE, CAST(db2gse.ST_AsText(db2gse.ST_Envelope(g1)) AS varchar(280))
"The envelope"
FROM ENVELOPE_TEST

```

The following result set is returned.

GEOTYPE	The envelope
Point	POINT (10.02000000 20.01000000)
Linestring	LINESTRING (10.01000000 20.01000000, 10.01000000 40.01000000)
Linestring	POLYGON ((10.02000000 20.01000000, 11.92000000 20.01000000, 11.92000000 25.64000000, 10.02000000 25.64000000, 10.02000000 20.01000000))
Polygon	POLYGON ((10.02000000 20.01000000, 25.02000000 20.01000000, 25.02000000 35.64000000, 10.02000000 35.64000000, 10.02000000 20.01000000))
Multipoint	POLYGON ((10.02000000 20.01000000, 11.92000000 20.01000000, 11.92000000 25.64000000, 10.02000000 25.64000000, 10.02000000 20.01000000))
Multilinestring	LINESTRING (10.01000000 20.01000000, 50.01000000 20.01000000)
Multilinestring	POLYGON ((9.55000000 20.01000000, 15.36000000 20.01000000, 15.36000000 30.11000000, 9.55000000 30.11000000, 9.55000000 20.01000000))
Multipolygon	POLYGON ((10.02000000 20.01000000, 73.36000000 20.01000000, 73.36000000 35.64000000, 10.02000000 35.64000000, 10.02000000 20.01000000))

8 record(s) selected.

ST_Equals

ST_Equals compares two geometries and returns 1 (TRUE) if the geometries are identical; otherwise it returns 0 (FALSE).

Syntax

```
db2gse.ST_Equals(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Integer

Examples

The city GIS technician suspects that some of the data in the BUILDINGFOOTPRINTS table was somehow duplicated. The technician queries the table to determine if any of the footprint's multipolygons are equal.

The BUILDINGFOOTPRINTS table is created with the following statement. The BUILDING_ID column uniquely identifies the buildings; the LOT_ID column identifies the building's lot; and the FOOTPRINT column stores the building's geometry.

```
CREATE TABLE BUILDINGFOOTPRINTS ( building_id integer,  
                                  lot_id      integer,  
                                  footprint   db2gse.ST_MultiPolygon);
```

The BUILDINGFOOTPRINTS table is spatially joined to itself by the ST_Equals predicate, which returns 1 whenever it finds two of the multipolygons that are equal. The bf1.building_id <> bf2.building_id condition is required to eliminate the comparison of the same geometry.

```
SELECT bf1.building_id, bf2.building_id  
FROM BUILDINGFOOTPRINTS bf1, BUILDINGFOOTPRINTS bf2  
WHERE db2gse.ST_Equals(bf1.footprint,bf2.footprint) = 1  
      and bf1.building_id <> bf2.building_id;
```

ST_ExteriorRing

ST_ExteriorRing takes a polygon and returns its exterior ring as a linestring.

Syntax

db2gse.ST_ExteriorRing(s db2gse.ST_Polygon)

Return type

db2gse.ST_LineString

Examples

An ornithologist who is studying the bird population on several south sea islands, knows that the feeding zone of a particular bird species is restricted to the shoreline. To calculate of the island's carrying capacity, the ornithologist requires the island's perimeter. Although some of the islands have several ponds on them, the shorelines of the ponds are inhabited exclusively by another more aggressive bird species. Therefore, the ornithologist requires the perimeter of the exterior ring of the islands.

The ID and NAME columns of the ISLANDS table identify each island, and the LAND column of type ST_Polygon stores the geometry of each.

```
CREATE TABLE ISLANDS (id integer,  
                        name varchar(32),  
                        land db2gse.ST_Polygon);
```

The ST_ExteriorRing function extracts the exterior ring from each island polygon as a linestring. The length of the linestring is established by the length function. The linestring lengths are summarized by the SUM function.

```
SELECT SUM(db2gse.ST_length(db2gse.ST_ExteriorRing (land))) FROM ISLANDS;
```

In Figure 33 on page 206, the exterior rings of the islands represent the ecological interface that each island shares with the sea. Some of the islands have lakes, which are represented by the interior rings of the polygons.

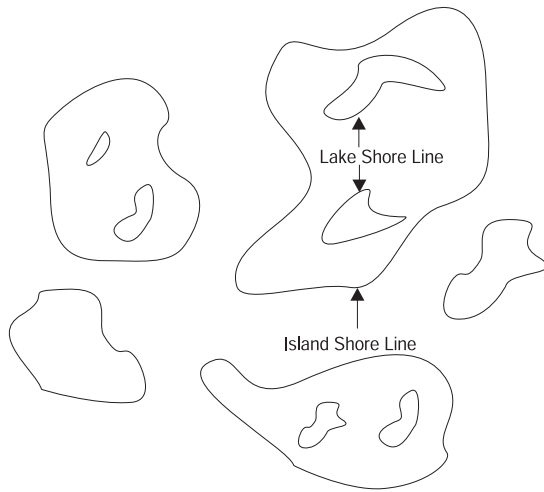


Figure 33. Using `ST_ExteriorRing` to determine the length of an island shore line

ST_GeometryFromText

ST_GeometryFromText takes a well-known text representation and a spatial reference system identity and returns a geometry object.

Syntax

```
db2gse.ST_GeometryFromText(geometryTaggedText Varchar(4000), cr
db2gse.coordref)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The GEOMETRY_TEST table contains the integer GID column, which uniquely identifies each row, and the G1 column, which stores the geometry.

```
CREATE TABLE GEOMETRY_TEST (gid smallint, g1 db2gse.ST_Geometry)
```

The INSERT statements inserts the data into the GID and G1 columns of the GEOMETRY_TEST table. The ST_GeometryFromText function converts the text representation of each geometry into its corresponding DB2 Spatial Extender instantiable subclass.

```
INSERT INTO GEOMETRY_TEST
VALUES(1, db2gse.ST_GeometryFromText('point (10.02 20.01)',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO GEOMETRY_TEST
VALUES (2,
db2gse.ST_GeometryFromText('linestring (10.01 20.01, 10.01 30.01,
10.01 40.01)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO GEOMETRY_TEST
VALUES(3,
db2gse.ST_GeometryFromText('polygon ((10.02 20.01,11.92 35.64,25.02 34.15,
19.15 33.94,10.02 20.01))',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO GEOMETRY_TEST
VALUES(4,
db2gse.ST_GeometryFromText('multipoint (10.02 20.01,10.32 23.98,
11.92 25.64)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO GEOMETRY_TEST
VALUES(5,
db2gse.ST_GeometryFromText('multilinestring ((10.02 20.01,10.32 23.98,
11.92 25.64),
( 9.55 23.75,15.36 30.11))',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO GEOMETRY_TEST
VALUES(6,
db2gse.ST_GeometryFromText('multipolygon (((10.02 20.01,11.92 35.64,
25.02 34.15, 19.15 33.94,10.02 20.01)),
```

```
((51.71 21.73,73.36 27.04,71.52 32.87,  
52.43 31.90,51.71 21.73)))',  
db2gse.coordref(..srid(0)))
```

ST_GeomFromWKB

ST_GeomFromWKB takes a well-known binary representation and a spatial reference system identity and returns a geometry object.

Syntax

db2gse.ST_GeomFromWKB(WKBGeometry Blob(1M), cr db2gse.coordref)

Return type

db2gse.ST_Geometry

Examples

The following C code fragment contains ODBC functions embedded with DB2 Spatial Extender SQL functions that insert data into the LOTS table.

The LOTS table was created with two columns: the LOT_ID column, which uniquely identifies each lot, and the LOT multipolygon column, which contains the geometry of each lot.

```
CREATE TABLE LOTS ( lot_id integer,
                    lot      db2gse.ST_MultiPolygon);
```

The ST_GeomFromWKB function converts WKB representations into DB2 Spatial Extender geometry. The entire INSERT statement is copied into wkb_sql char string. The INSERT statement contains parameter markers to accept the LOT_ID data and the LOT data dynamically.

```
/* Create the SQL insert statement to populate the lot id and the
   lot multipolygon. The question marks are parameter markers that
   indicate the lot_id and lot values that will be retrieved at
   runtime. */
strcpy (wkb_sql,"insert into LOTS (lot_id, lot) values (?,
db2gse.ST_GeomFromWKB

(cast(? as blob(1m)), db2gse.coordref()..srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);

/* Bind the integer key value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, 0, 0, &lot_id, 0, &pcbvalue1);

/* Bind the shape to the second parameter. */
pcbvalue2 = blob_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
```

```
        SQL_BLOB, blob_len, 0, shape_blob, blob_len, &pcbvalue2);  
/* Execute the insert statement. */  
rc = SQLExecute (hstmt);
```

ST_GeometryN

ST_GeometryN takes a collection and an integer index and returns the *n*th geometry object in the collection.

Syntax

```
db2gse.ST_GeometryN(g db2gse.ST_GeomCollection, n Integer)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The city engineer needs to know if the building footprints are all inside the first polygon of the lot's multipolygon.

The BUILDING_ID column uniquely identifies each row of the BUILDINGFOOTPRINTS table. The LOT_ID column identifies the building's lot. The FOOTPRINT column stores the building's geometry.

```
CREATE TABLE BUILDINGFOOTPRINTS ( building_id integer,  
                                lot_id      integer,  
                                footprint   db2gse.ST_MultiPolygon);
```

```
CREATE TABLE LOTS ( lot_id integer,  
                    lot     db2gse.ST_MultiPolygon);
```

The query lists the BUILDINGFOOTPRINTS building_id and lot_id for all building footprints that are all within the first lot polygon. The ST_GeometryN function returns a first lot polygon in the multipolygon array.

```
SELECT bf.building_id,bf.lot_id  
FROM BUILDINGFOOTPRINTS bf,LOTS  
WHERE db2gse.ST_Within(footprint, db2gse.ST_GeometryN (lot,1)) = 1  
      AND bf.lot_id = LOTS.lot_id;
```

ST_GeometryType

ST_GeometryType takes a ST_Geometry object and returns its geometry type as a string.

Syntax

```
db2gse.ST_GeometryType (g db2gse.ST_Geometry)
```

Return type

Varchar(4000)

Examples

The GEOMETRYTYPE_TEST table contains the G1 geometry column.

```
CREATE TABLE GEOMETRYTYPE_TEST(g1 db2gse.ST_Geometry)
```

The following INSERT statements insert each geometry subclass into the G1 column.

```
INSERT INTO GEOMETRYTYPE_TEST
VALUES(db2gse.ST_GeometryFromText('point (10.02 20.01)',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO GEOMETRYTYPE_TEST
VALUES (db2gse.ST_GeometryFromText('linestring (10.01 20.01, 10.01 30.01,
10.01 40.01)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO GEOMETRYTYPE_TEST
VALUES(db2gse.ST_Geometrytype_test values(db2gse.ST_GeomFromText('polygon
((10.02 20.01,11.92 35.64,25.02 34.15,19.15 33.94, 10.02 20.01))',
db2gse.coordref()..srid(0))))
```

```
INSERT INTO GEOMETRYTYPE_TEST
VALUES(db2gse.ST_GeometryFromText('multipoint (10.02
20.01,10.32 23.98,
11.92 25.64)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO GEOMETRYTYPE_TEST
VALUES(db2gse.ST_GeometryFromText('multilinestring ((10.02 20.01,10.32 23.98,
11.92 25.64),
(9.55 23.75,15.36 30.11))',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO GEOMETRYTYPE_TEST
VALUES(db2gse.ST_GeometryFromText('multipolygon (((10.02 20.01,11.92 35.64,
25.02 34.15, 19.15 33.94,10.02 20.01)),
((51.71 21.73,73.36 27.04,71.52 32.87,
52.43 31.90,51.71 21.73)))',
db2gse.coordref()..srid(0)))
```

The following SELECT statement lists the geometry type of each subclass that is stored in the G1 geometry column.

```
SELECT db2gse.ST_GeometryType(g1) "Geometry type" FROM GEOMETRYTYPE_TEST
```

The following result set is returned.

Geometry type

ST_Point
ST_LineString
ST_Polygon
ST_MultiPoint
ST_MultiLineString
ST_MultiPolygon

6 record(s) selected.

ST_InteriorRingN

Returns the n th interior ring of a polygon as a linestring. The rings are not organized by geometric orientation. They are organized according to the rules defined by the internal geometry verification routines. So, the order of the rings cannot be predefined.

Syntax

ST_InteriorRingN(p ST_Polygon, n Integer)

Return type

db2gse.ST_LineString

Examples

An ornithologist who is studying the bird population on several south sea islands knows that the feeding zone of a particular passive species is restricted to the seashore. Some of the islands have several lakes on them. The shorelines of the lakes are inhabited exclusively by another more aggressive species. The ornithologist knows that for each island, if the perimeter of the lakes exceeds a certain threshold, the aggressive species will become so numerous that it will threaten the passive seashore species. Therefore, the ornithologist requires the aggregated perimeter of the interior rings of the islands.

In Figure 34, the exterior rings of the islands represent the ecological interface that each island shares with the sea. Some of the islands have lakes, which are represented by the interior rings of the polygons.

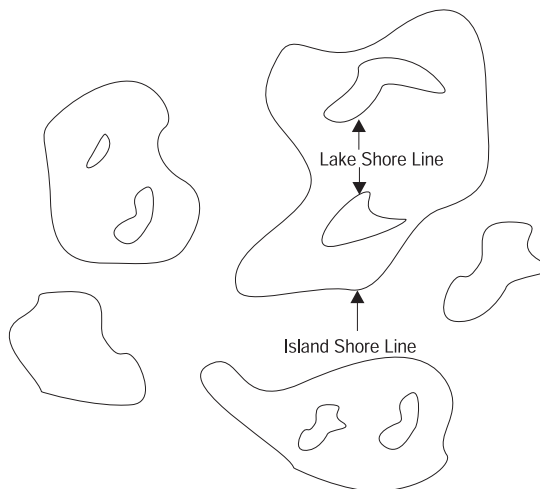


Figure 34. Using ST_InteriorRingN to determine the length of the lakeshores within each island

The ID and name columns of the ISLANDS table identify each island, while the land polygon column stores the island's geometry.

```
CREATE TABLE ISLANDS (id    integer,
                       name  varchar(32),
                       land  db2gse.ST_Polygon);
```

The following ODBC program uses the ST_InteriorRingN function to extract the interior ring (lake) from each island polygon as a linestring. The perimeter of the linestring that is returned by the length function is totaled and displayed along with the island's ID.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "sg.h"
#include "sgerr.h"
#include "sqlcli1.h"

/****                               ****
*** Change these constants ****
****                               ****/

#define USER_NAME    "sdetest" /* your user name */
#define USER_PASS    "acid.rain" /* your user password */
#define DB_NAME      "mydb" /* database to connect to */

static void check_sql_err (SQLHDBC handle,
                          SQLHSTMT hstmt,
                          LONG rc,
                          CHAR *str);

void main( argc, argv )
int argc;
char *argv[];
{
    SQLHDBC    handle;
    SQLHENV    henv;
    CHAR       sql_stmt[256];
    LONG       rc,
              total_perimeter,
              num_lakes,
              lake_number,
              island_id,
              lake_perimeter;
    SQLHSTMT   island_cursor,
              lake_cursor;
    SDWORD     pcbvalue,
              id_ind,
              lake_ind,
              length_ind;

    /* Allocate memory for the ODBC environment handle henv and initialize
```

```

the application. */

rc = SQLAllocEnv (&henv);
if (rc != SQL_SUCCESS)
{
    printf ("SQLAllocEnv failed with %d\n", rc);
    exit(0);
}

/* Allocate memory for a connection handle within the henv environment. */

rc = SQLAllocConnect (henv, &handle);
if (rc != SQL_SUCCESS)
{
    printf ("SQLAllocConnect failed with %d\n", rc);
    exit(0);
}

/* Load the ODBC driver and connect to the data source identified by the database,
user, and password.*/

rc = SQLConnect (handle,
                (UCHAR *)DB_NAME,
                SQL_NTS,
                (UCHAR *)USER_NAME,
                SQL_NTS,
                (UCHAR *)USER_PASS,
                SQL_NTS);

check_sql_err (handle, NULL, rc, "SQLConnect");

/* Allocate memory to the SQL statement handle island_cursor. */

rc = SQLAllocStmt (handle, &island_cursor);
check_sql_err (handle, NULL, rc, "SQLAllocStmt");

/* Prepare and execute the query to get the island IDs and number of
lakes (interior rings) */

strcpy (sql_stmt, "select id, db2gse.ST_NumInteriorRings(land) from ISLANDS");

rc = SQLExecDirect (island_cursor, (UCHAR *)sql_stmt, SQL_NTS);
check_sql_err (NULL, island_cursor, rc, "SQLExecDirect");

/* Bind the island table's ID column to the variable island_id */

rc = SQLBindCol (island_cursor, 1, SQL_C_SLONG, &island_id, 0, &id_ind);
check_sql_err (NULL, island_cursor, rc, "SQLBindCol");

/* Bind the result of numinteriorrings(land) to the num_lakes variable. */

rc = SQLBindCol (island_cursor, 2, SQL_C_SLONG, &num_lakes, 0, &lake_ind);
check_sql_err (NULL, island_cursor, rc, "SQLBindCol");

/* Allocate memory to the SQL statement handle lake_cursor. */

```

```

rc = SQLAllocStmt (handle, &lake_cursor);
check_sql_err (handle, NULL, rc, "SQLAllocStmt");

/* Prepare the query to get the length of an interior ring. */

strcpy (sql_stmt,
        "select Length(db2gse.ST_InteriorRingN(land, cast (? as
        integer))) from ISLANDS where id = ?");

rc = SQLPrepare (lake_cursor, (UCHAR *)sql_stmt, SQL_NTS);
check_sql_err (NULL, lake_cursor, rc, "SQLPrepare");

/* Bind the lake_number variable as the first input parameter */

pcbvalue = 0;
rc = SQLBindParameter (lake_cursor, 1, SQL_PARAM_INPUT, SQL_C_LONG,
        SQL_INTEGER, 0, 0, &lake_number, 0, &pcbvalue);
check_sql_err (NULL, lake_cursor, rc, "SQLBindParameter");

/* Bind the island_id as the second input parameter */

pcbvalue = 0;
rc = SQLBindParameter (lake_cursor, 2, SQL_PARAM_INPUT, SQL_C_LONG,
        SQL_INTEGER, 0, 0, &island_id, 0, &pcbvalue);
check_sql_err (NULL, lake_cursor, rc, "SQLBindParameter");

/* Bind the result of the Length(db2gse.ST_InteriorRingN(land, cast
(? as integer))) to the variable lake_perimeter */

rc = SQLBindCol (lake_cursor, 1, SQL_C_SLONG, &lake_perimeter, 0,
        &length_ind);
check_sql_err (NULL, island_cursor, rc, "SQLBindCol");

/* Outer loop, get the island ids and the number of lakes
(interior rings) */

while (SQL_SUCCESS == rc)
{
    /* Fetch an island */

    rc = SQLFetch (island_cursor);

    if (rc != SQL_NO_DATA)
    {
        check_sql_err (NULL, island_cursor, rc, "SQLFetch");

        /* Inner loop, for this island, get the perimeter of all of
        its lakes (interior rings) */

        for (total_perimeter = 0, lake_number = 1;
            lake_number <= num_lakes;
            lake_number++)
        {
            rc = SQLExecute (lake_cursor);

```

```

        check_sql_err (NULL, lake_cursor, rc, "SQLExecute");

        rc = SQLFetch (lake_cursor);
        check_sql_err (NULL, lake_cursor, rc, "SQLFetch");

        total_perimeter += lake_perimeter;

        SQLFreeStmt (lake_cursor, SQL_CLOSE);
    }
}

/* Display the Island id and the total perimeter of its lakes. */

    printf ("Island ID = %d, Total lake perimeter = %d\n",
            island_id, total_perimeter);

}

SQLFreeStmt (lake_cursor, SQL_DROP);
SQLFreeStmt (island_cursor, SQL_DROP);
SQLDisconnect (handle);
SQLFreeConnect (handle);
SQLFreeEnv (henv);

printf( "\nTest Complete ...\n" );

}

static void check_sql_err (SQLHDBC handle, SQLHSTMT hstmt, LONG rc,
                          CHAR *str)
{
    SDWORD dbms_err = 0;
    SWORD length;
    UCHAR err_msg[SQL_MAX_MESSAGE_LENGTH], state[6];

    if (rc != SQL_SUCCESS)
    {
        SQLError (SQL_NULL_HENV, handle, hstmt, state, &dbms_err,
                 err_msg, SQL_MAX_MESSAGE_LENGTH - 1, &length);
        printf ("%s ERROR (%d): DBMS code:%d, SQL state: %s, message:
                \n %s\n", str, rc, dbms_err, state, err_msg);

        if (handle)
        {
            SQLDisconnect (handle);
            SQLFreeConnect (handle);
        }
        exit(1);
    }
}
}

```

ST_Intersection

ST_Intersection takes two geometry objects and returns the intersection set as a geometry object.

Syntax

```
db2gse.ST_Intersection(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The fire marshall must obtain the areas of the hospitals, schools, and nursing homes that are intersected by the radius of a possible hazardous waste contamination.

The sensitive areas are stored in the table SENSITIVE_AREAS that is created with the following CREATE TABLE statement. The ZONE column is defined as a polygon that stores the outline of each of the sensitive areas.

```
CREATE TABLE SENSITIVE_AREAS (id    integer,
                               name  varchar(128),
                               size  float,
                               type  varchar(10),
                               zone  db2gse.ST_Polygon);
```

The hazardous sites are stored in the HAZARDOUS_SITES table that is created with the following CREATE TABLE statement. The LOCATION column, defined as a point, stores a location that is the geographic center of each hazardous site.

```
CREATE TABLE HAZARDOUS_SITES (site_id integer,
                               name     varchar(128),
                               location  db2gse.ST_Point);
```

The buffer function generates a five-mile buffer that surrounds the hazardous waste site locations. The ST_Intersection function generates polygons from the intersection of the buffered hazardous waste site polygons and the sensitive areas. The ST_Area function returns the intersection polygon's area, which is summarized for each hazardous site by the SUM function. The GROUP BY clause directs the query to aggregate the intersected areas by the hazardous waste site_ID.

```
SELECT hs.name, SUM(db2gse.ST_Area(db2gse.ST_Intersection (sa.zone,
db2gse.ST_buffer hs.location, (5 * 5280))))
FROM SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
GROUP BY hs.site_id;
```

In Figure 35 on page 220, the circles represent the buffered polygons that surround the hazardous waste sites. The intersection of these buffer polygons with the sensitive area polygons produces three other polygons. The hospital

in the upper left hand corner is intersected twice, while the school in the lower right hand corner is intersected only once.

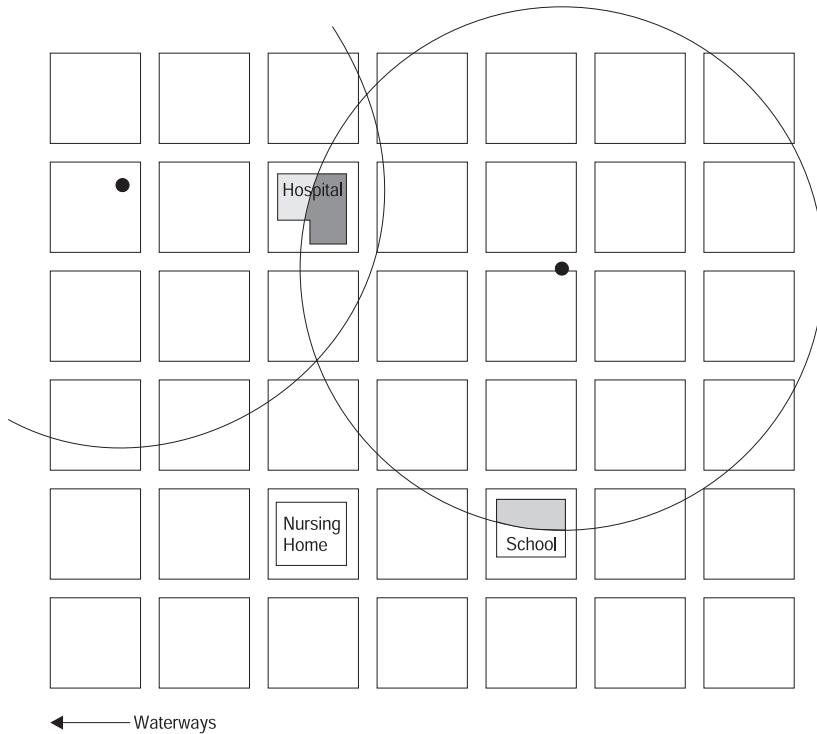


Figure 35. Using *ST_Intersection* to determine how large an area in each of the buildings might be affected by hazardous waste

ST_Intersects

ST_Intersects takes two geometries and returns 1 (TRUE), if the intersection of two geometries does not result in an empty set. Otherwise, it returns 0 (FALSE).

Syntax

```
db2gse.ST_Intersects(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Integer

Examples

The fire marshall needs a list of all sensitive areas within a five-mile radius of a hazardous waste site.

The sensitive areas are stored in the table SENSITIVE_AREAS that is created with the following CREATE TABLE statement. The ZONE column is defined as a polygon that stores the outline of each of the sensitive areas.

```
CREATE TABLE SENSITIVE_AREAS (id          integer,
                               name        varchar(128),
                               size        float,
                               type        varchar(10),
                               zone        db2gse.ST_Polygon);
```

The hazardous sites are stored in the HAZARDOUS_SITES table created with the following CREATE TABLE statement. The LOCATION column, defined as a point, stores a location that is the geographic center of each hazardous site.

```
CREATE TABLE HAZARDOUS_SITES (site_id   integer,
                               name        varchar(128),
                               location    db2gse.ST_Point);
```

The query returns a list of sensitive areas and hazardous site names for sensitive areas that intersect the five-mile buffer of the hazardous sites.

```
SELECT sa.name, hs.name
FROM SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
WHERE db2gse.ST_Intersects(db2gse.ST_Buffer(hs.location,(5 * 5280)),sa.zone)
= 1;
```

ST_IsClosed

ST_IsClosed takes a linestring or multilinestring and returns 1 (TRUE) if it is closed; otherwise it returns 0 (FALSE).

Syntax

```
db2gse.ST_IsClosed(c db2gse.ST_Curve)
db2gse.ST_IsClosed(mc db2gse.ST_MultiCurve)
```

Return type

Integer

Examples

The following CREATE TABLE statement creates the CLOSED_LINESTRING table, which has a single linestring column.

```
CREATE TABLE CLOSED_LINESTRING (l1n1 db2gse.ST_LineString)
```

The following INSERT statements insert two records into the CLOSED_LINESTRING table. The first record is not a closed linestring, while the second is.

```
INSERT INTO CLOSED_LINESTRING
VALUES(db2gse.ST_LineFromText('linestring (10.02 20.01,10.32 23.98,
11.92 25.64)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO CLOSED_LINESTRING
VALUES(db2gse.ST_LineFromText('linestring (10.02 20.01,11.92 35.64,
25.02 34.15, 19.15 33.94,10.02 20.01)',
db2gse.coordref()..srid(0)))
```

The following SELECT statement and the corresponding result set shows the results of the ST_IsClosed function. The first row returns a 0 because the linestring is not closed, while the second row returns a 1 because the linestring is closed.

```
SELECT db2gse.ST_IsClosed(l1n1) "Is it closed" FROM CLOSED_LINESTRING
```

```
Is it closed
-----
           0
           1
```

2 record(s) selected.

The following CREATE TABLE statement creates the CLOSED_MULTILINESTRING table, which has a single multilinestring column.

```
CREATE TABLE CLOSED_MULTILINESTRING (m1n1 db2gse.ST_MultiLineString)
```

The following INSERT statements insert two records into CLOSED_MULTILINESTRING, a multilinestring record that is not closed and another that is.

```
INSERT INTO CLOSED_MULTILINESTRING
VALUES(db2gse.ST_MLineFromText('multilinestring ((10.02 20.01,10.32 23.98,
11.92 25.64), (9.55 23.75,15.36 30.11))',
db2gse.coordref()..srid(0)))

INSERT INTO CLOSED_MULTILINESTRING
VALUES(db2gse.ST_MLineFromText('multilinestring ((10.02 20.01,11.92 35.64,
25.02 34.15, 19.15 33.94,10.02 20.01),
(51.71 21.73,73.36 27.04,71.52 32.87,
52.43 31.90,51.71 21.73))',
db2gse.coordref()..srid(0)))
```

The following SELECT statement and the corresponding result set shows the results of the ST_IsClosed function. The first row returns 0 because the multilinestring is not closed, while the second row returns 1 because the multilinestring is closed. A multilinestring is closed if all of its linestring elements are closed.

```
SELECT db2gse.ST_IsClosed(m1n1) "Is it closed" FROM CLOSED_MULTILINESTRING
```

```
Is it closed
-----
          0
          1
```

2 record(s) selected.

ST_IsEmpty

ST_IsEmpty takes a geometry object and returns 1 (TRUE) if it is empty; otherwise it returns 0 (FALSE).

Syntax

```
db2gse.ST_IsEmpty(g db2gse.ST_Geometry)
```

Return type

Integer

Examples

The following CREATE TABLE statement creates the EMPTY_TEST table with two columns. The GEOTYPE column stores the data type of the subclasses that are stored in the G1 geometry column.

```
CREATE TABLE EMPTY_TEST (geotype varchar(20), g1 db2gse.ST_Geometry)
```

The following INSERT statements insert two records for the geometry subclasses point, linestring, and polygon. One record is empty and one is not.

```
INSERT INTO EMPTY_TEST
VALUES('Point', db2gse.ST_PointFromText('point (10.02 20.01)',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO EMPTY_TEST
VALUES('Point', db2gse.ST_PointFromText('point empty',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO EMPTY_TEST
VALUES('Linestring', db2gse.ST_LineFromText('linestring (10.02 20.01,
10.32 23.98, 11.92 25.64)',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO EMPTY_TEST
VALUES('Linestring', db2gse.ST_LineFromText('linestring empty',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO EMPTY_TEST
VALUES('Polygon', db2gse.ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,
25.02 34.15,19.15 33.94,10.02 20.01))',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO EMPTY_TEST
VALUES('Polygon', db2gse.ST_PolyFromText('polygon empty',
db2gse.coordref()..srid(0)))
```

The following SELECT statement and corresponding result set show the geometry type from the GEOTYPE column and the results of the ST_IsEmpty function.

```
SELECT geotype, db2gse.ST_IsEmpty(g1) "It is empty" FROM EMPTY_TEST
```

```
GEOTYPE                It is empty
```

```
-----  
ST_Point          0  
ST_Point          1  
ST_Linestring    0  
ST_Linestring    1  
ST_Polygon       0  
ST_Polygon       1
```

6 record(s) selected.

ST_IsRing

ST_IsRing takes a linestring and returns 1 (TRUE) if it is a ring (namely, the linestring is closed and simple); otherwise, it returns 0 (FALSE).

Syntax

```
db2gse.ST_IsRing(c db2gse.ST_Curve)
```

Return type

Integer

Examples

The following CREATE TABLE statement creates the RING_LINESTRING table, which has a single linestring column that is called LN1.

```
CREATE TABLE RING_LINESTRING (ln1 db2gse.ST_LineString)
```

The following INSERT statements insert three linestrings into the LN1 column. The first row contains a linestring that is not closed and therefore is not a ring. The second row contains a linestring that is closed and is simple and therefore is a ring. The third row contains a linestring that is closed but is not simple because it intersects its own interior, and therefore is not a ring.

```
INSERT INTO RING_LINESTRING
VALUES(db2gse.ST_LineFromText('linestring (10.02 20.01,10.32 23.98,
                               11.92 25.64)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO RING_LINESTRING
VALUES(db2gse.ST_LineFromText('linestring (10.02 20.01,11.92 35.64,25.02 34.15,
                               19.15 33.94, 10.02 20.01)',
                               db2gse.coordref()..srid(0)))
```

```
INSERT INTO RING_LINESTRING
VALUES(db2gse.ST_LineFromText('linestring (15.47 30.12,20.73 22.12,10.83 14.13,
                               16.45 17.24,21.56 13.37,11.23 22.56,
                               19.11 26.78,15.47 30.12)',
                               db2gse.coordref()..srid(0)))
```

The following SELECT statement and the corresponding result set shows the results of the ST_IsRing function. The first and third rows return a 0. This is because the linestrings are not rings, while the second row returns a 1 because it is a ring.

```
SELECT db2gse.ST_IsRing(ln1) "Is it ring" FROM RING_LINESTRING
```

```
Is it ring
-----
          0
          1
          0
```

3 record(s) selected.

ST_IsSimple

ST_IsSimple takes a geometry object and returns 1 (TRUE) if the object is simple; otherwise, it returns 0 (FALSE).

Syntax

```
db2gse.ST_IsSimple(g db2gse.ST_Geometry)
```

Return type

Integer

Examples

The following CREATE TABLE statement creates the ISSIMPLE_TEST table, which has two columns. The PID column, which is a smallint, contains the unique identifier for each row. The G1 geometry column stores the simple and non-simple geometry samples.

```
CREATE TABLE ISSIMPLE_TEST (pid smallint, g1 db2gse.ST_Geometry)
```

The following INSERT statements insert two records into the ISSIMPLE_TEST table. The first is simple because it is a linestring that does not intersect its interior. The second is non-simple because it does intersect its interior.

```
INSERT INTO ISSIMPLE_TEST  
VALUES (1, db2gse.ST_LineFromText('linestring (10 10, 20 20, 30 30)',  
db2gse.coordref()..srid(0)))
```

```
INSERT INTO ISSIMPLE_TEST  
VALUES (2, db2gse.ST_LineFromText('linestring (10 10, 20 20,20 30,10 30,10 20,  
20 10)', db2gse.coordref()..srid(0)))
```

The following SELECT statement and the corresponding result set shows the results of the ST_IsSimple function. The first record returns a 1 because the linestring is simple, while the second record returns a 0 because the linestring is not simple.

```
SELECT ST_IsSimple(g1)  
FROM ISSIMPLE_TEST
```

```
g1  
-----  
1  
0
```

ST_IsValid

ST_IsValid takes an ST_Geometry and returns 1 (TRUE) if it is valid, otherwise it returns 0 (FALSE). A geometry inserted into a DB2 database will always be valid because the DB2 Spatial Extender always validates its spatial data before accepting it. However, other DBMS vendors may not validate the input, but instead require that the application do so.

Syntax

```
db2gse.ST_IsValid(g db2gse.ST_Geometry)
```

Return type

Integer

Examples

The valid_test table is created with the columns geotype and g1. The geotype column stores the name of the geometry subclass stored in the g1 geometry column.

```
CREATE TABLE valid_test (geotype varchar(20), g1 db2gse.ST_Geometry)
```

The INSERT statements inserts a sample subclass into the valid_test table.

```
INSERT INTO valid_test VALUES(  
    'Point', db2gse.ST_PointFromText('point (10.02 20.01)',  
    db2gse.coordref()..srid(0))  
)
```

```
INSERT INTO valid_test VALUES(  
    'Linestring',  
    db2gse.ST_LineFromText('linestring (10.02 20.01,10.32 23.98,11.92 25.64)',  
    db2gse.coordref()..srid(0))  
)
```

```
INSERT INTO valid_test VALUES(  
    'Polygon', db2gse.ST_PolyFromText('polygon ((10.02 20.01,11.92 35.64,  
    25.02 34.15, 19.15 33.94,10.02 20.01))', db2gse.coordref()..srid(0))  
)
```

```
INSERT INTO valid_test VALUES(  
    'Multipoint', db2gse.ST_MPointFromText('multipoint (10.02 20.01,10.32 23.98,  
    11.92 25.64)', db2gse.coordref()..srid(0))  
)
```

```
INSERT INTO valid_test VALUES(  
    'Multilinestring', db2gse.ST_MLineFromText('multilinestring ((10.02 20.01,  
    10.32 23.98,11.92 25.64),(9.55 23.75,15.36 30.11))',  
    db2gse.coordref()..srid(0))  
)
```

```
INSERT INTO valid_test VALUES(  
    'Multipolygon',
```

```

db2gse.ST_MPolyFromText('multipolygon (((10.02 20.01,11.92 35.64,
25.02 34.15,19.15 33.94,10.02 20.01)),((51.71 21.73,73.36 27.04,71.52 32.87,
52.43 31.90,51.71 21.73)))', db2gse.coordref()..srid(0))
)

```

The SELECT statement lists the subclass name stored in the geotype column with the dimension of that geotype.

```

SELECT geotype, db2gse.ST_IsValid(g1) Valid FROM valid_test

```

GEOTYPE	Valid
ST_Point	1
ST_Linestring	1
ST_Polygon	1
ST_Multipoint	1
ST_Multilinestring	1
ST_Multipolygon	1

6 record(s) selected.

ST_Length

ST_Length takes a linestring or multilinestring and returns its length.

Syntax

```
db2gse.ST_Length(c db2gse.ST_Curve)
db2gse.ST_Length(mc db2gse.ST_MultiCurve)
```

Return type

Double

Examples

A local ecologist is studying the migratory patterns of the salmon population in the county's waterways. The ecologist wants to obtain the length of all stream and river systems running through the county.

The following CREATE TABLE statement creates the WATERWAYS table. The ID and NAME columns identify each stream and river system that is stored in the table. The WATER column is a multilinestring because the river and stream systems are often an aggregate of several linestrings.

```
CREATE TABLE WATERWAYS (id integer, name varchar(128),
water db2gse.ST_MultiLineString);
```

The following SELECT statement uses the ST_Length function to return the name and length of each waterway within the county.

```
SELECT name, db2gse.ST_Length(water) "Length"
FROM WATERWAYS;
```

Figure 36 on page 231 displays a the river and stream systems that lie within the county boundary.

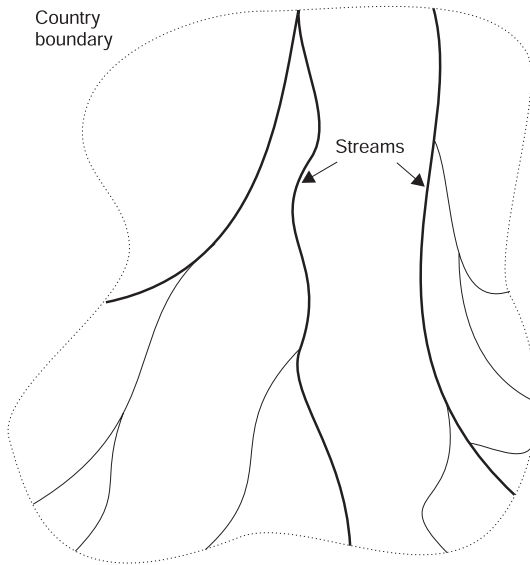


Figure 36. Using ST_Length to determine the total length of the waterways in a county

ST_LineFromText

ST_LineFromText takes a well-known text representation of type linestring and a spatial reference system identity and returns a linestring.

Syntax

```
db2gse.ST_LineFromText(lineStringTaggedText Varchar(4000), cr
db2gse.coordref)
```

Return type

```
db2gse.ST_LineString
```

Examples

The following CREATE TABLE statement creates the LINESSTRING_TEST table, which has a single LN1 linestring column.

```
CREATE TABLE LINESSTRING_TEST (ln1 db2gse.ST_LineString)
```

The following INSERT statement inserts a linestring into the LN1 column by using the ST_LineFromText function.

```
INSERT INTO LINESSTRING_TEST
VALUES (db2gse.ST_LineFromText('linestring(10.01 20.03,20.94 21.34,
35.93 19.04)', db2gse.coordref()..srid(0)))
```

ST_LineFromWKB

ST_LineFromWKB takes a well-known binary representation of the type linestring and a spatial reference system identity, and returns a linestring.

Syntax

```
db2gse.ST_LineFromWKB(WKBLinestring Blob(1M), cr db2gse.coordref)
```

Return type

```
db2gse.ST_LineString
```

Examples

The following code fragment populates the SEWERLINES table with the unique id, size class, and geometry of each sewer line.

The SEWERLINES table is created with three columns. The first column, SEWER_ID, uniquely identifies each sewer line. The second column, CLASS, of type integer identifies the type of sewer line, which is generally associated with the line's capacity. The third column, SEWER, of type linestring stores the sewer line's geometry.

```
CREATE TABLE SEWERLINES (sewer_id integer,
                        class integer,
                        sewer db2gse.ST_LineString);

/* Create the SQL insert statement to populate the sewer_id, size class
   and the sewer linestring. The question marks are parameter markers that
   indicate the sewer_id, class and sewer geometry values that will be
   retrieved at runtime. */
strcpy (wkb_sql,"insert into sewerlines (sewer_id,class,sewer)
values (?,?, db2gse.ST_LineFromWKB (cast(? as blob(1m)),
db2gse.coordref()..srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);

/* Bind the integer sewer_id value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, 0, 0, &sewer_id, 0, &pcbvalue1);

/* Bind the integer class value to the second parameter. */
pcbvalue2 = 0;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, 0, 0, &sewer_class, 0, &pcbvalue2);

/* Bind the shape to the third parameter. */
pcbvalue3 = blob_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
```

```
        SQL_BLOB, blob_len, 0, sewer_wkb, blob_len, &pcbvalue3);  
  
/* Execute the insert statement. */  
rc = SQLExecute (hstmt);
```

ST_MLineFromText

ST_MLineFromText takes a well-known text representation of type multilinestring and a spatial reference system identity and returns a multilinestring.

Syntax

```
db2gse.ST_MLineFromText(multiLineStringTaggedText String, cr
db2gse.coordref)
```

Return type

```
db2gse.ST_MultiLineString
```

Examples

The following CREATE TABLE statement creates the MLINESTRING_TEST table. MLINESTRING_TEST has two columns: the GID smallint column, which uniquely identifies the row, and the ML1 multilinestring column.

```
CREATE TABLE ST_MLINESTRING_TEST (gid smallint, ml1 db2gse.ST_MultiLineString)
```

The following INSERT statement inserts the multilinestring with the ST_MLineFromText function.

```
INSERT INTO MLINESTRING_TEST
VALUES (1, db2gse.ST_MLineFromText('multilinestring((10.01 20.03,10.52 40.11,
30.29 41.56,31.78 10.74),
(20.93 20.81, 21.52 40.10))',
db2gse.coordref()..srid(0)))
```

ST_MLineFromWKB

ST_MLineFromWKB takes a well-known binary representation of type multilinestring and a spatial reference system identity and returns a multilinestring.

Syntax

```
db2gse.ST_MLineFromWKB(WKBMultiLineString Blob(1M), cr
db2gse.coordref)
```

Return type

```
db2gse.ST_MultiLineString
```

Examples

The following code fragment populates the WATERWAYS table with a unique id, a name, and a water multilinestring.

The WATERWAYS table is created with the ID and NAME columns, which identify each stream and river system stored in the table. The WATER column is a multilinestring because the river and stream systems are often an aggregate of several linestrings.

```
CREATE TABLE WATERWAYS (id          integer,
                          name       varchar(128),
                          water      db2gse.ST_MultiLineString);

/* Create the SQL insert statement to populate the id, name and
   multilinestring. The question marks are parameter markers that
   indicate the id, name and water values that will be retrieved at
   runtime. */
strcpy (shp_sql, "insert into WATERWAYS (id,name,water)
values (?,?, db2gse.ST_MLineFromWKB (cast(? as blob(1m)),
db2gse.coordref()..srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)shp_sql, SQL_NTS);

/* Bind the integer id value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
SQL_INTEGER, 0, 0, &id, 0, &pcbvalue1);

/* Bind the varchar name value to the second parameter. */
pcbvalue2 = name_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, name_len, 0, &name, name_len, &pcbvalue2);

/* Bind the shape to the third parameter. */
pcbvalue3 = blob_len;
```

```
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,  
    SQL_BLOB, blob_len, 0, water_shape, blob_len, &pcbvalue3);  
  
/* Execute the insert statement. */  
rc = SQLExecute (hstmt);
```

ST_MPointFromText

ST_MPointFromText takes a well-known text representation of type multipoint and a spatial reference system identity and returns a multipoint.

Syntax

```
db2gse.ST_MPointFromText(multiPointTaggedText Varchar(4000), cr
db2gse.coordref)
```

Return type

```
db2gse.ST_MultiPoint
```

Examples

The following CREATE TABLE statement creates the MULTIPOINT_TEST table with a single multipoint column, MPT1.

```
CREATE TABLE MULTIPOINT_TEST (mpt1 db2gse.ST_MultiPoint)
```

The following INSERT statement inserts a multipoint into the MPT1 column by using the ST_MPointFromText column.

```
INSERT INTO MULTIPOINT_TEST
VALUES (1, db2gse.ST_MPointFromText('multipoint(10.01 20.03,10.52 40.11,
                                     30.29 41.56,31.78 10.74)',
                                     db2gse.coordref()..srid(0)))
```

ST_MPointFromWKB

ST_MPointFromWKB takes a well-known binary representation of type multipoint and a spatial reference system identity to return a multipoint.

Syntax

```
db2gse.ST_MPointFromWKB(WKBMultiPoint Blob(1M), cr db2gse.coordref)
```

Return type

```
db2gse.ST_MultiPoint
```

Examples

The following code fragment populates the SPECIES_SITINGS table.

The SPECIES_SITINGS table is created with three columns. The SPECIES and GENUS columns uniquely identify each row, while the SITINGS multipoint column stores the locations of the species sitings.

```
CREATE TABLE SPECIES_SITINGS (species varchar(32),
                               genus varchar(32),
                               sitings db2gse.ST_MultiPoint);

/* Create the SQL insert statement to populate the species, genus and
   sitings. The question marks are parameter markers that
   indicate the species, genus and sitings values that will be retrieved at
   runtime. */
strcpy (wkb_sql,"insert into SPECIES_SITINGS (species,genus,sitings)
values (?,?, db2gse.ST_MPointFromWKB (cast(? as blob(1m)),
db2gse.coordref(..srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);

/* Bind the varchar species value to the first parameter. */
pcbvalue1 = species_len;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, species_len, 0, &species, species_len, &pcbvalue1);
/* Bind the varchar genus value to the second parameter. */
pcbvalue2 = genus_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, genus_len, 0, &name, genus_len, &pcbvalue2);

/* Bind the shape to the third parameter. */
pcbvalue3 = sitings_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
SQL_BLOB, sitings_len, 0, sitings_wkb, sitings_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

ST_MPolyFromText

ST_MPolyFromText takes a well-known text representation of type multipolygon and a spatial reference system identity and returns a multipolygon.

Syntax

```
db2gse.ST_MPolyFromText(multiPolygonTaggedText Varchar(4000), cr
db2gse.coordref)
```

Return type

```
db2gse.ST_MultiPolygon
```

Examples

The following CREATE TABLE statement creates the MULTIPOLYGON_TEST table, which has a single multipolygon column, MPL1.

```
CREATE TABLE MULTIPOLYGON_TEST (mpl1 db2gse.ST_MultiPolygon)
```

The following INSERT statement inserts the a multipolygon into the MPL1 column using the ST_MPolyFromText function.

```
INSERT INTO MULTIPOLYGON_TEST VALUES (
db2gse.ST_MPolyFromText('multipolygon(((10.01 20.03,10.52 40.11,
30.29 41.56,31.78 10.74,10.01 20.03),(21.23 15.74,21.34 35.21,28.94 35.35,
29.02 16.83, 21.23 15.74)),((40.91 10.92,40.56 20.19,
50.01 21.12,51.34 9.81, 40.91 10.92)))',
db2gse.coordref()..srid(0)))
```

ST_MPolyFromWKB

ST_MPolyFromWKB takes a well-known binary representation of type multipolygon and a spatial reference system identity and returns a multipolygon.

Syntax

```
db2gse.ST_MPolyFromWKB(WKBMultiPolygon Blob(1M), cr db2gse.coordref)
```

Return type

```
db2gse.ST_MultiPolygon
```

Examples

The following code fragment populates the LOTS table.

The LOTS table stores the LOT_ID, which uniquely identifies each lot, and the LOT multipolygon, which contains the lot line geometry.

```
CREATE TABLE LOTS (lot_id integer, lot db2gse.ST_MultiPolygon);
```

```
/* Create the SQL insert statement to populate the lot_id, and lot. The
   question marks are parameter markers that indicate the lot_id, and lot
   values that will be retrieved at runtime. */
strcpy (wkb_sql,"insert into LOTS (lot_id,lot)
values (?, db2gse.ST_MPolyFromWKB (cast(? as blob(1m)),
db2gse.coordref(..srid(0)))");
```

```
/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);
```

```
/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);
```

```
/* Bind the lot_id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
SQL_INTEGER, 0, 0, &lot_id, 0, &pcbvalue1);
```

```
/* Bind the lot shape to the second parameter. */
pcbvalue2 = lot_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
SQL_BLOB, lot_len, 0, lot_wkb, lot_len, &pcbvalue2);
```

```
/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

ST_NumGeometries

ST_NumGeometries takes a collection and returns the number of geometries in the collection.

Syntax

```
db2gse.ST_NumGeometries(g db2gse.ST_GeomCollection)
```

Return type

Integer

Examples

The city engineer needs to know the actual number of distinct buildings associated with each building footprint.

The building footprints are stored in the BUILDINGFOOTPRINTS table that was created with the following CREATE TABLE statement.

```
CREATE TABLE BUILDINGFOOTPRINTS (  building_id integer,
                                   lot_id      integer,
                                   footprint   db2gse.ST_MultiPolygon);
```

The following SELECT statement uses the ST_NumGeometries function to list the BUILDING_ID that uniquely identifies each building and the number of buildings in each footprint.

```
SELECT building_id, db2gse.ST_NumGeometries (footprint) "Number of buildings"
FROM BUILDINGFOOTPRINTS;
```

ST_NumInteriorRing

ST_NumInteriorRing takes a polygon and returns the number of its interior rings.

Syntax

```
db2gse.NumInteriorRing(p db2gse.ST_Polygon)
```

Return type

Integer

Examples

An ornithologist, wishing to study a bird population on several south sea islands, knows that the feeding zone of a particular species is restricted to islands containing fresh water lakes. Therefore, she wants to know which islands contain one or more lakes.

The following CREATE TABLE statement creates the ISLANDS table. The ID and NAME columns of the ISLANDS table identify each island, and the LAND polygon column stores the island's geometry.

```
CREATE TABLE ISLANDS (id integer, name varchar(32), land db2gse.ST_Polygon);
```

Because interior rings represent the lakes, the ST_NumInteriorRing function is used to list only those islands that have at least one interior ring.

```
SELECT name FROM ISLANDS WHERE db2gse.ST_NumInteriorRing(land) > 0;
```

ST_NumPoints

ST_NumPoints takes a linestring and returns its number of points.

Syntax

```
db2gse.ST_NumPoints(l db2gse.ST_LineString)
```

Return type

Integer

Examples

The following CREATE TABLE statement creates the NUMPOINTS_TEST table. The GEOTYPE column contains the geometry type stored in the G1 geometry column.

```
CREATE TABLE NUMPOINTS_TEST (geotype varchar(12), g1 db2gse.ST_Geometry)
```

The following INSERT statement inserts a linestring.

```
INSERT INTO NUMPOINTS_TEST VALUES( linestring,  
db2gse.ST_LineFromText('linestring (10.02 20.01, 23.73 21.92)',  
db2gse.coordref()..srid(0)))
```

The following SELECT statement and the corresponding result set lists the geometry type and the number of points contained within each.

```
SELECT geotype, db2gse.ST_NumPoints(g1)  
FROM NUMPOINTS_TEST
```

```
GEOTYPE      Number of points  
-----  
ST_linestring      2  
1 record(s) selected.
```

ST_OrderingEquals

ST_OrderingEquals compares two geometries and returns 1 (TRUE) if the geometries are equal and the coordinates are in the same order; otherwise it returns 0 (FALSE).

Syntax

```
db2gse.ST_OrderingEquals(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Integer

Examples

The following CREATE TABLE statement creates the LINESTRING_TEST table, which has two linestring columns, L1 and L2.

```
CREATE TABLE LINESTRING_TEST (l1 integer, l1 db2gse.ST_LineString,  
l2 db2gse.ST_LineString);
```

The following INSERT statement inserts two linestrings into L1 and L2 that are equal and have the same coordinate ordering.

```
INSERT INTO linestring_test VALUES (1,  
db2gse.LineFromText('linestring (10.01 20.02, 21.50 12.10)',  
db2gse.coordref(..srid(0)),  
db2gse.LineFromText('linestring (10.01 20.02, 21.50 12.10)',  
db2gse.coordref(..srid(0))));
```

The following INSERT statement inserts two linestrings into L1 and L2 that are equal but do not have the same coordinate ordering.

```
INSERT INTO linestring_test VALUES (2,  
db2gse.LineFromText('linestring (10.01 20.02, 21.50 12.10)',  
db2gse.coordref(..srid(0)),  
db2gse.LineFromText('linestring (21.50 12.10,10.01 20.02)',  
db2gse.coordref(..srid(0))));
```

The following SELECT statement and corresponding result set shows how the ST_Equals function returns 1 (TRUE) regardless of the order of the coordinates. The ST_OrderingEquals function returns 0 (FALSE) if the geometries are not both equal and have the same coordinate ordering.

```
SELECT l1d, db2gse.ST_Equals(l1,l2) equals, db2gse.ST_OrderingEquals(l1,l2)  
OrderingEquals  
FROM linestring_test
```

l1d	equals	OrderingEquals
1	1	1
2	1	0

ST_Overlaps

ST_Overlaps takes two geometry objects and returns 1 (TRUE) if the intersection of the objects results in a geometry object of the same dimension but not equal to either source objects; otherwise it returns 0 (FALSE).

Syntax

```
db2gse.ST_Overlaps(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Integer

Examples

The county supervisor needs a list of hazardous waste sites whose five-mile radius overlaps sensitive areas.

The following CREATE TABLE statement creates the SENSITIVE_AREAS table. The SENSITIVE_AREAS table contains several columns that describe the threatened institutions in addition to the ZONE column, which stores the institution's polygon geometry.

```
CREATE TABLE SENSITIVE_AREAS (id          integer,
                               name        varchar(128),
                               size        float,
                               type        varchar(10),
                               zone        db2gse.ST_Polygon);
```

The HAZARDOUS_SITES table stores the identity of the sites in the SITE_ID and NAME columns, while the actual geographic location of each site is stored in the LOCATION point column.

```
CREATE TABLE HAZARDOUS_SITES (site_id   integer,
                               name        varchar(128),
                               location    db2gse.ST_Point);
```

In the following SELECT statement, the SENSITIVE_AREAS and HAZARDOUS_SITES tables are joined by the ST_Overlaps function. It returns 1 (TRUE) for all rows in the SENSITIVE_AREAS table whose zone polygons overlap the buffered five-mile radius of the HAZARDOUS_SITES location point.

```
SELECT hs.name
FROM HAZARDOUS_SITES hs, SENSITIVE_AREAS sa
WHERE db2gse.ST_Overlaps (buffer(hs.location,(5 * 5280)),sa.zone) = 1;
```

In Figure 37 on page 247, the hospital and the school overlap the five-mile radius of the county's two hazardous waste sites, while the nursing home does not.

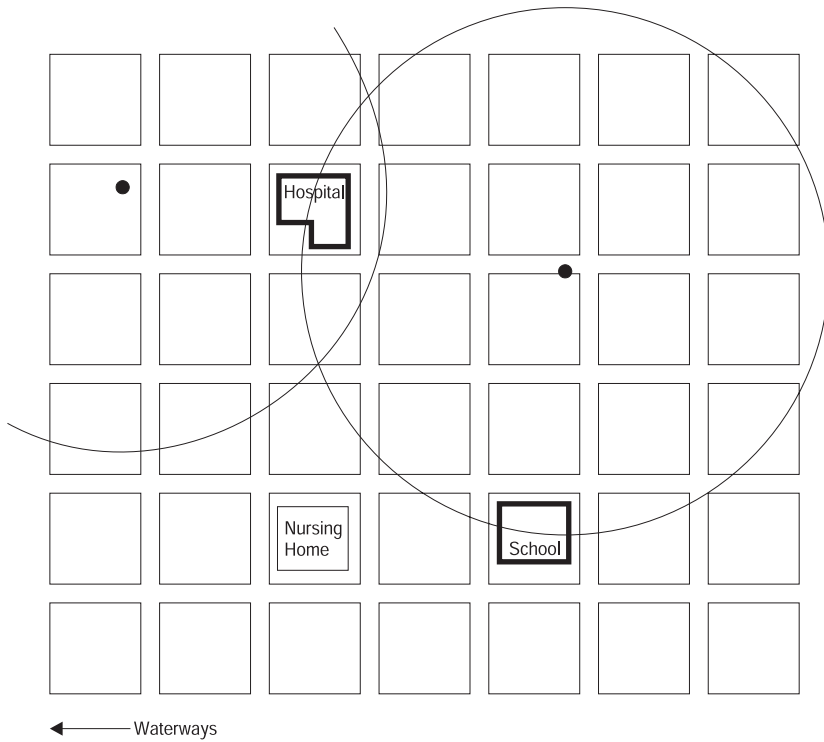


Figure 37. Using *ST_Overlaps* to determine the buildings that are at least partially within of a hazardous waste area

ST_Perimeter

ST_Perimeter returns the perimeter of an ST_Surface.

Syntax

```
db2gse.ST_Perimeter(s db2gse.ST_Surface)
db2gse.ST_Perimeter(ms db2gse.ST_MultiSurface)
```

Return type

Double

Examples

An ecologist studying shoreline birds needs to determine the shoreline for the lakes within a particular area. The lakes are stored as multipolygons in the WATERBODIES table that was created with the following CREATE TABLE statement.

```
CREATE TABLE WATERBODIES (wbid integer,
                           waterbody db2gse.ST_MultiPolygon);
```

In the following SELECT statement, the ST_Perimeter function returns the perimeter surrounding each body of water, while the SUM function aggregates the perimeters to return their total.

```
SELECT SUM(db2gse.ST_Perimeter(waterbody))
FROM waterbodies;
```

ST_PointFromText

ST_PointFromText takes a well-known text representation of type point and a spatial reference system identity and returns a point.

Syntax

```
db2gse.ST_PointFromText(pointTaggedText Varchar(4000), cr db2gse.coordref)
```

Return type

```
db2gse.ST_Point
```

Examples

The following CREATE TABLE statement creates the POINT_TEST table, which has a single point column, PT1.

```
CREATE TABLE POINT_TEST (pt1 db2gse.ST_Point)
```

Before the INSERT statement inserts the point into the PT1 column, the ST_PointFromText function converts the point text coordinates to the point format.

```
INSERT INTO POINT_TEST VALUES (  
    db2gse.ST_PointFromText ('point(10.01 20.03)',  
        db2gse.coordref()..srid(0))
```

ST_PointFromWKB

ST_PointFromWKB takes a well-known binary representation of type point and a spatial reference system identity to return a point.

Syntax

```
db2gse.ST_PointFromWKB(WKBPoint Blob(1M), srs SRID)
```

Return type

```
db2gse.ST_Point
```

Examples

The following code fragment populates the HAZARDOUS_SITES table.

The hazardous sites are stored in the HAZARDOUS_SITES table created with the following CREATE TABLE statement. The LOCATION column, defined as a point, stores a location that is the geographic center of each hazardous site.

```
CREATE TABLE HAZARDOUS_SITES (site_id integer,
                               name      varchar(128),
                               location  db2gse.ST_Point);

/* Create the SQL insert statement to populate the site_id, name and
   location. The question marks are parameter markers that indicate the
   site_id, name and location values that will be retrieved at runtime. */
strcpy (wkb_sql,"insert into HAZARDOUS_SITES (site_id, name, location)
values (?,?, db2gse.ST_PointFromWKB(cast(? as blob(1m)),
db2gse.coordref(..srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);

/* Bind the site_id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
SQL_INTEGER, 0, 0, &site_id, 0, &pcbvalue1);

/* Bind the name varchar value to the second parameter. */
pcbvalue2 = name_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, 0, 0, name, 0, &pcbvalue2);

/* Bind the location shape to the third parameter. */
pcbvalue3 = location_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
SQL_BLOB, location_len, 0, location_wkb, location_len, &pcbvalue3);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

ST_Point

ST_Point returns an ST_Point, given an x-coordinate, y-coordinate, and spatial reference.

Syntax

```
db2gse.ST_Point(X Double, Y Double, srs SRID)
```

Return type

```
db2gse.ST_Point
```

Examples

The following CREATE TABLE statement creates the POINT_TEST table, which has a single point column, PT1.

```
CREATE TABLE POINT_TEST (pt1 db2gse.ST_Point)
```

The ST_Point function converts the point coordinates into a point geometry before the INSERT statement inserts it into the PT1 column.

```
INSERT INTO point_test VALUES(  
    db2gse.ST_Point(10.01,20.03, db2gse.coordref()..srid(0))  
)
```

ST_PointN

ST_PointN takes a linestring and an integer index and returns a point that is the nth vertex in the linestring's path.

Syntax

```
db2gse.ST_PointN(l db2gse.ST_Curve, n Integer)
```

Return type

```
db2gse.ST_Point
```

Examples

The following CREATE TABLE statement creates the POINTN_TEST table, which has two columns: the GID column, which uniquely identifies each row, and the LN1 linestring column.

```
CREATE TABLE POINTN_TEST (gid integer, ln1 db2gse.ST_LineString)
```

The following INSERT statements insert two linestring values. The first linestring does not have Z coordinates or measures, while the second linestring has both.

```
INSERT INTO POINTN_TEST VALUES(1,
db2gse.ST_LineFromText('linestring (10.02 20.01,23.73 21.92,30.10 40.23)',
db2gse.coordref()..srid(0)))
```

```
INSERT INTO POINTN_TEST VALUES(2,
db2gse.ST_LineFromText('linestring zm (10.02 20.01 5.0 7.0,23.73 21.92 6.5 7.1,
30.10 40.23 6.9 7.2)', db2gse.coordref()..srid(0)))
```

The following SELECT statement and the corresponding result set lists the GID column and the second vertex of each linestring. The first row results in a point without a Z coordinate or measure, while the second row results in a point with a Z coordinate and a measure. The ST_PointN function returns points with a Z coordinate or a measure if they exist in the source linestring.

```
SELECT gid, CAST(db2gse.ST_AsText(db2gse.ST_PointN(ln1,2)) AS varchar(60))
"The 2nd vertice"
FROM POINTN_TEST
```

```
GID          The 2nd vertice
-----
          1 POINT ( 23.73000000 21.92000000)
          2 POINT ZM ( 23.73000000 21.92000000 7.00000000 7.10000000)
```

```
2 record(s) selected.
```

ST_PointOnSurface

ST_PointOnSurface takes both a polygon or a multipolygon and a returns an ST_Point.

Syntax

```
db2gse.ST_PointOnSurface(s db2gse.ST_Surface)
db2gse.ST_PointOnSurface(ms db2gse.ST_MultiSurface)
```

Return type

db2gse.ST_Point

Examples

The city engineer needs to create a label point for each of the building footprints.

The building footprints are stored in the BUILDINGFOOTPRINTS table that was created with the following CREATE TABLE statement.

```
CREATE TABLE BUILDINGFOOTPRINTS (
    building_id integer,
    lot_id       integer,
    footprint    db2gse.ST_MultiPolygon);
```

The ST_PointOnSurface function generates a point that is guaranteed to be on the surface of the building footprints. The ST_PointOnSurface function returns a point that the AsBinaryShape function converts to a shape casted to a 1 megabyte character string for use by the application.

```
SELECT CAST(db2gse.AsBinaryShape(db2gse.ST_PointOnSurface(footprint)) as
          blob(1m))
FROM BUILDINGFOOTPRINTS;
```

ST_PolyFromText

ST_PolyFromText takes a well-known text representation of type polygon and a spatial reference system identity and returns a polygon.

Syntax

```
db2gse.ST_PolyFromText(polygonTaggedText Varchar(4000), cr  
db2gse.coordref)
```

Return type

```
db2gse.ST_Polygon
```

Examples

The following CREATE TABLE statement creates the POLYGON_TEST table with the single polygon column.

```
CREATE TABLE POLYGON_TEST (p11 db2gse.ST_Polygon)
```

The following INSERT statement inserts a polygon into the polygon column by using the ST_PolyFromText function.

```
INSERT INTO POLYGON_TEST VALUES (1,  
db2gse.ST_PolyFromText('polygon((10.01 20.03,10.52 40.11,30.29 41.56,  
31.78 10.74,10.01 20.03))',  
db2gse.coordref()..srid(0)))
```

ST_PolyFromWKB

ST_PolyFromWKB takes a well-known binary representation of type polygon and a spatial reference system identity to return a polygon.

Syntax

db2gse.ST_PolyFromWKB(WKBPolygon Blob(1M), SRID Integer)

Return type

db2gse.ST_Polygon

Examples

The following code fragment populates the SENSITIVE_AREAS table.

The SENSITIVE_AREAS table contains several columns that describe the threatened institutions in addition to the zone column, which stores the institution's polygon geometry.

```
CREATE TABLE SENSITIVE_AREAS (id          integer,
                               name        varchar(128),
                               size        float,
                               type        varchar(10),
                               zone        db2gse.ST_Polygon);

/* Create the SQL insert statement to populate the id, name, size, type and
   zone. The question marks are parameter markers that indicate the id,name,
   size, type and zone values that will be retrieved at runtime. */
strcpy (shp_wkb,"insert into SENSITIVE_AREAS (id, name, size, type, zone)
values (?, ?, ?, ?, db2gse.ST_PolyFromWKB (cast(? as blob(1m)),
db2gse.coordref()..srid(0)))");

/* Allocate memory for the SQL statement handle and associate the
   statement handle with the connection handle. */
rc = SQLAllocStmt (handle, &hstmt);

/* Prepare the SQL statement for execution. */
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);

/* Bind the id integer value to the first parameter. */
pcbvalue1 = 0;
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTEGER,
SQL_INTEGER, 0, 0, &id, 0, &pcbvalue1);

/* Bind the name varchar value to the second parameter. */
pcbvalue2 = name_len;
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
SQL_CHAR, 0, 0, name, 0, &pcbvalue2);

/* Bind the size float to the third parameter. */
pcbvalue3 = 0;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT,
SQL_REAL, 0, 0, &size, 0, &pcbvalue3);
```

```
/* Bind the type varchar to the fourth parameter. */
pcbvalue4 = type_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR,
    SQL_VARCHAR, type_len, 0, type, type_len, &pcbvalue4);

/* Bind the zone polygon to the fifth parameter. */
pcbvalue5 = zone_len;
rc = SQLBindParameter (hstmt, 3, SQL_PARAM_INPUT, SQL_C_BINARY,
    SQL_BLOB, zone_len, 0, zone_wkb, zone_len, &pcbvalue5);

/* Execute the insert statement. */
rc = SQLExecute (hstmt);
```

ST_Polygon

ST_Polygon generates an ST_Polygon from an ST_LineString and a spatial reference system identifier.

Syntax

```
db2gse.ST_Polygon(l db2gse.ST_LineString, cr db2gse.coordref)
```

Return type

```
db2gse.ST_Polygon
```

Examples

The following CREATE TABLE statement creates the POLYGON_TEST table, which has a single column, P1.

```
CREATE TABLE POLYGON_TEST (p1 db2gse.ST_polygon)
```

The following INSERT statement converts a ring (a linestring that is both closed and simple) into a polygon and inserts it into the P1 column using the ST_LineFromText function within the ST_Polygon function.

```
INSERT INTO POLYGON_TEST VALUES (  
db2gse.ST_Polygon(db2gse.ST_LineFromText('linestring(10.01 20.03,20.94  
21.34,35.93 10.04,10.01 20.03)', db2gse.coordref()..srid(0))),  
db2gse.coordref()..srid(0))  
)
```

ST_Relate

ST_Relate compares two geometries and returns 1 (TRUE) if the geometries meet the conditions specified by the DE-9IM pattern matrix string; otherwise, 0 (FALSE) is returned. For information about DE-9IM pattern matrices, see “Predicate functions” on page 132.

Syntax

```
db2gse.ST_Relate(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry,  
patternMatrix String)
```

Return type

Integer

Examples

A DE-9IM pattern matrix is a device for comparing geometries. There are several types of such matrices. For example, the *equals* pattern matrix will tell you if any two geometries are equal.

In this example, an equals pattern matrix, shown in Table 56, is read left to right and top to bottom into a string (“T*F**FFF*”).

Table 56. Equals pattern matrix

		b		
		Interior	Boundary	Exterior
a	Interior	T	*	F
	Boundary	*	*	F
	Exterior	F	F	*

Next, the table RELATE_TEST is created with the following CREATE TABLE statement.

```
CREATE TABLE RELATE_TEST (rid integer, g1 db2gse.ST_Geometry,  
g2 db2gse.ST_Geometry, g3 db2gse.ST_Geometry);
```

The following INSERT statements insert a sample subclass into the RELATE_TEST table.

```
INSERT INTO RELATE_TEST VALUES(  
1,  
db2gse.ST_PointFromText('point (10.02 20.01)',  
db2gse.coordref()..srid(0),  
db2gse.ST_PointFromText('point (10.02 20.01)',  
db2gse.coordref()..srid(0),  
db2gse.ST_PointFromText('point (30.01 20.01)',  
db2gse.coordref()..srid(0)  
)
```


The following `SELECT` statement and the corresponding result set lists the subclass name stored in the `GEOTYPE` column with the dimension of that geotype.

```
SELECT rid, relate(g1,g2) equals, relate(g1,g3) not_equals
FROM relate_test
```

RID	equals	not_equals
1	1	0

1 record(s) selected.

ST_SRID

ST_SRID takes a geometry object and returns its spatial reference system identity.

Syntax

```
db2gse.ST_SRID(g1 db2gse.ST_Geometry)
```

Return type

Integer

Examples

During the installation of the DB2 Spatial Extender the SPATIAL_REFERENCES table is created. When a geometry is created, the SRID of that geometry is entered into the SPATIAL_REFERENCES table. The ST_SRID function returns the value of that entry.

For example, a geometry type is used in a CREATE TABLE statement:

```
CREATE TABLE SRID_TEST(g1 db2gse.ST_Geometry)
```

In the following INSERT statement, a point geometry located at coordinate 10.01,50.76 is inserted into the geometry column G1. When the point geometry was created by the ST_PointFromText function, it was assigned the srid value of 1.

```
INSERT INTO SRID_TEST
VALUES (db2gse.ST_PointFromText('point(10.01 50.76)',
    db2gse.coordref()..srid(0)))
```

The ST_SRID function returns the spatial reference system identity of the geometry just entered, as illustrated by the following SELECT statement and the corresponding result set.

```
SELECT db2gse.ST_SRID(g1) FROM SRID_TEST
```

```
g1
-----
1
```

ST_StartPoint

ST_StartPoint takes a linestring and returns a point that is the linestrings first point.

Syntax

```
db2gse.ST_StartPoint(c db2gse.ST_Curve)
```

Return type

```
db2gse.ST_Point
```

Examples

The following CREATE TABLE statement creates the STARTPOINT_TEST table. STARTPOINT_TEST has two columns: the GID integer column, which uniquely identifies the rows of the table, and the LN1 linestring column.

```
CREATE TABLE STARTPOINT_TEST (gid integer, ln1 db2gse.ST_LineString)
```

The following INSERT statements insert the linestrings into the LN1 column. The first linestring does not have Z coordinates or measures, while the second linestring has both.

```
INSERT INTO STARTPOINT_TEST VALUES(1,
db2gse.ST_LineFromText('linestring (10.02 20.01,23.73
21.92,30.10 40.23)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO STARTPOINT_TEST VALUES(2,
db2gse.ST_LineFromText('linestring zm (10.02 20.01 5.0 7.0,
23.73 21.92 6.5 7.1,30.10 40.23 6.9 7.2)',
db2gse.coordref()..srid(0)))
```

The following SELECT statement and the corresponding result set shows how the ST_StartPoint function extracts the first point of each linestring. The ST_AsText function converts the point to its text format. The first point in the list does not have a Z coordinate or a measure, while the second point has both because the source linestring did.

```
SELECT gid, CAST(db2gse.ST_AsText(db2gse.ST_StartPoint (ln1)) as varchar(60))
"Startpoint"
FROM STARTPOINT_TEST
```

```
GID          Startpoint
-----
1 POINT ( 10.02000000 20.01000000)
2 POINT ZM ( 10.02000000 20.01000000 5.00000000 7.00000000)
```

```
2 record(s) selected.
```

ST_SymmetricDiff

ST_SymmetricDiff takes two geometry objects and returns a geometry object that is the symmetrical difference of the source objects.

Syntax

```
db2gse.ST_SymmetricDiff(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The county supervisor must determine the area of sensitive areas and five-mile hazardous site radius that is not intersected.

The following CREATE TABLE statement creates the SENSITIVE_AREAS table, which contains several columns that describe the threatened institutions. The SENSITIVE_AREAS table also contains the ZONE column, which stores the institution's polygon geometry.

```
CREATE TABLE SENSITIVE_AREAS (id          integer,
                               name        varchar(128),
                               size        float,
                               type        varchar(10),
                               zone        db2gse.ST_Polygon);
```

The following CREATE TABLE statement creates the the HAZARDOUS_SITES table, which stores the identity of the sites in the SITE_ID and NAME columns, while the actual geographic location of each site is stored in the LOCATION point column.

```
CREATE TABLE HAZARDOUS_SITES (site_id   integer,
                               name       varchar(128),
                               location   point);
```

The ST_Buffer function generates a five-mile buffer surrounding the hazardous waste site locations. The ST_SymmetricDiff function generates polygons from the intersection of the buffered hazardous waste site polygons and the sensitive areas. The ST_Area function returns the intersection polygon's area for each hazardous site.

```
SELECT sa.name, hs.name,
       db2gse.ST_Area(db2gse.ST_SymmetricDiff (db2gse.ST_Buffer(hs.location,
(5 * 5280)),sa.zone))
FROM HAZARDOUS_SITES hs, SENSITIVE_AREAS sa
```

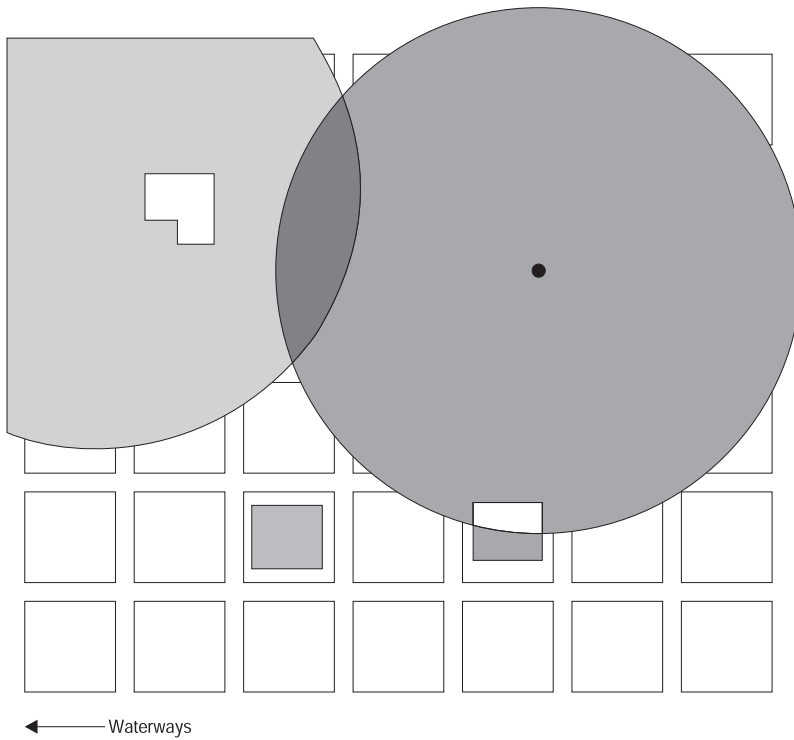


Figure 38. Using `ST_SymmetricDiff` to determine the hazardous waste areas that do not contain sensitive areas (inhabited buildings)

In Figure 38, the symmetric difference of the hazardous waste sites and the sensitive areas results in the subtraction of the intersected areas.

ST_Touches

ST_Touches returns 1 (TRUE) if none of the points common to both geometries intersect the interiors of both geometries; otherwise, it returns 0 (FALSE). At least one geometry must be a linestring, polygon, multilinestring, or multipolygon.

Syntax

```
db2gse.ST_Touches(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Integer

Examples

The GIS technician needs to provide a list of all sewer lines whose endpoints intersect another sewerline.

The following CREATE TABLE statement creates the SEWERLINES table, which has three columns. The first column, SEWER_ID, uniquely identifies each sewer line. The second column, CLASS, of type integer identifies the type of sewer line, which is generally associated with the line's capacity. The third column, SEWER, of type linestring stores the sewer line's geometry.

```
CREATE TABLE SEWERLINES (sewer_id integer, class integer, sewer  
    db2gse.ST_LineString);
```

The following SELECT statement returns an ordered list of SEWER_IDS that touch one another.

```
SELECT s1.sewer_id, s2.sewer_id  
FROM sewerlines s1, sewerlines s2  
WHERE db2gse.ST_Touches (s1.sewer, s2.sewer) = 1,  
ORDER BY 1,2;
```

ST_Transform

ST_Transform assigns a geometry to a spatial reference system other than the spatial reference system to which the geometry is currently assigned.

Syntax

```
db2gse.ST_Transform(g db2gse.ST_Geometry, cr db2gse.coordref)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The following CREATE TABLE statement creates the TRANSFORM_TEST table, which has two linestring columns, L1 and L2.

```
CREATE TABLE TRANSFORM_TEST (tid integer, l1 db2gse.ST_LineString,  
l2 db2gse.ST_LineString)
```

The following INSERT statement inserts a linestring into l1 with an SRID of 102.

```
INSERT INTO TRANSFORM_TEST VALUES (1, db2gse.ST_LineFromText('linestring  
(10.01 40.43, 92.32 29.89)',  
db2gse.coordref()..srid(102)),NULL)
```

The ST_Transform function converts the linestring of L1 from the coordinate reference assigned to SRID 102 to the coordinate reference assigned to SRID 105. The following UPDATE statement stores the transformed linestring in column l2.

```
UPDATE TRANSFORM_TEST SET l2 = db2gse.ST_Transform(l1,  
db2gse.coordref()..srid(105))
```

ST_Union

ST_Union takes two geometry objects and returns a geometry object that is the union of the source objects.

Syntax

```
db2gse.ST_Union(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

```
db2gse.ST_Geometry
```

Examples

The following CREATE TABLE statement creates the SENSITIVE_AREAS table, which contains several columns that describe the threatened institutions. The SENSITIVE_AREAS table also contains the ZONE column, which stores the institution's polygon geometry.

```
CREATE TABLE SENSITIVE_AREAS (id          integer,
                               name        varchar(128),
                               size        float,
                               type        varchar(10),
                               zone        db2gse.ST_Polygon);
```

The following CREATE TABLE statement creates the HAZARDOUS_SITES table, which stores the identity of the sites in the SITE_ID and NAME columns. The actual geographic location of each site is stored in the LOCATION point column.

```
CREATE TABLE HAZARDOUS_SITES (site_id integer, name varchar(128),
                               location db2gse.ST_Point);
```

The following SELECT statement uses the ST_Buffer function to generate a five-mile buffer surrounding the hazardous waste site locations. The ST_Union function generates polygons from the union of the buffered hazardous waste site polygons and the sensitive areas. The ST_Area function returns the union of polygon's area.

```
SELECT sa.name, hs.name,
       db2gse.ST_Area(db2gse.ST_Union(db2gse.ST_Buffer(hs.location,
(5 * 5280)),sa.zone))
FROM HAZARDOUS_SITES hs, SENSITIVE_AREAS sa;
```

ST_Within

ST_Within takes two geometry objects and returns 1 (TRUE) if the first object is completely within the second; otherwise it returns 0 (FALSE).

Syntax

```
db2gse.ST_Within(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

Return type

Integer

Examples

In the example below, two tables are created. The first table, BUILDINGFOOTPRINTS, contains a city's building footprints. The second table, LOTS, contains the city's lots. The city engineer wants to make sure that all the building footprints are completely inside their lots.

In both tables, the multipolygon data type stores the geometry of the building footprints and the lots. The database designer selected multipolygons for both features because lots can be disjointed by natural features, such as a river, and building footprints can often be made up of several buildings.

```
CREATE TABLE BUILDINGFOOTPRINTS (  building_id integer,
                                     lot_id      integer,
                                     footprint   db2gse.ST_MultiPolygon);
```

```
CREATE TABLE LOTS (  lot_id integer, lot db2gse.ST_MultiPolygon );
```

Using the following SELECT statement, the city engineer first selects the buildings that are not completely within a lot.

```
SELECT building_id
   FROM BUILDINGFOOTPRINTS, LOTS
  WHERE db2gse.ST_Within(footprint,lot) = 0;
```

Although the first query will provide a list of all BUILDING_IDS that have footprints outside of a lot polygon, it will not determine whether the rest have the correct lot_id assigned to them. This second SELECT statement performs a data integrity check on the LOT_ID column of the BUILDINGFOOTPRINTS table.

```
SELECT bf.building_id "building id",
       bf.lot_id     "buildings lot_id",
       LOTS.lot_id   "LOTS lot_id"
   FROM BUILDINGFOOTPRINTS bf, LOTS
  WHERE db2gse.ST_Within(footprint,lot) = 1 AND
        LOTS.lot_id <> bf.lot_id;
```

ST_WKBTToSQL

ST_WKBTToSQL constructs a ST_Geometry value given its well-known binary representation. An SRID value of 0 is automatically used.

Syntax

```
db2gse.ST_WKBTToSQL(WKBGeometry Blob(1M))
```

Return type

```
db2gse.ST_Geometry
```

Examples

The following CREATE TABLE statement creates the LOTS table, which has two columns: the LOT_ID column, which uniquely identifies each lot, and the LOT multipolygon column, which contains the geometry of each lot.

```
CREATE TABLE lots (lot_id integer,  
                   lot      db2gse.ST_MultiPolygon);
```

The following C code fragment contains ODBC functions embedded with DB2 Spatial Extender SQL functions that insert data into the LOTS table.

The ST_WKBTToSQL function converts WKB representations into DB2 Spatial Extender geometry. The entire INSERT statement is copied into a wkb_sql char string. The INSERT statement contains parameter markers to accept the LOT_ID data and the LOT data, dynamically.

```
/* Create the SQL insert statement to populate the lot id and the  
   lot multipolygon. The question marks are parameter markers that  
   indicate the lot_id and lot values that will be retrieved at  
   run time. */  
  
strcpy (wkb_sql,"insert into lots (lot_id, lot)  
        values(?, db2gse.ST_WKBTToSQL(cast(? as blob(1m))))");  
  
/* Allocate memory for the SQL statement handle and associate the  
   statement handle with the connection handle. */  
  
rc = SQLAllocStmt (handle, &hstmt);  
  
/* Prepare the SQL statement for execution. */  
  
rc = SQLPrepare (hstmt, (unsigned char *)wkb_sql, SQL_NTS);  
  
/* Bind the integer key value to the first parameter. */  
  
pcbvalue1 = 0;  
rc = SQLBindParameter (hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,  
                       SQL_INTEGER, 0, 0, &lot_id, 0, &pcbvalue1);  
  
/* Bind the shape to the second parameter. */  
  
pcbvalue2 = blob_len;
```

```
rc = SQLBindParameter (hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,  
    SQL_BLOB, blob_len, 0, shape_blob, blob_len, &pcbvalue2);  
  
/* Execute the insert statement. */  
  
rc = SQLExecute (hstmt);
```

ST_WKTTToSQL

ST_WKTTToSQL constructs a ST_Geometry value given its well-known textual representation. An SRID value of 0 is automatically used.

Syntax

```
db2gse.ST_WKTTToSQL(geometryTaggedText Varchar(4000))
```

Return type

```
db2gse.ST_Geometry
```

Examples

The following CREATE TABLE statement creates the GEOMETRY_TEST table, which contains two columns: the GID column of type integer, which uniquely identifies each row, and the G1 column, which stores the geometry.

```
CREATE TABLE GEOMETRY_TEST (gid smallint, g1 db2gse.ST_Geometry)
```

The following INSERT statements insert the data into the GID and G1 columns of the GEOMETRY_TEST table. The ST_WKTTToSQL function converts the text representation of each geometry into its corresponding DB2 Spatial Extender instantiable subclass.

```
INSERT INTO GEOMETRY_TEST VALUES(
1, db2gse.ST_WKTTToSQL ('point (10.02 20.01)')
)
```

```
INSERT INTO GEOMETRY_TEST VALUES(
2, db2gse.ST_WKTTToSQL('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)')
)
```

```
INSERT INTO GEOMETRY_TEST VALUES(
3, db2gse.ST_WKTTToSQL('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))')
)
```

```
INSERT INTO GEOMETRY_TEST VALUES(
4, db2gse.ST_WKTTToSQL('multipoint (10.02 20.01,10.32 23.98,11.92 25.64)')
)
```

```
INSERT INTO GEOMETRY_TEST VALUES(
5, db2gse.ST_WKTTToSQL('multilinestring ((10.02 20.01,      10.32 23.98,
11.92 25.64),(9.55 23.75,15.36 30.11))')
)
```

```
INSERT INTO GEOMETRY_TEST VALUES(
6, db2gse.ST_WKTTToSQL('multipolygon (((10.02 20.01, 11.92 35.64,
25.02 34.15, 19.15 33.94,10.02 20.01)),
((51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90, 51.71 21.73)))')
)
```

ST_X

ST_X takes a point and returns its X coordinate.

Syntax

```
ST_X(p ST_Point)
```

Return type

Double

Examples

The following CREATE TABLE statement creates the X_TEST table, which has two columns: the GID column, which uniquely identifies the row, and the PT1 point column.

```
CREATE TABLE X_TEST (gid integer, pt1 db2gse.ST_Point)
```

The following INSERT statements insert two rows. One is a point without a Z coordinate or a measure. The other column has both a Z coordinate and a measure.

```
INSERT INTO X_TEST VALUES(1,  
db2gse.ST_PointFromText('point (10.02 20.01)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO X_TEST VALUES(2,  
db2gse.ST_PointFromText('point zm (10.02 20.01 5.0 7.0)',  
db2gse.coordref()..srid(0)))
```

The following SELECT statement and the corresponding result set lists the GID column and the Double X coordinate of the points.

```
SELECT gid, db2gse.ST_X(pt1) "The X coordinate" FROM X_TEST
```

GID	The X coordinate
1	+1.0020000000000000E+001
2	+1.0020000000000000E+001

2 record(s) selected.

ST_Y

ST_Y takes a point and returns its Y coordinate.

Syntax

```
db2gse.ST_Y(p db2gse.ST_Point)
```

Return type

Double

Examples

The following CREATE TABLE statement creates the Y_TEST table, which has two columns: the GID column, which uniquely identifies the row, and the PT1 point column.

```
CREATE TABLE Y_TEST (gid integer, pt1 db2gse.ST_Point)
```

The INSERT statements insert two rows. One is a point without a Z coordinate or a measure. The other column has both a Z coordinate and a measure.

```
INSERT INTO Y_TEST VALUES(1,  
db2gse.ST_PointFromText('point (10.02 20.01)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO Y_TEST VALUES(2,  
db2gse.ST_PointFromText('point zm (10.02 20.01 5.0 7.0)',  
db2gse.coordref()..srid(0)))
```

The following SELECT statement and the corresponding result set lists the GID column and the Double Y coordinate of the points.

```
SELECT gid, db2gse.ST_Y(pt1) "The Y coordinate" FROM Y_TEST
```

```
GID          The Y coordinate  
-----  
1    +2.001000000000000E+001  
2    +2.001000000000000E+001
```

2 record(s) selected.

Z

Z takes a point and returns its Z coordinate.

Syntax

```
Z(p db2gse.ST_Point)
```

Return type

Double

Examples

The following CREATE TABLE statement creates the Z_TEST table is created with two columns: the GID column uniquely identifies the row, and the PT1 point column.

```
CREATE TABLE Z_TEST (gid integer, pt1 db2gse.ST_Point)
```

The following INSERT statements insert two rows. One is a point without a Z coordinate or a measure. The other column has both a Z coordinate and a measure.

```
INSERT INTO Z_TEST VALUES(1,  
db2gse.ST_PointFromText('point (10.02 20.01)', db2gse.coordref()..srid(0)))
```

```
INSERT INTO Z_TEST VALUES(2,  
db2gse.ST_PointFromText('point zm (10.02 20.01 5.0 7.0)',  
db2gse.coordref()..srid(0)))
```

The following SELECT statement and the corresponding result set lists the GID column and the Double Z coordinate of the points. The first row is NULL because the point does not have a Z coordinate.

```
SELECT gid, db2gse.Z(pt1) "The Z coordinate" FROM Z_TEST
```

```
GID          The Z coordinate  
-----  
1              -  
2    +5.00000000000000E+000
```

2 record(s) selected.

Chapter 15. Coordinate systems

The chapter provides reference information about the spatial reference system (SRS) and the coordinate values used to interpret spatial data.

- “Overview of coordinate systems”
- “Supported linear units” on page 277
- “Supported angular units” on page 277
- “Supported spheroids” on page 278
- “Supported geodetic datums” on page 279
- “Supported prime meridians” on page 281
- “Supported map projections” on page 282
- “Conic projections” on page 282
- “Azimuthal or planar projections” on page 283
- “Map projection parameters” on page 283

Overview of coordinate systems

The well-known text representation of spatial reference systems provides a standard textual representation for spatial reference system information. The definitions of the well-known text representation is modeled after the POSC/EPSS coordinate system data model.

A spatial reference system is a geographic (latitude-longitude), a projected (X,Y), or a geocentric (X,Y,Z) coordinate system. The coordinate system is composed of several objects. Each object has a keyword in uppercase (for example, DATUM or UNIT) followed by the comma-delimited defining parameters of the object in brackets. Some objects are composed of other objects, so the result is a nested structure.

Note: Implementations are free to substitute standard brackets () for square brackets [] and should be prepared to read both forms of brackets.

The EBNF (Extended Backus Naur Form) definition for the string representation of a coordinate system using square brackets is as follows (see note above regarding the use of brackets):

```
<coordinate system> = <projected cs> | <geographic cs> | <geocentric cs>
<projected cs> = PROJCS["<name>", <geographic cs>, <projection>, {<parameter>,*
    <linear unit>}]
<projection> = PROJECTION["<name>"]
<parameter> = PARAMETER["<name>", <value>]
<value> = <number>
```

A data set's coordinate system is identified by the PROJCS keyword if the data is in projected coordinates (by GEOGCS if in geographic coordinates, or by GEOCCS if in geocentric coordinates). The PROJCS keyword is followed by all of the "pieces" that define the projected coordinate system. The first piece of any object is always the name. Several objects follow the projected coordinate system name: The geographic coordinate system, the map projection, one or more parameters, and the linear unit of measure. All projected coordinate systems is based upon a geographic coordinate system, so this section describes the pieces specific to a projected coordinate system first. For example, UTM zone 10N on the NAD83 datum is defined:

```
PROJCS["NAD_1983_UTM_Zone_10N",
<geographic cs>,
PROJECTION["Transverse_Mercator"],
PARAMETER["False_Easting",500000.0],
PARAMETER["False_Northing",0.0],
PARAMETER["Central_Meridian",-123.0],
PARAMETER["Scale_Factor",0.9996],
PARAMETER["Latitude_of_Origin",0.0],
UNIT["Meter",1.0]]
```

The name and several objects define the geographic coordinate system object in turn: the datum, the prime meridian, and the angular unit of measure.

```
<geographic cs> = GEOGCS["<name>", <datum>, <prime meridian>, <angular unit>]
<datum> = DATUM["<name>", <spheroid>]
<spheroid> = SPHEROID["<name>", <semi-major axis>, <inverse flattening>]
<semi-major axis> = <number>
    (semi-major axis is measured in meters and must be > 0.)
<inverse flattening> = <number>
<prime meridian> = PRIMEM["<name>", <longitude>]
<longitude> = <number>
```

The geographic coordinate system string for UTM zone 10 on NAD83:

```
GEOGCS["GCS_North_American_1983",
DATUM["D_North_American_1983",
SPHEROID["GRS_1980",6378137,298.257222101]],
PRIMEM["Greenwich",0],
UNIT["Degree",0.0174532925199433]]
```

The UNIT object can represent angular or linear unit of measures:

```
<angular unit> = <unit>
<linear unit> = <unit>
<unit> = UNIT["<name>", <conversion factor>]
<conversion factor> = <number>
```

The conversion factor specifies number of meters (for a linear unit) or number of radians (for an angular unit) per unit and must be greater than zero.

So the full string representation of UTM Zone 10N is as follows:

```

PROJCS["NAD_1983_UTM_Zone_10N",
GEOGCS["GCS_North_American_1983",
DATUM["D_North_American_1983",SPHEROID["GRS_1980",6378137,298.257222101]],
PRIMEM["Greenwich",0],UNIT["Degree",0.0174532925199433]],
PROJECTION["Transverse_Mercator"],PARAMETER["False_Easting",500000.0],
PARAMETER["False_Northing",0.0],PARAMETER["Central_Meridian",-123.0],
PARAMETER["Scale_Factor",0.9996],PARAMETER["Latitude_of_Origin",0.0],
UNIT["Meter",1.0]]

```

A geocentric coordinate system is quite similar to a geographic coordinate system:

```
<geocentric cs> = GEOCCS["<name>", <datum>, <prime meridian>, <linear unit>]
```

Supported linear units

Table 57. Supported linear units

Unit	Conversion factor
Meter	1.0
Foot (International)	0.3048
U.S. Foot	12/39.37
Modified American Foot	12.0004584/39.37
Clarke's Foot	12/39.370432
Indian Foot	12/39.370141
Link	7.92/39.370432
Link (Benoit)	7.92/39.370113
Link (Sears)	7.92/39.370147
Chain (Benoit)	792/39.370113
Chain (Sears)	792/39.370147
Yard (Indian)	36/39.370141
Yard (Sears)	36/39.370147
Fathom	1.8288
Nautical Mile	1852.0

Supported angular units

Table 58. Supported angular units

Unit	Conversion factor
Radian	1.0
Decimal Degree	p/180

Table 58. Supported angular units (continued)

Unit	Conversion factor
Decimal Minute	$(p/180)/60$
Decimal Second	$(p/180)/36000$
Gon	$p/200$
Grad	$p/200$

Supported spheroids

Table 59. Supported spheroids

Name	Semi-major axis	Inverse flattening
Airy	6377563.396	299.3249646
Modified Airy	6377340.189	299.3249646
Australian	6378160	298.25
Bessel	6377397.155	299.1528128
Modified Bessel	6377492.018	299.1528128
Bessel (Namibia)	6377483.865	299.1528128
Clarke 1866	6378206.4	294.9786982
Clarke 1866 (Michigan)	6378693.704	294.978684677
Clarke 1880	6378249.145	293.465
Clarke 1880 (Arc)	6378249.145	293.466307656
Clarke 1880 (Benoit)	6378300.79	293.466234571
Clarke 1880 (IGN)	6378249.2	293.46602
Clarke 1880 (RGS)	6378249.145	293.465
Clarke 1880 (SGA)	6378249.2	293.46598
Everest 1830	6377276.345	300.8017
Everest 1975	6377301.243	300.8017
Everest (Sarawak and Sabah)	6377298.556	300.8017
Modified Everest 1948	6377304.063	300.8017
Fischer 1960	6378166	298.3
Fischer 1968	6378150	298.3
Modified Fischer (1960)	6378155	298.3
GEM10C	6378137	298.257222101
GRS 1980	6378137	298.257222101

Table 59. Supported spheroids (continued)

Name	Semi-major axis	Inverse flattening
Hayford 1909	6378388	297.0
Helmert 1906	6378200	298.3
Hough	6378270	297.0
International 1909	6378388	297.0
International 1924	6378388	297.0
New International 1967	6378157.5	298.2496
Krasovsky	6378245	298.3
Mercury 1960	6378166	298.3
Modified Mercury 1968	6378150	298.3
NWL9D	6378145	298.25
OSU_86F	6378136.2	298.25722
OSU_91A	6378136.3	298.25722
Plessis 1817	6376523	308.64
South American 1969	6378160	298.25
Southeast Asia	6378155	298.3
Sphere (radius = 1.0)	1	0
Sphere (radius = 6371000 m)	6371000	0
Sphere (radius = 6370997 m)	6370997	0
Struve 1860	6378297	294.73
Walbeck	6376896	302.78
War Office	6378300.583	296
WGS 1960	6378165	298.3
WGS 1966	6378145	298.25
WGS 1972	6378135	298.26
WGS 1984	6378137	298.257223563

Supported geodetic datums

Table 60. Supported geodetic datums

Adindan	Lisbon
Afgooye	Loma Quintana
Agadez	Lome

Table 60. Supported geodetic datums (continued)

Australian Geodetic Datum 1966	Luzon 1911
Australian Geodetic Datum 1984	Mahe 1971
Ain el Abd 1970	Makassar
Amersfoort	Malongo 1987
Aratu	Manoca
Arc 1950	Massawa
Arc 1960	Merchich
Ancienne Triangulation Francaise	Militar-Geographische Institute
Barbados	Mhast
Batavia	Minna
Beduaram	Monte Mario
Beijing 1954	M'poraloko
Reseau National Belge 1950	NAD Michigan
Reseau National Belge 1972	North American Datum 1927
Bermuda 1957	North American Datum 1983
Bern 1898	Nahrwan 1967
Bern 1938	Naparima 1972
Bogota	Nord de Guerre
Bukit Rimpah	NGO 1948
Camacupa	Nord Sahara 1959
Campo Inchauspe	NSWC 9Z-2
Cape	Nouvelle Triangulation Francaise
Carthage	New Zealand Geodetic Datum 1949
Chua	OS (SN) 1980
Conakry 1905	OSGB 1936
Corrego Alegre	OSGB 1970 (SN)
Cote d'Ivoire	Padang 1884
Datum 73	Palestine 1923
Deir ez Zor	Pointe Noire
Deutsche Hauptdreiecksnetz	Provisional South American Datum 1956
Douala	Pulkovo 1942
European Datum 1950	Qatar
European Datum 1987	Qatar 1948

Table 60. Supported geodetic datums (continued)

Egypt 1907	Qornoq
European Reference System 1989	RT38
Fahud	South American Datum 1969
Gandajika 1970	Sapper Hill 1943
Garoua	Schwarzeck
Geocentric Datum of Australia 1994	Segora
Guyane Francaise	Serindung
Herat North	Stockholm 1938
Hito XVIII 1963	Sudan
Hu Tzu	Shan Tananarive 1925
Hungarian Datum 1972	Timbalai 1948
Indian 1954	TM65
Indian 1975	TM75
Indonesian Datum 1974	Tokyo
Jamaica 1875	Trinidad 1903
Jamaica 1969	Trucial Coast 1948
Kalianpur	Voirol 1875
Kandawala	Voirol Unifie 1960
Kertau	WGS 1972
Kuwait Oil Company	WGS 1972 Transit Broadcast Ephemeris
La Canoa	WGS 1984
Lake	Yacare
Leigon	Yoff
Liberia 1964	Zanderij

Supported prime meridians

Table 61. Supported prime meridians

Location	Coordinates
Greenwich	0° 0' 0"
Bern	7° 26' 22.5" E
Bogota	74° 4' 51.3" W
Brussels	4° 22' 4.71" E

Table 61. Supported prime meridians (continued)

Location	Coordinates
Ferro	17° 40' 0" W
Jakarta	106° 48' 27.79" E
Lisbon	9° 7' 54.862" W
Madrid	3° 41' 16.58" W
Paris	2° 20' 14.025" E
Rome	12° 27' 8.4" E
Stockholm	18° 3' 29" E

Supported map projections

Table 62. Supported map projections

Cylindrical projections	Pseudocylindrical projections
Behrmann	Craster parabolic
Cassini	Eckert I
Cylindrical equal area	Eckert II
Equirectangular	Eckert III
Gall's stereographic	Eckert IV
Gauss-Kruger	Eckert V
Mercator	Eckert VI
Miller cylindrical	McBryde-Thomas flat polar quartic
Oblique	Mercator (Hotine) Mollweide
Plate-Carée	Robinson
Times	Sinusoidal (Sansom-Flamsteed)
Transverse Mercator	Winkel I

Conic projections

Table 63. Conic projections

Albers conic equal-area	Chamberlin trimetric
Bipolar oblique conformal conic	Two-point equidistant
Bonne	Hammer-Aitoff equal-area
Equidistant conic	Van der Grinten I

Table 63. Conic projections (continued)

Lambert conformal conic	Miscellaneous
Polyconic	Alaska series E
Simple conic	Alaska Grid (Modified-Stereographic by Snyder)

Azimuthal or planar projections

- Azimuthal equidistant
- General vertical near-side perspective
- Gnomonic
- Lambert Azimuthal equal-area
- Orthographic
- Polar-Stereographic
- Stereographic

Map projection parameters

Table 64. Map projection parameters

Parameter	Description
central_meridian	The line of longitude chosen as the origin of x-coordinates.
scale_factor	Used generally to reduce the amount of distortion in a map projection.
standard_parallel_1	A line of latitude that has no distortion generally. Also used for "latitude of true scale."
standard_parallel_2	A line of latitude that has no distortion generally.
longitude_of_center	The longitude that defines the center point of the map projection.
latitude_of_center	The latitude that defines the center point of the map projection.
latitude_of_origin	The latitude chosen as the origin of y-coordinates.
false_easting	Added to x-coordinates. Used to give positive values.
false_northing	Added to y-coordinates. Used to give positive values.

Table 64. Map projection parameters (continued)

Parameter	Description
azimuth	The angle east of north that defines the center line of an oblique projection.
longitude_of_point_1	The longitude of the first point needed for a map projection.
latitude_of_point_1	The latitude of the first point needed for a map projection.
longitude_of_point_2	The longitude of the second point needed for a map projection.
latitude_of_point_2	The latitude of the second point needed for a map projection.
longitude_of_point_3	The longitude of the third point needed for a map projection.
latitude_of_point_3	The latitude of the third point needed for a map projection.
landsat_number	The number of a Landsat satellite.
path_number	The orbital path number for a particular satellite.
perspective_point_height	The height above the earth of the perspective point of the map projection.
fipszone	State Plane Coordinate System zone number.
zone	UTM zone number.

Chapter 16. File formats for spatial data

This chapter documents the DB2 Spatial Extender well-known representations. The representations are described as *well-known* because they are provided by ESRI and are not specific to DB2 Spatial Extender. Three kinds of spatial values are important to understand for the importing and exporting of spatial data:

- The Open GIS Consortium (OGC) well-known text representations
- The OGC well-known binary (WKB) representations
- The ESRI shape representations

The OGC well-known text representations

DB2 Spatial Extender has several functions that generate geometries from text descriptions:

ST_GeomFromText

Creates a geometry from a text representation of any geometry type.

ST_PointFromText

Creates a point from a point text representation.

ST_LineFromText

Creates a linestring from a linestring text representation.

ST_PolyFromText

Creates a polygon from a polygon text representation.

ST_MPointFromText

Creates a multipoint from a multipoint text representations.

ST_MLineFromText

Creates a multilinestring from a multilinestring text representation.

ST_MPolyFromText

Creates a multipolygon from a multipolygon text representation.

The text representation is an ASCII text format string that allows geometry to be exchanged in ASCII text form. You can use these functions in a third- or fourth-generation language (3GL or 4GL) program because they don't require the definitions of any special program structures. The `ST_AsText` function converts an existing geometry into a text representation.

Each geometry type has a well-known text representation, which can be used both to construct new instances of the type and to convert existing instances to textual form for alphanumeric display.

The well-known text representation of a geometry is defined as follows: the notation {}* denotes 0 or more repetitions of the tokens within the braces; the braces do not appear in the output token list.

```
<Geometry Tagged Text> :=
| <Point Tagged Text>
| <LineString Tagged Text>
| <Polygon Tagged Text>
| <MultiPoint Tagged Text>
| <MultiLineString Tagged Text>
| <MultiPolygon Tagged Text>

<Point Tagged Text> :=
POINT <Point Text>

<LineString Tagged Text> :=
LINESTRING <LineString Text>

<Polygon Tagged Text> :=
POLYGON <Polygon Text>

<MultiPoint Tagged Text> :=
MULTIPOINT <Multipoint Text>

<MultiLineString Tagged Text> :=
MULTILINESTRING <MultiLineString Text>

<MultiPolygon Tagged Text> :=
MULTIPOLYGON <MultiPolygon Text>

<Point Text> := EMPTY
| <Point>
| Z <PointZ>
| M <PointM>
| ZM <PointZM>

<Point> := <x> <y>
<x> := double precision literal
<y> := double precision literal
<PointZ> := <x> <y> <z>
<x> := double precision literal
<y> := double precision literal
<z> := double precision literal
<PointM> := <x> <y> <m>
<x> := double precision literal
<y> := double precision literal
<m> := double precision literal
<PointZM> := <x> <y> <z> <m>
<x> := double precision literal
<y> := double precision literal
```

```

<z> := double precision literal
<m> := double precision literal

<LineString Text> := EMPTY
| ( <Point Text > {, <Point Text> }* )
| Z ( <PointZ Text > {, <PointZ Text> }* )
| M ( <PointM Text > {, <PointM Text> }* )
| ZM ( <PointZM Text > {, <PointZM Text> }* )

<Polygon Text> := EMPTY
| ( <LineString Text > {,< LineString Text > }* )

<Multipoint Text> := EMPTY
| ( <Point Text > {, <Point Text > }* )

<MultiLineString Text> := EMPTY
| ( <LineString Text > {,< LineString Text>}* )

<MultiPolygon Text> := EMPTY
| ( < Polygon Text > {, < Polygon Text > }* )

```

The basic function syntax is:

```
function (<text description>,<SRID>)
```

The SRID, the spatial reference identifier, and primary key to the SPATIAL_REFERENCES table, identifies the geometry's spatial reference system that is stored in the SPATIAL_REFERENCES table. Before a geometry is inserted into a spatial column, its SRID must match the SRID of the spatial column.

The text description is made up of three basic components that are enclosed in single quotation marks, for example:

```
<'geometry type'> ['coordinate type'] ['coordinate list']
```

where:

geometry type

Is one of the following: point, linestring, polygon, multipoint, multilinestring, or multipolygon.

coordinate type

Specifies whether or not the geometry has Z coordinates or measures. Leave this argument blank if the geometry has neither. Otherwise, set the coordinate type to Z for geometries containing Z coordinates, M for geometries with measures, and ZM for geometries that have both.

coordinate list

Defines the vertices of the geometry. Coordinate lists are comma delimited and enclosed by parentheses. Geometries with multiple

components require sets of parentheses to enclose each component part. If the geometry is empty, the EMPTY keyword replaces the coordinate.

Table 65 shows a complete list of examples of all possible text representations.

Table 65. Geometry types and their text representations

Geometry type	Text Description	Comment
point	point empty	empty point
point	point z empty	empty point with z coordinate
point	point m empty	empty point with measure
point	point zm empty	empty point with z coordinate and measure
point	point (10.05 10.28)	point
point	point z (10.05 10.28 2.51)	point with z coordinate
point	point m (10.05 10.28 4.72)	point with measure
point	point zm (10.05 10.28 2.51 4.72)	point with z coordinate and measure
linestring	linestring empty	empty linestring
linestring	linestring z empty	empty linestring with z coordinates
linestring	linestring m empty	empty linestring with measures
linestring	linestring zm empty	empty linestring with z coordinates and measures
linestring	linestring (10.05 10.28 , 20.95 20.89)	linestring
linestring	linestring z (10.05 10.28 3.09, 20.95 31.98 4.72, 21.98 29.80 3.51)	linestring with z coordinates
linestring	linestring m (10.05 10.28 5.84, 20.95 31.98 9.01, 21.98 29.80 12.84)	linestring with measures
linestring	linestring zm ()	linestring with z coordinates and measures
polygon	polygon empty	empty polygon
polygon	polygon z empty	empty polygon with z coordinates

Table 65. Geometry types and their text representations (continued)

Geometry type	Text Description	Comment
polygon	polygon m empty	empty polygon with measures
polygon	polygon zm empty	empty polygon with z coordinates and measures
polygon	polygon ((10 10, 10 20, 20 20, 20 15, 10 10))	polygon
polygon	polygon z (())	polygon with z coordinates
polygon	polygon m (())	polygon with measures
polygon	polygon zm (())	polygon with z coordinates and measures
multipoint	multipoint empty	empty multipoint
multipoint	multipoint z empty	empty multipoint with z coordinates
multipoint	multipoint m empty	empty multipoint with measures
multipoint	multipoint zm empty	empty multipoint with z coordinates with measures
multipoint	multipoint empty	empty multipoint
multipoint	multipoint (10 10, 20 20)	multipoint with two points
multipoint	multipoint z (10 10 2, 20 20 3)	multipoint with z coordinates
multipoint	multipoint m (10 10 4, 20 20 5)	multipoint with measures
multipoint	multipoint zm (10 10 2 4, 20 20 3 5)	multipoint with z coordinates and measures
multilinestring	multilinestring empty	empty multilinestring
multilinestring	multilinestring z empty	empty multilinestring with z coordinates
multilinestring	multilinestring m empty	empty multilinestring with measures
multilinestring	multilinestring zm empty	empty multilinestring with z coordinates and measures
multilinestring	multilinestring (())	multilinestring
multilinestring	multilinestring z (())	multilinestring with z coordinates
multilinestring	multilinestring m (())	multilinestring with measures

Table 65. Geometry types and their text representations (continued)

Geometry type	Text Description	Comment
multilinestring	multilinestring zm (())	multilinestring with z coordinates and measures
multipolygon	multipolygon empty	empty multipolygon
multipolygon	multipolygon z empty	empty multipolygon with z coordinates
multipolygon	multipolygon m empty	empty multipolygon with measures
multipolygon	multipolygon z	empty multipolygon with z coordinates and measures
multipolygon	multipolygon ((()))	multipolygon
multipolygon	multipolygon z ((()))	multipolygon with z coordinates
multipolygon	multipolygon m (((10 10 2, 10 20 3, 20 20 4, 20 15 5, 10 10 2), (50 40 7, 50 50 3, 60 50 4, 60 40 5, 50 40 7)))	multipolygon with measures
multipolygon	multipolygon zm ((()))	multipolygon with z coordinates and measures

The OGC well-known binary (WKB) representations

DB2 Spatial Extender has several functions that generate geometries from binary representations:

ST_GeomFromWKB

Creates a geometry from a WKB representation of any geometry type.

ST_PointFromWKB

Creates a point from a point WKB representation.

ST_LineFromWKB

Creates a linestring from a linestring WKB representation.

ST_PolyFromWKB

Creates a polygon from a polygon WKB representation.

ST_MPointFromWKB

Creates a multipoint from a multipoint WKB representation.

ST_MLineFromWKB

Creates a multilinestring from a multilinestring WKB representation.

ST_MPolyFromWKB

Creates a multipolygon from a multipolygon WKB representation.

The well-known binary representation is a contiguous stream of bytes. It permits geometry to be exchanged between an ODBC client and an SQL database in binary form. Because these geometry functions require the definition of C programming language structures to map the binary representation, they are intended for use within a third generation language (3GL) program. They are not suited for a fourth generation language (4GL) environment. The `ST_AsBinary` function converts an existing geometry value into a well-known binary representation.

The well-known binary representation for geometry is obtained by serializing a geometry instance as a sequence of numeric types. These types are drawn from the set (unsigned integer, double), and then each numeric type is serialized as a sequence of bytes. The types are serialized using one of two well-defined, standard, binary representations for numeric types (NDR, XDR). A one-byte tag that precedes the serialized bytes describes the specific binary encoding (NDR or XDR) used for a geometry byte stream. The only difference between the two encoding of geometry is one of byte order: The XDR encoding is Big Endian; the NDR encoding is Little Endian.

Numeric type definitions

An *unsigned integer* is a 32 bit (4 byte) data type that encodes a non-negative integer in the range [0, 4294967295].

A *double* is a 64 bit (8 byte) double precision data type that encodes a double precision number using the IEEE 754 double precision format.

These definitions are common to both XDR and NDR.

XDR (Big Endian) encoding of numeric types

The XDR representation of an unsigned integer is Big Endian (most significant byte first).

The XDR representation of a double is Big Endian (sign bit is first byte).

NDR (Little Endian) encoding of numeric types

The NDR representation of an unsigned integer is Little Endian (least significant byte first).

The NDR representation of a double is Little Endian (sign bit is last byte).

Conversion between NDR and XDR

Conversion between the NDR and XDR data types for unsigned integers and doubles is a simple operation. It involving reversing the order of bytes within each unsigned integer or double in the byte stream.

Description of WKBGeometry byte streams

This section describes the well-known binary representation for geometry. The basic building block is the byte stream for a point, which consists of two doubles. The byte streams for other geometries are built using the byte streams for geometries that are already defined.

```
// Basic Type definitions
// byte : 1 byte
// uint32 : 32 bit unsigned integer (4 bytes)
// double : double precision number (8 bytes)

// Building Blocks : Point, LinearRing

Point {
    double x;
    double y;
};
LinearRing {
    uint32 numPoints;
    Point points[numPoints];
};
enum wkbGeometryType {
    wkbPoint = 1,
    wkbLineString = 2,
    wkbPolygon = 3,
    wkbMultiPoint = 4,
    wkbMultiLineString = 5,
    wkbMultiPolygon = 6,
};
enum wkbByteOrder {
    wkbXDR = 0, // Big Endian
    wkbNDR = 1 // Little Endian
};
WKBPoint {
    byte byteOrder;
    uint32 wkbType; // 1
    Point point;
};
WKBLineString {
    byte byteOrder;
    uint32 wkbType; // 2
    uint32 numPoints;
    Point points[numPoints];
}

WKBPolygon {
    byte byteOrder;
    uint32 wkbType; // 3
    uint32 numRings;
    LinearRing rings[numRings];
}
WKBMultiPoint {
    byte byteOrder;
    uint32 wkbType; // 4
    uint32 num_wkbPoints;
```

```

    WKBPoint          WKBPoints[num_wkbPoints];
}
WKBMultiLineString {
    byte              byteOrder;
    uint32            wkbType;                // 5
    uint32            num_wkbLineStrings;
    WKBLineString    WKBLineStrings[num_wkbLineStrings];
}

wkbMultiPolygon {
    byte              byteOrder;
    uint32            wkbType;                // 6
    uint32            num_wkbPolygons;
    WKBPolygon        wkbPolygons[num_wkbPolygons];
}

WKBGeometry {
    union {
        WKBPoint          point;
        WKBLineString    linestring;
        WKBPolygon        polygon;
        WKBMultiPoint    mpoint;
        WKBMultiLineString mlinestring;
        WKBMultiPolygon  mpolygon;
    }
};

```

The following figure shows an NDR representation.

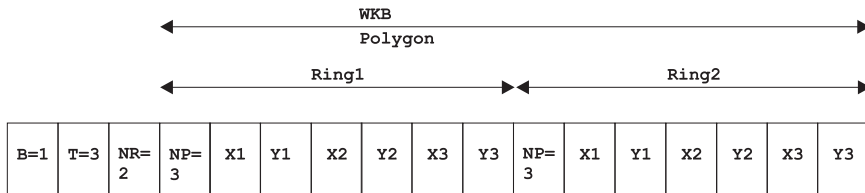


Figure 39. Representation in NDR format. (B=1) of type polygon (T=3) with 2 linears (NR=2), each ring having 3 points (NP=3).

Assertions for the WKB representation

The well-known binary representation for geometry is designed to represent instances of the geometry types described in the Geometry Object Model and in the OpenGIS Abstract Specification.

These assertions imply the following for rings, polygons, and multipolygons:

Linear rings

Rings are simple and closed, which means that linear rings cannot self intersect.

Polygons

No two linear rings in the boundary of a polygon can cross each other. The linear rings in the boundary of a polygon can intersect at most at a single point, but only as a tangent.

Multipolygons

The interiors of two polygons that are elements of a multipolygon cannot intersect. The boundaries of any two polygons that are elements of a multipolygon can touch at only a finite number of points.

The ESRI shape representations

DB2 Spatial Extender has several functions that generate geometries from ESRI shape representations. In addition to the two-dimensional representation supported by the open GIS well-known binary representation, the ESRI shape representation also supports optional Z coordinates and measures. The following functions generate geometry from an ESRI shape:

ST_GeometryFromShape

Creates a geometry from a shape representation of any geometry type.

ST_PointFromShape

Creates a point from a point shape representation.

ST_LineFromShape

Creates a linestring from a linestring shape representation.

ST_PolyFromShape

Creates a polygon from a polygon shape representation.

ST_MPointFromShape

Creates a multipoint from a multipoint shape representation.

ST_MLineFromShape

Creates a multilinestring from a multilinestring shape representation.

ST_MPolyFromShape

Creates a multipolygon from a multipolygon shape representation.

The general syntax of these functions is the same. The first argument is the shape representation entered as a binary large object (BLOB) data type. The second argument is the spatial reference identifier integer to assign to the geometry. The GeometryFromShape function has the following syntax:

```
db2gse.GeometryFromShape(ShapeGeometry Blob(1M), cr db2gse.coordref)
```

Because these shape functions require the definition of C programming language structures to map the binary representation, they are intended for

use within a 3GL program and are not suited to a 4GL environment. The AsShape function converts the geometry value into an ESRI shape representation.

A shape type of 0 indicates a null shape, with no geometric data for the shape.

Value	Shape Type
0	Null Shape
1	Point
3*	PolyLine
5	Polygon
8	MultiPoint
11	PointZ
13	PolyLineZ
15	PolygonZ
18	MultiPointZ
21	PointM
23	PolyLineM
25	PolygonM
28	MultiPointM

Note: * Shape types that are not specified above (2, 4, 6, and so forth) are reserved for future use.

Shape types in XY space

Point

A point consists of a pair of double precision coordinates in the order X, Y.

Table 66. Point byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	1	Integer	1	Little
Byte 4	X	X	Double	1	Little
Byte 12	Y	Y	Double	1	Little

MultiPoint

A MultiPoint consists of a collection of points. The bounding box is stored in the order Xmin, Ymin, Xmax, Ymax.

Table 67. MultiPoint byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	8	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumPoints	NumPoints	Integer	1	Little
Byte 40	Points	Points	Point	NumPoints	Little

PolyLine

A PolyLine is an ordered set of vertices that consists of one or more parts. A part is a connected sequence of two or more points. Points might or might not be connected to each other. Parts might or might not intersect each other.

Because this specification does not forbid consecutive points with identical coordinates, shapefile readers must handle such cases. On the other hand, the degenerate, zero length parts that might result are not allowed.

The fields for a PolyLine are:

Box The bounding box for the PolyLine stored in the order Xmin, Ymin, Xmax, Ymax.

NumParts

The number of parts in the PolyLine.

NumPoints

The total number of points for all parts.

Parts An array of length NumParts. Each PolyLine stores the index of its first point in the points array. Array indexes are with respect to 0.

Points An array of length NumPoints. The points for each part in the PolyLine are stored end to end. The points for part 2 follow the points for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the points array between parts.

Table 68. PolyLine byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	3	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little

Note: $X = 44 + 4 * \text{NumParts}$.

Polygon

A polygon consists of one or more rings. A ring is a connected sequence of four or more points that forms a closed, non-self-intersecting loop. A polygon can contain multiple outer rings. The order of vertices or orientation for a ring indicates which side of the ring is the interior of the polygon. The neighborhood to the right of an observer walking along the ring in vertex order is the neighborhood inside the polygon. Vertices of rings that define holes in polygons are in a counter-clockwise direction. Vertices for a single, ringed polygon are, therefore, always in clockwise order. The rings of a polygon are called parts.

Because this specification does not forbid consecutive points with identical coordinates, shapefile readers must handle such cases. On the other hand, the degenerate, zero length, or zero area parts that might result are not allowed.

The fields for a polygon are:

Box The bounding box for the polygon stored in the order Xmin, Ymin, Xmax, Ymax.

NumParts

The number of rings in the polygon.

NumPoints

The total number of points for all rings.

Parts An array of length NumParts. Stores, for each ring, the index of its first point in the points array. Array indexes are with respect to 0.

Points An array of length NumPoints. The points for each ring in the polygon are stored end to end. The points for Ring 2 follow the points for Ring 1, and so on. The parts array holds the array index of the starting point for each ring. There is no delimiter in the points array between rings.

Important notes about Polygon shapes:

- The rings are closed (the first and last vertex of a ring **MUST** be the same).
- The order of rings in the points array is not significant.
- Polygons stored in a shapefile must be clean. A clean polygon is one that:
 - Has no self-intersections. This means that a segment belonging to one ring can not intersect a segment belonging to another ring.

The rings of a polygon can touch each other at vertices but not along segments. Colinear segments are considered intersecting.

- Has the inside of the polygon on the "correct" side of the line that defines it. The neighborhood to the right of an observer walking along the ring in vertex order is the inside of the polygon. Vertices for a single, ringed polygon is, therefore, always in clockwise order. Rings defining holes in these polygons have a counterclockwise orientation.

"Dirty" polygons occur when the rings that define holes in the polygon also go clockwise, which causes overlapping interiors.

An Example Polygon Instance:

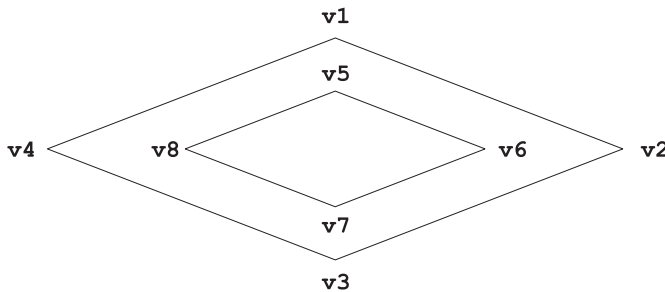


Figure 40. A polygon with a hole and eight vertices

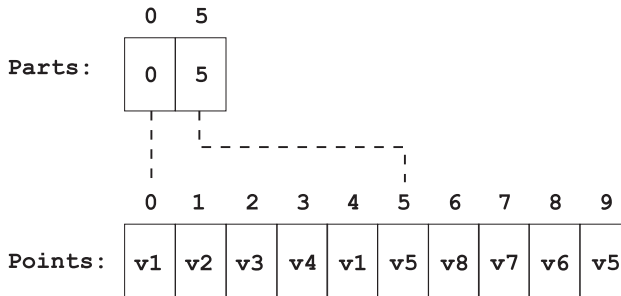


Figure 41. Contents of the polygon byte stream. NumParts equals 2 and NumPoints equals 10. Note that the order of the points for the donut (hole) polygon are reversed.

Table 69. Polygon byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	5	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little

Table 69. Polygon byte stream contents (continued)

Position	Field	Value	Type	Number	Order
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little

Note: $X = 44 + 4 * \text{NumParts}$.

Measured shape types in XY space

PointM

A PointM consists of a pair of double precision coordinates in the order X, Y, plus a measure M.

Table 70. PointM byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	21	Integer	1	Little
Byte 4	X	X	Double	1	Little
Byte 12	Y	Y	Double	1	Little
Byte 20	M	M	Double	1	Little

MultiPointM

The fields for a MultiPointM are:

Box The bounding box for the MultiPointM stored in the order Xmin, Ymin, Xmax, Ymax.

NumPoints

The number of Points.

Points An array of Points of length NumPoints.

NumMs

The number of Measures that follow. NumMs can only have two values zero if no Measures follow this field; or equal to NumPoints if Measures are present.

M Range

The minimum and maximum measures for the MultiPointM stored in the order Mmin, Mmax.

M Array

An array of Measures of length NumPoints.

Table 71. MultiPointM byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	28	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumPoints	NumPoints	Integer	1	Little
Byte 40	Points	Points	Point	NumPoints	Little
Byte X	NumMs	NumMs	Integer	1	Little
Byte X+4*	Mmin	Mmin	Double	1	Little
Byte X+12*	Mmax	Mmax	Double	1	Little
Byte X+20*	Marray	Marray	Double	NumPoints	Little

Notes:

1. $X = 40 + (16 * \text{NumPoints})$
2. * optional

PolyLineM

A shapefile PolyLineM consists of one or more parts. A part is a connected sequence of two or more points. Parts might or might not be connected to each other. Parts might or might not intersect one another.

The fields for a PolyLineM are:

Box The bounding box for the PolyLineM stored in the order Xmin, Ymin, Xmax, Ymax.

NumParts

The number of parts in the PolyLineM.

NumPoints

The total number of points for all parts.

Parts An array of length NumParts. Stores, for each part, the index of its first point in the points array. Array indexes are with respect to 0.

Points An array of length NumPoints. The points for each part in the PolyLineM are stored end to end. The points for part 2 follow the points for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the points array between parts.

NumMs

The number of Measures that follow. NumMs can only have two values zero if no Measures follow this field; or equal to NumPoints if Measures are present.

M Range

The minimum and maximum measures for the PolyLineM stored in the order Mmin, Mmax.

M Array

An array of length NumPoints. The measures for each part in the PolyLineM are stored end to end. The measures for part 2 follow the measures for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the measure array between parts.

Table 72. PolyLineM byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	13	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little
Byte Y	NumMs	NumMs	Integer	1	Little
Byte Y+4*	Mmin	Mmin	Double	1	Little
Byte Y+12*	Mmax	Mmax	Double	1	Little
Byte Y+20*	Marray	Marray	Double	NumPoints	Little

Notes:

1. $X = 44 + (4 * \text{NumParts})$, $Y = X + (16 * \text{NumPoints})$.
2. * optional

PolygonM

A PolygonM consists of a number of rings. A ring is a closed, non-self-intersecting loop. Note that intersections are calculated in XY space, *not* in XYM space. A PolygonM can contain multiple outer rings. The rings of a PolygonM are called parts.

The fields for a PolygonM are:

Box The bounding box for the PolygonM stored in the order Xmin, Ymin, Xmax, Ymax.

NumParts

The number of rings in the PolygonM.

NumPoints

The total number of points for all rings.

Parts An array of length NumParts. Stores, for each ring, the index of its first point in the points array. Array indexes are with respect to 0.

Points An array of length NumPoints. The points for each ring in the PolygonM are stored end to end. The points for Ring 2 follow the points for Ring 1, and so on. The parts array holds the array index of the starting point for each ring. There is no delimiter in the points array between rings.

NumMs
The number of Measures that follow. NumMs can have only two zero values if no Measures follow this field, or equal to NumPoints if Measures are present.

M Range
The minimum and maximum measures for the PolygonM stored in the order Mmin, Mmax.

M Array
An array of length NumPoints. The measures for each ring in the PolygonM are stored end to end. The measures for Ring 2 follow the measures for Ring 1, and so on. The parts array holds the array index of the starting measure for each ring. There is no delimiter in the measure array between rings.

Important notes about PolygonM shapes:

- The rings are closed (the first and last vertex of a ring must be the same).
- The order of rings in the points array is not significant.

Table 73. PolygonM byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	15	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little
Byte Y	NumMs	NumMs	Integer	1	Little
Byte Y+4*	Mmin	Mmin	Double	1	Little
Byte Y+12*	Mmax	Mmax	Double	1	Little
Byte Y+20*	Marray	Marray	Double	NumPoints	Little

Notes:

1. $X = 44 + (4 * \text{NumParts})$, $Y = X + (16 * \text{NumPoints})$.
2. * optional

Shape types in XYZ space**PointZ**

A PointZ consists of triplet, double precision coordinates in the order X, Y, Z plus a measure.

Table 74. PointZ byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	11	Integer	1	Little
Byte 4	X	X	Double	1	Little
Byte 12	Y	Y	Double	1	Little
Byte 20	Z	Z	Double	1	Little
Byte 28	Measure	M	Double	1	Little

MultiPointZ

A MultiPointZ represents a set of PointZs, as follows:

- The bounding box is stored in the order Xmin, Ymin, Xmax, Ymax.
- The bounding Z range is stored in the order Zmin, Zmax. Bounding M Range is stored in the order Mmin, Mmax.

Table 75. MultiPointZ byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	18	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumPoints	NumPoints	Integer	1	Little
Byte 40	Points	Points	Point	NumPoints	Little
Byte X	Zmin	Zmin	Double	1	Little
Byte X+8	Zmax	Zmax	Double	1	Little
Byte X+16	Zarray	Zarray	Double	NumPoints	Little
Byte Y	NumMs	NumMs	Integer	1	Little
Byte Y+4*	Mmin	Mmin	Double	1	Little
Byte Y+12*	Mmax	Mmax	Double	1	Little
Byte Y+20*	Marray	Marray	Double	NumPoints	Little

Notes:

1. $X = 40 + (16 * \text{NumPoints}); Y = X + 16 + (8 * \text{NumPoints})$
2. * optional

PolyLineZ

A PolyLineZ consists of one or more parts. A part is a connected sequence of two or more points. Parts might or might not be connected to each other. Parts might or might not intersect one another.

The fields for a PolyLineZ are:

Box The bounding box for the PolyLineZ stored in the order Xmin, Ymin, Xmax, Ymax.

NumParts

The number of parts in the PolyLineZ.

NumPoints

The total number of points for all parts.

Parts An array of length NumParts. Stores, for each part, the index of its first point in the points array. Array indexes are with respect to 0.

Points An array of length NumPoints. The points for each part in the PolyLineZ are stored end to end. The points for part 2 follow the points for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the points array between parts.

Z Range

The minimum and maximum Z values for the PolyLineZ stored in the order Zmin, Zmax.

Z Array

An array of length NumPoints. The Z values for each part in the PolyLineZ are stored end to end. The Z values for part 2 follow the Z values for part 1, and so on. The parts array holds the array index of the starting point for each part. There is no delimiter in the Z array between parts.

NumMs

The number of Measures that follow. NumMs can only have two values zero if no Measures follow this field; or equal to NumPoints if Measures are present.

M Range

The minimum and maximum measures for the PolyLineZ stored in the order Mmin, Mmax.

M Array

An array of length NumPoints. The measures for each part in the

PolyLineZ are stored end to end. The measures for part 2 follow the measures for part 1, and so on. The parts array holds the array index of the starting measure for each part. There is no delimiter in the measure array between parts.

Table 76. PolyLineZ byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	13	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little
Byte Y	Zmin	Zmin	Double	1	Little
Byte Y+8	Zmax	Zmax	Double	1	Little
Byte Y+16	Zarray	Zarray	Double	NumPoints	Little
Byte Z	NumMs	NumMs	Integer	1	Little
Byte Z+4*	Mmin	Mmin	Double	1	Little
Byte Z+12*	Mmax	Mmax	Double	1	Little
Byte Z+20*	Marray	Marray	Double	NumPoints	Little

Notes:

1. $X = 44 + (4 * \text{NumParts})$, $Y = X + (16 * \text{NumPoints})$, $Z = Y + 16 + (8 * \text{NumPoints})$
2. * optional

PolygonZ

A PolygonZ consists of a number of rings. A ring is a closed, non-self-intersecting loop. A PolygonZ can contain multiple outer rings. The rings of a PolygonZ are called parts.

The fields for a PolygonZ are:

Box The bounding box for the PolygonZ stored in the order Xmin, Ymin, Xmax, Ymax.

NumParts

The number of rings in the PolygonZ.

NumPoints

The total number of points for all rings.

Parts An array of length NumParts. Stores, for each ring, the index of its first point in the points array. Array indexes are with respect to 0.

Points An array of length NumPoints. The points for each ring in the PolygonZ are stored end to end. The points for Ring 2 follow the points for Ring 1, and so on. The parts array holds the array index of the starting point for each ring. There is no delimiter in the points array between rings.

Z Range
The minimum and maximum Z values for the arc stored in the order Zmin, Zmax.

Z Array
An array of length NumPoints. The Z values for each ring in the PolygonZ are stored end to end. The Z values for Ring 2 follow the Z values for Ring 1, and so on. The parts array holds the array index of the starting Z value for each ring. There is no delimiter in the Z value array between rings.

NumMs
The number of Measures that follow. NumMs can only have two values zero if no Measures follow this field; or equal to NumPoints if Measures are present.

M Range
The minimum and maximum measures for the PolygonZ stored in the order Mmin, Mmax.

M Array
An array of length NumPoints. The measures for each ring in the PolygonZ are stored end to end. The measures for Ring 2 follow the measures for Ring 1, and so on. The parts array holds the array index of the starting measure for each ring. There is no delimiter in the measure array between rings.

Important notes about PolygonZ shapes:

- The rings are closed (the first and last vertex of a ring MUST be the same).
- The order of rings in the points array is not significant.

Table 77. PolygonZ byte stream contents

Position	Field	Value	Type	Number	Order
Byte 0	Shape Type	15	Integer	1	Little
Byte 4	Box	Box	Double	4	Little
Byte 36	NumParts	NumParts	Integer	1	Little
Byte 40	NumPoints	NumPoints	Integer	1	Little

Table 77. PolygonZ byte stream contents (continued)

Position	Field	Value	Type	Number	Order
Byte 44	Parts	Parts	Integer	NumParts	Little
Byte X	Points	Points	Point	NumPoints	Little
Byte Y	Zmin	Zmin	Double	1	Little
Byte Y+8	Zmax	Zmax	Double	1	Little
Byte Y+16	Zarray	Zarray	Double	NumPoints	Little
Byte Z	NumMs	NumMs	Integer	1	Little
Byte Z+4*	Mmin	Mmin	Double	1	Little
Byte Z+12*	Mmax	Mmax	Double	1	Little
Byte Z+20*	Marray	Marray	Double	NumPoints	Little

Part 3. Appendixes

Notices

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make

improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licenses of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Canada Limited
Office of the Lab Director
1150 Eglinton Ave. East
North York, Ontario
M3C 1H7
CANADA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

Trademarks

The following terms, which may be denoted by an asterisk(*), are trademarks of International Business Machines Corporation in the United States, other countries, or both.

ACF/VTAM	IBM
AISPO	IMS
AIX	IMS/ESA
AIX/6000	LAN DistanceMVS
AIXwindows	MVS/ESA
AnyNet	MVS/XA
APPN	Net.Data
AS/400	OS/2
BookManager	OS/390
CICS	OS/400
C Set++	PowerPC
C/370	QBIC
DATABASE 2	QMF
DataHub	RACF
DataJoiner	RISC System/6000
DataPropagator	RS/6000
DataRefresher	S/370
DB2	SP
DB2 Connect	SQL/DS
DB2 Extenders	SQL/400
DB2 OLAP Server	System/370
DB2 Universal Database	System/390
Distributed Relational Database Architecture	SystemView
DRDA	VisualAge
eNetwork	VM/ESA
Extended Services	VSE/ESA
FFST	VTAM
First Failure Support Technology	WebExplorer
	WIN-OS/2

The following terms are trademarks or registered trademarks of other companies:

Microsoft, Windows, and Windows NT are trademarks or registered trademarks of Microsoft Corporation.

Java or all Java-based trademarks and logos, and Solaris are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Tivoli and NetView are trademarks of Tivoli Systems Inc. in the United States, other countries, or both.

UNIX is a registered trademark in the United States, other countries or both and is licensed exclusively through X/Open Company Limited.

Other company, product, or service names, which may be denoted by a double asterisk(**) may be trademarks or service marks of others.

Index

A

AIX

- installing DB2 Spatial Extender 17
- where macro definitions for constants are stored 63
- where reference data is stored 21

angular units 277

applications

- guidelines for writing 53
- stored procedures 63

ArcExplorer

- downloading 19
- using as interface 9, 49

AsBinaryShape 152, 156

attribute data 4

automatic geocoding 38

azimuthal projections 283

B

B tree indexes 110

batch geocoding 38

boundary 120, 123

C

catalog views

- DB2GSE.COORD_REF_SYS 105
- DB2GSE.GEOMETRY_COLUMNS 106
- DB2GSE.SPATIAL_GEOCODER 106
- DB2GSE.SPATIAL_REF_SYS 107

class 122

conic projections 282

coordinate systems 275

- DB2GSE.COORD_REF_SYS catalog view 105

- deriving spatial reference systems from 24

- description 5, 23

coordinates

description 5

X coordinates

- description 23
- properties of geometries 122

Y coordinates

- description 23
- properties of geometries 122

coordinates (*continued*)

Z coordinates

- description 23
- properties of geometries 122

Create Spatial Index window 47

Create Spatial Layer window

- for registering a table column as a layer 34
- for registering a view column as a layer 36

Create Spatial Reference

window 27, 28

D

data items 6

databases

- disabling support for spatial operations
 - db2gse.gse_disable_db 68
 - sample program 54
- enabling for spatial operations
 - DB2 Control Center menu choices 22
 - db2gse.gse_enable_db 73
 - discussion 21
 - sample program 54

DB2 Control Center

- Create Spatial Index window 47
- Create Spatial Layer window
 - for registering a table column as a layer 34
 - for registering a view column as a layer 36

Create Spatial Reference window 27, 28

- Export Spatial Data window 45
- Import Spatial Data window 42, 43, 44

invoking DB2 Spatial Extender from 20

Run Geocoder window 39, 40

DB2 instance update utility

(db2iupdt) 19

DB2 Spatial Extender

- applications
 - guidelines for writing 53
 - stored procedures 63
- catalog views 105
- configuration 15

DB2 Spatial Extender (*continued*)

error, warning, and informational messages 97

installing

- hardware and software requirements 15
- on AIX 17
- on Windows NT 17
- verification 18

interfaces to 8

invoking from DB2 Control Center 20

purpose 3

resources

- for spatial operations 21
- reference data 21
- summary 21
- sample program
 - compiling and running 18
 - description 53
- spatial functions 155
- stored procedures 63
- tasks, summaries of
 - carried out by stored procedures 64
 - overview 9
- sample program 53
- scenario 11

DB2GSE.COORD_REF_SYS 105

DB2GSE.GEOMETRY_COLUMNS 106

db2gse.gse_disable_autogc 66

db2gse.gse_disable_db 68

db2gse.gse_disable_sref 69

db2gse.gse_enable_autogc 70

db2gse.gse_enable_db 73

db2gse.gse_enable_idx 74

db2gse.gse_enable_sref 76

db2gse.gse_export_shape 78

db2gse.gse_import_sde 80

db2gse.gse_import_shape 82

db2gse.gse_register_gc 84

db2gse.gse_register_layer 86

db2gse.gse_run_gc 92

db2gse.gse_unregist_gc 94

db2gse.gse_unregist_layer 95

DB2GSE.SPATIAL_GEOCODER 106

DB2GSE.SPATIAL_REF_SYS 107

db2iupdt (DB2 instance update utility) 19

default geocoder 37
dimension 124
disk space requirements 16

E

EBNF (Extended Backus Naur) 275
empty or not empty 123
enabling databases for spatial operations
 DB2 Control Center menu choices 22
 description 9
 discussion 21
envelope 111, 123
EnvelopesIntersect 137, 158
error messages 97
ESRI shape representations
 associated spatial functions 151
 discussion 294
Export Spatial Data window 45
exterior 120, 123

F

false M
 specifying 25, 29
false X
 specifying 25, 28
false Y
 specifying 25, 28
false Z
 specifying 25, 28

G

geocentric coordinate system 277
geocoders
 DB2GSE.SPATIAL_GEOCODER catalog view 106
 default geocoder 37
 disabling automatic geocoding
 db2gse.gse_disable_autogc 66
 Run Geocoder window 40
 sample program 57
 enabling automatic geocoding
 Create Spatial Layer window 34
 db2gse.gse_enable_autogc 70
 discussion 31, 38
 sample program 57
 non-default geocoders
 discussion 37
 using db2gse.gse_register_gc to register 84
 using db2gse.gse_unregister_gc to unregister 94
 running in batch mode
 db2gse.gse_run_gc 92

geocoders (*continued*)
 running in batch mode (*continued*)
 discussion 38
 Run Geocoder window 39
 sample program 56, 57

geocoding
 batch 38
 description 6
 discussion 37
 incremental 38
 precision 14
geodetic datums 279
GEOGCS keyword 276
geographic features
 associated data types 32
 description 3
 represented by data 4
geographic information system (GIS)
 creating 9
 description 3
 using 11
geometries
 correspondence with spatial data types 121
 discussion 119
 linestrings 121, 126
 multilinestrings 121, 129
 multipoints 121, 129
 multipolygons 121, 130
 points 120, 125
 polygons 121, 127
 properties
 boundary 120, 123
 class 122
 dimension 124
 empty or not empty 123
 envelope 111, 123
 exterior 120, 123
 interior 120, 123
 measures 122
 simple or non-simple 123
 spatial reference system
 identifier (SRID) 124
 X coordinates 122
 Y coordinates 122
 Z coordinates 122
 spatial index grids 111
GeometryFromShape 152, 157
grid indexes 47

I

Import Spatial Data window 42, 43, 44
incremental geocoding 38
informational messages 97

installing DB2 Spatial Extender
 hardware and software requirements 15
 on AIX 17
 on Windows NT 17
 verification 18
interfaces to DB2 Spatial Extender 8
interior 120, 123
Is3d 122, 160
IsMeasured 122, 161

J

Java 2 Runtime Environment (JRE) v1.2.2 19

L

layers
 DB2GSE.GEOMETRY_COLUMNS catalog view 106
 description 11
 registering table columns as
 Create Spatial Layer window 34
 db2gse.gse_register_layer 86
 sample program 56
 registering view columns as
 Create Spatial Layer window 36
 db2gse.gse_register_layer 86
 sample program 58
 using db2gse.gse_unregister_layer to unregister 95
linear rings 293
linear units 277
LineFromShape 152, 162
linestrings 121, 126
LocateAlong 148, 164
LocateBetween 148, 166

M

M 126, 168
M units
 specifying 26, 29
map projection parameters 283
map projections 282
Measured shape types in XY spaces 299
measures
 description 23, 122
 properties of geometries 122
messages 97
MLine FromShape 169
MLineFromShape 152
MPPointFromShape 152, 171
MPPolyFromShape 152, 172

- multilinestrings 121, 129
- MultiPoint byte stream contents 295
- MultiPointM byte stream contents 299, 300
- multipoints 121, 129
- MultiPointZ byte stream contents 303
- multipolygons 121, 130

N

- NDR encoding 291

O

- offset factors
 - specifying 25, 28

P

- pattern matrices 133
- planar projections 283
- Point byte stream contentse 295
- PointFromShape 152, 173
- PointM byte stream contentse 299
- points 120, 125
- PointZ byte stream contents 303
- PolyFromShape 152, 174
- Polygon byte stream contents 298
- PolygonM byte stream contents 302
- polygons 121, 127
- PolygonZ byte stream contents 306
- PolyLine byte stream contents 296
- PolyLineM byte stream contents 301
- PolyLineZ byte stream contents 305
- POSC/EPSB coordinate system model 275
- precision
 - geocoding 14, 39
 - preserving for spatial reference systems 24
- prime meridians 281
- PROJCS keyword 276
- projections
 - azimuthal 283
 - conic 282
 - map
 - parameters 283
 - types 282
 - planar 283

Q

- queries
 - exploiting spatial indexes 51
 - interfaces for submitting 9, 49
 - sample program 58

- queries (*continued*)
 - types of spatial functions to use 49
 - using spatial predicate functions 50

R

- reference data 21
- Run Geocoder window 39, 40

S

- sample program
 - compiling and running 18
 - description 53
- scale factors
 - specifying 25, 28
- scenario of tasks 11
- shapes
 - in XY space 295
 - in XYZ space 303
- ShapeToSQL 152, 176
- simple or non-simple 123
- software requirements 16
- source data 5
- spatial columns 37
- spatial data
 - derived from attribute data 6
 - derived from other spatial data
 - discussion 7
 - spatial functions that derive the data 144
- exporting
 - db2gse.gse_export_shape 78
 - discussion 41
 - Export Spatial Data window 45
 - sample program 59
- file formats
 - ESRI shape
 - representations 151, 294
 - WKB (well-known binary)
 - representations 150, 290
 - WKT (well-known text)
 - representations 149, 285
- importing
 - db2gse.gse_import_sde 80
 - db2gse.gse_import_shape 82
 - discussion 8, 41
 - Import Spatial Data window 41, 43
 - sample program 56
 - nature of 5
- spatial data types 31
 - correspondence with geometries 121
 - description 31

- spatial functions

- .ST_Area 178
- AsBinaryShape 152, 156
- categorized by operations
 - performed 49
- EnvelopesIntersect 137, 158
- GeometryFromShape 152
- Is3d 122, 160
- IsMeasured 122, 161
- LineFromShape 152, 162
- LocateAlong 148, 164
- LocateBetween 148, 166
- M 126, 168
- MLine FromShape 169
- MLineFromShape 152
- MPointFromShape 152, 171
- MPolyFromShape 152, 172
- PointFromShape 152, 173
- PolyFromShape 152, 174
- predicates 50
- ShapeToSQL 152, 176
- ST_Area 128, 131
- ST_AsBinary 151, 180
- ST_AsText 150, 181
- ST_Boundary 123, 182
- ST_Buffer 146, 184
- ST_Centroid 128, 131, 186
- ST_Contains 142, 187
- ST_Convexhull 189
- ST_ConvexHull 149
- ST_CoordDim 125, 191
- ST_Crosses 139, 193
- ST_Difference 145, 195
- ST_Dimension 124, 196
- ST_Disjoint 135, 198
- ST_Distance 143, 200
- ST_Endpoint 126, 201
- ST_Envelope 124, 202
- ST_Equals 134, 204
- ST_ExteriorRing 128, 205
- ST_GeometryFromText 207
- ST_GeometryN 129, 211
- ST_GeometryType 122, 212
- ST_GeomFromText 150, 285
- ST_GeomFromWKB 151, 209, 290
- ST_InteriorRingN 128, 214
- ST_Intersection 144, 219
- ST_Intersects 136, 221
- ST_IsClosed 127, 129, 222
- ST_IsEmpty 123, 224
- ST_IsRing 127, 226
- ST_IsSimple 123, 227
- ST_IsValid 122, 228
- ST_Length 127, 129, 230

- spatial functions (*continued*)
 - ST_LineFromText 150, 232, 285
 - ST_LineFromWKB 151, 233, 290
 - ST_MLineFromText 150, 235, 285
 - ST_MLineFromWKB 151, 236, 290
 - ST_MPointFromText 150, 238, 285
 - ST_MPointFromWKB 151, 239, 290
 - ST_MPolyFromText 150, 240, 285
 - ST_MPolyFromWKB 151, 241, 290
 - ST_NumGeometries 129, 242
 - ST_NumInteriorRing 128, 243
 - ST_NumPoints 127, 244
 - ST_OrderingEquals 135, 245
 - ST_Overlaps 138, 246
 - ST_Perimeter 128, 248
 - ST_Point 125, 251
 - ST_PointFromText 126, 249, 285
 - ST_PointFromWKB 151, 250, 290
 - ST_PointN 126, 252
 - ST_PointOnSurface 128, 253
 - ST_PolyFromText 150, 254, 285
 - ST_PolyFromWKB 151, 255, 290
 - ST_Polygon 149, 257
 - ST_Relate 143, 258
 - ST_SRID 124, 260
 - ST_StartPoint 126, 261
 - ST_SymmetricDiff 262
 - ST_Touches 137, 264
 - ST_Transform 125, 265
 - ST_Union 146, 266
 - ST_Within 141, 267
 - ST_WKBToSQL 150, 268
 - ST_WKTToSQL 149, 270
 - ST_X 126, 271
 - ST_Y 126, 272
- types
 - associated with instantiable geometries 125
 - associated with properties of geometries 121
 - data exchange 149
 - functions that compare geometries 132
 - functions that generate geometries 144
 - functions that show relationships between geometries 132
- spatial functions (*continued*)
 - types (*continued*)
 - predicate functions 132
 - using to exploit spatial indexes 51
 - Z 126
- spatial indexes 109
 - creating
 - Create Spatial Index window 47
 - db2gse.gse_enable_idx 74
 - determining grid size 48, 116
 - sample program 56
 - exploiting 51
 - grid indexes 47
 - how they are generated 111
 - using 115
- spatial information
 - description 3
 - retrieving and analyzing
 - exploiting spatial indexes 51
 - interfaces to use 9, 49
 - sample program 58
 - types of spatial functions to use 49
 - using spatial predicate functions 50
- spatial reference system identifier (SRID) 124
- spatial reference systems
 - creating
 - Create Spatial Reference window 26
 - db2gse.gse_enable_sref 76
 - discussion 23
 - sample program 54
 - DB2GSE.SPATIAL_REF_SYS
 - catalog view 107
 - description 10
 - dropping
 - db2gse.gse_disable_sref 69
 - sample program 54
 - specifying parameters
 - false M 25, 29
 - false X 25, 28
 - false Y 25, 28
 - false Z 25, 28
 - M units 26, 29
 - offset factors 25, 28
 - scale factors 25, 28
 - XY units 25, 28
 - Z units 26, 28
- spheroids 278
- SRID (spatial reference system identifier) 287
- ST_Area 128, 131, 178
- ST_AsBinary 151, 180
- ST_AsText 150, 181
- ST_Boundary 123, 182
- ST_Buffer 146, 184
- ST_Centroid 128, 131, 186
- ST_Contains 142, 187
- ST_Convexhull 189
- ST_ConvexHull 149
- ST_CoordDim 125, 191
- ST_Crosses 139, 193
- ST_Difference 145, 195
- ST_Dimension 124, 196
- ST_Disjoint 135, 198
- ST_Distance 143, 200
- ST_Endpoint 126, 201
- ST_Envelope 124, 202
- ST_Equals 134, 204
- ST_ExteriorRing 128, 205
- ST_GeometryFromText 207
- ST_GeometryN 129, 211
- ST_GeometryType 122, 212
- ST_GeomFromText 150, 285
- ST_GeomFromWKB 151, 209, 290
- ST_InteriorRingN 128, 214
- ST_Intersection 144, 219
- ST_Intersects 136, 221
- ST_IsClosed 127, 129, 222
- ST_IsEmpty 123, 224
- ST_IsRing 127, 226
- ST_IsSimple 123, 227
- ST_IsValid 122, 228
- ST_Length 127, 129, 230
- ST_LineFromText 150, 232, 285
- ST_LineFromWKB 151, 233, 290
- ST_MLineFromText 150, 235, 285
- ST_MLineFromWKB 151, 236, 290
- ST_MPointFromText 150, 238, 285
- ST_MPointFromWKB 151, 239, 290
- ST_MPolyFromText 150, 240, 285
- ST_MPolyFromWKB 151, 241, 290
- ST_NumGeometries 129, 242
- ST_NumInteriorRing 128, 243
- ST_NumPoints 127, 244
- ST_OrderingEquals 135, 245
- ST_Overlaps 138, 246
- ST_Perimeter 128, 248
- ST_Point 125, 251
- ST_PointFromText 126, 249, 285
- ST_PointFromWKB 151, 250, 290
- ST_PointN 126, 252
- ST_PointOnSurface 128, 253
- ST_PolyFromText 150, 254, 285
- ST_PolyFromWKB 151, 255, 290
- ST_Polygon 149, 257

- ST_Relate 143, 258
- ST_SRID 124, 260
- ST_StartPoint 126, 261
- ST_SymmetricDiff 262
- ST_Touches 137, 264
- ST_Transform 125, 265
- ST_Union 146, 266
- ST_Within 141, 267
- ST_WKBTtoSQL 150, 268
- ST_WKTTtoSQL 149, 270
- ST_X 126, 271
- ST_Y 126, 272
- stored procedures
 - db2gse.gse_disable_autogc 66
 - db2gse.gse_disable_db 68
 - db2gse.gse_disable_sref 69
 - db2gse.gse_enable_autogc 70
 - db2gse.gse_enable_db 73
 - db2gse.gse_enable_idx 74
 - db2gse.gse_enable_sref 76
 - db2gse.gse_export_shape 78
 - db2gse.gse_import_sde 80
 - db2gse.gse_import_shape 82
 - db2gse.gse_register_gc 84
 - db2gse.gse_register_layer 86
 - db2gse.gse_run_gc 92
 - db2gse.gse_unregist_gc 94
 - db2gse.gse_unregist_layer 95

T

- tessellation 149
- triggers
 - disabling automatic geocoding
 - db2gse.gse_disable_autogc 66
 - enabling automatic geocoding
 - db2gse.gse_enable_autogc 70
 - using to invoke geocoder 31, 38

U

- UNIT keyword 276

W

- warning messages 97
- Windows NT
 - installing DB2 Spatial Extender 17
 - where macro definitions for constants are stored 63
 - where reference data is stored 21
- WKB (well-known binary)
 - representations
 - associated spatial functions 150
 - discussion 290
- WKBGeometry 292
- WKBGeometry byte streams 292

- WKT (well-known text)
 - representations
 - associated spatial functions 149
 - discussion 285

X

- X coordinates
 - description 23
 - property of geometries 122
- XDR encoding 291
- XY units
 - specifying 25, 28

Y

- Y coordinates
 - description 23
 - property of geometries 122

Z

- Z 126
- Z coordinates
 - description 23
 - property of geometries 122
- Z units
 - specifying 26, 28

Contacting IBM

If you have a technical problem, please review and carry out the actions suggested by the *Troubleshooting Guide* before contacting DB2 Customer Support. This guide suggests information that you can gather to help DB2 Customer Support to serve you better.

For information or to order any of the DB2 Universal Database products contact an IBM representative at a local branch office or contact any authorized IBM software remarketer.

If you live in the U.S.A., then you can call one of the following numbers:

- 1-800-237-5511 for customer support
- 1-888-426-4343 to learn about available service options

Product Information

If you live in the U.S.A., then you can call one of the following numbers:

- 1-800-IBM-CALL (1-800-426-2255) or 1-800-3IBM-OS2 (1-800-342-6672) to order products or get general information.
- 1-800-879-2755 to order publications.

<http://www.ibm.com/software/data/>

The DB2 World Wide Web pages provide current DB2 information about news, product descriptions, education schedules, and more.

<http://www.ibm.com/software/data/db2/library/>

The DB2 Product and Service Technical Library provides access to frequently asked questions, fixes, books, and up-to-date DB2 technical information.

Note: This information may be in English only.

<http://www.elink.ibm.com/pbl/pbl/>

The International Publications ordering Web site provides information on how to order books.

<http://www.ibm.com/education/certify/>

The Professional Certification Program from the IBM Web site provides certification test information for a variety of IBM products, including DB2.

ftp.software.ibm.com

Log on as anonymous. In the directory /ps/products/db2, you can find demos, fixes, information, and tools relating to DB2 and many other products.

comp.databases.ibm-db2, bit.listserv.db2-l

These Internet newsgroups are available for users to discuss their experiences with DB2 products.

On CompuServe: GO IBMDB2

Enter this command to access the IBM DB2 Family forums. All DB2 products are supported through these forums.

For information on how to contact IBM outside of the United States, refer to Appendix A of the *IBM Software Support Handbook*. To access this document, go to the following Web page: <http://www.ibm.com/support/>, and then select the IBM Software Support Handbook link near the bottom of the page.

Note: In some countries, IBM-authorized dealers should contact their dealer support structure instead of the IBM Support Center.



Part Number: CT7C0NA



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC27-0701-00



(1P) P/N: CT7C0NA



Spine information:



IBM® DB2® Spatial Extender

DB2 Spatial Extender User's Guide and Reference

Version 7