

IBM DB2 10.5
for Linux, UNIX, and Windows

*Spatial Extender User's Guide and
Reference*



IBM DB2 10.5
for Linux, UNIX, and Windows

*Spatial Extender User's Guide and
Reference*



Note

Before using this information and the product it supports, read the general information under Appendix B, "Notices," on page 433.

Edition Notice

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.

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at <http://www.ibm.com/shop/publications/order>
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at <http://www.ibm.com/planetwide/>

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

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 IBM Corporation 1998, 2013.

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

Contents

Chapter 1. DB2 Spatial Extender 1

How data represents geographic features	2
The nature of spatial data	3
Where spatial data comes from	4
How features, spatial information, spatial data, and geometries fit together	5

Chapter 2. Geometries 7

Properties of geometries	9
Geometry types	9
Geometry coordinates	10
X and Y coordinates	10
Z coordinates	10
M coordinates	10
Interior, boundary, and exterior	10
Simple or non-simple geometries	10
Closed geometries	10
Empty or not empty geometries	11
Minimum bounding rectangle (MBR)	11
Dimension of geometries	11
Spatial reference system identifier	11

Chapter 3. How to use DB2 Spatial Extender 13

Interfaces to DB2 Spatial Extender and associated functionality	13
Setting up DB2 Spatial Extender	13
Creating projects that use spatial data	15

Chapter 4. Getting started with DB2 Spatial Extender 19

Setting up and installing DB2 Spatial Extender	19
System requirements for installing Spatial Extender	20
Installing DB2 Spatial Extender using the DB2 Setup wizard (Windows)	21
Installing DB2 Spatial Extender using the DB2 Setup wizard (Linux and UNIX)	22
Verifying the Spatial Extender installation	24

Chapter 5. Upgrading to DB2 Spatial Extender Version 10.5 27

Upgrading DB2 Spatial Extender	27
Updating from 32-bit to 64-bit DB2 Spatial Extender	28

Chapter 6. Setting up spatial resources for a database 29

Inventory of resources supplied for your database	29
Enabling a database for spatial operations	29
Registering a geocoder	30

Chapter 7. Setting up spatial resources for a project 33

How to use coordinate systems	33
---	----

Coordinate systems	33
Geographic coordinate system	33
Projected coordinate systems	38
Determine what coordinate system to use	39
How to set up spatial reference systems	40
Spatial reference systems	40
Deciding whether to use an existing reference system or create a new system	42
Spatial reference systems supplied with DB2 Spatial Extender	42
Conversion factors that transform coordinate data into integers	44
Creating a spatial reference system	46
Calculating scale factors	47
Conversion factors that transform coordinate data into integers	48
Determining minimum and maximum coordinates and measures	48
Calculating offset values	49
Creating a spatial reference system	49

Chapter 8. Setting up spatial columns 51

Spatial columns visualization	51
Spatial data types	51
Data types for single-unit features	52
Data types for multi-unit features	52
A data type for all features	53
Creating spatial columns	53
Registering spatial columns	54

Chapter 9. Populating spatial columns 55

About importing and exporting spatial data	55
Importing shape data to a new or existing table	56
Exporting data to a shapefile	57
How to use a geocoder	57
Geocoders and geocoding	57
Setting up geocoding operations	59
Setting up a geocoder to run automatically	61
Running a geocoder in batch mode	62

Chapter 10. DB2 Spatial Extender in a partitioned database environment 63

Creating and loading spatial data in a partitioned database environment	63
Improving query performance on spatial data in a partitioned environment	64

Chapter 11. Using indexes and views to access spatial data 67

Spatial grid indexes	67
Generation of spatial grid indexes	67
Use of spatial functions in a query	68
How a query uses a spatial grid index	68

Considerations for number of index levels and grid sizes	69
Number of grid levels	69
Grid cell sizes	69
Creating spatial grid indexes	73
CREATE INDEX statement for a spatial grid index	74
Tuning spatial grid indexes with the Index Advisor	75
Determining grid sizes for a spatial grid index	76
Analyzing spatial grid index statistics	76
gseidx command	81
Using views to access spatial columns	84

Chapter 12. Analyzing and Generating spatial information 85

Environments for performing spatial analysis	85
Examples of how spatial functions operate	85
Functions that use indexes to optimize queries	86

Chapter 13. Writing applications and using the sample program 89

Including the DB2 Spatial Extender header file in spatial applications	89
Calling DB2 Spatial Extender stored procedures from an application.	89
DB2 Spatial Extender sample program	91

Chapter 14. Identifying DB2 Spatial Extender problems 97

How to interpret DB2 Spatial Extender messages	97
DB2 Spatial Extender stored procedure output parameters	99
DB2 Spatial Extender function messages	101
DB2 Spatial Extender CLP messages.	102
Tracing DB2 Spatial Extender problems with the db2trc command	104
The administration notification file	105

Chapter 15. Catalog views 107

DB2GSE.ST_COORDINATE_SYSTEMS catalog view	107
DB2GSE.ST_GEOMETRY_COLUMNS catalog view	108
DB2GSE.ST_GEOCODER_PARAMETERS catalog view	109
DB2GSE.ST_GEOCODERS catalog view	111
DB2GSE.ST_GEOCODING catalog view	111
DB2GSE.ST_GEOCODING_PARAMETERS catalog view	112
DB2GSE.ST_SIZINGS catalog view	114
DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view	114
DB2GSE.ST_UNITS_OF_MEASURE catalog view	117
DB2GSE.SPATIAL_REF_SYS catalog view	117

Chapter 16. DB2 Spatial Extender commands 119

Invoking commands for setting up DB2 Spatial Extender and developing projects.	119
db2se alter_cs command.	120
db2se alter_srs command	121

db2se create_cs command	124
db2se create_srs command	125
db2se disable_autogc command	129
db2se disable_db command	130
db2se drop_cs command	132
db2se drop_srs command	132
db2se enable_autogc command	133
db2se enable_db command	135
db2se export_shape command.	136
db2se import_shape command	139
db2se register_gc command	146
db2se register_spatial_column command	148
db2se remove_gc_setup command	150
db2se restore_indexes command	151
db2se save_indexes command	152
db2se run_gc command	152
db2se setup_gc command	154
db2se shape_info command	157
db2se unregister_gc command.	158
db2se unregister_spatial_column command	159
db2se upgrade command	160
db2se migrate command	162

Chapter 17. Stored procedures 165

ST_ALTER_COORDSYS procedure	166
ST_ALTER_SRS procedure	167
ST_CREATE_COORDSYS procedure.	171
ST_CREATE_SRS procedure	173
ST_DISABLE_AUTOGEOCODING procedure	179
ST_DISABLE_DB procedure	181
ST_DROP_COORDSYS procedure	182
ST_DROP_SRS procedure	183
ST_ENABLE_AUTOGEOCODING procedure.	185
ST_ENABLE_DB procedure	187
ST_EXPORT_SHAPE procedure	189
ST_IMPORT_SHAPE procedure	192
ST_REGISTER_GEOCODER procedure	199
ST_REGISTER_SPATIAL_COLUMN procedure	203
ST_REMOVE_GEOCODING_SETUP procedure	205
ST_RUN_GEOCODING procedure	207
ST_SETUP_GEOCODING procedure	209
ST_UNREGISTER_GEOCODER procedure	213
ST_UNREGISTER_SPATIAL_COLUMN procedure	214

Chapter 18. Spatial functions 217

Considerations and associated data types for spatial functions	217
Treating values of ST_Geometry as values of a subtype	218
Spatial functions according to input type	218
Categories and uses for spatial functions	221
Constructor functions for converting to and from data exchange formats	221
Comparison functions for geographic features	227
Functions for obtaining information about geometries and indexes	231
Functions for generating new geometries from existing geometries	235
EnvelopesIntersect function.	236
MBR Aggregate functions	238

ST_AppendPoint function	240	ST_MBR function	325
ST_Area function	241	ST_MBRIntersects function	326
ST_AsBinary function	243	ST_LocateBetween or ST_MeasureBetween function	327
ST_AsGML function	244	ST_LocateBetween or ST_MeasureBetween function	329
ST_AsShape function	245	ST_MidPoint function	330
ST_AsText function	246	ST_MinM function	331
ST_Boundary function	248	ST_MinX function	333
ST_Buffer function	249	ST_MinY function	334
MBR Aggregate functions	252	ST_MinZ function	335
Union aggregate functions	253	ST_MLineFromText function	336
ST_Centroid function	255	ST_MLineFromWKB function	338
ST_ChangePoint function	256	ST_MPointFromText function	339
ST_Contains function	257	ST_MPointFromWKB function	340
ST_ConvexHull function	260	ST_MPolyFromText function	342
ST_CoordDim function	261	ST_MPolyFromWKB function	343
ST_Crosses function	263	ST_MultiLineString function	345
ST_Difference function	264	ST_MultiPoint function	346
ST_Dimension function	266	ST_MultiPolygon function	348
ST_Disjoint function	267	ST_NumGeometries function	349
ST_Distance function	268	ST_NumInteriorRing function	350
ST_DistanceToPoint function	271	ST_NumLineStrings function	351
ST_Endpoint function	272	ST_NumPoints function	352
ST_Envelope function	273	ST_NumPolygons function	353
ST_EnvIntersects function	274	ST_Overlaps function	354
ST_EqualCoordsys function	275	ST_Perimeter function	356
ST_Equals function	276	ST_PerpPoints function	357
ST_EqualsSRS function	278	ST_Point function	359
ST_ExteriorRing function	278	ST_PointAtDistance function	362
ST_FindMeasure or ST_LocateAlong function	280	ST_PointFromText function	363
ST_Generalize function	281	ST_PointFromWKB function	364
ST_GeomCollection function	283	ST_PointN function	365
ST_GeomCollFromTxt function	285	ST_PointOnSurface function	366
ST_GeomCollFromWKB function	286	ST_PolyFromText function	367
ST_Geometry function	287	ST_PolyFromWKB function	368
ST_GeometryN function	289	ST_Polygon function	369
ST_GeometryType function	290	ST_PolygonN function	372
ST_GeomFromText function	291	ST_Relate function	373
ST_GeomFromWKB function	292	ST_RemovePoint function	374
MBR Aggregate functions	294	ST_SrsId or ST_SRID function	375
Union aggregate functions	295	ST_SrsId or ST_SRID function	376
ST_GetIndexParms function	296	ST_SrsName function	378
ST_InteriorRingN function	299	ST_StartPoint function	378
ST_Intersection function	300	ST_SymDifference function	379
ST_Intersects function	301	ST_ToGeomColl function	381
ST_Is3d function	304	ST_ToLineString function	383
ST_IsClosed function	305	ST_ToMultiLine function	384
ST_IsEmpty function	306	ST_ToMultiPoint function	385
ST_IsMeasured function	307	ST_ToMultiPolygon function	386
ST_IsRing function	308	ST_ToPoint function	387
ST_IsSimple function	309	ST_ToPolygon function	388
ST_IsValid function	310	ST_Touches function	389
ST_Length function	311	ST_Transform function	390
ST_LineFromText function	313	ST_Union function	392
ST_LineFromWKB function	314	ST_Within function function	394
ST_LineString function	315	ST_WKBToSQL function	397
ST_LineStringN function	317	ST_WKTToSQL function	398
ST_M function function	318	ST_X function	399
ST_MaxM function	319	ST_Y function	400
ST_MaxX function	320	ST_Z function	401
ST_MaxY function	322	Union aggregate functions	403
ST_MaxZ function	323		

Chapter 19. Transform groups 405

ST_WellKnownText transform group	405
ST_WellKnownBinary transform group	406
ST_Shape transform group	408
ST_GML transform group	409

Chapter 20. Supported data formats 411

Well-known text (WKT) representation	411
Well-known binary (WKB) representation	416
Shape representation	418
Geography Markup Language (GML) representation	418

Chapter 21. Supported coordinate systems. 419

Coordinate systems syntax	419
Supported linear units	421

Supported angular units.	421
Supported spheroids	422
Supported prime meridians	424
Supported map projections.	424

Appendix A. Overview of the DB2 technical information 427

DB2 technical library in hardcopy or PDF format	427
Displaying SQL state help from the command line processor.	430
Accessing different versions of the DB2 Information Center	430
Terms and conditions.	430

Appendix B. Notices 433

Index 437

Chapter 1. DB2 Spatial Extender

Use DB2[®] Spatial Extender to generate and analyze spatial information about geographic features, and to store and manage the data on which this information is based. A geographic feature (sometimes called feature in this discussion, for short) is anything in the real world that has an identifiable location, or anything that could be imagined as existing at an identifiable location. A feature can be:

- An object (that is, a concrete entity of any sort); for example, a river, forest, or range of mountains.
- A space; for example, a safety zone around a hazardous site, or the marketing area serviced by a particular business.
- An event that occurs at a definable location; for example, an auto accident that occurred at a particular intersection, or a sales transaction at a specific store.

Features exist in multiple environments. For example, the objects mentioned in the preceding list—river, forest, mountain range—belong to the natural environment. Other objects, such as cities, buildings, and offices, belong to the cultural environment. Still others, such as parks, zoos, and farmland, represent a combination of the natural and cultural environments.

In this discussion, the term spatial information refers to the kind of information that DB2 Spatial Extender makes available to its users—namely, facts and figures about the locations of geographic features. Examples of spatial information are:

- Locations of geographic features on the map (for example, longitude and latitude values that define where cities are situated)
- The location of geographic features with respect to one another (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 data, can help you with such activities as defining the areas in which you provide services, and determining locations of possible markets. 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 and DB2 can produce information to answer these questions. Furthermore, front-end tools, though not required, can play a part. To illustrate: a visualization tool can put information produced by DB2 Spatial Extender—for

example, the location of concentrations of clientele and the proximity of major highways to proposed restaurants-in graphic form on a map. Business intelligence tools can put associated information-for example, names and descriptions of competing restaurants-in report form.

How data represents geographic features

In DB2 Spatial Extender, a geographic feature can be represented by one or more data items; for example, the data items in a row of a table. (A data item is the value or values that occupy a cell of a relational table.) For example, consider office buildings and residences. In Figure 1, each row of the BRANCHES table represents a branch office of a bank. Similarly, each row of the CUSTOMERS table in Figure 1, taken as a whole, represents a customer of the bank. However, a subset of each row-specifically, the data items that constitute a customer's address-represent the customer's residence.

BRANCHES

ID	NAME	ADDRESS	CITY	POSTAL CODE	STATE_PROV	COUNTRY
937	Airzone-Multern	92467 Airzone Blvd	San Jose	95141	CA	USA

CUSTOMERS

ID	LAST NAME	FIRST NAME	ADDRESS	CITY	POSTAL CODE	STATE_PROV	COUNTRY	CHECKING	SAVINGS
59-6396	Kriner	Endela	9 Concourt Circle	San Jose	95141	CA	USA	A	A

Figure 1. Data that represents geographic features. The row of data in the BRANCHES table represents a branch office of a bank. The address data in the CUSTOMERS table represents 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. This discussion refers to such data as business data.

A subset of the business data-the values that denote the branches' and customers' addresses-can be translated into values from which spatial information is generated. For example, as shown in Figure 1, one branch office's address is 92467 Airzone Blvd., San Jose, CA 95141, USA. A customer's address is 9 Concourt Circle, San Jose, CA 95141, USA. DB2 Spatial Extender can translate these addresses into values that indicate where the branch and the customer's home are located with respect to one another. Figure 2 on page 3 shows the BRANCHES and CUSTOMERS tables with new columns that are designated to contain such values.

BRANCHES

ID	NAME	ADDRESS	CITY	POSTAL CODE	STATE_PROV	COUNTRY	LOCATION
937	Airzone-Multern	92467 Airzone Blvd	San Jose	95141	CA	USA	

CUSTOMERS

ID	LAST NAME	FIRST NAME	ADDRESS	CITY	POSTAL CODE	STATE_PROV	COUNTRY	LOCATION	CHECKING	SAVINGS
59-6396	Kriner	Endela	9 Concourt Circle	San Jose	95141	CA	USA		A	A

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

Because spatial information will be derived from the data items stored in the LOCATION column, these data items are referred to in this discussion as spatial data.

The nature of spatial data

Spatial data is made up of coordinates that identify a location. Spatial Extender works with two-dimensional coordinates specified by x and y or longitude and latitude values.

A coordinate is a number that denotes either:

- A position along an axis relative to an origin, given a unit of length.
- A direction relative to a base line or plane, given a unit of angular measure.

For example, latitude is a coordinate that denotes an angle relative to the equatorial plane, usually in degrees. Longitude is a coordinate that denotes an angle relative to the Greenwich meridian, also usually in degrees. Thus, on a map, the position of Yellowstone National Park is defined by latitude 44.45 degrees north of the equator and longitude 110.40 degrees west of the Greenwich meridian. More precisely, these coordinates reference the center of Yellowstone National Park in the USA.

The definitions of latitude and longitude, their points, lines, and planes of reference, units of measure, and other associated parameters are referred to collectively as a coordinate system. Coordinate systems can be based on values other than latitude and longitude. These coordinate systems have their own points, lines, and planes of reference, units of measure, and additional associated parameters (such as the projection transformation).

The simplest spatial data item consists of a single coordinate pair that defines the position of a single geographic location. A more extensive spatial data item consists of several coordinates that define a linear path that a road or river might form. A third kind consists of coordinates that define the boundary of an area; for example, the boundary of a land parcel or flood plain.

Each spatial data item is an instance of a spatial data type. The data type for coordinates that mark a single location is ST_Point; the data type for coordinates that define a linear path is ST_LineString; and the data type for coordinates that define the boundary of an area is ST_Polygon. These types, together with the other spatial data types, are structured types that belong to a single hierarchy.

Where spatial data comes from

Spatial data can be obtained by using several methods.

You can obtain spatial data by:

- Deriving it from business data
- Generating it from spatial functions
- Importing it from external sources

Using business data as source data

DB2 Spatial Extender can derive spatial data from business data, such as addresses. This process is called geocoding.

To see the sequence involved, consider Figure 2 on page 3 as a “before” picture and Figure 3 as an “after” picture. Figure 2 on page 3 shows that the BRANCHES table and the CUSTOMERS table both have a 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 illustrates this result.

BRANCHES

ID	NAME	ADDRESS	CITY	POSTAL CODE	STATE_PROV	COUNTRY	LOCATION
937	Airzone-Multern	92467 Airzone Blvd	San Jose	95141	CA	USA	1653 3094

CUSTOMERS

ID	LAST NAME	FIRST NAME	ADDRESS	CITY	POSTAL CODE	STATE_PROV	COUNTRY	LOCATION	CHECKING	SAVINGS
59-6396	Kriner	Endela	9 Concourt Circle	San Jose	95141	CA	USA	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 were derived from the address in the ADDRESS, CITY, POSTAL CODE, STATE_PROV, and COUNTRY columns. Similarly, the LOCATION column in the BRANCHES table contains coordinates that were derived from the address in this table's ADDRESS, CITY, POSTAL CODE, STATE_PROV, and COUNTRY columns.

DB2 Spatial Extender uses a function, called a geocoder, to translate business data into coordinates to allow spatial functions to operate on the data.

Using functions to generate spatial data

You can use functions to generate spatial data from input data.

Spatial data can be generated not only by geocoders, but by other functions as well. 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 needs to define the zone that lies within a specified 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 zones. Figure 4 on page 5 shows the BRANCHES table with information that is supplied by ST_Buffer.

BRANCHES

ID	NAME	ADDRESS	CITY	POSTAL CODE	STATE_PROV	COUNTRY	LOCATION	SALES_AREA
937	Airzone-Multern	92467 Airzone Blvd	San Jose	95141	CA	USA	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.

Importing spatial data

Spatial Extender provides services to import spatial data in Shapefile format.

Spatial data in Shapefile format is available from many sources through the internet. You can download data and maps for US and world-wide features such as countries, states, cities, rivers, and more by selecting the DB2 Spatial Extender Sample Map Data offering in the Trials and demos Web page available from <http://www.ibm.com/software/data/spatial/db2spatial>.

You can import spatial data from files provided by external data sources. These files typically contain data that is applied to maps: street networks, flood plains, earthquake faults, and so on. By using such data in combination with spatial data that you produce, you can augment the spatial 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 this zone.

How features, spatial information, spatial data, and geometries fit together

A summary of several basic concepts that underlie the operations of DB2 Spatial Extender such as geographic features, spatial information, spatial data, and geometries is provided.

DB2 Spatial Extender lets you obtain facts and figures that pertain to things that can be defined geographically—that is, in terms of their location on earth, or within a region of the earth. The DB2 documentation refers to such facts and figures as *spatial information*, and to the things as *geographic features* (called *features* here, for short).

For example, you could use DB2 Spatial Extender to determine whether any populated areas overlap the proposed site for a landfill. The populated areas and the proposed site are features. A finding as to whether any overlap exists would be an example of spatial information. If overlap is found to exist, the extent of it would also be an example of spatial information.

To produce spatial information, DB2 Spatial Extender must process data that defines the locations of features. Such data, called *spatial data*, consists of coordinates that reference the locations on a map or similar projection. For

example, to determine whether one feature overlaps another, DB2 Spatial Extender must determine where the coordinates of one of the features are situated with respect to the coordinates of the other.

In the world of spatial information technology, it is common to think of features as being represented by symbols called *geometries*. Geometries are partly visual and partly mathematical. Consider their visual aspect. The symbol for a feature that has width and breadth, such as a park or town, is a multisided figure. Such a geometry is called a *polygon*. The symbol for a linear feature, such as a river or a road, is a line. Such a geometry is called a *linestring*.

A geometry has properties that correspond to facts about the feature that it represents. Most of these properties can be expressed mathematically. For example, the coordinates for a feature collectively constitute one of the properties of the feature's corresponding geometry. Another property, called *dimension*, is a numeric value that indicates whether a feature has length or breadth.

Spatial data and certain spatial information can be viewed in terms of geometries. Consider the example, described earlier, of the populated areas and the proposed landfill site. The spatial data for the populated areas includes coordinates stored in a column of a table in a DB2 database. The convention is to regard what is stored not simply as data, but as actual geometries. Because populated areas have width and breadth, you can see that these geometries are polygons.

Like spatial data, certain spatial information is also viewed in terms of geometries. For example, to determine whether a populated area overlaps a proposed landfill site, DB2 Spatial Extender must compare the coordinates in the polygon that symbolizes the site with the coordinates of the polygons that represent populated areas. The resulting information—that is, the areas of overlap—are themselves regarded as polygons: geometries with coordinates, dimensions, and other properties.

Chapter 2. Geometries

In DB2 Spatial Extender, the operational definition of geometry is “a model of a geographic feature.”

Webster's Revised Unabridged Dictionary defines *geometry* as “That branch of mathematics which investigates the relations, properties, and measurement of solids, surfaces, lines, and angles; the science which treats of the properties and relations of magnitudes; the science of the relations of space.” The word geometry has also been used to denote the geometric features that, for the past millennium or more, cartographers have used to map the world. An abstract definition of this new meaning of geometry is “a point or aggregate of points representing a feature on the ground.”

In DB2 Spatial Extender, the model can be expressed in terms of the feature's coordinates. The model conveys information; for example, the coordinates identify the position of the feature with respect to fixed points of reference. 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 represents are regarded as properties of the geometry. Several kinds of geometries have other properties as well; for example, area, length, and boundary.

The geometries supported by DB2 Spatial Extender form a hierarchy, which is shown in the following figure. The geometry hierarchy is defined by the OpenGIS Consortium, Inc. (OGC) document "OpenGIS Simple Features Specification for SQL". Seven members of the hierarchy are instantiable. That is, they can be defined with specific coordinate values and rendered visually as the figure shows.

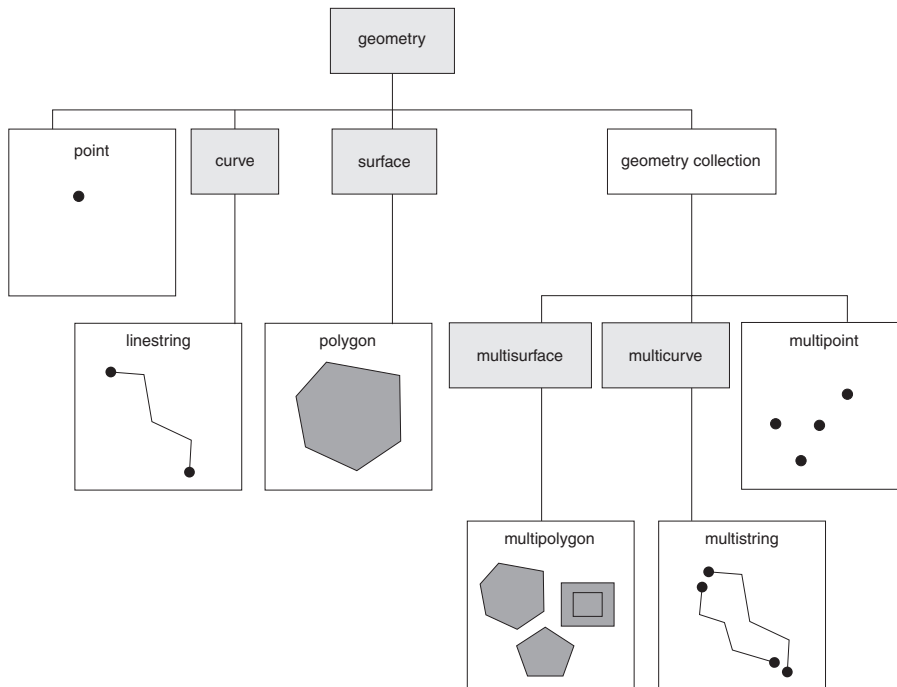


Figure 5. Hierarchy of geometries supported by DB2 Spatial Extender. Instantiable geometries in this figure include examples of how they might be rendered visually.

The spatial data types supported by DB2 Spatial Extender are implementations of the geometries shown in the figure.

As the figure indicates, a superclass called geometry is the root of the hierarchy. The root type and other proper subtypes in the hierarchy are not instantiable. Additionally, users can define their own instantiable or not instantiable proper subtypes.

The subtypes are divided into two categories: the base geometry subtypes, and the homogeneous collection subtypes.

The base geometries include:

Points A single point. Points represent 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 world map shows that each city on the map is located at the intersection of a parallel and a meridian. A point could represent each city.

Linestrings

A line between two or more points. It does not have to be a straight line. Linestrings represent linear geographic features (for example, streets, canals, and pipelines).

Polygons

A polygon or surface within a polygon. Polygons represent multisided geographic features (for example, welfare districts, forests, and wildlife habitats).

The homogeneous collections include:

Multipoints

A multiple point geometry collection. Multipoints represent 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

A multiple curve geometry collection with multiple linestrings. Multilinestrings represent multipart features that are made up (for example, river systems and highway systems).

Multipolygons

A multiple surface geometry collection with multiple polygons. Multipolygons represent 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.

Properties of geometries

This topic describes properties that are available for geometries.

These properties are:

- The type that a geometry belongs to
- Geometry coordinates
- 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 minimum bounding rectangle or envelope
- Dimension
- The identifier of the spatial reference system with which a geometry is associated

Geometry types

Each geometry belongs to a type in the hierarchy of geometries supported by DB2 Spatial Extender.

The types in the hierarchy to be defined with specific coordinate values are:

- Points
- Linestrings
- Polygons
- Geometry collections
- Multipoints
- Multilinestrings
- Multipolygons

Geometry coordinates

All geometries include at least one X coordinate and one Y coordinate, unless they are empty geometries, in which case they contain no coordinates at all.

In addition, a geometry can include one or more Z coordinates and M coordinates. X, Y, Z, and M coordinates are represented as double-precision numbers. The following subsections explain:

- X and Y coordinates
- Z coordinates
- M coordinates

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.

Z coordinates

For geometries that 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.

M coordinates

An M coordinate (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 highways in your application. 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 locations along the highway. M coordinates are represented as double-precision numbers.

Interior, boundary, and exterior

All geometries occupy a position in space defined by their interiors, boundaries, and exteriors.

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.

Simple or non-simple geometries

The values of some geometry subtypes (linestrings, multipoints, and multilinestrings) are either simple or non-simple. A geometry is simple if it obeys all the topological rules imposed on its subtype 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. Points, surfaces, multisurfaces, and empty geometries are always simple.

Closed geometries

A curve is closed if its start and end points are the same. A multicurve is closed if all of its elements are closed. A ring is a simple, closed curve.

Empty or not empty geometries

A geometry is empty if it does not have any points. The envelope, boundary, interior, and exterior of an empty geometry are not defined and will be represented as null.

An empty geometry is always simple. Empty polygons and multipolygons have an area of 0.

Minimum bounding rectangle (MBR)

The MBR of a geometry is the bounding geometry formed by the minimum and maximum (X,Y) coordinates.

Except for the following special cases, the MBRs of geometries form a boundary rectangle when:

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

Dimension of geometries

A geometry can have a dimension value to indicate whether a geometry is empty or the geometry has a length or area.

The dimensions values are as follows:

- 1 Indicates that a geometry is empty
- 0 Indicates that a geometry has no length and an area of 0 (zero)
- 1 Indicates that a geometry has a length larger than 0 (zero) and an area of 0 (zero)
- 2 Indicates that a geometry has an area that is larger than 0 (zero)

The point and multipoint subtypes have a dimension of zero. Points represent dimensional features that can be modeled with a single tuple of coordinates, while multipoint subtypes represent data that must be modeled with a set of points.

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

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

Spatial reference system identifier

The numeric identifier for a spatial reference system determines which spatial reference system is used to represent the geometry.

All spatial reference systems known to the database can be accessed through the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view.

Chapter 3. How to use DB2 Spatial Extender

Support and use of DB2 Spatial Extender involves two main activities: setting up DB2 Spatial Extender and working on projects that use spatial data. This topic introduces the interfaces you can use to perform spatial tasks.

Interfaces to DB2 Spatial Extender and associated functionality

There are a wide variety of interfaces that can be used to set up DB2 Spatial Extender.

You can use any of the following interfaces to set up DB2 Spatial Extender and create projects that use spatial data:

- Application programs that call DB2 Spatial Extender stored procedures.
- A command line processor (CLP) provided by DB2 Spatial Extender. The DB2 Spatial Extender CLP is often referred to as the db2se CLP.
- SQL queries that you submit from the DB2 CLP, or from an application program.
- Open source projects that include support for DB2 Spatial Extender. Some open source projects with this support include:
 - IBM Data Management Geobrowser for DB2 and Informix is a geobrowser to visualize spatial tables and the results of spatial analysis in a graphical form. For more information, see <http://www.ibm.com/developerworks/data/tutorials/dm-1103db2geobrowser/index.html>.
 - GeoTools () is a Java™ library for building spatial applications. For more information, see <http://www.geotools.org/>.
 - GeoServer is a Web map server and Web feature server. For more information, see <http://geoserver.org/display/GEOS/Welcome>.
 - uDIG is a desktop spatial data visualization and analysis application. For more information, see <http://udig.refrains.net/>.
 - GIS products such as Esri ArcGIS

Setting up DB2 Spatial Extender

This procedure describes the steps that you should perform to set up DB2 Spatial Extender.

Procedure

To set up DB2 Spatial Extender:

1. Plan and make preparations (decide what projects to create, decide what interface or interfaces to use, select personnel to administer DB2 Spatial Extender and create the projects, and so on).
2. Install DB2 Spatial Extender.
3. Review your database configuration parameter settings and adjust them if necessary. You must ensure that your database has enough memory and space for spatial functions, log files, and DB2 Spatial Extender applications.

4. Set up spatial resources for your database. These resources include a system catalog, spatial data types, spatial functions, a geocoder, and other objects. For more information, see [Enabling a database for spatial operations](#).

Example

The following example shows the steps performed by a fictional company, called Safe Harbor Real Estate Insurance Company, to set up DB2 Spatial Extender:

1. The Safe Harbor Real Estate Insurance Company's information systems environment includes a DB2 database system and a separate file system for spatial data only. To an extent, query results can include combinations of data from both systems. For example, a DB2 table stores information about revenue, and a file in the file system contains locations of the company's branch offices. Therefore, it is possible to find out which offices bring in revenues of specified amounts, and then to determine where these offices are located. But data from the two systems cannot be integrated (for example, users cannot join DB2 columns with file system records, and DB2 services such as query optimization are unavailable to the file system.) To overcome these disadvantages, Safe Harbor acquires DB2 Spatial Extender and establishes a new Spatial Development department (called a Spatial department, for short).

The Spatial department's first mission is to include DB2 Spatial Extender in Safe Harbor's DB2 environment:

- 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.
 - Because the administration team has a strong UNIX[®] background, it decides to use the `db2se CLP` to administer DB2 Spatial Extender.
 - 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`. The `CUSTOMERS` table has `LATITUDE` and `LONGITUDE` columns that were populated as part of the address cleansing process.
2. The spatial administration team installs DB2 Spatial Extender on a DB2 environmentUNIX.
 3. A member of the spatial administration team adjusts the transaction log characteristics, application heap size, and application control heap size to values suited to the requirements for DB2 Spatial Extender.
 4. The spatial administration team sets up resources that will be required by the projects that it is planning.
 - A member of the team issues the `db2se enable_db` command to obtain the resources that enable the database for spatial operations. These resources include the DB2 Spatial Extender catalog, spatial data types, spatial functions, and so on.

What to do next

After setting up DB2 Spatial Extender, you can start creating projects that use spatial data.

Creating projects that use spatial data

After setting up DB2 Spatial Extender, you are ready to undertake projects that use spatial data. This procedure describes the steps involved in creating such projects.

Before you begin

- Set up DB2 Spatial Extender

Procedure

To create a project that uses spatial data:

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 whether an existing spatial reference system meets your needs. If none does, create one.

A spatial reference system is a set of parameter values that includes:

- Coordinates that define the maximum possible extent of space referenced by a given range of coordinates. You need to determine the maximum possible range of coordinates that can be determined from the coordinate system that you are using, and to select or create a spatial reference system that reflects this range.
 - The name of the coordinate system from which the coordinates are derived.
 - Numbers used in mathematical operations to convert coordinates received as input into values that can be processed with maximum efficiency. The coordinates are stored in their converted form and returned to the user in their original form.
3. Create spatial columns as needed. Note that in many cases, if data in a spatial column is to be read by a visualization tool, the column must be the only spatial column in the table or view to which it belongs. Alternatively, if the column is one of multiple spatial columns in a table, it could be included in a view that has no other spatial columns, and visualization tools could read the data from this view.
 4. Set up spatial columns for access by visualization tools, as needed, by registering the columns in the DB2 Spatial Extender catalog. When you register a spatial column, DB2 Spatial Extender imposes a constraint that all data in the column must belong to the same spatial reference system. This constraint enforces integrity of the data—a requirement of most visualization tools.
 5. Populate spatial columns by performing one of the following actions:
 - a. For a project that requires spatial data to be imported, import the data.
 - b. For a project that requires a spatial column to be set from the LATITUDE and LONGITUDE columns, run an SQL UPDATE statement using the `db2gse.ST_Point` constructor.
 6. Facilitate access to spatial columns, as needed. This involves defining indexes that enable DB2 to access spatial data quickly, and defining views that enable users to retrieve interrelated data efficiently. If you want visualization tools to access the views' spatial columns, you might need to register these columns with DB2 Spatial Extender as well.
 7. Analyze generated spatial information and related business information. This task involves querying spatial columns and related non-spatial columns. In such queries, you can include DB2 Spatial Extender functions that return a wide variety of information. For example, coordinates that define a proposed

safety zone around a hazardous waste site, or the minimum distance between this site and the nearest public building.

Example

The following example is a continuation from the example in Setting up DB2 Spatial Extender. It shows the steps performed by Safe Harbor Real Estate Insurance Company to create a project for the integration of business and spatial data.

1. The Spatial department prepares to develop a project; for example:
 - The management team sets these 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)
 - This particular project will be concerned with customers and offices in the United States. Therefore, the spatial administration team decides to use the GCS_NORTH_AMERICAN_1983 coordinate system for the United States that DB2 Spatial Extender provides.
 - The spatial administration team decides what data is needed to meet the project's goals and what tables will contain this data.
2. DB2 Spatial Extender provides a spatial reference system, called NAD83_SRS_1, that is designed to be used with GCS_NORTH_AMERICAN_1983. The spatial administration team decides to use NAD83_SRS_1.
3. The spatial administration team defines columns to contain spatial data.
 - The team verifies that the table already contains customer LATITUDE and LONGITUDE columns. The values in these columns will be used subsequently to translate to spatial point values.
 - The team creates the OFFICE_LOCATIONS and OFFICE_SALES tables to contain data that is now stored in a separate file system using an industry-standard shapefile format. 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 derived by the geocoder will go into a LOCATION column in the OFFICE_LOCATIONS table, and the data that defines the zones will go into a SALES_AREA column in the OFFICE_SALES table.
4. The spatial administration team expects to use visualization tools to render the content of the LOCATION columns and the SALES_AREA column graphically on a map. Therefore, the team registers all three columns.
5. The spatial administration team populates the LOCATION column in the CUSTOMER table, the OFFICE_LOCATIONS table, the OFFICE_SALES table, and a new HAZARD_ZONES table:
 - The team uses the statement `UPDATE CUSTOMERS SET LOCATION = db2gse.ST_Point(LONGITUDE, LATITUDE,1)` to populate the LOCATION value from LATITUDE and LONGITUDE.
 - The team imports data to the OFFICE_LOCATIONS and OFFICE_SALES tables by using the **db2se import_shape** command.
 - The team creates a HAZARD_ZONES table, registers its spatial columns, and imports data to it. The data comes from a file supplied by a map vendor.

6. The spatial administration team creates indexes for the registered columns. Then, they create a view that joins columns from the CUSTOMERS and HAZARD ZONES tables and register the spatial columns in this view.
7. The spatial analysis team runs queries to obtain information that will help them determine where to establish new branch offices, and to adjust premiums on the basis of customers' proximity to hazard areas.

Chapter 4. Getting started with DB2 Spatial Extender

You should get familiar with the instructions for installing and configuring Spatial Extender in supported operating systems. Also, learn how to troubleshoot some of the installation and configuration problems that you might encounter as you work Spatial Extender.

Setting up and installing DB2 Spatial Extender

Setting up and installing DB2 Spatial Extender is required to create an environment that supports spatial operations.

Before you begin

Ensure that you meet the installation requirements for DB2 Spatial Extender and DB2 database products. If your system fails to meet any of the software prerequisites, the installation will fail.

About this task

A DB2 Spatial Extender environment consists of a server and a client.

A DB2 Spatial Extender server consist of a DB2 data server installation with the DB2 Spatial Extender software installed.

A DB2 Spatial Extender client consist of anIBM® data server client installation with the DB2 Spatial Extender software installed.

Databases enabled for spatial operations are located in the server. They can be accessed from a client. Spatial data residing in the databases can be accessed using DB2 Spatial Extender stored procedures and spatial queries from the server or the client.

Also generally part of a typical component of a DB2 Spatial Extender environment is a geobrowser. Although they are not required, they are useful to visually render the results of spatial queries, generally in the form of maps. A geobrowser is not included with a DB2 Spatial Extender installation. However, you can obtain the IBM Data Management Geobrowser for DB2 & Informix® to view spatial data.

Procedure

To set up and install DB2 Spatial Extender on a DB2 server or IBM data server client:

1. Ensure that your system meets all software requirements. Refer to “System requirements for installing Spatial Extender” on page 20.
2. Install DB2 Spatial Extender by performing one of the following tasks:
 - “Installing DB2 Spatial Extender using the DB2 Setup wizard (Windows)” on page 21
 - “Installing DB2 Spatial Extender using the DB2 Setup wizard (Linux and UNIX)” on page 22
 - Installing a DB2 product using a response file (Windows)
 - Installing a DB2 product using a response file (Linux and UNIX)

For installations using a response file, you must indicate the following keywords for DB2 server installation:

```
INSTALL_TYPE=CUSTOM  
COMP=SPATIAL_EXTENDER_CLIENT_SUPPORT  
COMP=SPATIAL_EXTENDER_SERVER_SUPPORT
```

You must indicate the following keywords for an IBM data server client installation:

```
INSTALL_TYPE=CUSTOM  
COMP=SPATIAL_EXTENDER_CLIENT_SUPPORT
```

3. Create a DB2 instance if you do not already have one. To do this use the **db2icrt** DB2 command from a DB2 command window.
4. Verify the DB2 Spatial Extender installation was successful by testing the DB2 Spatial Extender environment. Refer to “Verifying the Spatial Extender installation” on page 24.
5. Optional: Download and install the IBM Data Management Geobrowser for DB2 and Informix. A free copy can be downloaded from <https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=swg-dm-geobrowser>.

Related information:

 Analyze DB2 spatial data with a free geobrowser

System requirements for installing Spatial Extender

Before you install DB2 Spatial Extender, ensure that your system meets all the software space requirements.

Operating systems

You can install DB2 Spatial Extender on 32-bit operating systems on Intel-based computers such as:

- Windows
- Linux.

You can also install DB2 Spatial Extender on 64-bit operating systems such as:

- AIX®,
- HP-UX,
- Solaris Operating System SPARC,
- Linux x86,
- Linux for System z®, and
- Windows

Software requirements

To install DB2 Spatial Extender, you must meet the software requirements for DB2 database products. For more details, see “Installation requirements for DB2 database products” in *Installing DB2 Servers*.

Spatial Extender on DB2 servers

You can install the Spatial Extender as part of a DB2 database product or install it on an existing DB2 copy. You must meet the software requirements for the DB2 database server products.

Spatial Extender on IBM data server clients

If you install the IBM data server client on Windows operating systems, the custom installation includes Spatial Extender.

If you install a DB2 database server product on AIX, HP-UX, Solaris Operating System, Linux for Intel, or Linux on System z operating systems, Spatial Extender can be optionally installed by selecting the installation type as custom.

You must meet the software requirements for IBM data server client.

Installing DB2 Spatial Extender using the DB2 Setup wizard (Windows)

You can use the DB2 Setup wizard to install DB2 Spatial Extender on Windows operating systems as part of a DB2 installation.

Before you begin

Before you start the DB2 Setup wizard:

- Ensure that your system meets installation, memory, and disk requirements.
- You must have a local Administrator user account with the recommended user rights to perform the installation. In DB2 database servers with single-partition databases where LocalSystem can be used as the DAS and DB2 instance user, a non-administrator user with elevated privileges can perform the installation.

Note: If a non-Administrator user account is going to do the product installation, then the VS2010 runtime library must be installed before attempting to install a DB2 database product. The VS2010 runtime library is needed on the operating system before the DB2 database product can be installed. The VS2010 runtime library is available from the Microsoft runtime library download website. There are two choices: choose `vcredist_x86.exe` for 32-bit systems or `vcredist_x64.exe` for 64-bit systems.

- You should close all programs so that the installation program can update any files on the computer without requiring a reboot.
- For installations from a virtual drive, you must map the network drive to a Windows drive letter. The DB2 Setup wizard does not support installation from a virtual drive or an unmapped network drive (such as `\\hostname\sharename` in Windows Explorer).

About this task

This task is part of the larger task of “Setting up and installing DB2 Spatial Extender” on page 19.

Procedure

To install DB2 Spatial Extender as part of a DB2 server or IBM data server client:

1. Log on to the system with the local Administrator account that you have identified for the DB2 installation.
2. If you have the DB2 database product DVD, insert it into the drive. If enabled, the autorun feature automatically starts the DB2 Setup Launchpad. If the autorun does not work, use Windows Explorer to browse the DB2 database product DVD and double-click the **setup** icon to start the DB2 Setup Launchpad.

3. If you downloaded the DB2 database product from Passport Advantage®, run the executable file to extract the DB2 database product installation files. Use Windows Explorer to browse the DB2 installation files and double-click the **setup** icon to start the DB2 Setup Launchpad.
4. Review installation prerequisites and the release notes from the DB2 Setup launchpad for late-breaking information, or you can proceed directly to the installation.
5. Click **Install a Product** and the Install a Product window displays the products available for installation.
6. Install Spatial Extender by using one of the following options:
 - To create a new DB2 copy with Spatial Extender, click **Install New** to start the installation. Select the installation type as custom to include the Spatial Extender as part of the installation. Proceed through the installation following the DB2 Setup wizard prompts.
 - To install Spatial Extender on an existing DB2 copy, click **Work with Existing**. Select the installation type as custom to include the Spatial Extender as part of the installation. Proceed through the installation following the DB2 Setup wizard prompts.

After the installation is complete, look for any warning or error messages in the identified log file.

What to do next

After you install DB2 Spatial Extender, create a DB2 instance in new DB2 copies, and then verify that the installation was successful.

Installing DB2 Spatial Extender using the DB2 Setup wizard (Linux and UNIX)

To install DB2 Spatial Extender on Linux or UNIX operating systems, use the DB2 Setup wizard, or a response file.

Before you begin

Before you start the DB2 Setup wizard:

- Ensure that your system meets installation, memory, and disk requirements.
- Ensure you have a supported browser installed.
- Ensure that the DB2 database product image is available on the computer. You can obtain a DB2 installation image either by purchasing a physical DB2 database product DVD, or by downloading an installation image from Passport Advantage.
- If you are installing a non-English version of a DB2 database product, ensure that you have the appropriate National Language Packages available.
- Ensure that you have installed X Linux software capable of rendering a graphical user, the X Linux server is running, the *DISPLAY* variable is defined. The DB2 Setup wizard is a graphical installer.
- If you are using security software in your environment, you must manually create required DB2 users before you start the DB2 Setup wizard.

About this task

This task is part of the larger task of “Setting up and installing DB2 Spatial Extender” on page 19.

You can install a DB2 database product using either root or non-root authority.

Procedure

To install DB2 Spatial Extender as part of a DB2 server or IBM data server client:

1. If you have a physical DB2 database product DVD, change to the directory where the DB2 database product DVD is mounted by entering the following command:

```
cd /dvdrom
```

where */dvdrom* represents the mount point of the DB2 database product DVD.

2. If you downloaded the DB2 database product image, you must extract and untar the product file.

- a. Extract the product file:

```
gzip -d product.tar.gz
```

where *product* is the name of the product that you downloaded.

- b. Untar the product file:

On Linux operating systems

```
tar -xvf product.tar
```

On AIX, HP-UX, and Solaris operating systems

```
guntar -xvf product.tar
```

where *product* is the name of the product that you downloaded.

- c. Change directory:

```
cd ./product
```

where *product* is the name of the product that you downloaded.

Note: If you downloaded a National Language Package, untar it into the same directory. This will create the subdirectories (for example *./nlpack*) in the same directory, and allows the installer to automatically find the installation images without prompting.

3. Enter the **./db2setup** command from the directory where the database product image resides to start the DB2 Setup wizard.
4. The IBM DB2 Setup Launchpad opens. From this window, you can view installation prerequisites and the release notes, or you can proceed directly to the installation. You can also review the installation prerequisites and release notes for late-breaking information.
5. Click **Install a Product** and the **Install a Product** window will display the products available for installation.
6. Install Spatial Extender by using one of the following options:
 - To create a new DB2 copy with Spatial Extender, click **Install New** to start the installation. Select the installation type as custom to include the Spatial Extender as part of the installation. Proceed through the installation following the DB2 Setup wizard prompts.

- To install Spatial Extender on an existing DB2 copy, click **Work with Existing**. Select the installation type as custom to include the Spatial Extender as part of the installation. Proceed through the installation following the DB2 Setup wizard prompts.

After the installation is complete, look for any warning or error messages in the identified log file.

What to do next

After you install DB2 Spatial Extender, create a DB2 instance in new DB2 copies, and then verify that the installation was successful.

Verifying the Spatial Extender installation

After installing DB2 Spatial Extender you should validate the installation.

Before you begin

The following prerequisites must be completed before you can validate a DB2 Spatial Extender installation:

- DB2 Spatial Extender must be installed on a computer.
- A DB2 instance must have been created on the computer where the DB2 data server is installed.

About this task

This task is one that should be performed following the task of setting up and configuring DB2 Spatial Extender. For more information, see “Setting up and installing DB2 Spatial Extender” on page 19. The task comprises of creating a DB2 database and running a DB2 Spatial Extender sample application that is provided with DB2 that can be used to verify that a core set of DB2 Spatial Extender functionality is working correctly. This test is sufficient to validate that DB2 Spatial Extender has been installed and configured correctly.

Procedure

To verify the DB2 Spatial Extender installation:

1. Linux and UNIX: Log on to the system with the user ID that corresponds to the DB2 instance owner role.
2. Create a DB2 database. To do this, open a DB2 Command Window and enter the following:

```
db2 create database mydb
```

where *mydb* is the database name.

3. Ensure that you have a system temporary table space with a page size of 8 KB or larger and with a minimum size of 500 pages. If you do not have such a table space, see “Creating temporary table spaces” in *Database Administration Concepts and Configuration Reference* for details on how to create it. This is a requirement to run the runGSEdemo program.
4. For Windows operating systems, set the **agent_stack_sz** database manager configuration parameter to a value of 100 or larger. The following example shows the CLP command to set the parameter to 110:

```
UPDATE DBM CFG USING AGENT_STACK_SZ 110
```


5. Go to the directory where the runGSEdemo program resides:

- For DB2 Spatial Extender installations on Linux and UNIX operating systems, enter:

```
cd $HOME/sqlllib/samples/extenders/spatial
```

where *\$HOME* is the instance owner's home directory.

- For DB2 Spatial Extender installation on Windows operating systems, enter:

```
cd c:\Program Files\IBM\sqlllib\samples\extenders\spatial
```

where *c:\Program Files\IBM\sqlllib* is the directory in which you installed DB2 Spatial Extender.

6. Run the installation check program. At the DB2 command line, enter the **runGseDemo** command:

```
runGseDemo mydb userID password
```

where *mydb* is the database name.

Chapter 5. Upgrading to DB2 Spatial Extender Version 10.5

Upgrading to DB2 Spatial Extender to V10.5 on systems that you installed DB2 Spatial Extender Version 9.5 or Version 9.7 requires more than installing DB2 Spatial Extender to V10.5. You must perform the appropriate upgrade task for these systems.

The following tasks describe all of the steps to upgrade DB2 Spatial Extender from Version 9.5 or Version 9.7 to V10.5:

- “Upgrading DB2 Spatial Extender”
- “Updating from 32-bit to 64-bit DB2 Spatial Extender” on page 28

If your DB2 environment has other components such as DB2 servers, clients, and database applications, refer to “Upgrade to DB2 V10.5” in *Upgrading to DB2 Version 10.5* for details about how to upgrade these components.

Upgrading DB2 Spatial Extender

Upgrading DB2 Spatial Extender requires that you upgrade your DB2 server first and then upgrade specific database objects and data in spatially enabled databases.

Before you begin

Before you start the upgrade process:

- Ensure that your system meets the installation requirements for DB2 Spatial Extender Version 10.1.
- Ensure that you have DBADM and DATAACCESS authority on the spatially enabled databases.
- Ensure that you have a system temporary table space with a page size of 8 KB or larger and with a minimum size of 500 pages.

About this task

If you have installed DB2 Spatial Extender Version 9.5 or Version 9.7, you must complete the following steps before using an existing spatially enabled database with DB2 Spatial Extender Version 10.1. This topic describes the steps required to upgrade spatially enabled databases from an earlier version of DB2 Spatial Extender.

Procedure

To upgrade DB2 Spatial Extender to Version 10.1:

1. Upgrade your DB2 server from Version 9.5 or Version 9.7 to Version 10.1 using one of the following tasks:
 - “Upgrading a DB2 servers (Windows)” in *Upgrading to DB2 Version 10.5*
 - “Upgrading a DB2 servers (Linux and UNIX)” in *Upgrading to DB2 Version 10.5*

In the upgrade task, you must install DB2 Spatial Extender Version 10.1 after you install DB2 Version 10.1 to successfully upgrade your instances.

2. Terminate all connections to the database.

3. Upgrade your spatially enabled databases from Version 9.5 or Version 9.7 to Version 10.1 using the **db2se upgrade** command.

Results

Check the messages file for details on any errors you receive. The messages file also contains useful information such as indexes, views, and the geocoding setup that was upgraded.

Updating from 32-bit to 64-bit DB2 Spatial Extender

Updating from 32-bit DB2 Spatial Extender V10.5 to 64-bit DB2 Spatial Extender V10.5 requires that you back up spatial indexes, update your DB2 server to 64-bit DB2 V10.5, install 64-bit DB2 Spatial Extender V10.5, and then restore the spatial indexes that you backed up.

Before you begin

- Ensure that you have a system temporary table space with a page size of 8 KB or larger and with a minimum size of 500 pages.

About this task

If you are upgrading from 32-bit DB2 Spatial Extender Version 9.5 or Version 9.7 to 64-bit DB2 Spatial Extender Version 9.7, you need to perform the “Upgrading DB2 Spatial Extender” on page 27 task instead.

Procedure

To update a 32-bit DB2 Spatial Extender Version 9.7 server to 64-bit DB2 Spatial Extender Version 9.7:

1. Back up your database.
2. Save the existing spatial indexes by issuing the **db2se save_indexes** command from an operating-system command prompt.
3. Update your 32-bit DB2 server from Version 9.5 or Version 9.7 to 64-bit DB2 V10.5 using one of the following tasks:
 - “Updating your 32-bit DB2 instances to 64-bit instances (Windows)” in *Installing DB2 Servers* .
 - “Updating DB2 copies (Linux and UNIX)” in *Database Administration Concepts and Configuration Reference*.
4. Install DB2 Spatial Extender V10.5.
5. Restore the spatial indexes by issuing the **db2se restore_indexes** command from an operating-system command prompt.

Chapter 6. Setting up spatial resources for a database

After you set up your database to accommodate spatial data, you are ready to supply the database with resources that you will need when you create and manage spatial columns and analyze spatial data.

The spatial resources include:

- Objects provided by Spatial Extender to support spatial operations. For example, stored procedures to administrate a database, and spatial data types and utilities for geocoding, importing, or exporting spatial data.
- Any geocoders that users or vendors provide.

To make these resources available, you must enable your database for spatial operations, set up access to reference data, and register third-party geocoders.

Inventory of resources supplied for your database

DB2 Spatial Extender provides the database with several resources to enable it for support of spatial operations.

These resources are:

- Stored procedures. Spatial operations such as commands to import spatial data, invoke one of these stored procedures to perform the operation.
- Spatial data types. You must assign a spatial data type to each table or view column that is to contain spatial data.
- DB2 Spatial Extender's catalog. Certain operations depend on this catalog. For example, before you can access a spatial column from the visualization tools, the tool might require that the spatial column be registered in the catalog.
- A spatial grid index. You can define grid indexes on spatial columns.
- Spatial functions. You use these to work with spatial data in a number of ways such as determining relationships between geometries and to generate more spatial data.
- Definitions of coordinate systems.
- Default spatial reference systems.
- Two schemas: DB2GSE and ST_INFORMTN_SCHEMA. DB2GSE contains the objects just listed: stored procedures, spatial data types, the DB2 Spatial Extender catalog, and so on. Views in the catalog are available also in ST_INFORMTN_SCHEMA to conform with the SQL/MM standard..

Enabling a database for spatial operations

Enabling the database for spatial operations consist of having DB2 Spatial Extender supply a database with resources for creating spatial columns and manipulating spatial data.

Before you begin

Before you enable a database for spatial operations:

- Ensure that your user ID have DBADM authority on the database.

- Ensure that you have a system temporary table space with a page size of 8 KB or larger and with a minimum size of 500 pages.

Procedure

Enable a database for spatial operations in any of the following ways:

- Issue the **db2se enable_db** command.
- Run an application that calls the DB2GSE.ST_ENABLE_DB procedure.

You can explicitly choose the table space in which you want the DB2 Spatial Extender catalog to reside. If you do not do so, the DB2 database system will use the default table space.

If you want to use spatial data in a partitioned database environment, do not use the default table space. For best results, enable the database in a table space that is defined for a single nodes. Typically a single node table space is defined for small tables where partitioning is not helpful.

For example, you can define a single node table space using the db2se command line processor:

```
db2se enable_db my_db -tableCreationParameters "IN NODE0TBS"
```

Many queries can be performed more efficiently if the spatial reference systems table DB2GSE.GSE_SPATIAL_REFERENCE_SYSTEMS is replicated across all nodes that are used for business tables that will be queried. You can re-create the DB2GSE.GSE_SRS_REPLICATED_AST table with statements like this:

```
drop table db2gse.gse_srs_replicated_ast;
-- MQT to replicate the SRS information across all nodes
-- in a partitioned database environment
CREATE TABLE db2gse.gse_srs_replicated_ast AS
  ( SELECT srs_name, srs_id, x_offset, x_scale, y_offset, z_offset, z_scale,
        m_offset, m_scale, definition
    FROM db2gse.gse_spatial_reference_systems )
DATA INITIALLY DEFERRED
REFRESH IMMEDIATE
ENABLE QUERY OPTIMIZATION
REPLICATED
IN ts_data partitioned_tablespace
;

REFRESH TABLE db2gse.gse_srs_replicated_ast;

CREATE INDEX db2gse.gse_srs_id_ast
ON db2gse.gse_srs_replicated_ast ( srs_id )
;

RUNSTATS ON TABLE db2gse.gse_srs_replicated_ast and indexes all;
```

Registering a geocoder

Before geocoders can be used, they must be registered.

Before you begin

Before you can register a geocoder, your user ID must hold DBADM authority on the database in which the geocoder resides.

Procedure

You can register a geocoder in any of the following ways:

- Issue the **db2se register_gc** command.

- Run an application that calls the DB2GSE.ST_REGISTER_GEOCODER procedure.

Chapter 7. Setting up spatial resources for a project

After your database is enabled for spatial operations, you are ready to create projects that use spatial data.

Among the resources that each project requires are a coordinate system to which spatial data conforms and a spatial reference system that defines the extent of the geographical area that is referenced by the data.

It is important that you learn about the nature of coordinate systems and what spatial reference systems are.

How to use coordinate systems

When you plan a project that uses spatial data, you need to determine whether the data should be based on one of the coordinate systems that are registered to the Spatial Extender catalog.

If none of these coordinate systems meet your requirements, you can create one that does. This discussion explains the concept of coordinate systems and introduces the tasks of selecting one to use and creating a new one.

Coordinate systems

A coordinate system is a framework for defining the relative locations of things in a given area; for example, an area on the earth's surface or the earth's surface as a whole.

DB2 Spatial Extender supports the following types of coordinate systems to determine the location of a geographic feature:

Geographic coordinate system

A *geographic coordinate system* is a reference system that uses a three-dimensional spherical surface to determine locations on the earth. Any location on earth can be referenced by a point with latitude and longitude coordinates based on angular units of measure.

Projected coordinate system

A *projected coordinate system* is a flat, two-dimensional representation of the earth. It uses rectilinear (Cartesian) coordinates based on linear units of measure. It is based on a spherical (or spheroidal) earth model, and its coordinates are related to geographic coordinates by a projection transformation.

Geographic coordinate system

A *geographic coordinate system* is a that uses a three-dimensional spherical surface to determine locations on the earth. Any location on earth can be referenced by a point with longitude and latitude coordinates.

For example, Figure 6 on page 34 shows a geographic coordinate system where a location is represented by the coordinates longitude 80 degree East and latitude 55 degree North.

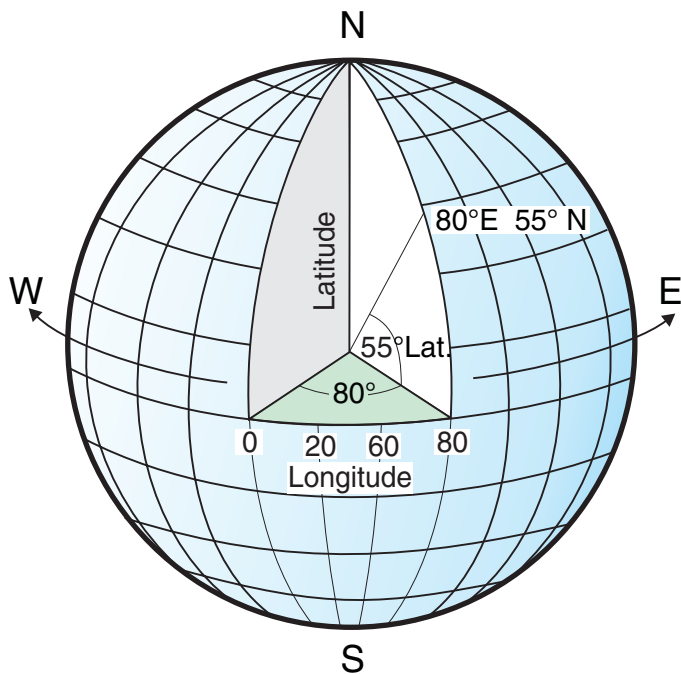


Figure 6. A geographic coordinate system

The lines that run east and west each have a constant latitude value and are called *parallels*. They are equidistant and parallel to one another, and form concentric circles around the earth. The *equator* is the largest circle and divides the earth in half. It is equal in distance from each of the poles, and the value of this latitude line is zero. Locations north of the equator have positive latitudes that range from 0 to +90 degrees, while locations south of the equator have negative latitudes that range from 0 to -90 degrees.

Figure 7 illustrates latitude lines.

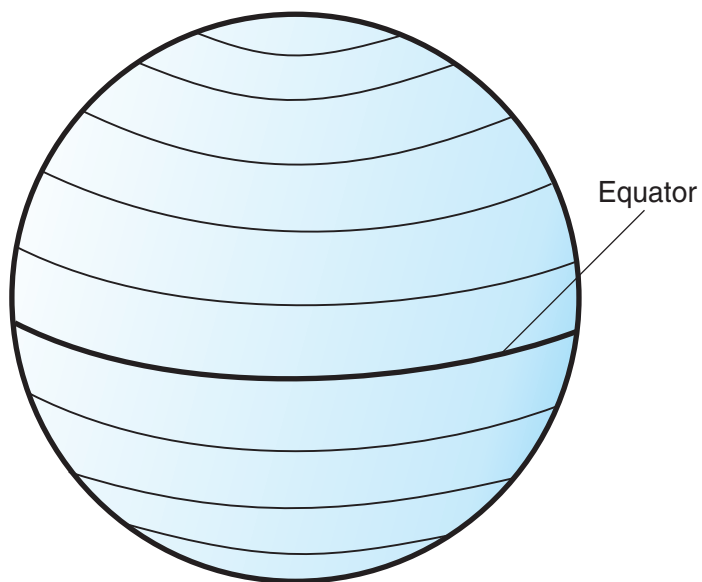


Figure 7. Latitude lines

The lines that run north and south each have a constant longitude value and are called *meridians*. They form circles of the same size around the earth, and intersect at the poles. The *prime meridian* is the line of longitude that defines the origin (zero degrees) for longitude coordinates. One of the most commonly used prime meridian locations is the line that passes through Greenwich, England. However, other longitude lines, such as those that pass through Bern, Bogota, and Paris, have also been used as the prime meridian. Locations east of the prime meridian up to its *antipodal* meridian (the continuation of the prime meridian on the other side of the globe) have positive longitudes ranging from 0 to +180 degrees. Locations west of the prime meridian have negative longitudes ranging from 0 to -180 degrees.

Figure 8 illustrates longitude lines.

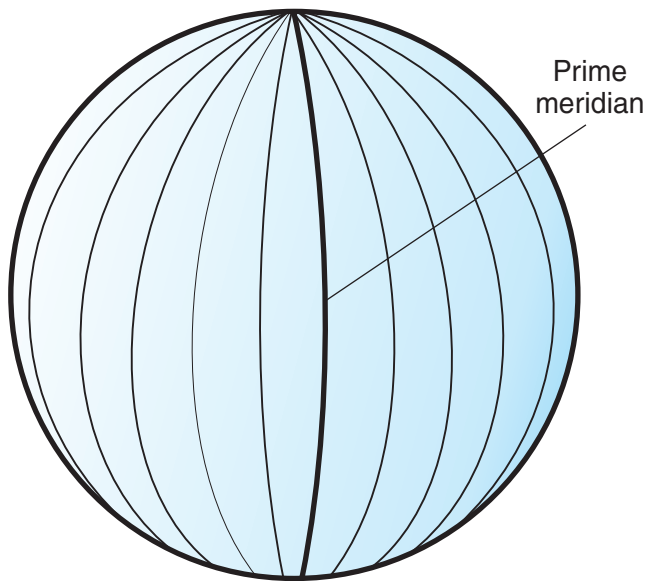


Figure 8. Longitude lines

The latitude and longitude lines can cover the globe to form a grid, called a *graticule*. The point of origin of the graticule is (0,0), where the equator and the prime meridian intersect. The equator is the only place on the graticule where the linear distance corresponding to one degree latitude is approximately equal the distance corresponding to one degree longitude. Because the longitude lines converge at the poles, the distance between two meridians is different at every parallel. Therefore, as you move closer to the poles, the distance corresponding to one degree latitude will be much greater than that corresponding to one degree longitude.

It is also difficult to determine the lengths of the latitude lines using the graticule. The latitude lines are concentric circles that become smaller near the poles. They form a single point at the poles where the meridians begin. At the equator, one degree of longitude is approximately 111.321 kilometers, while at 60 degrees of latitude, one degree of longitude is only 55.802 km (this approximation is based on the Clarke 1866 spheroid). Therefore, because there is no uniform length of degrees of latitude and longitude, the distance between points cannot be measured accurately by using angular units of measure.

Figure 9 on page 36 shows the different dimensions between locations on the graticule.

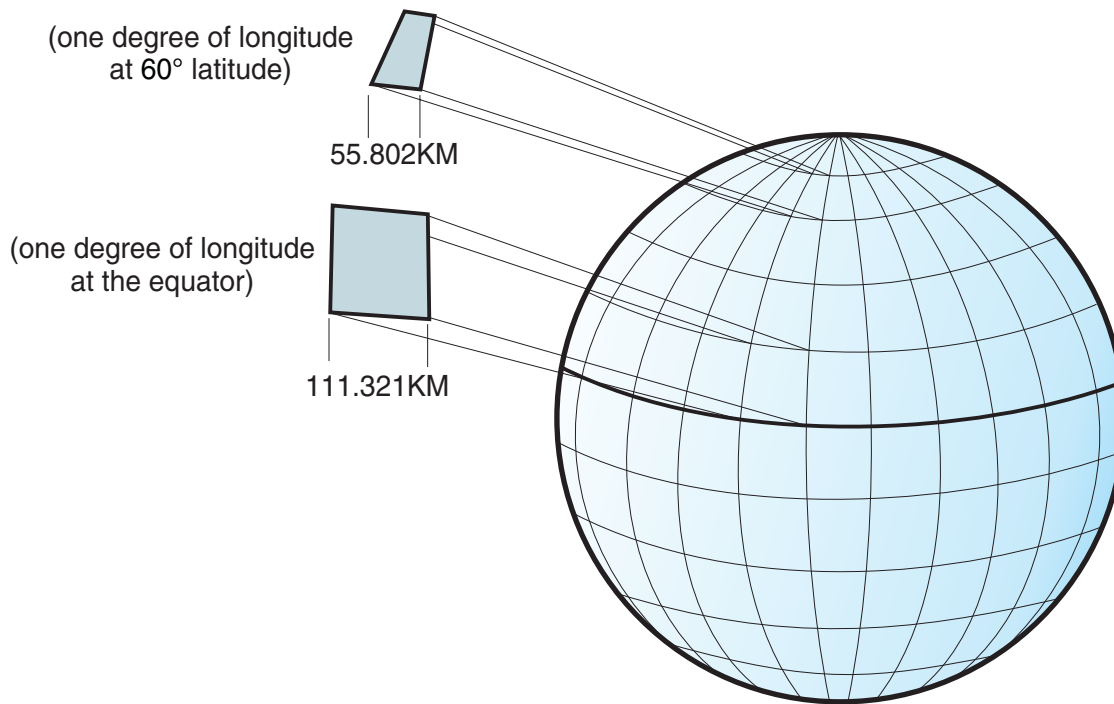
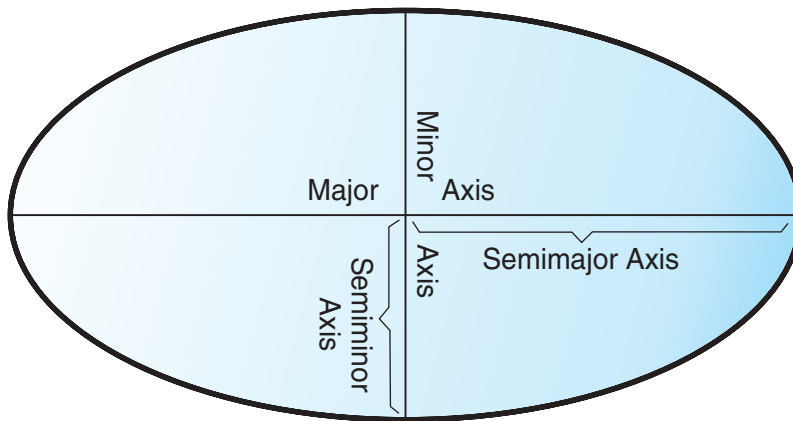
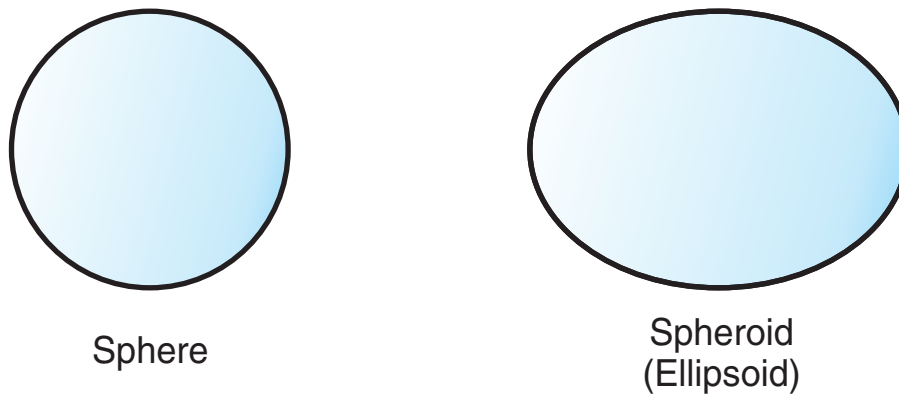


Figure 9. Different dimensions between locations on the graticule

A coordinate system can be defined by either a sphere or a spheroid approximation of the earth's shape. Because the earth is not perfectly round, a spheroid can help maintain accuracy for a map, depending on the location on the earth. A *spheroid* is an ellipsoid, that is based on an ellipse, whereas a sphere is based on a circle.

The shape of the ellipse is determined by two radii. The longer radius is called the semimajor axis, and the shorter radius is called the semiminor axis. An ellipsoid is a three-dimensional shape formed by rotating an ellipse around one of its axes.

Figure 10 on page 37 shows the sphere and spheroid approximations of the earth and the major and minor axes of an ellipse.



The major and minor axes of an ellipse

Figure 10. Sphere and spheroid approximations

A *datum* is a set of values that defines the position of the spheroid relative to the center of the earth. The datum provides a frame of reference for measuring locations and defines the origin and orientation of latitude and longitude lines. Some datums are global and intend to provide good average accuracy around the world. A local datum aligns its spheroid to closely fit the earth's surface in a particular area. Therefore, the coordinate system's measurements are not accurate if they are used with an area other than the one that they were designed.

Figure 11 on page 38 shows how different datums align with the earth's surface. The local datum, NAD27, more closely aligns with Earth's surface than the Earth-centered datum, WGS84, at this particular location.

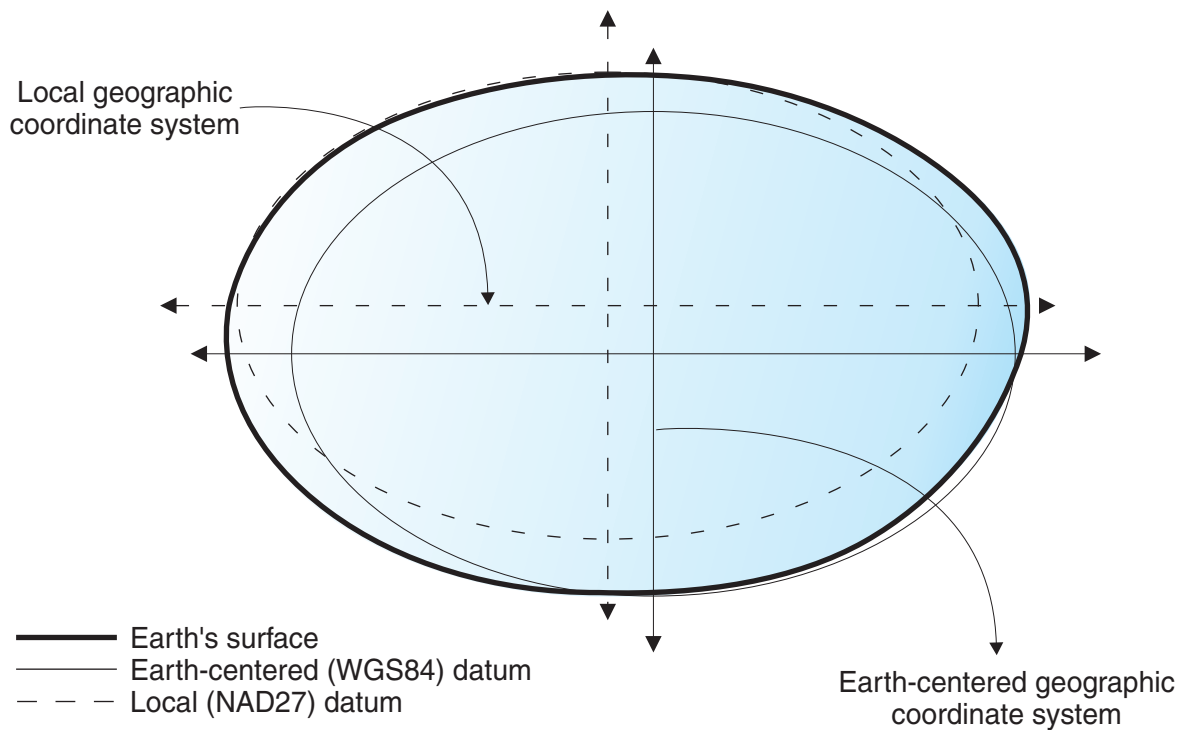


Figure 11. Datum alignments

Whenever you change the datum, the geographic coordinate system is altered and the coordinate values will change. For example, the coordinates in DMS of a control point in Redlands, California using the North American Datum of 1983 (NAD 1983) are: "-117 12 57.75961 34 01 43.77884". The coordinates of the same point on the North American Datum of 1927 (NAD 1927) are: "-117 12 54.61539 34 01 43.72995".

Projected coordinate systems

A *projected coordinate system* is a flat, two-dimensional representation of the Earth. It is based on a sphere or spheroid geographic coordinate system, but it uses linear units of measure for coordinates, so that calculations of distance and area are easily done in terms of those same units.

The latitude and longitude coordinates are converted to x, y coordinates on the flat projection. The x coordinate is usually the eastward direction of a point, and the y coordinate is usually the northward direction of a point. The center line that runs east and west is referred to as the x axis, and the center line that runs north and south is referred to as the y axis.

The intersection of the x and y axes is the origin and usually has a coordinate of (0,0). The values above the x axis are positive, and the values below the x axis are negative. The lines parallel to the x axis are equidistant from each other. The values to the right of the y axis are positive, and the values to the left of the y axis are negative. The lines parallel to the y axis are equidistant.

Mathematical formulas are used to convert a three-dimensional geographic coordinate system to a two-dimensional flat projected coordinate system. The transformation is referred to as a *map projection*. Map projections usually are classified by the projection surface used, such as conic, cylindrical, and planar surfaces. Depending on the projection used, different spatial properties will appear

distorted. Projections are designed to minimize the distortion of one or two of the data's characteristics, yet the distance, area, shape, direction, or a combination of these properties might not be accurate representations of the data that is being modeled. There are several types of projections available. While most map projections attempt to preserve some accuracy of the spatial properties, there are others that attempt to minimize overall distortion instead, such as the *Robinson* projection. The most common types of map projections include:

Equal area projections

These projections preserve the area of specific features. These projections distort shape, angle, and scale. The *Albers Equal Area Conic* projection is an example of an equal area projection.

Conformal projections

These projections preserve local shape for small areas. These projections preserve individual angles to describe spatial relationships by showing perpendicular graticule lines that intersect at 90 degree angles on the map. All of the angles are preserved; however, the area of the map is distorted. The *Mercator* and *Lambert Conformal Conic* projections are examples of conformal projections.

Equidistant projections

These projections preserve the distances between certain points by maintaining the scale of a given data set. Some of the distances will be true distances, which are the same distances at the same scale as the globe. If you go outside the data set, the scale will become more distorted. The *Sinusoidal* projection and the *Equidistant Conic* projection are examples of equidistant projections.

True-direction or azimuthal projections

These projections preserve the direction from one point to all other points by maintaining some of the great circle arcs. These projections give the directions or azimuths of all points on the map correctly with respect to the center. Azimuthal maps can be combined with equal area, conformal, and equidistant projections. The *Lambert Equal Area Azimuthal* projection and the *Azimuthal Equidistant* projection are examples of azimuthal projections.

Determine what coordinate system to use

A first step in planning a project is to determine what coordinate system to use.

Before you begin

Before you create a coordinate system, your user ID must have DBADM authority on the database that has been enabled for spatial operations. No authorization is required to use an existing coordinate system.

About this task

After you enable a database for spatial operations, you are ready to plan projects that use spatial data. You can use a coordinate system that was shipped with DB2 Spatial Extender or one that was created by elsewhere. Over 2000 coordinate systems are shipped with DB2 Spatial Extender.

To find out more about these coordinate systems, and to determine what other coordinate systems were shipped with DB2 Spatial Extender, and what (if any) coordinate systems have been created by other users, consult the

DB2SE.ST_COORDINATE_SYSTEMS catalog view.

Procedure

To determine what coordinate system to use:

1. Review existing coordinate systems that are shipped with DB2 Spatial Extender and use the corresponding spatial reference system that is based on the coordinate system of your choice.

The most used coordinate systems are:

- GCS_NORTH_AMERICAN_1983. Use this coordinate system when you need to define locations in the United States; for example:
 - When you import spatial data for the United States from the Spatial Map Data available for download.
- A coordinate system that DB2 Spatial Extender refers to as Unspecified. Use this coordinate system when:
 - You need to define locations that have no direct relationship to the earth's surface; for example, locations of offices within an office building or locations of shelves within a storage room.
 - You can define these locations in terms of positive coordinates that include few or no decimal values.

2. If you cannot find an existing coordinate systems in Spatial Extender, you can create one using one of the following methods:

- Issue the db2se create_cs command from the db2se command line processor.
- Run an application that invokes the DB2SE.ST_CREATE_COORDSYS procedure.

Creating a coordinate system is rarely necessary. You need to also create an spatial reference system based on the new coordinate system.

How to set up spatial reference systems

When you plan a project that uses spatial data, you need to determine whether any of the spatial reference systems available to you can be used for this data.

If none of the available systems are appropriate for the data, you can create one that is. This section explains the concept of spatial reference systems and describes the tasks of selecting which one to use and creating one.

Spatial reference systems

A spatial reference system is a set of parameters that is used to represent a geometry.

These parameters are:

- The name of the coordinate system from which the coordinates are derived.
- The numeric identifier that uniquely identifies the spatial reference system.
- Coordinates that define the maximum possible extent of space that is referenced by a given range of coordinates.
- Numbers that, when applied in certain mathematical operations, convert coordinates received as input into values that can be processed with maximum efficiency.

The following sections discuss the parameter values that define an identifier, a maximum extent of space, and conversion factors.

Spatial reference system identifier

The spatial reference system identifier (SRID) is used as an input parameter for various spatial functions.

Defining the space that encompasses coordinates stored in a spatial column

The coordinates in a spatial column typically define locations that span across part of the Earth. The space over which the span extends—from east to west and from north to south—is called a *spatial extent*. For example, consider a body of flood plains whose coordinates are stored in a spatial column. Suppose that the westernmost and easternmost of these coordinates are latitude values of -24.556 and -19.338, respectively, and that the northernmost and southernmost of the coordinates are longitude values of 18.819 and 15.809 degrees, respectively. The spatial extent of the flood plains is a space that extends on a west-east plane between the two latitudes and on a north-south plane between the two longitudes. You can include these values in a spatial reference system by assigning them to certain parameters. If the spatial column includes Z coordinates and measures, you would need to include the highest and lowest Z coordinates and measures in the spatial reference system as well.

The term spatial extent can refer not only to an actual span of locations, as in the previous paragraph; but also to a potential one. Suppose that the flood plains in the preceding example were expected to broaden over the next five years. You could estimate what the westernmost, easternmost, northernmost, and southernmost coordinates of the plains would be at the end of the fifth year. You could then assign these estimates, rather than the current coordinates, to the parameters for a spatial extent. That way, you could retain the spatial reference system as the plains expand and their wider latitudes and longitudes are added to the spatial column. Otherwise, if the spatial reference system is limited to the original latitudes and longitudes, it would need to be altered or replaced as the flood plains grew.

Converting to values that improve performance

Typically, most coordinates in a coordinate system are decimal values; some are integers. In addition, coordinates to the east of the origin are positive; those to the west are negative. Before being stored by Spatial Extender, the negative coordinates are converted to positive values, and the decimal coordinates are converted into integers. As a result, all coordinates are stored by Spatial Extender as positive integers. The purpose is to enhance performance when the coordinates are processed.

Certain parameters in a spatial reference system are used to make the conversions described in the preceding paragraph. One parameter, called an *offset*, is subtracted from each negative coordinate, which leaves a positive value as a remainder. Each decimal coordinate is multiplied by another parameter, called a *scale factor*, which results in an integer whose precision is the same as that of the decimal coordinate. (The offset is subtracted from positive coordinates as well as negative; and the nondecimal coordinates, as well as the decimal coordinates, are multiplied by the scale factor. This way, all positive and non-decimal coordinates remain commensurate with the negative and decimal ones.)

These conversions take place internally, and remain in effect only until coordinates are retrieved. Input and query results always contain coordinates in their original, unconverted form.

Deciding whether to use an existing reference system or create a new system

Answering the following series of questions can help you decide whether to use an existing spatial reference system or create a new system.

About this task

After you determine what coordinate system to use, you are ready to provide a spatial reference system that suits the coordinate data that you are working with. DB2 Spatial Extender provides five spatial reference systems for spatial data.

Procedure

To decide whether to use an existing reference system or create a new system:

1. Answer the following questions to determine whether you can use one of the existing spatial reference systems.
 - a. Does the coordinate system on which the existing spatial reference system is based cover the geographic area that you are working with? These coordinate systems are shown in “Spatial reference systems supplied with DB2 Spatial Extender.”
 - b. Do the conversion factors associated with one of the existing spatial reference systems work with your coordinate data?

Spatial Extender uses offset values and scale factors to convert the coordinate data that you provide to positive integers. To determine if your coordinate data works with the given offset values and scale factors for one of the exiting spatial reference systems:

 - 1) Review the information in “Conversion factors that transform coordinate data into integers” on page 44.
 - 2) Look at how these factors are defined for the exiting spatial reference systems. If, after applying the offset value to the minimum X and Y coordinates, these coordinates are not both greater than 0, you must create a new spatial reference system and define the offsets yourself. For more information about how to create a new spatial reference system, see “Creating a spatial reference system” on page 46.
 - c. Does the data that you are working with include height and depth coordinates (Z coordinates) or measures (M coordinates)? If you are working with Z or M coordinates, you might need to create a new spatial reference system with Z or M offsets and scale factors suitable to your data.
2. If the existing spatial reference systems do not work with your data, you need to “Creating a spatial reference system” on page 46.

What to do next

After you decide which spatial reference system you need, you specify this choice to Spatial Extender in functions or procedure calls.

Spatial reference systems supplied with DB2 Spatial Extender

The spatial reference system converts the coordinate data to positive integers.

DB2 Spatial Extender provides the spatial reference systems that are shown in the following table, along with the coordinate system on which each spatial reference system is based and the offset values and scale factors that DB2 Spatial Extender uses to convert the coordinate data to positive integers. You can find information about these spatial reference systems in the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view.

If you are working with decimal-degrees, the offset values and scale factors for the default spatial reference systems support the full range of latitude-longitude coordinates and preserve 6 decimal positions, equivalent to approximately 10 cm. The Sample Spatial Map data and Geocoder Reference data is in decimal-degrees.

If you plan to use the geocoder which works only with U.S. addresses, ensure that you select or create a spatial reference system that handles U.S. coordinates, such as the GCS_NORTH_AMERICAN_1983 coordinate system. If you do not specify what coordinate system your spatial data should derive from, Spatial Extender uses the DEFAULT_SRS spatial reference system.

If none of the default spatial reference systems meet your needs, you can create a new spatial reference system.

Table 1. Spatial reference systems provided with DB2 Spatial Extender

Spatial reference system	SRS ID	Coordinate system	Offset values	Scale factors	When to use
DEFAULT_SRS	0	None	xOffset = 0 yOffset = 0 zOffset = 0 mOffset = 0	xScale = 1 yScale = 1 zScale = 1 mScale = 1	You can select this system when your data is independent of a coordinate system or you cannot or do not need to specify one.
NAD83_SRS_1	1	GCS_NORTH_AMERICAN_1983	xOffset = -180 yOffset = -90 zOffset = 0 mOffset = 0	xScale = 1,000,000 yScale = 1,000,000 zScale = 1 mScale = 1	You can select this spatial reference system if you plan to use the U.S. sample data shipped with DB2 Spatial Extender. If the coordinate data that you are working with was collected after 1983, use this system instead of NAD27_SRS_1002.

Table 1. Spatial reference systems provided with DB2 Spatial Extender (continued)

Spatial reference system	SRS ID	Coordinate system	Offset values	Scale factors	When to use
NAD27_ SRS_1002	1002	GCS_NORTH _AMERICAN _1927	xOffset = -180 yOffset = -90 zOffset = 0 mOffset = 0	xScale = 5,965,232 yScale = 5,965,232 zScale = 1 mScale = 1	You can select this spatial reference system if you plan to use the U.S. sample data shipped with DB2 Spatial Extender. If the coordinate data that you are working with was collected before 1983, use this system instead of NAD83_SRS_1. This system provides a greater degree of precision than the other default spatial reference systems.
WGS84_ SRS_1003	1003	GCS_WGS _1984	xOffset = -180 yOffset = -90 zOffset = 0 mOffset = 0	xScale = 5,965,232 yScale = 5,965,232 zScale = 1 mScale = 1	You can select this spatial reference system if you are working with data outside the U.S. (This system handles worldwide coordinates.) Do not use this system if you plan to use the default geocoder shipped with DB2 Spatial Extender, because the geocoder is only for U.S. addresses.
DE_HDN _SRS_1004	1004	GCSW _DEUTSCHE _HAUPTDRE IECKSNETZ	xOffset = -180 yOffset = -90 zOffset = 0 mOffset = 0	xScale = 5,965,232 yScale = 5,965,232 zScale = 1 mScale = 1	This spatial reference system is based on a coordinate system for German addresses.

Conversion factors that transform coordinate data into integers

DB2 Spatial Extender uses offset values and scale factors to convert the coordinate data that you provide to positive integers. The default spatial reference systems already have offset value and scale factors associated with them. If you are

creating a new spatial reference system, determine the scale factors and, optionally, the offset values that work best with your data.

Offset values

An offset value is a number that is subtracted from all coordinates, leaving only positive values as a remainder.

Spatial Extender converts your coordinate data using the following formulas to ensure that all adjusted coordinate values are greater than 0.

Formula notation: In these formulas, the notation “min” represents “the minimum of all”. For example, “min(x)” means “the minimum of all x coordinates”. The offset for each geographic direction is represented as dimensionOffset. For example, xOffset is the offset value applied to all X coordinates.

$$\begin{aligned}\min(x) - \text{xOffset} &\geq 0 \\ \min(y) - \text{yOffset} &\geq 0 \\ \min(z) - \text{zOffset} &\geq 0 \\ \min(m) - \text{mOffset} &\geq 0\end{aligned}$$

Scale factors

A scale factor is a value that, when multiplied by decimal coordinates and measures, yields integers with at least the same number of significant digits as the original coordinates and measures.

Spatial Extender converts your decimal coordinate data using the following formulas to ensure that all adjusted coordinate values are positive integers. The converted values cannot exceed 2^{53} (approximately $9 * 10^{15}$).

Formula notation: In these formulas, the notation “max” represents “the maximum of all”. The offset for each geographic dimension is represented as dimensionOffset (for example, xOffset is the offset value applied to all X coordinates). The scale factor for each geographic dimension is represented as dimensionScale (for example, xScale is the scale factor applied to X coordinates).

$$\begin{aligned}(\max(x) - \text{xOffset}) * \text{xScale} &\leq 2^{53} \\ (\max(y) - \text{yOffset}) * \text{yScale} &\leq 2^{53} \\ (\max(z) - \text{zOffset}) * \text{zScale} &\leq 2^{53} \\ (\max(m) - \text{mOffset}) * \text{mScale} &\leq 2^{53}\end{aligned}$$

When you choose which scale factors work best with your coordinate data, ensure that:

- You use the same scale factor for X and Y coordinates.
- When multiplied by a decimal X coordinate or a decimal Y coordinate, the scale factor yields a value less than 2^{53} . One common technique is to make the scale factor a power of 10. That is, the scale factor should be 10 to the first power (10), 10 to the second power (100), 10 to the third power (1000), or, if necessary, a larger factor.
- The scale factor is large enough to ensure that the number of significant digits in the new integer is the same as the number of significant digits in the original decimal coordinate.

Example

Suppose that the ST_Point function is given input that consists of an X coordinate of 10.01, a Y coordinate of 20.03, and the identifier of a spatial reference system. When ST_Point is invoked, it multiplies the value of 10.01 and the value of 20.03 by the spatial reference system's scale factor for X and Y coordinates. If this scale

factor is 10, the resulting integers that Spatial Extender stores will be 100 and 200, respectively. Because the number of significant digits in these integers (3) is less than the number of significant digits in the coordinates (4), Spatial Extender will not be able to convert these integers back to the original coordinates, or to derive from them values that are consistent with the coordinate system to which these coordinates belong. But if the scale factor is 100, the resulting integers that DB2 Spatial Extender stores will be 1001 and 2003—values that can be converted back to the original coordinates or from which compatible coordinates can be derived.

Units for offset values and scale factors

Whether you use an existing spatial reference system or create a new one, the units for the offset values and scale factors will vary depending on the type of coordinate system that you are using.

For example, if you are using a geographic coordinate system, the values are in angular units such as decimal degrees; if you are using a projected coordinate system, the values are in linear units such as meters or feet.

Creating a spatial reference system

Create a new spatial reference system if none of the spatial reference systems that are provided with DB2 Spatial Extender work with your data.

Procedure

To create a spatial reference system:

1. Choose a spatial reference system ID (SRID) that is not already in use.
2. Decide on the degree of precision for the spatial reference system by using one of the following methods:
 - Indicating the extents of the geographical area that you are working with and the scale factors that you want to use with your coordinate data. Spatial Extender calculates the offset values.
 - Indicating both the offset values (required for Spatial Extender to convert negative values to positive values) and scale factors (required for Spatial Extender to convert decimal values to integers). Use this method when you must follow strict criteria for accuracy or precision.
3. Calculate the conversion information that Spatial Extender requires to convert coordinate data to positive integers. The information to calculate varies depending on the method that you chose in 2.
 - For the extents method, calculate the following information:
 - *Scale factors.* If any of the coordinates that you are working with are decimal values, calculate scale factors. Scale factors are numbers that, when multiplied by decimal coordinates and measures, yields integers with at least the same number of significant digits as the original coordinates and measures. If the coordinates are integers, the scale factors can be set to 1. If the coordinates are decimal values, set the scale factor to a number that converts the decimal portion to an integer value. For example, if the coordinate units are meters and the accuracy of the data is 1 cm, you would need a scale factor of 100.
 - *Minimum and maximum values* for your coordinates and measures.
 - For the offset method, calculate the following information:
 - *Offset values.* If your coordinate data includes negative numbers or measures, specify the offset values that you want to use. An offset is a number that is subtracted from all coordinates, leaving only positive

values as a remainder. If you are working with positive coordinates, set all offset values to 0. If you are not working with positive coordinates, select an offset that, when applied against the coordinate data, results in integers that are less than the largest positive integer value (9,007,199,254,740,992).

- *Scale factors.* If any of the coordinates for the locations that you are representing are decimal numbers, determine what scale factors to use and enter these scale factors in the Create Spatial Reference System window.

4. Issue the **db2se create_srs** command or call the DB2SE.ST_CREATE_SRS procedure to create the spatial reference system.

The following example shows how to create a spatial reference system called mysrs:

```
db2se create_srs mydb -srsName mysrs -srsID 100
-xScale 10 -coordsysName GCS_North_American_1983
```

Calculating scale factors

If you create a spatial reference system and any of the coordinates that you are working with are decimal values, calculate the appropriate scale factors for your coordinates and measures. Scale factors are numbers that, when multiplied by decimal coordinates and measures, yields integers with at least the same number of significant digits as the original coordinates and measures.

About this task

After you calculate scale factors, you need to determine the extent values. Then submit the **db2se create_srs** command or call the DB2SE.ST_CREATE_SRS procedure.

Procedure

To calculate the scale factors:

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. Find the decimal coordinate that has the longest decimal precision. Then determine by what power of 10 this coordinate can be multiplied to yield an integer of equal precision. For example, of the three decimal coordinates in the current example, 5.1235 has the longest decimal precision. Multiplying it by 10 to the fourth power (10000) yields the integer 51235.
3. Determine whether the integer produced by the multiplication just described is less than 2^{53} . 51235 is not too large. 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, 10000000006.789876. Because this coordinate's decimal precision is longer than that of the other three, you would multiply this coordinate-not 5.1235-by a power of 10. To convert it to an integer, you could multiply it by 10 to the sixth power (1000000). But the resulting value, 10000000006789876, is greater than 2^{53} . If DB2 Spatial Extender tried to store it, the results would be unpredictable.

To avoid this problem, select a power 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 fifth power (100000). Multiplying 100000 by 10000000006.789876 yields 1000000000678987.6. DB2 Spatial Extender would round this number to 1000000000678988, reducing its accuracy slightly.

Conversion factors that transform coordinate data into integers

DB2 Spatial Extender uses offset values and scale factors to convert the coordinate data that you provide to positive integers. The default spatial reference systems already have offset value and scale factors associated with them. If you are creating a new spatial reference system, determine the scale factors and, optionally, the offset values that work best with your data.

Determining minimum and maximum coordinates and measures

Determine minimum and maximum coordinates and measures if you specify extent transformations when you create a spatial reference system.

About this task

After you determine the extent values, if any of the coordinates are decimal values, you need to calculate scale factors. Otherwise, submit the **db2se create_srs** command or call the DB2SE.ST_CREATE_SRS procedure.

Procedure

To determine the minimum and maximum coordinates and measures of the locations that you want to represent:

1. Determine the minimum and maximum X coordinates. To find the minimum X coordinate, identify the X coordinate in your domain that is furthest west. (If the location lies to the west of the point of origin, this coordinate will be a negative value.) To find the maximum X coordinate, identify the X coordinate in your domain that is furthest east. For example, if you are representing oil wells, and each one is defined by a pair of X and Y coordinates, the X coordinate that indicates the location of the oil well that is furthest west is the minimum X coordinate, and the X coordinate that indicates the location of the oil well that is furthest east is the maximum X coordinate.

Tip: For multifeature types, such as multipolygons, ensure that you pick the furthest point on the furthest polygon in the direction that you are calculating. For example, if you are trying to identify the minimum X coordinate, identify the westernmost X coordinate of the polygon that is furthest west in the multipolygon.

2. Determine the minimum and maximum Y coordinates. To find the minimum Y coordinate, identify the Y coordinate in your domain that is furthest south. (If the location lies to the south of the point of origin, this coordinate will be a negative value.) To determine the maximum Y coordinate, find the Y coordinate in your domain that is furthest north.
3. Determine the minimum and maximum Z coordinates. The minimum Z coordinate is the greatest of the depth coordinates and the maximum Z coordinate is the greatest of the height coordinates.
4. Determine the minimum and maximum measures. If you are going to include measures in your spatial data, determine which measure has the highest numeric value and which has the lowest.

Calculating offset values

Specify offset values if your coordinate data includes negative numbers or measures. An offset is a number that is subtracted from all coordinates, leaving only positive values as a remainder.

About this task

If you create a spatial reference system and your coordinate data includes negative numbers or measures, you must specify the offset values that you want to use. You can improve the performance of spatial operations when the coordinates are positive integers instead of negative numbers or measures.

Procedure

To calculate the offset values for the coordinates that you are working with:

1. Determine the lowest negative X, Y, and Z coordinates within the range of coordinates for the locations that you want to represent. If your data is to include negative measures, determine the lowest of these measures.
2. Optional but recommended: Indicate to DB2 Spatial Extender that the domain that encompasses the locations that you are concerned with is larger than it actually is. Thus, after you write data about these locations to a spatial column, you can add data about locations of new features as they are added to outer reaches of the domain, without having to replace your spatial reference system with another one.

For each coordinate and measure that you identified in step 1, add an amount equal to five to ten percent of the coordinate or measure. The result is referred to as an *augmented value*. For example, if the lowest negative X coordinate is -100, you could add -5 to it, yielding an augmented value of -105. Later, when you create the spatial reference system, you will indicate that the lowest X coordinate is -105, rather than the true value of -100. DB2 Spatial Extender will then interpret -105 as the westernmost limit of your domain.

3. Find a value that, when subtracted from your augmented X value, leaves zero; this is the offset value for X coordinates. DB2 Spatial Extender subtracts this number from all X coordinates to produce only positive values.

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.

4. Repeat step 3 for the augmented Y value, augmented Z value, and augmented measure.

Creating a spatial reference system

Create a new spatial reference system if none of the spatial reference systems that are provided with DB2 Spatial Extender work with your data.

Procedure

To create a spatial reference system:

1. Choose a spatial reference system ID (SRID) that is not already in use.
2. Decide on the degree of precision for the spatial reference system by using one of the following methods:

- Indicating the extents of the geographical area that you are working with and the scale factors that you want to use with your coordinate data. Spatial Extender calculates the offset values.
 - Indicating both the offset values (required for Spatial Extender to convert negative values to positive values) and scale factors (required for Spatial Extender to convert decimal values to integers). Use this method when you must follow strict criteria for accuracy or precision.
3. Calculate the conversion information that Spatial Extender requires to convert coordinate data to positive integers. The information to calculate varies depending on the method that you chose in 2 on page 46.
 - For the extents method, calculate the following information:
 - *Scale factors.* If any of the coordinates that you are working with are decimal values, calculate scale factors. Scale factors are numbers that, when multiplied by decimal coordinates and measures, yields integers with at least the same number of significant digits as the original coordinates and measures. If the coordinates are integers, the scale factors can be set to 1. If the coordinates are decimal values, set the scale factor to a number that converts the decimal portion to an integer value. For example, if the coordinate units are meters and the accuracy of the data is 1 cm, you would need a scale factor of 100.
 - *Minimum and maximum values* for your coordinates and measures.
 - For the offset method, calculate the following information:
 - *Offset values.* If your coordinate data includes negative numbers or measures, specify the offset values that you want to use. An offset is a number that is subtracted from all coordinates, leaving only positive values as a remainder. If you are working with positive coordinates, set all offset values to 0. If you are not working with positive coordinates, select an offset that, when applied against the coordinate data, results in integers that are less than the largest positive integer value (9,007,199,254,740,992).
 - *Scale factors.* If any of the coordinates for the locations that you are representing are decimal numbers, determine what scale factors to use and enter these scale factors in the Create Spatial Reference System window.
 4. Issue the **db2se create_srs** command or call the DB2SE.ST_CREATE_SRS procedure to create the spatial reference system.

The following example shows how to create a spatial reference system called mysrs:

```
db2se create_srs mydb -srsName mysrs -srsID 100
                  -xScale 10 -coordsysName GCS_North_American_1983
```

Chapter 8. Setting up spatial columns

In preparing to obtain spatial data for a project, you should choose a coordinate system and spatial reference system and then provide one or more table columns to contain the data.

Spatial columns visualization

When you use a visualization tool, such as ArcExplorer for DB2, to query a spatial column, the tool returns results in the form of a graphical display; for example, a map of parcel boundaries or the layout of a road system.

Some visualization tools require all rows of the column to use the same spatial reference system. The way you enforce this constraint is by registering the column with a spatial reference system.

Spatial data types

When you enable a database for spatial operations, DB2 Spatial Extender supplies the database with a hierarchy of structured data types.

Figure 12 presents this hierarchy. In this figure, the instantiable types have a white background; the uninstantiable types have a shaded background.

Instantiable data types are ST_Point, ST_LineString, ST_Polygon, ST_GeomCollection, ST_MultiPoint, ST_MultiPolygon, and ST_MultiLineString.

Data types that are not instantiable are ST_Geometry, ST_Curve, ST_Surface, ST_MultiSurface, ST_MultiCurve, and ST_MultiLineString.

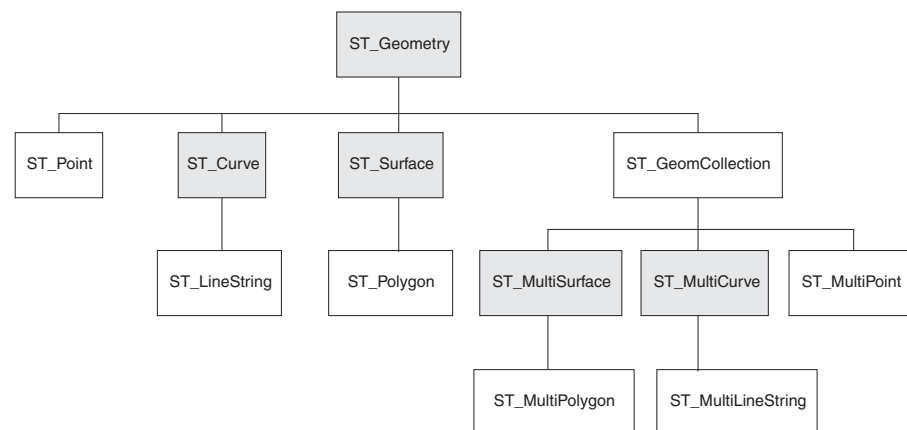


Figure 12. Hierarchy of spatial data types. Data types named in white boxes are instantiable. Data types named in shaded boxes are not instantiable.

The hierarchy in Figure 12 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, canal systems and groups of islands in a lake.

- 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 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 voting district, a forest, or a wildlife habitat. An `ST_Polygon` data item consists of the coordinates that define the boundary 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 an apartment complex. If you want to represent the point in space where each building in the complex is located, you would use `ST_Point` to store the X and Y coordinates that define each such point. Otherwise, if you want to represent the area occupied by the complex as a whole, you would use `ST_Polygon` to store the coordinates that define the boundary of this 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 are representing features made up of units whose locations are each referenced by an X coordinate and a Y coordinate. For example, consider a table whose rows represent island chains. The X coordinate and Y coordinate for each island has been identified. If you want the table to include these coordinates and the coordinates for each chain as a whole, define an `ST_MultiPoint` column to hold these coordinates.
- Use `ST_MultiLineString` when you are representing features made up of linear units, and you want to store the coordinates for the locations of these units and the location of each feature as a whole. For example, consider a table whose rows represent river systems. If you want the table to include coordinates for the locations of the systems and their components, define an `ST_MultiLineString` column to hold these coordinates.
- Use `ST_MultiPolygon` when you are representing features made up of multi-sided units, and you want to store the coordinates for the locations of these units and the location of each feature as a whole. For example, consider a table whose rows represent rural counties and the farms in each county. If you want the table to include coordinates for the locations of the counties and farms, define an `ST_MultiPolygon` column to hold these coordinates.

Multi-unit is not meant as a collection of individual entities. Rather, multi-unit refers to an aggregate of the parts that makes up the whole.

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 contain the same kinds of data items that columns of the other data types can contain.

Attention: Certain visualization tools do not support ST_Geometry columns, but only columns to which a proper subtype of ST_Geometry has been assigned.

Creating spatial columns

You must create spatial columns to store and retrieve spatial data.

Before you begin

Before you create a spatial column, your user ID must hold the authorizations that are needed for the DB2 SQL CREATE TABLE or ALTER TABLE statement. The user ID must have at least one of the following authorities or privileges:

- DBADM authority on the database where the table that has the column resides
- CREATETAB authority on the database and USE privilege on the table space as well as one of the following:
 - IMPLICIT_SCHEMA authority on the database, if the schema of the index does not exist
 - CREATEIN privilege on the schema, if the schema name of the index refers to an existing schema
- ALTER privilege on the table to be altered
- CONTROL privilege on the table to be altered
- ALTERIN privilege on the schema of the table to be altered

About this task

This task is part of a larger task "Setting up spatial resources for a project." After you choose a coordinate system and determine which spatial reference system to use for your data, you create a spatial column in an existing table or import spatial data into a new table.

Procedure

To create spatial columns:

You can provide your database with spatial columns in one of several ways:

- Use the CREATE TABLE SQL statement to create a table and to include a spatial column within that table.
- Use the ALTER TABLE SQL statement to add a spatial column to an existing table.
- If you are importing spatial data from a shape file, use DB2 Spatial Extender to create a table and to provide this table with a column to hold the data. See "Importing shape data to a new or existing table."

What to do next

The next task in setting up spatial resources is “Registering spatial columns.”

Registering spatial columns

Registering a spatial column creates a constraint on the table, if possible, to ensure that all geometries use the specified spatial reference system.

Before you begin

If you are using the db2se command line processor or an application program, your user ID must hold SYSADM or DBADM authority on the database.

About this task

You might want to register a spatial column in the following situations:

- Access by visualization tools

If you want certain visualization tools—for example, ArcExplorer for DB2—to generate graphical displays of the data in a spatial column, you need to ensure the integrity of the column's data. You do this by imposing a constraint that requires all rows of the column to use the same spatial reference system. To impose this constraint, register the column, specifying both its name and the spatial reference system that applies to it.

- Access by spatial indexes

Use the same coordinate system for all data in a spatial column on which you want to create an index to ensure that the spatial index returns the correct results. You register a spatial column to constrain all data to use the same spatial reference system and, correspondingly, the same coordinate system.

Procedure

Register a spatial column by using one of the following methods:

- Issue the db2se register_spatial_column command.
- Run an application that invokes the DB2GSE.ST_REGISTER_SPATIAL_COLUMN procedure.

Refer to the SRS_NAME column in the DB2GSE.ST_GEOMETRY_COLUMNS view to check the spatial reference system you chose for a particular column after you register the column.

Chapter 9. Populating spatial columns

After you create spatial columns and register the ones to be accessed by these visualization tools, you can populate the columns with spatial data by importing it, by using a geocoder to derive it from business data; or by using spatial functions to create it or to derive it from business data or other spatial data.

About importing and exporting spatial data

You can use DB2 Spatial Extender to exchange spatial data between your database and external data sources. More precisely, you can import spatial data from external sources by transferring it to your database in files, called data exchange files. You also can export spatial data from your database to data exchange files, from which external sources can acquire it. This section suggests some of the reasons for importing and exporting spatial data, and describes the nature of the data exchange files that DB2 Spatial Extender supports.

Reasons for importing and exporting spatial data

By importing spatial data, you can obtain a great deal of spatial information that is already available in the industry. By exporting it, you can make it available in a standard file format to existing applications. Consider these scenarios:

- Your database contains spatial data that represents your sales 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 environment. You proceed by using an Oracle utility to write the data to 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 are not connected to DB2, and want to use a geobrowser 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 to render the data in visual form.

Shape files

DB2 Spatial Extender supports shape files for data exchange. The term shape file actually refers to a collection of files with the same file name but different file extensions. The collection can include up to four files. They are:

- A file that contains spatial data in shape format, a de facto industry-standard format developed by ESRI. Such data is often called shape data. The extension of a file containing shape data is .shp.
- A file that contains business data that pertains to locations defined by shape data. This file's extension is .dbf.
- A file that contains an index to shape data. This file's extension is .shx.
- A file that contains a specification of the coordinate system on which the data in a .shp file is based. This file's extension is .prj.

Shape files are often used for importing data that originates in file systems, and for exporting data to files within file systems.

When you use DB2 Spatial Extender to import shape data, you receive at least one .shp file. In most cases, you receive one or more of the other three kinds of shape files as well.

Importing shape data to a new or existing table

You can import shape data to an existing table or view, or you can create a table and import shape data to it in a single operation.

Before you begin

Before you import shape data to an existing table or view, your user ID must hold one of the following authorities or privileges:

- DATAACCESS authority on the database that contains the table or view
- CONTROL privilege on the table or view
- INSERT privilege on the table or view
- SELECT privilege on the table or view (required only if the table includes an ID column that is not an IDENTITY column)

Plus one of the following privileges:

- Privileges to access the directories to which input files and error files belong
- Read privileges on the input files and write privileges on the error files

Before you begin to create a table automatically and import shape data to it, your user ID must hold the following authorities or privileges:

- DBADM authority on the database that will contain the table
- CREATETAB authority on the database that will contain the table
- If the schema already exists, CREATEIN privilege on the schema to which the table belongs
- If the schema specified for the table does not exist, IMPLICIT_SCHEMA authority on the database that contains the table

Plus one of the following privileges:

- Privileges to access the directories to which input files and error files belong
- Read privileges on the input files and write privileges on the error files

About this task

Choose one of the following methods to import shape data:

- Import the shape data to a spatial column in an existing table, an existing updateable view, or an existing view on which an INSTEAD OF trigger for INSERTs is defined.
- Automatically create a table with a spatial column and import the shape data to this column.

Procedure

To import shape data to a new or existing table:

Choose the method that you want to use to import shape data:

- Issue the `db2se import_shape` command.
- Run an application that calls the `DB2GSE.ST_IMPORT_SHAPE` procedure.

Exporting data to a shapefile

You can export spatial data returned in query results to a shapefile.

Before you begin

Before you can export data to a shapefile, your user ID must hold the following privileges:

- The privilege to execute a subselect that returns the results that you want to export
- The privilege to write to the directory where the file to which you will be exporting data resides
- The privilege to create a file to contain the exported data (required if such a file does not already exist)

To find out what these privileges are and how to obtain them, consult your database administrator.

About this task

The spatial data that you export to a shapefile might come from sources such as a base table, a join or union of multiple tables, result sets returned when you query views, or output of a spatial function.

If a file to which you want to export data exists, DB2 Spatial Extender can append the data to this file. If such a file does not exist, you can use DB2 Spatial Extender to create one.

Procedure

To export data to a shapefile:

Choose a method to export data to a shapefile:

- Issue the `db2se export_shape` command from the `db2se` command line processor.
- Run an application that calls the `DB2GSE.ST_EXPORT_SHAPE` procedure.

How to use a geocoder

Using a geocoder involves defining the work that you want a geocoder to do, setting up the geocoder, and running the geocoder in batch mode.

Geocoders and geocoding

The terms geocoder and geocoding are used in several contexts. This discussion sorts out these contexts, so that the terms' meanings can be clear each time you come across the terms. The discussion defines geocoder and geocoding, describes

the modes in which a geocoder operates, describes a larger activity to which geocoding belongs, and summarizes users' tasks that pertain to geocoding.

In DB2 Spatial Extender, a geocoder is a scalar function that translates existing data (the function's input) into data that you can understand in spatial terms (the function's output). Typically, the existing data is relational data that describes or names a location. DB2 Spatial Extender can support vendor-supplied and user-supplied geocoders.

One vendor-supplied geocoder might translate addresses into coordinates that DB2 does not store, but rather writes to a file. Another might be able to translate the number of an office in a commercial building into coordinates that define office's location in the building, or to translate the identifier of a shelf in a warehouse into coordinates that define the shelf's location in the warehouse.

In other cases, the existing data that a geocoder translates might be spatial data. For example, a user-supplied geocoder might translate X and Y coordinates into data that conforms to one of DB2 Spatial Extender's data types.

In DB2 Spatial Extender, geocoding is simply the operation in which a geocoder translates its input into output-translating addresses into coordinates, for example.

Modes

A geocoder operates in two modes:

- In batch mode, a geocoder attempts, in a single operation, to translate all its input from a single table (or, alternatively, all addresses in a specified subset of rows in the table).
- In automatic mode, a geocoder translates data as soon as it is inserted or updated in a table. The geocoder is activated by INSERT and UPDATE triggers that are defined on the table.

Geocoding processes

Geocoding is one of several operations by which the contents of a spatial column in a DB2 table are derived from other data. This discussion refers to these operations collectively as a geocoding process. Geocoding processes can vary from geocoder to geocoder. A geocoder may search files of known addresses to determine whether each address it receives as input matches a known address to a given degree. Because the known addresses are like reference material that people look up when they do research, these addresses are collectively called reference data. Other geocoders might not need reference data; they might verify their input in other ways.

The user's tasks

In DB2 Spatial Extender, the tasks that pertain to geocoding are:

- Prescribing how certain parts of the geocoding process should be executed for a given spatial column; for example, setting the minimum degree to which street names in input records and street names in reference data should match; setting the minimum degree to which addresses in input records and addresses in reference data should match; and determining how many records should be processed before each commit. This task can be referred to as setting up geocoding or setting up geocoding operations.

- Specifying that data should be automatically geocoded each time that it is added to, or updated in, a table. When automatic geocoding occurs, the instructions that the user specified when he or she set up geocoding operations will take effect (except for the instructions involving commits; they apply only to batch geocoding). This task is referred to as setting up a geocoder to run automatically.
- Running a geocoder in batch mode. If the user has set up geocoding operations already, his or her instructions will remain in effect during each batch session, unless the user overrides them. If the user has not set up geocoding operations before a given session, the user can specify that they should take effect set them up for that particular session. This task can be referred to as running a geocoder in batch mode and running geocoding in batch mode.

Setting up geocoding operations

DB2 Spatial Extender lets you set, in advance, the work that must be done when a geocoder is invoked.

Before you begin

Before you can set geocoding operations for a particular geocoder, your user ID must hold one of the following authorities or privileges:

- DATAACCESS authority on the database that contains the tables that the geocoder will operate on
- CONTROL privilege on each table that the geocoder operates on
- SELECT privilege and UPDATE privilege on each table that the geocoder operates on

About this task

You can specify the following parameters when a geocoder is invoked:

- What column the geocoder is to provide data for.
- Whether the input that the geocoder reads from a table or view should be limited to a subset of rows in the table or view.
- The range or number of records that the geocoder should geocode in batch sessions within a unit of work.
- Requirements for geocoder-specific operations. For example, a geocoder may geocode only those records that match their counterparts in the reference data to a specified degree or higher. This degree is called the minimum match score.

You must specify the parameters in the preceding list before you set up the geocoder to run in automatic mode. From then on, each time the geocoder is invoked (not only automatically, but also for batch runs), geocoding operations will be performed in accordance with your specifications. For example, if you specify that 45 records should be geocoded in batch mode within each unit of work, a commit will be issued after every forty-fifth record is geocoded. (Exception: You can override your specifications for individual sessions of batch geocoding.)

You do not have to establish defaults for geocoding operations before you run the geocoder in batch mode. Rather, at the time that you initiate a batch session, you can specify how the operations are to be performed for the length of the run. If you do establish defaults for batch sessions, you can override them, as needed, for individual sessions.

Procedure

To set up geocoding operations:

Choose which way you want to set up geocoding operations:

- Issue the `db2se setup_gc` command.
- Run an application that calls the `DB2GSE.ST_SETUP_GEOCODING` procedure.

What to do next

Recommendations: When a geocoder reads a record of address data, it tries to match that record with a counterpart in the reference data. In broad outline, the way it proceeds is as follows: First, it searches the reference data for streets whose zip code is the same as the zip code in the record. If it finds a street name that is similar to the one in the record to a certain minimum degree, or to a degree higher than this minimum, it goes on to look for an entire address. If it finds an entire address that is similar to the one in the record to a certain minimum degree, or to a degree higher than this minimum, it geocodes the record. If it does not find such an address, it returns a null.

The minimum degree to which the street names must match is referred to as spelling sensitivity. The minimum degree to which the entire addresses must match is called the minimum match score. For example, if the spelling sensitivity is 80, then the match between the street names must be at least 80 percent accurate before the geocoder will search for the entire address. If the minimum match score is 60, then the match between the addresses must be at least 60 percent accurate before the geocoder will geocode the record.

You can specify what the spelling sensitivity and minimum match score should be. Be aware that you might need to adjust them. For example, suppose that the spelling sensitivity and minimum match score are both 95. If the addresses that you want geocoded have not been carefully validated, matches of 95 percent accuracy are highly unlikely. As a result, the geocoder is likely to return a null when it processes these records. In such a case, it is advisable to lower the spelling sensitivity and minimum match score, and run the geocoder again. Recommended scores for spelling sensitivity and the minimum match score are 70 and 60, respectively.

As noted at the start of this discussion, you can determine whether the input that the geocoder reads from a table or view should be limited to a subset of rows in the table or view. For example, consider the following scenarios :

- You invoke the geocoder to geocode addresses in a table in batch mode. Unfortunately, the minimum match score is too high, causing the geocoder to return a null when it processes most of the addresses. You reduce the minimum match score when you run the geocoder again. To limit its input to those addresses that were not geocoded, you specify that it should select only those rows that contain the null that it had returned earlier.
- The geocoder selects only rows that were added after a certain date.
- The geocoder selects only rows that contain addresses in a particular area; for example, a block of counties or a state.

As noted at the start of this discussion, you can determine the number of records that the geocoder should process in batch sessions within a unit of work. You can have the geocoder process the same number of records in each unit of work, or

you can have it process all the records of a table within a single unit of work. If you choose the latter alternative, be aware that:

- You have less control over the size of the unit of work than the former alternative affords. Consequently, you cannot control how many locks are held or how many log entries are made as the geocoder operates.
- If the geocoder encounters an error that necessitates a rollback, you need to run the geocoder to run against all the records again. The resulting cost in resources can be expensive if the table is extremely large and the error and rollback occur after most records have been processed.

Setting up a geocoder to run automatically

You can set up a geocoder to automatically translate data as soon as the data is added to, or updated in, a table.

Before you begin

Before you can set up a geocoder to run automatically:

- You must set up geocoding operations for each spatial column that is to be populated by output from the geocoder.
- Your user ID must hold the following authorities or privileges:
 - DBADM and DATAACCESS authority on the database that contains the table on which triggers to invoke the geocoder will be defined
 - CONTROL privilege
 - ALTER, SELECT, and UPDATE privileges.
 - The privileges required to create triggers on this table.

About this task

You can set up a geocoder to run automatically before you invoke it in batch mode. Therefore, it is possible for automatic geocoding to precede batch geocoding. If that happens, the batch geocoding is likely to involve processing the same data that was processed automatically. This redundancy will not result in duplicate data, because when spatial data is produced twice, the second yield of data overrides the first. However, it can degrade performance.

Before you decide whether to geocode the address data within a table in batch mode or automatic mode, consider that:

- Performance is better in batch geocoding than in automatic geocoding. A batch session opens with one initialization and ends with one cleanup. In automatic geocoding, each data item is geocoded in a single operation that begins with initialization and concludes with cleanup.
- On the whole, a spatial column populated by means of automatic geocoding is likely to be more up to date than a spatial column populated by means of batch geocoding. After a batch session, address data can accumulate and remain ungeocoded until the next session. But if automatic geocoding is already enabled, address data is geocoded as soon as it is stored in the database.

Procedure

To set up a geocoder to run automatically:

Choose which method to use to set up automatic geocoding:

- Issue the `db2se enable_autogc` command.

- Run an application that calls the DB2GSE.ST_ENABLE_AUTOGEOCODING procedure.

Running a geocoder in batch mode

When you run a geocoder in batch mode, you translate multiple records into spatial data that goes into a specific column.

Before you begin

Before you can run a geocoder in batch mode, your user ID must hold one of the following authorities or privileges:

- DATAACCESS authority on the database that contains the table whose data is to be geocoded
- CONTROL privilege on this table
- UPDATE privilege on this table

You also need the SELECT privilege on this table, so that you can specify the number of records to be processed before each commit. If you specify WHERE clauses to limit the rows on which the geocoder is to operate, you might also require the SELECT privilege on any tables and views that you reference in these clauses. Ask your database administrator.

About this task

At any time before you run a geocoder to populate a particular spatial column, you can set up geocoding operations for that column. Setting up the operations involves specifying how certain requirements are to be met when the geocoder is run. For example, suppose that you require DB2 Spatial Extender to issue a commit after every 100 input records are processed by the geocoder. When you set up the operations, you would specify 100 as the required number.

When you are ready to run the geocoder, you can override any of the values that you specified when you set up operations. Your overrides will remain in effect only for the length of the run.

If you do not set up operations, you must, each time you are ready to run the geocoder, specify how the requirements are to be met during the run.

Procedure

To run a geocoder in batch mode:

Choose how to invoke a geocoder to run in batch mode:

- Issue the db2se run_gc command.
- Run an application that calls the DB2GSE.ST_RUN_GEOCODING procedure.

Chapter 10. DB2 Spatial Extender in a partitioned database environment

The DB2 Spatial Extender can be used effectively in a partitioned database environment to perform spatial analysis against large customer data tables. Partitioned database environments are useful for running data warehousing applications that you can create with IBM InfoSphere® Warehouse.

Partitioned database environments support high-performance and highly scalable database processing by parallel processing across a set of nodes. Each node has its own processor, memory, and disk storage.

In a partitioned database environment, you have three alternatives for storing DB2 tables that contain spatial columns. You can place them on a single node, distribute them across multiple nodes or replicate them across multiple nodes. Spatial indexes are supported for each alternative. The alternative that you select depends on the size of the tables and how they will be used in spatial queries. For more information about how to partition a database, see “Partitioned database environments” in *Database Administration Concepts and Configuration Reference*.

Creating and loading spatial data in a partitioned database environment

The partitioned table must have a partitioning key defined which is used to control the distribution of rows among the partitions.

Before you begin

Ensure that the database is enabled for spatial processing by creating spatial catalog tables and creating the spatial types and functions in the database. See “Enabling a database for spatial operations” on page 29 for more information.

About this task

For optimum performance, ensure that the tables that contain spatial columns are distributed evenly across partitions. The default hashing algorithm and partition map do this.

Procedure

For best results, specify one or more columns in the CREATE TABLE statement as the partitioning key. If you do not specify a partitioning key, DB2 selects the first numeric or character column as the partitioning key by default. This default might not distribute the rows evenly across partitions.

Example

The following example shows how to create a table to contain spatial data and use the DB2 Spatial Extender import utility to specify the partitioning key:

```
CREATE TABLE myschema.counties (id INTEGER PRIMARY KEY,  
    name VARCHAR(20),  
    geom db2gse.st_polygon)  
IN nodestbs
```

```

DISTRIBUTE BY HASH(id);

db2se import_shape mydb
-tableName counties
-tableSchema myschema
-spatialColumn shape
-fileName /shapefiles/counties.shp
-messagesFile counties.msg
-createTable 0
-client 1
-srsName NAD83_SRS_1
-commitScope 10000
-idcolumn id
-idColumnIsIdentity 1

```

Improving query performance on spatial data in a partitioned environment

To achieve optimal performance of queries in a partitioned database environment, the table rows that are needed to satisfy the join condition must be co-located. That is, these joins must use table rows that exist on one node without having to access rows from a table, or parts of a table, on another node.

About this task

Spatial joins are often used in applications that find customers within particular polygons or determine the flood risk of customers. Here is an example of a query to determine flood risk:

```

Select
    c.name,
    c.address,
    f.risk
from customers as c,
     floodpoly as f
where db2gse.st_within(c.location, f.polygeom) = 1

```

In this example, the customers table is quite large and distributed across multiple partitions. The flood polygon information is small. To implement the query efficiently, ensure that the entire flood polygon table is available on each partition that contains customer data.

Procedure

1. Import the polygon data onto a single partition.
2. Create a materialized query table (MQT) that is replicated across the partitions that contain the customer data. For example, after the table `floodpoly` has been created and loaded, you can create the replicated MQT with a statement like this:

```

CREATE TABLE floodpoly
AS ( SELECT * FROM floodpoly)
DATA INITIALLY DEFERRED
REFRESH DEFERRED
ENABLE QUERY OPTIMIZATION
MAINTAINED BY SYSTEM
DISTRIBUTE BY REPLICATION IN nodestbs

REFRESH TABLE floodpoly;

```


The replicated MQT with a spatial column can be either user-maintained or system-maintained, but it must use the REFRESH DEFERRED option, not REFRESH IMMEDIATE.

3. To have the DB2 query compiler choose the MQT instead of the base table, you can set the following DB2 parameters:

```
UPDATE DB CFG USING DFT_REFRESH_AGE ANY;  
UPDATE DB CFG USING DFT_MTTB_TYPES ALL;
```

For more information about these parameters, see “UPDATE DATABASE CONFIGURATION command ” in the *Command Reference*.

What to do next

Use the DB2 explain tools to determine how efficiently the query will run.

Chapter 11. Using indexes and views to access spatial data

Use indexes and views to query spatial columns.

Before you query spatial columns, you should learn the particulars in creating indexes and views to access spatial columns. To create such indexes, you must understand the nature of the indexes that Spatial Extender uses to expedite access to spatial data.

Spatial grid indexes

Indexes improve application query performance, especially when the queried table or tables contain many rows. If you create appropriate indexes that the query optimizer chooses to run your query, you can greatly reduce the number of rows to process.

DB2 Spatial Extender provides a grid index that is optimized for two dimensional data. The index is created on the X and Y dimensions of a geometry.

The following aspects of a grid index are helpful to understand:

- The generation of the index
- The use of spatial functions in a query
- How a query uses a spatial grid index

Generation of spatial grid indexes

Spatial Extender generates a spatial grid index using the minimum bounding rectangle (MBR) of a geometry.

For most geometries, the MBR is a rectangle that surrounds the geometry.

A spatial grid index divides a region into logical square grids with a fixed size that you specify when you create the index. The spatial index is constructed on a spatial column by making one or more entries for the intersections of each geometry's MBR with the grid cells. An index entry consists of the grid cell identifier, the geometry MBR, and the internal identifier of the row that contains the geometry.

You can define up to three spatial index levels (grid levels). Using several grid levels is beneficial because it allows you to optimize the index for different sizes of spatial data.

If a geometry intersects four or more grid cells, the geometry is promoted to the next larger level. In general, the larger geometries will be indexed at the larger levels. If a geometry intersects 10 or more grid cells at the largest grid size, a built-in overflow index level is used. This overflow level prevents the generation of too many index entries. For best performance, define your grid sizes to avoid the use of this overflow level.

For example, if multiple grid levels exist, the indexing algorithm attempts to use the lowest grid level possible to provide the finest resolution for the indexed data. When a geometry intersects more than four grid cells at a given level, it is promoted to the next higher level, (provided that there is another level). Therefore,

a spatial index that has the three grid levels of 10.0, 100.0, and 1000.0 will first intersect each geometry with the level 10.0 grid. If a geometry intersects with more than four grid cells of size 10.0, it is promoted and intersected with the level 100.0 grid. If more than four intersections result at the 100.0 level, the geometry is promoted to the 1000.0 level. If more than 10 intersections result at the 1000.0 level, the geometry is indexed in the overflow level.

Use of spatial functions in a query

Using spatial functions in the WHERE clause makes the DB2 optimizer consider spatial grid indexes for the access plan.

The spatial functions that have this effect on the DB2 optimizer are:

- ST_Contains
- ST_Crosses
- ST_Distance
- ST_EnvIntersects
- EnvelopesIntersect
- ST_Equals
- ST_Intersects
- ST_MBRIntersects
- ST_Overlaps
- ST_Touches
- ST_Within

How a query uses a spatial grid index

When the query optimizer chooses a spatial grid index, the query execution uses a multiple-step filter process.

The filter process includes the following steps:

1. Determine the grid cells that intersect the query window. The *query window* is the geometry that you are interested in and that you specify as the second parameter in a spatial function (see following examples).
2. Scan the index for entries that have matching grid cell identifiers.
3. Compare the geometry MBR values in the index entries with the query window and discard any values that are outside the query window.
4. Perform further analysis as appropriate. The candidate set of geometries from the previous steps might undergo further analysis to determine if they satisfy the spatial function (ST_Contains, ST_Distance, and so on). The spatial function EnvelopesIntersect omits this step and typically has the best performance.

The following examples of spatial queries have a spatial grid index on the column C.GEOMETRY:

```
SELECT name
FROM counties AS c
WHERE EnvelopesIntersect(c.geometry, -73.0, 42.0, -72.0, 43.0, 1) = 1
```

```
SELECT name
FROM counties AS c
WHERE ST_Intersects(c.geometry, :geometry2) = 1
```

In the first example, the four coordinate values define the query window. These coordinate values specify the lower-left and upper-right corners (42.0 -73.0 and 43.0 -72.0) of a rectangle.

In the second example, Spatial Extender computes the MBR of the geometry specified by the host variable :geometry2 and uses it as the query window.

When you create a spatial grid index, you should specify appropriate grid sizes for the most common query window sizes that your spatial application is likely to use. If a grid size is larger, index entries for geometries that are outside of the query window must be scanned because they reside in grid cells that intersect the query window, and these extra scans degrade performance. However, a smaller grid size might generate more index entries for each geometry and more index entries must be scanned, which also degrades query performance.

DB2 Spatial Extender provides an Index Advisor utility that analyzes the spatial column data and suggests appropriate grid sizes for typical query window sizes.

Considerations for number of index levels and grid sizes

Use the Index Advisor to determine appropriate grid sizes for your spatial grid indexes because it is the best way to tune the indexes and make your spatial queries most efficient.

Number of grid levels

You can have up to three grid levels.

For each grid level in a spatial grid index, a separate index search is performed during a spatial query. Therefore, if you have more grid levels, your query is less efficient.

If the values in the spatial column are about the same relative size, use a single grid level. However, a typical spatial column does not contain geometries of the same relative size, but geometries in a spatial column can be grouped according to size. You should correspond your grid levels with these geometry groupings.

For example, suppose you have a table of county land parcels with a spatial column that contains groupings of small urban parcels surrounded by larger rural parcels. Because the sizes of the parcels can be grouped into two groups (small urban ones and larger rural ones), you would specify two grid levels for the spatial grid index.

Grid cell sizes

The general rule is to decrease the grid sizes as much as possible to get the finest resolution while minimizing the number of index entries.

- A small value should be used for the finest grid size to optimize the overall index for small geometries in the column. This avoids the overhead of evaluating geometries that are not within the search area. However, the finest grid size also produces the highest number of index entries. Consequently, the number of index entries processed at query time increases, as does the amount of storage needed for the index. These factors reduce overall performance.

- Using larger grid sizes, the index can be optimized further for larger geometries. The larger grid sizes produce fewer index entries for large geometries than the finest grid size would. Consequently, storage requirements for the index are reduced, increasing overall performance.

The following figures show the effects of different grid sizes.

Figure 13 shows a map of land parcels, each parcel represented by a polygon geometry. The black rectangle represents a query window. Suppose you want to find all of the geometries whose MBR intersects the query window. Figure 13 shows that 28 geometries (highlighted in pink) have an MBR that intersects the query window.

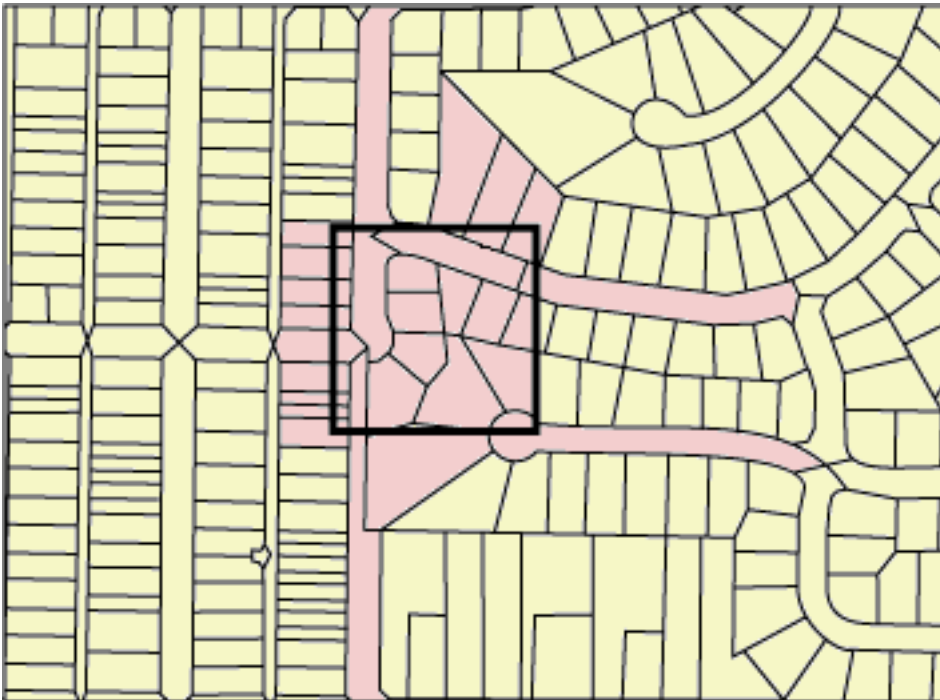


Figure 13. Land parcels in a neighborhood

Figure 14 on page 71 shows a small grid size (25) that provides a close fit to the query window.

- The query returns only the 28 geometries that are highlighted, but the query must examine and discard three additional geometries whose MBRs intersect the query window.
- This small grid size results in many index entries per geometry. During execution, the query accesses all index entries for these 31 geometries. Figure 14 on page 71 shows 256 grid cells that overlay the query window. However, the query execution accesses 578 index entries because many geometries are indexed with the same grid cells.

For this query window, this small grid size results in an excessive number of index entries to scan.

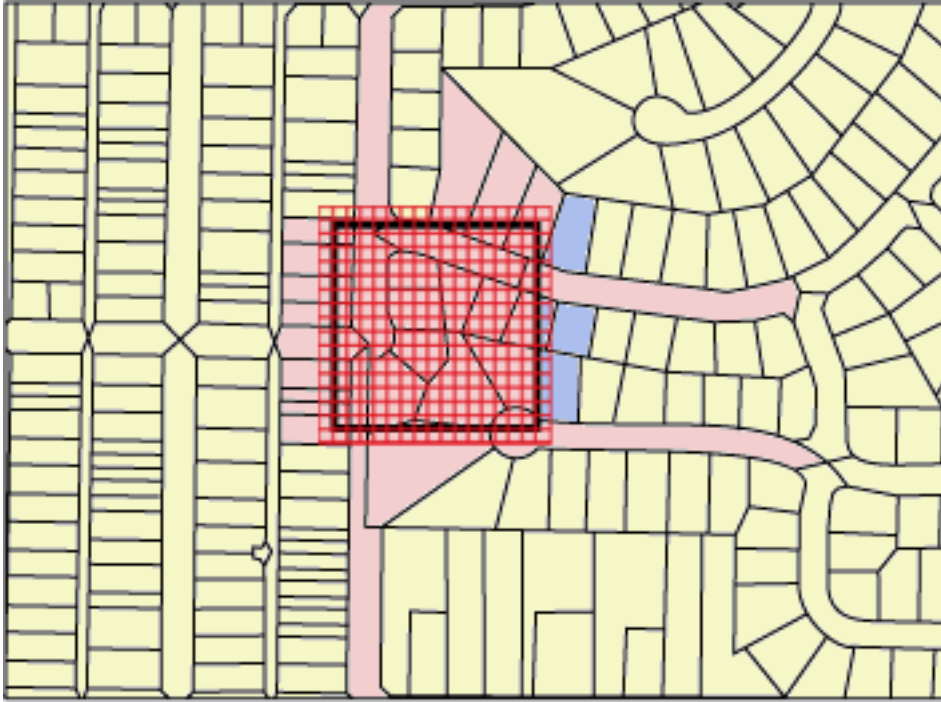


Figure 14. Small grid size (25) on land parcels

Figure 15 on page 72 shows a large grid size (400[®]) that encompasses a considerably larger area with many more geometries than the query window.

- This large grid size results in only one index entry per geometry, but the query must examine and discard 59 additional geometries whose MBRs intersect the grid cell.
- During execution, the query accesses all index entries for the 28 geometries that intersect the query window, plus the index entries for the 59 additional geometries, for a total of 112 index entries.

For this query window, this large grid size results in an excessive number of geometries to examine.

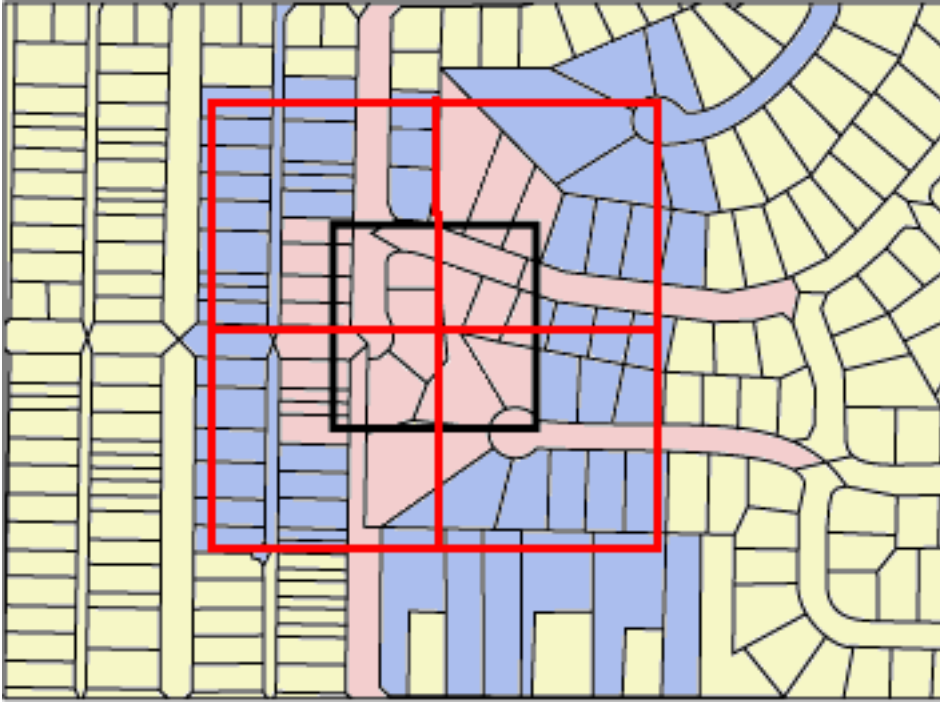


Figure 15. Large grid size (400) on land parcels

Figure 16 on page 73 shows a medium grid size (100) that provides a close fit to the query window.

- The query returns only the 28 geometries that are highlighted, but the query must examine and discard five additional geometries whose MBRs intersect the query window.
- During execution, the query accesses all index entries for the 28 geometries that intersect the query window, plus the index entries for the 5 additional geometries, for a total of 91 index entries.

For this query window, this medium grid size is the best because it results in significantly fewer index entries than the small grid size and the query examines fewer additional geometries than the large grid size.

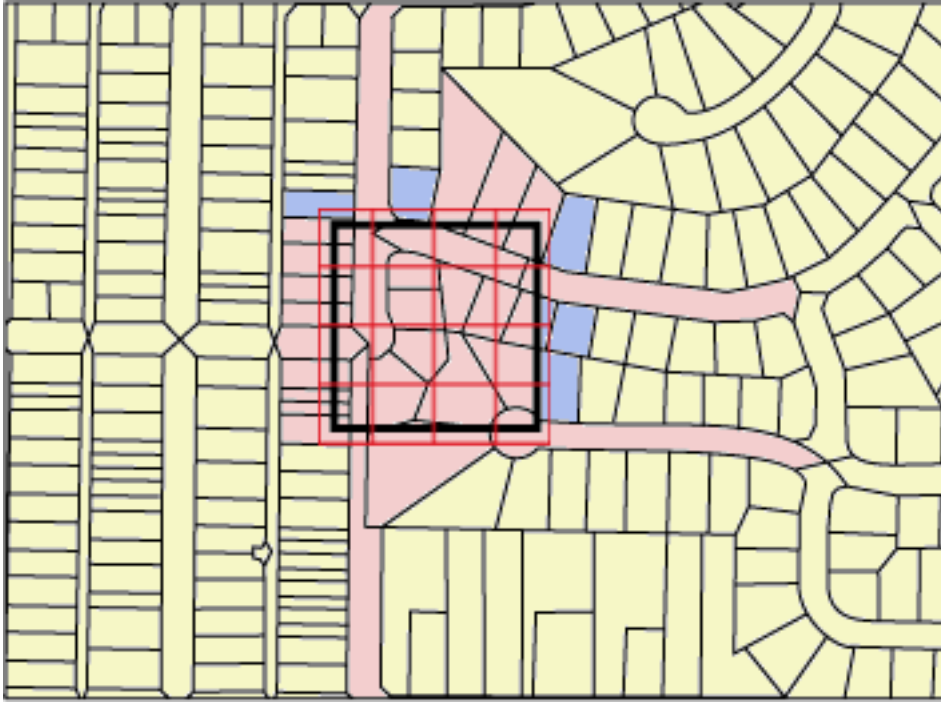


Figure 16. Medium grid size (100) on land parcels

Creating spatial grid indexes

Create spatial grid indexes to define two-dimensional grid indexes on spatial columns to help optimize spatial queries.

Before you begin

Before you create a spatial grid index:

- Your user ID must hold the authorizations that are needed for the CREATE INDEX SQL statement.
- You must know the values that you want to specify for the fully qualified spatial grid index name and the three grid sizes that the index will use.

Recommendations:

- Before you create a spatial grid index on a column, use the Index Advisor to determine the parameters for the index. The Index Advisor can analyze the spatial column data and suggest appropriate grid sizes for your spatial grid index.
- If you plan to do an initial load of data into the column, you should create the spatial grid index after you complete the load process. That way, you can choose optimal grid cell sizes that are based on the characteristics of the data by using the Index Advisor. In addition, loading the data before creating the index will improve the performance of the load process because then the spatial grid index does not need to be maintained during the load process.

table_schema.

Name of the schema to which the table that contains *column_name* belongs. If you do not specify a name, DB2 uses the schema name that is stored in the CURRENT SCHEMA special register.

table_name

Unqualified name of the table that contains *column_name*.

column_name

Name of the spatial column on which the spatial grid index is created.

finest_grid_size, middle_grid_size, coarsest_grid_size

Grid sizes for the spatial grid index. These parameters must adhere to the following conditions:

- *finest_grid_size* must be larger than 0.
- *middle_grid_size* must either be larger than *finest_grid_size* or be 0.
- *coarsest_grid_size* must either be larger than *middle_grid_size* or be 0.

When you create the spatial grid index using the CREATE INDEX statement, the validity of the grid sizes are checked when the first geometry is indexed. Therefore, if the grid sizes that you specify do not meet the conditions of their values, an error condition is raised at the times described in these situations:

- If all of the geometries in the spatial column are null, Spatial Extender successfully creates the index without verifying the validity of the grid sizes. Spatial Extender validates the grid sizes when you insert or update a non-null geometry in that spatial column. If the specified grid sizes are not valid, an error occurs when you insert or update the non-null geometry.
- If non-null geometries exist in the spatial column when you create the index, Spatial Extender validates the grid sizes at that time. If the specified grid sizes are not valid, an error occurs immediately, and the spatial grid index is not created.

Example

The following example CREATE INDEX statement creates the TERRIDX spatial grid index on the spatial column TERRITORY in the BRANCHES table:

```
CREATE INDEX terridx
  ON branches (territory)
  EXTEND USING db2gse.spatial_index (1.0, 10.0, 100.0)
```

Tuning spatial grid indexes with the Index Advisor

DB2 Spatial Extender provides the Index Advisor utility to help you optimize access to spatial data.

Use the Index Advisor to:

- Determine appropriate grid sizes for your spatial grid indexes.
The Index Advisor analyzes the geometries in a spatial column and recommends optimal grid sizes for your spatial grid index.
- Analyze an existing grid index.
The Index Advisor can collect and display statistics from which you can determine how well the current grid cell sizes facilitate retrieval of the spatial data.

Determining grid sizes for a spatial grid index

Before you creating a spatial grid index, use the Index Advisor to determine appropriate grid sizes.

Before you begin

- Your user ID must hold the SELECT privilege on this table.
- If your table has more than one million rows, you might want to use the ANALYZE clause to analyze a subset of the rows to have reasonable processing time.

Procedure

To determine appropriate grid sizes for a spatial grid index:

1. Use the Index Advisor to determine a recommended grid cell size for the index that you want to create.

- a. Enter the command that invokes the Index Advisor with the ADVISE keyword to request grid cell sizes. For example, to invoke the Index Advisor for the SHAPE column in the COUNTIES table, enter:

```
gseidx CONNECT TO mydb USER userID USING password GET GEOMETRY
STATISTICS FOR COLUMN userID.counties(shape) ADVISE
```

Restriction: If you enter this **gseidx** command from an operating system prompt, you must type the entire command on a single line. Alternatively, you can run **gseidx** commands from a CLP file, which allows you to split the command over multiple lines.

The Index Advisor returns recommended grid cell sizes. For example, **gseidx** command with the **ADVISE** keyword shown previously returns the following recommended cell sizes for the SHAPE column:

Query Window Size	Suggested Grid Sizes			Cost
-----	-----	-----	-----	-----
0.1	0.7,	2.8,	14.0	2.7
0.2	0.7,	2.8,	14.0	2.9
0.5	1.4,	3.5,	14.0	3.5
1	1.4,	3.5,	14.0	4.8
2	1.4,	3.5,	14.0	8.2
5	1.4,	3.5,	14.0	24
10	2.8,	8.4,	21.0	66
20	4.2,	14.7,	37.0	190
50	7.0,	14.0,	70.0	900
100	42.0,	0,	0	2800

- b. Choose an appropriate query window size from the **gseidx** output, based on the width of the coordinates that you display on your screen.

In this example, latitude and longitude values in decimal degrees represent the coordinates. If your typical map display has a width of about 0.5 degrees (approximately 55 kilometers), go to the row that has the value 0.5 in the Query Window Size column. This row has suggested grid sizes of 1.4, 3.5, and 14.0.

2. Create the index with the suggested grid sizes. For the example in the previous step, you can execute the following SQL statement:

```
CREATE INDEX counties_shape_idx ON userID.counties(shape)
EXTEND USING DB2GSE.SPATIAL_INDEX(1.4,3.5,14.0);
```

Analyzing spatial grid index statistics

Analyzing statistics on an existing spatial grid index can tell you whether the index is efficient, or whether it should be replaced by a more efficient index.

Before you begin

Before you can analyze the data that you want to index:

- Your user ID must hold the SELECT privilege on this table.
- If your table has more than one million rows, you might want to use the ANALYZE clause to analyze a subset of the rows to have reasonable processing time. You must have a USER TEMPORARY table space available to use the ANALYZE clause. Set the page size of this table space to at least 8 KB and ensure that you have USE privileges on it. For example, the following DDL statements create a buffer pool with the same page size as the user temporary table space and grant the USE privilege to anyone:

```
CREATE BUFFERPOOL bp8k SIZE 1000 PAGESIZE 8 K;  
CREATE USER TEMPORARY TABLESPACE usertempts  
    PAGESIZE 8K  
    MANAGED BY SYSTEM USING ('c:\tempts')  
    BUFFERPOOL bp8k  
GRANT USE OF TABLESPACE usertempts TO PUBLIC;
```

About this task

Use the Index Advisor to obtain statistics on an existing spatial grid indexes. Analyze these statistics and determine if you should replace any indexes.

Tip: Equally important to tuning your index is verifying that it is being used by your queries. To determine if a spatial index is being used, run a command line tool such as **db2exfmt** on your query. In the “Access Plan” section of the explain output, if you see an EISCAN operator and the name of your spatial index, then the query uses your index.

Procedure

To analyze statistics on an existing spatial grid indexes and determine whether they should be replaced by more efficient indexes:

Obtain statistics on a spatial grid index and, if necessary, to replace the index:

1. Have the Index Advisor collect statistics based on the grid cell sizes of the existing index. You can ask for statistics on either a subset of the indexed data or all of the data.
 - To obtain statistics on indexed data in a subset of rows, enter the **gseidx** command and specify the **ANALYZE** keyword and its parameters in addition to the existing-index clause and **DETAIL** keyword. You can specify either the number or percentage of rows that the Index Advisor is to analyze to obtain statistics. For example, to obtain statistics on a subset of the data indexed by the COUNTIES_SHAPE_IDX index, enter:

```
gseidx CONNECT TO mydb USER userID USING password GET GEOMETRY  
STATISTICS FOR INDEX userID.counties_shape_idx DETAIL ANALYZE 25 PERCENT  
ADVISE
```

- To obtain statistics on all indexed data, enter the **gseidx** command and specify its existing-index clause. Include the **DETAIL** keyword. For example, to invoke the Index Advisor for the COUNTIES_SHAPE_IDX index, enter:

```
gseidx CONNECT TO mydb USER userID USING password GET GEOMETRY  
STATISTICS FOR INDEX userID.counties_shape_idx DETAIL SHOW HISTOGRAM ADVISE
```

The Index Advisor returns statistics, a histogram of the data, and recommended cell sizes for the existing index. For example, the preceding **gseidx** command for all data indexed by COUNTIES_SHAPE_IDX returns the following statistics:

Grid Level 1

Grid Size : 0.5
 Number of Geometries : 2936
 Number of Index Entries : 12197

Number of occupied Grid Cells : 2922
 Index Entry/Geometry ratio : 4.154292
 Geometry/Grid Cell ratio : 1.004791
 Maximum number of Geometries per Grid Cell: 14
 Minimum number of Geometries per Grid Cell: 1

Index Entries :	1	2	3	4	10
-----	-----	-----	-----	-----	-----
Absolute :	86	564	72	1519	695
Percentage (%):	2.93	19.21	2.45	51.74	23.67

Grid Level 2

Grid Size : 0.0
 No geometries indexed on this level.

Grid Level 3

Grid Size : 0.0
 No geometries indexed on this level.

Grid Level X

Number of Geometries : 205
 Number of Index Entries : 205

- Determine how well the grid cell sizes of the existing index facilitate retrieval. Assess the statistics returned in the previous step.

Tip:

- The statistic "Index Entry/Geometry ratio" should be a value in the range of 1 to 4, preferably values closer to 1.
- The number of index entries per geometry should be less than 10 at the largest grid size to avoid the overflow level.

The appearance of the "Grid Level X" section in the Index Advisor output indicates that an overflow level exists.

The index statistics obtained in the previous step for the COUNTIES_SHAPE_IDX indicate that the grid sizes (0.5, 0, 0) are not appropriate for the data in this column because:

- For Grid Level 1, the "Index Entry/Geometry ratio" value 4.154292 is greater than the guideline of 4.

The "Index Entries" line has the values 1, 2, 3, 4, and 10, which indicates the number of index entries per geometry. The "Absolute" values below each "Index Entries" column indicates the number of geometries that have that specific number of index entries. For example, the output in the previous step shows 1519 geometries have 4 index entries. The "Absolute" value for 10 index entries is 695 which indicates that 695 geometries have between 5 and 10 index entries.

- The appearance of the “Grid Level X” section indicates that an overflow index level exists. The statistics show that 205 geometries have more than 10 index entries each.
3. If the statistics are not satisfactory, look at the Histogram section and the appropriate rows in the Query Window Size and Suggested Grid Sizes columns in the Index Advisor output.
 - a. Find the MBR size with the largest number of geometries. The “Histogram” section lists the MBR sizes and the number of geometries that have that MBR size. In the following sample histogram, the largest number of geometries (437) is in MBR size 0.5.

Histogram:

MBR Size	Geometry Count
0.040000	1
0.045000	3
0.050000	1
0.055000	3
0.060000	3
0.070000	4
0.075000	3
0.080000	4
0.085000	1
0.090000	2
0.095000	1
0.150000	10
0.200000	9
0.250000	15
0.300000	23
0.350000	83
0.400000	156
0.450000	282
0.500000	437
0.550000	397
0.600000	341
0.650000	246
0.700000	201
0.750000	154
0.800000	120
0.850000	66
0.900000	79
0.950000	59
1.000000	47
1.500000	230
2.000000	89
2.500000	34
3.000000	10
3.500000	5
4.000000	3
5.000000	3
5.500000	2
6.000000	2
6.500000	3
7.000000	2
8.000000	1
15.000000	3
25.000000	2
30.000000	1

- b. Go to the Query Window Size row with the value 0.5 to obtain the suggested grid sizes (1.4, 3.5, 14.0).

Query Window Size	Suggested Grid Sizes	Cost
0.1	0.7, 2.8, 14.0	2.7

0.2	0.7,	2.8,	14.0	2.9
0.5	1.4,	3.5,	14.0	3.5
1	1.4,	3.5,	14.0	4.8
2	1.4,	3.5,	14.0	8.2
5	1.4,	3.5,	14.0	24
10	2.8,	8.4,	21.0	66
20	4.2,	14.7,	37.0	190
50	7.0,	14.0,	70.0	900
100	42.0,	0,	0	2800

4. Verify that the recommended sizes meet the guidelines in step 2. Run the **gseidx** command with the suggested grid sizes:

```
gseidx CONNECT TO mydb USER userID USING password GET GEOMETRY
STATISTICS FOR COLUMN userID.counties(shape) USING GRID SIZES (1.4, 3.5, 14.0)
```

Grid Level 1

```
Grid Size           : 1.4
Number of Geometries : 3065
Number of Index Entries : 5951
```

```
Number of occupied Grid Cells : 513
Index Entry/Geometry ratio    : 1.941599
Geometry/Grid Cell ratio      : 5.974659
Maximum number of Geometries per Grid Cell: 42
Minimum number of Geometries per Grid Cell: 1
```

Index Entries :	1	2	3	4	10
-----	-----	-----	-----	-----	-----
Absolute :	1180	1377	15	493	0
Percentage (%):	38.50	44.93	0.49	16.08	0.00

Grid Level 2

```
Grid Size           : 3.5
Number of Geometries : 61
Number of Index Entries : 143
```

```
Number of occupied Grid Cells : 56
Index Entry/Geometry ratio    : 2.344262
Geometry/Grid Cell ratio      : 1.089286
Maximum number of Geometries per Grid Cell: 10
Minimum number of Geometries per Grid Cell: 1
```

Index Entries :	1	2	3	4	10
-----	-----	-----	-----	-----	-----
Absolute :	15	28	0	18	0
Percentage (%):	24.59	45.90	0.00	29.51	0.00

Grid Level 3

```
Grid Size           : 14.0
Number of Geometries : 15
Number of Index Entries : 28
```

```
Number of occupied Grid Cells : 9
Index Entry/Geometry ratio    : 1.866667
Geometry/Grid Cell ratio      : 1.666667
Maximum number of Geometries per Grid Cell: 10
Minimum number of Geometries per Grid Cell: 1
```

Index Entries :	1	2	3	4	10
-----------------	---	---	---	---	----

Absolute	:	7	5	1	2	0
Percentage (%)	:	46.67	33.33	6.67	13.33	0.00

The statistics now show values within the guidelines:

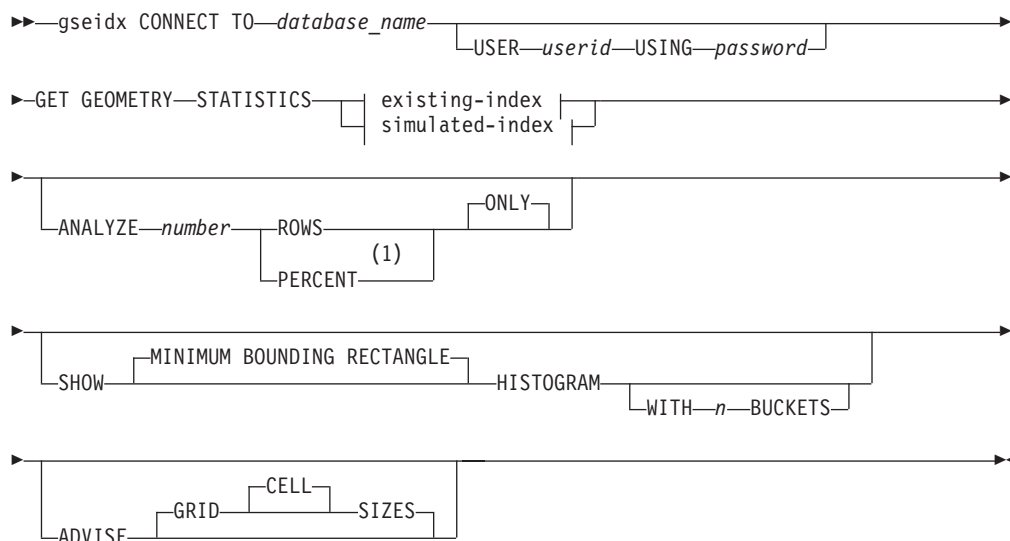
- The “Index Entry/Geometry ratio” values are 1.941599 for Grid Level 1, 2.344262 for Grid Level 2, and 1.866667 for Grid Level 3. These values are all within the guideline value range of 1 to 4.
 - The absence of the “Grid Level X” section indicates that no index entries are in the overflow level.
5. Drop the existing index and replace it with an index that specifies the advised grid sizes. For the sample in the previous step, run the following DDL statements:

```
DROP INDEX userID.counties_shape_idx;
CREATE INDEX counties_shape_idx ON userID.counties(shape) EXTEND USING
DB2GSE.SPATIAL_INDEX(1.4,3.5,14.0);
```

gseidx command

Use the **gseidx** command to invoke the Index Advisor for spatial grid indexes.

Syntax

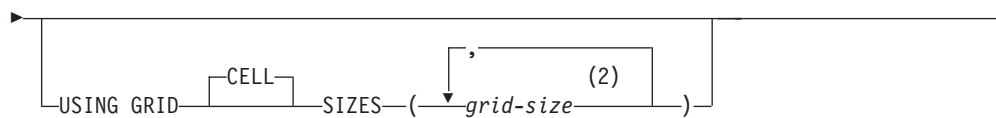


existing-index:

```
FOR INDEX [index-schema.] index-name [DETAIL]
```

simulated-index:

```
FOR COLUMN [table-schema.] table-name (column-name)
```



Notes:

- 1 Instead of the PERCENT keyword, you can specify a percentage sign (%).
- 2 You can specify cell sizes for one, two, or three grid levels.

Parameters

database_name

The name of the database in which the spatial table resides.

userid The user ID that has DATAACCESS authority on the database in which the index or table resides or SELECT authority on the table. If you log on to the DB2 command environment with the user ID of the database owner, you do not need to specify *userid* and *password* in the **gseidx** command.

password

Password for the user ID.

existing-index

References an existing index to gather statistics on.

index-schema

Name of the schema that includes the existing index.

index-name

Unqualified name of the existing index.

DETAIL

Shows the following information about each grid level:

- The size of the grid cells
- The number of geometries indexed
- The number of index entries
- The number of grid cells that contain geometries
- The average number of index entries per geometry
- The average number of geometries per grid cell
- The number of geometries in the cell that contains the most geometries
- The number of geometries in the cell that contains the fewest geometries

simulated-index

References a table column and a simulated index for this column.

table-schema

Name of the schema that includes the table with the column for which the simulated index is intended.

table-name

Unqualified name of the table with the column for which the simulated index is intended.

column-name

Unqualified name of the table column for which the simulated index is intended.

grid-size

Sizes of the cells in each grid level (finest level, middle level, and coarsest level) of a simulated index. You must specify a cell size for at least one level. If you do not want to include a level, either do not specify a grid cell size for it or specify a grid cell size of zero (0.0) for it.

When you specify the *grid-size* parameter, the Index Adviser returns the same kinds of statistics that it returns when you include the DETAIL keyword in the existing-index clause.

ANALYZE number ROWS | PERCENT ONLY

Specify the approximate quantity or approximate percentage of the rows used to gather statistics on data. If your table has more than one million rows, use the ANALYZE clause to gather statistics for a data subset so that you can have a reasonable processing time.

SHOW MINIMUM BOUNDING RECTANGLE HISTOGRAM

Displays a chart that shows the sizes of the geometries' minimum bounding rectangles (MBRs) and the number of geometries whose MBRs are of the same size.

WITH n BUCKETS

Specifies to the number of groupings for the MBRs of all analyzed geometries. Small MBRs are grouped together with other small geometries. The larger MBRs are grouped with other larger geometries.

If you do not specify this parameter or specify 0 buckets, the Index Advisor displays logarithmic bucket sizes. For example, the MBR sizes might be logarithmic values such as 1.0, 2.0, 3.0,... 10.0, 20.0, 30.0,... 100.0, 200.0, 300.0,...

If you specify a number of buckets greater than 0, the Index Advisor displays equal-sized values. For example, the MBR sizes might be equal-sized values such as 8.0, 16.0, 24.0,... 320.0, 328.0, 334.0.

The default is to use logarithmic-sized buckets.

ADVISE GRID CELL SIZES

Computes close-to-optimal grid cell sizes.

Usage note

If you enter the **gseidx** command from an operating system prompt, you must type the entire command on a single line.

Example

The following example is a request to return detailed information about an existing grid index whose name is COUNTIES_SHAPE_IDX and suggest appropriate grid index sizes:

```
gseidx CONNECT TO mydb USER user ID USING password GET GEOMETRY
STATISTICS FOR INDEX userID.counties_shape_idx DETAIL ADVISE
```

Using views to access spatial columns

You can define a view that uses a spatial column in the same way as you define views in DB2 for other data types.

About this task

This is particularly useful when a table has multiple spatial columns as many visualization applications can only work with a table or view with a single spatial column. The view definition can select the appropriate spatial column and other non-spatial columns to be made available to the application.

Chapter 12. Analyzing and Generating spatial information

Analyzing and Generating spatial information requires an understanding of the environments in which you can submit queries and the guidelines on using spatial functions in conjunction with spatial indexes. Review examples of the various types of spatial functions that you can invoke in a query to learn about the capabilities that Spatial Extender offers.

Environments for performing spatial analysis

You can perform spatial analysis by using SQL and spatial functions from the DB2 command line processor and from Application programs in all languages supported by DB2.

Examples of how spatial functions operate

DB2 Spatial Extender provides functions that perform various operations on spatial data. These functions can be categorized according to the type of operation that they perform.

Table 2 lists these categories, along with examples. The text following Table 2 shows coding for these examples.

Table 2. Spatial functions and operations

Category of function	Example of operation
Returns information about specific geometries.	Return the extent, in square miles, of the sales area of Store 10.
Makes comparisons.	Determine whether the location of a customer's home lies within the sales area of Store 10.
Derives new geometries from existing ones.	Derive the sales area of a store from its location.
Converts geometries to and from data exchange formats.	Convert customer information in GML format into a geometry, so that the information can be added to a DB2 database.

Example 1: Returns information about specific geometries

In this example, the ST_Area function returns a numeric value that represents the sales area of store 10. The function will return the area in the same units as the units of the coordinate system that is being used to define the area's location.

```
SELECT db2gse.ST_Area(sales_area)
FROM   stores
WHERE  id = 10
```

The following example shows the same operation as the preceding one, but with ST_Area invoked as a method and returning the area in units of square miles.

```
SELECT sales_area..ST_Area('STATUTE MILE')
FROM   stores
WHERE  id = 10
```

Example 2: Makes comparisons

In this example, the ST_Within function compares the coordinates of the geometry representing a customer's residence with the coordinates of a geometry representing the sales area of store 10. The function's output will signify whether the residence lies within the sales area.

```
SELECT c.first_name, c.last_name, db2gse.ST_Within(c.location, s.sales_area)
FROM   customers as c, stores AS s
WHERE  s.id = 10
```

Example 3: Derives new geometries from existing ones

In this example, the function ST_Buffer derives a geometry representing a store's sales area from a geometry representing the store's location.

```
UPDATE stores
SET    sales_area = db2gse.ST_Buffer(location, 10, 'KILOMETERS')
WHERE  id = 10
```

The following example shows the same operation as the preceding one, but with ST_Buffer invoked as a method.

```
UPDATE stores
SET    sales_area = location..ST_Buffer(10, 'KILOMETERS')
WHERE  id = 10
```

Example 4: Converts geometries to and from data exchange formats.

In this example, customer information coded in GML is converted into a geometry, so that it can be stored in a DB2 database.

```
INSERT
INTO   c.name,c.phoneNo,c.address
VALUES ( 123, 'Mary Anne', Smith', db2gse.ST_Point('
<gml:Point><gml:coord><gml:X>-130.876</gml:X>
<gml:Y>41.120</gml:Y></gml:coord></gml:Point>, 1) )
```

Functions that use indexes to optimize queries

A specialized group of spatial functions, called *comparison functions*, can improve query performance by exploiting spatial grid indexes. Each of these functions compares two geometries with one another.

If the results of the comparison meet certain criteria, the function returns a value of 1; if the results fail to meet the criteria, the function returns a value of 0. If the comparison cannot be performed, the function can return a null value.

For example, the function ST_Overlaps compares two geometries that have the same dimension (for example, two linestrings or two polygons). If the geometries overlap partway, and if the space covered by the overlap has the same dimension as the geometries, ST_Overlaps returns a value of 1.

Table 3 shows which comparison functions can use a spatial grid index:

Table 3. Comparison functions that can use a spatial grid index

Comparison function	Can use spatial grid index
EnvelopesIntersect	Yes
ST_Contains	Yes

Table 3. Comparison functions that can use a spatial grid index (continued)

Comparison function	Can use spatial grid index
ST_Crosses	Yes
ST_Distance	Yes
ST_EnvIntersects	Yes
ST_Equals	Yes
ST_Intersects	Yes
ST_MBRIntersects	Yes
ST_Overlaps	Yes
ST_Touches	Yes
ST_Within	Yes

Because of the time and memory required to execute a function, such execution can involve considerable processing. Furthermore, the more complex the geometries that are being compared, the more complex and time-intensive the comparison will be. The specialized functions listed previously can complete their operations more quickly if they can use a spatial index to locate geometries. To enable such a function to use a spatial index, observe all of the following rules:

- The function must be specified in a WHERE clause. If it is specified in a SELECT, HAVING, or GROUP BY clause, a spatial index cannot be used.
- The function must be the expression on left of the predicate.
- The operator that is used in the predicate that compares the result of the function with another expression must be an equal sign, with one exception: the ST_Distance function must use the less than operator.
- The expression on the right of the predicate must be the constant 1, except when ST_Distance is the function on the left.
- The operation must involve a search in a spatial column on which a spatial index is defined.

For example:

```
SELECT c.name, c.address, c.phone
FROM customers AS c, bank_branches AS b
WHERE db2gse.ST_Distance(c.location, b.location) < 10000
      and b.branch_id = 3
```

Table 4 shows correct and incorrect ways of creating spatial queries to utilize a spatial index.

Table 4. Demonstration of how spatial functions can adhere to and violate rules for utilizing a spatial index.

Queries that reference spatial functions	Rules violated
SELECT * FROM stores AS s WHERE db2gse.ST_Contains(s.sales_zone, ST_Point(-121.8,37.3, 1)) = 1	No condition is violated in this example.
SELECT * FROM stores AS s WHERE db2gse.ST_Length(s.location) > 10	The spatial function ST_Length does not compare geometries and cannot utilize a spatial index.
SELECT * FROM stores AS s WHERE 1=db2gse.ST_Within(s.location,:BayArea)	The function must be an expression on the left side of the predicate.

Table 4. Demonstration of how spatial functions can adhere to and violate rules for utilizing a spatial index. (continued)

Queries that reference spatial functions	Rules violated
<pre>SELECT * FROM stores AS s WHERE db2gse.ST_Contains(s.sales_zone, ST_Point(-121.8,37.3, 1)) <> 0</pre>	Equality comparisons must use the integer constant 1.
<pre>SELECT * FROM stores AS s WHERE db2gse.ST_Contains(ST_Polygon('POLYGON((10 10, 10 20, 20 20, 20 10, 10 10))', 1), ST_Point(-121.8, 37.3, 1)) = 1</pre>	No spatial index exists on either of the arguments for the function, so no index can be utilized.

Chapter 13. Writing applications and using the sample program

To write applications for Spatial Extender, you must understand the requirements and review the sample program.

Including the DB2 Spatial Extender header file in spatial applications

DB2 Spatial Extender provides a header file that defines constants that can be used with the stored procedures and functions of the DB2 Spatial Extender.

About this task

Recommendation:

If you plan to call DB2 Spatial Extender stored procedures or functions from C or C++ programs, include this header file in your spatial applications.

Procedure

To include the DB2 Spatial Extender header file in spatial applications:

1. Ensure that your DB2 Spatial Extender applications can use the necessary definitions in this header file.
 - a. Include the DB2 Spatial Extender header file in your application program. The header file has the following name:
`db2gse.h`
The header file is located in the *db2path*/include directory, where *db2path* is the installation directory where the DB2 database system is installed.
 - b. Ensure that the path of the include directory is specified in your makefile with the compilation option.
2. If you are building Windows 64-bit applications on a Windows 32-bit system, change the DB2_LIBS parameter in the `samples/extenders/spatial/makefile.nt` file to accommodate 64-bit applications. The necessary changes are highlighted in the following example:
`DB2_LIBS = $(DB2_DIR)\lib\Win64\db2api.lib`

Calling DB2 Spatial Extender stored procedures from an application

If you plan to write application programs that call any of the DB2 Spatial Extender stored procedures, you use the SQL CALL statement and specify the name of the stored procedure.

About this task

DB2 Spatial Extender stored procedures are created when you enable the database for spatial operations.

Procedure

To call DB2 Spatial Extender stored procedures from an application:

1. Call the DB2GSE.ST_ENABLE_DB stored procedure to enable a database for spatial operations.

Specify the stored procedure name as follows:

```
CALL DB2GSE!ST_ENABLE_DB
```

The DB2GSE! in this call represents the DB2 Spatial Extender library name. The ST_ENABLE_DB procedure is the only one in which you need to include an exclamation mark in the call (that is, DB2GSE!).

2. Call other DB2 Spatial Extender stored procedures. Specify the stored procedure name in the following form, where DB2GSE is the schema name for all DB2 Spatial Extender stored procedures, and *spatial_procedure_name* is the name of the stored procedure. Do not include an exclamation mark in the call.

```
CALL DB2GSE.spatial_procedure_name
```

The DB2 Spatial Extender stored procedures are shown in the following table.

Table 5. DB2 Spatial Extender stored procedures

Stored procedure	Description
ST_ALTER_COORDSYS	Updates an attribute of a coordinate system in the database.
ST_ALTER_SRS	Updates an attribute of a spatial reference system in the database.
ST_CREATE_COORDSYS	Creates a coordinate system in the database.
ST_CREATE_SRS	Creates a spatial reference system in the database.
ST_DISABLE_AUTOGEOCODING	Specifies that DB2 Spatial Extender is to stop synchronizing a geocoded column with its associated geocoding columns.
ST_DISABLE_DB	Removes resources that allow DB2 Spatial Extender to store spatial data and to support operations that are performed on this data.
ST_DROP_COORDSYS	Deletes a coordinate system from the database.
ST_DROP_SRS	Deletes a spatial reference system from the database.
ST_ENABLE_AUTOGEOCODING	Specifies that DB2 Spatial Extender is to synchronize a geocoded column with its associated geocoding columns.
ST_ENABLE_DB	Supplies a database with the resources that it needs to store spatial data and to support operations.
ST_EXPORT_SHAPE	Exports selected data in the database to a shape file.
ST_IMPORT_SHAPE	Imports a shape file to a database.
ST_REGISTER_GEOCODER	Registers a geocoder other than DB2SE_USA_GEOCODER, which is part of the DB2 Spatial Extender product.
ST_REGISTER_SPATIAL_COLUMN	Registers a spatial column and associates a spatial reference system with it.
ST_REMOVE_GEOCODING_SETUP	Removes all the geocoding setup information for the geocoded column.
ST_RUN_GEOCODING	Runs a geocoder in batch mode.
ST_SETUP_GEOCODING	Associates a column that is to be geocoded with a geocoder and sets up the corresponding geocoding parameter values.

Table 5. DB2 Spatial Extender stored procedures (continued)

Stored procedure	Description
ST_UNREGISTER_GEOCODER	Unregisters a geocoder.
ST_UNREGISTER_SPATIAL_COLUMN	Removes the registration of a spatial column.

DB2 Spatial Extender sample program

The DB2 Spatial Extender sample program, `runGseDemo`, has two purposes. You can use the sample program to become familiar with application programming for DB2 Spatial Extender, and you can use the program to verify the DB2 Spatial Extender installation.

The location of the `runGseDemo` program varies and depends on the operating system where DB2 Spatial Extender is installed.

- On UNIX, you can locate the `runGseDemo` program in the following path:

`$HOME/sql1ib/samples/extenders/spatial`

where `$HOME` is the instance owner's home directory.

- On Windows®, you can locate the `runGseDemo` program in the following path:

`c:\Program Files\IBM\sql1ib\samples\extenders\spatial`

where `c:\Program Files\IBM\sql1ib` is the directory in which you installed DB2 Spatial Extender.

The DB2 Spatial Extender `runGseDemo` sample program makes application programming easier. Using this sample program, you can enable a database for spatial operations and perform spatial analysis on data in that database. This database will contain tables with fictitious information about customers and flood zones. From this information you can experiment with Spatial Extender and determine which customers are at risk of suffering damage from a flood.

With the sample program, you can:

- See the steps typically required to create and maintain a spatially-enabled database.
- Understand how to call spatial stored procedures from an application program.
- Cut and paste sample code into your own applications.

Use the following sample program to code tasks for DB2 Spatial Extender. For example, suppose that you write an application that uses the database interface to call DB2 Spatial Extender stored procedures. From the sample program, you can copy code to customize your application. If you are unfamiliar with the programming steps for DB2 Spatial Extender, you can run the sample program, which shows each step in detail. For instructions on running the sample program, see “Related tasks” at the end of this topic.

The following table describes each step in the sample program. In each step you will perform an action and, in many cases, reverse or undo that action. For example, in the first step you will enable the spatial database and then disable the spatial database. In this way, you will become familiar with many of the Spatial Extender stored procedures.

Table 6. DB2 Spatial Extender sample program steps

Steps	Action and description
Enable or disable the spatial database	<ul style="list-style-type: none"> • Enable the spatial database 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, new index types, and a set of spatial catalog tables and views. • Disable the spatial database This step is usually performed when you have enabled spatial capabilities for the wrong database, or you no longer need to perform spatial operations in this database. When you disable a spatial database, you remove the set of spatial types, the set of spatial functions, the set of spatial predicates, new index types, and the set of spatial catalog tables and views associated with that database. • Enable the spatial database Same as the previous action.
Create or drop a coordinate system	<ul style="list-style-type: none"> • Create a coordinate system named NORTH_AMERICAN This step creates a new coordinate system in the database. • Drop the coordinate system named NORTH_AMERICAN This step drops the coordinate system NORTH_AMERICAN from the database. • Create a coordinate system named KY_STATE_PLANE This step creates a new coordinate system, KY_STATE_PLANE, which will be used by the spatial reference system created in the next step.
Create or drop a spatial reference system	<ul style="list-style-type: none"> • Create a spatial reference system named SRSDEMO1 This step defines a new spatial reference system (SRS) that is used to interpret the coordinates. The SRS 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 spatial column, the coordinates that are applicable to that spatial column can be stored in the associated column of the CUSTOMERS table. • Drop the SRS named SRSDEMO1 This step is performed if you no longer need the SRS in the database. When you drop an SRS, you remove the SRS definition from the database. • Create the SRS named KY_STATE_SRS

Table 6. DB2 Spatial Extender sample program steps (continued)

Steps	Action and description
Create and populate the spatial tables	<ul style="list-style-type: none"> • Create the CUSTOMERS table • Populate the CUSTOMERS table The CUSTOMERS table represents business data that has been stored in the database for several years. • Alter the CUSTOMERS table by adding the LOCATION column 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. • Create the OFFICES table 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 subsequent step involves importing attribute data into the OFFICES table from a shape file.
Populate the columns	<ul style="list-style-type: none"> • Geocode the addresses data for the LOCATION column of the CUSTOMERS table with the geocoder named KY_STATE_GC This step performs batch spatial 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. • Load the previously-created OFFICES table from the shape file using spatial reference system KY_STATE_SRS This step loads the OFFICES table with existing spatial data in the form of a shape file. Because the OFFICES table exists, the LOAD utility will append the new records to an existing table. • Create and load the FLOODZONES table from the shape file using spatial reference system KY_STATE_SRS This step loads the FLOODZONES table with existing data in the form of a shape file. Because the table does not exist, the LOAD utility will create the table before the data is loaded. • Create and load the REGIONS table from the shape file using spatial reference system KY_STATE_SRS
Register or unregister the geocoder	<ul style="list-style-type: none"> • Register the geocoder named SAMPLEGC • Unregister the geocoder named SAMPLEGC • Register the geocoder KY_STATE_GC <p>These steps register and unregister the geocoder named SAMPLEGC and then create a new geocoder, KY_STATE_GC, to use in the sample program.</p>

Table 6. DB2 Spatial Extender sample program steps (continued)

Steps	Action and description
Create spatial indexes	<ul style="list-style-type: none"> • Create the spatial grid index for the LOCATION column of the CUSTOMERS table • Drop the spatial grid index for the LOCATION column of the CUSTOMERS table • Create the spatial grid index for the LOCATION column of the CUSTOMERS table • Create the spatial grid index for the LOCATION column of the OFFICES table • Create the spatial grid index for the LOCATION column of the FLOODZONES table • Create the spatial grid index for the LOCATION column of the REGIONS table <p>These steps create the spatial grid index for the CUSTOMERS, OFFICES, FLOODZONES, and REGIONS tables.</p>
Enable automatic geocoding	<ul style="list-style-type: none"> • Set up geocoding for the LOCATION column of the CUSTOMERS table with geocoder KY_STATE_GC This step associates the LOCATION column of the CUSTOMERS table with geocoder KY_STATE_GC and sets up the corresponding values for geocoding parameters. • Enable automatic geocoding for the LOCATION column of the CUSTOMERS table This step turns on the automatic invocation of the geocoder. Using automatic geocoding causes the LOCATION, LATITUDE, and LONGITUDE columns of the CUSTOMERS table to be synchronized with each other for subsequent insert and update operations.
Perform insert, update, and delete operations on the CUSTOMERS table	<p>These steps demonstrate insert, update, and delete operations on the LATITUDE, LONGITUDE, STREET, CITY, STATE, and ZIP columns of the CUSTOMERS table. After the automatic geocoding is enabled, data that is inserted or updated in these columns is automatically geocoded into the LOCATION column. This process was enabled in the previous step.</p> <ul style="list-style-type: none"> • Insert some records with a different street • Update some records with a new address • Delete all records from the table
Disable automatic geocoding	<p>These steps disable the automatic invocation of the geocoder and the spatial index in preparation for the next step. The next step involves re-geocoding the entire CUSTOMERS table.</p> <ul style="list-style-type: none"> • Disable automatic geocoding for the LOCATION column in the CUSTOMERS table • Remove the geocoding setup for the LOCATION column of the CUSTOMERS table • Drop the spatial index for the LOCATION column of the CUSTOMERS table <p>Recommendation: If you are loading a large amount of geodata, drop the spatial index before you load the data, and then re-create it after the data is loaded.</p>

Table 6. DB2 Spatial Extender sample program steps (continued)

Steps	Action and description
Create a view and register the spatial column in the view	<p>These steps create a view and register its spatial column.</p> <ul style="list-style-type: none"> • Create a view called HIGHRISKCUSTOMERS based on the join of the CUSTOMERS table and the FLOODZONES table • Register the view's spatial column
Perform spatial analysis	<p>These steps perform spatial analysis using the spatial predicates and functions in DB2 SQL. The DB2 query optimizer exploits the spatial index on the spatial columns to improve the query performance whenever possible.</p> <ul style="list-style-type: none"> • Find the number of customers served by each region (ST_Within) • For offices and customers with the same region, find the number of customers that are within a specific distance of each office (ST_Within, ST_Distance) • For each region, find the average income and premium of each customer (ST_Within) • Find the number of flood zones that each office zone overlaps (ST_Overlaps) • Find the nearest office from a specific customer location, assuming that the office is located in the centroid of the office zone (ST_Distance) • Find the customers whose location is close to the boundary of a specific flood zone (ST_Buffer, ST_Intersects) • Find those high-risk customers within a specified distance from a specific office (ST_Within) <p>All of these steps use the gseRunSpatialQueries internal function.</p>
Export spatial data into shape files	<p>This step shows an example of exporting the HIGHRISKCUSTOMERS view to shape files. Exporting data from a database format to another file format enables the information to be used by other tools (such as ArcExplorer for DB2).</p> <ul style="list-style-type: none"> • Export the HIGHRISKCUSTOMERS view to shape files

Chapter 14. Identifying DB2 Spatial Extender problems

To identify a DB2 Spatial Extender problem, you must determine the cause of the problem.

You can troubleshoot problems with DB2 Spatial Extender in these ways:

- You can use message information to diagnose the problem.
- When working with Spatial Extender stored procedures and functions, DB2 returns information about the success or failure of the stored procedure or function. The information returned will be a message code (in the form of an integer), message text, or both depending on the interface that you use to work with DB2 Spatial Extender.
- You can view the DB2 administration notification file, which records diagnostic information about errors.
- If you have a recurring and reproducible Spatial Extender problem, an IBM customer support representative might ask you to use the DB2 trace facility to help them diagnose the problem.

How to interpret DB2 Spatial Extender messages

Understanding how DB2 Spatial Extender messages are structured and how to obtain additional information for them can help you determine whether the spatial operation that you requested completed successfully or resulted in an error.

You can work with DB2 Spatial Extender by using any of the following interfaces:

- DB2 Spatial Extender stored procedures
- DB2 Spatial Extender functions
- DB2 Spatial Extender Command Line Processor (CLP)

All these interfaces return DB2 Spatial Extender messages.

The following table explains each part of this sample DB2 Spatial Extender message text:

GSE0000I: The operation was completed successfully.

Table 7. The parts of the DB2 Spatial Extender message text

Message text part	Description
GSE	The message identifier. All DB2 Spatial Extender messages begin with the three-letter prefix GSE.
0000	The message number. A four digit number that ranges from 0000 through 9999.
I	The message type. A single letter that indicates the severity of message: C Critical error messages N Non-critical error messages W Warning messages I Informational messages
The operation was completed successfully.	The message explanation.

The explanation that appears in the message text is the brief explanation. You can retrieve additional information about the message that includes the detailed explanation and suggestions to avoid or correct the problem. To display this additional information:

1. Open an operating system command prompt.
2. Enter the DB2 help command with the message identifier and message number to display additional information about the message. For example:

```
DB2 "? GSEnnnn"
```

where *nnnn* is the message number.

You can type the GSE message identifier and letter indicating the message type in uppercase or lowercase. Typing DB2 "? GSE0000I" will yield the same result as typing db2 "? gse0000i".

You can omit the letter after the message number when you type the command. For example, typing DB2 "? GSE0000" will yield the same result as typing DB2 "? GSE0000I".

Suppose the message code is GSE4107N. When you type DB2 "? GSE4107N" at the command prompt, the following information is displayed:

GSE4107N Grid size value "<grid-size>" is not valid where it is used.

Explanation: The specified grid size "<grid-size>" is not valid.

One of the following invalid specifications was made when the grid index was created with the CREATE INDEX statement:

- A number less than 0 (zero) was specified as the grid size for the first, second, or third grid level.
- 0 (zero) was specified as the grid size for the first grid level.
- The grid size specified for the second grid level is less than the grid size of the first grid level but it is not 0 (zero).
- The grid size specified for the third grid level is less than the grid size of the second grid level but it is not 0 (zero).
- The grid size specified for the third grid level is greater than 0 (zero) but the grid size specified for the second grid level is 0 (zero).

User Response: Specify a valid value for the grid size.

msgcode: -4107

sqlstate: 38SC7

If the information is too long to display on a single screen and your operating system supports the **more** executable program and pipes, type this command:

```
db2 "? GSEnnnn" | more
```

Using the **more** program will force the display to pause after each screen of data so that you can read the information.

DB2 Spatial Extender stored procedure output parameters

Use the DB2 Spatial Extender stored procedure output parameters to diagnose problems when explicitly calling stored procedures in application programs or from the DB2 command line processor.

DB2 Spatial Extender stored procedures have two output parameters: the message code (msg_code) and the message text (msg_text). The parameter values indicate the success or failure of a stored procedure.

msg_code

The msg_code parameter is an integer, which can be positive, negative, or zero (0). Positive numbers are used for warnings, negative numbers are used for errors (both critical and non-critical), and zero (0) is used for informational messages.

The absolute value of the msg_code is included in the msg_text as the message number. For example

- If the msg_code is 0, the message number is 0000.
- If the msg_code is -219, the message number is 0219. The negative msg_code indicates that the message is a critical or non-critical error.
- If the msg_code is +1036, the message number is 1036. The positive msg_code number indicates that the message is a warning.

The msg_code numbers for Spatial Extender stored procedures are divided into the three categories shown in the following table:

Table 8. Stored procedure message codes

Codes	Category
0000 - 0999	Common messages
1000 - 1999	Administrative messages
2000 - 2999	Import and export messages

msg_text

The msg_text parameter consists of the message identifier, the message number, the message type, and the explanation. An example of a stored procedure msg_text value is:

```
GSE0219N  An EXECUTE IMMEDIATE statement
          failed. SQLERROR = "<sql-error>".
```

The explanation that appears in the msg_text parameter is the brief explanation. You can retrieve additional information about the message that includes the detailed explanation and suggestions to avoid or correct the problem.

For a detailed explanation of the parts of the msg_text parameter, and information on how to retrieve additional information about the message, see the topic: How to interpret DB2 Spatial Extender messages.

To diagnose stored procedures called implicitly by issuing the DB2 Spatial Extender commands, use the messages returned by the DB2 Spatial Extender CLP. For more details, see “How to interpret DB2 Spatial Extender messages” on page 97

Working with stored procedures in applications

When you call a DB2 Spatial Extender stored procedure from an application, you will receive the `msg_code` and `msg_text` as output parameters. You can:

- Program your application to return the output parameter values to the application user.
- Perform some action based on the type of `msg_code` value returned.

Working with stored procedures from the DB2 command line

When you invoke a DB2 Spatial Extender stored procedure from the DB2 command line, you receive the `msg_code` and the `msg_text` output parameters. These output parameters indicate the success or failure of the stored procedure.

Suppose you connect to a database and want to invoke the `ST_DISABLE_DB` procedure. The following example uses a DB2 `CALL` command to disable the database for spatial operations and shows the output value results. A force parameter value of 0 is used, along with two question marks at the end of the `CALL` command to represent the `msg_code` and `msg_text` output parameters. The values for these output parameters are displayed after the stored procedure runs.

```
call db2gse.st_disable_db(0, ?, ?)
```

```
Value of output parameters
```

```
-----
```

```
Parameter Name : MSGCODE
```

```
Parameter Value : 0
```

```
Parameter Name : MSGTEXT
```

```
Parameter Value : GSE0000I The operation was completed successfully.
```

```
Return Status = 0
```

Suppose the `msg_text` returned is `GSE2110N`. Use the DB2 help command to display more information about the message. For example:

```
"? GSE2110"
```

The following information is displayed:

```
GSE2110N    The spatial reference system for the
             geometry in row "<row-number>" is invalid.
             The spatial reference system's
             numeric identifier is "<srs-id>".
```

Explanation: In row *row-number*, the geometry that is to be exported uses an invalid spatial reference system. The geometry cannot be exported.

User Response: Correct the indicated geometry or exclude the row from the export operation by modifying the `SELECT` statement accordingly.

```
msg_code: -2110
```

```
sqlstate: 38S9A
```

DB2 Spatial Extender function messages

Messages returned by DB2 Spatial Extender functions are typically embedded in an SQL message. The SQLCODE returned in the message indicates if an error occurred with the function or that a warning is associated with the function.

The following messages are examples that indicate an error and a warning:

- The SQLCODE -443 (message number SQL0443) indicates that an error occurred with the function.
- The SQLCODE +462 (message number SQL0462) indicates that a warning is associated with the function.

The following table explains the significant parts of this sample message:

DB21034E The command was processed as an SQL statement because it was not a valid Command Line Processor command. During SQL processing it returned: SQL0443N Routine "DB2GSE.GSEGEOMFROMWKT" (specific name "GSEGEOMWKT1") has returned an error SQLSTATE with diagnostic text "GSE3421N Polygon is not closed.". SQLSTATE=38SSL

Table 9. The significant parts of DB2 Spatial Extender function messages

Message part	Description
SQL0443N	The SQLCODE indicates the type of problem.
GSE3421N	The DB2 Spatial Extender message number and message type. The message numbers for functions range from GSE3000 to GSE3999. Additionally, common messages can be returned when you work with DB2 Spatial Extender functions. The message numbers for common messages range from GSE0001 to GSE0999.
Polygon is not closed SQLSTATE=38SSL	The DB2 Spatial Extender message explanation. An SQLSTATE code that further identifies the error. An SQLSTATE code is returned for each statement or row. <ul style="list-style-type: none">• The SQLSTATE codes for Spatial Extender function errors are 38Sxx, where each <i>x</i> is a character letter or number.• The SQLSTATE codes for Spatial Extender function warnings are 01HSx, where the <i>x</i> is a character letter or number.

An example of an SQL0443 error message

Suppose that you attempt to insert the values for a polygon into the table POLYGON_TABLE, as shown in the following statement:

```
INSERT INTO polygon_table ( geometry )  
VALUES ( ST_Polygon ( 'polygon (( 0 0, 0 2, 2 2, 1 2)) ' ) )
```

This results in an error message because you did not provide the end value to close the polygon. The error message returned is:

DB21034E The command was processed as an SQL statement because it was not a valid Command Line Processor command. During SQL processing it returned: SQL0443N Routine "DB2GSE.GSEGEOMFROMWKT" (specific name "GSEGEOMWKT1") has returned an error SQLSTATE with diagnostic text "GSE3421N Polygon is not closed.". SQLSTATE=38SSL

The SQL message number SQL0443N indicates that an error occurred and the message includes the Spatial Extender message text GSE3421N Polygon is not closed.

When you receive this type of message:

1. Locate the GSE message number within the DB2 or SQL error message.
2. Use the DB2 help command (DB2 ?) to see the Spatial Extender message explanation and user response. Using this example, type the following command in an operating system command line prompt:

```
DB2 "? GSE3421"
```

The message is repeated, along with a detailed explanation and recommended user response.

DB2 Spatial Extender CLP messages

The DB2 Spatial Extender CLP messages indicate the success or failure of an operation. Also, they can provide shape information.

The DB2 Spatial Extender CLP returns messages for:

- Stored procedures, if invoked implicitly.
- Shape information, if you have invoked the **shape_info** subcommand program from the DB2 Spatial Extender CLP. These are informational messages.
- Upgrade operations.
- Import and export shape operations to and from the client.

Examples of messages for stored procedures returned by the DB2 Spatial Extender CLP

Most of the messages returned through the DB2 Spatial Extender CLP are for DB2 Spatial Extender stored procedures. When you invoke a stored procedure from the DB2 Spatial Extender CLP, you will receive message text that indicates the success or failure of the stored procedure.

The message text consists of the message identifier, the message number, the message type, and the explanation. For example, if you enable a database using the command `db2se enable_db testdb`, the message text returned by the Spatial Extender CLP is:

```
Enabling database. Please wait ...
```

```
GSE1036W  The operation was successful. But
          values of certain database manager and
          database configuration parameters
          should be increased.
```

Likewise, if you disable a database using the command `db2se disable_db testdb` the message text returned by the Spatial Extender CLP is:

```
GSE0000I  The operation was completed successfully.
```

The explanation that appears in the message text is the brief explanation. You can retrieve additional information about the message that includes the detailed explanation and suggestions to avoid or correct the problem. The steps to retrieve this information, and a detailed explanation of how to interpret the parts of the message text, are discussed in a separate topic.

If you are invoking stored procedures through an application program or from the DB2 command line, there is a separate topic that discusses diagnosing the output parameters.

Example of messages for shape information returned by the Spatial Extender CLP

Suppose you decide to display information for a shape file named `office`. Through the Spatial Extender CLP (db2se) you would issue this command:

```
db2se shape_info -fileName /tmp/offices
```

This is an example of the information that displays:

Shape file information

```
-----
File code                = 9994
File length (16-bit words) = 484
Shape file version       = 1000
Shape type               = 1 (ST_POINT)
Number of records        = 31
```

```
Minimum X coordinate = -87.053834
Maximum X coordinate = -83.408752
Minimum Y coordinate = 36.939628
Maximum Y coordinate = 39.016477
Shapes do not have Z coordinates.
Shapes do not have M coordinates.
```

Shape index file (extension `.shx`) is present.

Attribute file information

```
-----
dBase file code          = 3
Date of last update      = 1901-08-15
Number of records        = 31
Number of bytes in header = 129
Number of bytes in each record = 39
Number of columns        = 3
```

Column Number	Column Name	Data Type	Length	Decimal
1	NAME	C (Character)	16	0
2	EMPLOYEES	N (Numeric)	11	0
3	ID	N (Numeric)	11	0

```
Coordinate system definition: "GEOGCS["GCS_North_American_1983",
DATUM["D_North_American_1983",SPHEROID["GRS_1980",6378137,298.257222101]],
PRIMEM["Greenwich",0],UNIT["Degree",0.017453292519943295]]"
```

Examples of messages for upgrade operations returned by the Spatial Extender CLP

When you invoke commands that perform upgrade operations, messages are returned that indicate the success or failure of that operation.

Suppose you upgrade the spatially enabled database `mydb` using the following command:

```
db2se upgrade mydb -messagesFile /tmp/db2se_upgrade.msg
```

The message text returned by the Spatial Extender CLP is:

```
Upgrading database. Please wait ...
GSE0000I The operation was completed successfully.
```

Tracing DB2 Spatial Extender problems with the **db2trc** command

When you have a recurring and reproducible DB2 Spatial Extender problem, you can use the DB2 trace facility to capture information about the problem.

Before you begin

Ensure that you have the proper authorization to issue the **db2trc** command. On UNIX operating systems, you must have SYSADM, SYSCTRL, or SYSMAINT authorization to trace a DB2 instance. On Windows operating systems, no special authorization is required.

About this task

The DB2 trace facility is activated by the **db2trc** command. The DB2 trace facility can:

- Trace events
- Dump the trace data to a file
- Format trace data into a readable format

Restrictions

- Activate this facility only when directed by a DB2 technical support representative.
- The **db2trc** command must be entered at an operating system command prompt or in a shell script. It cannot be used in the DB2 Spatial Extender command-line interface (**db2se**) or in the DB2 CLP.

Procedure

To troubleshoot problems in DB2 Spatial Extender by using the DB2 trace facility:

1. Shut down all other applications.
2. Turn on the trace.

The DB2 Support technical support representative provides you with the specific parameters for this step. The basic command is:

```
db2trc on
```

You can trace to memory or to a file. The preferred method for tracing is to trace to memory. If the problem being recreated suspends the workstation and prevents you from dumping the trace, trace to a file.

3. Reproduce the problem.
4. Dump the trace to a file immediately after the problem occurs.

For example:

```
db2trc dump january23trace.dmp
```

This command creates a file (january23trace.dmp) in the current directory with the name that you specify, and dumps the trace information in that file. You can specify a different directory by including the file path. For example, to place the dump file in the /tmp/spatial/errors directory, the syntax is:

```
db2trc dump /tmp/spatial/errors/january23trace.dmp
```

5. Turn off the trace.

For example:

```
db2trc off
```

6. Format the data as an ASCII file. You can sort the data two ways:

- Use the **flw** option to sort the data by process or thread. For example:
`db2trc flw january23trace.dmp january23trace.flw`
- Use the **fmt** option to list every event chronologically. For example:
`db2trc fmt january23trace.dmp january23trace.fmt`

The administration notification file

Diagnostic information about errors is recorded in the administration notification file. This information is used for problem determination and is intended for DB2 technical support.

The administration notification file contains text information logged by the DB2 database system as well as by DB2 Spatial Extender. It is located in the directory specified by the **diagpath** database manager configuration parameter. On Windows operating systems, the DB2 administration notification file is found in the event log and can be reviewed through the Windows Event Viewer.

The information that the DB2 database manager records in the administration log is determined by the **diaglevel** and **notifylevel** settings.

Use a text editor to view the file on the machine where you suspect a problem to have occurred. The most recent events recorded are the furthest down the file. Generally, each entry contains the following parts:

- A timestamp.
- The location reporting the error. Application identifiers allow you to match up entries pertaining to an application on the logs of servers and clients.
- A diagnostic message (usually beginning with "DIA" or "ADM") explaining the error.
- Any available supporting data, such as SQLCA data structures and pointers to the location of any extra dump or trap files.

If the database is behaving normally, this type of information is not important and can be ignored.

The administration notification file grows continuously. When it gets too large, back it up and then erase the file. A new file is generated automatically the next time it is required by the system.

Chapter 15. Catalog views

You can use catalog views for Spatial Extender to obtain useful information about your spatial data.

Spatial Extender's catalog views contain information about:

"DB2GSE.ST_COORDINATE_SYSTEMS catalog view"

Coordinate systems that you can use

"DB2GSE.ST_GEOMETRY_COLUMNS catalog view" on page 108

Spatial columns that you can populate or update.

"DB2GSE.ST_GEOCODERS catalog view" on page 111 and

"DB2GSE.ST_GEOCODER_PARAMETERS catalog view" on page 109

Geocoders that you can use

"DB2GSE.ST_GEOCODING catalog view" on page 111 and

"DB2GSE.ST_GEOCODING_PARAMETERS catalog view" on page 112

Specifications for setting up a geocoder to run automatically and for setting, in advance, operations to be performed during batch geocoding.

"DB2GSE.ST_SIZINGS catalog view" on page 114

Allowable maximum lengths of values that you can assign to variables.

"DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view" on page 114

Spatial reference systems that you can use.

"DB2GSE.ST_UNITS_OF_MEASURE catalog view" on page 117

The units of measure (meters, miles, feet, and so on) in which distances generated by spatial functions can be expressed.

DB2GSE.ST_COORDINATE_SYSTEMS catalog view

Query the DB2GSE.ST_COORDINATE_SYSTEMS catalog view to retrieve information about registered coordinate systems.

Spatial Extender automatically registers coordinate systems in the Spatial Extender catalog at the following times:

- When you enable a database for spatial operations.
- When users define additional coordinate systems to the database.

For a description of columns in this view, see the following table.

Table 10. Columns in the DB2GSE.ST_COORDINATE_SYSTEMS catalog view

Name	Data type	Nullable?	Content
COORDSYS_NAME	VARCHAR(128)	No	Name of this coordinate system. The name is unique within the database.

Table 10. Columns in the DB2GSE.ST_COORDINATE_SYSTEMS catalog view (continued)

Name	Data type	Nullable?	Content
COORDSYS_TYPE	VARCHAR(128)	No	Type of this coordinate system: PROJECTED Two-dimensional. GEOGRAPHIC Three-dimensional. Uses X and Y coordinates. GEOCENTRIC Three-dimensional. Uses X, Y, and Z coordinates. UNSPECIFIED Abstract or non-real world coordinate system. The value for this column is obtained from the DEFINITION column.
DEFINITION	VARCHAR(2048)	No	Well-known text representation of the definition of this coordinate system.
ORGANIZATION	VARCHAR(128)	Yes	Name of the organization (for example, a standards body such as the European Petrol Survey Group, or ESPG) that defined this coordinate system.
ORGANIZATION_COORDSYS_ID	INTEGER	Yes	This column is null if the ORGANIZATION_COORDSYS_ID column is null. Numeric identifier assigned to this coordinate system by the organization that defined the coordinate system. This identifier and the value in the ORGANIZATION column uniquely identify the coordinate system unless the identifier and the value are both null. If the ORGANIZATION column is null, then the ORGANIZATION_COORDSYS_ID column is also null.
DESCRIPTION	VARCHAR(256)	Yes	Description of the coordinate system that indicates its application.

DB2GSE.ST_GEOMETRY_COLUMNS catalog view

Use the DB2GSE.ST_GEOMETRY_COLUMNS catalog view to find information about all spatial columns in all tables that contain spatial data in the database.

If a spatial column was registered in association with a spatial reference system, you can also use the view to find out the spatial reference system name and numeric identifier. For additional information about spatial columns, query the SYSCAT.COLUMN catalog view.

If a spatial column was registered in association with a spatial reference system, and **compute_extents** was specified, you can also use this view to query information about the geographic extents for the spatial column and the date when they were last computed.

For a description of DB2GSE.ST_GEOMETRY_COLUMNS, see the following table.

Table 11. Columns in the DB2GSE.ST_GEOMETRY_COLUMNS catalog view

Name	Data type	Nullable?	Content
TABLE_SCHEMA	VARCHAR(128)	No	Name of the schema to which the table that contains this spatial column belongs.
TABLE_NAME	VARCHAR(128)	No	Unqualified name of the table that contains this spatial column.
COLUMN_NAME	VARCHAR(128)	No	Name of this spatial column.
TYPE_SCHEMA	VARCHAR(128)	No	The combination of TABLE_SCHEMA, TABLE_NAME, and COLUMN_NAME uniquely identifies the column. Name of the schema to which the declared data type of this spatial column belongs. This name is obtained from the DB2 catalog.
TYPE_NAME	VARCHAR(128)	No	Unqualified name of the declared data type of this spatial column. This name is obtained from the DB2 catalog.
SRS_NAME	VARCHAR(128)	Yes	Name of the spatial reference system that is associated with this spatial column. If no spatial reference system is associated with the column, then SRS_NAME is null.
SRS_ID	INTEGER	Yes	Numeric identifier of the spatial reference system that is associated with this spatial column. If no spatial reference system is associated with the column, then SRS_ID is null.
MIN_X	DOUBLE	Yes	Minimum <i>x</i> value of all spatial values in the column.
MIN_Y	DOUBLE	Yes	Minimum <i>y</i> value of all spatial values in the column.
MIN_Z	DOUBLE	Yes	Minimum <i>z</i> value of all spatial values in the column.
MAX_X	DOUBLE	Yes	Maximum <i>x</i> value of all spatial values in the column.
MAX_Y	DOUBLE	Yes	Maximum <i>y</i> value of all spatial values in the column.
MAX_Z	DOUBLE	Yes	Maximum <i>z</i> value of all spatial values in the column.
MIN_M	DOUBLE	Yes	Minimum <i>m</i> value of all spatial values in the column.
MAX_M	DOUBLE	Yes	Maximum <i>m</i> value of all spatial values in the column.
EXTENT_TIME	TIMESTAMP	Yes	Timestamp of when the extents were last computed.

DB2GSE.ST_GEOCODER_PARAMETERS catalog view

Use the DB2GSE.ST_GEOCODER_PARAMETERS catalog view to retrieve information about the parameters of registered geocoders.

For more information about geocoder parameters, query the SYSCAT.ROUTINEPARMS catalog view.

For a description of columns in this view, see the following table.

Table 12. Columns in the DB2GSE.ST_GEOCODER_PARAMETERS

Name	Data type	Nullable?	Content
GEOCODER_NAME	VARCHAR(128)	No	Name of the geocoder to which this parameter belongs.
ORDINAL	SMALLINT	No	Position of this parameter (that is, the parameter specified in the PARAMETER_NAME column) in the signature of the function that serves as the geocoder specified in the GEOCODER_NAME column. The combined values in the GEOCODER_NAME and ORDINAL columns uniquely identify this parameter. A record in the SYSCAT.ROUTINEPARMS catalog view also contains information about this parameter. This record contains a value that appears in the ORDINAL column of SYSCAT.ROUTINEPARMS. This value is the same one that appears in the ORDINAL column of the DB2GSE.ST_GEOCODER_PARAMETERS view.
PARAMETER_NAME	VARCHAR(128)	Yes	Name of this parameter. If a name was not specified when the function to which this parameter belongs was created, the PARAMETER_NAME column is null. The content of the PARAMETER_NAME column is obtained from the DB2 catalog.
TYPE_SCHEMA	VARCHAR(128)	No	Name of the schema to which this parameter belongs. This name is obtained from the DB2 catalog.
TYPE_NAME	VARCHAR(128)	No	Unqualified name of the data type of the values assigned to this parameter. This name is obtained from the DB2 catalog.
PARAMETER_DEFAULT	VARCHAR(2048)	Yes	The default value that is to be assigned to this parameter. DB2 will interpret this value as an SQL expression. If the value is enclosed in quotation marks, it will be passed to the geocoder as a string. Otherwise, the evaluation of the SQL expression will determine what parameter's data type will be when it is passed to the geocoder. If the PARAMETER_DEFAULT column contains a null, then this null value will be passed to the geocoder. The default value can have a corresponding value in the DB2GSE.ST_GEOCODING_PARAMETERS catalog view. It can also have a corresponding value in the input parameters for the ST_RUN_GEOCODING procedure. If either corresponding value differs from the default value, the corresponding value will override the default value.
DESCRIPTION	VARCHAR(256)	Yes	Description of the parameter indicating its application.

DB2GSE.ST_GEOCODERS catalog view

When you want to make additional geocoders available to users, you need to register these geocoders. To retrieve information about registered geocoders, query the DB2GSE.ST_GEOCODERS catalog view.

For information about geocoders' parameters, query the DB2GSE.ST_GEOCODER_PARAMETERS catalog view and the SYSCAT.ROUTINEPARMS catalog view. For information about functions that are used as geocoders, query the SYSCAT.ROUTINES catalog view.

For a description of columns in DB2GSE.ST_GEOCODERS view, see the following table.

Table 13. Columns in the DB2GSE.ST_GEOCODERS catalog view

Name	Data type	Nullable?	Content
GEOCODER_NAME	VARCHAR(128)	No	Name of this geocoder. It is unique within the database.
FUNCTION_SCHEMA	VARCHAR(128)	No	Name of the schema to which the function that is being used as this geocoder belongs.
FUNCTION_NAME	VARCHAR(128)	No	Unqualified name of the function that is being used as this geocoder.
SPECIFIC_NAME	VARCHAR(128)	No	Specific name of the function that is being used as this geocoder.
RETURN_TYPE_SCHEMA	VARCHAR(128)	No	The combined values of FUNCTION_SCHEMA and SPECIFIC_NAME uniquely identify the function that is being used as this geocoder.
RETURN_TYPE_NAME	VARCHAR(128)	No	Name of the schema to which the data type of this geocoder output parameter belongs. This name is obtained from the DB2 catalog.
VENDOR	VARCHAR(256)	Yes	Unqualified name of the data type of this geocoder output parameter. This name is obtained from the DB2 catalog.
DESCRIPTION	VARCHAR(256)	Yes	Name of the vendor that created this geocoder.
			Description of the geocoder that indicates its application.

DB2GSE.ST_GEOCODING catalog view

When you set up geocoding operations, the particulars of your settings are automatically recorded in the DB2 Spatial Extender catalog. To find out these particulars, query the DB2GSE.ST_GEOCODING and DB2GSE.ST_GEOCODING_PARAMETERS catalog views.

The DB2GSE.ST_GEOCODING catalog view, which is described in Table 14 on page 112, contains particulars of all settings; for example, the number of records that a geocoder is to process before each commit.

The DB2GSE.ST_GEOCODING_PARAMETERS catalog view contains particulars that are specific to each geocoder. For example, the minimum degree to which addresses given as input and actual addresses must match in order for the geocoder to geocode the input. This minimum requirement, called the minimum match score, is recorded in the DB2GSE.ST_GEOCODING_PARAMETERS catalog view.

Table 14. Columns in the DB2GSE.ST_GEOCODING catalog view

Name	Data type	Nullable?	Content
TABLE_SCHEMA	VARCHAR(128)	No	Name of the schema that contains the table that contains the column identified in the COLUMN_NAME column.
TABLE_NAME	VARCHAR(128)	No	Unqualified name of the table that contains the column identified in the COLUMN_NAME column.
COLUMN_NAME	VARCHAR(128)	No	Name of the spatial column to be populated according to the specifications shown in this catalog view.
GEOCODER_NAME	VARCHAR(128)	No	The combined values in the TABLE_SCHEMA, TABLE_NAME, and COLUMN_NAME columns uniquely identify the spatial column. Name of the geocoder that is to produce data for the spatial column specified in the COLUMN_NAME column. Only one geocoder can be assigned to a spatial column.
MODE	VARCHAR(128)	No	Mode for the geocoding process: BATCH Only batch geocoding is enabled. AUTO Automatic geocoding is set up and activated. INVALID An inconsistency in the spatial catalog tables was detected; the geocoding entry is invalid.
SOURCE_COLUMNS	VARCHAR(10000)	Yes	Names of table columns set up for automatic geocoding. Whenever these columns are updated, a trigger prompts the geocoder to geocode the updated data.
WHERE_CLAUSE	VARCHAR(10000)	Yes	Search condition within a WHERE clause. This condition indicates that when the geocoder runs in batch mode, it is geocode only data within a specified subset of records.
COMMIT_COUNT	INTEGER	Yes	The number of rows that are to be processed during batch geocoding before a commit is issued. If the value in the COMMIT_COUNT column is 0 (zero) or null, then no commits are issued.

DB2GSE.ST_GEOCODING_PARAMETERS catalog view

When you set up geocoding operations for a particular geocoder, geocoder-specific aspects of the settings are available through the DB2GSE.ST_GEOCODING_PARAMETERS catalog view.

For example, an operation can be comparing addresses given as input to reference data.

When you set up operations for a geocoder, you specify what the degree, called the minimum match score, should be; and your specification is recorded in the catalog.

To find out the geocoder-specific aspects of a settings for geocoding operations, query the DB2GSE.ST_GEOCODING_PARAMETERS catalog view. This view is described in the following table.

Certain defaults for set-ups of geocoding operations are available in the DB2GSE.ST_GEOCODER_PARAMETERS catalog view. Values in the DB2GSE.ST_GEOCODING_PARAMETERS view override the defaults.

Table 15. Columns in the DB2GSE.ST_GEOCODING_PARAMETERS catalog view

Name	Data type	Nullable?	Content
TABLE_SCHEMA	VARCHAR(128)	No	Name of the schema that contains the table that contains the column identified in the COLUMN_NAME column.
TABLE_NAME	VARCHAR(128)	No	Unqualified name of the table that contains the spatial column.
COLUMN_NAME	VARCHAR(128)	No	Name of the spatial column to be populated according to the specifications shown in this catalog view.
ORDINAL	SMALLINT	No	<p>The combined values in the TABLE_SCHEMA, TABLE_NAME, and COLUMN_NAME columns uniquely identify this spatial column.</p> <p>Position of this parameter (that is, the parameter specified in the PARAMETER_NAME column) in the signature of the function that serves as the geocoder for the column identified in the COLUMN_NAME column.</p> <p>A record in the SYSCAT.ROUTINEPARMS catalog view also contains information about this parameter. This record contains a value that appears in the ORDINAL column of SYSCAT.ROUTINEPARMS. This value is the same one that appears in the ORDINAL column of the DB2GSE.ST_GEOCODING_PARAMETERS view.</p>
PARAMETER_NAME	VARCHAR(128)	Yes	<p>Name of a parameter in the definition of the geocoder. If no name was specified when the geocoder was defined, PARAMETER_NAME is null.</p> <p>This content of the PARAMETER_NAME column is obtained from the DB2 catalog.</p>
PARAMETER_VALUE	VARCHAR(2048)	Yes	<p>The value that is assigned to this parameter. DB2 will interpret this value as an SQL expression. If the value is enclosed in quotation marks, it will be passed to the geocoder as a string. Otherwise, the evaluation of the SQL expression will determine what the parameter's data type will be when it is passed to the geocoder. If the PARAMETER_VALUE column contains a null, then this null is passed to the geocoder.</p> <p>The PARAMETER_VALUE column corresponds to the PARAMETER_DEFAULT column in the DB2GSE.ST_GEOCODER_PARAMETERS catalog view. If the PARAMETER_VALUE column contains a value, this value overrides the default value in the PARAMETER_DEFAULT column. If the PARAMETER_VALUE column is null, the default value will be used.</p>

DB2GSE.ST_SIZINGS catalog view

Use the DB2GSE.ST_SIZINGS catalog view to retrieve information about supported variables and their maximum length.

This catalog view returns the following information:

- All the variables supported by Spatial Extender; for example, coordinate system name, geocoder name, and variables to which well-known text representations of spatial data can be assigned.
- The allowable maximum length, if known, of values assigned to these variables (for example, the maximum allowable lengths of names of coordinate systems, of names of geocoders, and of well-known text representations of spatial data).

For a description of columns in this view, see the following table.

Table 16. Columns in the DB2GSE.ST_SIZINGS catalog view

Name	Data type	Nullable?	Content
VARIABLE_NAME	VARCHAR(128)	No	Term that denotes a variable. The term is unique within the database.
SUPPORTED_VALUE	INTEGER	Yes	Allowable maximum length of the values assigned to the variable shown in the VARIABLE_NAME column. Possible values in the SUPPORTED_VALUE column are: A numeric value other than 0 The allowable maximum length of values assigned to this variable. 0 Either any length is allowed, or the allowable length cannot be determined. NULL Spatial Extender does not support this variable.
DESCRIPTION	VARCHAR(128)	Yes	Description of this variable.

DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view

Query the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view to retrieve information about registered spatial reference systems.

Spatial Extender automatically registers spatial reference systems in the Spatial Extender catalog at the following times:

- When you enable a database for spatial operations, it registers five default spatial reference systems.
- When users create additional spatial reference systems.

To get full value from the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view, you need to understand that each spatial reference system is associated with a coordinate system. The spatial reference system is designed partly to convert coordinates derived from the coordinate system into values that DB2 can process with maximum efficiency, and partly to define the maximum possible extent of space that these coordinates can reference.

To find out the name and type of the coordinate system associated with a given spatial reference system, query the COORDSYS_NAME and COORDSYS_TYPE columns of the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view. For

more information about the coordinate system, query the DB2GSE.ST_COORDINATE_SYSTEMS catalog view.

Table 17. Columns in the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view

Name	Data type	Nullable?	Content
SRS_NAME	VARCHAR(128)	No	Name of the spatial reference system. This name is unique within the database.
SRS_ID	INTEGER	No	numeric identifier of the spatial reference system. Each spatial reference system has a unique numeric identifier.
X_OFFSET	DOUBLE	No	Spatial functions specify spatial reference systems by their numeric identifiers rather than by their names. Offset to be subtracted from all X coordinates of a geometry. The subtraction is a step in the process of converting the geometry's coordinates into values that DB2 can process with maximum efficiency. A subsequent step is to multiply the figure resulting from the subtraction by the scale factor shown in the X_SCALE column.
X_SCALE	DOUBLE	No	Scale factor by which to multiply the figure that results when an offset is subtracted from an X coordinate. This factor is identical to the value shown in the Y_SCALE column.
Y_OFFSET	DOUBLE	No	Offset to be subtracted from all Y coordinates of a geometry. The subtraction is a step in the process of converting the geometry's coordinates into values that DB2 can process with maximum efficiency. A subsequent step is to multiply the figure resulting from the subtraction by the scale factor shown in the Y_SCALE column.
Y_SCALE	DOUBLE	No	Scale factor by which to multiply the figure that results when an offset is subtracted from a Y coordinate. This factor is identical to the value shown in the X_SCALE column.
Z_OFFSET	DOUBLE	No	Offset to be subtracted from all Z coordinates of a geometry. The subtraction is a step in the process of converting the geometry's coordinates into values that DB2 can process with maximum efficiency. A subsequent step is to multiply the figure resulting from the subtraction by the scale factor shown in the Z_SCALE column.
Z_SCALE	DOUBLE	No	Scale factor by which to multiply the figure that results when an offset is subtracted from a Z coordinate.
M_OFFSET	DOUBLE	No	Offset to be subtracted from all measures associated with a geometry. The subtraction is a step in the process of converting the measures into values that DB2 can process with maximum efficiency. A subsequent step is to multiply the figure resulting from the subtraction by the scale factor shown in the M_SCALE column.
M_SCALE	DOUBLE	No	Scale factor by which to multiply the figure that results when an offset is subtracted from a measure.

Table 17. Columns in the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view (continued)

Name	Data type	Nullable?	Content
MIN_X	DOUBLE	No	Minimum possible value for X coordinates in the geometries to which this spatial reference system applies. This value is derived from the values in the X_OFFSET and X_SCALE columns.
MAX_X	DOUBLE	No	Maximum possible value for X coordinates in the geometries to which this spatial reference system applies. This value is derived from the values in the X_OFFSET and X_SCALE columns.
MIN_Y	DOUBLE	No	Minimum possible value for Y coordinates in the geometries to which this spatial reference system applies. This value is derived from the values in the Y_OFFSET and Y_SCALE columns.
MAX_Y	DOUBLE	No	Maximum possible value for Y coordinates in the geometries to which this spatial reference system applies. This value is derived from the values in the Y_OFFSET and Y_SCALE columns.
MIN_Z	DOUBLE	No	Minimum possible value for Z coordinates in geometries to which this spatial reference system applies. This value is derived from the values in the Z_OFFSET and Z_SCALE columns.
MAX_Z	DOUBLE	No	Maximum possible value for Z coordinates in geometries to which this spatial reference system applies. This value is derived from the values in the Z_OFFSET and Z_SCALE columns.
MIN_M	DOUBLE	No	Minimum possible value for measures that can be stored with geometries to which this spatial reference system applies. This value is derived from the values in the M_OFFSET and M_SCALE columns.
MAX_M	DOUBLE	No	Maximum possible value for measures that can be stored with geometries to which this spatial reference system applies. This value is derived from the values in the M_OFFSET and M_SCALE columns.
COORDSYS_NAME	VARCHAR(128)	No	Identifying name of the coordinate system on which this spatial reference system is based.
COORDSYS_TYPE	VARCHAR(128)	No	Type of the coordinate system on which this spatial reference system is based.
ORGANIZATION	VARCHAR(128)	Yes	Name of the organization (for example, a standards body) that defined the coordinate system on which this spatial reference system is based. ORGANIZATION is null if ORGANIZATION_COORDSYS_ID is null.
ORGANIZATION_COORDSYS_ID	INTEGER	Yes	Name of the organization (for example, a standards body) that defined the coordinate system on which this spatial reference system is based. ORGANIZATION_COORDSYS_ID is null if ORGANIZATION is null.
DEFINITION	VARCHAR(2048)	No	Well-known text representation of the definition of the coordinate system.
DESCRIPTION	VARCHAR(256)	Yes	Description of the spatial reference system.

DB2GSE.ST_UNITS_OF_MEASURE catalog view

Consult the DB2GSE.ST_UNITS_OF_MEASURE catalog view to determine what units of measure you can choose from.

Certain spatial functions accept or return values that denote a specific distance. In some cases, you can choose what unit of measure the distance is to be expressed in. For example, ST_Distance returns the minimum distance between two specified geometries. On one occasion you might require ST_Distance to return the distance in terms of miles; on another, you might require a distance expressed in terms of meters.

For a description of columns in this view, see the following table.

Table 18. Columns in the DB2GSE.ST_UNITS_OF_MEASURE catalog view

Name	Data type	Nullable?	Content
UNIT_NAME	VARCHAR(128)	No	Name of the unit of measure. This name is unique in the database.
UNIT_TYPE	VARCHAR(128)	No	Type of the unit of measure. Possible values are: LINEAR The unit of measure is linear. ANGULAR The unit of measure is angular.
CONVERSION_FACTOR	DOUBLE	No	Numeric value used to convert this unit of measure to its base unit. The base unit for linear units of measure is METER; the base unit for angular units of measure is RADIAN.
DESCRIPTION	VARCHAR(256)	Yes	The base unit itself has a conversion factor of 1.0. Description of the unit of measure.

DB2GSE.SPATIAL_REF_SYS catalog view

Use the DB2GSE.SPATIAL_REF_SYS catalog view to obtain information about spatial reference systems that are registered in DB2 Spatial Extender .

Important: This catalog view has been deprecated and replaced by the “DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view” on page 114.

For a description of columns in this view, see the following table.

Table 19. 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.
CSID	INTEGER	No	Numeric identifier for the coordinate system that underlies this spatial reference system.
CS_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.

Table 19. Columns in the DB2GSE.SPATIAL_REF_SYS catalog view (continued)

Name	Data Type	Nullable?	Content
SRTEXT	VARCHAR(2048)	No	Annotation text for this spatial reference system.
FALSEX	FLOAT	No	A number that, when subtracted from a negative X coordinate value, leaves a non-negative number (that is, a positive number or a zero).
FALSEY	FLOAT	No	A number that, when subtracted from a negative Y coordinate value, leaves a non-negative number (that is, a positive number or a zero).
XYUNITS	FLOAT	No	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.
FALSEZ	FLOAT	No	A number that, when subtracted from a negative Z coordinate value, leaves a non-negative number (that is, a positive number or a zero).
ZUNITS	FLOAT	No	A number that, when multiplied by a decimal Z coordinate, yields an integer that can be stored as a 32-bit data item.
FALSEM	FLOAT	No	A number that, when subtracted from a negative measure, leaves a non-negative number (that is, a positive number or a zero).
MUNITS	FLOAT	No	A number that, when multiplied by a decimal measure, yields an integer that can be stored as a 32-bit data item.

Chapter 16. DB2 Spatial Extender commands

Use these commands to set up DB2 Spatial Extender and to develop projects that use spatial data.

Invoking commands for setting up DB2 Spatial Extender and developing projects

Use the DB2 Spatial Extender command-line processor (CLP), called **db2se**, to set up Spatial Extender and create projects that use spatial data. This topic explains how to use **db2se** to run DB2 Spatial Extender commands.

Prerequisites

Before you can issue **db2se** commands, you must be authorized to do so. To find out what authorization is required for a given command, see the Authorization section in each command.

Enter **db2se** commands from an operating system prompt.

To find out what **db2se** commands and parameters are available:

- Type **db2se** or **db2se -h**; then press Enter. A list of **db2se** subcommands is displayed.
- Type **db2se** and a command, or **db2se** and a command followed by **-h** parameter. Then press Enter. The syntax required for the subcommand is displayed. In this syntax:
 - Each parameter is preceded by a dash and followed by a placeholder for a parameter value.
 - Parameters enclosed by brackets are optional. The other parameters are required.

To issue a **db2se** command, type **db2se**. Then type a command, followed by the parameters and parameter values that the **db2se** command requires. Finally, press Enter.

You might need to type the user ID and password that give you access to the database that you just specified. For example, type the user ID and password if you want to connect to the database as a user other than yourself. Always precede the user ID with the **-userId** parameter and precede the password with the **-pw** parameter. If you do not specify a user ID and password, your current user ID and password will be used by default.

Values that you enter are not case-sensitive by default. To make them case-sensitive, enclose them in double quotation marks. For example, to specify the lowercase table name **mytable** type "mytable".

You might have to escape the quotation marks to ensure they are not interpreted by the system prompt (shell), for example, specify \"mytable\" to refer to the table called **mytable**. If a case-sensitive value is qualified by another case-sensitive value, delimit the two values individually; for example, "myschema"."mytable" Enclose strings in double quotation marks, as follows: "select * from newtable"

Important: Do not include the entire list of parameter names and values in double quotation marks.

Most of the **db2se** commands run a corresponding stored procedure on the DB2 server with the exception of the **db2se shape_info** command. This command runs on the client and might access coordinate and spatial reference system information from a spatially enabled database.

The **db2se import_shape** and **db2se export_shape** command can also run on the client. This is convenient as it allows access to local files on the client.

db2se alter_cs command

The **db2se alter_cs** command updates a coordinate system definition.

You can use this command to update the definition string, the organization name, the organization coordinate system identifier, and the description in a coordinated system defined by the **db2se create_cs** command or the ST_CREATE_COORDSYS stored procedure. The information about the coordinate system is available through the DB2GSE.ST_COORDINATE_SYSTEMS catalog view.

Authorization

The user ID must have DBADM and DATAACCESS authority on the spatially enabled database to run this command.

Command syntax

db2se alter_cs command

```
db2se> alter_cs database_name [-userId user_id [-pw password]]
db2se> --coordsysName coordsys_name [-definition def_string]
db2se> [-organization org_string [-organizationCoordsysId org_cs_id]]
db2se> [-description description_string]
```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to alter the coordinate system.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-coordsysName *coordsys_name*

Uniquely identifies the coordinate system. The maximum length for this parameter is 128 characters.

The *coordsys_name* value is converted to uppercase unless you enclose it in double quotation marks.

-definition *def_string*

Defines the coordinate system. The vendor that supplies the coordinate system usually provides the information for this parameter. The maximum length for this parameter is 2048 characters.

-organization *org_string*

Defines the name of the organization that defined the coordinate system and provided the definition for it; for example, "European Petroleum Survey Group (EPSG)." The maximum length for this parameter is 128 characters.

The combination of the *org_string* and *org_cs_id* uniquely identifies the coordinate system.

-organizationCoordsysId *org_cs_id*

Specifies a numeric identifier. The entity that is specified in *org_string* assigns this value. This value is not necessarily unique across all coordinate systems.

The combination of the *org_string* and *org_cs_id* uniquely identifies the coordinate system.

-description *description_string*

Describes the coordinate system by explaining its application. The maximum length for this parameter is 256 characters.

Usage notes

If you use this command to change the definition of the coordinate system and you have existing spatial data that is associated with a spatial reference system that is based on this coordinate system, you might inadvertently change the spatial data. If spatial data is affected, you are responsible for ensuring that the changed spatial data is still accurate and valid.

Example

The following example updates the definition of a coordinate system named MYCOORDSYS with a new organization name.

```
db2se alter_cs mydb -coordsysName mycoordsys -organization myNeworganizationb
```

db2se alter_srs command

The **db2se alter_srs** command updates a spatial reference system definition.

You can use this command to change the offset, scale, or coordinate system name of a spatial reference system. The information about the coordinate system is available through the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view.

Authorization

The user ID must have DBADM and DATAACCESS authority on the spatially enabled database to run this command.

Command syntax

db2se alter_srs command

```
►►alter_srs—database_name—┐
                             └─userId—user_id—pw—password┘
►--srsName—srs_name—┐
                    └─srsId—srs_id┘┐
                                └─xOffset—x_offset┘
►┐
  └─xScale—x_scale┘┐
                  └─yOffset—y_offset┘┐
                              └─yScale—y_scale┘
►┐
  └─zOffset—z_offset┘┐
                  └─zScale—z_scale┘┐
                              └─mOffset—m_offset┘
►┐
  └─mScale—m_scale┘┐
                  └─coordsysName—coordsys_name┘
►┐
  └─description—description_string┘
```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to update a spatial reference system definition.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-srsName *srs_name*

Identifies the spatial reference system that you want to update. The *srs_name* value is converted to uppercase unless you enclose it in double quotation marks.

-srsId *srs_id*

Uniquely identifies the spatial reference system. This identifier is used as an input parameter for various spatial functions.

-xOffset *x_offset*

Specifies the offset for all X coordinates of geometries that are represented in this spatial reference system. The offset is subtracted before the *x_scale* scale factor is applied when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, well-known text (WKT), well-known binary (WKB), and shape.

-xScale *x_scale*

Specifies the scale factor for all X coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the *x_offset* offset is subtracted when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-yOffset *y_offset*

Specifies the offset for all Y coordinates of geometries that are represented in this spatial reference system. The offset is subtracted before the *y_scale* scale factor is applied when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-yScale *y_scale*

Specifies the scale factor for all Y coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the *y_offset* offset is subtracted when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape. This scale factor must be the same as *x_scale*.

-zOffset *z_offset*

Specifies the offset for all Z coordinates of geometries that are represented in this spatial reference system. The offset is subtracted before the *z_scale* scale factor is applied when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-zScale *z_scale*

Specifies the scale factor for all Z coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the *z_offset* offset is subtracted when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-mOffset *m_offset*

Specifies the offset for all M coordinates of geometries that are represented in this spatial reference system. The offset is subtracted before the *m_scale* scale factor is applied when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-mScale *m_scale*

Specifies the scale factor for all M coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the *m_offset* offset is subtracted when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-coordsysName *coordsys_name*

Uniquely identifies the coordinate system on which this spatial reference system is based. The coordinate system must be listed in the view DB2GSE.ST_COORDINATE_SYSTEMS.

The *coordsys_name* value is converted to uppercase unless you enclose it in double quotation marks. The maximum length for this parameter is 128 characters.

-description *description_string*

Describes the coordinate system by explaining its application. The maximum length for this parameter is 256 characters.

Usage notes

If you use this command to change offset, scale, or *coordsys_name* parameters of the spatial reference system, and if you have existing spatial data that is associated

with the spatial reference system, you might inadvertently change the spatial data. If spatial data is affected, you are responsible for ensuring that the changed spatial data is still accurate and valid.

Example

The following example alters a spatial reference system named MYSRs with a different description.

```
db2se alter_srs mydb -srsName mysrs
      -description "This is my own spatial reference system."
```

The following example alters a spatial reference system named MYSRs_35 with a different xOffset and description because there is no spatial data that uses MYSRs_35. If you have spatial using it, the data becomes unusable.

```
db2se alter_srs mydb -srsName mysrs_35 -xOffset 35
      -description "This is my own spatial reference system with xOffset=35."
```

db2se create_cs command

The **db2se create_cs** command creates a coordinate system.

This command stores information in the database about a new coordinate system. The information about the coordinate system is available through the DB2GSE.ST_COORDINATE_SYSTEMS catalog view.

Authorization

The user ID must have DBADM and DATAACCESS authority on the spatially enabled database to run this command.

Command syntax

db2se create_cs command

```

>> create_cs database_name [ -userId user_id -pw password ]
> --coordsysName coordsys_name --definition def_string
> [ -organization org_string -organizationCoordsysId org_cs_id ]
> [ -description description_string ]

```

Command parameters

Where:

database_name

The name of the database for which you want to create the coordinate system.

-userId *user_id*

The database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

The password for *user_id*.

-coordsysName *coordsys_name*

Uniquely identifies the coordinate system. The maximum length for this parameter is 128.

The *coordsys_name* value is converted to uppercase unless you enclose it in double quotation marks.

-definition *def_string*

Defines the coordinate system. The vendor that supplies the coordinate system usually provides the information for this parameter. The maximum length for this parameter is 2048.

-organization *org_string*

Defines the name of the organization that defined the coordinate system and provided the definition for it; for example, "European Petroleum Survey Group (EPSG)."

The combination of the *org_string* and *org_cs_id* uniquely identifies the coordinate system. The maximum length for this parameter is 128.

-organizationCoordsysId *org_cs_id*

Specifies a numeric identifier. The entity that is specified in *org_string* assigns this value. This value is not necessarily unique across all coordinate systems.

The combination of the *org_string* and *org_cs_id* uniquely identifies the coordinate system.

-description *description_string*

Describes the coordinate system by explaining its application. The maximum length for this parameter is 256.

Usage notes

DB2 Spatial Extender provides over 4000 coordinate systems. Creating a coordinate system is rarely necessary. Also, you must create a spatial reference system based on the new coordinate system.

Example

The following example creates a coordinate system named MYCOORDSYS.

```
db2se create_cs mydb -coordsysName mycoordsys
                    -definition GEOCS["GCS_NORTH_AMERICAN_1983",
                    DATUM["D_North_American_1983",
                    SPHEROID["GRS_1980",6387137,298.257222101]],
                    PRIMEM["Greenwich",0],
                    UNIT["Degree", 0.0174532925199432955]]
```

db2se create_srs command

The **db2se create_srs** command creates a spatial reference system.

You can use this command to create a spatial reference system definition. A spatial reference system is defined by the coordinate system, the precision, and the extents of coordinates that are represented in the specified spatial reference system. The extents are the minimum and maximum coordinate values for the X, Y, Z, and M coordinates. The information about the coordinate system is available through the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view.

- Using conversion factors (offsets and scale factors)
- Using extents and precision (conversion factors are calculated)

The user ID must have DBADM and DATAACCESS authority on the spatially enabled database to run this command.

db2se create_srs command (using conversion factors)

db2se create_srs command (using extents and precision)

Command parameters

126 Spatial Extender User's Guide and Reference

database_name

Specifies the name of the database for which you want to create a spatial reference system definition.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-srsName *srs_name*

Identifies the spatial reference system that you want to create. The *srs_name* value is converted to uppercase unless you enclose it in double quotation marks.

-srsId *srs_id*

Uniquely identifies the spatial reference system. This identifier is used as an input parameter for various spatial functions.

-xMin *x_min*

Specifies the minimum X coordinate value for all geometries that use the specified spatial reference system.

-xMax *x_max*

Specifies the maximum X coordinate value for all geometries that use the specified spatial reference system. Depending on the *x_scale* value, the maximum X coordinate value returned by the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS view might be larger than the value specified by *x_max*. The value returned by the view is correct.

-xOffset *x_offset*

Specifies the offset for all X coordinates of geometries that are represented in the specified spatial reference system. The offset is subtracted before the *x_scale* scale factor is applied when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, well-known text(WKT), well-known binary (WKB), and shape.

-xScale *x_scale*

Specifies the scale factor for all X coordinates of geometries that are represented in the specified spatial reference system. The scale factor is applied (multiplication) after the *x_offset* offset is subtracted when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-yMin *y_min*

Specifies the minimum Y coordinate value for all geometries that use the specified spatial reference system.

-yMax *y_max*

Specifies the maximum Y coordinate value for all geometries that use the specified spatial reference system. Depending on the *y_scale* value, the maximum Y coordinate value returned by the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS view might be larger than the *y_max* value specified. The value returned by the view is correct.

-yOffset *y_offset*

Specifies the offset for all Y coordinates of geometries that are represented in the specified spatial reference system. The offset is subtracted before the *y_scale*

scale factor is applied when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-yScale *y_scale*

Specifies the scale factor for all Y coordinates of geometries that are represented in the specified spatial reference system. The scale factor is applied (multiplication) after the *y_offset* offset is subtracted when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape. This scale factor must be the same as *x_scale*.

-zMin *z_min*

Specifies the minimum Z coordinate value for all geometries that use the specified spatial reference system.

-zMax *z_max*

Specifies the maximum Z coordinate value for all geometries that use the specified spatial reference system. Depending on the *z_scale* value, the maximum Z coordinate value returned by the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS view might be larger than the *z_max* value specified. The value returned by the view is correct.

-zOffset *z_offset*

Specifies the offset for all Z coordinates of geometries that are represented in the specified spatial reference system. The offset is subtracted before the *z_scale* scale factor is applied when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-zScale *z_scale*

Specifies the scale factor for all Z coordinates of geometries that are represented in the specified spatial reference system. The scale factor is applied (multiplication) after the *z_offset* offset is subtracted when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-mMin *m_min*

Specifies the minimum M coordinate value for all geometries that use the specified spatial reference system.

-mMax *m_max*

Specifies the maximum M coordinate value for all geometries that use the specified spatial reference system. Depending on the *m_scale* value, the maximum M coordinate value returned by the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS view might be larger than the *m_max* value specified. The value returned by the view is correct.

-mOffset *m_offset*

Specifies the offset for all M coordinates of geometries that are represented in the specified spatial reference system. The offset is subtracted before the *m_scale* scale factor is applied when geometries are converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-mScale *m_scale*

Specifies the scale factor for all M coordinates of geometries that are represented in the specified spatial reference system. The scale factor is applied (multiplication) after the *m_offset* offset is subtracted when geometries are

converted to the DB2 Spatial Extender internal representation from external representations such as, WKT, WKB, and shape.

-coordsysName *coordsys_name*

Uniquely identifies the coordinate system on which the specified spatial reference system is based. The coordinate system must be listed in the view DB2GSE.ST_COORDINATE_SYSTEMS.

The *coordsys_name* value is converted to uppercase unless you enclose it in double quotation marks. The maximum length for this parameter is 128.

-description *description_string*

Describes the coordinate system by explaining its application. The maximum length for this parameter is 256.

Usage notes

DB2 Spatial Extender includes support for five spatial reference systems and over 4000 coordinate systems that you can choose from. Creating a spatial reference system is rarely necessary.

Example

The following example creates a spatial reference system named MYSRs by using the command syntax for conversion factors.

```
db2se create_srs mydb -srsName mysrs -srsID 100
      -xOffset -180 -xScale 1000000 -yOffset -90
      -coordsysName \"GCS_North_American_1983\"
```

db2se disable_autogc command

The **db2se disable_autogc** command disables autogeocoding.

This command prevents DB2 Spatial Extender from synchronizing a geocoded column with its associated geocoding column or columns. A geocoding column is used as input to the geocoder. Each time that values are inserted into, or updated in, the geocoding column or columns, triggers are activated. These triggers invoke the associated geocoder to geocode the inserted or updated values and to place the resulting data in the geocoded column. The information about geocoded columns is available from the DB2GSE.ST_GEOCODING catalog view.

Authorization

The user ID must have one of the following authorities or privileges to run this command:

- DBADM and DATAACCESS authority on the database that contains the table on which the triggers that are being dropped are defined.
- CONTROL privilege on this table and DROPIN authority on the DB2GSE schema.
- ALTER and UPDATE privileges on this table and DROPIN authority on the DB2GSE schema.

Also, the user ID must have the necessary privileges on the DB2 server to create or write to the exception and messages files.

Command syntax

db2se disable_autogc command

```
db2se disable_autogc database_name [-userId user_id [-pw password] [-tableSchema table_schema] [-tableName table_name] [-columnName column_name]]
```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to disable autogeocoding.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-tableSchema *table_schema*

Specifies the schema name for the specified *table_name*. If you do not specify a schema name, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

-tableName *table_name*

Specifies the unqualified name of the table containing the specified column on which you want to disable autogeocoding. The triggers created to synchronize the geocoded column are dropped. The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

-columnName *column_name*

Specifies the names of the geocoded column that is maintained on which you want to disable autogeocoding. The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

Example

The following example disables the automatic geocoding for a geocoded column named MYCOLUMN in table MYTABLE.

```
db2se disable_autogc mydb -tableName mytable -columnName mycolumn
```

db2se disable_db command

The **db2se disable_db** command removes resources that allow DB2 Spatial Extender to store and support spatial data.

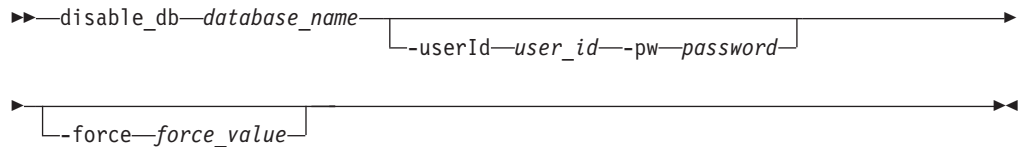
These resources include spatial data types, spatial index types, catalog views, supplied functions, and stored procedures that were created when you enable a database for support of spatial operations.

Authorization

The user ID must have DBADM and DATAACCESS authority on the spatially enabled database to run this command.

Command syntax

db2se disable_db command



Command parameters

Where:

database_name

Specifies the name of the database for which you want to disable DB2 Spatial Extender.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-force *force_value*

Specifies that you want to disable a database for spatial operations, even though you might have database objects that are dependent on the spatial types or spatial functions. Database objects that might have such dependencies include tables, views, constraints, triggers, generated columns, methods, functions, procedures, and other data types (subtypes or structured types with a spatial attribute).

If you specify a non-zero value for the **-force** parameter, the database is disabled and all resources of DB2 Spatial Extender are removed. If you specify zero as *force_value*, the database is disabled only if database objects are independent on spatial types or spatial functions.

Usage notes

If you already defined spatial columns but still want to disable a database for spatial operations, you must specify a value other than 0 (zero) for the force parameter to remove all spatial resources in the database that do not have other dependencies on them.

Example

The following example disables the support for spatial operations on the *MYDB* database even if there are database objects with dependencies to spatial types and functions.

```
db2se disable_db mydb -force 1
```


Authorization

The user ID has DBADM and DATAACCESS authority on the spatially enabled database to run this command.

Command syntax

db2se drop_srs command

```
►► drop_srs database_name [-userId user_id -pw password]
►--srsName srs_name◄◄
```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to delete a spatial reference system definition.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-srsName *srs_name*

Identifies the spatial reference system that you want to delete. The *srs_name* value is converted to uppercase unless you enclose it in double quotation marks.

Usage notes

If you registered spatial columns that use a spatial reference system, you cannot drop that spatial reference system.

Example

The following example drops a spatial reference system named MYSRs.

```
db2se drop_srs mydb -srsName mysrs
```

db2se enable_autogc command

The **db2se enable_autogc** command enables autogeocoding.

This command enables DB2 Spatial Extender to synchronize a geocoded column with its associated geocoding column or columns. A geocoding column is used as input to the geocoder. Each time that values are inserted into, or updated in, the geocoding column or columns, triggers are activated. These triggers invoke the associated geocoder to geocode the inserted or updated values and to place the resulting data in the geocoded column. The information about geocoded columns is available from the DB2GSE.ST_GEOCODING catalog view.

Authorization

The user ID must have one of the following authorities or privileges to run this command:

- DBADM and DATAACCESS authority on the database that contains the table on which the triggers that are created by this stored procedure are defined
- CONTROL privilege on the table
- ALTER privilege on the table

If the authorization ID of the statement does not have DBADM authority, the privileges that the authorization ID of the statement holds (without considering PUBLIC or group privileges) must include all of the following privileges as long as the trigger exists:

- SELECT privilege on the table on which autogeocoding is enabled or DATAACCESS authority.
- Necessary privileges to evaluate the SQL expressions that are specified for the parameters in the geocoding setup.

Command syntax

db2se enable_autogc command

```
►►enable_autogc—database_name—┐———►
                               └--userId—user_id—pw—password—┘

┐———►
└--tableSchema—table_schema—┘—tableName—table_name———►

►--columnName—column_name———►◄◄
```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to enable autogeocoding.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-tableSchema *table_schema*

Specifies the schema name for the specified *table_name*. If you do not specify a schema name, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

-tableName *table_name*

Specifies the unqualified name of the table containing the specified column on which you want to enable autogeocoding. The triggers are created to synchronize the geocoded column. The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

-columnName *column_name*

Identifies the column into which the geocoded data is to be inserted or updated. This column is referred to as the geocoded column. The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

Usage notes

Before enabling autogeocoding, you must perform the geocoding setup step to specifies the geocoder and the geocoding parameter values. It also identifies the geocoding columns that are to be synchronized with the geocoded columns.

You can enable autogeocoding only on tables on which INSERT and UPDATE triggers can be created. Consequently, you cannot enable autogeocoding on views or nicknames.

Example

The following example sets up automatic geocoding for a column named MYCOLUMN in table MYTABLE

```
db2se enable_autogeocoding mydb -tableName mytable -columnName mycolumn
```

db2se enable_db command

The **db2se enable_db** command supplies a database with the resources that it needs to store spatial data and to support spatial operations.

These resources include spatial data types, spatial index types, catalog views, supplied functions, and other stored procedures.

Authorization

The user ID must have DBADM, DATAACCESS, and CTRLACCESS authority on the spatially enabled database to run this command.

Command syntax

db2se enable_db command

```
➤ enable_db—database_name—┐
                             └--userId—user_id--pw—password—┘
└--tableCreationParameters—tc_params—┘
```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to enable DB2 Spatial Extender.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-tableCreationParameters *tc_params*

Specifies parameters for the CREATE TABLE statements that are used to create the DB2 Spatial Extender catalog tables. Use the parameter syntax for the CREATE TABLE statement. The following example in Windows operating system, specifies a table space in which to create the tables and a table space in which to create the table indexes:

`-tableCreationParameters "IN TS_name INDEX IN IDX_TS_name"`

The maximum length for this parameter is 32,672 characters.

Usage notes

Ensure that you have a system temporary table space with a page size of 8 KB or larger and with a minimum size of 500 pages. This is a requirement to run the **db2se enable_db** command successfully.

Example

The following example enables a database named MYDB for spatial operations.

```
db2se enable_db mydb
```

db2se export_shape command

The **db2se export_shape** command exports a spatial column and its associated table to a shape file.

You can use this command to create a spatial reference system definition. A spatial reference system is defined by the coordinate system, the precision, and the extents of coordinates that are represented in the specified spatial reference system. The extents are the minimum and maximum possible coordinate values for the X, Y, Z, and M coordinates. The information about the coordinate system is available through the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view.

Authorization

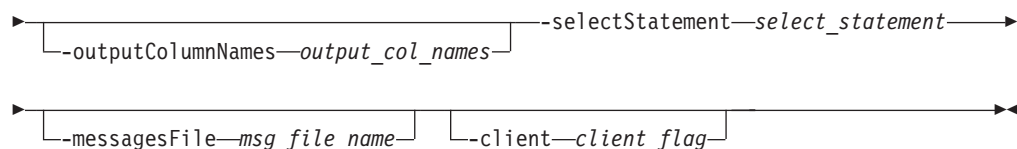
The user must have the necessary privileges to successfully execute the SELECT statement from which the data is to be exported.

Also, the DB2 instance owner ID must have the necessary privileges on the DB2 server to create or write to the shape files and messages file.

Command syntax

db2se export_shape command

```
►►—export_shape—database_name—┐
                                └—-userId—user_id—-pw—password—┘
►—-fileName—file_name—┐
                       └—-appendFlag—append_flag—┘
```

Command parameters

Where:

database_name

Specifies the name of the database containing the table to be exported.

```
-userId user_id
```

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-fileName *file name*

Specifies the full path name of a shape file to which the specified data is to be exported. See “Usage notes” on page 138 for a complete list of files that are written on the DB2 server. The maximum length for this parameter is 256.

If you are exporting to a new file, you can specify the optional file extension as .shp or .SHP. If you specify .shp or .SHP as the file extension, DB2 Spatial Extender creates the file with the specified *file_name* value. If you do not specify the optional file extension, DB2 Spatial Extender creates the file with the name *file_name.shp*.

If you are exporting data by appending the data to an existing file, DB2 Spatial Extender first looks for an exact match of the name that you specify with the **-fileName** parameter. If DB2 Spatial Extender does not find an exact match, it looks first for a file with the .shp extension, and then for a file with the .SHP extension. If the *append_flag* value indicates that you are not appending to an existing file, but the file already exists, DB2 Spatial Extender returns an error and does not overwrite the file.

-appendFlag *append_flag*

Indicates whether the data that is to be exported is to be appended to an existing shape file. The possible values for this parameter are:

- A non-zero value in *append_flag* to indicate that you want to append data to an existing shape file. If the existing file structure does not match the exported data, an error is returned.
- A value of 0 in *append_flag* to indicate that you want to export to a new file. DB2 Spatial Extender does not overwrite any existing files.

-outputColumnNames *output col names*

Specifies one or more column names (separated by commas) that are used for non-spatial columns in the output dBASE file. If this parameter is not specified, the column names from the SELECT statement are used.

If the column names are not enclosed in double quotation marks, the column names are converted to uppercase. The number of specified columns must match the number of columns that are returned from the SELECT statement indicated in the *select_statement* parameter, excluding the spatial column.

The maximum length for this parameter is 32 672 characters.

-selectStatement *select_statement*

Specifies the subselect that returns the data that is to be exported. The subselect must reference exactly one spatial column and any number of attribute columns. The maximum length for this parameter is 32 672 characters.

-messagesFile *msg_file_name*

Specifies the full path name of the file in the DB2 server to which DB2 Spatial Extender writes messages about the export operation. If you specify this parameter and the file already exists, an error is returned and the export operation terminated. If you do not specify this parameter, DB2 Spatial Extender does not create a messages file.

The following type of messages are written to the messages file:

- Informational messages, such as a summary of the export operation
- Error messages for data that could not be exported, for example because of different coordinate systems

The maximum length for this parameter is 256.

-client *client_flag*

Specifies whether the export operation takes place on the client or the DB2 server and where the files are created. The possible values for this parameter are:

- 0 to indicate the export operation takes place on the DB2 server and the files are created on the DB2 server.
- 1 to indicate the export operation takes place on the client and the files are created on the client.

If you do not specify this parameter, the 0 value is the default.

Usage notes

You can export only one spatial column at a time.

You can perform the export process on the client where the command is executed. This is often more convenient as it does not require access to the DB2 server file system.

The **db2se export_shape** creates or writes to the following four files:

- The main shape file (.shp extension).
- The shape index file (.shx extension).
- A dBASE file that contains data for non-spatial columns (.dbf extension). This file is created only if attribute columns actually need to be exported
- A projection file that specifies the coordinate system that is associated with the spatial data, if the coordinate system is not equal to "UNSPECIFIED" (.prj extension). The coordinate system is obtained from the first spatial record. An error occurs if subsequent records have different coordinate systems.

The following table describes how DB2 data types are stored in dBASE attribute files. All other DB2 data types are not supported.

Table 20. Storage of DB2 data types in attribute files

SQL type	.dbf type	.dbf length	.dbf decimals	Comments
SMALLINT	N	6	0	
INTEGER	N	11	0	

Table 20. Storage of DB2 data types in attribute files (continued)

SQL type	.dbf type	.dbf length	.dbf decimals	Comments
BIGINT	N	20	0	
DECIMAL	N	precision+2	scale	
REAL FLOAT(1) through FLOAT(24)	F	14	6	
DOUBLE FLOAT(25) through FLOAT(53)	F	19	9	
CHARACTER, VARCHAR, LONG VARCHAR, and DATALINK	C	<i>len</i>	0	length ≤ 255
DATE	D	8	0	
TIME	C	8	0	
TIMESTAMP	C	26	0	

All synonyms for data types and distinct types that are based on the types listed in the preceding table are supported.

Example

The following example exports a spatial column named MYCOLUMN and its associated table, MYTABLE, to the myshapefile shape file that is located in the client.

```
db2se export_shape mydb -fileName /home/myaccount/myshapefile
      -selectStatement "select * from mytable" -client 1
```

db2se import_shape command

The **db2se import_shape** command imports a shape file to a database that is enabled for spatial operations.

This command can import shape and attribute data into an existing table or a new table.

Authorization

The DB2 instance owner ID must have the necessary privileges on the DB2 server to create or write to the exception and messages files.

The user ID must have additional authorization requirements to run this command. The requirements vary depending on whether you are importing into an existing table or into a new table.

Requirements to import into an existing table

The user ID must hold one of the following authorities or privileges:

- DATAACCESS
- CONTROL privilege on the table or view
- INSERT and SELECT privilege on the table or view

Requirements to import into a new table

The user ID must hold one of the following authorities or privileges:

- DBADM and DATAACCESS
- CREATETAB authority on the database

- IMPLICIT_SCHEMA authority on the database, if the schema name of the table does not exist
- CREATEIN privilege on the schema, if the schema of the table exists

Command syntax

db2se import_shape command

```

▶▶--import_shape—database_name—┐
└─-userId—user_id—-pw—password—┘
▶--fileName—file_name—┐
└─-inputColumnNames—input_col_names—┘
▶--srsName—srs_name—┐
└─-tableSchema—table_schema—┘
└─-tableName—table_name—┘
▶┐
└─-tableAttrColumns—attr_columns—┘
└─-createTableFlag—create_flag—┘
▶┐
└─-tableCreationParameters—tc_parameters—┘
└─-spatialColumn—spatial_column—┘
▶┐
└─-typeSchema—type_schema—┘
└─-typeName—type_name—┘
▶┐
└─-inlineLength—in_length—┘
└─-idColumn—id_column—┘
▶┐
└─-idColumnIsIdentity—id_flag—┘
└─-restartCount—rs_count—┘
▶┐
└─-commitScope—commit_count—┘
└─-exceptionFile—e_file_name—┘
▶┐
└─-messagesFile—msg_file_name—┘
└─-client—client_flag—┘
▶▶

```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to import the shape file.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-fileName *file_name*

Specifies the full path name of a shape file to which the specified data is to be imported. If you specify .shp or .SHP as the file extension, DB2 Spatial Extender first looks for an exact match of the name that you specify with the **-fileName** parameter. If DB2 Spatial Extender does not find an exact match, it

looks first for a file with the .shp extension, and then for a file with the .SHP extension. See “Usage notes” on page 144 for a complete list of files that are written on the DB2 server.

The maximum length for this parameter is 256 characters.

-inputColumnNames *input_col_names*

Specifies a list of attribute columns to import from the dBASE file. If this parameter is not specified, all columns in the file are imported. Use any of the following formats to specify a list of attributes:

- A comma-separated list of column names to be imported from the dBASE file as shown in the following example:

N(COLUMN1,COLUMN5,COLUMN3,COLUMN7)

If the column names are not enclosed in double quotation marks, the column names are converted to uppercase. The resulting names must exactly match the column names in the dBASE file.

- A comma-separated list of column numbers to be imported from the dBASE file as shown in the following example:

P(1,5,3,7)

Columns are numbered beginning with 1. Each number in the list must be separated by a comma.

- An empty string "" to indicate that no attribute data is to be imported.

The maximum length for this parameter is 32,672 characters.

-srsName *srs_name*

Identifies the spatial reference system (SRS) to be used for the geometries that are imported into the spatial column. The *srs_name* value is converted to uppercase unless you enclose it in double quotation marks.

The spatial column is not registered. The SRS must exist before the data is imported. The import process does not implicitly create the SRS, but it does compare the coordinate system of the SRS with the coordinate system that is specified in the .prj file (if this file is available with the shape file).

The import process also verifies that the extents of the data in the shape file can be represented in the specified SRS. That is, the import process verifies that the extents lie within the minimum and maximum X, Y, Z, and M coordinates of the SRS.

-tableSchema *table_schema*

Specifies the schema name for the specified *table_name*. If you do not specify a schema name, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

-tableName *table_name*

Specifies the unqualified name of the table into which the data in the shape file is to be imported. The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

-tableAttrColumns *attr_columns*

Specifies the table column names where attribute data from the dBASE file is to be stored. If this parameter is not specified, the names of the columns in the dBASE file are used.

The number of specified columns must match the number of columns to be imported from the dBASE file. If the table exists, the column definitions must match the incoming data. See “Usage notes” on page 144 for an explanation of how attribute data types are mapped to DB2 data types.

If the column names are not enclosed in double quotation marks, the column names are converted to uppercase. The maximum length for this parameter is 32,672 characters.

-createTableFlag *create_flag*

Specifies whether the import process is to create a table. The possible values for this parameter are:

- A non-zero value in *create_flag* to create a table. If the table exists, an error is returned.
- A value of 0 in *create_flag* to use an existing table.

If this parameter is not specified, a new table is created.

-tableCreationParameters *tc_params*

Specifies any options that are to be added to the CREATE TABLE statement that creates the specified *table_name*.

To specify any CREATE TABLE options, use the syntax of the CREATE TABLE statement. For example, to specify a table space in which to create the tables, indexes, and large objects, specify in *tc_params*:

IN *tsName* INDEX IN *indexTsName* LONG IN *longTsName*

The maximum length for this parameter is 32,672 characters.

-spatialColumn *spatial_column*

Specifies the name of the spatial column in the table into which the shape data is to be imported.

For a new table, this parameter specifies the name of the new spatial column that is to be created. For an existing table, this parameter specifies the name of an existing spatial column in the table.

The *spatial_column* value is converted to uppercase unless you enclose it in double quotation marks.

-typeSchema *type_schema*

Indicates the schema name of the spatial data type specified in the *type_name* value. If this parameter is not specified, DB2GSE is used as the schema name.

The *type_schema* value is converted to uppercase unless you enclose it in double quotation marks.

-typeName *type_name*

Specifies the data type name to be used for the spatial values. If this parameter is not specified, the data type is determined by the shape file from any of the following data types:

- ST_Point
- ST_MultiPoint
- ST_MultiLineString
- ST_MultiPolygon

Shape files, by definition, allow a distinction only between points and multipoints. There is no distinction between polygons and multipolygons or between linestrings and multilinestrings.

If you are importing into a new table, *type_name* data type is also used for the data type of the spatial column. In this case, the data type can also be a super type of ST_Point, ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon.

The *type_name* value is converted to uppercase unless you enclose it in double quotation marks.

-inlineLength *inline_length*

For a new table, specifies the maximum number of bytes that are to be allocated for the spatial column within the table. If this parameter is not specified, the default inline length is used.

Spatial records that exceed the *inline_length* size are stored separately in the LOB table space, which might be slower to access.

The typical sizes that are needed for various spatial types are as follows:

- **One point:** 292 bytes.
- **Multipoint, line, or polygon:** As large a value as possible. Consider that the total number of bytes in one row should not exceed the limit for the page size of the table space for which the table is created.

For a complete description of the *inline_length* value, see the CREATE TABLE statement in the DB2 documentation. Use the ADMIN_EST_INLINE_LENGTH table function to help you estimate the inline length required for geometries in existing tables.

-idColumn *id_column*

Specifies the column name to be created to contain a unique number for each row of data. The unique values for that column are generated automatically during the import process. You must not specify an *id_column* name that matches the name of any column in the dBASE file. Some spatial tools require a column with a unique identifier.

The requirements and effect of this parameter depend on whether the table already exists.

- For an existing table, the data type of the *id_column* parameter can be any integer type such as INTEGER, SMALLINT, or BIGINT.
- For a new table, the column is defined as follows:

```
INTEGER NOT NULL PRIMARY KEY
```

If *id_column_is_identity* has a non-zero value, the definition is expanded as follows:

```
INTEGER NOT NULL PRIMARY KEY GENERATED ALWAYS AS IDENTITY  
( START WITH 1 INCREMENT BY 1 )
```

The *id_column* value is converted to uppercase unless you enclose it in double quotation marks.

-idColumnIsIdentity *id_column_is_identity*

Indicates whether the specified *id_column* is to be created using the IDENTITY clause. If you specify a non-zero value in *id_column_is_identity*, the *id_column* column is created as an identity column. This parameter is ignored for tables that already exist.

-restartCount *restart_count*

Specifies that the import operation starts with record $n + 1$. The first n records are skipped. If this parameter is not specified, all records, starting with record number 1, are imported.

-commitScope *commit_scope*

Specifies that a COMMIT is to be performed after at least n records are imported. If this parameter is not specified, a COMMIT is performed at the end of the operation. This can result in large logfile utilization and data lost on operations that are interrupted.

-exceptionFile *exception_file*

Specifies the full path name of a shape file in which the shape data that could not be imported is written. If the parameter is not specified, an exception file is not created.

If you specify a value for the parameter and include an optional file extension, specify either .shp or .SHP. If the extension you do not specify an extension, the .shp extension is appended to *exception_file*.

The exception file contains the complete set of rows for the insert statement that failed. One insert statement can add a multiple number of rows. For example, assume that one row cannot be imported because the shape data is incorrectly encoded. A single insert statement attempts to import 20 rows, including the one with incorrect shape data. Because the insert statement fails, the entire set of 20 rows is written to the exception file.

Records that are written to the exception file only when those records can be correctly identified, as is the case when the shape record type is not valid. Some types of corruption to the shape data (.shp files) and shape index (.shx files) do not allow the appropriate records to be identified. In this case, records cannot be written to the exception file, and an error message is issued to report the problem.

If you specify a value for this parameter, four files are created on the DB2 server. See "Usage notes" for an explanation these files.

If the *exception_file* file already exists, the command returns an error.

The maximum length for this parameter is 256 characters.

-messagesFile *msg_file_name*

Specifies the full path name of the file in the DB2 server to which DB2 Spatial Extender writes messages about the import operation. If you do not specify this parameter, DB2 Spatial Extender does not create a messages file.

The following type of messages are written to the messages file:

- Informational messages, such as a summary of the import operation
- Error messages for data that could not be imported, for example because of different coordinate systems. These error messages correspond to the shape data that is stored in the specified *exception_file* exception file.

If the *msg_file_name* file already exists, the command returns an error.

The maximum length for this parameter is 256 characters.

-client *client_flag*

Specifies whether the import operation takes place on the client or the DB2 server and where the files are created. The possible values for this parameter are:

- 0 to indicate the import operation takes place on the DB2 server and the files are accessed from the DB2 server.
- 1 to indicate the import operation takes place on the client and the files are accessed from the client.

If you do not specify this parameter, the 0 value is the default.

Usage notes

You can perform the import process on the client where the command is executed. This is often more convenient as it does not require access to the DB2 server file system.

The **db2se import_shape** creates or writes to the following four files:

- The main shape file (.shp extension). This file is required.
- The shape index file (.shx extension). This file is optional. If it is present, performance of the import operation might improve.
- A dBASE file that contains attribute data (.dbf extension). This file is required only if attribute data is to be imported.
- The projection file that specifies the coordinate system of the shape data (.prj extension). This file is optional. If this file is present, the coordinate system that is defined in it is compared with the coordinate system of the spatial reference system that is specified by the *srs_id* parameter.

The following table describes how dBASE attribute data types are mapped to DB2 data types. All other attribute data types are not supported.

Table 21. Relationship between DB2 data types and dBASE attribute data types

.dbf type	.dbf length ¹	.dbf decimals ²	SQL type	Comments
N	< 5	0	SMALLINT	
N	< 10	0	INTEGER	
N	< 20	0	BIGINT	
N	<i>len</i>	<i>dec</i>	DECIMAL(<i>len</i> , <i>dec</i>)	<i>len</i> <32
F	<i>len</i>	<i>dec</i>	REAL	<i>len</i> + <i>dec</i> < 7
F	<i>len</i>	<i>dec</i>	DOUBLE	
C	<i>len</i>		CHAR(<i>len</i>)	
L			CHAR(1)	
D			DATE	

Note: This table includes the following variables, both of which are defined in the header of the dBASE file:

1. *len*, which represents the total length of the column in the dBASE file. DB2 Spatial Extender uses this value for two purposes:
 - To define the precision for the SQL data type DECIMAL or the length for the SQL data type CHAR
 - To determine which of the integer or floating-point types is to be used
2. *dec*, which represents the maximum number of digits to the right of the decimal point of the column in the dBASE file. DB2 Spatial Extender uses this value to define the scale for the SQL data type DECIMAL.

For example, assume that the dBASE file contains a column of data whose length (*len*) is defined as 20. Assume that the number of digits to the right of the decimal point (*dec*) is defined as 5. When DB2 Spatial Extender imports data from that column, it uses the values of *len* and *dec* to derive the following SQL data type: DECIMAL(20,5).

Example

The following command imports the data from the myfile shape file located in the client to the MYTABLE table. The spatial data in myfile is inserted into the MYCOLUMN column in the MYTABLE table.

```
db2se import_shape mydb -fileName myfile -srsName NAD83_SRS_1
      -tableName mytable -spatialColumnName mycolumn -client 1
```

db2se register_gc command

The **db2se register_gc** command registers a geocoder.

This command registers a geocoder so that it can be used then to geocode values in a table and store or update the resulting geometry value. The information about registered geocoders is available from the DB2GSE.ST_GEOCODERS catalog view.

Authorization

The user ID must have DBADM and DATAACCESS authority on the spatially enabled database to run this command.

Command syntax

db2se register_gc command

```
➤➤register_gc—database_name—┐
                               └--userId—user_id—pw—password┘➤
➤--geocoderName—geocoder_name—┐
                               └--functionSchema—function_schema┘➤
➤--functionName—function_name—┐➤
                               └┐
                               └--defaultParameterValues—default_param_values┘➤
➤┐➤
└--parameterDescriptions—parameter_descriptions┘└--vendor—vendor_name┘➤
➤┐➤
└--description—description_string┘➤➤
```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to register a geocoder.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-geocoderName *geocoder_name*

Uniquely identifies the geocoder that you want to register. The *geocoder_name* value is converted to uppercase unless you enclose it in double quotation marks. The maximum length for this parameter is 128 characters.

-functionSchema *function_schema*

Specifies the schema name for the function that implements this geocoder. If this parameter is not specified, the value in the CURRENT SCHEMA special register is used as the schema name for the function.

The *function_schema* value is converted to uppercase unless you enclose it in double quotation marks.

-functionName *function_name*

Specifies the unqualified name of the function that implements this geocoder. The function must already be created and listed in the SYSCAT.ROUTINES catalog view.

The *function_name* value, together with the implicitly or explicitly defined *function_schema* value, must uniquely identify the function.

The *function_name* value is converted to uppercase unless you enclose it in double quotation marks.

-defaultParameterValues *default_parameter_values*

Specifies the list of default geocoding parameter values for the geocoder function.

You must specify the parameter values in the order that the function defined them, and separate them with a comma. For example:

default_parm1_value, default_parm2_value, ...

Each parameter value must be an SQL expression. Follow these guidelines to specify default parameter values:

- If a value is a string, enclose it in single quotation marks.
- If a parameter value is a number, do not enclose it in single quotation marks.
- If the parameter value is null, cast it to the correct type. For example, specify the following expression to indicate a NULL for an integer parameter:
CAST(NULL AS INTEGER)
- If the geocoding parameter is to be a geocoding column, do not specify a default parameter value.

Use two consecutive commas (*...,...*) to omit default values for parameters that you indicate when you set up geocoding or run geocoding in batch mode by using the **-parameterValues** parameter with the **db2se setup_gc** command or the **db2se run_gc** command.

The maximum length for this parameter is 32,672 characters.

-parameterDescriptions *parameter_descriptions*

Specifies the list of geocoding parameter descriptions for the geocoder function. The maximum length for this parameter is 32,672 characters.

Each parameter description that you specify explains the meaning and usage of the parameter, and can be up to 256 characters long. The descriptions for the parameters must be separated by commas and must appear in the order of the parameters as defined by the function. To use a comma within the description of a parameter, enclose the string in single or double quotation marks. For example:

description, 'description2, which contains a comma', description3

-vendor *vendor_name*

Specifies the name of the vendor who implemented the geocoder. The maximum length for this parameter is 128 characters.

-description *description_string*

Describes the geocoder by explaining its application. The maximum length for this parameter is 256 characters.

Usage notes

The return type for the geocoder function must match the data type of the geocoded column. The geocoding parameters can be either a column name (called a geocoding column) which contains data that the geocoder needs. For example, the geocoder parameters can identify addresses or a value of particular meaning to the geocoder, such as the minimum match score. If the geocoding parameter is a column name, the column must be in the same table or view as the geocoded column.

The return type for the geocoder function serves as the data type for the geocoded column. The return type can be any DB2 data type, user-defined type, or structured type. If a user-defined type or structured type is returned, the geocoder function is responsible for returning a valid value of the respective data type. If the geocoder function returns values of a spatial type, that is ST_Geometry or one of its subtypes, the geocoder function is responsible for constructing a valid geometry. The geometry must be represented using an existing spatial reference system. The geometry is valid if you invoke the ST_IsValid spatial function on the geometry and a value of 1 is returned. The returned data from the geocoder function is updated in or is inserted into the geocoded column, depending on which operation (INSERT or UPDATE) caused the generation of the geocoded value.

Example

The following example registers a geocoder named “mygeocoder”, which is implemented by a function named “myschema.myfunction”.

```
db2se register_gc mydb -geocoderName \"mygeocoder\" \
    -functionSchema \"myschema\" -functionName \"myfunction\" \
    -defaultParameterValues \"1, 'string',,cast(null as varchar(50))\" \
    -vendor myvendor -description \"myvendor geocoder \
    returning well-known text\"
```

db2se register_spatial_column command

The **db2se register_spatial_column** command registers a spatial column and associates it to a spatial reference system (SRS).

Registering a spatial column creates a constraint on the table, if possible, to ensure that all geometries use the specified SRS.

Also, you can use this command to update the spatial extent information.

The information about registered spatial columns and spatial extents is available from the DB2GSE.ST_GEOMETRY_COLUMNS catalog view.

Authorization

The user ID must hold one of the following authorities or privileges to run this command:

- DBADM and DATAACCESS authority on the database that contains the table to which the spatial column that is being registered belongs
- CONTROL privilege on this table
- ALTER privilege on this table

Command syntax

db2se register_spatial_column command

```
►►register_spatial_column—database_name—————►
►
┌-userId—user_id—pw—password—┐┌-tableSchema—table_schema—┐►
└──────────────────────────┘└──────────────────────────┘
►-tableName—table_name—columnName—column_name—srsName—srs_name————►
►
┌-computeExtents—ce_value—┐—————►◄
```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to registers a spatial column.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-tableSchema *table_schema*

Specifies the schema name for the specified *table_name*. If you do not specify a schema name, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

-tableName *table_name*

Specifies the unqualified name of the table that contains the column that is being registered. The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

-columnName *column_name*

Identifies the column to be registered. The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

-srsName *srs_name*

Identifies the spatial reference system that is to be used for this spatial column. The *srs_name* value is converted to uppercase unless you enclose it in double quotation marks.

computeExtents *ce_value*

Indicates whether to compute the geographic extents of a specified column and make them available through the DB2GSE.ST_GEOMETRY_COLUMNS catalog view. The possible values for this parameter are:

- A value greater than 0 to compute the geographic extents.
- Null, 0, or negative value to prevent this computation.

If you omit this parameter, it has the same effect as specifying 0. The extent computation is not performed.

Example

The following example registers a spatial column named MYCOLUMN in table MYTABLE, with spatial reference system "USA_SRS_1".

```
db2se register_spatial_column mydb
      -tableName mytable -columnName mycolumn -srsName USA_SRS_1
```

db2se remove_gc_setup command

The **db2se remove_gc_setup** command removes all the geocoding setup information for a geocoded column.

The information that is associated with the specified geocoded column is no longer available from the DB2GSE.ST_GEOCODING and DB2GSE.ST_GEOCODING_PARAMETERS catalog views.

Authorization

The user ID must hold one of the following authorities or privileges to run this command:

- DATAACCESS authority on the database that contains the table on which the specified geocoder is to operate
- CONTROL privilege on this table
- UPDATE privilege on this table

Command syntax

db2se remove_gc_setup command

```
➤--remove_gc_setup--database_name-----➤
                        |--userId--user_id--pw--password--|
➤
|--tableSchema--table_schema--|--tableName--table_name-----➤
|--columnName--column_name-----➤
```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to remove all the geocoding setup information for a geocoded column.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-tableSchema *table_schema*

Specifies the schema name for the specified *table_name*. If you do not specify a schema name, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

-tableName *table_name*

Specifies the unqualified name of the table for the specified *column_name*. The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

-columnName *column_name*

Identifies the column name from which you want to remove the geocoding setup. The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

Usage notes

You cannot remove a geocoding setup if autogeocoding is enabled for the geocoded column.

Example

The following example removes a setup for geocoding operations that apply to a spatial column named MYCOLUMN in table MYTABLE.

```
db2se remove_gc_setup mydb -tableName mytable -columnName mycolumn
```

db2se restore_indexes command

The **db2se restore_indexes** command restores the spatial indexes that you previously saved by issuing the **db2se save_indexes** command to a spatially enabled database.

This command is used to re-create spatial indexes after upgrading from a 32-bit to a 64-bit instance.

Authorization

DBADM and DATAACCESS authority on the spatially enabled database.

Command syntax

db2se restore_indexes command

```
db2se restore_indexes database_name
[-userId user_id] [-pw password] [-messagesFile messages_filename]
```

Command parameters

database_name

The name of the database to be upgraded.

-userId user_id

The database user ID which has either SYSADM or DBADM authority on the database that is being upgraded.

-pw password

Your user password.

-messagesFile messages_filename

The file name containing the report of migration actions. The file name you provide must be a fully-qualified file name on the server.

db2se save_indexes command

The db2se save_indexes command saves the spatial indexes that are defined in a spatially enabled database.

This command is used to save current spatial index definitions when upgrading from a 32-bit to a 64-bit instance.

Authorization

DBADM and DATAACCESS authority on the spatially enabled database.

Command syntax

db2se save_indexes command

```
db2se save_indexes database_name  
-userId user_id -pw password -messagesFile messages_filename
```

Command parameters

database_name

The name of the database to be upgraded.

-userId user_id

The database user ID which has either SYSADM or DBADM authority on the database that is being upgraded.

-pw password

Your user password.

-messagesFile messages_filename

The file name containing the report of migration actions. The file name you provide must be a fully-qualified file name on the server.

db2se run_gc command

The db2se run_gc command runs a geocoder in batch mode on a geocoded column.

The information that is associated with the specified geocoded column is no longer available from the DB2GSE.ST_GEOCODING and DB2GSE.ST_GEOCODING_PARAMETERS catalog views.

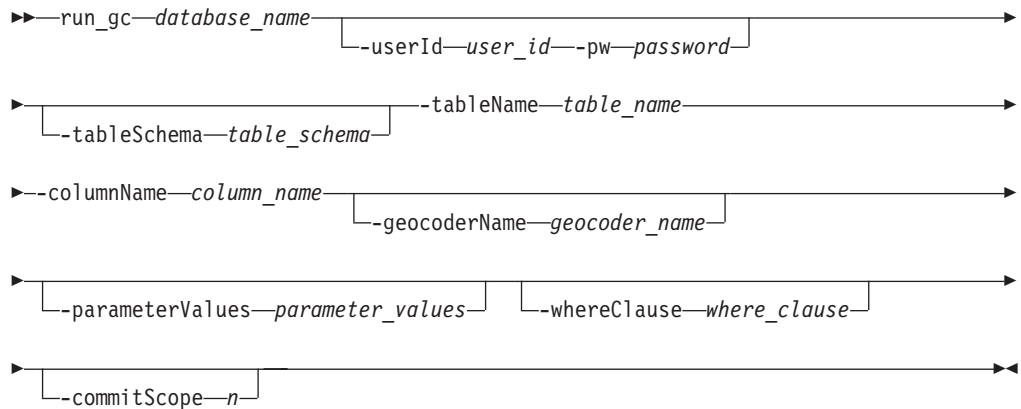
Authorization

The user ID must hold one of the following authorities or privileges to run this command:

- DATAACCESS authority on the database that contains the table on which the specified geocoder is to operate
- CONTROL privilege on this table
- UPDATE privilege on this table

Command syntax

db2se run_gc command



Command parameters

Where:

database_name

Specifies the name of the database for which you want to run a geocoder in batch mode on a geocoded column.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-tableSchema *table_schema*

Specifies the schema name for the specified *table_name*. If you do not specify a schema name, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

-tableName *table_name*

Specifies the unqualified name of the table for the specified *column_name*. The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

-columnName *column_name*

Identifies the column name into which the geocoded data is to be inserted or updated. The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

-geocoderName *geocoder_name*

Uniquely identifies the geocoder that is to perform the geocoding. The *geocoder_name* value is converted to uppercase unless you enclose it in double quotation marks. The maximum length for this parameter is 128 characters.

-parameterValues *parameter_values*

Specifies the list of geocoding parameter values for the geocoder function. If this parameter is not specified, the values that are used are either the parameter values that were specified when you set up the geocoder or the default parameter values that were specified when you register the geocoder.

You must specify the parameter values in the order that the function defined them, and separate them with a comma. For example:

default_parm1_value, default_parm2_value, ...

Each parameter value must be an SQL expression. Follow these guidelines to specify default parameter values:

- If a value is a string, enclose it in single quotation marks.
- If a parameter value is a number, do not enclose it in single quotation marks.
- If the parameter value is null, cast it to the correct type. For example, specify the following expression to indicate a NULL for an integer parameter: `CAST(NULL AS INTEGER)`.
- If the geocoding parameter is to be a geocoding column, do not specify a default parameter value.

Use two consecutive commas (`...,...`) to omit values for parameters that you indicated a value when you set up or register the geocoder.

The maximum length for this parameter is 32,672 characters.

-whereClause *where_clause*

Specifies the text for a search condition of a WHERE clause to filter the set of records to be geocoded. If this parameter is not specified, the *where_clause* value specified during the geocoding setup is used. If a *where_clause* value was not specified during the geocoding setup, all the rows in the table are geocoded.

You can specify a clause that references any column in the table or view that the geocoder is to operate on.

Do not specify the keyword WHERE in *where_clause*.

The maximum length for this parameter is 32,672 characters.

-commitScope *n*

Specifies that a COMMIT is to be performed after every *n* records are geocoded. If this parameter is not specified, the value specified with the **-commitScope** parameter during the geocoding setup is used. If this parameter is not specified and a value was not specified during the geocoding setup, a COMMIT is performed at the end of the operation. One COMMIT at the end of the operation can result in large log file utilization and data lost on operations that are interrupted.

Example

The following example runs a geocoder in batch mode to populate a column named MYCOLUMN in a table named MYTABLE.

```
db2se run_gc mydb -tableName mytable -columnName mycolumn
```

db2se setup_gc command

The **db2se setup_gc** command associates a column that is to be geocoded with a geocoder and to set up the corresponding geocoding parameters.

The information about this setup is available from the DB2GSE.ST_GEOCODING and DB2GSE.ST_GEOCODING_PARAMETERS catalog views.

Authorization

The user ID must hold one of the following authorities or privileges to run this command:

- DATAACCESS authority on the database that contains the table on which the specified geocoder is to operate
- CONTROL privilege on this table
- UPDATE privilege on this table

Command syntax

db2se setup_gc command

```

>> --setup_gc--database_name--
      |--userId--user_id--pw--password--
>
> |--tableSchema--table_schema-- --tableName--table_name--
>
> --columnName--column_name-- --geocoderName--geocoder_name--
>
> |--parameterValues--parameter_values--
>
> |--autogeocodingColumns--auto_gc_columns-- |--whereClause--where_clause--
>
> |--commitScope--n--
>>

```

Command parameters

Where:

database name

Specifies the name of the database for which you want to set up a geocoder.

-userId *user id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-tableSchema *table_schema*

Specifies the schema name for the specified *table_name*. If you do not specify a schema name, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

-tableName *table name*

Specifies the unqualified name of the table for the specified *column_name*. The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

-columnName *column name*

Identifies the column name into which the geocoded data is to be inserted or updated. The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

-geocoderName *geocoder_name*

Uniquely identifies a previously registered geocoder that is to perform the geocoding. The *geocoder_name* value is converted to uppercase unless you enclose it in double quotation marks. The maximum length for this parameter is 128 characters.

-parameterValues *parameter_values*

Specifies the list of geocoding parameter values for the geocoder function. If this parameter is not specified, the default parameter values that were specified when you register the geocoder are used.

You must specify the parameter values in the order that the function defined them, and separate them with a comma. For example:

default_parm1_value,default_parm2_value,...

Each parameter value must be an SQL expression. Follow these guidelines to specify default parameter values:

- If a value is a string, enclose it in single quotation marks.
- If a parameter value is a number, do not enclose it in single quotation marks.
- If the parameter value is null, cast it to the correct type. For example, specify the following expression to indicate a NULL for an integer parameter:
CAST(NULL AS INTEGER).
- If the geocoding parameter is to be a geocoding column, do not specify a default parameter value.

Use two consecutive commas (... ,...) to omit values for parameters that you indicate when you set up or register the geocoder.

The maximum length for this parameter is 32,672 characters.

-whereClause *where_clause*

Specifies the text for a search condition of a WHERE clause to filter the set of records to be geocoded. If this parameter is not specified, all the rows in the table are geocoded.

You can specify a clause that references any column in the table or view that the geocoder is to operate on.

Do not specify the keyword WHERE in *where_clause*.

The maximum length for this parameter is 32,672 characters.

-commitScope *n*

Specifies that a COMMIT is to be performed after every *n* records are geocoded. If this parameter is not specified, a COMMIT is performed at the end of the operation. One COMMIT at the end of the operation can result in large log file utilization and data lost on operations that are interrupted.

Usage notes

This command does not invoke geocoding. It provides a way for you to specify parameter settings for the column that is to be geocoded. Parameter settings that are specified in geocoding setup override any of the default parameter values specified in the geocoder registration.

You must run this command before enabling autogeocoding. Setting up the geocoding parameters is required before enabling autogeocoding.

Example

```
db2se setup_gc mydb -tableName mytable -columnName mycolumn
      -parameterValues "address,city,state,zip,2,90,70,20,1.1,'meter',4.."
      -autogeocodingColumns address,city,state,zip
      -commitScope 10
```

The **db2se shape_info** command shows information about a shape file and its contents. For a specified database, this command can optionally display all compatible coordinate systems and spatial reference systems that are included in the shape file.

The user ID must have the necessary privileges on the DB2 server to read the shape files.

Command syntax

```

>>shape_info--fileName--file_name--database--database_name
>
--userId--user id--pw--password
<<

```

Specifies the full path name of the shape file that you want to display information about.

The maximum length for this parameter is 256 characters.

Specifies the name of the database for which you want to find all compatible coordinate systems and spatial reference systems included in the *file_name* shape file.

-userId *user_id*
 Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*
 Specifies the password for *user_id*.

Example

The following example shows information about a shape file named MYFILE, which is located in the current directory.

```
db2se shape_info -fileName myfile
```

The following example shows information about a sample UNIX shape file named offices. The *-database* parameter finds all compatible coordinate systems and spatial reference systems in the named database (in this case, MYDB).

```
db2se shape_info -fileName ~/sqllib/samples/extenders/spatial/data/offices
                  -database myDB
```

db2se unregister_gc command

The **db2se unregister_gc** command unregisters a geocoder.

To find information about the geocoder that you want to unregister, consult the DB2GSE.ST_GEOCODERS catalog view.

Authorization

The user ID must have DBADM and DATAACCESS authority on the spatially enabled database to run this command.

Command syntax

db2se unregister_gc command

```

▶▶--unregister_gc--database_name-----┐
                                     └--userId--user_id--pw--password--┘
▶--geocoderName--geocoder_name-----▶▶

```

Command parameters

Where:

database_name
 Specifies the name of the database for which you want to unregister a geocoder.

-userId *user_id*
 Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*
 Specifies the password for *user_id*.

-geocoderName *geocoder_name*
 Uniquely identifies the geocoder that you want to unregister. The

geocoder_name value is converted to uppercase unless you enclose it in double quotation marks. The maximum length for this parameter is 128 characters.

Usage notes

You cannot unregister a geocoder if it is specified in the geocoding setup for any column. To determine whether a geocoder is specified in the geocoding setup for a column, check the DB2GSE.ST_GEOCODING and DB2GSE.ST_GEOCODING_PARAMETERS catalog views.

Example

The following example unregisters a geocoder named MYGEOCODER.

```
db2se unregister_gc mydb -geocoderName mygeocoder
```

db2se unregister_spatial_column command

The **db2se unregister_spatial_column** command removes the registration of a spatial column.

This command removes the registration by:

- Removing association of the spatial reference system with the spatial column. The DB2GSE.ST_GEOMETRY_COLUMNS catalog view continues to show the spatial column, but the column is no longer associated with any spatial reference system.
- For a base table, drop the constraint created on this table to ensure that the geometry values in this spatial column are all represented in the same spatial reference system.

Authorization

The user ID must hold one of the following authorities or privileges to run this command:

- DBADM and DATAACCESS authority on the database that contains the table to which the spatial column that is being registered belongs
- CONTROL privilege on this table
- ALTER privilege on this table

Command syntax

db2se unregister_spatial_column command

```
►►—unregister_spatial_column—database_name—————►
|
| ┌—userId—user_id—pw—password—┐ ┌—tableSchema—table_schema—┐
| └──────────────────────────────────┘ └────────────────────────────────┘
|
| ►—tableName—table_name—columnName—column_name—►◀
```

Command parameters

Where:

database_name

Specifies the name of the database for which you want to remove the registration of a spatial column.

-userId *user_id*

Specifies the database user ID that has DATAACCESS authority on the database indicated by *database_name*.

-pw *password*

Specifies the password for *user_id*.

-tableSchema *table_schema*

Specifies the schema name for the specified *table_name*. If you do not specify a schema name, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

-tableName *table_name*

Specifies the unqualified name of the table for the specified *column_name*. The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

-columnName *column_name*

Identifies the column to be unregistered. The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

Example

The following example unregisters a spatial column named MYCOLUMN in table MYTABLE.

```
db2se unregister_spatial_column mydb -tableName mytable -columnName mycolumn
```

db2se upgrade command

The **db2se upgrade** command upgrades an spatially enabled database from Version 9.5 or Version 9.7 to Version 10.1.

This command might drop and re-create spatial indexes to complete the database upgrade, which can take a significant amount of time depending on the sizes of your tables. For example, your indexes will be dropped and re-created if your data is moving from a 32-bit instance to a 64-bit instance.

Tip: Run the **db2se upgrade** command with the option **-force 0** and specify a messages file to find out which indexes must be upgraded without performing additional upgrade processing.

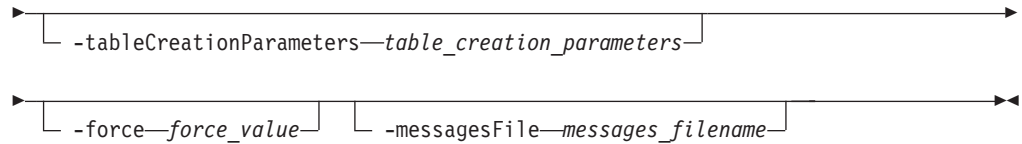
Authorization

DBADM and DATAACCESS authority on the spatially enabled database that you want to upgrade.

Command syntax

db2se upgrade command

```
➤ db2se upgrade database_name [ -userId user_id -pw password ] ➤
```

Command parameters

Where:

database_name

The name of the database to be upgraded.

-userId user_id

The database user ID that has DATAACCESS authority on the database that is being upgraded.

-pw password

Your user password.

-tableCreationParameters table_creation_parameters

The parameters to be used in the creation of the Spatial Extender catalog tables.

-force force_value

- 0: Default value. Attempt to upgrade the database, but it stops if any user-defined objects such as views, functions, triggers, or spatial indexes reference Spatial Extender objects.
- 1: Automatically saves and restores application-defined objects. Saves and restores spatial indexes if necessary.
- 2: Automatically saves and restores application-defined objects. Saves spatial index information, but does not automatically restore spatial indexes.

-messagesFile messages_filename

The file name containing the report of the upgrade actions. The file name you provide must be a fully-qualified file name on the server.

Tip: Specify this parameter to help you troubleshoot upgrade problems.

Restriction: You cannot specify an existing file.

Usage notes

The **db2se upgrade** command verifies several conditions and returns one or more of the following errors if any of these conditions are not true:

- Database is not currently spatially enabled.
- Database name is not valid.
- Other connections to the database exist. Cannot proceed with the upgrade.
- Spatial catalog is not consistent.
- User is not authorized.
- Password is not valid.
- Some user-defined objects could not be upgraded.

Ensure that you have a system temporary table space with a page size of 8 KB or larger and with a minimum size of 500 pages. This is a requirement to run the **db2se upgrade** command successfully.

db2se migrate command

The **db2se migrate** command migrates an spatially-enabled database to Version 9.7. This command is deprecated and will be removed in a future release. Use the **db2se upgrade** command instead.

This command might drop and re-create spatial indexes to complete the migration, which can take a significant amount of time depending on the sizes of your tables. For example, your indexes will be dropped and re-created if your data is moving from a 32-bit instance to a 64-bit instance.

Tip: Run the **db2se migrate** command with the option **-force 0** and specify a messages file to find out which indexes must be migrated without performing additional migration processing.

Authorization

SYSADM or DBADM authority on the spatially-enabled database that you want to migrate.

Command syntax

db2se migrate command

```
db2se migrate database_name [-userId user_id -pw password]
                             [-tableCreationParameters table_creation_parameters]
                             [-force force_value] [-messagesFile messages_filename]
```

Command parameters

Where:

database_name

The name of the database to be migrated.

-userId user_id

The database user ID which has either SYSADM or DBADM authority on the database that is being migrated.

-pw password

Your user password.

-tableCreationParameters table_creation_parameters

The parameters to be used in the creation of the Spatial Extender catalog tables.

-force force_value

- 0: Default value. Attempts database migration, but stops if any application-defined objects such as views, functions, triggers, or spatial indexes have been based on Spatial Extender objects.
- 1: Automatically saves and restores application-defined objects. Saves and restores spatial indexes if necessary.
- 2: Automatically saves and restores application-defined objects. Saves spatial index information, but does not automatically restore spatial indexes.

-messagesFile messages_filename

The file name containing the report of migration actions. The file name you provide must be a fully-qualified file name on the server.

Tip: Specify this parameter to help you troubleshoot migration problems.

Restriction: You cannot specify an existing file.

You might receive one or more of the following errors during migration:

- Database is not currently spatially enabled.
- Database name is not valid.
- Other connections to the database exist. Cannot be run.
- Spatial catalog is not consistent.
- User is not authorized.
- Password is not valid.
- Some user objects could not be migrated.

Chapter 17. Stored procedures

Use DB2 Spatial Extender stored procedures to set up DB2 Spatial Extender and create projects that use spatial data.

When you set up DB2 Spatial Extender or the DB2 command line processor, you invoke these stored procedures implicitly. For example, when you issue the CLP command **db2se create_srs**, the DB2GSE.ST_CREATE_SRS procedure is called.

Alternatively, you can invoke the DB2 Spatial Extender stored procedures explicitly in an application program.

Before invoking most of the DB2 Spatial Extender stored procedures on a database perform the following tasks:

1. Ensure that you have a system temporary table space with a page size of 8 KB or larger and with a minimum size of 500 pages. This is a requirement to be able to successfully run the ST_ENABLE_DB STORED procedure or the **db2se enable_db** command.
2. Enable the database for spatial operations by invoking the ST_ENABLE_DB procedure. See “ST_ENABLE_DB procedure” on page 187 for details.

After a database is enabled for spatial operations, you can invoke any DB2 Spatial Extender stored procedure, either implicitly or explicitly, on that database if you are connected to that database.

This chapter provides topics for all the DB2 Spatial Extender stored procedures, as follows:

- “ST_ALTER_COORDSYS procedure” on page 166
- “ST_ALTER_SRS procedure” on page 167
- “ST_CREATE_COORDSYS procedure” on page 171
- “ST_CREATE_SRS procedure” on page 173
- “ST_DISABLE_AUTOGEOCODING procedure” on page 179
- “ST_DISABLE_DB procedure” on page 181
- “ST_DROP_COORDSYS procedure” on page 182
- “ST_DROP_SRS procedure” on page 183
- “ST_ENABLE_AUTOGEOCODING procedure” on page 185
- “ST_ENABLE_DB procedure” on page 187
- “ST_EXPORT_SHAPE procedure” on page 189
- “ST_IMPORT_SHAPE procedure” on page 192
- “ST_REGISTER_GEOCODER procedure” on page 199
- “ST_REGISTER_SPATIAL_COLUMN procedure” on page 203
- “ST_REMOVE_GEOCODING_SETUP procedure” on page 205
- “ST_RUN_GEOCODING procedure” on page 207
- “ST_SETUP_GEOCODING procedure” on page 209
- “ST_UNREGISTER_GEOCODER procedure” on page 213
- “ST_UNREGISTER_SPATIAL_COLUMN procedure” on page 214

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

ST_ALTER_COORDSYS procedure

Use this stored procedure to update a coordinate system definition in the database. When this stored procedure is processed, information about the coordinate system is updated in the DB2GSE.ST_COORDINATE_SYSTEMS catalog view.

Attention: Use care with this stored procedure. If you use this stored procedure to change the definition of the coordinate system and you have existing spatial data that is associated with a spatial reference system that is based on this coordinate system, you might inadvertently change the spatial data. If spatial data is affected, you are responsible for ensuring that the changed spatial data is still accurate and valid.

Authorization

The user ID under which the stored procedure is invoked must have DBADM authority.

Syntax

```

▶▶ DB2GSE.ST_ALTER_COORDSYS (—coordsys_name—, —definition—, —
                                null—
                                )
▶ —organization—, —organization_coordsys_id—, —description—, —
  null—          null—          null—
▶ —msg_code—, —msg_text—)

```

Parameter descriptions

coordsys_name

Uniquely identifies the coordinate system. You must specify a non-null value for this parameter.

The *coordsys_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

definition

Defines the coordinate system. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the definition of the coordinate system is not changed.

The data type of this parameter is VARCHAR(2048).

organization

Names the organization that defined the coordinate system and provided the definition for it; for example, "European Petroleum Survey Group (EPSG)." Although you must specify a value for this parameter, the value can be null.

If this parameter is null, the organization of the coordinate system is not changed. If this parameter is not null, the *organization_coordsys_id* parameter cannot be null; in this case, the combination of the *organization* and *organization_coordsys_id* parameters uniquely identifies the coordinate system.

The data type of this parameter is VARCHAR(128).

organization_coordsys_id

Specifies a numeric identifier that is assigned to this coordinate system by the entity listed in the *organization* parameter. Although you must specify a value for this parameter, the value can be null.

If this parameter is null, the *organization* parameter must also be null; in this case, the organization's coordinate system identifier is not changed. If this parameter is not null, the *organization* parameter cannot be null; in this case, the combination of the *organization* and *organization_coordsys_id* parameters uniquely identifies the coordinate system.

The data type of this parameter is INTEGER.

description

Describes the coordinate system by explaining its application. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the description information about the coordinate system is not changed.

The data type of this parameter is VARCHAR(256).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST ALTER COORDSYS stored procedure. This example uses a DB2 CALL command to update a coordinate system named NORTH_AMERICAN_TEST. This CALL command assigns a value of 1002 to the *coordsys_id* parameter:

```
call DB2GSE.ST_ALTER_COORDSYS('NORTH_AMERICAN_TEST',NULL,NULL,1002,NULL,?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST ALTER SRS procedure

Use this stored procedure to update a spatial reference system definition in the database. When this stored procedure is processed, information about the spatial reference system is updated in the DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view.

Internally, DB2 Spatial Extender stores the coordinate values as positive integers. Thus during computation, the impact of rounding errors (which are heavily dependent on the actual value for floating-point operations) can be reduced. Performance of the spatial operations can also improve significantly.

Restriction: You cannot alter a spatial reference system if a registered spatial column uses that spatial reference system.

Attention: Use care with this stored procedure. If you use this stored procedure to change offset, scale, or *coordsys_name* parameters of the spatial reference system, and if you have existing spatial data that is associated with the spatial reference system, you might inadvertently change the spatial data. If spatial data is affected, you are responsible for ensuring that the changed spatial data is still accurate and valid.

Authorization

The user ID under which the stored procedure is invoked must have DBADM authority.

Syntax

```

►► DB2GSE.ST_ALTER_SRS (—srs_name—, —srs_id—, —x_offset—, —
                        [null]      [null]
► x_scale—, —y_offset—, —y_scale—, —z_offset—, —
  [null]    [null]    [null]    [null]
► z_scale—, —m_offset—, —m_scale—, —coordsys_name—, —
  [null]    [null]    [null]    [null]
► description—, —msg_code—, —msg_text—) ►►
  [null]

```

Parameter descriptions

srs_name

Identifies the spatial reference system. You must specify a non-null value for this parameter.

The *srs_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

srs_id

Uniquely identifies the spatial reference system. This identifier is used as an input parameter for various spatial functions. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the numeric identifier of the spatial reference system is not changed.

The data type of this parameter is INTEGER.

x_offset

Specifies the offset for all X coordinates of geometries that are represented in this spatial reference system. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value for this parameter in the definition of the spatial reference system is not changed.

The offset is subtracted before the scale factor *x_scale* is applied when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. (WKT is well-known text, and WKB is well-known binary.)

The data type of this parameter is DOUBLE.

x_scale

Specifies the scale factor for all X coordinates of geometries that are represented in this spatial reference system. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value for this parameter in the definition of the spatial reference system is not changed.

The scale factor is applied (multiplication) after the offset *x_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation.

The data type of this parameter is DOUBLE.

y_offset

Specifies the offset for all Y coordinates of geometries that are represented in this spatial reference system. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value for this parameter in the definition of the spatial reference system is not changed.

The offset is subtracted before the scale factor *y_scale* is applied when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation.

The data type of this parameter is DOUBLE.

y_scale

Specifies the scale factor for all Y coordinates of geometries that are represented in this spatial reference system. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value for this parameter in the definition of the spatial reference system is not changed.

The scale factor is applied (multiplication) after the offset *y_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. This scale factor must be the same as *x_scale*.

The data type of this parameter is DOUBLE.

z_offset

Specifies the offset for all Z coordinates of geometries that are represented in this spatial reference system. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value for this parameter in the definition of the spatial reference system is not changed.

The offset is subtracted before the scale factor *z_scale* is applied when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation.

The data type of this parameter is DOUBLE.

z_scale

Specifies the scale factor for all Z coordinates of geometries that are represented in this spatial reference system. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value for this parameter in the definition of the spatial reference system is not changed.

The scale factor is applied (multiplication) after the offset *z_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation.

The data type of this parameter is DOUBLE.

m_offset

Specifies the offset for all M coordinates of geometries that are represented in this spatial reference system. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value for this parameter in the definition of the spatial reference system is not changed.

The offset is subtracted before the scale factor *m_scale* is applied when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation.

The data type of this parameter is DOUBLE.

m_scale

Specifies the scale factor for all M coordinates of geometries that are represented in this spatial reference system. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value for this parameter in the definition of the spatial reference system is not changed.

The scale factor is applied (multiplication) after the offset *m_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation.

The data type of this parameter is DOUBLE.

coordsys_name

Uniquely identifies the coordinate system on which this spatial reference system is based. The coordinate system must be listed in the view ST_COORDINATE_SYSTEMS. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the coordinate system that is used for this spatial reference system is not changed.

The *coordsys_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

description

Describes the spatial reference system by explaining its application. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the description information about the spatial reference system is not changed.

The data type of this parameter is VARCHAR(256).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_ALTER_SRS stored procedure. This example uses a DB2 CALL command to change the *description* parameter value of a spatial reference system named SRSDemo:

```
call DB2GSE.ST_ALTER_SRS('SRSDemo',NULL,NULL,NULL,NULL,NULL,NULL,NULL,
NULL,NULL,'SRS for GSE Demo Program: offices table',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_CREATE_COORDSYS procedure

Use this stored procedure to create a new coordinate system. When this stored procedure is processed, information about the coordinate system is added to the DB2GSE.ST_COORDINATE_SYSTEMS catalog view.

Authorization

The user ID under which the stored procedure is invoked must have DBADM authority.

Syntax

```
►►DB2GSE.ST_CREATE_COORDSYS—(—coordsys_name—,—definition—,—  
►[organization], [organization_coordsys_id], [description],  
►[null] [null] [null]  
►—msg_code—,—msg_text—)►►
```

Parameter descriptions

coordsys_name

Uniquely identifies the coordinate system. You must specify a non-null value for this parameter.

The *coordsys_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

definition

Defines the coordinate system. You must specify a non-null value for this parameter. The vendor that supplies the coordinate system usually provides the information for this parameter.

The data type of this parameter is VARCHAR(2048).

organization

Names the organization that defined the coordinate system and provided the definition for it; for example, "European Petroleum Survey Group (EPSG)."
Although you must specify a value for this parameter, the value can be null.

If this parameter is null, the *organization_coordsys_id* parameter must also be null. If this parameter is not null, the *organization_coordsys_id* parameter cannot be null; in this case, the combination of the *organization* and *organization_coordsys_id* parameters uniquely identifies the coordinate system.

The data type of this parameter is VARCHAR(128).

organization_coordsys_id

Specifies a numeric identifier. The entity that is specified in the *organization* parameter assigns this value. This value is not necessarily unique across all coordinate systems. Although you must specify a value for this parameter, the value can be null.

If this parameter is null, the *organization* parameter must also be null. If this parameter is not null, the *organization* parameter cannot be null; in this case, the combination of the *organization* and *organization_coordsys_id* parameters uniquely identifies the coordinate system.

The data type of this parameter is INTEGER.

description

Describes the coordinate system by explaining its application. Although you must specify a value for this parameter, the value can be null. If this parameter is null, no description information about the coordinate system is recorded.

The data type of this parameter is VARCHAR(256).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_CREATE_COORDSYS stored procedure. This example uses a DB2 CALL command to create a coordinate system with the following parameter values:

- *coordsys_name* parameter: NORTH_AMERICAN_TEST
- *definition* parameter:

```
GEOGCS["GCS_North_American_1983",
DATUM["D_North_American_1983",
SPHEROID["GRS_1980",6378137.0,298.257222101]],
PRIMEM["Greenwich",0.0],
UNIT["Degree",0.0174532925199433]]
```

- *organization* parameter: EPSG
- *organization_coordsys_id* parameter: 1001
- *description* parameter: Test Coordinate Systems

```
call DB2GSE.ST_CREATE_COORDSYS('NORTH_AMERICAN_TEST',
'GEOGCS["GCS_North_American_1983",DATUM["D_North_American_1983",
SPHEROID["GRS_1980",6378137.0,298.257222101]],
PRIMEM["Greenwich",0.0],UNIT["Degree",
0.0174532925199433]]','EPSG',1001,'Test Coordinate Systems',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_CREATE_SRS procedure

Use the ST_CREATE_SRS stored procedure to create a spatial reference system.

A spatial reference system is defined by the coordinate system, the precision, and the extents of coordinates that are represented in this spatial reference system. The extents are the minimum and maximum possible coordinate values for the X, Y, Z, and M coordinates.

Internally, DB2 Spatial Extender stores the coordinate values as positive integers. Thus during computation, the impact of rounding errors (which are heavily dependent on the actual value for floating-point operations) can be reduced. Performance of the spatial operations can also improve significantly.

This stored procedure has two variations:

- The first variation takes the conversion factors (offsets and scale factors) as input parameters.
- The second variation takes the extents and the precision as input parameters and calculates the conversion factors internally.

Authorization

None required.

Syntax

With conversion factors (version 1)

```
►►DB2GSE.ST_CREATE_SRS—(—srs_name—,—srs_id—,—x_offset—,—
                        null—,
►x_scale—,—y_offset—,—y_scale—,—z_offset—,—
                        null—null—null—,
►z_scale—,—m_offset—,—m_scale—,—coordsys_name—,—
                        null—null—null—,
                        —, —)
```


y_offset

Specifies the offset for all Y coordinates of geometries that are represented in this spatial reference system. The offset is subtracted before the scale factor *y_scale* is applied when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. Although you must specify a value for this parameter, the value can be null. If this parameter is the null value, a value of 0 (zero) is used.

The data type of this parameter is DOUBLE.

y_scale

Specifies the scale factor for all Y coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the offset *y_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. Either the *y_offset* value is specified explicitly, or a default *y_offset* value of 0 is used. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value of the *x_scale* parameter is used. If you specify a value other than null for this parameter, the value that you specify must match the value of the *x_scale* parameter.

The data type of this parameter is DOUBLE.

z_offset

Specifies the offset for all Z coordinates of geometries that are represented in this spatial reference system. The offset is subtracted before the scale factor *z_scale* is applied when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. Although you must specify a value for this parameter, the value can be null. If this parameter is null, a value of 0 (zero) is used.

The data type of this parameter is DOUBLE.

z_scale

Specifies the scale factor for all Z coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the offset *z_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. Either the *z_offset* value is specified explicitly, or a default *z_offset* value of 0 is used. Although you must specify a value for this parameter, the value can be null. If this parameter is null, a value of 1 is used.

The data type of this parameter is DOUBLE.

m_offset

Specifies the offset for all M coordinates of geometries that are represented in this spatial reference system. The offset is subtracted before the scale factor *m_scale* is applied when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. Although you must specify a value for this parameter, the value can be null. If this parameter is null, a value of 0 (zero) is used.

The data type of this parameter is DOUBLE.

m_scale

Specifies the scale factor for all M coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the offset *m_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. Either the *m_offset* value is specified explicitly, or a default *m_offset* value of 0 is used. Although you must specify a value for this parameter, the value can be null. If this parameter is null, a value of 1 is used.

The data type of this parameter is DOUBLE.

coordsys_name

Uniquely identifies the coordinate system on which this spatial reference system is based. The coordinate system must be listed in the view ST_COORDINATE_SYSTEMS. You must supply a non-null value for this parameter.

The *coordsys_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

description

Describes the spatial reference system by explaining the application's purpose. Although you must specify a value for this parameter, the value can be null. If this parameter is null, no description information is recorded.

The data type of this parameter is VARCHAR(256).

With maximum possible extend (version 2)**srs_name**

Identifies the spatial reference system. You must specify a non-null value for this parameter.

The *srs_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

srs_id Uniquely identifies the spatial reference system. This numeric identifier is used as an input parameter for various spatial functions. You must specify a non-null value for this parameter.

The data type of this parameter is INTEGER.

x_min Specifies the minimum possible X coordinate value for all geometries that use this spatial reference system. You must specify a non-null value for this parameter.

The data type of this parameter is DOUBLE.

x_max Specifies the maximum possible X coordinate value for all geometries that use this spatial reference system. You must specify a non-null value for this parameter.

Depending on the value of *x_scale*, the value that is shown in the view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS might be larger than the value that is specified here. The value from the view is correct.

The data type of this parameter is DOUBLE.

x_scale

Specifies the scale factor for all X coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the offset *x_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. The calculation of the offset *x_offset* is based on the *x_min* value. You must supply a non-null value for this parameter.

If both the *x_scale* and *y_scale* parameters are specified, the values must match.

The data type of this parameter is DOUBLE.

y_min Specifies the minimum possible Y coordinate value for all geometries that use this spatial reference system. You must supply a non-null value for this parameter.

The data type of this parameter is DOUBLE.

y_max Specifies the maximum possible Y coordinate value for all geometries that use this spatial reference system. You must supply a non-null value for this parameter.

Depending on the value of *y_scale*, the value that is shown in the view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS might be larger than the value that is specified here. The value from the view is correct.

The data type of this parameter is DOUBLE.

y_scale

Specifies the scale factor for all Y coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the offset *y_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. The calculation of the offset *y_offset* is based on the *y_min* value. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value of the *x_scale* parameter is used. If both the *y_scale* and *x_scale* parameters are specified, the values must match.

The data type of this parameter is DOUBLE.

z_min Specifies the minimum possible Z coordinate value for all geometries that use this spatial reference system. You must specify a non-null value for this parameter.

The data type of this parameter is DOUBLE.

z_max Specifies the maximum possible Z coordinate value for all geometries that use this spatial reference system. You must specify a non-null value for this parameter.

Depending on the value of *z_scale*, the value that is shown in the view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS might be larger than the value that is specified here. The value from the view is correct.

The data type of this parameter is DOUBLE.

z_scale

Specifies the scale factor for all Z coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the offset *z_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. The calculation of the offset *z_offset* is based on the *z_min* value. Although you must specify a value for this parameter, the value can be null. If this parameter is null, a value of 1 is used.

The data type of this parameter is DOUBLE.

m_min

Specifies the minimum possible M coordinate value for all geometries that use this spatial reference system. You must specify a non-null value for this parameter.

The data type of this parameter is DOUBLE.

m_max

Specifies the maximum possible M coordinate value for all geometries that use this spatial reference system. You must specify a non-null value for this parameter.

Depending on the value of *m_scale*, the value that is shown in the view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS might be larger than the value that is specified here. The value from the view is correct.

The data type of this parameter is DOUBLE.

m_scale

Specifies the scale factor for all M coordinates of geometries that are represented in this spatial reference system. The scale factor is applied (multiplication) after the offset *m_offset* is subtracted when geometries are converted from external representations (WKT, WKB, shape) to the DB2 Spatial Extender internal representation. The calculation of the offset *m_offset* is based on the *m_min* value. Although you must specify a value for this parameter, the value can be null. If this parameter is null, a value of 1 is used.

The data type of this parameter is DOUBLE.

coordsys_name

Uniquely identifies the coordinate system on which this spatial reference system is based. The coordinate system must be listed in the view ST_COORDINATE_SYSTEMS. You must specify a non-null value for this parameter.

The *coordsys_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

description

Describes the spatial reference system by explaining the application's purpose. Although you must specify a value for this parameter, the value can be null. If this parameter is null, no description information is recorded.

The data type of this parameter is VARCHAR(256).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_CREATE_SRS stored procedure. This example uses a DB2 CALL command to create a spatial reference system named SRSDemo with the following parameter values:

- *srs_id*: 1000000
- *x_offset*: -180
- *x_scale*: 1000000
- *y_offset*: -90
- *y_scale*: 1000000

```
call DB2GSE.ST_CREATE_SRS('SRSDemo',1000000,  
                           -180,1000000, -90, 1000000,  
                           0, 1, 0, 1,'NORTH_AMERICAN',  
                           'SRS for GSE Demo Program: customer table',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_DISABLE_AUTOGEOCODING procedure

Use this stored procedure to specify that DB2 Spatial Extender is to stop synchronizing a geocoded column with its associated geocoding column or columns.

A *geocoding column* is used as input to the geocoder.

Authorization

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

- DBADM and DATAACCESS authority on the database that contains the table on which the triggers that are being dropped are defined
- CONTROL privilege on this table
- ALTER and UPDATE privileges on this table

Note: For CONTROL and ALTER privileges, you must have DROPIN authority on the DB2GSE schema.

Syntax

```
►►DB2GSE.ST_DISABLE_AUTOGEOCODING—(—table_schema—,—table_name—,—column_name—,—msg_code—,—msg_text—)►►
```

table_schema can be null.

Parameter descriptions

table_schema

Names the schema to which the table or view that is specified in the *table_name* parameter belongs. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

The *table_schema* value is converted to uppercase unless you enclose it in quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

table_name

Specifies the unqualified name of the table on which the triggers that you want dropped are defined. You must specify a non-null value for this parameter.

The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

column_name

Names the geocoded column that is maintained by the triggers that you want dropped. You must specify a non-null value for this parameter.

The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this

parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_DISABLE_AUTOGEOCODING stored procedure. This example uses a DB2 CALL command to disable autogeocoding on the LOCATION column in the table named CUSTOMERS:

```
call DB2GSE.ST_DISABLE_AUTOGEOCODING(NULL,'CUSTOMERS','LOCATION',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_DISABLE_DB procedure

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

This stored procedure helps you resolve problems or issues that arise after you enable your database for spatial operations. For example, you might enable a database for spatial operations and then decide to use another database with DB2 Spatial Extender instead. If you did not define any spatial columns or import any spatial data, you can invoke this stored procedure to remove all spatial resources from the first database. Because of the interdependency between spatial columns and the type definitions, you cannot drop the type definitions when columns of those types exist. If you already defined spatial columns but still want to disable a database for spatial operations, you must specify a value other than 0 (zero) for the *force* parameter to remove all spatial resources in the database that do not have other dependencies on them.

Authorization

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

Syntax

```
►► DB2GSE.ST_DISABLE_DB—(—force—, —msg_code—, —msg_text—)——►◄  
                          └—null—┘
```

Parameter descriptions

force

Specifies that you want to disable a database for spatial operations, even though you might have database objects that are dependent on the spatial types or spatial functions. Although you must specify a value for this parameter, the value can be null. If you specify a value other than 0 (zero) or null for the *force* parameter, the database is disabled, and all resources of the DB2 Spatial Extender are removed (if possible). If you specify 0 (zero) or null, the database is not disabled if any database objects are dependent on spatial types or spatial functions. Database objects that might have such dependencies include tables, views, constraints, triggers, generated columns, methods, functions, procedures, and other data types (subtypes or structured types with a spatial attribute).

The data type of this parameter is SMALLINT.

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_DISABLE_DB stored procedure. This example uses a DB2 CALL command to disable the database for spatial operations, with a *force* parameter value of 1:

```
call DB2GSE.ST_DISABLE_DB(1,?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_DROP_COORDSYS procedure

Use this stored procedure to delete information about a coordinate system from the database. When this stored procedure is processed, information about the coordinate system is no longer available from the DB2GSE.ST_COORDINATE_SYSTEMS catalog view.

Restriction:

You cannot drop a coordinate system on which a spatial reference system is based.

Authorization

The user ID under which the stored procedure is invoked must have DBADM authority.

Syntax

►►—DB2GSE.ST_DROP_COORDSYS—(—*coordsys_name*—,—*msg_code*—,—*msg_text*—)—►►

Parameter descriptions

coordsys_name

Uniquely identifies the coordinate system. You must specify a non-null value for this parameter.

The *coordsys_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_DROP_COORDSYS stored procedure. This example uses a DB2 CALL command to delete a coordinate system named NORTH_AMERICAN_TEST from the database:

```
call DB2GSE.ST_DROP_COORDSYS('NORTH_AMERICAN_TEST',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_DROP_SRS procedure

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.ST_SPATIAL_REFERENCE_SYSTEMS catalog view.

Restriction: You cannot drop a spatial reference system if a spatial column that uses that spatial reference system is registered.

Important:

Use care when you use this stored procedure. If you use this stored procedure to drop a spatial reference system, and if any spatial data is associated with that spatial reference system, you can no longer perform spatial operations on the spatial data.

Authorization

The user ID under which the stored procedure is invoked must have DBADM authority.

Syntax

►► DB2GSE.ST_DROP_SRS—(*—srs_name—*,*—msg_code—*,*—msg_text—*)————►►

Parameter descriptions

srs_name

Identifies the spatial reference system. You must specify a non-null value for this parameter.

The *srs_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

This example shows how to use the DB2 command line processor to invoke the ST_DROP_SRS stored procedure. This example uses a DB2 CALL command to delete a spatial reference system named SRSDEMO:

```
call DB2GSE.ST_DROP_SRS('SRSDemo',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_ENABLE_AUTOGEOCODING procedure

Use this stored procedure to specify that DB2 Spatial Extender is to synchronize a geocoded column with its associated geocoding column or columns.

A *geocoding column* is used as input to the geocoder. Each time that values are inserted into, or updated in, the geocoding column or columns, triggers are activated. These triggers invoke the associated geocoder to geocode the inserted or updated values and to place the resulting data in the geocoded column.

Restriction: You can enable autogeocoding only on tables on which INSERT and UPDATE triggers can be created. Consequently, you cannot enable autogeocoding on views or nicknames.

Prerequisite: Before enabling autogeocoding, you must perform the geocoding setup step by calling the ST_SETUP_GEOCODING procedure. The geocoding setup step specifies the geocoder and the geocoding parameter values. It also identifies the geocoding columns that are to be synchronized with the geocoded columns.

Authorization

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

- DBADM authority on the database that contains the table on which the triggers that are created by this stored procedure are defined
- CONTROL privilege on the table
- ALTER privilege on the table

If the authorization ID of the statement does not have DBADM authority, the privileges that the authorization ID of the statement holds (without considering PUBLIC or group privileges) must include all of the following privileges as long as the trigger exists:

- SELECT privilege on the table on which autogeocoding is enabled or DATAACCESS authority
- Necessary privileges to evaluate the SQL expressions that are specified for the parameters in the geocoding setup

Syntax

►► DB2GSE.ST_ENABLE_AUTOGEOCODING—(table_schema , table_name ,)

►—*column_name*—,—*msg_code*—,—*msg_text*—)—————►

Parameter descriptions

table_schema

Identifies the schema to which the table that is specified in the *table_name* parameter belongs. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value in the CURRENT SCHEMA special register is used as the schema name for the table.

The *table_schema* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

table_name

Specifies the unqualified name of the table that contains the column into which the geocoded data is to be inserted or updated. You must specify a non-null value for this parameter.

The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

column_name

Identifies the column into which the geocoded data is to be inserted or updated. This column is referred to as the geocoded column. You must specify a non-null value for this parameter.

The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_ENABLE_AUTOGEOCODING stored procedure. This example uses a DB2 CALL command to enable autogeocoding on the LOCATION column in the table named CUSTOMERS:

```
call DB2GSE.ST_ENABLE_AUTOGEOCODING(NULL,'CUSTOMERS','LOCATION',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_ENABLE_DB procedure

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

This stored procedure replaces DB2GSE.gse_enable_db.

Authorization

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

Syntax

```
► DB2GSE.ST_ENABLE_DB ( ( table_creation_parameters , msg_code , msg_text ) ) ►
```

table_creation_parameters can be null.

Parameter descriptions

table_creation_parameters

Specifies any options that are to be added to the CREATE TABLE statements for the DB2 Spatial Extender catalog tables. Although you must specify a value for this parameter, the value can be null. If this parameter is null, no options are added to the CREATE TABLE statements.

To specify these options, use the syntax of the DB2 CREATE TABLE statement. For example, to specify a table space in which to create the tables, use:

```
IN tsName INDEX IN indexTsName
```

The data type of this parameter is VARCHAR(32K).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

The following example shows how to use Call Level Interface (CLI) to invoke the ST_ENABLE_DB stored procedure:

```
SQLHANDLE henv;
SQLHANDLE hdbc;
SQLHANDLE hstmt;
SQLCHAR uid[MAX_UID_LENGTH + 1];
SQLCHAR pwd[MAX_PWD_LENGTH + 1];
SQLINTEGER ind[3];
SQLINTEGER msg_code = 0;
char msg_text[1024] = "";
SQLRETURN rc;
char *table_creation_parameters = NULL;

/* Allocate environment handle */
rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

/* Allocate database handle */
rc = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

/* Establish a connection to database "testdb" */
rc = SQLConnect(hdbc, (SQLCHAR *)"testdb", SQL_NTS, (SQLCHAR *)uid, SQL_NTS,
               (SQLCHAR *)pwd, SQL_NTS);

/* Allocate statement handle */
rc = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

/* Associate SQL statement to call the ST_ENABLE_DB stored procedure */
/* with statement handle and send the statement to DBMS to be prepared. */
rc = SQLPrepare(hstmt, "call DB2GSE!ST_ENABLE_DB(?,?,?)", SQL_NTS);

/* Bind 1st parameter marker in the SQL call statement, the input */
/* parameter for table creation parameters, to variable */
/* table_creation_parameters. */
ind[0] = SQL_NULL_DATA;
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_OUTPUT, SQL_C_CHAR,
                     SQL_VARCHAR, 255, 0, table_creation_parameters, 256, &ind[0]);

/* Bind 2nd parameter marker in the SQL call statement, the output */
/* parameter for returned message code, to variable msg_code. */
ind[1] = 0;
rc = SQLBindParameter(hstmt, 2, SQL_PARAM_OUTPUT, SQL_C_LONG,
                     SQL_INTEGER, 0, 0, &msg_code, 4, &ind[1]);

/* Bind 3rd parameter marker in the SQL call statement, the output */
/* parameter returned message text, to variable msg_text. */
ind[2] = 0;
rc = SQLBindParameter(hstmt, 3, SQL_PARAM_OUTPUT, SQL_C_CHAR,
                     SQL_VARCHAR, (sizeof(msg_text)-1), 0, msg_text,
                     sizeof(msg_text), &ind[2]);
rc = SQLExecute(hstmt);
```

ST_EXPORT_SHAPE procedure

Use this stored procedure to export a spatial column and its associated table to a shape file.

Authorization

The user ID under which this stored procedure is invoked must have the necessary privileges to successfully execute the SELECT statement from which the data is to be exported.

The stored procedure, which runs as a process that is owned by the DB2 instance owner, must have the necessary privileges on the server machine to create or write to the shape files.

Syntax

```
► DB2GSE.ST_EXPORT_SHAPE (—file_name—, —append_flag—, —————►
                           |null|
                           ▼
► —output_column_names—, —select_statement—, —messages_file—, —————►
   |null|                               |null|
   ▼                                   ▼
► —msg_code—, —msg_text—) —————►◀
```

Parameter descriptions

file_name

Specifies the full path name of a shape file to which the specified data is to be exported. You must specify a non-null value for this parameter.

You can use the ST_EXPORT_SHAPE stored procedure to export a new file or to export to an existing file by appending the exported data to it:

- If you are exporting to a new file, you can specify the optional file extension as .shp or .SHP. If you specify .shp or .SHP for the file extension, DB2 Spatial Extender creates the file with the specified *file_name* value. If you do not specify the optional file extension, DB2 Spatial Extender creates the file that has the name of the *file_name* value that you specify and with an extension of .shp.
- If you are exporting data by appending the data to an existing file, DB2 Spatial Extender first looks for an exact match of the name that you specify for the *file_name* parameter. If DB2 Spatial Extender does not find an exact match, it looks first for a file with the .shp extension, and then for a file with the .SHP extension.

If the value of the *append_flag* parameter indicates that you are not appending to an existing file, but the file that you name in the *file_name* parameter already exists, DB2 Spatial Extender returns an error and does not overwrite the file.

See Usage notes for a list of files that are written on the server machine. The stored procedure, which runs as a process that is owned by the DB2 instance owner, must have the necessary privileges on the server machine to create or write to the files.

The data type of this parameter is VARCHAR(256).

append_flag

Indicates whether the data that is to be exported is to be appended to an

existing shape file. Although you must specify a value for this parameter, the value can be null. Indicate whether you want to append to an existing shape file as follows:

- If you want to append data to an existing shape file, specify any value other than 0 (zero) and null. In this case, the file structure must match the exported data; otherwise an error is returned.
- If you want to export to a new file, specify 0 (zero) or null. In this case, DB2 Spatial Extender does not overwrite any existing files.

The data type of this parameter is SMALLINT.

output_column_names

Specifies one or more column names (separated by commas) that are to be used for non-spatial columns in the output dBASE file. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the names that are derived from the SELECT statement are used.

If you specify this parameter but do not enclose column names in double quotation marks, the column names are converted to uppercase. The number of specified columns must match the number of columns that are returned from the SELECT statement, as specified in the *select_statement* parameter, excluding the spatial column.

The data type of this parameter is VARCHAR(32K).

select_statement

Specifies the subselect that returns the data that is to be exported. The subselect must reference exactly one spatial column and any number of attribute columns. You must specify a non-null value for this parameter.

The data type of this parameter is VARCHAR(32K).

messages_file

Specifies the full path name of the file (on the server machine) that is to contain messages about the export operation. Although you must specify a value for this parameter, the value can be null. If this parameter is null, no file for DB2 Spatial Extender messages is created.

The messages that are sent to this messages file can be:

- Informational messages, such as a summary of the export operation
- Error messages for data that could not be exported, for example because of different coordinate systems

The stored procedure, which runs as a process that is owned by the DB2 instance owner, must have the necessary privileges on the server to create the file.

The data type of this parameter is VARCHAR(256).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Usage notes

You can export only one spatial column at a time.

The ST_EXPORT_SHAPE stored procedure creates or writes to the following four files:

- The main shape file (.shp extension).
- The shape index file (.shx extension).
- A dBASE file that contains data for non-spatial columns (.dbf extension). This file is created only if attribute columns actually need to be exported
- A projection file that specifies the coordinate system that is associated with the spatial data, if the coordinate system is not equal to "UNSPECIFIED" (.prj extension). The coordinate system is obtained from the first spatial record. An error occurs if subsequent records have different coordinate systems.

The following table describes how DB2 data types are stored in dBASE attribute files. All other DB2 data types are not supported.

Table 22. Storage of DB2 data types in attribute files

SQL type	.dbf type	.dbf length	.dbf decimals	Comments
SMALLINT	N	6	0	
INTEGER	N	11	0	
BIGINT	N	20	0	
DECIMAL	N	precision+2	scale	
REAL FLOAT(1) through FLOAT(24)	F	14	6	
DOUBLE FLOAT(25) through FLOAT(53)	F	19	9	
CHARACTER, VARCHAR, LONG VARCHAR, and DATALINK	C	<i>len</i>	0	length ≤ 255
DATE	D	8	0	
TIME	C	8	0	
TIMESTAMP	C	26	0	

All synonyms for data types and distinct types that are based on the types listed in the preceding table are supported.

Example

This example shows how to use the DB2 command line processor to invoke the ST_EXPORT_SHAPE stored procedure. This example uses a DB2 CALL command to export all rows from the CUSTOMERS table to a shape file that is to be created and named /tmp/export_file:

```
call DB2GSE.ST_EXPORT_SHAPE('/tmp/export_file',0,NULL,  
    'select * from customers','/tmp/export_msg',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_IMPORT_SHAPE procedure

Use this stored procedure to import a shape file to a database that is enabled for spatial operations.

The stored procedure can operate in either of two ways, based on the *create_table_flag* parameter:

- DB2 Spatial Extender can create a table that has a spatial column and attribute columns, and it can then load the table's columns with the file's data.
- Otherwise, the shape and attribute data can be loaded into an existing table that has a spatial column and attribute columns that match the file's data.

Authorization

The owner of the DB2 instance must have the necessary privileges on the server machine for reading the input files and optionally writing error files. Additional authorization requirements vary based on whether you are importing into an existing table or into a new table.

- **When importing into an existing table**, the user ID under which this stored procedure is invoked must hold one of the following authorities or privileges:
 - DATAACCESS
 - CONTROL privilege on the table or view
 - INSERT and SELECT privilege on the table or view
- **When importing into a new table**, the user ID under which this stored procedure is invoked must hold one of the following authorities or privileges:
 - DBADM
 - CREATETAB authority on the database

The user ID must also have one of the following authorities:

- IMPLICIT_SCHEMA authority on the database, if the schema name of the table does not exist
- CREATEIN privilege on the schema, if the schema of the table exists

Syntax

```

▶▶ DB2GSE.ST_IMPORT_SHAPE (—file_name—, —input_attr_columns—, —————▶
                             —null—
▶ —srs_name—, —table_schema—, —table_name—, —table_attr_columns—, —————▶
                             —null— —null—
▶ —create_table_flag—, —table_creation_parameters—, —spatial_column—▶
  —null— —null—
▶ —, —type_schema—, —type_name—, —inline_length—, —id_column—▶
  —null— —null— —null— —null—
▶ —, —id_column_is_identity—, —restart_count—, —commit_scope—, —————▶
  —null— —null— —null—

```


► exception_file, messages_file, msg_code, msg_text—►
 null null

Parameter descriptions

file_name

Specifies the full path name of the shape file that is to be imported. You must specify a non-null value for this parameter.

If you specify the optional file extension, specify either .shp or .SHP. DB2 Spatial Extender first looks for an exact match of the specified file name. If DB2 Spatial Extender does not find an exact match, it looks first for a file with the .shp extension, and then for a file with the .SHP extension.

See Usage notes for a list of required files, which must reside on the server machine. The stored procedure, which runs as a process that is owned by the DB2 instance owner, must have the necessary privileges on the server to read the files.

The data type of this parameter is VARCHAR(256).

input_attr_columns

Specifies a list of attribute columns to import from the dBASE file. Although you must specify a value for this parameter, the value can be null. If this parameter is null, all columns are imported. If the dBASE file does not exist, this parameter must be the empty string or null.

To specify a non-null value for this parameter, use one of the following specifications:

- **List the attribute column names.** The following example shows how to specify a list of the names of the attribute columns that are to be imported from the dBASE file:

```
N(COLUMN1,COLUMN5,COLUMN3,COLUMN7)
```

If a column name is not enclosed in double quotation marks, it is converted to uppercase. Each name in the list must be separated by a comma. The resulting names must exactly match the column names in the dBASE file.

- **List the attribute column numbers.** The following example shows how to specify a list of the numbers of the attribute columns that are to be imported from the dBASE file:

```
P(1,5,3,7)
```

Columns are numbered beginning with 1. Each number in the list must be separated by a comma.

- **Indicate that no attribute data is to be imported.** Specify "", which is an empty string that explicitly specifies that DB2 Spatial Extender is to import *no* attribute data.

The data type of this parameter is VARCHAR(32K).

srs_name

Identifies the spatial reference system that is to be used for the geometries that are imported into the spatial column. You must specify a non-null value for this parameter.

The spatial column will not be registered. The spatial reference system (SRS) must exist before the data is imported. The import process does not implicitly create the SRS, but it does compare the coordinate system of the SRS with the

coordinate system that is specified in the .prj file (if available with the shape file). The import process also verifies that the extents of the data in the shape file can be represented in the given spatial reference system. That is, the import process verifies that the extents lie within the minimum and maximum possible X, Y, Z, and M coordinates of the SRS.

The *srs_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

table_schema

Names the schema to which the table that is specified in the *table_name* parameter belongs. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

The *table_schema* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

table_name

Specifies the unqualified name of the table into which the imported shape file is to be loaded. You must specify a non-null value for this parameter.

The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

table_attr_columns

Specifies the table column names where attribute data from the dBASE file is to be stored. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the names of the columns in the dBASE file are used.

If this parameter is specified, the number of names must match the number of columns that are imported from the dBASE file. If the table exists, the column definitions must match the incoming data. See Usage notes for an explanation of how attribute data types are mapped to DB2 data types.

The data type of this parameter is VARCHAR(32K).

create_table_flag

Specifies whether the import process is to create a new table. Although you must specify a value for this parameter, the value can be null. If this parameter is null or any other value other than 0 (zero), a new table is created. (If the table already exists, an error is returned.) If this parameter is 0 (zero), no table is created, and the table must already exist.

The data type of this parameter is INTEGER.

table_creation_parameters

Specifies any options that are to be added to the CREATE TABLE statement that creates a table into which data is to be imported. Although you must specify a value for this parameter, the value can be null. If this parameter is null, no options are added to the CREATE TABLE statement.

To specify any CREATE TABLE options, use the syntax of the DB2 CREATE TABLE statement. For example, to specify a table space in which to create the tables, specify:

```
IN tsName INDEX IN indexTsName LONG IN longTsName
```

The data type of this parameter is VARCHAR(32K).

spatial_column

Name of the spatial column in the table into which the shape data is to be loaded. You must specify a non-null value for this parameter.

For a new table, this parameter specifies the name of the new spatial column that is to be created. Otherwise, this parameter specifies the name of an existing spatial column in the table.

The *spatial_column* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

type_schema

Specifies the schema name of the spatial data type (specified by the *type_name* parameter) that is to be used when creating a spatial column in a new table. Although you must specify a value for this parameter, the value can be null. If this parameter is null, a value of DB2GSE is used.

The *type_schema* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

type_name

Names the data type that is to be used for the spatial values. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the data type is determined by the shape file and is one of the following types:

- ST_Point
- ST_MultiPoint
- ST_MultiLineString
- ST_MultiPolygon

Note that shape files, by definition, allow a distinction only between points and multipoints, but not between polygons and multipolygons or between linestrings and multilinestrings.

If you are importing into a table that does not yet exist, this data type is also used for the data type of the spatial column. In that case, the data type can also be a super type of ST_Point, ST_MultiPoint, ST_MultiLineString, or ST_MultiPolygon.

The *type_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

inline_length

Specifies, for a new table, the maximum number of bytes that are to be allocated for the spatial column within the table. Although you must specify a

value for this parameter, the value can be null. If this parameter is null, no explicit `INLINE LENGTH` option is used in the `CREATE TABLE` statement, and DB2 defaults are used implicitly.

Spatial records that exceed this size are stored separately in the LOB table space, which might be slower to access.

Typical sizes that are needed for various spatial types are as follows:

- **One point:** 292.
- **Multipoint, line, or polygon:** As large a value as possible. Consider that the total number of bytes in one row should not exceed the limit for the page size of the table space for which the table is created.

See the DB2 documentation about the `CREATE TABLE SQL` statement for a complete description of this value. See also the `db2dart` utility to determine the number of inline geometries for existing tables and the ability to alter the inline length.

The data type of this parameter is `INTEGER`.

id_column

Names a column that is to be created to contain a unique number for each row of data. (ESRI tools require a column named `SE_ROW_ID`.) The unique values for that column are generated automatically during the import process. Although you must specify a value for this parameter, the value can be null if no column (with a unique ID in each row) exists in the table or if you are not adding such a column to a newly created table. If this parameter is null, no column is created or populated with unique numbers.

Restriction: You cannot specify an *id_column* name that matches the name of any column in the dBASE file.

The requirements and effect of this parameter depend on whether the table already exists.

- **For an existing table,** the data type of the *id_column* parameter can be any integer type (`INTEGER`, `SMALLINT`, or `BIGINT`).
- **For a new table that is to be created,** the column is added to the table when the stored procedure creates it. The column will be defined as follows:

```
INTEGER NOT NULL PRIMARY KEY
```

If the value of the *id_column_is_identity* parameter is not null and not 0 (zero), the definition is expanded as follows:

```
INTEGER NOT NULL PRIMARY KEY GENERATED ALWAYS AS IDENTITY  
( START WITH 1 INCREMENT BY 1 )
```

The *id_column* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is `VARCHAR(128)` or, if you enclose the value in double quotation marks, `VARCHAR(130)`.

id_column_is_identity

Indicates whether the specified *id_column* is to be created using the `IDENTITY` clause. Although you must specify a value for this parameter, the value can be null. If this parameter is 0 (zero) or null, the column is not created as the identity column. If the parameter is any value other than 0 or null, the column is created as the identity column. This parameter is ignored for tables that already exist.

The data type of this parameter is `SMALLINT`.

restart_count

Specifies that an import operation is to be started at record $n + 1$. The first n records are skipped. Although you must specify a value for this parameter, the value can be null. If this parameter is null, all records (starting with record number 1) are imported.

The data type of this parameter is INTEGER.

commit_scope

Specifies that a COMMIT is to be performed after at least n records are imported. Although you must specify a value for this parameter, the value can be null. If this parameter is null, a value of 0 (zero) is used, and a COMMIT is performed at the end of the operation. This can result in large logfile utilization and data lost on operations that are interrupted.

The data type of this parameter is INTEGER.

exception_file

Specifies the full path name of a shape file in which the shape data that could not be imported is stored. Although you must specify a value for this parameter, the value can be null. If the parameter is null, no files are created.

If you specify a value for the parameter and include the optional file extension, specify either .shp or .SHP. If the extension is null, an extension of .shp is appended.

The exception file holds the complete block of rows for which a single insert statement failed. For example, assume that one row cannot be imported because the shape data is incorrectly encoded. A single insert statement attempts to import 20 rows, including the one that is in error. Because of the problem with the single row, the entire block of 20 rows is written to the exception file.

Records are written to the exception file only when those records can be correctly identified, as is the case when the shape record type is not valid. Some types of corruption to the shape data (.shp files) and shape index (.shx files) do not allow the appropriate records to be identified. In this case, no records are written to the exception file, and an error message is issued to report the problem.

If you specify a value for this parameter, four files are created on the server machine. See Usage notes for an explanation these files. The stored procedure, which runs as a process that is owned by the DB2 instance owner, must have the necessary privileges on the server to create the files. If the files already exist, the stored procedure returns an error.

The data type of this parameter is VARCHAR(256).

messages_file

Specifies the full path name of the file (on the server machine) that is to contain messages about the import operation. Although you must specify a value for this parameter, the value can be null. If the parameter is null, no file for DB2 Spatial Extender messages is created.

The messages that are written to the messages file can be:

- Informational messages, such as a summary of the import operation
- Error messages for data that could not be imported, for example because of different coordinate systems

These messages correspond to the shape data that is stored in the exception file (identified by the *exception_file* parameter).

The stored procedure, which runs as a process that is owned by the DB2 instance owner, must have the necessary privileges on the server to create the file. If the file already exists, the stored procedure returns an error.

The data type of this parameter is VARCHAR(256).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Usage notes

The ST_IMPORT_SHAPE stored procedure uses from one to four files:

- The main shape file (.shp extension). This file is required.
- The shape index file (.shx extension). This file is optional. If it is present, performance of the import operation might improve.
- A dBASE file that contains attribute data (.dbf extension). This file is required only if attribute data is to be imported.
- The projection file that specifies the coordinate system of the shape data (.prj extension). This file is optional. If this file is present, the coordinate system that is defined in it is compared with the coordinate system of the spatial reference system that is specified by the *srs_id* parameter.

The following table describes how dBASE attribute data types are mapped to DB2 data types. All other attribute data types are not supported.

Table 23. Relationship between DB2 data types and dBASE attribute data types

.dbf type	.dbf length ¹	.dbf decimals ²	SQL type	Comments
N	< 5	0	SMALLINT	
N	< 10	0	INTEGER	
N	< 20	0	BIGINT	
N	<i>len</i>	<i>dec</i>	DECIMAL(<i>len</i> , <i>dec</i>)	<i>len</i> < 32
F	<i>len</i>	<i>dec</i>	REAL	<i>len</i> + <i>dec</i> < 7
F	<i>len</i>	<i>dec</i>	DOUBLE	
C	<i>len</i>		CHAR(<i>len</i>)	
L			CHAR(1)	
D			DATE	

Note: This table includes the following variables, both of which are defined in the header of the dBASE file:

1. *len*, which represents the total length of the column in the dBASE file. DB2 Spatial Extender uses this value for two purposes:
 - To define the precision for the SQL data type DECIMAL or the length for the SQL data type CHAR
 - To determine which of the integer or floating-point types is to be used
2. *dec*, which represents the maximum number of digits to the right of the decimal point of the column in the dBASE file. DB2 Spatial Extender uses this value to define the scale for the SQL data type DECIMAL.

For example, assume that the dBASE file contains a column of data whose length (*len*) is defined as 20. Assume that the number of digits to the right of the decimal point (*dec*) is defined as 5. When DB2 Spatial Extender imports data from that column, it uses the values of *len* and *dec* to derive the following SQL data type: DECIMAL(20,5).

Example

This example shows how to use the DB2 command line processor to invoke the ST_IMPORT_SHAPE stored procedure. This example uses a DB2 CALL command to import a shape file named /tmp/officesShape into the table named OFFICES:

```
call DB2GSE.ST_IMPORT_SHAPE('/tmp/officesShape',NULL,'USA_SRS_1',NULL,
                             'OFFICES',NULL,0,NULL,'LOCATION',NULL,NULL,NULL,
                             NULL,NULL,NULL,NULL,'/tmp/import_msg',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_REGISTER_GEOCODER procedure

Use this stored procedure to register a geocoder.

Prerequisites: Before registering a geocoder:

- Ensure that the function that implements the geocoder is already created. Each geocoder function can be registered as a geocoder with a uniquely identified geocoder name.
- Obtain information from the geocoder vendor, such as:
 - The SQL statement that creates the function
 - The parameter values used to call the ST_CREATE_SRS procedure so that geometric data can be supported
 - Information for registering the geocoder, such as:
 - A description of the geocoder
 - Descriptions of the parameters for the geocoder
 - The default values of the geocoder parameters

The geocoder function's return type must match the data type of the geocoded column. The geocoding parameters can be either a column name (called a *geocoding column*) which contains data that the geocoder needs. For example, the geocoder parameters can identify addresses or a value of particular meaning to the geocoder, such as the minimum match score. If the geocoding parameter is a column name, the column must be in the same table or view as the geocoded column.

The geocoder function's return type serves as the data type for the geocoded column. The return type can be any DB2 data type, user-defined type, or structured type. If a user-defined type or structured type is returned, the geocoder function is responsible for returning a valid value of the respective data type. If the geocoder function returns values of a spatial type, that is ST_Geometry or one of its subtypes, the geocoder function is responsible for constructing a valid geometry. The geometry must be represented using an existing spatial reference system. The geometry is valid if you invoke the ST_IsValid spatial function on the geometry and a value of 1 is returned. The returned data from the geocoder function is updated in or is inserted into the geocoded column, depending on which operation (INSERT or UPDATE) caused the generation of the geocoded value.

To find out whether a geocoder is already registered, examine the DB2GSE.ST_GEOCODERS catalog view.

Authorization

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

Syntax

```

▶▶ DB2GSE.ST_REGISTER_GEOCODER (—geocoder_name—, —function_schema—, —
                                —null—,
▶ —function_name—, —specific_name—, —default_parameter_values—, —
  —null—, —null—, —null—,
▶ —parameter_descriptions—, —vendor—, —description—, —msg_code—
  —null—, —null—, —null—,
▶ —, —msg_text—)

```

Parameter descriptions

geocoder_name

Uniquely identifies the geocoder. You must specify a non-null value for this parameter.

The *geocoder_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

function_schema

Names the schema for the function that implements this geocoder. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value in the CURRENT SCHEMA special register is used as the schema name for the function.

The *function_schema* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

function_name

Specifies the unqualified name of the function that implements this geocoder. The function must already be created and listed in SYSCAT.ROUTINES.

For this parameter, you can specify null if the *specific_name* parameter is specified. If the *specific_name* parameter is not specified, the *function_name* value, together with the implicitly or explicitly defined *function_schema* value, must uniquely identify the function. If the *function_name* parameter is not specified, DB2 Spatial Extender retrieves the *function_name* value from the SYSCAT.ROUTINES catalog view.

The *function_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

specific_name

Identifies the specific name of the function that implements the geocoder. The function must already be created and listed in SYSCAT.ROUTINES.

For this parameter, you can specify null if the *function_name* parameter is specified and the combination of *function_schema* and *function_name* uniquely identifies the geocoder function. If the geocoder function name is overloaded, the *specific_name* parameter cannot be null. (A function name is *overloaded* if it has the same name, but not the same parameters or parameter data types, as one or more other functions.)

The *specific_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

default_parameter_values

Specifies the list of default geocoding parameter values for the geocoder function. Although you must specify a value for this parameter, the value can be null. If the entire *default_parameter_values* parameter is null, all parameter default values are null.

If you specify any parameter values, specify them in the order that the function defined them, and separate them with a comma. For example:

default_parm1_value, default_parm2_value, ...

Each parameter value is an SQL expression. Follow these guidelines:

- If a value is a string, enclose it in single quotation marks.
- If a parameter value is a number, do not enclose it in single quotation marks.
- If the parameter value is null, cast it to the correct type. For example, instead of specifying just NULL, specify:
CAST(NULL AS INTEGER)
- If the geocoding parameter is to be a geocoding column, do not specify the default parameter value.

If any parameter value is not specified (that is, if you specify two consecutive commas (...)), this parameter must be specified either when geocoding is set up or when geocoding is run in batch mode with the *parameter_values* parameter of the respective stored procedures.

The data type of this parameter is VARCHAR(32K).

parameter_descriptions

Specifies the list of geocoding parameter descriptions for the geocoder function. Although you must specify a value for this parameter, the value can be null.

If the entire *parameter_descriptions* parameter is null, all parameter descriptions are null. Each parameter description that you specify explains the meaning and usage of the parameter, and can be up to 256 characters long. The descriptions for the parameters must be separated by commas and must appear in the order of the parameters as defined by the function. If a comma shall be used within the description of a parameter, enclose the string in single or double quotation marks. For example:

description, 'description2, which contains a comma', description3

The data type of this parameter is VARCHAR(32K).

vendor

Names the vendor who implemented the geocoder. Although you must specify a value for this parameter, the value can be null. If this parameter is null, no information about the vendor who implemented the geocoder is recorded.

The data type of this parameter is VARCHAR(128).

description

Describes the geocoder by explaining its application. Although you must specify a value for this parameter, the value can be null. If this parameter is null, no description information about the geocoder is recorded.

The data type of this parameter is VARCHAR(256).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example assumes that you want to create a geocoder that takes latitude and longitude as input and geocodes into ST_Point spatial data. To do this, you first create a function named `lat_long_gc_func`. Then you register a geocoder named `SAMPLEGC`, which uses the function `lat_long_gc_func`.

Here is an example of the SQL statement that creates the function `lat_long_gc_func` that returns ST_Point:

```
CREATE FUNCTION lat_long_gc_func(latitude double,
    longitude double, srId integer)
    RETURNS DB2GSE.ST_Point
    LANGUAGE SQL
    RETURN DB2GSE.ST_Point(latitude, longitude, srId)
```

After the function is created, you can register it as a geocoder. This example shows how to use the DB2 command line processor CALL command to invoke the ST_REGISTER_GEOCODER stored procedure to register a geocoder named SAMPLEGC with function lat_long_gc_func:

```
call DB2GSE.ST_REGISTER_GEOCODER ('SAMPLEGC',NULL,'LAT_LONG_GC_FUNC',',,1'
    ,NULL,'My Company','Latitude/Longitude to
    ST_Point Geocoder'?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_REGISTER_SPATIAL_COLUMN procedure

Use this stored procedure to register a spatial column and to associate a spatial reference system (SRS) with it.

You can also use this stored procedure to compute the geographic extents of the spatial column.

After this stored procedure is processed, information about the registered spatial column and geographic extents is available from the DB2GSE.ST_GEOMETRY_COLUMNS catalog view. Registering a spatial column creates a constraint on the table, if possible, to ensure that all geometries use the specified SRS.

Authorization

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

- DBADM authority on the database that contains the table to which the spatial column that is being registered belongs
- CONTROL privilege on this table
- ALTER privilege on this table

Syntax

```
►►—DB2GSE.ST_REGISTER_SPATIAL_COLUMN—(—table_schema—,—table_name—,—►
    |—null—|
►—column_name—,—srs_name—,—|—compute_extents—,—msg_code—,—msg_text—►
    |—null—|
►—)——►
```

Parameter descriptions

table_schema

Names the schema to which the table or view that is specified in the *table_name*

parameter belongs. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

The *table_schema* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

table_name

Specifies the unqualified name of the table or view that contains the column that is being registered. You must specify a non-null value for this parameter.

The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

column_name

Names the column that is being registered. You must specify a non-null value for this parameter.

The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

srs_name

Names the spatial reference system that is to be used for this spatial column. You must specify a non-null value for this parameter.

The *srs_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

compute_extents

Indicates whether to compute the geographic extents of a specified column and make them available through the DB2GSE.ST_GEOMETRY_COLUMNS catalog view. The possible values for this parameter are:

- A value greater than 0 to compute the geographic extents.
- Null, 0, or a negative value to prevent this computation.

If you omit this parameter, it has the same effect as specifying 0. The extent computation is not performed.

The data type of this parameter is INTEGER.

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_REGISTER_SPATIAL_COLUMN stored procedure. This example uses a DB2 CALL command to register the spatial column named LOCATION in the table named CUSTOMERS. This CALL command specifies the *srs_name* parameter value as USA_SRS_1:

```
call DB2GSE.ST_REGISTER_SPATIAL_COLUMN(NULL, 'CUSTOMERS', 'LOCATION',  
    'USA_SRS_1', ?, ?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_REMOVE_GEOCODING_SETUP procedure

Use this stored procedure to remove all the geocoding setup information for the geocoded column.

This stored procedure removes information that is associated with the specified geocoded column from the DB2GSE.ST_GEOCODING and DB2GSE.ST_GEOCODING_PARAMETERS catalog views.

Restriction:

You cannot remove a geocoding setup if autogeocoding is enabled for the geocoded column.

Authorization

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

- DATAACCESS authority on the database that contains the table on which the specified geocoder is to operate
- CONTROL privilege on this table
- UPDATE privilege on this table

Syntax

```
►► DB2GSE.ST_REMOVE_GEOCODING_SETUP—(—table_schema—,—table_name—,—column_name—,—msg_code—,—msg_text—)—►►  
                                          └─null─┘
```

Parameter descriptions

table_schema

Names the schema to which the table or view that is specified in the *table_name*

parameter belongs. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

The *table_schema* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

table_name

Specifies the unqualified name of the table or view that contains the column into which the geocoded data is to be inserted or updated. You must specify a non-null value for this parameter.

The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

column_name

Names the column into which the geocoded data is to be inserted or updated. You must specify a non-null value for this parameter.

The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type for this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_REMOVE_GEOCODING_SETUP stored procedure. This example uses a DB2 CALL command to remove the geocoding setup for the table named CUSTOMER and the column named LOCATION:

```
call DB2GSE.ST_REMOVE_GEOCODING_SETUP(NULL, 'CUSTOMERS', 'LOCATION', ?, ?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_RUN_GEOCODING procedure

Use this stored procedure to run a geocoder in batch mode on a geocoded column.

Authorization

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

- DATAACCESS authority on the database that contains the table on which the specified geocoder is to operate
- CONTROL privilege on this table
- UPDATE privilege on this table

Syntax

```
►► DB2GSE.ST_RUN_GEOCODING—(—table_schema—,—table_name—,——————►  
                                |  
                                null  
►—column_name—,—geocoder_name—,—parameter_values—,——————►  
                                |  
                                null  
►—where_clause—,—commit_scope—,—msg_code—,—msg_text—)—————►  
                                |  
                                null
```

Parameter descriptions

table_schema

Names the schema to which the table or view that is specified in the *table_name* parameter belongs. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

The *table_schema* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

table_name

Specifies the unqualified name of the table or view that contains the column into which the geocoded data is to be inserted or updated. If a view name is specified, the view must be an updatable view. You must specify a non-null value for this parameter.

The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

column_name

Names the column into which the geocoded data is to be inserted or updated. You must specify a non-null value for this parameter.

The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

geocoder_name

Names the geocoder that is to perform the geocoding. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the geocoding is performed by the geocoder that was specified when geocoding was set up.

The *geocoder_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

parameter_values

Specifies the list of geocoding parameter values for the geocoder function. Although you must specify a value for this parameter, the value can be null. If the entire *parameter_values* parameter is null, the values that are used are either the parameter values that were specified when the geocoder was set up or the default parameter values for the geocoder if the geocoder was not set up.

If you specify any parameter values, specify them in the order that the function defined them, and separate them with a comma. For example:

parameter1-value,parameter2-value,...

Each parameter value can be a column name, a string, a numeric value, or null.

Each parameter value is an SQL expression. Follow these guidelines:

- If a parameter value is a geocoding column name, ensure that the column is in the same table or view where the geocoded column resides.
- If a parameter value is a string, enclose it in single quotation marks.
- If a parameter value is a number, do not enclose it in single quotation marks.
- If the parameter is null, cast it to the correct type. For example, instead of specifying just NULL, specify:
CAST(NULL AS INTEGER)

If any parameter value is not specified (that is, if you specify two consecutive commas (...)), this parameter must be specified either when geocoding is set up or when geocoding is run in batch mode with the *parameter_values* parameter of the respective stored procedures.

The data type of this parameter is VARCHAR(32K).

where_clause

Specifies the body of the WHERE clause, which defines a restriction on the set of records that are to be geocoded. Although you must specify a value for this parameter, the value can be null.

If the *where_clause* parameter is null, the resulting behavior depends on whether geocoding was set up for the column (specified in the *column_name* parameter) before the stored procedure runs. If the *where_clause* parameter is null, and:

- A value was specified when geocoding was set up, that value is used for the *where_clause* parameter.
- Either geocoding was not set up or no value was specified when geocoding was set up, no where clause is used.

You can specify a clause that references any column in the table or view that the geocoder is to operate on. Do not specify the keyword WHERE.

The data type of this parameter is VARCHAR(32K).

commit_scope

Specifies that a COMMIT is to be performed after every *n* records that are geocoded. Although you must specify a value for this parameter, the value can be null.

If the *commit_scope* parameter is null, the resulting behavior depends on whether geocoding was set up for the column (specified in the *column_name* parameter) before the stored procedure runs. If the *commit_scope* parameter is null and:

- A value was specified when geocoding was set up for the column, that value is used for the *commit_scope* parameter.
- Either geocoding was not set up or it was set up but no value was specified, the default value of 0 (zero) is used, and a COMMIT is performed at the end of the operation.

The data type of this parameter is INTEGER.

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_RUN_GEOCODING stored procedure. This example uses a DB2 CALL command to geocode the LOCATION column in the table named CUSTOMER. This CALL command specifies the *geocoder_name* parameter value as SAMPLEGC and the *commit_scope* parameter value as 10. A COMMIT is to be performed after every 10 records are geocoded:

```
call DB2GSE.ST_RUN_GEOCODING(NULL, 'CUSTOMERS', 'LOCATION',  
    'SAMPLEGC', NULL, NULL, 10, ?, ?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_SETUP_GEOCODING procedure

Use this stored procedure to associate a column that is to be geocoded with a geocoder and to set up the corresponding geocoding parameters.

Information about the set up is available through the DB2GSE.ST_GEOCODING and DB2GSE.ST_GEOCODING_PARAMETERS catalog views.

This stored procedure does not invoke geocoding. It provides a way for you to specify parameter settings for the column that is to be geocoded. With these settings, the subsequent invocation of either batch geocoding or autogeocoding can be done with a much simpler interface. Parameter settings that are specified in this setup step override any of the default parameter values for the geocoder that were specified when the geocoder was registered. You can also override these parameter settings by running the ST_RUN_GEOCODING procedure in batch mode.

This step is a prerequisite for autogeocoding. You cannot enable autogeocoding without first setting up the geocoding parameters. This step is not a prerequisite for batch geocoding. You can run geocoding in batch mode with or without performing the setup step. However, if the setup step is done prior to batch geocoding, parameter values are taken from the setup time if they are not specified at run time.

Authorization

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

- DATAACCESS authority on the database that contains the table on which the specified geocoder is to operate
- CONTROL privilege on this table
- UPDATE privilege on this table

Syntax

```

>> DB2GSE.ST_SETUP_GEOCODING—( table_schema , table_name ,
                                null
                                , column_name , geocoder_name , parameter_values ,
                                null
                                , autogeocoding_columns , where_clause , commit_scope ,
                                null
                                , msg_code , msg_text )

```

Parameter descriptions

table_schema

Names the schema to which the table or view that is specified in the *table_name* parameter belongs. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

The *table_schema* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

table_name

Specifies the unqualified name of the table or view that contains the column

into which the geocoded data is to be inserted or updated. If a view name is specified, the view must be updatable. You must specify a non-null value for this parameter.

The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

column_name

Names the column into which the geocoded data is to be inserted or updated. You must specify a non-null value for this parameter.

The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

geocoder_name

Names the geocoder that is to perform the geocoding. You must specify a non-null value for this parameter.

The *geocoder_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

parameter_values

Specifies the list of geocoding parameter values for the geocoder function. Although you must specify a value for this parameter, the value can be null. If the entire *parameter_values* parameter is null, the values that are used are taken from the default parameter values at the time the geocoder was registered.

If you specify parameter values, specify them in the order that the function defined them, and separate them with a comma. For example:

parameter1-value,parameter2-value,...

Each parameter value is an SQL expression and can be a column name, a string, a numeric value, or null. Follow these guidelines:

- If a parameter value is a geocoding column name, ensure that the column is in the same table or view where the geocoded column resides.
- If a parameter value is a string, enclose it in single quotation marks.
- If a parameter value is a number, do not enclose it in single quotation marks.
- If the parameter value is specified as a null value, cast it to the correct type. For example, instead of specifying just NULL, specify:

CAST(NULL AS INTEGER)

If any parameter value is not specified (that is, if you specify two consecutive commas (...)), this parameter must be specified either when geocoding is set up or when geocoding is run in batch mode with the *parameter_values* parameter of the respective stored procedures.

The data type of this parameter is VARCHAR(32K).

autogeocoding_columns

Specifies the list of column names on which the trigger is to be created.

Although you must specify a value for this parameter, the value can be null. If

this parameter is null and autogeocoding is enabled, an update of any column in the table causes the trigger to be activated.

If you specify a value for the *autogeocoding_columns* parameter, specify column names in any order, and separate column names with a comma. The column name must exist in the same table where the geocoded column resides.

This parameter setting applies only to subsequent autogeocoding.

The data type of this parameter is VARCHAR(32K).

where_clause

Specifies the body of the WHERE clause, which defines a restriction on the set of records that are to be geocoded. Although you must specify a value for this parameter, the value can be null. If this parameter is null, no restrictions are defined in the WHERE clause.

The clause can reference any column in the table or view that the geocoder is to operate on. Do not specify the keyword WHERE.

This parameter setting applies only to subsequent batch-mode geocoding.

The data type of this parameter is VARCHAR(32K).

commit_scope

Specifies that a COMMIT is to be performed for every *n* records that are geocoded. Although you must specify a value for this parameter, the value can be null. If this parameter is null, a COMMIT is performed after all records are geocoded.

This parameter setting applies only to subsequent batch-mode geocoding.

The data type of this parameter is INTEGER.

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_SETUP_GEOCODING stored procedure. This example uses a DB2 CALL command to set up a geocoding process for the geocoded column named LOCATION in the table named CUSTOMER. This CALL command specifies the *geocoder_name* parameter value as SAMPLEGC:

```
call DB2GSE.ST_SETUP_GEOCODING(NULL, 'CUSTOMERS', 'LOCATION',
'SAMPLEGC','ADDRESS,CITY,STATE,ZIP,1,100,80,,,,$HOME/sql1lib/
gse/refdata/ky.edg",$HOME/sql1lib/samples/extenders/spatial/EDGEsample.loc',
'ADDRESS,CITY,STATE,ZIP',NULL,10,?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs

ST_UNREGISTER_GEOCODER procedure

Use this stored procedure to unregister a geocoder.

Restriction:

You cannot unregister a geocoder if it is specified in the geocoding setup for any column.

To determine whether a geocoder is specified in the geocoding setup for a column, check the DB2GSE.ST_GEOCODING and DB2GSE.ST_GEOCODING_PARAMETERS catalog views. To find information about the geocoder that you want to unregister, consult the DB2GSE.ST_GEOCODERS catalog view.

Authorization

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

Syntax

```
►►—DB2GSE.ST_UNREGISTER_GEOCODER—(—geocoder_name—,—msg_code—,—msg_text—►►
►—)———►►
```

Parameter descriptions

geocoder_name

Uniquely identifies the geocoder. You must specify a non-null value for this parameter.

The *geocoder_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_UNREGISTER_GEOCODER stored procedure. This example uses a DB2 CALL command to unregister the geocoder named SAMPLEGC:

```
call DB2GSE.ST_UNREGISTER_GEOCODER('SAMPLEGC',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

ST_UNREGISTER_SPATIAL_COLUMN procedure

Use this stored procedure to remove the registration of a spatial column.

The stored procedure removes the registration by:

- Removing association of the spatial reference system with the spatial column. The ST_GEOMETRY_COLUMNS catalog view continues to contain the spatial column, but the column is no longer associated with any spatial reference system.
- For a base table, dropping the constraint that DB2 Spatial Extender placed on this table to ensure that the geometry values in this spatial column are all represented in the same spatial reference system.

Authorization

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

- DBADM authority
- CONTROL privilege on this table
- ALTER privilege on this table

Syntax

```
►►DB2GSE.ST_UNREGISTER_SPATIAL_COLUMN(—table_schema—,—table_name—,—column_name—,—msg_code—,—msg_text—)►►
```

table_schema can be null.

Parameter descriptions

table_schema

Names the schema to which the table that is specified in the *table_name* parameter belongs. Although you must specify a value for this parameter, the value can be null. If this parameter is null, the value in the CURRENT SCHEMA special register is used as the schema name for the table or view.

The *table_schema* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

table_name

Specifies the unqualified name of the table that contains the column that is specified in the *column_name* parameter. You must specify a non-null value for this parameter.

The *table_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

column_name

Names the spatial column that you want to unregister. You must specify a non-null value for this parameter.

The *column_name* value is converted to uppercase unless you enclose it in double quotation marks.

The data type of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

Output parameters

msg_code

Specifies the message code that is returned from the stored procedure. The value of this output parameter identifies the error, success, or warning condition that was encountered during the processing of the procedure. If this parameter value is for a success or warning condition, the procedure finished its task. If the parameter value is for an error condition, no changes to the database were performed.

The data type of this output parameter is INTEGER.

msg_text

Specifies the actual message text, associated with the message code, that is returned from the stored procedure. The message text can include additional information about the success, warning, or error condition, such as where an error was encountered.

The data type of this output parameter is VARCHAR(1024).

Example

This example shows how to use the DB2 command line processor to invoke the ST_UNREGISTER_SPATIAL_COLUMN stored procedure. This example uses a DB2 CALL command to unregister the spatial column named LOCATION in the table named CUSTOMERS:

```
call DB2GSE.ST_UNREGISTER_SPATIAL_COLUMN(NULL,'CUSTOMERS','LOCATION',?,?)
```

The two question marks at the end of this CALL command represent the output parameters, *msg_code* and *msg_text*. The values for these output parameters are displayed after the stored procedure runs.

Chapter 18. Spatial functions

Use spatial functions to program your applications. Learn about the factors that are common to all or most spatial functions and the function usage by category.

Considerations and associated data types for spatial functions

This section provides information that you need to know when you code spatial functions.

This information includes:

- Factors to consider: the requirement to specify the schema to which spatial functions belong, and the fact that some functions can be invoked as methods.
- How to address a situation in which a spatial function cannot process the type of geometries returned by another spatial function.
- A table showing which functions take values of each spatial data type as input

When you use spatial functions, be aware of these factors:

- Before a spatial function can be called, its name must be qualified by the name of the schema to which spatial functions belong: DB2GSE. One way to do this is to explicitly specify the schema in the SQL statement that references the function; for example:

```
SELECT db2gse.ST_Relate (g1, g2, 'T*F**FFF2') EQUALS FROM relate_test
```

Alternatively, to avoid specifying the schema each time a function is to be called, you can add DB2GSE to the CURRENT FUNCTION PATH special register. To obtain the current settings for this special register, type the following SQL command:

```
VALUES CURRENT FUNCTION PATH
```

To update the CURRENT FUNCTION PATH special register with DB2GSE, issue the following SQL command:

```
set CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

- Some spatial functions can be invoked as methods. In the following code, for example, ST_Area is invoked first as a function and then as a method. In both cases, ST_Area is coded to operate on a polygon that has an ID of 10 and that is stored in the SALES_ZONE column of a table named STORES. When invoked, ST_Area will return the area of the real-world feature-Sales Zone no. 10-that the polygon represents.

ST_Area invoked as a function:

```
SELECT ST_Area(sales_zone)
FROM   stores
WHERE  id = 10
```

ST_Area invoked as a method:

```
SELECT sales_zone..ST_Area()
FROM   stores
WHERE  id = 10
```

The functions ST_BuildMBRAggr and ST_BuildUnionAggr are described in "MBR Aggregate" and "Union Aggregate", respectively.

Treating values of ST_Geometry as values of a subtype

If a spatial function returns a geometry whose static type is a super type, and if the geometry is passed to a function that accepts only geometries of a type that is subordinate to this super type, a compile-time exception is raised.

For example, the static type of the output parameter of the ST_Union function is ST_Geometry, the super type of all spatial data types. The static input parameter for the ST_PointOnSurface function can be either ST_Polygon or ST_MultiPolygon, two subtypes of ST_Geometry. If DB2 Spatial Extender attempts to pass geometries returned by ST_Union to ST_PointOnSurface, DB2 Spatial Extender raises the following compile-time exception:

```
SQL00440N No function by the name "ST_POINTONSURFACE"
having compatible arguments was found in the function
path.      SQLSTATE=42884
```

This message indicates that DB2 Spatial Extender could not find a function that is named ST_PointOnSurface and that has an input parameter of ST_Geometry.

To let geometries of a super type pass to functions that accept only subtypes of the super type, use the TREAT operator. As indicated earlier, ST_Union returns geometries of a static type of ST_Geometry. It can also return geometries of a dynamic subtype of ST_Geometry. Suppose, for example, that it returns a geometry with a dynamic type of ST_MultiPolygon. In that case, the TREAT operator requires that this geometry be used with the static type ST_MultiPolygon. This matches one of the data types of the input parameter of ST_PointOnSurface. If ST_Union does not return an ST_MultiPolygon value, DB2 Spatial Extender raises a runtime exception.

If a function returns a geometry of a super type, the TREAT operator generally can tell DB2 Spatial Extender to regard this geometry as a subtype of this super type. But be aware that this operation succeeds only if the subtype matches or is subordinate to a static subtype defined as an input parameter of the function to which the geometry is passed. If this condition is not met, DB2 Spatial Extender raises a run-time exception.

Consider another example: suppose that you want to determine the perpendicular points for a given point on the boundary of a polygon that has no holes. You use the ST_Boundary function to derive the boundary from the polygon. The static output parameter of ST_Boundary is ST_Geometry, but ST_PerpPoints accepts ST_Curve geometries. Because all polygons have a linestring (which is also a curve) as a boundary, and because the data type of linestrings (ST_LineString) is subordinate to ST_Curve, the following operation will let an ST_Geometry polygon returned by ST_Boundary pass to ST_PerpPoints:

```
SELECT ST_AsText(ST_PerpPoints(TREAT(ST_Boundary(polygon) as ST_Curve)),
      ST_Point(30.5, 65.3, 1)))
FROM   polygon_table
```

Instead of invoking ST_Boundary and ST_PerpPoints as functions, you can invoke them as methods. To do so, specify the following code:

```
SELECT TREAT(ST_Boundary(polygon) as ST_Curve)..
      ST_PerpPoints(ST_Point(30.5, 65.3, ))..ST_AsText()
FROM   polygon_table
```

Spatial functions according to input type

List of spatial functions according to the type of input that they can accept.

Important: As noted elsewhere, the spatial data types form a hierarchy, with ST_Geometry as the root. When the documentation for DB2 Spatial Extender indicates that a value of a super type in this hierarchy can be used as input to a function, alternatively, a value of any subtype of this super type can also be used as input to the function.

For example, the first entries in Table 24 indicate that ST_Area and a number of other functions can take values of the ST_Geometry data type as input. Therefore, input to these functions can also be values of any subtype of ST_Geometry: ST_Point, ST_Curve, ST_LineString, and so on.

Table 24. Spatial functions listed according to input type

Data type of input parameter	Function
ST_Geometry	EnvelopesIntersect
	ST_Area
	ST_AsBinary
	ST_AsGML
	ST_AsShape
	ST_AsText
	ST_Boundary
	ST_Buffer
	ST_BuildMBrAggr
	ST_BuildUnionAggr
	ST_Centroid
	ST_Contains
	ST_ConvexHull
	ST_CoordDim
	ST_Crosses
	ST_Difference
	ST_Dimension
	ST_Disjoint
	ST_Distance
	ST_Envelope
	ST_EnvIntersects
	ST_Equals
	ST_FindMeasure or ST_LocateAlong
	ST_Generalize
	ST_GeometryType

Table 24. Spatial functions listed according to input type (continued)

Data type of input parameter	Function
ST_Geometry, continued	ST_Intersection ST_Intersects ST_Is3D ST_IsEmpty ST_IsMeasured ST_IsSimple ST_IsValid ST_MaxM ST_MaxX ST_MaxY ST_MaxZ ST_MBR ST_MBRIntersects ST_MeasureBetween or ST_LocateBetween ST_MinM ST_MinX ST_MinY ST_MinZ ST_NumPoints ST_Overlaps ST_Relate ST_SRID or ST_SrsId ST_SrsName ST_SymDifference ST_ToGeomColl ST_ToLineString ST_ToMultiLine ST_ToMultiPoint ST_ToMultiPolygon ST_ToPoint ST_ToPolygon ST_Touches ST_Transform ST_Union ST_Within
ST_Point	ST_M ST_X ST_Y ST_Z
ST_Curve	ST_AppendPoint ST_ChangePoint ST_EndPoint ST_IsClosed ST_IsRing ST_Length ST_MidPoint ST_PerpPoints ST_RemovePoint ST_StartPoint
ST_LineString	ST_PointN ST_Polygon

Table 24. Spatial functions listed according to input type (continued)

Data type of input parameter	Function
ST_Surface	ST_Perimeter ST_PointOnSurface
ST_GeomCollection	ST_GeometryN ST_NumGeometries ST_PointN
ST_MultiPoint	ST_IsClosed ST_Length ST_PerpPoints
ST_MultiCurve	ST_LineStringN ST_NumLineStrings ST_Polygon
ST_MultiLineString	ST_Perimeter ST_PointOnSurface
ST_MultiSurface	ST_NumPolygons ST_PolygonN
ST_MultiPolygon	

Categories and uses for spatial functions

This section introduces all the spatial functions and organizes them by category.

The DB2 Spatial Extender provides functions that:

- Convert geometries to and from various data exchange formats. These functions are called constructor functions.
- Compare geometries for boundaries, intersections, and other information. These functions are called comparison functions.
- Return information about properties of geometries, such as coordinates and measures within geometries, relationships between geometries, and boundary and other information.
- Generate new geometries from existing geometries.
- Measure the shortest distance between points in geometries.
- Provide information about index parameters.
- Provide projections and conversions between different coordinate systems.

Constructor functions for converting to and from data exchange formats

DB2 Spatial Extender provides spatial functions that convert geometries to and from data exchange formats.

The supported data exchange formats are:

- Well-known text (WKT) representation
- Well-known binary (WKB) representation
- ESRI shape representation
- Geography Markup Language (GML) representation

The functions for creating geometries from these formats are known as *constructor functions*. Constructor functions have the same name as the geometry data type of the column into which the data will be inserted. These functions operate consistently on each of the input data exchange formats. This section provides:

- The SQL for calling functions that operate on data exchange formats, and the type of geometry returned by these functions
- The SQL for calling a function that creates points from X and Y coordinates, and the type of geometry returned by this function
- Examples of code and result sets

Functions that convert from data exchange formats to geometries

Converting geometry representations from data exchange formats to geometry values allow representations to be exchanged as geometry values.

Functions that convert from geometries to well-known text (WKT) representation

Converting geometries to WKT representations allow geometries to be exchanged in ASCII text form. The WKT representations are CLOB values representing ASCII character strings.

The **ST_AsText** function converts a geometry value stored in a table to a WKT string. The following example uses a simple command-line query to select the values that were previously inserted into the SAMPLE_GEOMETRY table.

```
SELECT id, VARCHAR(db2gse.ST_AsText(geom), 50) AS WKTGEOM
FROM sample_geometry;
```

```
ID    WKTGEOM
-----
100    POINT ( 30.000000000 40.000000000)
200    LINESTRING ( 50.000000000 50.000000000, 100.000000000 100.000000000)
```

The following example uses embedded SQL to select the values that were previously inserted into the SAMPLE_GEOMETRY table.

```
EXEC SQL BEGIN DECLARE SECTION;
sqlint32 id = 0;
SQL TYPE IS CLOB(10000) wkt_buffer;
short wkt_buffer_ind = -1;
EXEC SQL END DECLARE SECTION;

EXEC SQL
SELECT id, db2gse.ST_AsText(geom)
INTO :id, :wkt_buffer :wkt_buffer_ind
FROM sample_geometry
WHERE id = 100;
```

Alternatively, you can use the ST_WellKnownText transform group to implicitly convert geometries to their well-known text representation when binding them out. The following sample code shows how to use the transform group.

```
EXEC SQL BEGIN DECLARE SECTION;
sqlint32 id = 0;
SQL TYPE IS CLOB(10000) wkt_buffer;
short wkt_buffer_ind = -1;
EXEC SQL END DECLARE SECTION;

EXEC SQL
SET CURRENT DEFAULT TRANSFORM GROUP = ST_WellKnownText;

EXEC SQL
```

```

SELECT id, geom
INTO :id, :wkt_buffer :wkt_buffer_ind
FROM sample_geometry
WHERE id = 100;

```

No spatial function is used in the SELECT statement to convert the geometry.

In addition to the functions explained in this section, DB2 Spatial Extender provides other functions that also convert geometries to and from well-known text representations. DB2 Spatial Extender provides these other functions to implement the OGC “Simple Features for SQL” specification and the ISO SQL/MM Part 3: Spatial standard. These functions are:

- **ST_WKTToSQL**
- **ST_GeomFromText**
- **ST_GeomCollFromText**
- **ST_PointFromText**
- **ST_LineFromText**
- **ST_PolyFromText**
- **ST_MPointFromText**
- **ST_MLineFromText**
- **ST_MPolyFromText**

Functions that convert from geometries to well-known binary (WKB) representation

Converting geometries to WKB representations allow geometries to be exchanged in binary form. The WKB representation consists of binary data structures that must be BLOB values. These BLOB values represent binary data structures that must be managed by an application program written in a programming language that DB2 supports and for which DB2 has a language binding.

The **ST_AsBinary** function converts a geometry value stored in a table to the well-known binary (WKB) representation, which can be fetched into a BLOB variable in program storage. The following example uses embedded SQL to select the values that were previously inserted into the SAMPLE_GEOMETRY table.

```

EXEC SQL BEGIN DECLARE SECTION;
    sqlint32 id = 0;
    SQL TYPE IS BLOB(10000) wkb_buffer;
    short wkb_buffer_ind = -1;
EXEC SQL END DECLARE SECTION;

EXEC SQL
    SELECT id, db2gse.ST_AsBinary(geom)
    INTO :id, :wkb_buffer :wkb_buffer_ind
    FROM sample_geometry
    WHERE id = 200;

```

Alternatively, you can use the ST_WellKnownBinary transform group to implicitly convert geometries to their well-known binary representation when binding them out. The following sample code shows how to use this transform group.

```

EXEC SQL BEGIN DECLARE SECTION;
    sqlint32 id = 0;
    SQL TYPE IS BLOB(10000) wkb_buffer;
    short wkb_buffer_ind = -1;
EXEC SQL END DECLARE SECTION;

EXEC SQL
    SET CURRENT DEFAULT TRANSFORM GROUP = ST_WellKnownBinary;

```

```
EXEC SQL
  SELECT id, geom
  INTO :id, :wkb_buffer :wkb_buffer_ind
  FROM sample_geometry
  WHERE id = 200;
```

No spatial function is used in the `SELECT` statement to convert the geometry.

In addition to the functions explained in this section, there are other functions that also convert geometries to and from well-known binary representations. DB2 Spatial Extender provides these other functions to implement the OGC “Simple Features for SQL” specification and the ISO SQL/MM Part 3: Spatial standard. These functions are:

- **ST_WKBTToSQL**
- **ST_GeomFromWKB**
- **ST_GeomCollFromWKB**
- **ST_PointFromWKB**
- **ST_LineFromWKB**
- **ST_PolyFromWKB**
- **ST_MPointFromWKB**
- **ST_MLineFromWKB**
- **ST_MPolyFromWKB**

Functions that convert from geometries to ESRI shape representation

Converting geometries to ESRI shape representations allow geometries to be exchanged in binary form. The ESRI Shape representation consists of binary data structures that must be managed by an application program written in a supported language.

The **ST_AsShape** function converts a geometry value stored in a table to the ESRI Shape representation, which can be fetched into a BLOB variable in program storage. The following example uses embedded SQL to select the values that were previously inserted into the `SAMPLE_GEOMETRY` table.

```
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 id;
  SQL TYPE IS BLOB(10000) shape_buffer;
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL
  SELECT id, db2gse.ST_AsShape(geom)
  INTO :id, :shape_buffer
  FROM sample_geometry;
```

Alternatively, you can use the `ST_Shape` transform group to implicitly convert geometries to their shape representation when binding them out. The following sample code shows how to use the transform group.

```
EXEC SQL BEGIN DECLARE SECTION;
  sqlint32 id = 0;
  SQL TYPE IS BLOB(10000) shape_buffer;
  short shape_buffer_ind = -1;
EXEC SQL END DECLARE SECTION;

EXEC SQL
  SET CURRENT DEFAULT TRANSFORM GROUP = ST_Shape;
```



```
EXEC SQL
  SELECT id, geom
  FROM sample_geometry
  WHERE id = 300;
```

No spatial function is used in the SELECT statement to convert the geometry.

Functions that convert from geometries to Geography Markup Language (GML) representation

Converting geometries to GML representations allow geometries to be exchanged in ASCII text form. GML representations are ASCII strings.

The **ST_AsGML** function converts a geometry value stored in a table to a GML text string. The following example selects the values that were previously inserted into the SAMPLE_GEOMETRY table. The results shown in the example have been reformatted for readability. The spacing in your results might vary according to your online display.

```
SELECT id, VARCHAR(db2gse.ST_AsGML(geom), 500) AS GMLGEOM
FROM sample_geometry;
```

ID	GMLGEOM
100	<gml:Point srsName="EPSG:4269"> <gml:coord><gml:X>30</gml:X><gml:Y>40</gml:Y></gml:coord> </gml:Point>
200	<gml:LineString srsName="EPSG:4269"> <gml:coord><gml:X>50</gml:X><gml:Y>50</gml:Y></gml:coord> <gml:coord><gml:X>100</gml:X><gml:Y>100</gml:Y></gml:coord> </gml:LineString>

Alternatively, you can use the ST_GML transform group to implicitly convert geometries to their HTML representation when binding them out.

```
SET CURRENT DEFAULT TRANSFORM GROUP = ST_GML
```

```
SELECT id, geom AS GMLGEOM
FROM sample_geometry;
```

ID	GMLGEOM
100	<gml:Point srsName="EPSG:4269"> <gml:coord><gml:X>30</gml:X><gml:Y>40</gml:Y></gml:coord> </gml:Point>
200	<gml:LineString srsName="EPSG:4269"> <gml:coord><gml:X>50</gml:X><gml:Y>50</gml:Y></gml:coord> <gml:coord><gml:X>100</gml:X><gml:Y>100</gml:Y></gml:coord> </gml:LineString>

No spatial function is used in the SELECT statement to convert the geometry.

Function that creates geometries from coordinates

The **ST_Point** function creates geometries not only from data exchange formats, but also from numeric coordinate values. This capability is very useful if your location data is already stored in your database.

Examples of calling constructor functions

Review the examples of code for calling constructor functions, creating tables to contain the output of constructor functions, and retrieving the output to learn how to use constructor functions.

The following example inserts a row into the SAMPLE_GEOMETRY table with ID 100 and a point value with an X coordinate of 30, a Y coordinate of 40, and in spatial reference system 1 using the coordinate representation and well-known text (WKT) representation. It then inserts another row with ID 200 and a linestring value with the coordinates indicated.

```
CREATE TABLE sample_geometry (id INT, geom db2gse.ST_Geometry);

INSERT INTO sample_geometry(id, geom)
VALUES(100,db2gse.ST_Geometry('point(30 40)', 1));

INSERT INTO sample_geometry(id, geom)
VALUES(200,db2gse.ST_Geometry('linestring(50 50, 100 100', 1));

SELECT id, TYPE_NAME(geom) FROM sample_geometry
```

ID	2
100	"ST_POINT"
200	"ST_LINESTRING"

If you know that the spatial column can only contain ST_Point values, you can use the following example, which inserts two points. Attempting to insert a linestring or any other type which is not a point results in an SQL error. The first insert creates a point geometry from the well-known-text representation (WKT). The second insert creates a point geometry from numeric coordinate values. Note that these input values could also be selected from existing table columns.

```
CREATE TABLE sample_points (id INT, geom db2gse.ST_Point);

INSERT INTO sample_points(id, geom)
VALUES(100,db2gse.ST_Point('point(30 40)', 1));

INSERT INTO sample_points(id, geom)
VALUES(101,db2gse.ST_Point(50, 50, 1));

SELECT id, TYPE_NAME(geom) FROM sample_geometry
```

ID	2
100	"ST_POINT"
101	"ST_POINT"

The following example uses embedded SQL and assumes that the application fills the data areas with the appropriate values.

```
EXEC SQL BEGIN DECLARE SECTION;
    sqlint32 id = 0;
    SQL TYPE IS CLOB(10000) wkt_buffer;
    SQL TYPE IS CLOB(10000) gml_buffer;
    SQL TYPE IS BLOB(10000) wkb_buffer;
    SQL TYPE IS BLOB(10000) shape_buffer;
EXEC SQL END DECLARE SECTION;

// * Application logic to read into buffers goes here */

EXEC SQL INSERT INTO sample_geometry(id, geom)
VALUES(:id, db2gse.ST_Geometry(:wkt_buffer,1));

EXEC SQL INSERT INTO sample_geometry(id, geom)
VALUES:id, db2gse.ST_Geometry(:wkb_buffer,1));

EXEC SQL INSERT INTO sample_geometry(id, geom)
VALUES(:id, db2gse.ST_Geometry(:gml_buffer,1));
```

```
EXEC SQL INSERT INTO sample_geometry(id, geom)
VALUES(:id, db2gse.ST_Geometry(:shape_buffer,1));
```

The following sample Java™ code uses JDBC to insert point geometries using X, Y numeric coordinate values and uses the WKT representation to specify the geometries.

```
String ins1 = "INSERT into sample_geometry (id, geom)
VALUES(?, db2gse.ST_PointFromText(CAST( ?
as VARCHAR(128)), 1))";
PreparedStatement pstmt = con.prepareStatement(ins1);
pstmt.setInt(1, 100);           // id value
pstmt.setString(2, "point(32.4 50.7)"); // wkt value
int rc = pstmt.executeUpdate();

String ins2 = "INSERT into sample_geometry (id, geom)
VALUES(?, db2gse.ST_Point(CAST( ? as double),
CAST(? as double), 1))";
pstmt = con.prepareStatement(ins2);
pstmt.setInt(1, 200);           // id value
pstmt.setDouble(2, 40.3);       // lat
pstmt.setDouble(3, -72.5);      // long
rc = pstmt.executeUpdate();
```

Comparison functions for geographic features

Certain spatial functions return information about ways in which geographic features relate to one another or compare with one another. Other spatial functions return information as to whether two definitions of coordinate systems or two spatial reference systems are the same.

In all cases, the information returned is a result of a comparison between geometries, between definitions of coordinate systems, or between spatial reference systems. The functions that provide this information are called comparison functions. The following table lists comparison functions by purpose.

Table 25. Comparison functions by purpose

Purpose	Functions
Determine whether the interior of one geometry intersects the interior of another.	<ul style="list-style-type: none"> ST_Contains ST_Within
Return information about intersections of geometries.	<ul style="list-style-type: none"> ST_Crosses ST_Intersects ST_Overlaps ST_Touches
Determine whether the smallest rectangle that encloses one geometry intersects with the smallest rectangle that encloses another geometry.	<ul style="list-style-type: none"> ST_EnvIntersects ST_MBRIntersects
Determine whether two objects are identical.	<ul style="list-style-type: none"> ST_Equals ST_EqualCoordsys ST_EqualSRS
Determines whether the geometries being compared meet the conditions of the DE-9IM pattern matrix string.	<ul style="list-style-type: none"> ST_Relate
Checks whether an intersection exists between two geometries.	<ul style="list-style-type: none"> ST_Disjoint

In DB2 Spatial Extender, comparison functions return a value of 1 (one) if a comparison meets certain criteria, a value of 0 (zero) if a comparison fails to meet the criteria, and a null value if the comparison could not be performed.

Comparisons cannot be performed if the comparison operation has not been defined for the input parameters, or if either of the parameters is null. Comparisons can be performed if geometries with different data types or dimensions are assigned to the parameters.

The Dimensionally Extended 9 Intersection Model (DE-9IM) 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 interiors, boundaries, and exteriors, 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 , respectively. 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 $E(b)$ produce 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 returns 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.

Results returned by comparison functions can be understood or verified by comparing the results returned by a comparison function 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, or 2.
- F** An intersection must not exist; `dim` = -1.
- *** It does not matter if an intersection exists; `dim` = -1, 0, 1, or 2.
- 0** An intersection must exist and its exact 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.

The `ST_Within` function returns a value of 1 when the interiors of both geometries intersect and when the interior or boundary of a does not intersect the exterior of b . All other conditions do not matter.

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

The DE-9IM was developed by Clementini and Felice, who dimensionally extended the 9 Intersection Model of Egenhofer and Herring. The DE-9IM is a collaboration of four authors (Clementini, Eliseo, Di Felice, and van Osstroom) who 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 M. J. Egenhofer and J. Herring (Springer-Verlag Singapore [1993]) 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.

Functions that determine whether a geometry contains another one

ST_Contains and ST_Within both take two geometries as input and determine whether the interior of one intersects the interior of the other.

In colloquial terms, ST_Contains determines whether the first geometry given to it encloses the second geometry (whether the first contains the second). ST_Within determines whether the first geometry is completely inside the second (whether the first is within the second).

Functions that determine whether geometries intersect

ST_Intersects, ST_Crosses, ST_Overlaps, and ST_Touches function determine whether one geometry intersects another.

These functions differ mainly as to the scope of intersection that they test for:

- ST_Intersects tests to determine whether the two geometries given to it meet one of four conditions: that the geometries' interiors intersect, that their boundaries intersect, that the boundary of the first geometry intersects with the interior of the second, or that the interior of the first geometry intersects with the boundary of the second.
- ST_Crosses is used to analyze the intersection of geometries of different dimensions, with one exception: it can also analyze the intersection of linestrings. In all cases, the place of intersection is itself considered a geometry; and ST_Crosses requires that this geometry be of a lesser dimension than the greater of the intersecting geometries (or, if both are linestrings, that the place of intersection be of a lesser dimension than a linestring). For example, the dimensions of a linestring and polygon are 1 and 2, respectively. If two such geometries intersect, and if the place of intersection is linear (the linestring's path along the polygon), then that place can itself be considered a linestring. And because a linestring's dimension (1) is lesser than a polygon's (2), ST_Crosses, after analyzing the intersection, would return a value of 1.
- The geometries given to ST_Overlaps as input must be of the same dimension. ST_Overlaps requires that these geometries overlap part-way, forming a new geometry (the region of overlap) that is the same dimension as they are.
- ST_Touches determines whether the boundaries of two geometries intersect.

Functions that compare geometry envelopes

ST_EnvIntersects and ST_MBRIntersects are similar functions that determine whether the smallest rectangle that encloses one geometry intersects with the smallest rectangle that encloses another geometry. Such a rectangle has traditionally been called an envelope.

Multipolygons, polygons, multilinestrings, and crooked linestrings abut against the sides of their envelopes; horizontal linestrings, vertical linestrings, and points are slightly smaller than their envelopes. ST_EnvIntersects tests to determine whether envelopes of geometries intersect.

The smallest rectangular area into which a geometry can fit is called a minimum bounding rectangle (MBR). The envelopes surrounding multipolygons, polygons, multilinestrings, and crooked linestrings are actually MBRs. But the envelopes surrounding horizontal linestrings, vertical linestrings, and points are not MBRs, because they do not constitute a minimum area in which these latter geometries fit. These latter geometries occupy no definable space and therefore cannot have MBRs. Nevertheless, a convention has been adopted whereby they are referred to as their own MBRs. Therefore, with respect to multipolygons, polygons, multilinestrings, and crooked linestrings, `ST_MBRIntersects` tests the intersection of the same surrounding rectangles that `ST_EnvIntersects` tests. But for horizontal linestrings, vertical linestrings, and points, `ST_MBRIntersects` tests the intersections of these geometries themselves.

Functions that check whether two items are identical

These functions compare spatial reference systems, coordinate system definitions, or geometries.

- `ST_EqualCoordsys`
- `ST_Equals`
- `ST_EqualSRS`

Function that determines whether geometries do not intersect

`ST_Disjoint` returns a value of 1 (one) if the intersection of the two geometries is an empty set. This function returns the exact opposite of what `ST_Intersects` returns.

The illustration shows different geometries and how the boundaries do not intersect at any point.

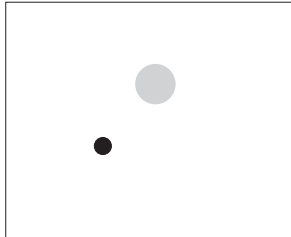
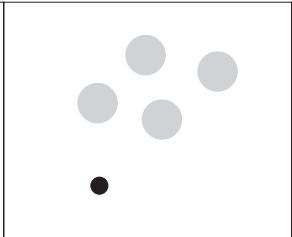
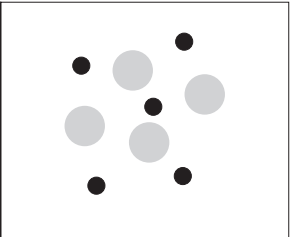
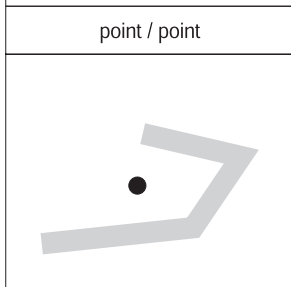
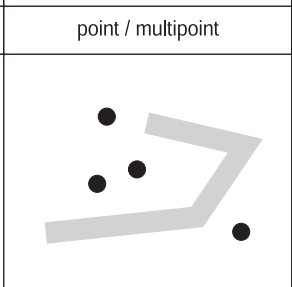
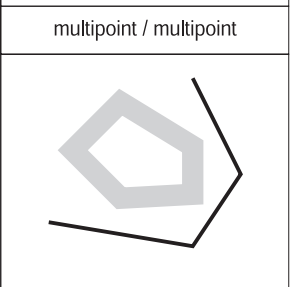
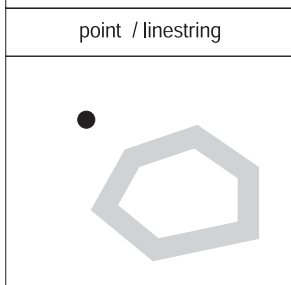
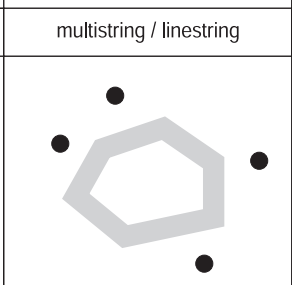
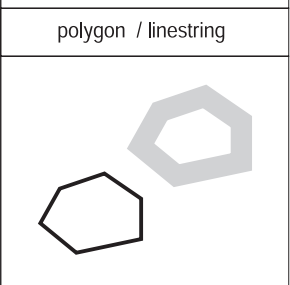
		
point / point	point / multipoint	multipoint / multipoint
		
point / linestring	multistring / linestring	polygon / linestring
		
point / polygon	multipoint / multipolygon	polygon / polygon

Figure 17. *ST_Disjoint*. The dark geometries represent geometry a; the gray geometries represent geometry b. In all cases, geometry a and geometry b are disjoint from one another.

The following matrix simply states that neither the interiors nor the boundaries of either geometry intersect.

Table 26. Matrix for *ST_Disjoint*

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

Function that compares two geometries with the DE-9IM pattern matrix string

The *ST_Relate* function compares two geometries and returns a value of 1 (one) if the geometries meet the conditions specified by the DE-9IM pattern matrix string; otherwise, the function returns a value of 0 (zero).

Functions for obtaining information about geometries and indexes

DB2 Spatial Extender provides functions that return information about properties of geometries and spatial indexes.

The information returned concerns:

- Data types of geometries
- Coordinates and measures within a geometry
- Rings, boundaries, envelopes, and minimum bounding rectangles (MBRs)
- Dimensions
- The qualities of being closed, empty, or simple
- Base geometries within a geometry collection
- Spatial reference systems
- Distance between geometries
- Parameters used to define a spatial index or an index on a spatial column

Some properties are geometries in their own right; for example, the exterior and interior rings of a surface, or the start- and endpoints of a curve. These geometries are produced by some of the functions in this category. Functions that produce other kinds of geometries—for example, geometries that represent zones that surround a given location—belong to the “Spatial functions that generate new geometries” category.

Function that returns data-type information

`ST_GeometryType` takes a geometry as an input parameter and returns the fully-qualified type name of the dynamic type of that geometry.

Functions that return coordinate and measure information

The following functions return information about the coordinates and measures within a geometry. For example, `ST_X` can return the X coordinate within a specified point, `ST_MaxX` returns the highest X coordinate within a geometry, and `ST_MinX` returns the lowest X coordinate within a geometry.

These functions are:

- `ST_CoordDim`
- `ST_IsMeasured`
- `ST_IsValid`
- `ST_Is3D`
- `ST_M`
- `ST_MaxM`
- `ST_MaxX`
- `ST_MaxY`
- `ST_MaxZ`
- `ST_MinM`
- `ST_MinX`
- `ST_MinY`
- `ST_MinZ`
- `ST_X`
- `ST_Y`
- `ST_Z`

Functions that return information about geometries within a geometry

The following functions return information about geometries within a geometry. Some functions identify specific points within a geometry; others return the number of base geometries within a collection.

These functions are:

- ST_Centroid
- ST_EndPoint
- ST_GeometryN
- ST_LineStringN
- ST_MidPoint
- ST_NumGeometries
- ST_NumLineStrings
- ST_NumPoints
- ST_NumPolygons
- ST_PointN
- ST_PolygonN
- ST_StartPoint

Functions that return information about boundaries, envelopes, and rings

The following functions return information about demarcations that divide an inner part of a geometry from an outer part, or that divide the geometry itself from the space external to it. For example, ST_Boundary returns a geometry's boundary in the form of a curve.

These functions are:

- ST_Boundary
- ST_Envelope
- ST_EnvIntersects
- ST_ExteriorRing
- ST_InteriorRingN
- ST_MBR
- ST_MBRIntersects
- ST_NumInteriorRing
- ST_Perimeter

Functions that return information about a geometry dimensions

The following functions return information about the dimension of a geometry such as, area of a given geometry covers or length of a given curve or multcurve.

These functions are:

- ST_Area
- ST_Dimension
- ST_Length

Functions that indicate whether a geometry is closed, empty, or simple

DB2 Spatial Extender provides functions that indicate whether a geometry is closed, empty, or simple.

These functions indicate:

- Whether a given curve or multcurve is closed (that is, whether the start point and end point of the curve or multcurve are the same)
- Whether a given geometry is empty (that is, devoid of points)

- Whether a curve, multicurve, or multipoint is simple (that is, whether such geometries have typical configurations)

These functions are:

- ST_IsClosed
- ST_IsEmpty
- ST_IsSimple

Functions that identify the spatial reference system associated to a geometry

The following functions return values that identify the spatial reference system that has been associated with the geometry. In addition, the ST_SrsID function can change the geometry's spatial reference system without changing or transforming the geometry.

These functions are:

- ST_SrsId (also called ST_SRID)
- ST_SrsName

Function that returns distance information between geometries

ST_Distance takes two geometries and, optionally, a unit as input parameters and returns the shortest distance between any point in the first geometry to any point in the second geometry, measured in the given units.

If the second geometry is not represented in the same spatial reference system as the first geometry, it will be converted to the other spatial reference system.

If any of the two given geometries is null or is empty, then null is returned.

For example, ST_Distance could report the shortest distance an aircraft must travel between two locations. Figure 18 illustrates this information.

The figure shows a map of the United States with a straight line between points labeled Los Angeles and Chicago.



Figure 18. 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.

Function that returns index information

ST_GetIndexParams takes either the identifier for a spatial index or for a spatial column as an input parameter and returns the parameters used to define the index or the index on the spatial column.

If an additional parameter number is specified, only the parameter identified by the number is returned.

Functions for generating new geometries from existing geometries

This section introduces the category of functions that derive new geometries from existing ones.

This category does not include functions that derive geometries that represent properties of other geometries. Rather, it is for functions that:

- Convert geometries into other geometries
- Create geometries that represent configurations of space
- Derive individual geometries from multiple geometries
- Create geometries based on measures
- Create modifications of geometries

Functions that convert a geometry from supertype to corresponding subtype

The following functions can convert geometries of a supertype into corresponding geometries of a subtype.

For example, the `ST_ToLineString` function can convert a linestring of type `ST_Geometry` into a linestring of `ST_LineString`. Some of these functions can also combine base geometries and geometry collections into a single geometry collection. For example, `ST_ToMultiLine` can convert a linestring and a multilinestring into a single multilinestring.

- `ST_Polygon`
- `ST_ToGeomColl`
- `ST_ToLineString`
- `ST_ToMultiLine`
- `ST_ToMultiPoint`
- `ST_ToMultiPolygon`
- `ST_ToPoint`
- `ST_ToPolygon`

Functions that create new geometries with different space configurations

Using existing geometries as a starting point, the following functions create new geometries that represent circular areas or other configurations of space. For example, given a point that represents the center of a proposed airport, `ST_Buffer` can create a surface that represents, in circular form, the proposed extent of the airport.

These functions are:

- `ST_Buffer`
- `ST_ConvexHull`
- `ST_Difference`
- `ST_Intersection`
- `ST_SymDifference`

Functions that derive a new geometry from multiple ones

The following functions derive individual geometries from multiple geometries. For example, combining two geometries into a single geometry.

- `MBR Aggregate`

- ST_Union
- Union Aggregate

Functions that create new geometry based on existing geometry measures

These functions can create a new geometry based on measures from an existing geometry. They can also return the distance to a location along the geometry.

These functions are:

- ST_DistanceToPoint
- ST_FindMeasure or ST_LocateAlong
- ST_MeasureBetween or ST_LocateBetween
- ST_PointAtDistance

Functions that create modified forms of existing geometries

The following functions create modified forms of existing geometries such as, creating extended versions of existing curves. Each version includes the points in an existing curve plus an additional point.

These functions are:

- ST_AppendPoint
- ST_ChangePoint
- ST_Generalize
- ST_M
- ST_PerpPoints
- ST_RemovePoint
- ST_X
- ST_Y
- ST_Z

Functions that convert geometries between coordinate systems

ST_Transform takes a geometry and a spatial reference system identifier as input parameters and transforms the geometry to be represented in the given spatial reference system.

Projections and conversions between different coordinate systems are performed and the coordinates of the geometries are adjusted accordingly.

EnvelopesIntersect function

Use the EnvelopesIntersect function to determine whether two geometries intersect or a geometry intersects with an envelope defined by the four type DOUBLE values.

EnvelopesIntersect accepts two types of input parameters:

- Two geometries
EnvelopesIntersect returns 1 if the envelope of the first geometry intersects the envelope of the second geometry. Otherwise, 0 (zero) is returned.
- A geometry, four type DOUBLE coordinate values that define the lower-left and upper-right corners of a rectangular window, and the spatial reference system identifier.

EnvelopesIntersect returns 1 if the envelope of the first geometry intersects with the envelope defined by the four type DOUBLE values. Otherwise, 0 (zero) is returned.

Syntax

```
db2gse.EnvelopesIntersect(geometry1, geometry2, rectangular-window)
```

rectangular-window:

```
x_min, y_min, x_max, y_max, srs_id
```

Parameters

geometry1

A value of type ST_Geometry or one of its subtypes that represents the geometry whose envelope is to be tested for intersection with the envelope of either *geometry2* or the rectangular window defined by the four type DOUBLE values.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the geometry whose envelope is to be tested for intersection with the envelope of *geometry1*.

x_min

Specifies the minimum X coordinate value for the envelope. You must specify a non-null value for this parameter.

The data type of this parameter is DOUBLE.

y_min

Specifies the minimum Y coordinate value for the envelope. You must specify a non-null value for this parameter.

The data type of this parameter is DOUBLE.

x_max

Specifies the maximum X coordinate value for the envelope. You must specify a non-null value for this parameter.

The data type of this parameter is DOUBLE.

y_max

Specifies the maximum Y coordinate value for the envelope. You must specify a non-null value for this parameter.

The data type of this parameter is DOUBLE.

srs_id

Uniquely identifies the spatial reference system. The spatial reference system identifier must match the spatial reference system identifier of the geometry parameter. You must specify a non-null value for this parameter.

The data type of this parameter is INTEGER.

Return type

INTEGER

Example

This example creates two polygons that represent counties and then determines if any of them intersect a geographic area specified by the four type DOUBLE values.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE counties (id INTEGER, name CHAR(20), geometry ST_Polygon)

INSERT INTO counties VALUES
    (1, 'County_1', ST_Polygon('polygon((0 0, 30 0, 40 30, 40 35,
    5 35, 5 10, 20 10, 20 5, 0 0))' ,0))

INSERT INTO counties VALUES
    (2, 'County_2', ST_Polygon('polygon((15 15, 15 20, 60 20, 60 15,
    15 15))' ,0))

INSERT INTO counties VALUES
    (3, 'County_3', ST_Polygon('polygon((115 15, 115 20, 160 20, 160 15,
    115 15))' ,0))

SELECT name
FROM counties as c
WHERE EnvelopesIntersect(c.geometry, 15, 15, 60, 20, 0) =1
```

Results:

```
Name
-----
County_1
County_2
```

MBR Aggregate functions

Use the ST_BuildMBRAggr and ST_GetAggrResult functions combined to aggregate a set of geometries from a column into a single geometry. The combination constructs a rectangle that represents the minimum bounding rectangle that encloses all the geometries in the column. Z and M coordinates are discarded when the aggregate is computed.

The following expression is an example that uses the MAX function with the db2gse.ST_BuildMBRAggr spatial function to compute the MBR of the geometries in the columnName column and the db2gse.ST_GetAggrResult spatial function to return the resulting geometry that was computed for the MBR:

```
db2gse.ST_Get_AggrResult(MAX(db2gse.ST_BuildMBRAggr(columnName)))
```

If all of the geometries to be combined are null, then null is returned. If all of the geometries are either null or empty, then an empty geometry is returned. If the minimum bounding rectangle of all the geometries to be combined results in a point, then this point is returned as an ST_Point value. If the minimum bounding rectangle of all the geometries to be combined results in a horizontal or vertical linestring, then this linestring is returned as an ST_LineString value. Otherwise, the minimum bounding rectangle is returned as an ST_Polygon value.

Syntax

```
►►db2gse.ST_GetAggrResult(—MAX—(—————)—————)—————►
►db2gse.ST_BuildMBRAggr(—geometries—)——)—————►◄
```

Parameter

geometries

A selected column that has a type of ST_Geometry or one of its subtypes and represents all the geometries for which the minimum bounding rectangle is to be computed.

Return type

db2gse.ST_Geometry

Restrictions

You cannot construct the union aggregate of a spatial column in a full-select in any of the following situations:

- In a partitioned database environment
- If GROUP BY clause is used in the full-select.
- If you use a function other than the DB2 aggregate function MAX.

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example shows how to use the ST_BuildMBRAggr function to obtain the maximum bounding rectangle of all of the geometries within a column. In this example, several points are added to the GEOMETRY column in the SAMPLE_POINTS table. The SQL code then determines the maximum bounding rectangle of all of the points put together.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_points (id integer, geometry ST_Point)
```

```
INSERT INTO sample_points (id, geometry)
VALUES
  (1, ST_Point(2, 3, 1)),
  (2, ST_Point(4, 5, 1)),
  (3, ST_Point(13, 15, 1)),
  (4, ST_Point(12, 5, 1)),
  (5, ST_Point(23, 2, 1)),
  (6, ST_Point(11, 4, 1))
```

```
SELECT cast(ST_GetAggrResult(MAX(ST_BuildMBRAggr
  (geometry)))..ST_AsText AS varchar(160))
  AS ";Aggregate_of_Points";
FROM sample_points
```

Results:

Aggregate_of_Points

```
-----
POLYGON (( 2.00000000 2.00000000, 23.00000000 2.00000000,
23.00000000 15.00000000, 2.00000000 15.00000000, 2.00000000
2.00000000))
```

ST_AppendPoint function

The ST_AppendPoint function takes a curve and a point as input parameters and extends the curve by the given point. If the given curve has Z or M coordinates, then the point must also have Z or M coordinates. The resulting curve is represented in the spatial reference system of the given curve.

If the point to be appended is not represented in the same spatial reference system as the curve, it will be converted to the other spatial reference system.

If the given curve is closed or simple, the resulting curve might not be closed or simple anymore. If the given curve or point is null, or if the curve is empty, then null is returned. If the point to be appended is empty, then the given curve is returned unchanged and a warning is raised (SQLSTATE 01HS3).

This function can also be called as a method.

Syntax

►►db2gse.ST_AppendPoint(—*curve*—,—*point*—)—————►►

Parameter

curve A value of type ST_Curve or one of its subtypes that represents the curve to which *point* will be appended.

point A value of type ST_Point that represents the point that is appended to *curve*.

Return type

db2gse.ST_Curve

Examples

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This code creates two linestrings, each with three points.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines(id integer, line ST_Linestring)

INSERT INTO sample_lines VALUES
    (1, ST_LineString('linestring (10 10, 10 0, 0 0 )', 0) )

INSERT INTO sample_lines VALUES
    (2, ST_LineString('linestring z (0 0 4, 5 5 5, 10 10 6)', 0) )
```

Example 1

This example adds the point (5, 5) to the end of a linestring.

```
SELECT CAST(ST_AsText(ST_AppendPoint(line, ST_Point(5, 5)))
        AS VARCHAR(120)) New
FROM   sample_lines
WHERE  id=1
```

Results:


```

NEW
-----
LINESTRING ( 10.00000000 10.00000000, 10.00000000 0.00000000,
0.00000000 0.00000000, 5.00000000 5.00000000)

```

Example 2

This example adds the point (15, 15, 7) to the end of a linestring with Z coordinates.

```

SELECT CAST(ST_AsText(ST_AppendPoint(line, ST_Point(15.0, 15.0, 7.0)))
AS VARCHAR(160)) New
FROM   sample_lines
WHERE  id=2

```

Results:

```

NEW
-----
LINESTRING Z ( 0.00000000 0.00000000 4.00000000, 5.00000000
5.00000000 5.00000000, 10.00000000 10.00000000 6.00000000,
15.00000000 15.00000000 7.00000000)

```

ST_Area function

The ST_Area function takes a geometry and, optionally, a unit as input parameters and returns the area covered by the geometry in either the default or given unit of measure.

If the geometry is a polygon or multipolygon, then the area covered by the geometry is returned. The area of points, linestrings, multipoints, and multilinestrings is 0 (zero). If the geometry is null or is an empty geometry, null is returned.

This function can also be called as a method.

Syntax

```

►► db2gse.ST_Area (—geometry— [ , —unit— ] ) ►►

```

Parameters

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry that determines the area.

unit

A VARCHAR(128) value that identifies the units in which the area is measured. The supported units of measure are listed in the DB2GSE.ST_UNITS_OF_MEASURE catalog view.

If the *unit* parameter is omitted, the following rules are used to determine the unit in which the area is measured:

- If *geometry* is in a projected or geocentric coordinate system, the linear unit associated with this coordinate system is used.
- If *geometry* is in a geographic coordinate system, the angular unit associated with this coordinate system is used.

Restrictions on unit conversions: An error (SQLSTATE 38SU4) is returned if any of the following conditions occur:

- The geometry is in an unspecified coordinate system and the *unit* parameter is specified.
- The geometry is in a projected coordinate system and an angular unit is specified.
- The geometry is in a geographic coordinate system, and a linear unit is specified.

Return type

DOUBLE

Examples

Example 1

The spatial analyst needs a list of the area covered by each sales region. The sales region polygons are stored in the SAMPLE_POLYGONS table. The area is calculated by applying the ST_Area function to the geometry column.

```
db2se create_srs se_bank -srsId 4000 -srsName new_york1983 -xOffset 0
      -yOffset 0 -xScale 1 -yScale 1
      -coordsysName NAD_1983_StatePlane_New_York_East_FIPS_3101_Feet
```

```
CREATE TABLE sample_polygons (id INTEGER, geometry ST_POLYGON)
```

```
INSERT INTO sample_polygons (id, geometry)
```

```
VALUES
```

```
(1, ST_Polygon('polygon((0 0, 0 10, 10 10, 10 0, 0 0))', 4000) ),
(2, ST_Polygon('polygon((20 0, 30 20, 40 0, 20 0))', 4000) ),
(3, ST_Polygon('polygon((20 30, 25 35, 30 30, 20 30))', 4000))
```

The following SELECT statement retrieves the sales region ID and area:

```
SELECT id, ST_Area(geometry) AS area
FROM   sample_polygons
```

Results:

ID	AREA
1	+1.00000000000000E+002
2	+2.00000000000000E+002
3	+2.50000000000000E+001

Example 2

The following SELECT statement retrieves the sales region ID and area in various units:

```
SELECT id,
      ST_Area(geometry) square_feet,
      ST_Area(geometry, 'METER') square_meters,
      ST_Area(geometry, 'STATUTE MILE') square_miles
FROM   sample_polygons
```

Results:

ID	SQUARE_FEET	SQUARE_METERS	SQUARE_MILES
1	+1.00000000000000E+002	+9.29034116132748E+000	+3.58702077598427E-006
2	+2.00000000000000E+002	+1.85806823226550E+001	+7.17404155196855E-006
3	+2.50000000000000E+001	+2.32258529033187E+000	+8.96755193996069E-007

Example 3

This example finds the area of a polygon defined in State Plane coordinates.

The State Plane spatial reference system with an ID of 3 is created with the following command:

```
db2se create_srs SAMP_DB -srsId 3 -srsName z3101a -xOffset 0
      -yOffset 0 -xScale 1 -yScale 1
      -coordsysName NAD_1983_StatePlane_New_York_East_FIPS_3101_Feet
```

The following SQL statements add the polygon, in spatial reference system 3, to the table and determines the area in square feet, square meters, and square miles.

```
SET current function path db2gse;
CREATE TABLE Sample_Poly3 (id integer, geometry ST_Polygon);
INSERT into Sample_Poly3 VALUES
  (1, ST_Polygon('polygon((567176.0 1166411.0,
                           567176.0 1177640.0,
                           637948.0 1177640.0,
                           637948.0 1166411.0,
                           567176.0 1166411.0 ))', 3));
SELECT id, ST_Area(geometry) "Square Feet",
       ST_Area(geometry, 'METER') "Square Meters",
       ST_Area(geometry, 'STATUTE MILE') "Square Miles"
FROM Sample_Poly3;
```

Results:

ID	SQUARE FEET	SQUARE METERS	SQUARE MILES
1	+7.94698788000000E+008	+7.38302286101346E+007	+2.85060106320552E+001

ST_AsBinary function

The ST_AsBinary function takes a geometry as an input parameter and returns its well-known binary representation. The Z and M coordinates are discarded and will not be represented in the well-known binary representation.

If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

```
►► db2gse.ST_AsBinary (—geometry—) ►►
```

Parameter

geometry

A value of type ST_Geometry or one of its subtypes to be converted to the corresponding well-known binary representation.

Return type

BLOB(2G)

Examples

The following code illustrates how to use the ST_AsBinary function to convert the points in the geometry columns of the SAMPLE_POINTS table into well-known binary (WKB) representation in the BLOB column.

```
CREATE TABLE SAMPLE_POINTS (id integer, geometry ST_POINT, wkb BLOB(32K))

INSERT INTO SAMPLE_POINTS (id, geometry)
VALUES
    (1100, ST_Point(10, 20, 1))
```

Example 1

This example populates the WKB column, with an ID of 1111, from the GEOMETRY column, with an ID of 1100.

```
INSERT INTO sample_points(id, wkb)
VALUES (1111,
    (SELECT ST_AsBinary(geometry)
     FROM sample_points
     WHERE id = 1100))

SELECT id, cast(ST_Point(wkb)..ST_AsText AS varchar(35)) AS point
FROM sample_points
WHERE id = 1111
```

Results:

ID	Point
1111	POINT (10.00000000 20.00000000)

Example 2

This example displays the WKB binary representation.

```
SELECT id, substr(ST_AsBinary(geometry), 1, 21) AS point_wkb
FROM sample_points
WHERE id = 1100
```

Results:

ID	POINT_WKB
1100	x'010100000000000000000000024400000000000003440'

ST_AsGML function

The ST_AsGML function takes a geometry as an input parameter and returns its representation using the geography markup language.

If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

```
db2gse.ST_AsGML(geometry, [gmlLevel—integer])
```

Parameter

gmlLevel

An optional parameter specifying the GML specification level to be used for formatting the GML data to be returned. Valid values are:

- 2 - Use GML specification level 2 with the <gml:coordinates> tag.
- 3 - Use GML specification level 3 with the <gml:poslist> tag.

If no parameter is specified, the output is returned using GML specification level 2 with the <gml:coord> tag.

geometry

A value of type ST_Geometry or one of its subtypes to be converted to the corresponding GML representation.

Return type

CLOB(2G)

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The following code fragment illustrates how to use the ST_AsGML function to view the GML fragment. This example populates the GML column, from the geometry column, with an ID of 2222.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE SAMPLE_POINTS (id integer, geometry ST_POINT, gml CLOB(32K))

INSERT INTO SAMPLE_POINTS (id, geometry)
VALUES
    (1100, ST_Point(10, 20, 1))

INSERT INTO sample_points(id, gml)
VALUES (2222,
    (SELECT ST_AsGML(geometry)
     FROM   sample_points
     WHERE  id = 1100))
```

The following SELECT statement lists the ID and the GML representations of the geometry.

```
SELECT id, cast(ST_AsGML(geometry) AS varchar(110)) AS gml_fragment
FROM   sample_points
WHERE  id = 1100
```

Results:

```
SELECT id,
    cast(ST_AsGML(geometry) AS varchar(110)) AS gml,
    cast(ST_AsGML(geometry,2) AS varchar(110)) AS gml2,
    cast(ST_AsGML(geometry,3) AS varchar(110)) AS gml3
FROM   sample_points
WHERE  id = 1100
```

The SELECT statement returns the following result set:

ID	GML	GML2	GML3
1100	<gml:Point srsName="EPSG:4269"> <gml:coord> <gml:X>10</gml:X><gml:Y>20</gml:Y> </gml:coord></gml:Point>	<gml:Point srsName="EPSG:4269"> <gml:coordinates> 10,20 </gml:coordinates></gml:Point>	<gml:Point srsName="EPSG:4269 srsDimension="2"> <gml:pos> 10,20 </gml:pos></gml:Point>

ST_AsShape function

The St_AsShape function takes a geometry as an input parameter and returns its ESRI shape representation.

If the given geometry is null, then null is returned.

This function can also be called as a method.

►►—db2gse.ST_AsShape—(*—geometry—*)—◄◄

geometry

Return type

Example

Returns:

```
-----
1100 x'01000000000000000000000024400000000000003440'
```

This function can also be called as a method.

Syntax

►—db2gse.ST_AsText—(*—geometry—*)—►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes to be converted to the corresponding well-known text representation.

Return type

CLOB(2G)

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

After capturing and inserting the data into the SAMPLE_GEOMETRIES table, an analyst wants to verify that the values inserted are correct by looking at the well-known text representation of the geometries.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_geometries(id SMALLINT, spatial_type varchar(18),
                                geometry ST_GEOMETRY)
```

```
INSERT INTO sample_geometries(id, spatial_type, geometry)
VALUES
  (1, 'st_point', ST_Point(50, 50, 0)),
  (2, 'st_linestring', ST_LineString('linestring
    (200 100, 210 130, 220 140)', 0)),
  (3, 'st_polygon', ST_Polygon('polygon((110 120, 110 140,
    130 140, 130 120, 110 120))', 0))
```

The following SELECT statement lists the spatial type and the WKT representation of the geometries. The geometry is converted to text by the ST_AsText function. It is then cast to a varchar(120) because the default output of the ST_AsText function is CLOB(2G).

```
SELECT id, spatial_type, cast(geometry..ST_AsText
    AS varchar(150)) AS wkt
FROM   sample_geometries
```

Results:

ID	SPATIAL_TYPE	WKT
1	st_point	POINT (50.00000000 50.00000000)
2	st_linestring	LINESTRING (200.00000000 100.00000000, 210.00000000 130.00000000, 220.00000000 140.00000000)
3	st_polygon	POLYGON ((110.00000000 120.00000000, 130.00000000 120.00000000, 130.00000000 140.00000000, 110.00000000 140.00000000, 110.00000000 120.00000000))

ST_Boundary function

The ST_Boundary function takes a geometry as an input parameter and returns its boundary as a new geometry. The resulting geometry is represented in the spatial reference system of the given geometry.

If the given geometry is a point, multipoint, closed curve, or closed multicurve, or if it is empty, then the result is an empty geometry of type ST_Point. For curves or multicurves that are not closed, the start points and end points of the curves are returned as an ST_MultiPoint value, unless such a point is the start or end point of an even number of curves. For surfaces and multisurfaces, the curve defining the boundary of the given geometry is returned, either as an ST_Curve or an ST_MultiCurve value. If the given geometry is null, then null is returned.

If possible, the specific type of the returned geometry will be ST_Point, ST_LineString, or ST_Polygon. For example, the boundary of a polygon with no holes is a single linestring, represented as ST_LineString. The boundary of a polygon with one or more holes consists of multiple linestrings, represented as ST_MultiLineString.

This function can also be called as a method.

Syntax

►►—db2gse.ST_Boundary—(*—geometry—*)—►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes. The boundary of this geometry is returned.

Return type

db2gse.ST_Geometry

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example creates several geometries and determines the boundary of each geometry.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
  (1, ST_Polygon('polygon((40 120, 90 120, 90 150, 40 150, 40 120))', 0))

INSERT INTO sample_geoms VALUES
  (2, ST_Polygon('polygon((40 120, 90 120, 90 150, 40 150, 40 120),
    (70 130, 80 130, 80 140, 70 140, 70 130))', 0))

INSERT INTO sample_geoms VALUES
  (3, ST_Geometry('linestring(60 60, 65 60, 65 70, 70 70)', 0))

INSERT INTO sample_geoms VALUES
```



```

(4, ST_Geometry('multilinestring((60 60, 65 60, 65 70, 70 70),
                                   (80 80, 85 80, 85 90, 90 90),
                                   (50 50, 55 50, 55 60, 60 60))' ,0))

INSERT INTO sample_geoms VALUES
(5, ST_Geometry('point(30 30)' ,0))

SELECT id, CAST(ST_AsText(ST_Boundary(geometry)) as VARCHAR(320)) Boundary
FROM   sample_geoms

Results
ID      BOUNDARY
-----
1 LINESTRING ( 40.00000000 120.00000000, 90.00000000 120.00000000,
               90.00000000 150.00000000, 40.00000000 150.00000000, 40.00000000
               120.00000000)

2 MULTILINESTRING (( 40.00000000 120.00000000, 90.00000000 120.00000000,
                     90.00000000 150.00000000, 40.00000000 150.00000000, 40.00000000
                     120.00000000),( 70.00000000 130.00000000, 70.00000000 140.00000000,
                     80.00000000 140.00000000, 80.00000000 130.00000000, 70.00000000
                     130.00000000))

3 MULTIPOINT ( 60.00000000 60.00000000, 70.00000000 70.00000000)

4 MULTIPOINT ( 50.00000000 50.00000000, 70.00000000 70.00000000,
               80.00000000 80.00000000, 90.00000000 90.00000000)

5 POINT EMPTY

```

ST_Buffer function

The ST_Buffer function takes a geometry, a distance, and, optionally, a unit as input parameters and returns the geometry that surrounds the given geometry by the specified distance, measured in the given unit.

Each point on the boundary of the resulting geometry is the specified distance away from the given geometry. The resulting geometry is represented in the spatial reference system of the given geometry.

Any circular curve in the boundary of the resulting geometry is approximated by linear strings. For example, the buffer around a point, which would result in a circular region, is approximated by a polygon whose boundary is a linestring.

If the given geometry is null or is empty, null will be returned.

This function can also be called as a method.

Syntax

```

>> db2gse.ST_Buffer(—geometry—,—distance—,—unit—)<<

```

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry to create the buffer around.

distance

A DOUBLE PRECISION value that specifies the distance to be used for the buffer around *geometry*.

unit A VARCHAR(128) value that identifies the unit in which *distance* is measured. The supported units of measure are listed in the DB2GSE.ST_UNITS_OF_MEASURE catalog view.

If the *unit* parameter is omitted, the following rules are used to determine the unit of measure used for distance:

- If *geometry* is in a projected or geocentric coordinate system, the linear unit associated with this coordinate system is the default.
- If *geometry* is in a geographic coordinate system, the angular unit associated with this coordinate system is the default.

Restrictions on unit conversions: An error (SQLSTATE 38SU4) is returned if any of the following conditions occur:

- The geometry is in an unspecified coordinate system and the *unit* parameter is specified.
- The geometry is in a projected coordinate system and an angular unit is specified.
- The geometry is in a geographic coordinate system, but is not an ST_Point value, and a linear unit is specified.

Return type

db2gse.ST_Geometry

Examples

In the following examples, the results have been reformatted for readability. The spacing in your results will vary according to your display.

Example 1

The following code creates a spatial reference system, creates the SAMPLE_GEOMETRIES table, and populates it.

```
db2se create_srs se_bank -srsId 4000 -srsName new_york1983
        -xOffset 0 -yOffset 0 -xScale 1 -yScale 1
        -coordsysName NAD_1983_StatePlane_New_York_East_FIPS_3101_Feet

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE
    sample_geometries (id INTEGER, spatial_type varchar(18),
    geometry ST_GEOMETRY)

INSERT INTO sample_geometries(id, spatial_type, geometry)
VALUES
    (1, 'st_point', ST_Point(50, 50, 4000)),
    (2, 'st_linestring',
    ST_LineString('linestring(200 100, 210 130,
    220 140)', 4000)),
    (3, 'st_polygon',
    ST_Polygon('polygon((110 120, 110 140, 130 140,
    130 120, 110 120))',4000)),
    (4, 'st_multipolygon',
    ST_MultiPolygon('multipolygon(((30 30, 30 40,
    35 40, 35 30, 30 30),(35 30, 35 40, 45 40,
    45 30, 35 30)))', 4000))
```

Example 2

The following SELECT statement uses the ST_Buffer function to apply a buffer of 10.

```
SELECT id, spatial_type,  
       cast(geometry..ST_Buffer(10)..ST_AsText AS varchar(470)) AS buffer_10  
FROM   sample_geometries
```

Results:

ID	SPATIAL_TYPE	BUFFER_10
1	st_point	POLYGON ((60.00000000 50.00000000, 59.00000000 55.00000000, 54.00000000 59.00000000, 49.00000000 60.00000000, 44.00000000 58.00000000, 41.00000000 53.00000000, 40.00000000 48.00000000,42.00000000 43.00000000, 47.00000000 41.00000000, 52.00000000 40.00000000, 57.00000000 42.00000000, 60.00000000 50.00000000))
2	st_linestring	POLYGON ((230.00000000 140.00000000, 229.00000000 145.00000000, 224.00000000 149.00000000, 219.00000000 150.00000000, 213.00000000 147.00000000, 203.00000000 137.00000000, 201.00000000 133.00000000, 191.00000000 103.00000000, 191.00000000 99.00000000, 192.00000000 95.00000000, 196.00000000 91.00000000, 200.00000000 91.00000000,204.00000000 91.00000000, 209.00000000 97.00000000, 218.00000000 124.00000000, 227.00000000 133.00000000, 230.00000000 140.00000000))
3	st_polygon	POLYGON ((140.00000000 120.00000000, 140.00000000 140.00000000, 139.00000000 145.00000000, 130.00000000 150.00000000, 110.00000000 150.00000000, 105.00000000 149.00000000, 100.00000000 140.00000000,100.00000000 120.00000000, 101.00000000 115.00000000, 110.00000000 110.00000000,130.00000000 110.00000000, 135.00000000 111.00000000, 140.00000000 120.00000000))
4	st_multipolygon	POLYGON ((55.00000000 30.00000000, 55.00000000 40.00000000, 54.00000000 45.00000000, 45.00000000 50.00000000, 30.00000000 50.00000000, 25.00000000 49.00000000, 20.00000000 40.00000000, 20.00000000 30.00000000, 21.00000000 25.00000000, 30.00000000 20.00000000, 45.00000000 20.00000000, 50.00000000 21.00000000, 55.00000000 30.00000000))

Example 3

The following SELECT statement uses the ST_Buffer function to apply a negative buffer of 5.

```
SELECT id, spatial_type,  
       cast(ST_AsText(ST_Buffer(geometry, -5)) AS varchar(150))  
       AS buffer_negative_5  
FROM   sample_geometries  
WHERE  id = 3
```

Results:

ID	SPATIAL_TYPE	BUFFER_NEGATIVE_5
3	st_polygon	POLYGON ((115.00000000 125.00000000, 125.00000000 125.00000000, 125.00000000 135.00000000, 115.00000000 135.00000000, 115.00000000 125.00000000))

Example 4

The following SELECT statement shows the result of applying a buffer with the unit parameter specified.

Results:

ID	SPATIAL_TYPE	BUFFER_10_METER
3	st_polygon	POLYGON ((163.00000000 120.00000000, 163.00000000 140.00000000, 162.00000000 149.00000000, 159.00000000 157.00000000, 152.00000000 165.00000000, 143.00000000 170.00000000, 130.00000000 173.00000000, 110.00000000 173.00000000, 101.00000000 172.00000000, 92.00000000 167.00000000, 84.00000000 160.00000000, 79.00000000 151.00000000, 77.00000000 140.00000000, 77.00000000, 120.00000000, 78.00000000 111.00000000, 83.00000000 102.00000000, 90.00000000 94.00000000, 99.00000000 89.00000000, 110.00000000 87.00000000, 130.00000000 87.00000000, 139.00000000 88.00000000, 147.00000000 91.00000000, 155.00000000 98.00000000, 160.00000000 107.00000000, 163.00000000 120.00000000))

MBR Aggregate functions

Use the `ST_BuildMBRAggr` and `ST_GetAggrResult` functions combined to aggregate a set of geometries from a column into a single geometry. The combination constructs a rectangle that represents the minimum bounding rectangle that encloses all the geometries in the column. Z and M coordinates are discarded when the aggregate is computed.

The following expression is an example that uses the MAX function with the db2gse.ST_BuildMBRAggr spatial function to compute the MBR of the geometries in the columnName column and the db2gse.ST_GetAggrResult spatial function to return the resulting geometry that was computed for the MBR:

```
db2qse.ST Get AggrResult(MAX(db2qse.ST BuildMBRAggr(columnName)))
```

If all of the geometries to be combined are null, then null is returned. If all of the geometries are either null or empty, then an empty geometry is returned. If the minimum bounding rectangle of all the geometries to be combined results in a point, then this point is returned as an ST_Point value. If the minimum bounding rectangle of all the geometries to be combined results in a horizontal or vertical linestring, then this linestring is returned as an ST_LineString value. Otherwise, the minimum bounding rectangle is returned as an ST_Polygon value.

Syntax

```

▶▶db2gse.ST_GetAggrResult←(←MAX←(←
▶db2gse.ST_BuildMBRAggr←(←geometries←)←)←)←

```

Parameter

geometries

A selected column that has a type of ST_Geometry or one of its subtypes and represents all the geometries for which the minimum bounding rectangle is to be computed.

Return type

db2gse.ST_Geometry

Restrictions

You cannot construct the union aggregate of a spatial column in a full-select in any of the following situations:

- In a partitioned database environment
- If GROUP BY clause is used in the full-select.
- If you use a function other than the DB2 aggregate function MAX.

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example shows how to use the ST_BuildMBRAggr function to obtain the maximum bounding rectangle of all of the geometries within a column. In this example, several points are added to the GEOMETRY column in the SAMPLE_POINTS table. The SQL code then determines the maximum bounding rectangle of all of the points put together.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_points (id integer, geometry ST_Point)
```

```
INSERT INTO sample_points (id, geometry)
VALUES
```

```
(1, ST_Point(2, 3, 1)),
(2, ST_Point(4, 5, 1)),
(3, ST_Point(13, 15, 1)),
(4, ST_Point(12, 5, 1)),
(5, ST_Point(23, 2, 1)),
(6, ST_Point(11, 4, 1))
```

```
SELECT cast(ST_GetAggrResult(MAX(ST_BuildMBRAggr
(geometry)))..ST_AsText AS varchar(160))
AS ";Aggregate_of_Points";
FROM sample_points
```

Results:

Aggregate_of_Points

```
-----
POLYGON (( 2.00000000 2.00000000, 23.00000000 2.00000000,
23.00000000 15.00000000, 2.00000000 15.00000000, 2.00000000
2.00000000))
```

Union aggregate functions

A union aggregate is the combination of the ST_BuildUnionAggr and ST_GetAggrResult functions. Use this combination to aggregate a column of geometries in a table to single geometry by constructing the union.

If all of the geometries to be combined in the union are null , then null is returned. If each of the geometries to be combined in the union are either null or are empty, then an empty geometry of type ST_Point is returned.

The ST_BuildUnionAggr function can also be called as a method.

Syntax

```
► db2gse.ST_GetAggrResult(—————►  
► MAX(—db2gse.ST_BuildUnionAggr(—geometries—)——)——)—————►◄
```

Parameters

geometries

A column in a table that has a type of ST_Geometry or one of its subtypes and represents all the geometries that are to be combined into a union.

Return type

db2gse.ST_Geometry

Restrictions

You cannot construct the union aggregate of a spatial column in a table in any of the following situations:

- In partitioned database environments
- If a GROUP BY clause is used in the select
- If you use a function other than the DB2 aggregate function MAX

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how a union aggregate can be used to combine a set of points into multipoints. Several points are added to the SAMPLE_POINTS table. The ST_GetAggrResult and ST_BuildUnionAggr functions are used to construct the union of the points.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse  
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)  
  
INSERT INTO sample_points  
VALUES (1, ST_Point (2, 3, 1) )  
INSERT INTO sample_points  
VALUES (2, ST_Point (4, 5, 1) )  
INSERT INTO sample_points  
VALUES (3, ST_Point (13, 15, 1) )  
INSERT INTO sample_points  
VALUES (4, ST_Point (12, 5, 1) )  
INSERT INTO sample_points  
VALUES (5, ST_Point (23, 2, 1) )  
INSERT INTO sample_points  
VALUES (6, ST_Point (11, 4, 1) )  
  
SELECT CAST (ST_AsText(  
    ST_GetAggrResult( MAX( ST_BuildUnionAggregate (geometry) ) ))  
    AS VARCHAR(160)) POINT_AGGREGATE  
FROM sample_points
```

Results:

POINT_AGGREGATE

```
MULTIPOINT ( 2.00000000 3.00000000, 4.00000000 5.00000000,  
            11.00000000 4.00000000, 12.00000000 5.00000000,  
            13.00000000 15.00000000, 23.00000000 2.00000000)
```

ST_Centroid function

The ST_Centroid function takes a geometry as an input parameter and returns the geometric center, which is the center of the minimum bounding rectangle of the given geometry, as a point. The resulting point is represented in the spatial reference system of the given geometry.

If the given geometry is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_Centroid—(*—geometry—*)—————►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry to determine the geometric center.

Return type

db2gse.ST_Point

Example

This example creates two geometries and finds the centroid of them.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse  
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)
```

```
INSERT INTO sample_geoms VALUES  
  (1, ST_Polygon('polygon  
    ((40 120, 90 120, 90 150, 40 150, 40 120),  
    (50 130, 80 130, 80 140, 50 140, 50 130))',0))
```

```
INSERT INTO sample_geoms VALUES  
  (2, ST_MultiPoint('multipoint(10 10, 50 10, 10 30)',0))
```

```
SELECT id, CAST(ST_AsText(ST_Centroid(geometry))  
  as VARCHAR(40)) Centroid  
FROM sample_geoms
```

Results:

ID	CENTROID
1	POINT (65.00000000 135.00000000)
2	POINT (30.00000000 20.00000000)

ST_ChangePoint function

The ST_ChangePoint function takes a curve and two points as input parameters. It replaces all occurrences of the first point in the given curve with the second point and returns the resulting curve. The resulting geometry is represented in the spatial reference system of the given geometry.

If the two points are not represented in the same spatial reference system as the curve, they will be converted to the spatial reference system used for the curve.

If the given curve is empty, then an empty value is returned. If the given curve is null, or if any of the given points is null or empty, then null is returned.

This function can also be called as a method.

Syntax

►►db2gse.ST_ChangePoint(—*curve*—,—*old_point*—,—*new_point*—)————►►

Parameter

curve A value of type ST_Curve or one of its subtypes that represents the curve in which the points identified by *old_point* are changed to *new_point*.

old_point

A value of type ST_Point that identifies the points in the curve that are changed to *new_point*.

new_point

A value of type ST_Point that represents the new locations of the points in the curve identified by *old_point*.

Return type

db2gse.ST_Curve

Restrictions

The point to be changed in the curve must be one of the points used to define the curve.

If the curve has Z or M coordinates, then the given points also must have Z or M coordinates.

Examples

Example 1

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The following code creates and populates the SAMPLE_LINES table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines(id INTEGER, line ST_Linestring)
```

```
INSERT INTO sample_lines VALUES
```



```
(1, ST_LineString('linestring (10 10, 5 5, 0 0, 10 0, 5 5, 0 10)', 0) )
INSERT INTO sample_lines VALUES
(2, ST_LineString('linestring z (0 0 4, 5 5 5, 10 10 6, 5 5 7)', 0) )
```

Example 2

This example changes all occurrences of the point (5, 5) to the point (6, 6) in the linestring.

```
SELECT cast(ST_AsText(ST_ChangePoint(line, ST_Point(5, 5),
                                     ST_Point(6, 6))) as VARCHAR(160))
FROM   sample_lines
WHERE  id=1
```

Example 3

This example changes all occurrences of the point (5, 5, 5) to the point (6, 6, 6) in the linestring.

```
SELECT cast(ST_AsText(ST_ChangePoint(line, ST_Point(5.0, 5.0, 5.0),
                                     ST_Point(6.0, 6.0, 6.0) )) as VARCHAR(180))
FROM   sample_lines
WHERE  id=2
```

Results:

```
NEW
-----
LINESTRING Z ( 0.00000000 0.00000000 4.00000000, 6.00000000 6.00000000
6.00000000, 10.00000000 10.00000000 6.00000000, 5.00000000 5.00000000
7.00000000)
```

ST_Contains function

Use the ST_Contains function to determine whether one geometry is completely contained by another geometry.

Syntax

```
►►—db2gse.ST_Contains—(—geometry1—,—geometry2—)——————►◄
```

Parameter

geometry1

A value of type ST_Geometry or one of its subtypes that represents the geometry that is to be tested to completely contain *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the geometry that is to be tested to be completely within *geometry1*.

Return type

INTEGER

Usage

ST_Contains takes two geometries as input parameter and returns 1 if the first geometry completely contains the second or the second geometry is completely contained by the first geometry. Otherwise, it returns 0 (zero) to indicate that the first geometry does not completely contain the second.

The ST_Contains function returns the exact opposite result of the ST_Within function.

If any of the given geometries is null or is empty, then null is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry and uses the same underlying datum, it will be converted to the other spatial reference system.

The pattern matrix of the ST_Contains function states that the interiors of both geometries must intersect and that the interior or boundary of the secondary (geometry *b*) must not intersect the exterior of the primary (geometry *a*). The asterisk (*) indicates that it does not matter if an intersection exists between these parts of the geometries.

Table 27. Matrix for ST_Contains

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

Examples

Figure 19 on page 259 shows examples of ST_Contains:

- A multipoint geometry contains a point or multipoint geometries when all of the points are within the first geometry.
- A polygon geometry contains a multipoint geometry when all of the points are either on the boundary of the polygon or in the interior of the polygon.
- A linestring geometry contains a point, multipoint, or linestring geometries when all of the points are within the first geometry.
- A polygon geometry contains a point, linestring or polygon geometries when the second geometry is in the interior of the polygon.

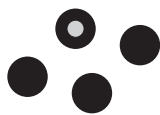
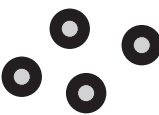
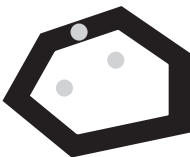






		
multipoint / point	multipoint / multipoint	polygon / multipoint
		
linestring / point	linestring / multipoint	linestring / linestring
		
polygon / point	polygon / linestring	polygon / polygon

Figure 19. *ST_Contains*. The dark geometries represent geometry a and the gray geometries represent geometry b. In all cases, geometry a contains geometry b completely.

Example 1

The following code creates and populates these tables.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sample_points(id SMALLINT, geometry ST_POINT)

CREATE TABLE sample_lines(id SMALLINT, geometry ST_LINESTRING)

CREATE TABLE sample_polygons(id SMALLINT, geometry ST_POLYGON)

INSERT INTO sample_points (id, geometry)
VALUES
  (1, ST_Point(10, 20, 1)),
  (2, ST_Point('point(41 41)', 1))

INSERT INTO sample_lines (id, geometry)
VALUES
  (10, ST_LineString('linestring (1 10, 3 12, 10 10)', 1) ),
  (20, ST_LineString('linestring (50 10, 50 12, 45 10)', 1) )
INSERT INTO sample_polygons(id, geometry)
VALUES
  (100, ST_Polygon('polygon((0 0, 0 40, 40 40, 40 0, 0 0))', 1) )
```

Example 2

The following code fragment uses the *ST_Contains* function to determine which points are contained by a particular polygon.

```

SELECT poly.id AS polygon_id,
       CASE ST_Contains(poly.geometry, pts.geometry)
         WHEN 0 THEN 'does not contain'
         WHEN 1 THEN 'does contain'
       END AS contains,
       pts.id AS point_id
FROM   sample_points pts, sample_polygons poly

```

Results:

POLYGON_ID	CONTAINS	POINT_ID
100	does contain	1
100	does not contain	2

Example 3

The following code fragment uses the ST_Contains function to determine which lines are contained by a particular polygon.

```

SELECT poly.id AS polygon_id,
       CASE ST_Contains(poly.geometry, line.geometry)
         WHEN 0 THEN 'does not contain'
         WHEN 1 THEN 'does contain'
       END AS contains,
       line.id AS line_id
FROM   sample_lines line, sample_polygons poly

```

Results:

POLYGON_ID	CONTAINS	LINE_ID
100	does contain	10
100	does not contain	20

ST_ConvexHull function

The ST_ConvexHull function takes a geometry as an input parameter and returns the convex hull of it.

The resulting geometry is represented in the spatial reference system of the given geometry.

If possible, the specific type of the returned geometry will be ST_Point, ST_LineString, or ST_Polygon. For example, the boundary of a polygon with no holes is a single linestring, represented as ST_LineString. The boundary of a polygon with one or more holes consists of multiple linestrings, represented as ST_MultiLineString.

If the given geometry is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

```

►► db2gse.ST_ConvexHull (—geometry—) ◀◀

```

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry to compute the convex hull.

Return type

db2gse.ST_Geometry

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The following code creates and populates the SAMPLE_GEOMETRIES table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sample_geometries(id INTEGER, spatial_type varchar(18),
    geometry ST_GEOMETRY)

INSERT INTO sample_geometries(id, spatial_type, geometry)
VALUES
    (1, 'ST_LineString', ST_LineString
      ('linestring(20 20, 30 30, 20 40, 30 50)', 0)),
    (2, 'ST_Polygon', ST_Polygon('polygon
      ((110 120, 110 140, 120 130, 110 120))', 0) ),
    (3, 'ST_Polygon', ST_Polygon('polygon((30 30, 25 35, 15 50,
      35 80, 40 85, 80 90,70 75, 65 70, 55 50, 75 40, 60 30,
      30 30))', 0) ),
    (4, 'ST_MultiPoint', ST_MultiPoint('multipoint(20 20, 30 30,
      20 40, 30 50)', 1))
```

The following SELECT statement calculates the convex hull for all the geometries constructed previously and displays the result.

```
SELECT id, spatial_type, cast(geometry..ST_ConvexHull..ST_AsText
    AS varchar(300)) AS convexhull
FROM   sample_geometries
```

Results:

ID	SPATIAL_TYPE	CONVEXHULL
1	ST_LineString	POLYGON ((20.00000000 40.00000000, 20.00000000 20.00000000, 30.00000000 30.00000000, 30.00000000 50.00000000, 20.00000000 40.00000000))
2	ST_Polygon	POLYGON ((110.00000000 140.00000000, 110.00000000 120.00000000, 120.00000000 130.00000000, 110.00000000 140.00000000))
3	ST_Polygon	POLYGON ((15.00000000 50.00000000, 25.00000000 35.00000000, 30.00000000 30.00000000, 60.00000000 30.00000000, 75.00000000 40.00000000, 80.00000000 90.00000000, 40.00000000 85.00000000, 35.00000000 80.00000000, 15.00000000 50.00000000))
4	ST_MultiPoint	POLYGON ((20.00000000 40.00000000, 20.00000000 20.00000000, 30.00000000 30.00000000, 30.00000000 50.00000000, 20.00000000 40.00000000))

ST_CoordDim function

The ST_CoordDim function takes a geometry as an input parameter and returns the dimensionality of its coordinates.

If the given geometry does not have Z and M coordinates, the dimensionality is 2. If it has Z coordinates and no M coordinates, or if it has M coordinates and no Z coordinates, the dimensionality is 3. If it has Z and M coordinates, the dimensionality is 4. If the geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►►db2gse.ST_CoordDim(—*geometry*—)—————►►

Parameter

geometry
A value of type ST_Geometry or one of its subtypes that represents the geometry to retrieve the dimensionality from.

Return type

INTEGER

Example

This example creates several geometries and then determines the dimensionality of their coordinates.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id CHARACTER(15), geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
    ('Empty Point', ST_Geometry('point EMPTY',0))

INSERT INTO sample_geoms VALUES
    ('Linestring', ST_Geometry('linestring (10 10, 15 20)',0))

INSERT INTO sample_geoms VALUES
    ('Polygon', ST_Geometry('polygon((40 120, 90 120, 90 150,
    40 150, 40 120))' ,0))

INSERT INTO sample_geoms VALUES
    ('Multipoint M', ST_Geometry('multipoint m (10 10 5, 50 10
    6, 10 30 8)' ,0))

INSERT INTO sample_geoms VALUES
    ('Multipoint Z', ST_Geometry('multipoint z (47 34 295,
    23 45 678)' ,0))

INSERT INTO sample_geoms VALUES
    ('Point ZM', ST_Geometry('point zm (10 10 16 30)' ,0))

SELECT id, ST_CoordDim(geometry) COORDDIM
FROM sample_geoms
```

Results:

ID	COORDDIM
Empty Point	2
Linestring	2

Polygon	2
Multipoint M	3
Multipoint Z	3
Point ZM	4

ST_Crosses function

The ST_Crosses function takes two geometries as input parameters and returns 1 if the first geometry crosses the second. Otherwise, 0 (zero) is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry, it will be converted to the other spatial reference system.

If the first geometry is a polygon or a multipolygon, or if the second geometry is a point or multipoint, or if any of the geometries is null value or is empty, then null is returned. If the intersection of the two geometries results in a geometry that has a dimension that is one less than the maximum dimension of the two given geometries, and if the resulting geometry is not equal any of the two given geometries, then 1 is returned. Otherwise, the result is 0 (zero).

Syntax

►►—db2gse.ST_Crosses—(—*geometry1*—,—*geometry2*—)—►►

Parameter

geometry1

A value of type ST_Geometry or one of its subtypes that represents the geometry that is to be tested for crossing *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the geometry that is to be tested to determine if it is crossed by *geometry1*.

Return Type

INTEGER

Example

This code determines if the constructed geometries cross each other.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
  (1, ST_Geometry('polygon((30 30, 30 50, 50 50, 50 30, 30 30))' ,0))

INSERT INTO sample_geoms VALUES
  (2, ST_Geometry('linestring(40 50, 50 40)' ,0))

INSERT INTO sample_geoms VALUES
  (3, ST_Geometry('linestring(20 20, 60 60)' ,0))

SELECT a.id, b.id, ST_Crosses(a.geometry, b.geometry) Crosses
FROM   sample_geoms a, sample_geoms b
```

Results:

ID	ID	CROSSES
	1	1
	2	1
	3	1
	1	2
	2	2
	3	2
	1	3
	2	3
	3	3

ST_Difference function

The ST_Difference function takes two geometries as input parameters and returns the part of the first geometry that does not intersect with the second geometry.

Both geometries must be of the same dimension. If either geometry is null, null is returned. If the first geometry is empty, an empty geometry of type ST_Point is returned. If the second geometry is empty, then the first geometry is returned unchanged.

If the second geometry is not represented in the same spatial reference system as the first geometry and uses the same underlying datum, it will be converted to the other spatial reference system.

This function can also be called as a method.

Syntax

►►—db2gse.ST_Difference—(—*geometry1*—,—*geometry2*—)—►►

Parameter

geometry1

A value of type ST_Geometry that represents the first geometry to use to compute the difference to *geometry2*.

geometry2

A value of type ST_Geometry that represents the second geometry that is used to compute the difference to *geometry1*.

Return type

db2gse.ST_Geometry

The dimension of the returned geometry is the same as that of the input geometries.

Examples

In the following example, the results have been reformatted for readability. The spacing in your results will vary according to your display.

The following code creates and populates the SAMPLE_GEOMETRIES table.


```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
(1, ST_Geometry('polygon((10 10, 10 20, 20 20, 20 10, 10 10))' ,0))

INSERT INTO sample_geoms VALUES
(2, ST_Geometry('polygon((30 30, 30 50, 50 50, 50 30, 30 30))' ,0))

INSERT INTO sample_geoms VALUES
(3, ST_Geometry('polygon((40 40, 40 60, 60 60, 60 40, 40 40))' ,0))

INSERT INTO sample_geoms VALUES
(4, ST_Geometry('linestring(70 70, 80 80)' ,0))

INSERT INTO sample_geoms VALUES
(5, ST_Geometry('linestring(75 75, 90 90)' ,0))

```

Example 1

This example finds the difference between two disjoint polygons.

```

SELECT a.id, b.id, CAST(ST_AsText(ST_Difference(a.geometry, b.geometry))
as VARCHAR(200)) Difference
FROM sample_geoms a, sample_geoms b
WHERE a.id = 1 and b.id = 2

```

Results:

ID	ID	DIFFERENCE
1	2	POLYGON ((10.00000000 10.00000000, 20.00000000 10.00000000, 20.00000000 20.00000000, 10.00000000 20.00000000, 10.00000000 10.00000000))

Example 2

This example finds the difference between two intersecting polygons.

```

SELECT a.id, b.id, CAST(ST_AsText(ST_Difference(a.geometry, b.geometry))
as VARCHAR(200)) Difference
FROM sample_geoms a, sample_geoms b
WHERE a.id = 2 and b.id = 3

```

Results:

ID	ID	DIFFERENCE
2	3	POLYGON ((30.00000000 30.00000000, 50.00000000 30.00000000, 50.00000000 40.00000000, 40.00000000 40.00000000, 40.00000000 50.00000000, 30.00000000 50.00000000, 30.00000000 30.00000000))

Example 3

This example finds the difference between two overlapping linestrings.

```

SELECT a.id, b.id, CAST(ST_AsText(ST_Difference(a.geometry, b.geometry))
as VARCHAR(100)) Difference
FROM sample_geoms a, sample_geoms b
WHERE a.id = 4 and b.id = 5

```

Results:

ID	ID	DIFFERENCE
4	5	LINestring (70.00000000 70.00000000, 75.00000000 75.00000000)

ST_Dimension function

The ST_Dimension function takes a geometry as an input parameter and returns its dimension.

If the given geometry is empty, then -1 is returned. For points and multipoints, the dimension is 0 (zero); for curves and multicurves, the dimension is 1; and for polygons and multipolygons, the dimension is 2. If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_Dimension—(*—geometry—*)—————►◄

Parameter

geometry

A value of type ST_Geometry that represents the geometry for which the dimension is returned.

Return type

INTEGER

Example

This example creates several different geometries and finds their dimensions.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id char(15), geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
  ('Empty Point', ST_Geometry('point EMPTY',0))

INSERT INTO sample_geoms VALUES
  ('Point ZM', ST_Geometry('point zm (10 10 16 30)',0))

INSERT INTO sample_geoms VALUES
  ('MultiPoint M', ST_Geometry('multipoint m (10 10 5,
    50 10 6, 10 30 8)',0))

INSERT INTO sample_geoms VALUES
  ('LineString', ST_Geometry('linestring (10 10, 15 20)',0))

INSERT INTO sample_geoms VALUES
  ('Polygon', ST_Geometry('polygon((40 120, 90 120, 90 150,
    40 150, 40 120))',0))

SELECT id, ST_Dimension(geometry) Dimension
FROM sample_geoms
```

Results:

ID	DIMENSION
Empty Point	-1
Point ZM	0

MultiPoint M	0
LineString	1
Polygon	2

ST_Disjoint function

The ST_Disjoint function takes two geometries as input parameters and returns 1 if the given geometries do not intersect. If the geometries do intersect, then 0 (zero) is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry, it will be converted to the other spatial reference system.

If any of the two geometries is null or is empty, then null value is returned.

This function can also be called as a method.

Syntax

►► db2gse.ST_Disjoint(—*geometry1*—,—*geometry2*—)◄◄

Parameter

geometry1

A value of type ST_Geometry that represents the geometry that is tested to be disjoint with *geometry2*.

geometry2

A value of type ST_Geometry that represents the geometry that that is tested to be disjoint with *geometry1*.

Return type

INTEGER

Examples

Example 1

This code creates several geometries in the SAMPLE_GEOMETRIES table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
  (1, ST_Geometry('polygon((20 30, 30 30, 30 40, 20 40, 20 30))',0))

INSERT INTO sample_geoms VALUES
  (2, ST_Geometry('polygon((30 30, 30 50, 50 50, 50 30, 30 30))',0))

INSERT INTO sample_geoms VALUES
  (3, ST_Geometry('polygon((40 40, 40 60, 60 60, 60 40, 40 40))',0))

INSERT INTO sample_geoms VALUES
  (4, ST_Geometry('linestring(60 60, 70 70)',0))

INSERT INTO sample_geoms VALUES
  (5, ST_Geometry('linestring(30 30, 40 40)',0))
```

Example 2

This example determines if the first polygon is disjoint from any of the geometries.

```
SELECT a.id, b.id, ST_Disjoint(a.geometry, b.geometry) DisJoint
FROM sample_geoms a, sample_geoms b
WHERE a.id = 1
```

Results:

ID	ID	DISJOINT
1	1	0
1	2	0
1	3	1
1	4	1
1	5	0

Example 3

This example determines if the third polygon is disjoint from any of the geometries.

```
SELECT a.id, b.id, ST_Disjoint(a.geometry, b.geometry) DisJoint
FROM sample_geoms a, sample_geoms b
WHERE a.id = 3
```

Results:

ID	ID	DISJOINT
3	1	1
3	2	0
3	3	0
3	4	0
3	5	0

Example 4

This example determines if the second linestring is disjoint from any of the geometries.

```
SELECT a.id, b.id, ST_Disjoint(a.geometry, b.geometry) DisJoint
FROM sample_geoms a, sample_geoms b
WHERE a.id = 5
```

Results:

ID	ID	DISJOINT
5	1	0
5	2	0
5	3	0
5	4	1
5	5	0

ST_Distance function

The ST_Distance function takes two geometries and, optionally, a unit as input parameters and returns the shortest distance between any point in the first geometry to any point in the second geometry, measured in the default or given units.

If any of the two geometries is null or is empty, null is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry and uses the same underlying datum, it will be converted to the other spatial reference system.

You can also call this function as a method when you provide a unit of measure.

Syntax

►► db2gse.ST_Distance(*—geometry1—*, *—geometry2—* , *—unit—*) ►►

Parameter

geometry1

A value of type ST_Geometry that represents the geometry that is used to compute the distance to *geometry2*.

geometry2

A value of type ST_Geometry that represents the geometry that is used to compute the distance to *geometry1*.

unit VARCHAR(128) value that identifies the unit in which the result is measured. The supported units of measure are listed in the DB2GSE.ST_UNITS_OF_MEASURE catalog view.

If the *unit* parameter is omitted, the following rules are used to determine the unit of measure used for the result:

- If *geometry1* is in a projected or geocentric coordinate system, the linear unit associated with this coordinate system is the default.
- If *geometry1* is in a geographic coordinate system, the angular unit associated with this coordinate system is the default.

Restrictions on unit conversions: An error (SQLSTATE 38SU4) is returned if any of the following conditions occur:

- The geometry is in an unspecified coordinate system and the *unit* parameter is specified.
- The geometry is in a projected coordinate system and an angular unit is specified.

Return type

DOUBLE

Examples

Example 1

The following SQL statements create and populate the SAMPLE_GEOMETRIES1 and SAMPLE_GEOMETRIES2 tables.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sample_geometries1(id SMALLINT, spatial_type varchar(13),
  geometry ST_GEOMETRY)

CREATE TABLE sample_geometries2(id SMALLINT, spatial_type varchar(13),
  geometry ST_GEOMETRY)

INSERT INTO sample_geometries1(id, spatial_type, geometry)
VALUES
  ( 1, 'ST_Point', ST_Point('point(100 100)', 1)),
  (10, 'ST_LineString', ST_LineString('linestring(125 125, 125 175)', 1)),
  (20, 'ST_Polygon', ST_Polygon('polygon
    ((50 50, 50 150, 150 150, 150 50, 50 50))', 1))

INSERT INTO sample_geometries2(id, spatial_type, geometry)
VALUES
  (101, 'ST_Point', ST_Point('point(200 200)', 1)),
```

```
(102, 'ST_Point', ST_Point('point(200 300)', 1)),
(103, 'ST_Point', ST_Point('point(200 0)', 1)),
(110, 'ST_LineString', ST_LineString('linestring(200 100, 200 200)', 1)),
(120, 'ST_Polygon', ST_Polygon('polygon
((200 0, 200 200, 300 200, 300 0, 200 0))', 1))
```

Example 2

The following SELECT statement calculates the distance between the various geometries in the SAMPLE_GEOMTRIES1 and SAMPLE_GEOMTRIES2 tables.

```
SELECT    sg1.id AS sg1_id, sg1.spatial_type AS sg1_type,
          sg2.id AS sg2_id, sg2.spatial_type AS sg2_type,
          cast(ST_Distance(sg1.geometry, sg2.geometry)
              AS Decimal(8, 4)) AS distance
FROM      sample_geometries1 sg1, sample_geometries2 sg2
ORDER BY  sg1.id
```

Results:

SG1_ID	SG1_TYPE	SG2_ID	SG2_TYPE	DISTANCE
1	ST_Point	101	ST_Point	141.4213
1	ST_Point	102	ST_Point	223.6067
1	ST_Point	103	ST_Point	141.4213
1	ST_Point	110	ST_LineString	100.0000
1	ST_Point	120	ST_Polygon	100.0000
10	ST_LineString	101	ST_Point	79.0569
10	ST_LineString	102	ST_Point	145.7737
10	ST_LineString	103	ST_Point	145.7737
10	ST_LineString	110	ST_LineString	75.0000
10	ST_LineString	120	ST_Polygon	75.0000
20	ST_Polygon	101	ST_Point	70.7106
20	ST_Polygon	102	ST_Point	158.1138
20	ST_Polygon	103	ST_Point	70.7106
20	ST_Polygon	110	ST_LineString	50.0000
20	ST_Polygon	120	ST_Polygon	50.0000

Example 3

The following SELECT statement illustrates how to find all the geometries that are within a distance of 100 of each other.

```
SELECT    sg1.id AS sg1_id, sg1.spatial_type AS sg1_type,
          sg2.id AS sg2_id, sg2.spatial_type AS sg2_type,
          cast(ST_Distance(sg1.geometry, sg2.geometry)
              AS Decimal(8, 4)) AS distance
FROM      sample_geometries1 sg1, sample_geometries2 sg2
WHERE     ST_Distance(sg1.geometry, sg2.geometry) <= 100
```

Results:

SG1_ID	SG1_TYPE	SG2_ID	SG2_TYPE	DISTANCE
1	ST_Point	110	ST_LineString	100.0000
1	ST_Point	120	ST_Polygon	100.0000
10	ST_LineString	101	ST_Point	79.0569
10	ST_LineString	110	ST_LineString	75.0000
10	ST_LineString	120	ST_Polygon	75.0000
20	ST_Polygon	101	ST_Point	70.7106
20	ST_Polygon	103	ST_Point	70.7106
20	ST_Polygon	110	ST_LineString	50.0000
20	ST_Polygon	120	ST_Polygon	50.0000

Example 4

The following SELECT statement calculates the distance in kilometers between the various geometries.

```

SAMPLE_GEOMETRIES1 and SAMPLE_GEOMETRIES2 tables.
SELECT sg1.id AS sg1_id, sg1.spatial_type AS sg1_type,
       sg2.id AS sg2_id, sg2.spatial_type AS sg2_type,
       cast(ST_Distance(sg1.geometry, sg2.geometry, 'KILOMETER')
           AS DECIMAL(10, 4)) AS distance
FROM   sample_geometries1 sg1, sample_geometries2 sg2
ORDER BY sg1.id

```

Results:

SG1_ID	SG1_TYPE	SG2_ID	SG2_TYPE	DISTANCE
1	ST_Point	101	ST_Point	12373.2168
1	ST_Point	102	ST_Point	16311.3816
1	ST_Point	103	ST_Point	9809.4713
1	ST_Point	110	ST_LineString	1707.4463
1	ST_Point	120	ST_Polygon	12373.2168
10	ST_LineString	101	ST_Point	8648.2333
10	ST_LineString	102	ST_Point	11317.3934
10	ST_LineString	103	ST_Point	10959.7313
10	ST_LineString	110	ST_LineString	3753.5862
10	ST_LineString	120	ST_Polygon	10891.1254
20	ST_Polygon	101	ST_Point	7700.5333
20	ST_Polygon	102	ST_Point	15039.8109
20	ST_Polygon	103	ST_Point	7284.8552
20	ST_Polygon	110	ST_LineString	6001.8407
20	ST_Polygon	120	ST_Polygon	14515.8872

ST_DistanceToPoint function

The ST_DistanceToPoint function takes a curve or multi-curve geometry and a point geometry as input parameters and returns the distance along the curve geometry to the specified point.

This function can also be called as a method.

Syntax

```

>> db2gse.ST_DistanceToPoint(—curve-geometry—,—point-geometry—)————><

```

Parameter

curve-geometry

A value of type ST_Curve or ST_MultiCurve or one of its subtypes that represents the geometry to process.

point-geometry

A value of type ST_Point that is a point along the specified curve.

Return type

DOUBLE

Example

The following SQL statement creates the SAMPLE_GEOMETRIES table with two columns. The ID column uniquely identifies each row. The GEOMETRY ST_LineString column stores sample geometries.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries(id INTEGER, geometry ST_LINESTRING)

```

The following SQL statement inserts two rows into the SAMPLE_GEOMETRIES table.

```
INSERT INTO sample_geometries(id, geometry)
VALUES
  (1,ST_LineString('LINESTRING ZM(0 0 0 0, 10 100 1000 10000)',1)),
  (2,ST_LineString('LINESTRING ZM(10 100 1000 10000, 0 0 0 0)',1))
```

The following SELECT statement and the corresponding result set shows how to use the ST_DistanceToPoint function to find the distance to the point at the location (1.5, 15.0).

```
SELECT ID,
  DECIMAL(ST_DistanceToPoint(geometry,ST_Point(1.5,15.0,1)),10,5) AS DISTANCE
FROM sample_geometries
```

ID	DISTANCE
1	15.07481
2	85.42394

2 record(s) selected.

ST_Endpoint function

The ST_Endpoint function takes a curve as an input parameter and returns the point that is the last point of the curve. The resulting point is represented in the spatial reference system of the given curve.

If the given curve is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►► db2gse.ST_EndPoint(—*curve*—) ◀◀

Parameter

curve A value of type ST_Curve that represents the geometry from which the last point is returned.

Return type

db2gse.ST_Point

Example

The SELECT statement finds the endpoint of each of the geometries in the SAMPLE_LINES table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines(id INTEGER, line ST_Linestring)
```

```
INSERT INTO sample_lines VALUES
  (1, ST_LineString('linestring (10 10, 5 5, 0 0, 10 0, 5 5, 0 10)', 0) )
```

```
INSERT INTO sample_lines VALUES
  (2, ST_LineString('linestring z (0 0 4, 5 5 5, 10 10 6, 5 5 7)', 0) )
```



```
SELECT id, CAST(ST_AsText(ST_EndPoint(line)) as VARCHAR(50)) Endpoint
FROM   sample_lines
```

Results:

ID	ENDPOINT
1	POINT (0.00000000 10.00000000)
2	POINT Z (5.00000000 5.00000000 7.00000000)

ST_Envelope function

The ST_Envelope function takes a geometry as an input parameter and returns an envelope around the geometry. The envelope is a rectangle that is represented as a polygon.

If the given geometry is a point, a horizontal linestring, or a vertical linestring, then a rectangle, which is slightly larger than the given geometry, is returned. Otherwise, the minimum bounding rectangle of the geometry is returned as the envelope. If the given geometry is null or is empty, then null is returned. To return the exact minimum bounding rectangle for all geometries, use the function ST_MBR.

This function can also be called as a method.

Syntax

```
db2gse.ST_Envelope(geometry)
```

Parameter

geometry

A value of type ST_Geometry that represents the geometry to return the envelope for.

Return type

db2gse.ST_Polygon

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example creates several geometries and then determines their envelopes. For the non-empty point and the linestring (which is horizontal), the envelope is a rectangle that is slightly larger than the geometry.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)
```

```
INSERT INTO sample_geoms VALUES
(1, ST_Geometry('point EMPTY',0))
```

```
INSERT INTO sample_geoms VALUES
(2, ST_Geometry('point zm (10 10 16 30)',0))
```

```

INSERT INTO sample_geoms VALUES
(3, ST_Geometry('multipoint m (10 10 5, 50 10 6, 10 30 8)' ,0))

INSERT INTO sample_geoms VALUES
(4, ST_Geometry('linestring (10 10, 20 10)',0))

INSERT INTO sample_geoms VALUES
(5, ST_Geometry('polygon((40 120, 90 120, 90 150, 40 150, 40 120))',0))

SELECT id, CAST(ST_AsText(ST_Envelope(geometry)) as VARCHAR(160)) Envelope
FROM sample_geoms

```

Results:

ID	ENVELOPE
1	-
2	POLYGON ((9.00000000 9.00000000, 11.00000000 9.00000000, 11.00000000 11.00000000, 9.00000000 11.00000000, 9.00000000 9.00000000))
3	POLYGON ((10.00000000 10.00000000, 50.00000000 10.00000000, 50.00000000 30.00000000, 10.00000000 30.00000000, 10.00000000 10.00000000))
4	POLYGON ((10.00000000 9.00000000, 20.00000000 9.00000000, 20.00000000 11.00000000, 10.00000000 11.00000000, 10.00000000 9.00000000))
5	POLYGON ((40.00000000 120.00000000, 90.00000000 120.00000000, 90.00000000 150.00000000, 40.00000000 150.00000000, 40.00000000 120.00000000))

ST_EnvIntersects function

The ST_EnvIntersects function takes two geometries as input parameters and returns 1 if the envelopes of two geometries intersect. Otherwise, 0 (zero) is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry, it will be converted to the other spatial reference system.

If any of the given geometries is null or is empty, then null value is returned.

Syntax

►►—db2gse.ST_EnvIntersects—(—*geometry1*—,—*geometry2*—)—►►

Parameter

geometry1

A value of type ST_Geometry or one of its subtypes that represents the geometry whose envelope is to be tested for intersection with the envelope of *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the geometry whose envelope is to be tested for intersection with the envelope of *geometry1*.

Return type

INTEGER

Example

This example creates two parallel linestrings and checks them for intersection. The linestrings themselves do not intersect, but the envelopes for them do.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
  (1, ST_Geometry('linestring (10 10, 50 50)',0))

INSERT INTO sample_geoms VALUES
  (2, ST_Geometry('linestring (10 20, 50 60)',0))

SELECT a.id, b.id, ST_Intersects(a.geometry, b.geometry) Intersects,
       ST_EnvIntersects(a.geometry, b.geometry) Envelope_Intersects
FROM   sample_geoms a , sample_geoms b
WHERE  a.id = 1 and b.id=2
```

Results:

ID	ID	INTERSECTS	ENVELOPE_INTERSECTS
1	2	0	1

ST_EqualCoordsys function

The ST_EqualCoordsys function takes two coordinate system definitions as input parameters and returns the integer value 1 (one) if the given definitions are identical. Otherwise, the integer value 0 (zero) is returned.

The coordinate system definitions are compared regardless of differences in spaces, parenthesis, uppercase and lowercase characters, and the representation of floating point numbers.

If any of the given coordinate system definitions is null, null is returned.

Syntax

►►—db2gse.ST_EqualCoordsys—(—*coordinate_system1*—,—*coordinate_system2*—)—►►

Parameter

coordinate_system1

A value of type VARCHAR(2048) that defines the first coordinate system to be compared with *coordinate_system2*.

coordinate_system2

A value of type VARCHAR(2048) that defines the second coordinate system to be compared with *coordinate_system1*.

Return type

INTEGER

Example

This example compares two Australian coordinate systems to see if they are the same.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
VALUES ST_EqualCoordSys(
  (SELECT definition
   FROM db2gse.ST_COORDINATE_SYSTEMS
   WHERE coordsys_name='GCS_AUSTRALIAN') ,
  (SELECT definition
   FROM db2gse.ST_COORDINATE_SYSTEMS
   WHERE coordsys_name='GCS_AUSTRALIAN_1984')
)
```

Results:

```
1
-----
0
```

ST_Equals function

The ST_Equals function takes two geometries as input parameters and returns 1 if the geometries are equal. Otherwise 0 (zero) is returned. The order of the points used to define the geometry is not relevant for the test for equality.

If the second geometry is not represented in the same spatial reference system as the first geometry, it will be converted to the other spatial reference system.

If any of the two given geometries is null, then null is returned.

Syntax

```
►►—db2gse.ST_Equals—(—geometry1—,—geometry2—)——————►►
```

Parameter

geometry1

A value of type ST_Geometry that represents the geometry that is to be compared with *geometry2*.

geometry2

A value of type ST_Geometry that represents the geometry that is to be compared with *geometry1*.

Return type

INTEGER

Examples

Example 1

This example creates two polygons that have their coordinates in a different order. ST_Equals is used to show that these polygons are considered equal.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
    (1, ST_Geometry('polygon((50 30, 30 30, 30 50, 50 50, 50 30))' ,0))

INSERT INTO sample_geoms VALUES
    (2, ST_Geometry('polygon((50 30, 50 50, 30 50, 30 30, 50 30))' ,0))

SELECT a.id, b.id, ST_Equals(a.geometry, b.geometry) Equals
FROM sample_geoms a, sample_geoms b
WHERE a.id = 1 and b.id = 2

```

Results:

ID	ID	EQUALS
-----	-----	-----
1	2	1

Example 2

In this example, two geometries are created with the same X and Y coordinates, but different M coordinates (measures). When the geometries are compared with the ST_Equal function, a 0 (zero) is returned to indicate that these geometries are not equal.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
    (3, ST_Geometry('multipoint m(80 80 6, 90 90 7)' ,0))

INSERT INTO sample_geoms VALUES
    (4, ST_Geometry('multipoint m(80 80 6, 90 90 4)' ,0))

SELECT a.id, b.id, ST_Equals(a.geometry, b.geometry) Equals
FROM sample_geoms a, sample_geoms b
WHERE a.id = 3 and b.id = 4

```

Results:

ID	ID	EQUALS
-----	-----	-----
3	4	0

Example 3

In this example, two geometries are created with a different set of coordinates, but both represent the same geometry. ST_Equal compares the geometries and indicates that both geometries are indeed equal.

```

SET current function path = current function path, db2gse
CREATE TABLE sample_geoms ( id INTEGER, geometry ST_Geometry )

INSERT INTO sample_geoms VALUES
    (5, ST_LineString('linestring ( 10 10, 40 40 )', 0)),
    (6, ST_LineString('linestring ( 10 10, 20 20, 40 40)', 0))

SELECT a.id, b.id, ST_Equals(a.geometry, b.geometry) Equals
FROM sample_geoms a, sample_geoms b
WHERE a.id = 5 AND b.id = 6

```

Results:

ID	ID	EQUALS
-----	-----	-----
5	6	1

ST_EqualSRS function

The ST_EqualSRS function takes two spatial reference system identifiers as input parameters and returns 1 if the given spatial reference systems are identical. Otherwise, 0 (zero) is returned. The offsets, scale factors, and the coordinate systems are compared.

If any of the given spatial reference system identifiers is null, null is returned.

Syntax

►►—db2gse.ST_EqualSRS—(—*srs_id1*—,—*srs_id2*—)——————►◄

Parameter

srs_id1

A value of type INTEGER that identifies the first spatial reference system to be compared with the spatial reference system identified by *srs_id2*.

srs_id2

A value of type INTEGER that identifies the second spatial reference system to be compared with the spatial reference system identified by *srs_id1*.

Return type

INTEGER

Example

Two similar spatial reference systems are created with the following calls to db2se.

```
db2se create_srs SAMP_DB -srsId 12 -srsName NYE_12 -xOffset 0 -yOffset 0
-xScale 1 -yScale 1 -coordsysName
NAD_1983_StatePlane_New_York_East_FIPS_3101_Feet
```

```
db2se create_srs SAMP_DB -srsId 22 -srsName NYE_22 -xOffset 0 -yOffset 0
-xScale 1 -yScale 1 -coordsysName
NAD_1983_StatePlane_New_York_East_FIPS_3101_Feet
```

These SRSs have the same offset and scale values, and they refer to the same coordinate systems. The only difference is in the defined name and the SRS ID. Therefore, the comparison returns 1, which indicates that they are the same.

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

VALUES ST_EqualSRS(12, 22)

Results:

```
1
-----
1
```

ST_ExteriorRing function

The ST_ExteriorRing function takes a polygon as an input parameter and returns its exterior ring as a curve. The resulting curve is represented in the spatial reference system of the given polygon.

If the given polygon is null or is empty, then null is returned. If the polygon does not have any interior rings, the returned exterior ring is identical to the boundary of the polygon.

This function can also be called as a method.

Syntax

►—db2gse.ST_ExteriorRing—(*—polygon—*)—◄

Parameter

polygon

A value of type ST_Polygon that represents the polygon for which the exterior ring is to be returned.

Return type

db2gse.ST_Curve

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example creates two polygons, one with two interior rings and one with no interior rings, then it determines their exterior rings.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)

INSERT INTO sample_polys VALUES
  (1, ST_Polygon('polygon((40 120, 90 120, 90 150, 40 150, 40 120),
    (50 130, 60 130, 60 140, 50 140, 50 130),
    (70 130, 80 130, 80 140, 70 140, 70 130))' ,0))

INSERT INTO sample_polys VALUES
  (2, ST_Polygon('polygon((10 10, 50 10, 10 30, 10 10))' ,0))

SELECT id, CAST(ST_AsText(ST_ExteriorRing(geometry))
  AS VARCHAR(180)) Exterior_Ring
FROM sample_polys
```

Results:

ID	EXTERIOR_RING
1	LINESTRING (40.00000000 120.00000000, 90.00000000 120.00000000, 90.00000000 150.00000000, 40.00000000 150.00000000, 40.00000000 120.00000000)
2	LINESTRING (10.00000000 10.00000000, 50.00000000 10.00000000, 10.00000000 30.00000000, 10.00000000 10.00000000)

ST_FindMeasure or ST_LocateAlong function

The ST_FindMeasure or ST_LocateAlong function takes a geometry and a measure as input parameters and returns a multipoint or multicurve of that part of the given geometry that has exactly the specified measure of the given geometry that contains the specified measure.

For points and multipoints, all the points with the specified measure are returned. For curves, multicurves, surfaces, and multisurfaces, interpolation is performed to compute the result. The computation for surfaces and multisurfaces is performed on the boundary of the geometry.

For points and multipoints, if the given measure is not found, then an empty geometry is returned. For all other geometries, if the given measure is lower than the lowest measure in the geometry or higher than the highest measure in the geometry, then an empty geometry is returned. If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►► `db2gse.ST_FindMeasure` `(—geometry—, —measure—)` ►►
 └─ `db2gse.ST_LocateAlong` ─┘

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry in which to search for parts whose M coordinates (measures) contain *measure*.

measure

A value of type DOUBLE that is the measure that the parts of *geometry* must be included in the result.

Return type

db2gse.ST_Geometry

Examples

Example 1

The following CREATE TABLE statement creates the SAMPLE_GEOMETRIES table. SAMPLE_GEOMETRIES has two columns: the ID column, which uniquely identifies each row, and the GEOMETRY ST_Geometry column, which stores sample geometry.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_geometries(id SMALLINT, geometry ST_GEOMETRY)
```

The following INSERT statements insert two rows. The first is a linestring; the second is a multipoint.


```

INSERT INTO sample_geometries(id, geometry)
VALUES
  (1, ST_LineString('linestring m (2 2 3, 3 5 3, 3 3 6, 4 4 8)', 1)),
  (2, ST_MultiPoint('multipoint m
    (2 2 3, 3 5 3, 3 3 6, 4 4 6, 5 5 6, 6 6 8)', 1))

```

Example 2

In the following SELECT statement and the corresponding result set, the ST_FindMeasure function is directed to find points whose measure is 7. The first row returns a point. However, the second row returns an empty point. For linear features (geometry with a dimension greater than 0), ST_FindMeasure can interpolate the point; however, for multipoints, the target measure must match exactly.

```

SELECT id, cast(ST_AsText(ST_FindMeasure(geometry, 7))
  AS varchar(45)) AS measure_7
FROM   sample_geometries

```

Results:

ID	MEASURE_7
1	POINT M (3.50000000 3.50000000 7.00000000)
2	POINT EMPTY

Example 3

In the following SELECT statement and the corresponding result set, the ST_FindMeasure function returns a point and a multipoint. The target measure of 6 matches the measures in both the ST_FindMeasure and multipoint source data.

```

SELECT id, cast(ST_AsText(ST_FindMeasure(geometry, 6))
  AS varchar(120)) AS measure_6
FROM   sample_geometries

```

Results:

ID	MEASURE_6
1	POINT M (3.00000000 3.00000000 6.00000000)
2	MULTIPOINT M (3.00000000 3.00000000 6.00000000, 4.00000000 4.00000000 6.00000000, 5.00000000 5.00000000 6.00000000)

ST_Generalize function

The ST_Generalize function takes a geometry and a threshold as input parameters and represents the given geometry with a reduced number of points, while preserving the general characteristics of the geometry.

The Douglas-Peucker line-simplification algorithm is used, by which the sequence of points that define the geometry is recursively subdivided until a run of the points can be replaced by a straight line segment. In this line segment, none of the defining points deviates from the straight line segment by more than the given threshold. Z and M coordinates are not considered for the simplification. The resulting geometry is in the spatial reference system of the given geometry.

If the given geometry is empty, an empty geometry of type ST_Point is returned. If the given geometry or the threshold is null, null is returned.

This function can also be called as a method.

Syntax

►—db2gse.ST_Generalize—(*—geometry—*,*—threshold—*)—►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry for which the line-simplification is applied.

threshold

A value of type DOUBLE that identifies the threshold to be used for the line-simplification algorithm. The threshold must be greater than or equal to 0 (zero). The larger the threshold, the smaller the number of points that will be used to represent the generalized geometry.

Return type

db2gse.ST_Geometry

Examples

In the following examples, the results have been reformatted for readability. The spacing in your results will vary according to your display.

Example 1

A linestring is created with eight points that go from (10, 10) to (80, 80). The path is almost a straight line, but some of the points are slightly off of the line. The ST_Generalize function can be used to reduce the number of points in the line.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines (id INTEGER, geometry ST_LineString)

INSERT INTO sample_lines VALUES
    (1, ST_LineString('linestring(10 10, 21 20, 34 26, 40 40,
                        52 50, 59 63, 70 71, 80 80)' ,0))
```

Example 2

When a generalization factor of 3 is used, the linestring is reduced to four coordinates, and is still very close to the original representation of the linestring.

```
SELECT CAST(ST_AsText(ST_Generalize(geometry, 3)) as VARCHAR(115))
       Generalize_3
FROM sample_lines
```

Results:

```
GENERALIZE 3
-----
LINESTRING ( 10.00000000 10.00000000, 34.00000000 26.00000000,
             59.00000000 63.00000000, 80.00000000 80.00000000)
```

Example 3

When a generalization factor of 6 is used, the linestring is reduced to only two coordinates. This produces a simpler linestring than the previous example, however it deviates more from the original representation.

```

SELECT CAST(ST_AsText(ST_Generalize(geometry, 6)) as VARCHAR(65))
       Generalize_6
FROM sample_lines

Results:
GENERALIZE 6
-----
LINESTRING ( 10.00000000 10.00000000, 80.00000000 80.00000000)

```

ST_GeomCollection function

Use the ST_GeomCollection function to construct a geometry collection.

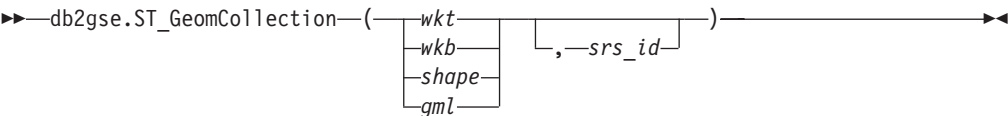
ST_GeomCollection constructs a geometry collection from one of the following inputs:

- A well-known text representation
- A well-known binary representation
- An ESRI shape representation
- A representation in the geography markup language (GML)

An optional spatial reference system identifier can be specified to identify the spatial reference system that the resulting geometry collection is in.

If the well-known text representation, the well-known binary representation, the ESRI shape representation, or the GML representation is null, then null is returned.

Syntax



Parameter

- | | |
|---------------|---|
| wkt | A value of type CLOB(2G) that contains the well-known text representation of the resulting geometry collection. |
| wkb | A value of type BLOB(2G) that contains the well-known binary representation of the resulting geometry collection. |
| shape | A value of type BLOB(2G) that represents the ESRI shape representation of the resulting geometry collection. |
| gml | A value of type CLOB(2G) that represents the resulting geometry collection using the geography markup language (GML). |
| srs_id | A value of type INTEGER that identifies the spatial reference system for the resulting geometry collection. |

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used implicitly.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an error is returned (SQLSTATE 38SU1).

Return type

db2gse.ST_GeomCollection

Notes

If the *srs_id* parameter is omitted, it might be necessary to cast *wkt* and *gml* explicitly to the CLOB data type. Otherwise, DB2 might resolve to the function used to cast values from the reference type REF(ST_GeomCollection) to the ST_GeomCollection type. The following example ensures that DB2 resolves to the correct function:

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The following code illustrates how the ST_GeomCollection function can be used to create and insert a multipoint, multiline, and multipolygon from well-known text (WKT) representation and a multipoint from geographic markup language (GML) into a GeomCollection column.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_geomcollections(id INTEGER,  
    geometry ST_GEOMCOLLECTION)
```

```
INSERT INTO sample_geomcollections(id, geometry)  
VALUES
```

```
(4001, ST_GeomCollection('multipoint(1 2, 4 3, 5 6)', 1) ),  
(4002, ST_GeomCollection('multilinestring(  
    (33 2, 34 3, 35 6),  
    (28 4, 29 5, 31 8, 43 12),  
    (39 3, 37 4, 36 7))', 1) ),  
(4003, ST_GeomCollection('multipolygon(((3 3, 4 6, 5 3, 3 3),  
    (8 24, 9 25, 1 28, 8 24),  
    (13 33, 7 36, 1 40, 10 43, 13 33)))', 1)),  
(4004, ST_GeomCollection('<gml:MultiPoint srsName="EPSG:4269"  
><gml:PointMember><gml:Point>  
<gml:coord><gml:X>10</gml:X>  
<gml:Y>20</gml:Y></gml:coord></gml:Point>  
</gml:PointMember><gml:PointMember>  
<gml:Point><gml:coord><gml:X>30</gml:X>  
<gml:Y>40</gml:Y></gml:coord></gml:Point>  
</gml:PointMember></gml:MultiPoint>', 1))
```

```
SELECT id, cast(geometry..ST_AsText AS varchar(350)) AS geomcollection  
FROM    sample_geomcollections
```

Results:

ID	GEOMCOLLECTION
4001	MULTIPOINT (1.00000000 2.00000000, 4.00000000 3.00000000, 5.00000000 6.00000000)
4002	MULTILINESTRING ((33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000),(28.00000000 4.00000000, 29.00000000 5.00000000, 31.00000000 8.00000000, 43.00000000 12.00000000),(39.00000000 3.00000000, 37.00000000 4.00000000, 36.00000000 7.00000000))
4003	MULTIPOLYGON (((13.00000000 33.00000000, 10.00000000 43.00000000, 1.00000000 40.00000000, 7.00000000 36.00000000,

```

13.00000000 33.00000000)),(( 8.00000000 24.00000000, 9.00000000
25.00000000, 1.00000000 28.00000000, 8.00000000 24.00000000)),
(( 3.00000000 3.00000000,5.00000000 3.00000000, 4.00000000
6.00000000,3.00000000 3.00000000)))

4004          MULTIPOINT ( 10.00000000 20.00000000, 30.00000000
40.00000000)

```

ST_GeomCollFromTxt function

The ST_GeomCollFromTxt function takes a well-known text representation of a geometry collection and, optionally, a spatial reference system identifier as input parameters and returns the corresponding geometry collection.

If the given well-known text representation is null, then null is returned.

The recommended function for achieving the same result is ST_GeomCollection. It is recommended because of its flexibility: ST_GeomCollection takes additional forms of input as well as the well-known binary representation.

Syntax

```

►► db2gse.ST_GeomCollFromTxt (—wkt— [,—srs_id—] )—►►

```

Parameter

wkt A value of type CLOB(2G) that contains the well-known text representation of the resulting geometry collection.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting geometry collection.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used implicitly.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an error is returned (SQLSTATE 38SU1).

Return type

db2gse.ST_GeomCollection

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The following code illustrates how the ST_GeomCollFromTxt function can be used to create and insert a multipoint, multiline, and multipolygon from a well-known text (WKT) representation into a GeomCollection column.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sample_geomcollections(id INTEGER, geometry ST_GEOMCOLLECTION)

INSERT INTO sample_geomcollections(id, geometry)
VALUES
(4011, ST_GeomCollFromTxt('multipoint(1 2, 4 3, 5 6)', 1) ),

```

```

(4012, ST_GeomCollFromTxt('multilinestring(
                        (33 2, 34 3, 35 6),
                        (28 4, 29 5, 31 8, 43 12),
                        (39 3, 37 4, 36 7))', 1) ),
(4013, ST_GeomCollFromTxt('multipolygon(((3 3, 4 6, 5 3, 3 3),
                        (8 24, 9 25, 1 28, 8 24),
                        (13 33, 7 36, 1 40, 10 43, 13 33)))', 1))

SELECT id, cast(geometry..ST_AsText AS varchar(340))
       AS geomcollection
FROM   sample_geomcollections

```

Results:

ID	GEOMCOLLECTION
4011	MULTIPOINT (1.00000000 2.00000000, 4.00000000 3.00000000, 5.00000000 6.00000000)
4012	MULTILINESTRING ((33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000),(28.00000000 4.00000000, 29.00000000 5.00000000, 31.00000000 8.00000000, 43.00000000 12.00000000),(39.00000000 3.00000000, 37.00000000 4.00000000, 36.00000000 7.00000000))
4013	MULTIPOLYGON (((13.00000000 33.00000000, 10.00000000 43.00000000, 1.00000000 40.00000000, 7.00000000 36.00000000, 13.00000000 33.00000000)), ((8.00000000 24.00000000, 9.00000000 25.00000000, 1.00000000 28.00000000, 8.00000000 24.00000000)),((3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000, 3.00000000 3.00000000)))

ST_GeomCollFromWKB function

The ST_GeomCollFromWKB function takes a well-known binary representation of a geometry collection and, optionally, a spatial reference system identifier as input parameters and returns the corresponding geometry collection.

If the given well-known binary representation is null, then null is returned.

The preferred version for this functionality is ST_GeomCollection.

Syntax

```

▶▶ db2gse.ST_GeomCollFromWKB (—wkb— [,—srs_id—]) ▶▶

```

Parameter

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting geometry collection.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting geometry collection.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used implicitly.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an error is returned (SQLSTATE 38SU1).

Return type

db2gse.ST_GeomCollection

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The following code illustrates how the ST_GeomCollFromWKB function can be used to create and query the coordinates of a geometry collection in a well-known binary representation. The rows are inserted into the SAMPLE_GEOMCOLLECTION table with IDs 4021 and 4022 and geometry collections in spatial reference system 1.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_geomcollections(id INTEGER,  
  geometry ST_GEOMCOLLECTION, wkb BLOB(32k))
```

```
INSERT INTO sample_geomcollections(id, geometry)  
VALUES  
  (4021, ST_GeomCollFromTxt('multipoint(1 2, 4 3, 5 6)', 1)),  
  (4022, ST_GeomCollFromTxt('multilinestring(  
    (33 2, 34 3, 35 6),  
    (28 4, 29 5, 31 8, 43 12))', 1))
```

```
UPDATE sample_geomcollections AS temp_correlated  
SET    wkb = geometry..ST_AsBinary  
WHERE  id = temp_correlated.id
```

```
SELECT id, cast(ST_GeomCollFromWKB(wkb)..ST_AsText  
  AS varchar(190)) AS GeomCollection  
FROM   sample_geomcollections
```

Results:

ID	GEOMCOLLECTION
4021	MULTIPOINT (1.00000000 2.00000000, 4.00000000 3.00000000, 5.00000000 6.00000000)
4022	MULTILINESTRING ((33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000),(28.00000000 4.00000000, 29.00000000 5.00000000, 31.00000000 8.00000000, 43.00000000 12.00000000))

ST_Geometry function

The ST_Geometry function constructs a geometry from a given representation.

ST_Geometry constructs a geometry from one of the following inputs:

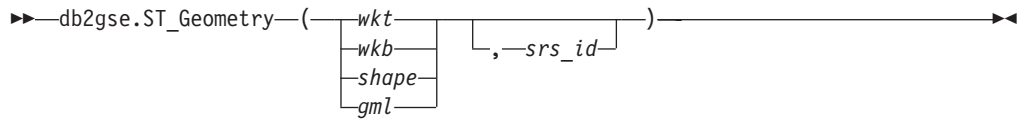
- A well-known text representation
- A well-known binary representation
- An ESRI shape representation
- A representation in the geography markup language (GML)

An optional spatial reference system identifier can be specified to identify the spatial reference system that the resulting geometry is in.

The dynamic type of the resulting geometry is one of the instantiable subtypes of ST_Geometry.

If the well-known text representation, the well-known binary representation, the ESRI shape representation, or the GML representation is null, then null is returned.

Syntax



Parameter

wkt A value of type CLOB(2G) that contains the well-known text representation of the resulting geometry.

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting geometry.

shape A value of type BLOB(2G) that represents the ESRI shape representation of the resulting geometry.

gml A value of type CLOB(2G) that represents the resulting geometry using the geography markup language (GML).

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting geometry.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used implicitly.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an error is returned (SQLSTATE 38SU1).

Return type

db2gse.ST_Geometry

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The following code illustrates how the ST_Geometry function can be used to create and insert a point from a well-known text (WKT) point representation or line from Geographic Markup Language (GML) line representation.

The ST_Geometry function is the most flexible of the spatial type constructor functions because it can create any spatial type from various geometry representations. ST_LineFromText can create only a line from WKT line representation. ST_WKTToSql can construct any type, but only from WKT representation.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_geometries(id INTEGER, geometry ST_GEOMETRY)
```



```

INSERT INTO sample_geometries(id, geometry)
VALUES
  (7001, ST_Geometry('point(1 2)', 1) ),
  (7002, ST_Geometry('linestring(33 2, 34 3, 35 6)', 1) ),
  (7003, ST_Geometry('polygon((3 3, 4 6, 5 3, 3 3))', 1)),
  (7004, ST_Geometry('<gml:Point srsName="";EPSG:4269";><gml:coord>
    <gml:X>50</gml:X><gml:Y>60</gml:Y></gml:coord>
  </gml:Point>', 1))

SELECT id, cast(geometry..ST_AsText AS varchar(120)) AS geometry
FROM   sample_geometries

```

Results:

ID	GEOMETRY
7001	POINT (1.00000000 2.00000000)
7002	LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000)
7003	POLYGON ((3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000, 3.00000000 3.00000000))
7004	POINT (50.00000000 60.00000000)

ST_GeometryN function

The ST_GeometryN function takes a geometry collection and an index as input parameters and returns the geometry in the collection that is identified by the index. The resulting geometry is represented in the spatial reference system of the given geometry collection.

If the given geometry collection is null or is empty, or if the index is smaller than 1 or larger than the number of geometries in the collection, then null is returned and a warning condition is raised (01HS0).

This function can also be called as a method.

Syntax

```

▶▶—db2gse.ST_GeometryN—(—collection—,—index—)—▶▶

```

Parameter

collection

A value of type ST_GeomCollection or one of its subtypes that represents the geometry collection to locate the *n*th geometry within.

index A value of type INTEGER that identifies the *n*th geometry that is to be returned from *collection*.

If *index* is smaller than 1 or larger than the number of geometries in the collection, then null is returned and a warning is returned (SQLSTATE 01HS0).

Return type

db2gse.ST_Geometry

Example

The following code illustrates how to choose the second geometry inside a geometry collection.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sample_geomcollections (id INTEGER,
  geometry ST_GEOMCOLLECTION)

INSERT INTO sample_geomcollections(id, geometry)
VALUES
  (4001, ST_GeomCollection('multipoint(1 2, 4 3)', 1) ),
  (4002, ST_GeomCollection('multilinestring(
    (33 2, 34 3, 35 6),
    (28 4, 29 5, 31 8, 43 12),
    (39 3, 37 4, 36 7))', 1) ),
  (4003, ST_GeomCollection('multipolygon(((3 3, 4 6, 5 3, 3 3),
    (8 24, 9 25, 1 28, 8 24),
    (13 33, 7 36, 1 40, 10 43, 13 33)))', 1))

SELECT id, cast(ST_GeometryN(geometry, 2)..ST_AsText AS varchar(110))
  AS second_geometry
FROM   sample_geomcollections
```

Results:

ID	SECOND_GEOMETRY
4001	POINT (4.00000000 3.00000000)
4002	LINESTRING (28.00000000 4.00000000, 29.00000000 5.00000000, 31.00000000 8.00000000, 43.00000000 12.00000000)
4003	POLYGON ((8.00000000 24.00000000, 9.00000000 25.00000000, 1.00000000 28.00000000, 8.00000000 24.00000000))

ST_GeometryType function

The ST_GeometryType function takes a geometry as input parameter and returns the fully qualified type name of the dynamic type of that geometry.

The DB2 functions TYPE_SCHEMA and TYPE_NAME have the same effect.

This function can also be called as a method.

Syntax

►►db2gse.ST_GeometryType(—*geometry*—)◄◄

Parameter

geometry

A value of type ST_Geometry for which the geometry type is to be returned.

Return type

VARCHAR(128)

Examples

The following code illustrates how to determine the type of a geometry.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_geometries (id INTEGER, geometry ST_GEOMETRY)
```

```
INSERT INTO sample_geometries(id, geometry)
VALUES
  (7101, ST_Geometry('point(1 2)', 1) ),
  (7102, ST_Geometry('linestring(33 2, 34 3, 35 6)', 1) ),
  (7103, ST_Geometry('polygon((3 3, 4 6, 5 3, 3 3))', 1)),
  (7104, ST_Geometry('multipoint(1 2, 4 3)', 1) )
```

```
SELECT id, geometry..ST_GeometryType AS geometry_type
FROM sample_geometries
```

Results:

ID	GEOMETRY_TYPE
7101	"DB2GSE "ST_POINT"
7102	"DB2GSE "ST_LINESTRING"
7103	"DB2GSE "ST_POLYGON"
7104	"DB2GSE "ST_MULTIPOINT"

ST_GeomFromText function

The `ST_GeomFromText` function takes a well-known text representation of a geometry and, optionally, a spatial reference system identifier as input parameters and returns the corresponding geometry.

If the given well-known text representation is null, then null is returned.

The preferred version for this functionality is ST_Geometry.

Syntax

```
db2gse.ST_GeomFromText((-wkt  
                        |, -srs id|))
```

Parameter

wkt	A value of type CLOB(2G) that contains the well-known text representation of the resulting geometry.
------------	--

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting geometry.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used implicitly.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an error is returned (SQLSTATE 38SU1).

Return type

db2gse.ST_Geometry

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

In this example the ST_GeomFromText function is used to create and insert a point from a well known text (WKT) point representation.

The following code inserts rows into the SAMPLE_POINTS table with IDs and geometries in spatial reference system 1 using WKT representation.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sample_geometries(id INTEGER, geometry ST_GEOMETRY)

INSERT INTO sample_geometries(id, geometry)
VALUES
    (1251, ST_GeomFromText('point(1 2)', 1) ),
    (1252, ST_GeomFromText('linestring(33 2, 34 3, 35 6)', 1) ),
    (1253, ST_GeomFromText('polygon((3 3, 4 6, 5 3, 3 3))', 1))
```

The following SELECT statement will return the ID and GEOMETRIES from the SAMPLE_GEOMETRIES table.

```
SELECT id, cast(geometry..ST_AsText AS varchar(105))
       AS geometry
FROM   sample_geometries
```

Results:

ID	GEOMETRY
1251	POINT (1.00000000 2.00000000)
1252	LINestring (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000)
1253	POLYGON ((3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000, 3.00000000 3.00000000))

ST_GeomFromWKB function

The ST_GeomFromWKB function takes a well-known binary representation of a geometry and, optionally, a spatial reference system identifier as input parameters and returns the corresponding geometry.

If the given well-known binary representation is null, then null is returned.

The preferred version for this functionality is ST_Geometry.

Syntax

►►db2gse.ST_GeomFromWKB(—wkb—
 └─,—srs_id─┘)────────────────────────────────►►

Parameter

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting geometry.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting geometry.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used implicitly.

If the specified *srs_id* parameter does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an error is returned (SQLSTATE 38SU1).

Return type

db2gse.ST_Geometry

Examples

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The following code illustrates how the ST_GeomFromWKB function can be used to create and insert a line from a well-known binary (WKB) line representation.

The following example inserts a record into the SAMPLE_GEOMETRIES table with an ID and a geometry in spatial reference system 1 in a WKB representation.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sample_geometries (id INTEGER, geometry ST_GEOMETRY,
                                wkb BLOB(32K))

INSERT INTO sample_geometries(id, geometry)
VALUES
    (1901, ST_GeomFromText('point(1 2)', 1) ),
    (1902, ST_GeomFromText('linestring(33 2, 34 3, 35 6)', 1) ),
    (1903, ST_GeomFromText('polygon((3 3, 4 6, 5 3, 3 3))', 1))

UPDATE sample_geometries AS temp_correlated
SET    wkb = geometry..ST_AsBinary
WHERE  id = temp_correlated.id

SELECT id, cast(ST_GeomFromWKB(wkb)..ST_AsText AS varchar(190))
       AS geometry
FROM    sample_geometries
```

Results:

ID	GEOMETRY
1901	POINT (1.00000000 2.00000000)
1902	LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000)
1903	POLYGON ((3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000, 3.00000000 3.00000000))

MBR Aggregate functions

Use the `ST_BuildMBRAggr` and `ST_GetAggrResult` functions combined to aggregate a set of geometries from a column into a single geometry. The combination constructs a rectangle that represents the minimum bounding rectangle that encloses all the geometries in the column. Z and M coordinates are discarded when the aggregate is computed.

The following expression is an example that uses the `MAX` function with the `db2gse.ST_BuildMBRAggr` spatial function to compute the MBR of the geometries in the `columnName` column and the `db2gse.ST_GetAggrResult` spatial function to return the resulting geometry that was computed for the MBR:

```
db2gse.ST_Get_AggrResult(MAX(db2gse.ST_BuildMBRAggr(columnName)))
```

If all of the geometries to be combined are null, then null is returned. If all of the geometries are either null or empty, then an empty geometry is returned. If the minimum bounding rectangle of all the geometries to be combined results in a point, then this point is returned as an `ST_Point` value. If the minimum bounding rectangle of all the geometries to be combined results in a horizontal or vertical linestring, then this linestring is returned as an `ST_LineString` value. Otherwise, the minimum bounding rectangle is returned as an `ST_Polygon` value.

Syntax

```
►►db2gse.ST_GetAggrResult(—MAX—(—————)—————)—————►
►db2gse.ST_BuildMBRAggr(—geometries—)——)—————►►
```

Parameter

geometries

A selected column that has a type of `ST_Geometry` or one of its subtypes and represents all the geometries for which the minimum bounding rectangle is to be computed.

Return type

`db2gse.ST_Geometry`

Restrictions

You cannot construct the union aggregate of a spatial column in a full-select in any of the following situations:

- In a partitioned database environment
- If `GROUP BY` clause is used in the full-select.
- If you use a function other than the DB2 aggregate function `MAX`.

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example shows how to use the `ST_BuildMBRAggr` function to obtain the maximum bounding rectangle of all of the geometries within a column. In this

example, several points are added to the GEOMETRY column in the SAMPLE_POINTS table. The SQL code then determines the maximum bounding rectangle of all of the points put together.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_points (id integer, geometry ST_Point)
```

```
INSERT INTO sample_points (id, geometry)
VALUES
  (1, ST_Point(2, 3, 1)),
  (2, ST_Point(4, 5, 1)),
  (3, ST_Point(13, 15, 1)),
  (4, ST_Point(12, 5, 1)),
  (5, ST_Point(23, 2, 1)),
  (6, ST_Point(11, 4, 1))
```

```
SELECT cast(ST_GetAggrResult(MAX(ST_BuildMbrAggr
  (geometry)))..ST_AsText AS varchar(160))
  AS ";Aggregate_of_Points";
FROM sample_points
```

Results:

```
Aggregate_of_Points
-----
POLYGON (( 2.00000000 2.00000000, 23.00000000 2.00000000,
23.00000000 15.00000000, 2.00000000 15.00000000, 2.00000000
2.00000000))
```

Union aggregate functions

A union aggregate is the combination of the ST_BuildUnionAggr and ST_GetAggrResult functions. Use this combination to aggregate a column of geometries in a table to single geometry by constructing the union.

If all of the geometries to be combined in the union are null , then null is returned. If each of the geometries to be combined in the union are either null or are empty, then an empty geometry of type ST_Point is returned.

The ST_BuildUnionAggr function can also be called as a method.

Syntax

```
►►—db2gse.ST_GetAggrResult—(—————)—————►
►—MAX—(—db2gse.ST_BuildUnionAggr—(—geometries—)—)—)—————►◄
```

Parameters

geometries

A column in a table that has a type of ST_Geometry or one of its subtypes and represents all the geometries that are to be combined into a union.

Return type

db2gse.ST_Geometry

Restrictions

You cannot construct the union aggregate of a spatial column in a table in any of the following situations:

- In partitioned database environments
- If a GROUP BY clause is used in the select
- If you use a function other than the DB2 aggregate function MAX

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how a union aggregate can be used to combine a set of points into multipoints. Several points are added to the SAMPLE_POINTS table. The ST_GetAggrResult and ST_BuildUnionAggr functions are used to construct the union of the points.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)

INSERT INTO sample_points
VALUES (1, ST_Point (2, 3, 1) )
INSERT INTO sample_points
VALUES (2, ST_Point (4, 5, 1) )
INSERT INTO sample_points
VALUES (3, ST_Point (13, 15, 1) )
INSERT INTO sample_points
VALUES (4, ST_Point (12, 5, 1) )
INSERT INTO sample_points
VALUES (5, ST_Point (23, 2, 1) )
INSERT INTO sample_points
VALUES (6, ST_Point (11, 4, 1) )

SELECT CAST (ST_AsText(
            ST_GetAggrResult( MAX( ST_BuildUnionAggregate (geometry) ) ))
        AS VARCHAR(160)) POINT_AGGREGATE
FROM sample_points
```

Results:

```
POINT_AGGREGATE
-----
MULTIPOINT ( 2.00000000 3.00000000, 4.00000000 5.00000000,
             11.00000000 4.00000000, 12.00000000 5.00000000,
             13.00000000 15.00000000, 23.00000000 2.00000000)
```

ST_GetIndexParms function

The ST_GetIndexParms function takes either the identifier for a spatial index or for a spatial column as an input parameter and returns the parameters used to define the index or the index on the spatial column. If an additional parameter number is specified, only the grid size identified by the number is returned.

Syntax

►►db2gse.ST_GetIndexParms—(—————►


```

▶ (index_schema—,—index_name—,
  table_schema—,—table_name—,—column_name—,
  —,—grid_size_number—)▶

```

Parameter

index_schema

A value of type VARCHAR(128) that identifies the schema in which the spatial index with the unqualified name *index_name* is in. The schema name is case-sensitive and must be listed in the SYSCAT.SCHEMATA catalog view.

If this parameter is null, then the value of the CURRENT SCHEMA special register is used as the schema name for the spatial index.

index_name

A value of type VARCHAR(128) that contains the unqualified name of the spatial index for which the index parameters are returned. The index name is case-sensitive and must be listed in the SYSCAT.INDEXES catalog view for the schema *index_schema*.

table_schema

A value of type VARCHAR(128) that identifies the schema in which the table with the unqualified name *table_name* is in. The schema name is case-sensitive and must be listed in the SYSCAT.SCHEMATA catalog view.

If this is parameter null, then the value of the CURRENT SCHEMA special register is used as the schema name for the spatial index.

table_name

A value of type VARCHAR(128) that contains the unqualified name of the table with the spatial column *column_name*. The table name is case-sensitive and must be listed in the SYSCAT.TABLES catalog view for the schema *table_schema*.

column_name

A value of type VARCHAR(128) that identifies the column in the table *table_schema.table_name* for which the index parameters of the spatial index on that column are returned. The column name is case-sensitive and must be listed in the SYSCAT.COLUMNS catalog view for the table *table_schema.table_name*.

If there is no spatial index defined in the column, then an error is raised (SQLSTATE 38SQ0).

grid_size_number

A DOUBLE value that identifies the parameter whose value or values are to be returned.

If this value is smaller than 1 or larger than 3, then an error is raised (SQLSTATE 38SQ1).

Return type

DOUBLE (if *grid_size_number* is specified)

If *grid_size_number* is not specified, then a table with the two columns ORDINAL and VALUE is returned. The column ORDINAL is of type INTEGER, and the column VALUE is of type DOUBLE.

If the parameters are returned for a grid index, the ORDINAL column contains the values 1, 2, and 3 for the first, second, and third grid size, respectively. The column VALUE contains the grid sizes.

The VALUE column contains the respective values for each of the parameters.

Examples

Example 1

This code creates a table with a spatial column and a spatial index.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sch.offices (name VARCHAR(30), location ST_Point )

CREATE INDEX sch.idx ON sch.offices(location)
    EXTEND USING db2gse.spatial_index(1e0, 10e0, 1000e0)
```

The ST_GetIndexParms function can be used to retrieve the values for the parameters that were used when the spatial index was created.

Example 2

This example shows how to retrieve the three grid sizes for a spatial grid index separately by explicitly specifying which parameter, identified by its number, is to be returned.

```
VALUES ST_GetIndexParms('SCH', 'OFFICES', 'LOCATION', 1)
```

Results:

```
1
-----
+1.000000000000000E+000
VALUES ST_GetIndexParms('SCH', 'OFFICES', 'LOCATION', 2)
```

Results:

```
1
-----
+1.000000000000000E+001
VALUES ST_GetIndexParms('SCH', 'IDX', 3)
```

Results:

```
1
-----
+1.000000000000000E+003
```

Example 3

This example shows how to retrieve all the parameters of a spatial grid index. The ST_GetIndexParms function returns a table that indicates the parameter number and the corresponding grid size.

```
SELECT * FROM TABLE ( ST_GetIndexParms('SCH', 'OFFICES', 'LOCATION') ) AS t
```

Results:

```
ORDINAL      VALUE
-----
1      +1.000000000000000E+000
2      +1.000000000000000E+001
3      +1.000000000000000E+003
SELECT * FROM TABLE ( ST_GetIndexParms('SCH', 'IDX') ) AS t
```

```
Results:
ORDINAL  VALUE
-----
1      +1.000000000000000E+000
2      +1.000000000000000E+001
3      +1.000000000000000E+003
```

ST_InteriorRingN function

The ST_InteriorRingN function takes a polygon and an index as input parameters and returns the interior ring identified by the given index as a linestring. The interior rings are organized according to the rules defined by the internal geometry verification routines.

If the given polygon is null or is empty, or if it does not have any interior rings, then null is returned. If the index is smaller than 1 or larger than the number of interior rings in the polygon, then null is returned and a warning condition is raised (1HS1).

This function can also be called as a method.

Syntax

```
►►db2gse.ST_InteriorRingN(—polygon—,—index—)◄◄
```

Parameter

polygon

A value of type ST_Polygon that represents the geometry from which the interior ring identified by *index* is returned.

index

A value of type INTEGER that identifies the *n*th interior ring that is returned. If there is no interior ring identified by *index*, then a warning condition is raised (01HS1).

Return type

db2gse.ST_Curve

Example

In this example, a polygon is created with two interior rings. The ST_InteriorRingN call is then used to retrieve the second interior ring.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)

INSERT INTO sample_polys VALUES
  (1, ST_Polygon('polygon((40 120, 90 120, 90 150, 40 150, 40 120),
    (50 130, 60 130, 60 140, 50 140, 50 130),
    (70 130, 80 130, 80 140, 70 140, 70 130))',0))

SELECT id, CAST(ST_AsText(ST_InteriorRingN(geometry, 2)) as VARCHAR(180))
       Interior_Ring
FROM sample_polys
```

Results:

ID	INTERIOR_RING
1	LINESTRING (70.00000000 130.00000000, 70.00000000 140.00000000, 80.00000000 140.00000000, 80.00000000 130.00000000, 70.00000000 130.00000000)

ST_Intersection function

The ST_Intersection function takes two geometries as input parameters and returns the geometry that is the intersection of the two given geometries. The intersection is the common part of the first geometry and the second geometry. The resulting geometry is represented in the spatial reference system of the first geometry.

If possible, the specific type of the returned geometry will be ST_Point, ST_LineString, or ST_Polygon. For example, the intersection of a point and a polygon is either empty or a single point, represented as ST_MultiPoint.

If any of the two geometries is null, null is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry and uses the same underlying datum, it will be converted to the other spatial reference system.

This function can also be called as a method.

Syntax

►►db2gse.ST_Intersection(—*geometry1*—,—*geometry2*—)◀◀

Parameter

geometry1

A value of type ST_Geometry or one of its subtypes that represents the first geometry to compute the intersection with *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the second geometry to compute the intersection with *geometry1*.

Return type

db2gse.ST_Geometry

The dimension of the returned geometry is that of the input with the lower dimension.

Example

In the following examples, the results have been reformatted for readability. The spacing in your results will vary according to your display.

This example creates several different geometries and then determines the intersection (if any) with the first one.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
```

```

        (1, ST_Geometry('polygon((30 30, 30 50, 50 50, 50 30, 30 30))' ,0))
INSERT INTO sample_geoms VALUES
        (2, ST_Geometry('polygon((20 30, 30 30, 30 40, 20 40, 20 30))' ,0))
INSERT INTO sample_geoms VALUES
        (3, ST_Geometry('polygon((40 40, 40 60, 60 60, 60 40, 40 40))' ,0))
INSERT INTO sample_geoms VALUES
        (4, ST_Geometry('linestring(60 60, 70 70)' ,0))
INSERT INTO sample_geoms VALUES
        (5, ST_Geometry('linestring(30 30, 60 60)' ,0))

SELECT a.id, b.id, CAST(ST_AsText(ST_Intersection(a.geometry, b.geometry))
        as VARCHAR(150)) Intersection
FROM sample_geoms a, sample_geoms b
WHERE a.id = 1

```

Results:

ID	ID	INTERSECTION
1	1	POLYGON ((30.00000000 30.00000000, 50.00000000 30.00000000, 50.00000000 50.00000000, 30.00000000 50.00000000, 30.00000000 30.00000000))
1	2	LINESTRING (30.00000000 40.00000000, 30.00000000 30.00000000)
1	3	POLYGON ((40.00000000 40.00000000, 50.00000000 40.00000000, 50.00000000 50.00000000, 40.00000000 50.00000000, 40.00000000 40.00000000))
1	4	POINT EMPTY
1	5	LINESTRING (30.00000000 30.00000000, 50.00000000 50.00000000)

5 record(s) selected.

ST_Intersects function

Use the the ST_Intersects function to determine whether two geometries intersect.

Syntax

```

►►—db2gse.ST_Intersects—(—geometry1—,—geometry2—)—►►

```

Parameter

geometry1

A value of type ST_Geometry or one of its subtypes that represents the geometry to test for intersection with *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the geometry to test for intersection with *geometry1*.

Return type

INTEGER

Usage

ST_Intersects takes two geometries as input parameters and returns 1 if the given geometries intersect. If the geometries do not intersect, 0 (zero) is returned.

If any of the two geometries is null or is empty, null is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry and uses the same underlying datum, it will be converted to the other spatial reference system.

ST_Intersects returns the exact opposite result of ST_Disjoint.

The ST_Intersects function returns 1 (one) if the conditions of any of the following pattern matrices returns TRUE.

Table 28. Matrix for ST_Intersects (1). The ST_Intersects function returns 1 (one) if the interiors of both geometries intersect.

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

Table 29. Matrix for ST_Intersects (2). The ST_Intersects function returns 1 (one) if the boundary of the first geometry intersects the boundary of the second geometry.

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

Table 30. Matrix for ST_Intersects (3). The ST_Intersects function returns 1 (one) if the boundary of the first geometry intersects the interior of the second.

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

Table 31. Matrix for ST_Intersects (4). The ST_Intersects function returns 1 (one) if the boundaries of either geometry intersect.

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

Usage

Example

The following statements create and populate the SAMPLE_GEOMETRIES1 and SAMPLE_GEOMETRIES2 tables.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sample_geometries1(id SMALLINT, spatial_type varchar(13),
    geometry ST_GEOMETRY);
CREATE TABLE sample_geometries2(id SMALLINT, spatial_type varchar(13),
    geometry ST_GEOMETRY);

INSERT INTO sample_geometries1(id, spatial_type, geometry)
VALUES
    ( 1, 'ST_Point', ST_Point('point(550 150)', 1) ),
    (10, 'ST_LineString', ST_LineString('linestring(800 800, 900 800)', 1)),
    (20, 'ST_Polygon', ST_Polygon('polygon((500 100, 500 200, 700 200,
        700 100, 500 100))', 1) )

INSERT INTO sample_geometries2(id, spatial_type, geometry)
VALUES
    (101, 'ST_Point', ST_Point('point(550 150)', 1) ),
    (102, 'ST_Point', ST_Point('point(650 200)', 1) ),
    (103, 'ST_Point', ST_Point('point(800 800)', 1) ),
    (110, 'ST_LineString', ST_LineString('linestring(850 250, 850 850)', 1)),
    (120, 'ST_Polygon', ST_Polygon('polygon((650 50, 650 150, 800 150,
        800 50, 650 50))', 1)),
    (121, 'ST_Polygon', ST_Polygon('polygon((20 20, 20 40, 40 40, 40 20,
        20 20))', 1) )
```

The following SELECT statement determines whether the various geometries in the SAMPLE_GEOMETRIES1 and SAMPLE_GEOMETRIES2 tables intersect.

```
SELECT    sg1.id AS sg1_id, sg1.spatial_type AS sg1_type,
          sg2.id AS sg2_id, sg2.spatial_type AS sg2_type,
          CASE ST_Intersects(sg1.geometry, sg2.geometry)
              WHEN 0 THEN 'Geometries do not intersect'
              WHEN 1 THEN 'Geometries intersect'
          END AS intersects
FROM      sample_geometries1 sg1, sample_geometries2 sg2
ORDER BY  sg1.id
```

Results:

SG1_ID	SG1_TYPE	SG2_ID	SG2_TYPE	INTERSECTS
1	ST_Point	101	ST_Point	Geometries intersect
1	ST_Point	102	ST_Point	Geometries do not intersect
1	ST_Point	103	ST_Point	Geometries do not intersect
1	ST_Point	110	ST_LineString	Geometries do not intersect
1	ST_Point	120	ST_Polygon	Geometries do not intersect
1	ST_Point	121	ST_Polygon	Geometries do not intersect
10	ST_LineString	101	ST_Point	Geometries do not intersect
10	ST_LineString	102	ST_Point	Geometries do not intersect
10	ST_LineString	103	ST_Point	Geometries intersect
10	ST_LineString	110	ST_LineString	Geometries intersect
10	ST_LineString	120	ST_Polygon	Geometries do not intersect
10	ST_LineString	121	ST_Polygon	Geometries do not intersect
20	ST_Polygon	101	ST_Point	Geometries intersect
20	ST_Polygon	102	ST_Point	Geometries intersect
20	ST_Polygon	103	ST_Point	Geometries do not intersect
20	ST_Polygon	110	ST_LineString	Geometries do not intersect
20	ST_Polygon	120	ST_Polygon	Geometries intersect
20	ST_Polygon	121	ST_Polygon	Geometries do not intersect

ST_Is3d function

The ST_Is3d function takes a geometry as an input parameter and returns 1 if the given geometry has Z coordinates. Otherwise, 0 (zero) is returned.

If the given geometry is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►►db2gse.ST_Is3D(—*geometry*—)—————►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry that is to be tested for the existence of Z coordinates.

Return type

INTEGER

Example

In this example, several geometries are created with and without Z coordinates and M coordinates (measures). ST_Is3d is then used to determine which of them contain Z coordinates.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
  (1, ST_Geometry('point EMPTY',0))

INSERT INTO sample_geoms VALUES
  (2, ST_Geometry('polygon((40 120, 90 120, 90 150, 40 150, 40 120))' ,0))

INSERT INTO sample_geoms VALUES
  (3, ST_Geometry('multipoint m (10 10 5, 50 10 6, 10 30 8)' ,0))

INSERT INTO sample_geoms VALUES
  (4, ST_Geometry('linestring z (10 10 166, 20 10 168)',0))

INSERT INTO sample_geoms VALUES
  (5, ST_Geometry('point zm (10 10 16 30)' ,0))

SELECT id, ST_Is3d(geometry) Is_3D
FROM sample_geoms
```

Results:

ID	IS_3D
1	0
2	0
3	0
4	1
5	1

ST_IsClosed function

The ST_IsClosed function takes a curve or multicurve as an input parameter and returns 1 if the given curve or multicurve is closed. Otherwise, 0 (zero) is returned.

A curve is closed if the start point and end point are equal. If the curve has Z coordinates, the Z coordinates of the start and end point must be equal. Otherwise, the points are not considered equal, and the curve is not closed. A multicurve is closed if each of its curves are closed.

If the given curve or multicurve is empty, then 0 (zero) is returned. If it is null, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_IsClosed—(—*curve*—)—————►►

Parameter

curve A value of type ST_Curve or ST_MultiCurve or one of their subtypes that represent the curve or multicurve that is to be tested.

Return type

INTEGER

Examples

Example 1

This example creates several linestrings. The last two linestrings have the same X and Y coordinates, but one linestring contains varying Z coordinates that cause the linestring to not be closed, and the other linestring contains varying M coordinates (measures) that do not affect whether the linestring is closed.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines (id INTEGER, geometry ST_Linestring)

INSERT INTO sample_lines VALUES
    (1, ST_Linestring('linestring EMPTY',0))

INSERT INTO sample_lines VALUES
    (2, ST_Linestring('linestring(10 10, 20 10, 20 20)' ,0))

INSERT INTO sample_lines VALUES
    (3, ST_Linestring('linestring(10 10, 20 10, 20 20, 10 10)' ,0))

INSERT INTO sample_lines VALUES
    (4, ST_Linestring('linestring m(10 10 1, 20 10 2, 20 20 3,
    10 10 4)' ,0))

INSERT INTO sample_lines VALUES
    (5, ST_Linestring('linestring z(10 10 5, 20 10 6, 20 20 7,
    10 10 8)' ,0))

SELECT id, ST_IsClosed(geometry) Is_Closed
FROM sample_lines
```

Results:

ID	IS_CLOSED
1	0
2	0
3	1
4	1
5	0

Example 2

In this example, two multilinestrings are created. ST_IsClosed is used to determine if the multilinestrings are closed. The first one is not closed, even though all of the curves together form a complete closed loop. This is because each curve itself is not closed.

The second multilinestring is closed because each curve itself is closed.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mlines (id INTEGER, geometry ST_MultiLinestring)
INSERT INTO sample_mlines
VALUES
(6, ST_MultiLinestring('multilinestring((10 10, 20 10, 20 20),
                                   (20 20, 30 20, 30 30),
                                   (30 30, 10 30, 10 10))',0))

INSERT INTO sample_mlines
VALUES
(7,ST_MultiLinestring('multilinestring((10 10, 20 10, 20 20, 10 10),
                                   (30 30, 50 30, 50 50, 30 30))',0))

SELECT id, ST_IsClosed(geometry) Is_Closed
FROM sample_mlines
```

Results:

ID	IS_CLOSED
6	0
7	1

ST_IsEmpty function

The ST_IsEmpty function takes a geometry as an input parameter and returns 1 if the given geometry is empty. Otherwise 0 (zero) is returned.

If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_IsEmpty—(*—geometry—*)—►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry that is to be tested.

Return type

INTEGER

Example

The following code creates three geometries and then determines if they are empty.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
  (1, ST_Geometry('point EMPTY',0))

INSERT INTO sample_geoms VALUES
  (2, ST_Geometry('polygon((40 120, 90 120, 90 150, 40 150, 40 120))' ,0))

INSERT INTO sample_geoms VALUES
  (3, ST_Geometry('multipoint m (10 10 5, 50 10 6, 10 30 8)' ,0))

INSERT INTO sample_geoms VALUES
  (4, ST_Geometry('linestring z (10 10 166, 20 10 168)',0))

INSERT INTO sample_geoms VALUES
  (5, ST_Geometry('point zm (10 10 16 30)' ,0))

SELECT id, ST_IsEmpty(geometry) Is_Empty
FROM sample_geoms
```

Results:

ID	IS_EMPTY
1	1
2	0
3	0
4	0
5	0

ST_IsMeasured function

The ST_IsMeasured function takes a geometry as an input parameter. If the given geometry has M coordinates (measures), it returns 1. Otherwise, it returns 0 (zero).

If the given geometry is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_IsMeasured—(—*geometry*—)—————◄◄

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry to be tested for the existence of M coordinates (measures).

Return type

INTEGER

Example

In this example, several geometries are created with and without Z coordinates and M coordinates (measures). ST_IsMeasured is then used to determine which of them contained measures.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
    (1, ST_Geometry('point EMPTY',0))

INSERT INTO sample_geoms VALUES
    (2, ST_Geometry('polygon((40 120, 90 120, 90 150, 40 150, 40 120))' ,0))

INSERT INTO sample_geoms VALUES
    (3, ST_Geometry('multipoint m (10 10 5, 50 10 6, 10 30 8)' ,0))

INSERT INTO sample_geoms VALUES
    (4, ST_Geometry('linestring z (10 10 166, 20 10 168)',0))

INSERT INTO sample_geoms VALUES
    (5, ST_Geometry('point zm (10 10 16 30)' ,0))

SELECT id, ST_IsMeasured(geometry) Is_Measured
FROM sample_geoms
```

Results:

ID	IS_MEASURED
1	0
2	0
3	1
4	0
5	1

ST_IsRing function

The ST_IsRing function takes a curve as an input parameter and returns 1 if it is a ring. Otherwise, 0 (zero) is returned. A curve is a ring if it is simple and closed.

If the given curve is empty, then 0 (zero) is returned. If it is null, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_IsRing—(—*curve*—)————►►

Parameter

curve A value of type ST_Curve or one of its subtypes that represents the curve to be tested.

Return type

INTEGER

Examples

In this example, four linestrings are created. ST_IsRing is used to check if they are rings. The last one is not considered a ring even though it is closed because the path crosses over itself.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines (id INTEGER, geometry ST_Linestring)

INSERT INTO sample_lines VALUES
  (1, ST_Linestring('linestring EMPTY',0))

INSERT INTO sample_lines VALUES
  (2, ST_Linestring('linestring(10 10, 20 10, 20 20)' ,0))

INSERT INTO sample_lines VALUES
  (3, ST_Linestring('linestring(10 10, 20 10, 20 20, 10 10)' ,0))

INSERT INTO sample_lines VALUES
  (4, ST_Linestring('linestring(10 10, 20 10, 10 20, 20 20, 10 10)' ,0))

SELECT id, ST_IsClosed(geometry) Is_Closed, ST_IsRing(geometry) Is_Ring
FROM sample_lines
```

Results:

ID	IS_CLOSED	IS_RING
1	1	0
2	0	0
3	1	1
4	1	0

ST_IsSimple function

The ST_IsSimple function takes a geometry as an input parameter and returns 1 if the given geometry is simple. Otherwise, 0 (zero) is returned.

Points, surfaces, and multisurfaces are always simple. A curve is simple if it does not pass through the same point twice; a multipoint is simple if it does not contain two equal points; and a multicurve is simple if all of its curves are simple and the only intersections occur at points that are on the boundary of the curves in the multicurve.

If the given geometry is empty, then 1 is returned. If it is null, null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_IsSimple—(—*geometry*—)—————►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry to be tested.

Return type

INTEGER

Examples

In this example, several geometries are created and checked if they are simple. The geometry with an ID of 4 is not considered simple because it contains more than one point that is the same. The geometry with an ID of 6 is not considered simple because the linestring crosses over itself.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
    (1, ST_Geometry('point EMPTY' ,0))

INSERT INTO sample_geoms VALUES
    (2, ST_Geometry('point (21 33)' ,0))

INSERT INTO sample_geoms VALUES
    (3, ST_Geometry('multipoint(10 10, 20 20, 30 30)' ,0))

INSERT INTO sample_geoms VALUES
    (4, ST_Geometry('multipoint(10 10, 20 20, 30 30, 20 20)' ,0))

INSERT INTO sample_geoms VALUES
    (5, ST_Geometry('linestring(60 60, 70 60, 70 70)' ,0))

INSERT INTO sample_geoms VALUES
    (6, ST_Geometry('linestring(20 20, 30 30, 30 20, 20 30 )' ,0))

INSERT INTO sample_geoms VALUES
    (7, ST_Geometry('polygon((40 40, 50 40, 50 50, 40 40 ))' ,0))

SELECT id, ST_IsSimple(geometry) Is_Simple
FROM sample_geoms
```

Results:

ID	IS_SIMPLE
-----	-----
1	1
2	1
3	1
4	0
5	1
6	0
7	1

ST_IsValid function

The ST_IsValid function takes a geometry as an input parameter and returns 1 if it is valid. Otherwise 0 (zero) is returned.

A geometry is valid only if all of the attributes in the structured type are consistent with the internal representation of geometry data, and if the internal representation is not corrupted.

If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►—db2gse.ST_IsValid—(*—geometry—*)—◄◄

Parameter

geometry

A value of type ST_Geometry or one of its subtypes.

Return type

INTEGER

Example

This example creates several geometries and uses ST_IsValid to check if they are valid. All of the geometries are valid because the constructor routines, such as ST_Geometry, do not allow invalid geometries to be constructed.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms VALUES
    (1, ST_Geometry('point EMPTY',0))

INSERT INTO sample_geoms VALUES
    (2, ST_Geometry('polygon((40 120, 90 120, 90 150, 40 150, 40 120))' ,0))

INSERT INTO sample_geoms VALUES
    (3, ST_Geometry('multipoint m (10 10 5, 50 10 6, 10 30 8)' ,0))

INSERT INTO sample_geoms VALUES
    (4, ST_Geometry('linestring z (10 10 166, 20 10 168)',0))

INSERT INTO sample_geoms VALUES
    (5, ST_Geometry('point zm (10 10 16 30)' ,0))

SELECT id, ST_IsValid(geometry) Is_Valid
FROM sample_geoms
```

Results:

ID	IS_VALID
1	1
2	1
3	1
4	1
5	1

ST_Length function

The ST_Length function takes a curve or multcurve and, optionally, a unit as input parameters and returns the length of the given curve or multcurve in the default or given unit of measure.

If the given curve or multcurve is null or is empty, null is returned.

This function can also be called as a method.

Syntax

►► db2gse.ST_Length—(—*curve*—,—*unit*—)—►►

Parameter

- curve** A value of type ST_Curve or ST_MultiCurve that represents the curves for which the length is returned.
- unit** A VARCHAR(128) value that identifies the units in which the length of the curve is measured. The supported units of measure are listed in the DB2GSE.ST_UNITS_OF_MEASURE catalog view.

If the *unit* parameter is omitted, the following rules are used to determine the unit in which the length is measured:

- If *curve* is in a projected or geocentric coordinate system, the linear unit associated with this coordinate system is the default.
- If *curve* is in a geographic coordinate system, the angular unit associated with this coordinate system is the default.

Restrictions on unit conversions: An error (SQLSTATE 38SU4) is returned if any of the following conditions occur:

- The *curve* is in an unspecified coordinate system and the *unit* parameter is specified.
- The *curve* is in a projected coordinate system and an angular unit is specified.
- The *curve* is in a geographic coordinate system, and a linear unit is specified.

Return type

DOUBLE

Examples

Example 1

The following SQL statements create a table SAMPLE_GEOMETRIES and insert a line and a multiline into the table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_geometries(id SMALLINT, spatial_type varchar(20),  
    geometry ST_GEOMETRY)
```

```
INSERT INTO sample_geometries(id, spatial_type, geometry)  
VALUES  
    (1110, 'ST_LineString', ST_LineString('linestring(50 10, 50 20)', 1)),  
    (1111, 'ST_MultiLineString', ST_MultiLineString('multilinestring  
        ((33 2, 34 3, 35 6),  
        (28 4, 29 5, 31 8, 43 12),  
        (39 3, 37 4, 36 7))', 1))
```

Example 2

The following SELECT statement calculates the length of the line in the SAMPLE_GEOMTRIES table.

```
SELECT id, spatial_type, cast(ST_Length(geometry..ST_ToLineString)  
    AS DECIMAL(7, 2)) AS "Line Length"  
FROM sample_geometries  
WHERE id = 1110
```


ID	SPATIAL_TYPE	Line Length
1110	ST LineString	10.00

The following SELECT statement calculates the length of the multiline in the SAMPLE_GEOMETRIES table.

Results:

ID	SPATIAL_TYPE	MULTILINE_LENGTH
1111	ST MultiLineString	+2.76437123387202E+001

The ST_LineFromText function takes a well-known text representation of a linestring and, optionally, a spatial reference system identifier as input parameters and returns the corresponding linestring.

The preferred version for this functionality is `ST_LineString`.

```
db2gse.ST_LineFromText(-(wkt  
                        .-srs id))
```

wkt	A value of type CLOB(2G) that contains the well-known text representation of the resulting linestring.
------------	--

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting linestring.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an error is returned (SQLSTATE 38SU1).

db2gse.ST_LineString

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The following code uses the ST_LineFromText function to create and insert a line from a well-known text (WKT) line representation. The rows are inserted into the SAMPLE_LINES table with an ID and a line value in spatial reference system 1 in WKT representation.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sample_lines(id SMALLINT, geometry ST_LineString)

INSERT INTO sample_lines(id, geometry)
VALUES
    (1110, ST_LineFromText('linestring(850 250, 850 850)', 1) ),
    (1111, ST_LineFromText('linestring empty', 1) )

SELECT id, cast(geometry..ST_AsText AS varchar(75)) AS linestring
FROM   sample_lines
```

Results:

ID	LINESTRING
1110	LINESTRING (850.000000000 250.000000000, 850.000000000 850.000000000)
1111	LINESTRING EMPTY

ST_LineFromWKB function

The ST_LineFromWKB function takes a well-known binary representation of a linestring and, optionally, a spatial reference system identifier as input parameters and returns the corresponding linestring.

If the given well-known binary representation is null, then null is returned.

The preferred version for this functionality is ST_LineString.

Syntax

►►db2gse.ST_LineFromWKB(—wkb—, —srs_id—)◄◄

Parameter

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting linestring.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting linestring.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an error is returned (SQLSTATE 38SU1).

Return type

db2gse.ST_LineString

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The following code uses the ST_LineFromWKB function to create and insert a line from a well-known binary representation. The row is inserted into the SAMPLE_LINES table with an ID and a line in spatial reference system 1 in WKB representation.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE TABLE sample_lines(id SMALLINT, geometry ST_LineString, wkb BLOB(32k))

INSERT INTO sample_lines(id, geometry)
VALUES
  (1901, ST_LineString('linestring(850 250, 850 850)', 1) ),
  (1902, ST_LineString('linestring(33 2, 34 3, 35 6)', 1) )

UPDATE sample_lines AS temp_correlated
SET    wkb = geometry..ST_AsBinary
WHERE  id = temp_correlated.id

SELECT id, cast(ST_LineFromWKB(wkb)..ST_AsText AS varchar(90)) AS line
FROM    sample_lines
```

Results:

ID	LINE
1901	LINESTRING (850.00000000 250.00000000, 850.00000000 850.00000000)
1902	LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000)

ST_LineString function

The ST_LineString function constructs a linestring from given input.

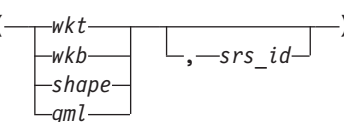
The inputs can be given in one of the following formats:

- A well-known text representation
- A well-known binary representation
- An ESRI shape representation
- A representation in the geography markup language (GML)

A spatial reference system identifier can be provided optionally to identify the spatial reference system that the resulting linestring is in.

If the well-known text representation, the well-known binary representation, the ESRI shape representation, or the GML representation is null, then null is returned.

Syntax

►►db2gse.ST_LineString—() ►►

Parameter

- wkt** A value of type CLOB(2G) that contains the well-known text representation of the resulting polygon.
- wkb** A value of type BLOB(2G) that contains the well-known binary representation of the resulting polygon.
- shape** A value of type BLOB(2G) that represents the ESRI shape representation of the resulting polygon.
- gml** A value of type CLOB(2G) that represents the resulting polygon using the geography markup language (GML).
- srs_id** A value of type INTEGER that identifies the spatial reference system for the resulting polygon.
- If the *srs_id* parameter is omitted, then the spatial reference system with the numeric identifier 0 (zero) is used.
- If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an error is returned (SQLSTATE 38SU1).

Return type

db2gse.ST_LineString

Examples

The following code uses the ST_LineString function to create and insert a line from a well-known text (WKT) line representation or from a well-known binary (WKB) representation.

The following example inserts a row into the SAMPLE_LINES table with an ID and line in spatial reference system 1 in WKT and GML representation

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_lines(id SMALLINT, geometry ST_LineString)
```

```
INSERT INTO sample_lines(id, geometry)
VALUES
```

```
  (1110, ST_LineString('linestring(850 250, 850 850)', 1) ),
  (1111, ST_LineString('<gml:LineString srsName=";EPSG:4269";><gml:coord>
    <gml:X>90</gml:X><gml:Y>90</gml:Y>
    </gml:coord><gml:coord><gml:X>100</gml:X>
    <gml:Y>100</gml:Y></gml:coord>
    </gml:LineString>', 1) )
```

```
SELECT id, cast(geometry..ST_AsText AS varchar(75)) AS linestring
FROM   sample_lines
```

Results:

```
ID      LINESTRING
```

```
-----
1110 LINESTRING ( 850.00000000 250.00000000, 850.00000000 850.00000000)
1111 LINESTRING ( 90.00000000 90.00000000, 100.00000000 100.00000000)
```

ST_LineStringN function

The ST_LineStringN function takes a multilinestring and an index as input parameters and returns the linestring that is identified by the index. The resulting linestring is represented in the spatial reference system of the given multilinestring.

If the given multilinestring is null or is empty, or if the index is smaller than 1 or larger than the number of linestrings, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_LineStringN—(—*multi_linestring*—,—*index*—)—►►

Parameter

multi_linestring

A value of type ST_MultiLineString that represents the multilinestring from which the linestring that is identified by *index* is returned.

index A value of type INTEGER that identifies the *n*th linestring, which is to be returned from *multi_linestring*.

If *index* is smaller than 1 or larger than the number of linestrings in *multi_linestring*, then null is returned and a warning condition is returned (SQLSTATE 01HS0).

Return type

db2gse.ST_LineString

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The SELECT statement illustrates how to choose the second geometry inside a multilinestring in the SAMPLE_MLINES table.

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

```
CREATE TABLE sample_mlines (id INTEGER,
  geometry ST_MULTILINESTRING)
```

```
INSERT INTO sample_mlines(id, geometry)
VALUES
```

```
  (1110, ST_MultiLineString('multilinestring
                             ((33 2, 34 3, 35 6),
                             (28 4, 29 5, 31 8, 43 12),
                             (39 3, 37 4, 36 7))', 1) ),
```

```
  (1111, ST_MLineFromText('multilinestring(
                             (61 2, 64 3, 65 6),
                             (58 4, 59 5, 61 8),
                             (69 3, 67 4, 66 7, 68 9))', 1) )
```

```
SELECT id, cast(ST_LineStringN(geometry, 2)..ST_AsText
  AS varchar(110)) AS second_linestring
FROM   sample_mlines
```

Results:

ID	SECOND_LINESTRING
1110	LINESTRING (28.00000000 4.00000000, 29.00000000 5.00000000, 31.00000000 8.00000000, 43.00000000 12.00000000)
1111	LINESTRING (58.00000000 4.00000000, 59.00000000 5.00000000, 61.00000000 8.00000000)

ST_M function function

The ST_M function takes a point as an input parameter and returns its measure (M) coordinate. You can optionally indicate an M coordinate as input parameter in addition to the point and the function returns the point itself with its M coordinate set to the given value.

If the specified M coordinate is null, then the M coordinate of the point is removed.

If the specified point is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►►db2gse.ST_M(*point* , *m_coordinate*)►►

Parameters

point A value of type ST_Point for which the M coordinate is returned or modified.

m_coordinate

A value of type DOUBLE that represents the new M coordinate for *point*.

If *m_coordinate* is null, then the M coordinate is removed from *point*.

Return types

- DOUBLE, if *m_coordinate* is not specified
- db2gse.ST_Point, if *m_coordinate* is specified

Examples

Example 1

This example illustrates the use of the ST_M function. Three points are created and inserted into the SAMPLE_POINTS table. They are all in the spatial reference system that has an ID of 1.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)
```

```
INSERT INTO sample_points
VALUES (1, ST_Point (2, 3, 32, 5, 1))
```

```
INSERT INTO sample_points
VALUES (2, ST_Point (4, 5, 20, 4, 1))
```

```
INSERT INTO sample_points
VALUES (3, ST_Point (3, 8, 23, 7, 1))
```

Example 2

This example finds the M coordinate of the points in the SAMPLE_POINTS table.

```
SELECT id, ST_M (geometry) M_COORD
FROM sample_points
```

Results:

ID	M_COORD
1	+5.000000000000000E+000
2	+4.000000000000000E+000
3	+7.000000000000000E+000

Example 3

This example returns one of the points with its M coordinate set to 40.

```
SELECT id, CAST (ST_AsText (ST_M (geometry, 40) )
AS VARCHAR(60) ) M_COORD_40
FROM sample_points
WHERE id=3
```

Results:

ID	M_COORD_40
3	POINT ZM (3.00000000 8.00000000 23.00000000 40.00000000)

ST_MaxM function

The ST_MaxM function takes a geometry as an input parameter and returns its maximum M coordinate.

If the given geometry is null or is empty, or if it does not have M coordinates, then null is returned.

This function can also be called as a method.

Syntax

```
►►—db2gse.ST_MaxM—(—geometry—)—————►◄
```

Parameter

geometry

A value of type ST_Geometry or one of its subtypes for which the maximum M coordinate is returned.

Return type

DOUBLE

Examples

Example 1

This example illustrates the use of the ST_MaxM function. Three polygons are created and inserted into the SAMPLE_POLYS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)

INSERT INTO sample_polys
VALUES (1, ST_Polygon('polygon zm ((110 120 20 3,
                                     110 140 22 3,
                                     120 130 26 4,
                                     110 120 20 3))', 0) )

INSERT INTO sample_polys
VALUES (2, ST_Polygon('polygon zm ((0 0 40 7,
                                     0 4 35 9,
                                     5 4 32 12,
                                     5 0 31 5,
                                     0 0 40 7))', 0) )

INSERT INTO sample_polys
VALUES (3, ST_Polygon('polygon zm ((12 13 10 16,
                                     8 4 10 12,
                                     9 4 12 11,
                                     12 13 10 16))', 0) )
```

Example 2

This example finds the maximum M coordinate of each polygon in SAMPLE_POLYS.

```
SELECT id, CAST ( ST_MaxM(geometry) AS INTEGER) MAX_M
FROM sample_polys
```

Results:

ID	MAX_M
1	4
2	12
3	16

Example 3

This example finds the maximum M coordinate that exists for all polygons in the GEOMETRY column.

```
SELECT CAST ( MAX ( ST_MaxM(geometry) ) AS INTEGER) OVERALL_MAX_M
FROM sample_polys
```

Results:

OVERALL_MAX_M
16

ST_MaxX function

The ST_MaxX function takes a geometry as an input parameter and returns its maximum X coordinate.

If the given geometry is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_MaxX—(—*geometry*—)————►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes for which the maximum X coordinate is returned.

Return type

DOUBLE

Examples

Example 1

This example illustrates the use of the ST_MaxX function. Three polygons are created and inserted into the SAMPLE_POLYS table. The third example illustrates how you can use all of the functions that return the maximum and minimum coordinate values to assess the spatial range of the geometries that are stored in a particular spatial column.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)
```

```
INSERT INTO sample_polys
VALUES (1, ST_Polygon('polygon zm ((110 120 20 3,
                                110 140 22 3,
                                120 130 26 4,
                                110 120 20 3))', 0) )
```

```
INSERT INTO sample_polys
VALUES (2, ST_Polygon('polygon zm ((0 0 40 7,
                                0 4 35 9,
                                5 4 32 12,
                                5 0 31 5,
                                0 0 40 7))', 0) )
```

```
INSERT INTO sample_polys
VALUES (3, ST_Polygon('polygon zm ((12 13 10 16,
                                8 4 10 12,
                                9 4 12 11,
                                12 13 10 16))', 0) )
```

Example 2

This example finds the maximum X coordinate of each polygon in SAMPLE_POLYS.

```
SELECT id, CAST ( ST_MaxX(geometry) AS INTEGER) MAX_X_COORD
FROM sample_polys
```

Results:

ID	MAX_X_COORD
1	120
2	5
3	12

Example 3

This example finds the maximum X coordinate that exists for all polygons in the GEOMETRY column.

```
SELECT CAST ( MAX ( ST_MaxX(geometry) ) AS INTEGER) OVERALL_MAX_X
FROM sample_polys
```

Results:

```
OVERALL_MAX_X
-----
120
```

Example 4

This example finds the spatial extent (overall minimum to overall maximum) of all the polygons in the SAMPLE_POLYS table. This calculation is typically used to compare the actual spatial extent of the geometries to the spatial extent of the spatial reference system associated with the data to determine if the data has room to grow.

```
SELECT CAST ( MIN (ST_MinX (geometry)) AS INTEGER) MIN_X,
       CAST ( MIN (ST_MinY (geometry)) AS INTEGER) MIN_Y,
       CAST ( MIN (ST_MinZ (geometry)) AS INTEGER) MIN_Z,
       CAST ( MIN (ST_MinM (geometry)) AS INTEGER) MIN_M,
       CAST ( MAX (ST_MaxX (geometry)) AS INTEGER) MAX_X,
       CAST ( MAX (ST_MaxY (geometry)) AS INTEGER) MAX_Y,
       CAST ( MAX (ST_MaxZ (geometry)) AS INTEGER) MAX_Z,
       CAST ( MAX (ST_MaxmM(geometry)) AS INTEGER) MAX_M,
FROM sample_polys
```

Results:

```
MIN_X  MIN_Y  MIN_Z  MIN_M  MAX_X  MAX_Y  MAX_Z  MAX_M
-----
0      0      10     3     120   140   40    16
```

ST_MaxY function

The ST_MaxY function takes a geometry as an input parameter and returns its maximum Y coordinate.

If the given geometry is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

```
►►db2gse.ST_MaxY(—geometry—)◄◄
```

Parameter

geometry

A value of type ST_Geometry or one of its subtypes for which the maximum Y coordinate is returned.

Return type

DOUBLE

Examples

Example 1

This example illustrates the use of the ST_MaxY function. Three polygons are created and inserted into the SAMPLE_POLYS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)
```

```
INSERT INTO sample_polys
VALUES (1, ST_Polygon('polygon zm ((110 120 20 3,
                                     110 140 22 3,
                                     120 130 26 4,
                                     110 120 20 3))', 0) )
```

```
INSERT INTO sample_polys
VALUES (2, ST_Polygon('polygon zm ((0 0 40 7,
                                     0 4 35 9,
                                     5 4 32 12,
                                     5 0 31 5,
                                     0 0 40 7))', 0) )
```

```
INSERT INTO sample_polys
VALUES (3, ST_Polygon('polygon zm ((12 13 10 16,
                                     8 4 10 12,
                                     9 4 12 11,
                                     12 13 10 16))', 0) )
```

Example 2

This example finds the maximum Y coordinate of each polygon in SAMPLE_POLYS.

```
SELECT id, CAST ( ST_MaxY(geometry) AS INTEGER) MAX_Y
FROM sample_polys
```

Results:

ID	MAX_Y
1	140
2	4
3	13

Example 3

This example finds the maximum Y coordinate that exists for all polygons in the GEOMETRY column.

```
SELECT CAST ( MAX ( ST_MaxY(geometry) ) AS INTEGER) OVERALL_MAX_Y
FROM sample_polys
```

Results:

OVERALL_MAX_Y
140

ST_MaxZ function

The ST_MaxZ function takes a geometry as an input parameter and returns its maximum Z coordinate.

If the given geometry is null or is empty, or if it does not have Z coordinates, then null is returned.

This function can also be called as a method.

Syntax

►► db2gse.ST_MaxZ(—*geometry*—) ◀◀

Parameter

geometry

A value of type ST_Geometry or one of its subtypes for which the maximum Z coordinate is returned.

Return type

DOUBLE

Examples

Example 1

This example illustrates the use of the ST_MaxZ function. Three polygons are created and inserted into the SAMPLE_POLYS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)
```

```
INSERT INTO sample_polys
VALUES (1, ST_Polygon('polygon zm ((110 120 20 3,
                                110 140 22 3,
                                120 130 26 4,
                                110 120 20 3))', 0) )
```

```
INSERT INTO sample_polys
VALUES (2, ST_Polygon('polygon zm ((0 0 40 7,
                                0 4 35 9,
                                5 4 32 12,
                                5 0 31 5,
                                0 0 40 7))', 0) )
```

```
INSERT INTO sample_polys
VALUES (3, ST_Polygon('polygon zm ((12 13 10 16,
                                8 4 10 12,
                                9 4 12 11,
                                12 13 10 16))', 0) )
```

Example 2

This example finds the maximum Z coordinate of each polygon in SAMPLE_POLYS.

```
SELECT id, CAST ( ST_MaxZ(geometry) AS INTEGER) MAX_Z
FROM sample_polys
```

Results:

ID	MAX_Z
1	26
2	40
3	12

Example 3

This example finds the maximum Z coordinate that exists for all polygons in the GEOMETRY column.

```
SELECT CAST ( MAX ( ST_MaxZ(geometry) ) AS INTEGER) OVERALL_MAX_Z
FROM sample_polys
```

```
Results:
OVERALL_MAX_Z
-----
40
```

ST_MBR function

The ST_MBR function takes a geometry as an input parameter and returns its minimum bounding rectangle.

If the given geometry is a point, then the point itself is returned. If the geometry is a horizontal linestring or a vertical linestring, the horizontal or vertical linestring itself is returned. If the given geometry is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

```
►► db2gse.ST_MBR(—geometry—)◄◄
```

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry for which the minimum bounding rectangle is returned.

Return type

db2gse.ST_Geometry

Example

This example illustrates how the ST_MBR function can be used to return the minimum bounding rectangle of a polygon. Because the specified geometry is a polygon, the minimum bounding rectangle is returned as a polygon.

In the following examples, the lines of results have been reformatted here for readability. The spacing in your results will vary according to your online display.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)

INSERT INTO sample_polys
VALUES (1, ST_Polygon ('polygon (( 5 5, 7 7, 5 9, 7 9, 9 11, 13 9,
                             15 9, 13 7, 15 5, 9 6, 5 5))', 0) )

INSERT INTO sample_polys
VALUES (2, ST_Polygon ('polygon (( 20 30, 25 35, 30 30, 20 30))', 0) )

SELECT id, CAST (ST_AsText ( ST_MBR(geometry)) AS VARCHAR(150) ) MBR
FROM sample_polys
```

Results:

```
ID          MBR
-----
1 POLYGON (( 5.00000000 5.00000000, 15.00000000 5.00000000,
              15.00000000 11.00000000, 5.00000000 11.00000000,
```

```

5.00000000 5.00000000))
2 POLYGON (( 20.00000000 30.00000000, 30.00000000 30.00000000,
30.00000000 35.00000000, 20.00000000 35.00000000,
20.00000000 30.00000000 ))

```

ST_MBRIntersects function

The ST_MBRIntersects function takes two geometries as input parameters and returns 1 if the minimum bounding rectangles of the two geometries intersect. Otherwise, 0 (zero) is returned. The minimum bounding rectangle of a point and a horizontal or vertical linestring is the geometry itself.

If the second geometry is not represented in the same spatial reference system as the first geometry, it will be converted to the other spatial reference system.

If either of the given geometries is null or is empty, then null is returned.

Syntax

►►db2gse.ST_MBRIntersects(—*geometry1*—,—*geometry2*—)◄◄

Parameters

geometry1

A value of type ST_Geometry or one of its subtypes that represents the geometry whose minimum bounding rectangle is to be tested for intersection with the minimum bounding rectangle of *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the geometry whose minimum bounding rectangle is to be tested for intersection with the minimum bounding rectangle of *geometry1*.

Return type

INTEGER

Examples

Example 1

This example illustrates the use of ST_MBRIntersects to get an approximation of whether two nonintersecting polygons are close to each other by seeing if their minimum bounding rectangles intersect. The first example uses the SQL CASE expression. The second example uses a single SELECT statement to find those polygons that intersect the minimum bounding rectangle of the polygon with ID = 2.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)

```

```

INSERT INTO sample_polys
VALUES (1, ST_Polygon ('polygon (( 0 0, 30 0, 40 30, 40 35,
5 35, 5 10, 20 10, 20 5, 0 0 ))', 0) )

```

```

INSERT INTO sample_polys
VALUES (2, ST_Polygon ('polygon (( 15 15, 15 20, 60 20, 60 15,
15 15 ))', 0) )

```

```
INSERT INTO sample_polys
VALUES (3, ST_Polygon ('polygon (( 115 15, 115 20, 160 20, 160 15,
115 15 ))', 0) )
```

Example 2

The following SELECT statement uses a CASE expression to find the IDs of the polygons that have minimum bounding rectangles that intersect.

```
SELECT a.id, b.id,
CASE ST_MBRIntersects (a.geometry, b.geometry)
WHEN 0 THEN 'MBRs do not intersect'
WHEN 1 THEN 'MBRs intersect'
END AS MBR_INTERSECTS
FROM sample_polys a, sample_polys b
WHERE a.id <= b.id
```

Results:

ID	ID	MBR_INTERSECTS
1	1	1 MBRs intersect
1	2	2 MBRs intersect
2	2	2 MBRs intersect
1	3	3 MBRs do not intersect
2	3	3 MBRs do not intersect
3	3	3 MBRs intersect

Example 3

The following SELECT statement determines whether the minimum bounding rectangles for the geometries intersect that for the polygon with ID = 2.

```
SELECT a.id, b.id, ST_MBRIntersects (a.geometry, b.geometry) MBR_INTERSECTS
FROM sample_polys a, sample_polys b
WHERE a.id = 2
```

Results

ID	ID	MBR_INTERSECTS
2	1	1
2	2	1
2	3	0

ST_LocateBetween or ST_MeasureBetween function

The ST_LocateBetween or ST_MeasureBetween function take a geometry and two M coordinates (measures) as input parameters and returns that part of the given geometry that represents the set of disconnected paths or points between the two M coordinates.

For curves, multicurves, surfaces, and multisurfaces, interpolation is performed to compute the result. The resulting geometry is represented in the spatial reference system of the given geometry.

If the given geometry is a surface or multisurface, then ST_MeasureBetween or ST_LocateBetween will be applied to the exterior and interior rings of the geometry. If none of the parts of the given geometry are in the interval defined by the given M coordinates, then an empty geometry is returned. If the given geometry is null , then null is returned.

If the resulting geometry is not empty, a multipoint or multilinestring type is returned.

Both functions can also be called as methods.

Syntax

```
db2gse.ST_MeasureBetween
db2gse.ST_LocateBetween

(—geometry—,—startMeasure—,—endMeasure—)
```

Parameters

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry in which those parts with measure values between *startMeasure* to *endMeasure* are to be found.

startMeasure

A value of type DOUBLE that represents the lower bound of the measure interval. If this value is null, no lower bound is applied.

endMeasure

A value of type DOUBLE that represents the upper bound of the measure interval. If this value is null, no upper bound is applied.

Return type

db2gse.ST_Geometry

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The M coordinate (measure) of a geometry is defined by the user. It is very versatile because it can represent anything that you want to measure; for example, distance along a highway, temperature, pressure, or pH measurements.

This example illustrates the use of the M coordinate to record collected data of pH measurements. A researcher collects the pH of the soil along a highway at specific places. Following his standard operating procedures, he writes down the values that he needs at every place at which he takes a soil sample: the X and Y coordinates of that place and the pH that he measures.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines (id INTEGER, geometry ST_LineString)
```

```
INSERT INTO sample_lines
VALUES (1, ST_LineString ('linestring m (2 2 3, 3 5 3,
                                     3 3 6, 4 4 6,
                                     5 5 6, 6 6 8)', 1 ) )
```

To find the path where the acidity of the soil varies between 4 and 6, the researcher would use this SELECT statement:

```
SELECT id, CAST( ST_AsText( ST_MeasureBetween( 4, 6) )
AS VARCHAR(150) ) MEAS_BETWEEN_4_AND_6
FROM sample_lines
```

Results:

ID	MEAS_BETWEEN_4_AND_6
1	LINESTRING M (3.00000000 4.33333300 4.00000000, 3.00000000 3.00000000 6.00000000, 4.00000000 4.00000000 6.00000000, 5.00000000 5.00000000 6.00000000)

ST_LocateBetween or ST_MeasureBetween function

The ST_LocateBetween or ST_MeasureBetween function take a geometry and two M coordinates (measures) as input parameters and returns that part of the given geometry that represents the set of disconnected paths or points between the two M coordinates.

For curves, multicurves, surfaces, and multisurfaces, interpolation is performed to compute the result. The resulting geometry is represented in the spatial reference system of the given geometry.

If the given geometry is a surface or multisurface, then ST_MeasureBetween or ST_LocateBetween will be applied to the exterior and interior rings of the geometry. If none of the parts of the given geometry are in the interval defined by the given M coordinates, then an empty geometry is returned. If the given geometry is null , then null is returned.

If the resulting geometry is not empty, a multipoint or multilinestring type is returned.

Both functions can also be called as methods.

Syntax

```

db2gse.ST_MeasureBetween
db2gse.ST_LocateBetween
(—geometry—,—startMeasure—,—endMeasure—)

```

Parameters

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry in which those parts with measure values between *startMeasure* to *endMeasure* are to be found.

startMeasure

A value of type DOUBLE that represents the lower bound of the measure interval. If this value is null, no lower bound is applied.

endMeasure

A value of type DOUBLE that represents the upper bound of the measure interval. If this value is null, no upper bound is applied.

Return type

db2gse.ST_Geometry

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

The M coordinate (measure) of a geometry is defined by the user. It is very versatile because it can represent anything that you want to measure; for example, distance along a highway, temperature, pressure, or pH measurements.

This example illustrates the use of the M coordinate to record collected data of pH measurements. A researcher collects the pH of the soil along a highway at specific places. Following his standard operating procedures, he writes down the values that he needs at every place at which he takes a soil sample: the X and Y coordinates of that place and the pH that he measures.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines (id INTEGER, geometry ST_LineString)
```

```
INSERT INTO sample_lines
VALUES (1, ST_LineString ('linestring m (2 2 3, 3 5 3,
                                     3 3 6, 4 4 6,
                                     5 5 6, 6 6 8)', 1 ) )
```

To find the path where the acidity of the soil varies between 4 and 6, the researcher would use this SELECT statement:

```
SELECT id, CAST( ST_AsText( ST_MeasureBetween( 4, 6) )
AS VARCHAR(150) ) MEAS_BETWEEN_4_AND_6
FROM sample_lines
```

Results:

ID	MEAS_BETWEEN_4_AND_6
1	LINESTRING M (3.00000000 4.33333300 4.00000000, 3.00000000 3.00000000 6.00000000, 4.00000000 4.00000000 6.00000000, 5.00000000 5.00000000 6.00000000)

ST_MidPoint function

The ST_MidPoint function takes a curve as an input parameter and returns the point on the curve that is equidistant from both end points of the curve, measured along the curve. The resulting point is represented in the spatial reference system of the given curve.

If the given curve is empty, then an empty point is returned. If the given curve is null, then null is returned.

If the curve contains Z coordinates or M coordinates (measures), the midpoint is determined solely by the values of the X and Y coordinates in the curve. The Z coordinate and measure in the returned point are interpolated.

This function can also be called as a method.

Syntax

►►—db2gse.ST_MidPoint—(—curve—)—————►►

Parameter

curve A value of type ST_Curve or one of its subtypes that represents the curve for which the point in the middle is returned.

Return type

db2gse.ST_Point

Example

This example illustrates the use of ST_MidPoint for returning the midpoint of curves.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines (id INTEGER, geometry ST_LineString)

INSERT INTO sample_lines (id, geometry)
VALUES (1, ST_LineString ('linestring (0 0, 0 10, 0 20, 0 30, 0 40)', 1 ) )

INSERT INTO sample_lines (id, geometry)
VALUES (2, ST_LineString ('linestring (2 2, 3 5, 3 3, 4 4, 5 5, 6 6)', 1 ) )

INSERT INTO sample_lines (id, geometry)
VALUES (3, ST_LineString ('linestring (0 10, 0 0, 10 0, 10 10)', 1 ) )

INSERT INTO sample_lines (id, geometry)
VALUES (4, ST_LineString ('linestring (0 20, 5 20, 10 20, 15 20)', 1 ) )

SELECT id, CAST( ST_AsText( ST_MidPoint(geometry) ) AS VARCHAR(60) ) MID_POINT
FROM sample_lines
```

Results:

ID	MID_POINT
1	POINT (0.00000000 20.00000000)
2	POINT (3.00000000 3.45981800)
3	POINT (5.00000000 0.00000000)
4	POINT (7.50000000 20.00000000)

ST_MinM function

The ST_MinM function takes a geometry as an input parameter and returns its minimum M coordinate.

If the given geometry is null or is empty, or if it does not have M coordinates, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_MinM—(—*geometry*—)—————►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes for which the minimum M coordinate is returned.

Return type

DOUBLE

Examples

Example 1

This example illustrates the use of the ST_MinM function. Three polygons are created and inserted into the SAMPLE_POLYS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)
```

```
INSERT INTO sample_polys
VALUES (1, ST_Polygon('polygon zm ((110 120 20 3,
                                   110 140 22 3,
                                   120 130 26 4,
                                   110 120 20 3))', 0) )
```

```
INSERT INTO sample_polys
VALUES (2, ST_Polygon('polygon zm ((0 0 40 7,
                                   0 4 35 9,
                                   5 4 32 12,
                                   5 0 31 5,
                                   0 0 40 7))', 0) )
```

```
INSERT INTO sample_polys
VALUES (3, ST_Polygon('polygon zm ((12 13 10 16,
                                   8 4 10 12,
                                   9 4 12 11,
                                   12 13 10 16))', 0) )
```

Example 2

This example finds the minimum M coordinate of each polygon in SAMPLE_POLYS.

```
SELECT id, CAST ( ST_MinM(geometry) AS INTEGER) MIN_M
FROM sample_polys
```

Results:

ID	MIN_M
1	3
2	5
3	11

Example 3

This example finds the minimum M coordinate that exists for all polygons in the GEOMETRY column.

```
SELECT CAST ( MIN ( ST_MinM(geometry) ) AS INTEGER) OVERALL_MIN_M
FROM sample_polys
```

Results:

OVERALL_MIN_M
3

ST_MinX function

The ST_MinX function takes a geometry as an input parameter and returns its minimum X coordinate.

If the given geometry is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_MinX—(—*geometry*—)—————►◄

Parameter

geometry

A value of type ST_Geometry or one of its subtypes for which the minimum X coordinate is returned.

Return type

DOUBLE

Examples

Example 1

This example illustrates the use of the ST_MinX function. Three polygons are created and inserted into the SAMPLE_POLYS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)
```

```
INSERT INTO sample_polys
VALUES (1, ST_Polygon('polygon zm ((110 120 20 3,
                                   110 140 22 3,
                                   120 130 26 4,
                                   110 120 20 3))', 0) )
```

```
INSERT INTO sample_polys
VALUES (2, ST_Polygon('polygon zm ((0 0 40 7,
                                   0 4 35 9,
                                   5 4 32 12,
                                   5 0 31 5,
                                   0 0 40 7))', 0) )
```

```
INSERT INTO sample_polys
VALUES (3, ST_Polygon('polygon zm ((12 13 10 16,
                                   8 4 10 12,
                                   9 4 12 11,
                                   12 13 10 16))', 0) )
```

Example 2

This example finds the minimum X coordinate of each polygon in SAMPLE_POLYS.

```
SELECT id, CAST ( ST_MinX(geometry) AS INTEGER) MIN_X
FROM sample_polys
```

Results:

ID	MIN_X
1	110
2	0
3	8

Example 3

This example finds the minimum X coordinate that exists for all polygons in the GEOMETRY column.

```
SELECT CAST ( MIN ( ST_MinX(geometry) ) AS INTEGER) OVERALL_MIN_X
FROM sample_polys
```

Results:

OVERALL_MIN_X
0

ST_MinY function

The ST_MinY function takes a geometry as an input parameter and returns its minimum Y coordinate.

If the given geometry is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_MinY—(—*geometry*—)—————►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes for which the minimum Y coordinate is returned.

Return type

DOUBLE

Examples

Example 1

This example illustrates the use of the ST_MinY function. Three polygons are created and inserted into the SAMPLE_POLYS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)
```

```
INSERT INTO sample_polys
VALUES (1, ST_Polygon('polygon zm ((110 120 20 3,
                                110 140 22 3,
                                120 130 26 4,
                                110 120 20 3))', 0) )
```

```
INSERT INTO sample_polys
VALUES (2, ST_Polygon('polygon zm ((0 0 40 7,
                                0 4 35 9,
                                5 4 32 12,
```

```

5 0 31 5,
0 0 40 7))', 0) )

INSERT INTO sample_polys
VALUES (3, ST_Polygon('polygon zm ((12 13 10 16,
8 4 10 12,
9 4 12 11,
12 13 10 16))', 0) )

```

Example 2

This example finds the minimum Y coordinate of each polygon in SAMPLE_POLYS.

```

SELECT id, CAST ( ST_MinY(geometry) AS INTEGER) MIN_Y
FROM sample_polys

```

Results:

ID	MIN_Y
1	120
2	0
3	4

Example 3

This example finds the minimum Y coordinate that exists for all polygons in the GEOMETRY column.

```

SELECT CAST ( MIN ( ST_MinY(geometry) ) AS INTEGER) OVERALL_MIN_Y
FROM sample_polys

```

Results:

OVERALL_MIN_Y
0

ST_MinZ function

The ST_MinZ function takes a geometry as an input parameter and returns its minimum Z coordinate.

If the given geometry is null or is empty, or if it does not have Z coordinates, then null is returned.

This function can also be called as a method.

Syntax

```

►►—db2gse.ST_MinZ—(—geometry—)—————►◄

```

Parameter

geometry

A value of type ST_Geometry or one of its subtypes for which the minimum Z coordinate is returned.

Return type

DOUBLE

Examples

Example 1

This example illustrates the use of the ST_MinZ function. Three polygons are created and inserted into the SAMPLE_POLYS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)
```

```
INSERT INTO sample_polys
VALUES (1, ST_Polygon('polygon zm ((110 120 20 3,
                                   110 140 22 3,
                                   120 130 26 4,
                                   110 120 20 3))', 0) )
```

```
INSERT INTO sample_polys
VALUES (2, ST_Polygon('polygon zm ((0 0 40 7,
                                   0 4 35 9,
                                   5 4 32 12,
                                   5 0 31 5,
                                   0 0 40 7))', 0) )
```

```
INSERT INTO sample_polys
VALUES (3, ST_Polygon('polygon zm ((12 13 10 16,
                                   8 4 10 12,
                                   9 4 12 11,
                                   12 13 10 16))', 0) )
```

Example 2

This example finds the minimum Z coordinate of each polygon in SAMPLE_POLYS.

```
SELECT id, CAST ( ST_MinZ(geometry) AS INTEGER) MIN_Z
FROM sample_polys
```

Results:

ID	MIN_Z
1	20
2	31
3	10

Example 3

This example finds the minimum Z coordinate that exists for all polygons in the GEOMETRY column.

```
SELECT CAST ( MIN ( ST_MinZ(geometry) ) AS INTEGER) OVERALL_MIN_Z
FROM sample_polys
```

Results:

OVERALL_MIN_Z
10

ST_MLineFromText function

The ST_MLineFromText function takes a well-known text representation of a multilinestring and, optionally, a spatial reference system identifier as input parameters and returns the corresponding multilinestring.

If the given well-known text representation is null, then null is returned.

The recommended function for achieving the same result is the `ST_MultiLineString` function. It is recommended because of its flexibility: `ST_MultiLineString` takes additional forms of input as well as the well-known text representation.

Syntax

```
►► db2gse.ST_MLineFromText ( ( wkt , srs_id ) ) ►►
```

Parameters

wkt A value of type `CLOB(2G)` that contains the well-known text representation of the resulting multilinestring.

srs_id A value of type `INTEGER` that identifies the spatial reference system for the resulting multilinestring.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If the specified *srs_id* does not identify a spatial reference system listed in the catalog view `DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS`, then an exception condition is raised (`SQLSTATE 38SU1`).

Return type

`db2gse.ST_MultiLineString`

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how `ST_MLineFromText` can be used to create and insert a multilinestring from its well-known text representation. The record that is inserted has `ID = 1110`, and the geometry is a multilinestring in spatial reference system 1. The multilinestring is in the well-known text representation of a multilinestring. The X and Y coordinates for this geometry are:

- Line 1: (33, 2) (34, 3) (35, 6)
- Line 2: (28, 4) (29, 5) (31, 8) (43, 12)
- Line 3: (39, 3) (37, 4) (36, 7)

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mlines (id INTEGER, geometry ST_MultiLineString)
```

```
INSERT INTO sample_mlines
VALUES (1110, ST_MLineFromText ('multilinestring ( (33 2, 34 3, 35 6),
                                     (28 4, 29 5, 31 8, 43 12),
                                     (39 3, 37 4, 36 7) )', 1) )
```

The following `SELECT` statement returns the multilinestring that was recorded in the table:

```
SELECT id, CAST( ST_AsText( geometry ) AS VARCHAR(280) ) MULTI_LINE_STRING
FROM sample_mlines
WHERE id = 1110
```

Results:

ID	MULTI_LINE_STRING
1110	MULTILINESTRING ((33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000), (28.00000000 4.00000000, 29.00000000 5.00000000, 31.00000000 8.00000000, 43.00000000 12.00000000), (39.00000000 3.00000000, 37.00000000 4.00000000, 36.00000000 7.00000000))

ST_MLineFromWKB function

The ST_MLineFromWKB function takes a well-known binary representation of a multilinestring and, optionally, a spatial reference system identifier as input parameters and returns the corresponding multilinestring.

If the given well-known binary representation is null, then null is returned.

The recommended function for achieving the same result is the ST_MultiLineString function. It is recommended because of its flexibility: ST_MultiLineString takes additional forms of input as well as the well-known binary representation.

Syntax

```
db2gse.ST_MLineFromWKB(—wkb—, —srs_id—)
```

Parameters

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting multilinestring.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting multilinestring.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If the specified *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_MultiLineString

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how ST_MLineFromWKB can be used to create a multilinestring from its well-known binary representation. The geometry is a multilinestring in spatial reference system 1. In this example, the multilinestring gets stored with ID = 10 in the GEOMETRY column of the SAMPLE_MLINES table, and then the WKB column is updated with its well-known binary representation (using the ST_AsBinary function). Finally, the ST_MLineFromWKB function is used to return the multilinestring from the WKB column. The X and Y coordinates for this geometry are:

- Line 1: (61, 2) (64, 3) (65, 6)
- Line 2: (58, 4) (59, 5) (61, 8)
- Line 3: (69, 3) (67, 4) (66, 7) (68, 9)

The SAMPLE_MLINES table has a GEOMETRY column, where the multilinestring is stored, and a WKB column, where the multilinestring's well-known binary representation is stored.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mlines (id INTEGER, geometry ST_MultiLineString,
                             wkb BLOB(32K))
```

```
INSERT INTO sample_mlines
VALUES (10, ST_MultiLineString ('multilinestring
( (61 2, 64 3, 65 6),
  (58 4, 59 5, 61 8),
  (69 3, 67 4, 66 7, 68 9) )', 1) )
```

```
UPDATE sample_mlines AS temporary_correlated
SET wkb = ST_AsBinary( geometry )
WHERE id = temporary_correlated.id
```

In the following SELECT statement, the ST_MLineFromWKB function is used to retrieve the multilinestring from the WKB column.

```
SELECT id, CAST( ST_AsText( ST_MLineFromWKB (wkb) )
AS VARCHAR(280) ) MULTI_LINE_STRING
FROM sample_mlines
WHERE id = 10
```

Results:

ID	MULTI_LINE_STRING
10	MULTILINESTRING ((61.00000000 2.00000000, 64.00000000 3.00000000, 65.00000000 6.00000000), (58.00000000 4.00000000, 59.00000000 5.00000000, 61.00000000 8.00000000), (69.00000000 3.00000000, 67.00000000 4.00000000, 66.00000000 7.00000000, 68.00000000 9.00000000))

ST_MPointFromText function

The ST_MPointFromText function takes a well-known text representation of a multipoint and, optionally, a spatial reference system identifier as input parameters and returns the corresponding multipoint.

If the given well-known text representation is null, then null is returned.

The recommended function for achieving the same result is the ST_MultiPoint function. It is recommended because of its flexibility: ST_MultiPoint takes additional forms of input as well as the well-known text representation.

Syntax

```
►► db2gse.ST_MPointFromText (—wkt— [—srs_id—]) ►►
```

Parameters

wkt A value of type CLOB(2G) that contains the well-known text representation of the resulting multipoint.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting multipoint.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If the specified *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_MultiPoint

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how ST_MPointFromText can be used to create and insert a multipoint from its well-known text representation. The record that is inserted has ID = 1110, and the geometry is a multipoint in spatial reference system 1. The multipoint is in the well-known text representation of a multipoint. The X and Y coordinates for this geometry are: (1, 2) (4, 3) (5, 6).

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mpoints (id INTEGER, geometry ST_MultiPoint)
```

```
INSERT INTO sample_mpoints
VALUES (1110, ST_MPointFromText ('multipoint (1 2, 4 3, 5 6) '), 1 )
```

The following SELECT statement returns the multipoint that was recorded in the table:

```
SELECT id, CAST( ST_AsText( geometry ) AS VARCHAR(280) ) MULTIPOINT
FROM sample_mpoints
WHERE id = 1110
```

Results:

ID	MULTIPOINT
1110	MULTIPOINT (1.00000000 2.00000000, 4.00000000 3.00000000, 5.00000000 6.00000000)

ST_MPointFromWKB function

The ST_MPointFromWKB function takes a well-known binary representation of a multipoint and, optionally, a spatial reference system identifier as input parameters and returns the corresponding multipoint.

If the given well-known binary representation is null, then null is returned.

The recommended function for achieving the same result is the ST_MultiPoint function. It is recommended because of its flexibility: ST_MultiPoint takes additional forms of input as well as the well-known binary representation.

Syntax

►► db2gse.ST_MPointFromWKB ((*wkb* , *srs_id*)) ►►

Parameters

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting multipoint.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting multipoint.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If the specified *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_MultiPoint

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how ST_MPointFromWKB can be used to create a multipoint from its well-known binary representation. The geometry is a multipoint in spatial reference system 1. In this example, the multipoint gets stored with ID = 10 in the GEOMETRY column of the SAMPLE_MPOINTS table, and then the WKB column is updated with its well-known binary representation (using the ST_AsBinary function). Finally, the ST_MPointFromWKB function is used to return the multipoint from the WKB column. The X and Y coordinates for this geometry are: (44, 14) (35, 16) (24, 13).

The SAMPLE_MPOINTS table has a GEOMETRY column, where the multipoint is stored, and a WKB column, where the multipoint's well-known binary representation is stored.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mpoints (id INTEGER, geometry ST_MultiPoint,
                             wkb BLOB(32K))

INSERT INTO sample_mpoints
VALUES (10, ST_MultiPoint ('multipoint ( 4 14, 35 16, 24 13)', 1))

UPDATE sample_mpoints AS temporary_correlated
SET wkb = ST_AsBinary( geometry )
WHERE id = temporary_correlated.id

In the following SELECT statement, the ST_MPointFromWKB function is used to
retrieve the multipoint from the WKB column.

SELECT id, CAST( ST_AsText( ST_MLineFromWKB (wkb)) AS VARCHAR(100)) MULTIPOINT
FROM sample_mpoints
WHERE id = 10
```

Results:

ID	MULTIPOINT
----	------------

10	MULTIPOINT (44.00000000 14.00000000, 35.00000000 16.00000000 24.00000000 13.00000000)
----	--

ST_MPolyFromText function

The ST_MPolyFromText function takes a well-known text representation of a multipolygon and, optionally, a spatial reference system identifier as input parameters and returns the corresponding multipolygon.

If the given well-known text representation is null, then null is returned.

The recommended function for achieving the same result is the ST_MultiPolygon function. It is recommended because of its flexibility: ST_MultiPolygon takes additional forms of input as well as the well-known text representation.

Syntax

```
db2gse.ST_MPolyFromText(wkt [, srs_id])
```

Parameters

wkt A value of type CLOB(2G) that contains the well-known text representation of the resulting multipolygon.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting multipolygon.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If the specified *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_MultiPolygon

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how ST_MPolyFromText can be used to create and insert a multipolygon from its well-known text representation. The record that is inserted has ID = 1110, and the geometry is a multipolygon in spatial reference system 1. The multipolygon is in the well-known text representation of a multipolygon. The X and Y coordinates for this geometry are:

- Polygon 1: (3, 3) (4, 6) (5, 3) (3, 3)
- Polygon 2: (8, 24) (9, 25) (1, 28) (8, 24)
- Polygon 3: (13, 33) (7, 36) (1, 40) (10, 43) (13, 33)

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mpolys (id INTEGER, geometry ST_MultiPolygon)

INSERT INTO sample_mpolys
VALUES (1110,
       ST_MPolyFromText ('multipolygon (( (3 3, 4 6, 5 3, 3 3),
                                           (8 24, 9 25, 1 28, 8 24),
                                           (13 33, 7 36, 1 40, 10 43 13 33) ))', 1) )

```

The following SELECT statement returns the multipolygon that was recorded in the table:

```

SELECT id, CAST( ST_AsText( geometry ) AS VARCHAR(350) ) MULTI_POLYGON
FROM sample_mpolys
WHERE id = 1110

```

Results:

ID	MULTI_POLYGON
1110	MULTIPOLYGON (((13.00000000 33.00000000, 10.00000000 43.00000000, 1.00000000 40.00000000, 7.00000000 36.00000000, 13.00000000 33.00000000)), ((8.00000000 24.00000000, 9.00000000 25.00000000, 1.00000000 28.00000000, 8.00000000 24.00000000)), (3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000, 3.00000000 3.00000000)))

ST_MPolyFromWKB function

The ST_MPolyFromWKB function takes a well-known binary representation of a multipolygon and, optionally, a spatial reference system identifier as input parameters and returns the corresponding multipolygon.

If the given well-known binary representation is null, then null is returned.

The recommended function for achieving the same result is the ST_MultiPolygon function. It is recommended because of its flexibility: ST_MultiPolygon takes additional forms of input as well as the well-known binary representation.

Syntax

```

>> db2gse.ST_MPolyFromWKB (—wkb— [—srs_id—]) <<

```

Parameters

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting multipolygon.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting multipolygon.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If the specified *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_MultiPolygon

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how ST_MPolyFromWKB can be used to create a multipolygon from its well-known binary representation. The geometry is a multipolygon in spatial reference system 1. In this example, the multipolygon gets stored with ID = 10 in the GEOMETRY column of the SAMPLE_MPOLYS table, and then the WKB column is updated with its well-known binary representation (using the ST_AsBinary function). Finally, the ST_MPolyFromWKB function is used to return the multipolygon from the WKB column. The X and Y coordinates for this geometry are:

- Polygon 1: (1, 72) (4, 79) (5, 76) (1, 72)
- Polygon 2: (10, 20) (10, 40) (30, 41) (10, 20)
- Polygon 3: (9, 43) (7, 44) (6, 47) (9, 43)

The SAMPLE_MPOLYS table has a GEOMETRY column, where the multipolygon is stored, and a WKB column, where the multipolygon's well-known binary representation is stored.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mpolys (id INTEGER,
    geometry ST_MultiPolygon, wkb BLOB(32K))
```

```
INSERT INTO sample_mpolys
VALUES (10, ST_MultiPolygon ('multipolygon
    (( (1 72, 4 79, 5 76, 1 72),
      (10 20, 10 40, 30 41, 10 20),
      (9 43, 7 44, 6 47, 9 43) ))', 1))
```

```
UPDATE sample_mpolys AS temporary_correlated
SET wkb = ST_AsBinary( geometry )
WHERE id = temporary_correlated.id
```

In the following SELECT statement, the ST_MPolyFromWKB function is used to retrieve the multipolygon from the WKB column.

```
SELECT id, CAST( ST_AsText( ST_MPolyFromWKB (wkb) )
    AS VARCHAR(320) ) MULTIPOLYGON
FROM sample_mpolys
WHERE id = 10
```

Results:

ID	MULTIPOLYGON
10	MULTIPOLYGON (((10.00000000 20.00000000, 30.00000000 41.00000000, 10.00000000 40.00000000, 10.00000000 20.00000000)), (1.00000000 72.00000000, 5.00000000 76.00000000, 4.00000000 79.00000000, 1.00000000 72.00000000)), (9.00000000 43.00000000, 6.00000000 47.00000000, 7.00000000 44.00000000, 9.00000000 43.00000000)))

ST_MultiLineString function

The ST_MultiLineString function constructs a multilinestring from a given input.

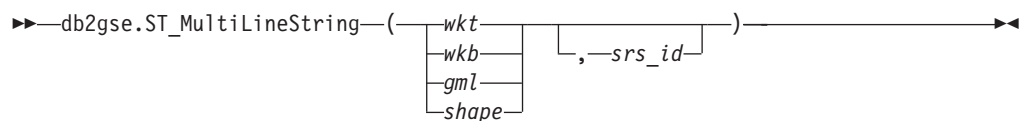
The input can be given in one of the following formats:

- A well-known text representation
- A well-known binary representation
- A shape representation
- A representation in the geography markup language (GML)

An optional spatial reference system identifier can be specified to identify the spatial reference system that the resulting multilinestring is in.

If the well-known text representation, the well-known binary representation, the shape representation, or the GML representation is null, then null is returned.

Syntax



Parameters

- wkt** A value of type CLOB(2G) that contains the well-known text representation of the resulting multilinestring.
- wkb** A value of type BLOB(2G) that contains the well-known binary representation of the resulting multilinestring.
- gml** A value of type CLOB(2G) that represents the resulting multilinestring using the geography markup language.
- shape** A value of type BLOB(2G) that represents the shape representation of the resulting multilinestring.
- srs_id** A value of type INTEGER that identifies the spatial reference system for the resulting multilinestring.

If the *srs_id* parameter is omitted, then the spatial reference system with the numeric identifier 0 (zero) is used.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_MultiLineString

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how ST_MultiLineString can be used to create and insert a multilinestring from its well-known text representation. The record that is inserted

has ID = 1110, and the geometry is a multilinestring in spatial reference system 1. The multilinestring is in the well-known text representation of a multilinestring. The X and Y coordinates for this geometry are:

- Line 1: (33, 2) (34, 3) (35, 6)
- Line 2: (28, 4) (29, 5) (31, 8) (43, 12)
- Line 3: (39, 3) (37, 4) (36, 7)

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mlines (id INTEGER,
                             geometry ST_MultiLineString)

INSERT INTO sample_mlines
VALUES (1110,
        ST_MultiLineString ('multilinestring ( (33 2, 34 3, 35 6),
                                                (28 4, 29 5, 31 8, 43 12),
                                                (39 3, 37 4, 36 7) )', 1) )
```

The following SELECT statement returns the multilinestring that was recorded in the table:

```
SELECT id,
       CAST( ST_AsText( geometry ) AS VARCHAR(280) )
MULTI_LINE_STRING
FROM sample_mlines
WHERE id = 1110
```

Results:

ID	MULTI_LINE_STRING
1110	MULTILINESTRING ((33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000), (28.00000000 4.00000000, 29.00000000 5.00000000, 31.00000000 8.00000000, 43.00000000 12.00000000), (39.00000000 3.00000000, 37.00000000 4.00000000, 36.00000000 7.00000000))

ST_MultiPoint function

The ST_MultiPoint function constructs a multipoint from a given input.

The input can be given in one of the following formats:

- A well-known text representation
- A well-known binary representation
- A shape representation
- A representation in the geography markup language (GML)

An optional spatial reference system identifier can be specified to indicate the spatial reference system the resulting multipoint is in.

If the well-known text representation, the well-known binary representation, the shape representation, or the GML representation is null, then null is returned.

Syntax

```
db2gse.ST_MultiPoint( ( wkt | wkb | gml | shape ) [, srs_id] )
```

Parameters

- wkt** A value of type CLOB(2G) that contains the well-known text representation of the resulting multipoint.
- wkb** A value of type BLOB(2G) that contains the well-known binary representation of the resulting multipoint.
- gml** A value of type CLOB(2G) that represents the resulting multipoint using the geography markup language.
- shape** A value of type BLOB(2G) that represents the shape representation of the resulting multipoint.
- srs_id** A value of type INTEGER that identifies the spatial reference system for the resulting multipoint.
- If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.
- If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_Point

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how ST_MultiPoint can be used to create and insert a multipoint from its well-known text representation. The record that is inserted has ID = 1110, and the geometry is a multipoint in spatial reference system 1. The multipoint is in the well-known text representation of a multipoint. The X and Y coordinates for this geometry are: (1, 2) (4, 3) (5, 6).

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mpoints (id INTEGER, geometry ST_MultiPoint)
```

```
INSERT INTO sample_mpoints
VALUES (1110, ST_MultiPoint ('multipoint (1 2, 4 3, 5 6) '), 1))
```

The following SELECT statement returns the multipoint that was recorded in the table:

```
SELECT id, CAST( ST_AsText(geometry) AS VARCHAR(90)) MULTIPOINT
FROM sample_mpoints
WHERE id = 1110
```

Results:

ID	MULTIPOINT
1110	MULTIPOINT (1.00000000 2.00000000, 4.00000000 3.00000000, 5.00000000 6.00000000)

ST_MultiPolygon function

The ST_MultiPolygon constructs a multipolygon from a given input.

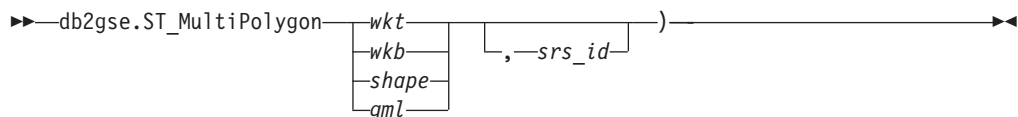
The input can be given in one of the following format:

- A well-known text representation
- A well-known binary representation
- A shape representation
- A representation in the geography markup language (GML)

An optional spatial reference system identifier can be specified to identify the spatial reference system that the resulting multipolygon is in.

If the well-known text representation, the well-known binary representation, the shape representation, or the GML representation is null, then null is returned.

Syntax



Parameters

wkt A value of type CLOB(2G) that contains the well-known text representation of the resulting multipolygon.

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting multipolygon.

gml A value of type CLOB(2G) that represents the resulting multipolygon using the geography markup language.

shape A value of type BLOB(2G) that represents the shape representation of the resulting multipolygon.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting multipolygon.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If *srs_id* does not identify a spatial reference system listed in the catalog view `DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS`, then an exception condition is raised (SQLSTATE 38SU1).

Return type

`db2gse.ST_MultiPolygon`

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how `ST_MultiPolygon` can be used to create and insert a multipolygon from its well-known text representation. The record that is inserted

has ID = 1110, and the geometry is a multipolygon in spatial reference system 1. The multipolygon is in the well-known text representation of a multipolygon. The X and Y coordinates for this geometry are:

- Polygon 1: (3, 3) (4, 6) (5, 3) (3, 3)
- Polygon 2: (8, 24) (9, 25) (1, 28) (8, 24)
- Polygon 3: (13, 33) (7, 36) (1, 40) (10, 43) (13, 33)

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mpolys (id INTEGER, geometry ST_MultiPolygon)
```

```
INSERT INTO sample_mpolys
VALUES (1110,
       ST_MultiPolygon ('multipolygon (( (3 3, 4 6, 5 3, 3 3),
                                           (8 24, 9 25, 1 28, 8 24),
                                           (13 33, 7 36, 1 40, 10 43 13 33) ))', 1) )
```

The following SELECT statement returns the multipolygon that was recorded in the table:

```
SELECT id, CAST( ST_AsText( geometry ) AS VARCHAR(350) ) MULTI_POLYGON
FROM sample_mpolys
WHERE id = 1110
```

Results:

ID	MULTI_POLYGON
1110	MULTIPOLYGON (((13.00000000 33.00000000, 10.00000000 43.00000000, 1.00000000 40.00000000, 7.00000000 36.00000000, 13.00000000 33.00000000)), ((8.00000000 24.00000000, 9.00000000 25.00000000, 1.00000000 28.00000000, 8.00000000 24.00000000)), ((3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000, 3.00000000 3.00000000)))

ST_NumGeometries function

The ST_NumGeometries function takes a geometry collection as an input parameter and returns the number of geometries in the collection.

If the given geometry collection is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

```
►►—db2gse.ST_NumGeometries—(—collection—)—————►►
```

Parameter

collection

A value of type ST_GeomCollection or one of its subtypes that represents the geometry collection for which the number of geometries is returned.

Return Type

INTEGER

Example

Two geometry collections are stored in the SAMPLE_GEOMCOLL table. One is a multipolygon, and the other is a multipoint. The ST_NumGeometries function determines how many individual geometries are within each geometry collection.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geomcoll (id INTEGER, geometry ST_GeomCollection)

INSERT INTO sample_geomcoll
VALUES (1,
       ST_MultiPolygon ('multipolygon (( (3 3, 4 6, 5 3, 3 3),
                                         (8 24, 9 25, 1 28, 8 24),
                                         (13 33, 7 36, 1 40, 10 43, 13 33) ))', 1) )

INSERT INTO sample_geomcoll
VALUES (2, ST_MultiPoint ('multipoint (1 2, 4 3, 5 6, 7 6, 8 8)', 1) )

SELECT id, ST_NumGeometries (geometry) NUM_GEOMS_IN_COLL
FROM sample_geomcoll
```

Results:

ID	NUM_GEOMS_IN_COLL
1	3
2	5

ST_NumInteriorRing function

the ST_NumInteriorRing function takes a polygon as an input parameter and returns the number of its interior rings.

If the given polygon is null or is empty, then null is returned.

If the polygon has no interior rings, then 0 (zero) is returned.

This function can also be called as a method.

Syntax

```
►►—db2gse.ST_NumInteriorRing—(—polygon—)—————◄◄
```

Parameter

polygon

A value of type ST_Polygon that represents the polygon for which the number of interior rings is returned.

Return type

INTEGER

Example

The following example creates two polygons:

- One with two interior rings
- One without any interior rings

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)
```

```
INSERT INTO sample_polys
VALUES (1, ST_Polygon('polygon
((40 120, 90 120, 90 150, 40 150, 40 120),
(50 130, 60 130, 60 140, 50 140, 50 130),
(70 130, 80 130, 80 140, 70 140, 70 130))' , 0) )
```

```
INSERT INTO sample_polys
VALUES (2, ST_Polygon('polygon ((5 15, 50 15, 50 105, 5 15))' , 0) )
```

The ST_NumInteriorRing function is used to return the number of rings in the geometries in the table:

```
SELECT id, ST_NumInteriorRing(geometry) NUM_RINGS
FROM sample_polys
```

Results:

ID	NUM_RINGS
1	2
2	0

ST_NumLineStrings function

The ST_NumLineStrings function takes a multilinestring as an input parameter and returns the number of linestrings that it contains.

If the given multilinestring is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

```
►►—db2gse.ST_NumLineStrings—(—multilinestring—)—————►►
```

Parameter

multilinestring

A value of type ST_MultiLineString that represents the multilinestring for which the number of linestrings is returned.

Return type

INTEGER

Example

Multilinestrings are stored in the SAMPLE_MLINES table. The ST_NumLineStrings function determines how many individual geometries are within each multilinestring.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mlines (id INTEGER, geometry ST_MultiLineString)
```

```
INSERT INTO sample_mlines
VALUES (110, ST_MultiLineString ('multilinestring
( (33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12),
```

```

        (39 3, 37 4, 36 7))', 1) )
INSERT INTO sample_mlines
VALUES (111, ST_MultiLineString ('multilinestring
  ( 3 2, 4 3, 5 6),
  (8 4, 9 5, 3 8, 4 12))', 1) )

SELECT id, ST_NumLineStrings (geometry) NUM_WITHIN
FROM sample_mlines

```

Results:

ID	NUM_WITHIN
110	3
111	2

ST_NumPoints function

The ST_NumPoints function takes a geometry as an input parameter and returns the number of points that were used to define that geometry. For example, if the geometry is a polygon and five points were used to define that polygon, then the returned number is 5.

If the given geometry is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_NumPoints—(*—geometry—*)—►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry for which the number of points is returned.

Return type

INTEGER

Example

A variety of geometries are stored in the table. The ST_NumPoints function determines how many points are within each geometry in the SAMPLE_GEOMETRIES table.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries (spatial_type VARCHAR(18), geometry ST_Geometry)

```

```

INSERT INTO sample_geometries
VALUES ('st_point',
      ST_Point (2, 3, 0) )

```

```

INSERT INTO sample_geometries
VALUES ('st_linestring',
      ST_LineString ('linestring (2 5, 21 3, 23 10)', 0) )

```

```

INSERT INTO sample_geometries
VALUES ('st_polygon',

```



```

        ST_Polygon ('polygon ((110 120, 110 140, 120 130, 110 120))', 0) )

SELECT spatial_type, ST_NumPoints (geometry) NUM_POINTS
FROM sample_geometries

```

Results:

SPATIAL_TYPE	NUM_POINTS
st_point	1
st_linestring	3
st_polygon	4

ST_NumPolygons function

The ST_NumPolygons function takes a multipolygon as an input parameter and returns the number of polygons that it contains.

If the given multipolygon is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

```

►►—db2gse.ST_NumPolygons—(—multipolygon—)—————►►

```

Parameter

multipolygon

A value of type ST_MultiPolygon that represents the multipolygon for which the number of polygons is returned.

Return type

INTEGER

Example

Multipolygons are stored in the SAMPLE_MPOLYS table. The ST_NumPolygons function determines how many individual geometries are within each multipolygon.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mpolys (id INTEGER, geometry ST_MultiPolygon)

INSERT INTO sample_mpolys
VALUES (1,
        ST_MultiPolygon ('multipolygon (( (3 3, 4 6, 5 3, 3 3),
                                           (8 24, 9 25, 1 28, 8 24),
                                           (13 33, 7 36, 1 40, 10 43, 13 33) ))', 1) )

INSERT INTO sample_mpolys
VALUES (2,
        ST_MultiPolygon ('multipolygon empty', 1) )

INSERT INTO sample_mpolys
VALUES (3,
        ST_MultiPolygon ('multipolygon (( (3 3, 4 6, 5 3, 3 3),

```

```
(13 33, 7 36, 1 40, 10 43, 13 33) ))', 1) )
```

```
SELECT id, ST_NumPolygons (geometry) NUM_WITHIN
FROM sample_mpolys
```

Results:

ID	NUM_WITHIN
1	3
2	0
3	2

ST_Overlaps function

The ST_Overlaps function takes two geometries as input parameters. If the intersection of the geometries results in a geometry of the same dimension but is not equal to either of the given geometries, it returns 1 . Otherwise, it returns 0 (zero).

If any of the two geometries is null or is empty, then null is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry, it will be converted to the other spatial reference system.

Syntax

```
►►—db2gse.ST_Overlaps—(—geometry1—,—geometry2—)—►►
```

Parameters

geometry1

A value of type ST_Geometry or one of its subtypes that represents the geometry that is tested to overlap with *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the geometry that is tested to overlap with *geometry1*.

Return type

INTEGER

Examples

Example 1

This example illustrates the use of ST_Overlaps. Various geometries are created and inserted into the SAMPLE_GEOMETRIES table

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries (id INTEGER, geometry ST_Geometry)
```

```
INSERT INTO sample_geometries
VALUES
(1, ST_Point (10, 20, 1)),
(2, ST_Point ('point(41 41)', 1) ),
(10, ST_LineString('linestring(1 10, 3 12, 10 10)', 1) ),
(20, ST_LineString('linestring(50 10, 50 12, 45 10)', 1)),
(30, ST_LineString('linestring(50 12, 50 10, 60 8)', 1)),
```

```
(100, ST_Polygon('polygon((0 0, 0 40, 40 40, 40 0, 0 0))', 1)),
(110, ST_Polygon('polygon((30 10, 30 30, 50 30, 50 10, 30 10))', 1)),
(120, ST_Polygon('polygon((0 50, 0 60, 40 60, 40 50, 0 50))', 1))
```

Example 2

This example finds the IDs of points that overlap.

```
SELECT sg1.id, sg2.id
  CASE ST_Overlaps (sg1.geometry, sg2.geometry)
    WHEN 0 THEN 'Points_do_not_overlap'
    WHEN 1 THEN 'Points_overlap'
  END
AS OVERLAP
FROM sample_geometries sg1, sample_geometries sg2
WHERE sg1.id < 10 AND sg2.id < 10 AND sg1.id >= sg2.id
```

Results:

ID	ID	OVERLAP
	1	1 Points_do_not_overlap
	2	1 Points_do_not_overlap
	2	2 Points_do_not_overlap

Example 3

This example finds the IDs of lines that overlap.

```
SELECT sg1.id, sg2.id
  CASE ST_Overlaps (sg1.geometry, sg2.geometry)
    WHEN 0 THEN 'Lines_do_not_overlap'
    WHEN 1 THEN 'Lines_overlap'
  END
AS OVERLAP
FROM sample_geometries sg1, sample_geometries sg2
WHERE sg1.id >= 10 AND sg1.id < 100
  AND sg2.id >= 10 AND sg2.id < 100
  AND sg1.id >= sg2.id
```

Results:

ID	ID	OVERLAP
	10	10 Lines_do_not_overlap
	20	10 Lines_do_not_overlap
	30	10 Lines_do_not_overlap
	20	20 Lines_do_not_overlap
	30	20 Lines_overlap
	30	30 Lines_do_not_overlap

Example 4

This example finds the IDs of polygons that overlap.

```
SELECT sg1.id, sg2.id
  CASE ST_Overlaps (sg1.geometry, sg2.geometry)
    WHEN 0 THEN 'Polygons_do_not_overlap'
    WHEN 1 THEN 'Polygons_overlap'
  END
AS OVERLAP
FROM sample_geometries sg1, sample_geometries sg2
WHERE sg1.id >= 100 AND sg2.id >= 100 AND sg1.id >= sg2.id
```

Results:

ID	ID	OVERLAP
	100	100 Polygons_do_not_overlap
	110	100 Polygons_overlap
	120	100 Polygons_do_not_overlap

```

110      110 Polygons_do_not_overlap
120      110 Polygons_do_not_overlap
120      120 Polygons_do_not_overlap

```

ST_Perimeter function

The ST_Perimeter function takes a surface or multisurface and, optionally, a unit as input parameters and returns the perimeter of the surface or multisurface, that is the length of its boundary, measured in the default or given units.

If the given surface or multisurface is null or is empty, null is returned.

This function can also be called as a method.

Syntax

```

▶▶—db2gse.ST_Perimeter—(—surface—[,—unit—])————▶▶

```

Parameters

surface

A value of type ST_Surface, ST_MultiSurface, or one of their subtypes for which the perimeter is returned.

unit

A VARCHAR(128) value that identifies the units in which the perimeter is measured. The supported units of measure are listed in the DB2GSE.ST_UNITS_OF_MEASURE catalog view.

If the *unit* parameter is omitted, the following rules are used to determine the unit in which the perimeter is measured:

- If *surface* is in a projected or geocentric coordinate system, the linear unit associated with this coordinate system is the default.
- If *surface* is in a geographic coordinate system, the angular unit associated with this coordinate system is the default.

Restrictions on unit conversions: An error (SQLSTATE 38SU4) is returned if any of the following conditions occur:

- The geometry is in an unspecified coordinate system and the *unit* parameter is specified.
- The geometry is in a projected coordinate system and an angular unit is specified.
- The geometry is in a geographic coordinate system and a linear unit is specified.

Return type

DOUBLE

Examples

Example 1

This example illustrates the use of the ST_Perimeter function. A spatial reference system with an ID of 4000 is created using a call to db2se, and a polygon is created in that spatial reference system.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

db2se create_srs se_bank -srsId 4000 -srsName new_york1983
-xOffset 0 -yOffset 0 -xScale 1 -yScale 1
-coordsysName NAD_1983_StatePlane_New_York_East_FIPS_3101_Feet

```

The SAMPLE_POLYS table is created to hold a geometry with a perimeter of 18.

```

CREATE TABLE sample_polys (id SMALLINT, geometry ST_Polygon)

INSERT INTO sample_polys
VALUES (1, ST_Polygon ('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))', 4000))

```

Example 2

This example lists the ID and perimeter of the polygon.

```

SELECT id, ST_Perimeter (geometry) AS PERIMETER
FROM sample_polys

```

Results:

ID	PERIMETER
1	+1.8000000000000000E+001

Example 3

This example lists the ID and perimeter of the polygon with the perimeter measured in meters.

```

SELECT id, ST_Perimeter (geometry, 'METER') AS PERIMETER_METER
FROM sample_polys

```

Results:

ID	PERIMETER_METER
1	+5.48641097282195E+000

ST_PerpPoints function

The ST_PerpPoints function takes a curve or multicurve and a point as input parameters and returns the perpendicular projection of the given point on the curve or multicurve.

The point with the smallest distance between the given point and the perpendicular point is returned. If two or more such perpendicular projected points are equidistant from the given point, they are all returned. If no perpendicular point can be constructed, then an empty point is returned.

If the given curve or multicurve has Z or M coordinates, the Z or M coordinate of the resulting points are computed by interpolation on the given curve or multicurve.

If the given curve or point is empty, then an empty point is returned. If the given curve or point is null, then null is returned.

This function can also be called as a method.

Syntax

►► db2gse.ST_PerpPoints(—*curve*—,—*point*—)◄◄

Parameters

- curve** A value of type ST_Curve, ST_MultiCurve, or one of their subtypes that represents the curve or multicurve in which the perpendicular projection of the *point* is returned.
- point** A value of type ST_Point that represents the point that is perpendicular projected onto *curve*.

Return type

db2gse.ST_MultiPoint

Examples

Example 1

This example illustrates the use of the ST_PerpPoints function to find points that are perpendicular to the linestring stored in the following table. The ST_LineString function is used in the INSERT statement to create the linestring.

```
SET CURRENT FUNCTION PATH=CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines (id INTEGER, line ST_LineString)

INSERT INTO sample_lines (id, line)
VALUES (1, ST_LineString('linestring (0 10, 0 0, 10 0, 10 10)' , 0) )
```

Example 2

This example finds the perpendicular projection on the linestring of a point with coordinates (5, 0). The ST_AsText function is used to convert the returned value (a multipoint) to its well-known text representation.

```
SELECT CAST(ST_AsText(ST_PerpPoints(line,ST_Point(5,0)))
AS VARCHAR(50)) PERP
FROM sample_lines
```

Results:

```
PERP
-----
MULTIPOINT(5.00000000 0.00000000)
```

Example 3

This example finds the perpendicular projections on the linestring of a point with coordinates (5, 5). In this case, there are three points on the linestring that are equidistant to the given location. Therefore, a multipoint that consists of all three points is returned.

```
SELECT CAST(ST_AsText(ST_PerpPoints(line, ST_Point(5,5)))
AS VARCHAR(160)) PERP
FROM sample_lines
```

Results:

```
PERP
-----
MULTIPOINT(0.00000000 5.00000000,5.00000000 0.00000000,10.00000000 5.00000000)
```

Example 4

This example finds the perpendicular projections on the linestring of a point with coordinates (5, 10). In this case there are three different perpendicular points that can be found. However, the ST_PerpPoints function only returns those points that are closest to the given point. Thus, a multipoint that consists of only the two closest points is returned. The third point is not included.

```
SELECT CAST(ST_AsText(ST_PerpPoints(line,ST_Point(5, 10)))
          AS VARCHAR(80)) PERP
FROM sample_lines
```

Results:

```
PERP
-----
MULTIPOINT(0.00000000 10.00000000, 10.00000000 10.00000000 )
```

Example 5

This example finds the perpendicular projection on the linestring of a point with coordinates (5, 15).

```
SELECT CAST(ST_AsText(ST_PerpPoints(line,ST_Point('point(5 15)',0)))
          AS VARCHAR(80)) PERP
FROM sample_lines
```

Results:

```
PERP
-----
MULTIPOINT ( 5.00000000 0.00000000 )
```

Example 6

In this example, the specified point with coordinates (15 15) has no perpendicular projection on the linestring. Therefore, an empty geometry is returned.

```
SELECT CAST(ST_AsText(ST_PerpPoints(line,ST_Point(15,15)))
          AS VARCHAR(80)) PERP
FROM sample_lines
```

Results:

```
PERP
-----
MULTIPOINT EMPTY
```

ST_Point function

The ST_Point function constructs a point from coordinate information or a resulting point of a representation.

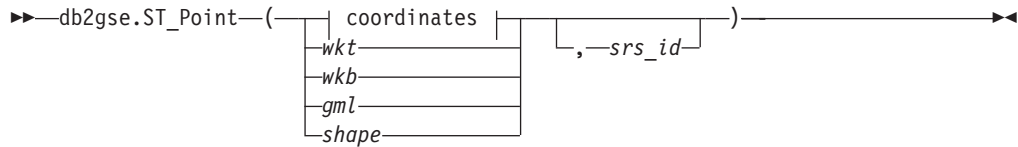
ST_Point constructs a point from one of the following sets of input:

- X and Y coordinates only
- X, Y, and Z coordinates
- X, Y, Z, and M coordinates
- A well-known text representation
- A well-known binary representation
- A shape representation
- A representation in the geography markup language (GML)

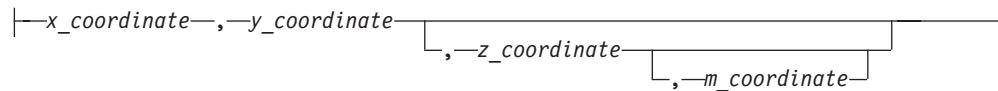
An optional spatial reference system identifier can be specified to indicate the spatial reference system that the resulting point is in.

If the point is constructed from coordinates, and if the X or Y coordinate is null, then an exception condition is raised (SQLSTATE 38SUP). If the Z or M coordinate is null, then the resulting point will not have a Z or M coordinate, respectively. If the point is constructed from its well-known text representation, its well-known binary representation, its shape representation, or its GML representation, and if the representation is null, then null is returned.

Syntax



coordinates:



Parameters

wkt A value of type CLOB(2G) that contains the well-known text representation of the resulting point.

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting point.

gml A value of type CLOB(2G) that represents the resulting point using the geography markup language.

shape A value of type BLOB(2G) that represents the shape representation of the resulting point.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting point.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

x_coordinate

A value of type DOUBLE that specifies the X coordinate for the resulting point.

y_coordinate

A value of type DOUBLE that specifies the Y coordinate for the resulting point.

z_coordinate

A value of type DOUBLE that specifies the Z coordinate for the resulting point.

If the *z_coordinate* parameter is omitted, the resulting point will not have a Z coordinate. The result of ST_Is3D is 0 (zero) for such a point.

m_coordinate

A value of type DOUBLE that specifies the M coordinate for the resulting point.

If the *m_coordinate* parameter is omitted, the resulting point will not have a measure. The result of ST_IsMeasured is 0 (zero) for such a point.

Return type

db2gse.ST_Point

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

Example 1

This example illustrates how ST_Point can be used to create and insert points. The first point is created using a set of X and Y coordinates. The second point is created using its well-known text representation. Both points are geometries in spatial reference system 1.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)
```

```
INSERT INTO sample_points
VALUES (1100, ST_Point (10, 20, 1) )
```

```
INSERT INTO sample_points
VALUES (1101, ST_Point ('point (30 40)', 1) )
```

The following SELECT statement returns the points that were recorded in the table:

```
SELECT id, CAST( ST_AsText( geometry ) AS VARCHAR(90)) POINTS
FROM sample_points
```

Results:

ID	POINTS
1100	POINT (10.00000000 20.00000000)
1101	POINT (30.00000000 40.00000000)

Example 2

This example inserts a record into the SAMPLE_POINTS table with ID 1103 and a point value with an X coordinate of 120, a Y coordinate of 358, an M coordinate of 34, but no Z coordinate.

```
INSERT INTO SAMPLE_POINTS(ID, GEOMETRY)
VALUES(1103, db2gse.ST_Point(120, 358, CAST(NULL AS DOUBLE), 34, 1))
SELECT id, CAST( ST_AsText( geometry ) AS VARCHAR(90)) POINTS
FROM sample_points
```

Results:

ID	POINTS
1103	POINT M (120.0000000 358.0000000 34.00000000)

Example 3

This example inserts a row into the SAMPLE_POINTS table with ID 1104 and a point value with an X coordinate of 1003, a Y coordinate of 9876, a Z

coordinate of 20, and in spatial reference system 0, using the geography markup language for its representation.

```
INSERT INTO SAMPLE_POINTS(ID, GEOMETRY)
VALUES(1104, db2gse.ST_Point('<gml:Point><gml:coord>
    <gml:x>1003</gml:X><gml:Y>9876</gml:Y><gml:Z>20</gml:Z>
</gml:coord></gml:Point>', 1))
SELECT id, CAST( ST_AsText( geometry ) AS VARCHAR(90) ) POINTS
FROM sample_points
```

Results:

ID	POINTS
1104	POINT Z (1003.000000 9876.000000 20.00000000)

ST_PointAtDistance function

The ST_PointAtDistance function takes a curve or multi-curve geometry and a distance as input parameters and returns the point geometry at the given distance along the curve geometry.

This function can also be called as a method.

Syntax

►►—db2gse.ST_PointAtDistance—(—*geometry*—,—*distance*—)————►►

Parameter

geometry

A value of type ST_Curve or ST_MultiCurve or one of its subtypes that represents the geometry to process.

distance

A value of type DOUBLE that is the distance along the geometry to locate the point.

Return type

db2gse.ST_Point

Example

The following SQL statement creates the SAMPLE_GEOMETRIES table with two columns. The ID column uniquely identifies each row. The GEOMETRY ST_LineString column stores sample geometries.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries(id INTEGER, geometry ST_LINESTRING)
```

The following SQL statement inserts two rows in the SAMPLE_GEOMETRIES table.

```
INSERT INTO sample_geometries(id, geometry)
VALUES
(1,ST_LineString('LINESTRING ZM(0 0 0 0, 10 100 1000 10000)',1)),
(2,ST_LineString('LINESTRING ZM(10 100 1000 10000, 0 0 0 0)',1))
```

The following SELECT statement and the corresponding result set shows how to use the ST_PointAtDistance function to find points at a distance of 15 coordinate units from the start of the linestring.

```
SELECT ID, VARCHAR(ST_AsText(ST_PointAtDistance(geometry, 15)), 50) AS POINTAT
FROM   sample_geometries
```

ID	POINTAT

1	POINT ZM(1.492556 14.925558 149 1493)
2	POINT ZM(8.507444 85.074442 851 8507)

2 record(s) selected.

ST_PointFromText function

The ST_PointFromText function takes a well-known text representation of a point and, optionally, a spatial reference system identifier as input parameters and returns the corresponding point.

If the given well-known text representation is null, then null is returned.

The recommended function for achieving the same result is the ST_Point function. It is recommended because of its flexibility: ST_Point takes additional forms of input as well as the well-known text representation.

Syntax

```
db2gse.ST_PointFromText(—wkt—, —srs_id—)
```

Parameters

- wkt** A value of type CLOB(2G) that contains the well-known text representation of the resulting point.
- srs_id** A value of type INTEGER that identifies the spatial reference system for the resulting point.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_Point

Example

This example illustrates how ST_PointFromText can be used to create and insert a point from its well-known text representation. The record that is inserted has ID = 1110, and the geometry is a point in spatial reference system 1. The point is in the well-known text representation of a point. The X and Y coordinates for this geometry are: (10, 20).

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)

INSERT INTO sample_points
VALUES (1110, ST_PointFromText ('point (30 40)', 1) )

```

The following SELECT statement returns the polygon that was recorded in the table:

```

SELECT id, CAST( ST_AsText( geometry ) AS VARCHAR(35) ) POINTS
FROM sample_points
WHERE id = 1110

```

Results:

ID	POINTS
1110	POINTS (30.00000000 40.00000000)

ST_PointFromWKB function

The ST_PointFromWKB function takes a well-known binary representation of a point and, optionally, a spatial reference system identifier as input parameters and returns the corresponding point.

If the given well-known binary representation is null, then null is returned.

The recommended function for achieving the same result is the ST_Point function. It is recommended because of its flexibility: ST_Point takes additional forms of input as well as the well-known binary representation.

Syntax

```

db2gse.ST_PointFromWKB ( ( wkb , srs_id ) )

```

Parameters

- wkb** A value of type BLOB(2G) that contains the well-known binary representation of the resulting point.
- srs_id** A value of type INTEGER that identifies the spatial reference system for the resulting point.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used implicitly.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_Point

Example

This example illustrates how ST_PointFromWKB can be used to create a point from its well-known binary representation. The geometries are points in spatial reference system 1. In this example, the points get stored in the GEOMETRY column of the

SAMPLE_POLYS table, and then the WKB column is updated with their well-known binary representations (using the ST_AsBinary function). Finally, the ST_PointFromWKB function is used to return the points from the WKB column.

The SAMPLE_POINTS table has a GEOMETRY column, where the points are stored, and a WKB column, where the points' well-known binary representations are stored.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point, wkb BLOB(32K))
```

```
INSERT INTO sample_points
VALUES (10, ST_Point ('point (44 14)', 1) ),
VALUES (11, ST_Point ('point (24 13)', 1))
```

```
UPDATE sample_points AS temporary_correlated
SET wkb = ST_AsBinary( geometry )
WHERE id = temporary_correlated.id
```

In the following SELECT statement, the ST_PointFromWKB function is used to retrieve the points from the WKB column.

```
SELECT id, CAST( ST_AsText( ST_PolyFromWKB (wkb) ) AS VARCHAR(35) ) POINTS
FROM sample_points
```

Results:

ID	POINTS
10	POINT (44.00000000 14.00000000)
11	POINT (24.00000000 13.00000000)

ST_PointN function

The ST_PointN function takes a linestring or a multipoint and an index as input parameters and returns that point in the linestring or multipoint that is identified by the index. The resulting point is represented in the spatial reference system of the given linestring or multipoint.

If the given linestring or multipoint is null or is empty, then null is returned. If the index is smaller than 1 or larger than the number of points in the linestring or multipoint, then null is returned and a warning condition is returned (SQLSTATE 01HS2).

This function can also be called as a method.

Syntax

```
►►—db2gse.ST_PointN—(—geometry—,—index—)—————►◄
```

Parameters

geometry

A value of type ST_LineString or ST_MultiPoint that represents the geometry from which the point that is identified by *index* is returned.

index

A value of type INTEGER that identifies the *n*th point which is to be returned from *geometry*.

Return type

db2gse.ST_Point

Example

The following example illustrates the use of ST_PointN.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines (id INTEGER, line ST_LineString)

INSERT INTO sample_lines
VALUES (1, ST_LineString ('linestring (10 10, 5 5, 0 0, 10 0, 5 5, 0 10)', 0) )

SELECT id, CAST ( ST_AsText (ST_PointN (line, 2) ) AS VARCHAR(60) ) SECOND_INDEX
FROM sample_lines
```

Results:

ID	SECOND_INDEX
1	POINT (5.000000000 5.000000000)

ST_PointOnSurface function

The ST_PointOnSurface function takes a surface or a multisurface as an input parameter and returns a point that is guaranteed to be in the interior of the surface or multisurface. This point is the paracentroid of the surface.

The resulting point is represented in the spatial reference system of the given surface or multisurface.

If the given surface or multisurface is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►►db2gse.ST_PointOnSurface(—*surface*—)◄◄

Parameter

surface

A value of type ST_Surface, ST_MultiSurface, or one of their subtypes that represents the geometry for which a point on the surface is returned.

Return type

db2gse.ST_Point

Example

In the following example, two polygons are created and then ST_PointOnSurface is used. One of the polygons has a hole in its center. The returned points are on the surface of the polygons. They are not necessarily at the exact center of the polygons.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)

INSERT INTO sample_polys
VALUES (1,
       ST_Polygon ('polygon ( (40 120, 90 120, 90 150, 40 150, 40 120) ,
                           (50 130, 80 130, 80 140, 50 140, 50 130) )' ,0) )
INSERT INTO sample_polys
VALUES (2,
       ST_Polygon ('polygon ( (10 10, 50 10, 10 30, 10 10) )', 0) )

SELECT id, CAST (ST_AsText (ST_PointOnSurface (geometry) ) AS VARCHAR(80) )
       POINT_ON_SURFACE
FROM sample_polys

```

Results:

ID	POINT_ON_SURFACE
1	POINT (65.00000000 125.00000000)
2	POINT (30.00000000 15.00000000)

ST_PolyFromText function

The ST_PolyFromText function takes a well-known text representation of a polygon and, optionally, a spatial reference system identifier as input parameters and returns the corresponding polygon.

If the given well-known text representation is null, then null is returned.

The recommended function for achieving the same result is the ST_Polygon function. It is recommended because of its flexibility: ST_Polygon takes additional forms of input as well as the well-known text representation.

Syntax

```

>> db2gse.ST_PolyFromText (—wkt—, —srs_id—)

```

Parameters

wkt A value of type CLOB(2G) that contains the well-known text representation of the resulting polygon.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting polygon.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_Polygon

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how `ST_PolyFromText` can be used to create and insert a polygon from its well-known text representation. The record that is inserted has ID = 1110, and the geometry is a polygon in spatial reference system 1. The polygon is in the well-known text representation of a polygon. The X and Y coordinates for this geometry are: (50, 20) (50, 40) (70, 30).

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)

INSERT INTO sample_polys
VALUES (1110, ST_PolyFromText ('polygon ((50 20, 50 40, 70 30, 50 20))', 1) )
```

The following `SELECT` statement returns the polygon that was recorded in the table:

```
SELECT id, CAST( ST_AsText( geometry ) AS VARCHAR(120) ) POLYGON
FROM sample_polys
WHERE id = 1110
```

Results:

ID	POLYGON
1110	POLYGON ((50.00000000 20.00000000, 70.00000000 30.00000000, 50.00000000 40.00000000, 50.00000000 20.00000000))

ST_PolyFromWKB function

The `ST_PolyFromWKB` function takes a well-known binary representation of a polygon and, optionally, a spatial reference system identifier as input parameters and returns the corresponding polygon.

If the given well-known binary representation is null, then null is returned.

The recommended function for achieving the same result is the `ST_Polygon` function. It is recommended because of its flexibility: `ST_Polygon` takes additional forms of input as well as the well-known binary representation.

Syntax

►►db2gse.ST_PolyFromWKB(—wkb—,—srs_id—)—►►

Parameters

- wkb** A value of type `BLOB(2G)` that contains the well-known binary representation of the resulting polygon.
- srs_id** A value of type `INTEGER` that identifies the spatial reference system for the resulting polygon.

If the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_Polygon

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how ST_PolyFromWKB can be used to create a polygon from its well-known binary representation. The geometry is a polygon in spatial reference system 1. In this example, the polygon gets stored with ID = 1115 in the GEOMETRY column of the SAMPLE_POLYS table, and then the WKB column is updated with its well-known binary representation (using the ST_AsBinary function). Finally, the ST_PolyFromWKB function is used to return the multipolygon from the WKB column. The X and Y coordinates for this geometry are: (50, 20) (50, 40) (70, 30).

The SAMPLE_POLYS table has a GEOMETRY column, where the polygon is stored, and a WKB column, where the polygon's well-known binary representation is stored.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon,
    wkb BLOB(32K))

INSERT INTO sample_polys
VALUES (10, ST_Polygon ('polygon ((50 20, 50 40, 70 30, 50 20))', 1) )

UPDATE sample_polys AS temporary_correlated
SET wkb = ST_AsBinary( geometry )
WHERE id = temporary_correlated.id
```

In the following SELECT statement, the ST_PolyFromWKB function is used to retrieve the polygon from the WKB column.

```
SELECT id, CAST( ST_AsText( ST_PolyFromWKB (wkb) )
AS VARCHAR(120) ) POLYGON
FROM sample_polys
WHERE id = 1115
```

Results:

ID	POLYGON
1115	POLYGON ((50.00000000 20.00000000, 70.00000000 30.00000000,50.00000000 40.00000000, 50.00000000 20.00000000))

ST_Polygon function

The ST_Polygon constructs a polygon from a given input.

The input can be given in one of the following formats:

- A closed linestring that defines the exterior ring of the resulting polygon
- A well-known text representation

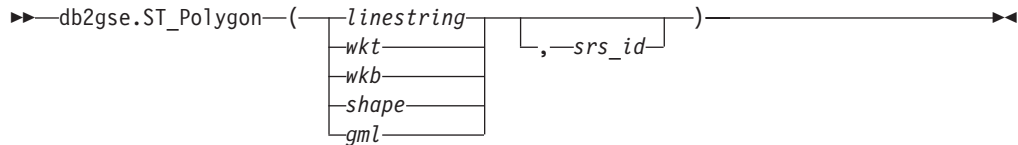
- A well-known binary representation
- A shape representation
- A representation in the geography markup language (GML)

An optional spatial reference system identifier can be specified to identify the spatial reference system that the resulting polygon is in.

If the polygon is constructed from a linestring and the given linestring is null, then null is returned. If the given linestring is empty, then an empty polygon is returned. If the polygon is constructed from its well-known text representation, its well-known binary representation, its shape representation, or its GML representation, and if the representation is null, then null is returned.

This function can also be called as a method for the following cases only: ST_Polygon(*linestring*) and ST_Polygon(*linestring*, *srs_id*).

Syntax



Parameters

linestring

A value of type ST_LineString that represents the linestring that defines the exterior ring for the outer boundary. If *linestring* is not closed and simple, an exception condition is raised (SQLSTATE 38SSL).

wkt A value of type CLOB(2G) that contains the well-known text representation of the resulting polygon.

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting polygon.

shape A value of type BLOB(2G) that represents the shape representation of the resulting polygon.

gml A value of type CLOB(2G) that represents the resulting polygon using the geography markup language.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting polygon.

If the polygon is constructed from a given *linestring* parameter and the *srs_id* parameter is omitted, then the spatial reference system from *linestring* is used implicitly. Otherwise, if the *srs_id* parameter is omitted, the spatial reference system with the numeric identifier 0 (zero) is used.

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_Polygon

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how ST_Polygon can be used to create and insert polygons. Three polygons are created and inserted. All of them are geometries in spatial reference system 1.

- The first polygon is created from a ring (a closed and simple linestring). The X and Y coordinates for this polygon are: (10, 20) (10, 40) (20, 30).
- The second polygon is created using its well-known text representation. The X and Y coordinates for this polygon are: (110, 120) (110, 140) (120, 130).
- The third polygon is a donut polygon. A donut polygon consists of an interior and an exterior polygon. This donut polygon is created using its well-known text representation. The X and Y coordinates for the exterior polygon are: (110, 120) (110, 140) (130, 140) (130, 120) (110, 120). The X and Y coordinates for the interior polygon are: (115, 125) (115, 135) (125, 135) (125, 135) (115, 125).

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)
```

```
INSERT INTO sample_polys
VALUES (1100,
       ST_Polygon (ST_LineString ('linestring
                                   (10 20, 10 40, 20 30, 10 20)',1), 1))
```

```
INSERT INTO sample_polys
VALUES (1101,
       ST_Polygon ('polygon
                   ((110 120, 110 140, 120 130, 110 120))', 1))
```

```
INSERT INTO sample_polys
VALUES (1102,
       ST_Polygon ('polygon
                   ((110 120, 110 140, 130 140, 130 120, 110 120),
                    (115 125, 115 135, 125 135, 125 135, 115 125))', 1))
```

The following SELECT statement returns the polygons that were recorded in the table:

```
SELECT id, CAST( ST_AsText( geometry ) AS VARCHAR(120) ) POLYGONS
FROM sample_polys
```

Results:

ID	POLYGONS
1110	POLYGON ((10.00000000 20.00000000, 20.00000000 30.00000000 10.00000000 40.00000000, 10.00000000 20.00000000))
1101	POLYGON ((110.00000000 120.00000000, 120.00000000 130.00000000 110.00000000 140.00000000, 110.00000000 120.00000000))
1102	POLYGON ((110.00000000 120.00000000, 130.00000000 120.00000000 130.00000000 140.00000000, 110.00000000 140.00000000 110.00000000 120.00000000), (115.00000000 125.00000000, 115.00000000 135.00000000 125.00000000 135.00000000, 125.00000000 135.00000000 115.00000000 125.00000000))

ST_PolygonN function

The ST_PolygonN function takes a multipolygon and an index as input parameters and returns the polygon that is identified by the index. The resulting polygon is represented in the spatial reference system of the given multipolygon.

If the given multipolygon is null or is empty, or if the index is smaller than 1 or larger than the number of polygons, then null is returned.

This function can also be called as a method.

Syntax

►►db2gse.ST_PolygonN(*—multipolygon—*,*—index—*)—————►◄

Parameters

multipolygon

A value of type ST_MultiPolygon that represents the multipolygon from which the polygon that is identified by *index* is returned.

index A value of type INTEGER that identifies the *n*th polygon that is to be returned from *multipolygon*.

Return type

db2gse.ST_Polygon

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display

This example illustrates the use of ST_PolygonN.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_mpolys (id INTEGER, geometry ST_MultiPolygon)

INSERT INTO sample_mpolys
VALUES (1, ST_Polygon ('multipolygon (((3 3, 4 6, 5 3, 3 3),
                                     (8 24, 9 25, 1 28, 8 24)
                                     (13 33, 7 36, 1 40, 10 43,
                                     13 33)))', 1))

SELECT id, CAST ( ST_AsText (ST_PolygonN (geometry, 2) )
AS VARCHAR(120) ) SECOND_INDEX
FROM sample_mpolys
```

Results:

ID	SECOND_INDEX
1	POLYGON ((8.00000000 24.00000000, 9.00000000 25.00000000, 1.00000000 28.00000000, 8.00000000 24.00000000))

ST_Relate function

The ST_Relate function takes two geometries and a Dimensionally Extended 9 Intersection Model (DE-9IM) matrix as input parameters. If the given geometries meet the conditions specified by the matrix, it returns 1. Otherwise, it returns 0.

If any of the given geometries is null or is empty, then null is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry, it will be converted to the other spatial reference system.

This function can also be called as a method.

Syntax

►—db2gse.ST_Relate—(—*geometry1*—,—*geometry2*—,—*matrix*—)————►

Parameters

geometry1

A value of type ST_Geometry or one of its subtypes that represents the geometry that is tested against *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the geometry that is tested against *geometry1*.

matrix A value of CHAR(9) that represents the DE-9IM matrix which is to be used for the test of *geometry1* and *geometry2*.

Return type

INTEGER

Example

The following code creates two separate polygons. Then, the ST_Relate function is used to determine several relationships between the two polygons. For example, whether the two polygons overlap.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_polys (id INTEGER, geometry ST_Polygon)

INSERT INTO sample_polys
VALUES (1,
        ST_Polygon('polygon ( (40 120, 90 120, 90 150, 40 150, 40 120) )', 0))
INSERT INTO sample_polys
VALUES (2,
        ST_Polygon('polygon ( (30 110, 50 110, 50 130, 30 130, 30 110) )', 0))

SELECT ST_Relate(a.geometry, b.geometry, 'T***T**') "Overlaps ",
       ST_Relate(a.geometry, b.geometry, 'T***FF*') "Contains ",
       ST_Relate(a.geometry, b.geometry, 'T*F**F***') "Within   ",
       ST_Relate(a.geometry, b.geometry, 'T*****')  "Intersects",
       ST_Relate(a.geometry, b.geometry, 'T*F**FFF2') "Equals   "
FROM sample_polys a, sample_polys b
WHERE a.id = 1 AND b.id = 2
```

Results:

Overlaps	Contains	Within	Intersects	Equals
-----	-----	-----	-----	-----
1	0	0	1	0

ST_RemovePoint function

The ST_RemovePoint function takes a curve and a point as input parameters and returns the given curve with all points equal to the specified point removed from it. If the given curve has Z or M coordinates, then the point must also have Z or M coordinates. The resulting geometry is represented in the spatial reference system of the given geometry.

If the given curve is empty, then an empty curve is returned. If the given curve is null, or if the given point is null or empty, then null is returned.

This function can also be called as a method.

Syntax

►► db2gse.ST_RemovePoint(—*curve*—,—*point*—)————►►

Parameters

curve A value of type ST_Curve or one of its subtypes that represents the curve from which *point* is removed.

point A value of type ST_Point that identifies the points that are removed from *curve*.

Return type

db2gse.ST_Curve

Examples

Example 1

In the following example, two linestrings are added to the SAMPLE_LINES table. These linestrings are used in the following examples.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines (id INTEGER, line ST_LineString)
```

```
INSERT INTO sample_lines
VALUES (1,ST_LineString('linestring
(10 10, 5 5, 0 0, 10 0, 5 5, 0 10)', 0))
```

```
INSERT INTO sample_lines
VALUES (2, ST_LineString('linestring z
(0 0 4, 5 5 5, 10 10 6, 5 5 7, 0 0 8)', 0))
```

In the following examples, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

Example 2

The following example removes the point (5, 5) from the linestring that has ID = 1. This point occurs twice in the linestring. Therefore, both occurrences are removed.

```
SELECT CAST(ST_AsText (ST_RemovePoint (line, ST_Point(5, 5) ) )
AS VARCHAR(120) ) RESULT
FROM sample_lines
WHERE id = 1
```

Results:

```
RESULT
-----
LINESTRING ( 10.00000000 10.00000000, 0.00000000 0.00000000,
10.00000000 0.00000000, 0.00000000 10.00000000)
```

Example 3

The following example removes the point (5, 5, 5) from the linestring that has ID = 2. This point occurs only once, so only that occurrence is removed.

```
SELECT CAST (ST_AsText (ST_RemovePoint (line, ST_Point(5.0, 5.0, 5.0)))
AS VARCHAR(160) ) RESULT
FROM sample_lines
WHERE id=2
```

Results:

```
RESULT
-----
LINESTRING Z ( 0.00000000 0.00000000 4.00000000, 10.00000000 10.00000000
6.00000000, 5.00000000 5.00000000 7.00000000, 0.00000000
0.00000000 8.00000000)
```

ST_SrsId or ST_SRID function

The ST_SrsId or ST_SRID function takes a geometry and, optionally, a spatial reference system identifier as input parameters.

What it returns depends on what input parameters are specified:

- If the spatial reference system identifier is specified, it returns the geometry with its spatial reference system changed to the specified spatial reference system. No transformation of the geometry is performed.
- If no spatial reference system identifier is given as an input parameter, the current spatial reference system identifier of the given geometry is returned.

If the given geometry is null, then null is returned.

These functions can also be called as methods.

Syntax

```

>> db2gse.ST_SrsId (—geometry—)
db2gse.ST_SRID (—srs_id—)

```

Parameters

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry for which the spatial reference system identifier is to be set or returned.

srs_id A value of type INTEGER that identifies the spatial reference system to be used for the resulting geometry.

Attention: If this parameter is specified, the geometry is not transformed, but is returned with its spatial reference system changed to the specified spatial reference system. As a result of changing to the new spatial reference system, the data might be corrupted. For transformations, use `ST_Transform` instead.

If `srs_id` does not identify a spatial reference system listed in the catalog view `DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS`, then an exception condition is raised (SQLSTATE 38SU1).

Return types

- `INTEGER`, if an `srs_id` is not specified
- `db2gse.ST_Geometry`, if an `srs_id` is specified

Example

Two points are created in two different spatial reference systems. The ID of the spatial reference system that is associated with each point can be found by using the `ST_SrsId` function.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)

INSERT INTO sample_points
VALUES (1, ST_Point( 'point (80 180)', 0 ) )
INSERT INTO sample_points
VALUES (2, ST_Point( 'point (-74.21450127 + 42.03415094)', 1 ) )

SELECT id, ST_SRSId(geometry) SRSID
FROM sample_points
```

Results:

ID	SRSID
1	0
2	1

ST_SrsId or ST_SRID function

The `ST_SrsId` or `ST_SRID` function takes a geometry and, optionally, a spatial reference system identifier as input parameters.

What it returns depends on what input parameters are specified:

- If the spatial reference system identifier is specified, it returns the geometry with its spatial reference system changed to the specified spatial reference system. No transformation of the geometry is performed.
- If no spatial reference system identifier is given as an input parameter, the current spatial reference system identifier of the given geometry is returned.

If the given geometry is null, then null is returned.

These functions can also be called as methods.

Syntax

► `db2gse.ST_SrsId` (`—geometry` `—srs_id`) ►

Parameters

geometry

A value of type `ST_Geometry` or one of its subtypes that represents the geometry for which the spatial reference system identifier is to be set or returned.

srs_id A value of type `INTEGER` that identifies the spatial reference system to be used for the resulting geometry.

Attention: If this parameter is specified, the geometry is not transformed, but is returned with its spatial reference system changed to the specified spatial reference system. As a result of changing to the new spatial reference system, the data might be corrupted. For transformations, use `ST_Transform` instead.

If `srs_id` does not identify a spatial reference system listed in the catalog view `DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS`, then an exception condition is raised (SQLSTATE 38SU1).

Return types

- `INTEGER`, if an `srs_id` is not specified
- `db2gse.ST_Geometry`, if an `srs_id` is specified

Example

Two points are created in two different spatial reference systems. The ID of the spatial reference system that is associated with each point can be found by using the `ST_SrsId` function.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)

INSERT INTO sample_points
VALUES (1, ST_Point( 'point (80 180)', 0 ) )
INSERT INTO sample_points
VALUES (2, ST_Point( 'point (-74.21450127 + 42.03415094)', 1 ) )

SELECT id, ST_SRSId (geometry) SRSID
FROM sample_points
```

Results:

ID	SRSID
1	0
2	1

ST_SrsName function

The ST_SrsName function takes a geometry as an input parameter and returns the name of the spatial reference system in which the given geometry is represented.

If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►—db2gse.ST_SrsName—(*—geometry—*)—◄◄

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry for which the name of the spatial reference system is returned.

Return type

VARCHAR(128)

Example

Two points are created in different spatial reference systems. The ST_SrsName function is used to find out the name of the spatial reference system that is associated with each point.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry, ST_Point)
```

```
INSERT INTO sample_points
VALUES (1, ST_Point ('point (80 180)', 0) )
```

```
INSERT INTO sample_points
VALUES (2, ST_Point ('point (-74.21450127 + 42.03415094)', 1) )
```

```
SELECT id, ST_SrsName (geometry) SRSNAME
FROM sample_points
```

Results:

ID	SRSNAME
1	DEFAULT_SRS
2	NAD83_SRS_1

ST_StartPoint function

The ST_StartPoint function takes a curve as an input parameter and returns the point that is the first point of the curve.

The resulting point is represented in the spatial reference system of the given curve. This result is equivalent to the function call ST_PointN(*curve*, 1)

If the given curve is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

►►db2gse.ST_StartPoint(—*curve*—)◄◄

Parameters

curve A value of type ST_Curve or one of its subtypes that represents the geometry from which the first point is returned.

Return type

db2gse.ST_Point

Example

In the following example, two linestrings are added to the SAMPLE_LINES table. The first one is a linestring with X and Y coordinates. The second one is a linestring with X, Y, and Z coordinates. The ST_StartPoint function is used to return the first point in each linestring.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_lines (id INTEGER, line ST_LineString)
```

```
INSERT INTO sample_lines
VALUES (1, ST_LineString ('linestring
(10 10, 5 5, 0 0, 10 0, 5 5, 0 10)', 0))
```

```
INSERT INTO sample_lines
VALUES (1, ST_LineString ('linestring z
(0 0 4, 5 5 5, 10 10 6, 5 5 7, 0 0 8)', 0))
```

```
SELECT id, CAST( ST_AsText( ST_StartPoint( line ) ) AS VARCHAR(80))
START_POINT
FROM sample_lines
```

Results:

ID	START_POINT
1	POINT (10.00000000 10.00000000)
2	POINT Z (0.00000000 0.00000000 4.00000000)

ST_SymDifference function

The ST_SymDifference function takes two geometries as input parameters and returns the geometry that is the symmetrical difference of the two geometries. The symmetrical difference is the non-intersecting part of the two given geometries.

The resulting geometry is represented in the spatial reference system of the first geometry. The dimension of the returned geometry is the same as that of the input geometries. Both geometries must be of the same dimension.

If the second geometry is not represented in the same spatial reference system as the first geometry and uses the same underlying datum, it will be converted to the other spatial reference system.

If the geometries are equal, an empty geometry of type ST_Point is returned. If either geometry is null, then null is returned.

The resulting geometry is represented in the most appropriate spatial type. If it can be represented as a point, linestring, or polygon, then one of those types is used. Otherwise, the multipoint, multilinestring, or multipolygon type is used.

This function can also be called as a method.

Syntax

►►db2gse.ST_SymDifference(*—geometry1—*,*—geometry2—*)◄◄

Parameters

geometry1

A value of type ST_Geometry or one of its subtypes that represents the first geometry to compute the symmetrical difference with *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the second geometry to compute the symmetrical difference with *geometry1*.

Return type

db2gse.ST_Geometry

Examples

Example 1

This example illustrates the use of the ST_SymDifference function. The geometries are stored in the SAMPLE_GEOMS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geoms
VALUES
(1, ST_Geometry('polygon((10 10,10 20,20 20,20 10,10 10))',0))

INSERT INTO sample_geoms
VALUES
(2, ST_Geometry('polygon((30 30,30 50,50 50,50 30,30 30))',0))

INSERT INTO sample_geoms
VALUES
(3, ST_Geometry('polygon((40 40,40 60,60 60,60 40,40 40))',0))

INSERT INTO sample_geoms
VALUES
(4, ST_Geometry('linestring(70 70, 80 80)',0))

INSERT INTO sample_geoms
VALUES
(5,ST_Geometry('linestring(75 75,90 90)',0));
```

In the following examples, the results have been reformatted for readability. Your results will vary according to your display.

Example 2

This example uses ST_SymDifference to return the symmetric difference of two disjoint polygons in the SAMPLE_GEOMS table.

```
SELECT a.id, b.id,
       CAST (ST_AsText (ST_SymDifference (a.geometry, b.geometry) )
       AS VARCHAR(350) ) SYM_DIFF
FROM sample_geoms a, sample_geoms b
WHERE a.id = 1 AND b.id = 2
```

Results:

ID	ID	SYM_DIFF
1	2	MULTIPOLYGON (((10.00000000 10.00000000, 20.00000000 10.00000000, 20.00000000 20.00000000, 10.00000000 20.00000000, 10.00000000 10.00000000)), ((30.00000000 30.00000000, 50.00000000 30.00000000, 50.00000000 50.00000000, 30.00000000 50.00000000, 30.00000000 30.00000000)))

Example 3

This example uses ST_SymDifference to return the symmetric difference of two intersecting polygons in the SAMPLE_GEOMS table.

```
SELECT a.id, b.id,
       CAST (ST_AsText (ST_SymDifference (a.geometry, b.geometry) )
       AS VARCHAR(500) ) SYM_DIFF
FROM sample_geoms a, sample_geoms b
WHERE a.id = 2 AND b.id = 3
```

Results:

ID	ID	SYM_DIFF
2	3	MULTIPOLYGON (((40.00000000 50.00000000, 50.00000000 50.00000000, 50.00000000 40.00000000, 60.00000000 40.00000000, 60.00000000 60.00000000, 40.00000000 60.00000000, 40.00000000 50.00000000)), ((30.00000000 30.00000000, 50.00000000 30.00000000, 50.00000000 40.00000000, 40.00000000 40.00000000, 40.00000000 50.00000000, 30.00000000 50.00000000, 30.00000000 30.00000000)))

Example 4

This example uses ST_SymDifference to return the symmetric difference of two intersecting linestrings in the SAMPLE_GEOMS table.

```
SELECT a.id, b.id,
       CAST (ST_AsText (ST_SymDifference (a.geometry, b.geometry) )
       AS VARCHAR(350) ) SYM_DIFF
FROM sample_geoms a, sample_geoms b
WHERE a.id = 4 AND b.id = 5
```

Results:

ID	ID	SYM_DIFF
4	5	MULTILINESTRING ((70.00000000 70.00000000, 75.00000000 75.00000000), (80.00000000 80.00000000, 90.00000000 90.00000000))

ST_ToGeomColl function

The ST_ToGeomColl function takes a geometry as an input parameter and converts it to a geometry collection. The resulting geometry collection is represented in the spatial reference system of the given geometry.

If the specified geometry is empty, then it can be of any type. However, it is then converted to ST_Multipoint, ST_MultiLineString, or ST_MultiPolygon as appropriate.

If the specified geometry is not empty, then it must be of type ST_Point, ST_LineString, or ST_Polygon. These are then converted to ST_Multipoint, ST_MultiLineString, or ST_MultiPolygon respectively.

If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►►db2gse.ST_ToGeomColl(—*geometry*—)————►►

Parameter

geometry
A value of type ST_Geometry or one of its subtypes that represents the geometry that is converted to a geometry collection.

Return type

db2gse.ST_GeomCollection

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display

This example illustrates the use of the ST_ToGeomColl function.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geometries
VALUES (1, ST_Polygon ('polygon ((3 3, 4 6, 5 3, 3 3))', 1)),
      (2, ST_Point ('point (1 2)', 1))
```

In the following SELECT statement, the ST_ToGeomColl function is used to return geometries as their corresponding geometry collection subtypes.

```
SELECT id, CAST( ST_AsText( ST_ToGeomColl(geometry) )
AS VARCHAR(120) ) GEOM_COLL
FROM sample_geometries
```

Results:

ID	GEOM_COLL
1	MULTIPOLYGON (((3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000, 3.00000000 3.00000000)))
2	MULTIPOINT (1.00000000 2.00000000)

ST_ToLineString function

The ST_ToLineString function takes a geometry as an input parameter and converts it to a linestring. The resulting linestring is represented in the spatial reference system of the given geometry.

The given geometry must be empty or a linestring. If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_ToLineString—(—*geometry*—)—————►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry that is converted to a linestring.

A geometry can be converted to a linestring if it is empty or a linestring. If the conversion cannot be performed, then an exception condition is raised (SQLSTATE 38SUD).

Return type

db2gse.ST_LineString

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display

This example illustrates the use of the ST_ToLineString function.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geometries
VALUES (1, ST_Geometry ('linestring z (0 10 1, 0 0 3, 10 0 5)', 0)),
       (2, ST_Geometry ('point empty', 1) ),
       (3, ST_Geometry ('multipolygon empty', 1) )
```

In the following SELECT statement, the ST_ToLineString function is used to return linestrings converted to ST_LineString from the static type of ST_Geometry.

```
SELECT CAST( ST_AsText( ST_ToLineString(geometry) )
           AS VARCHAR(130) ) LINES
FROM sample_geometries
```

Results:

```
LINES
-----
LINESTRING Z ( 0.00000000 10.00000000 1.00000000, 0.00000000
               0.00000000 3.00000000, 10.00000000 0.00000000
               5.00000000)
LINESTRING EMPTY
LINESTRING EMPTY
```

ST_ToMultiLine function

The ST_ToMultiLine function takes a geometry as an input parameter and converts it to a multilinestring. The resulting multilinestring is represented in the spatial reference system of the given geometry.

The given geometry must be empty, a multilinestring, or a linestring. If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_ToMultiLine—(*—geometry—*)—►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry that is converted to a multilinestring.

A geometry can be converted to a multilinestring if it is empty, a linestring, or a multilinestring. If the conversion cannot be performed, then an exception condition is raised (SQLSTATE 38SUD).

Return type

db2gse.ST_MultiLineString

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display

This example illustrates the use of the ST_ToMultiLine function.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries (id INTEGER, geometry ST_Geometry)
```

```
INSERT INTO sample_geometries
VALUES (1, ST_Geometry ('multilinestring ((0 10 1, 0 0 3, 10 0 5),
                                         (23 43, 27 34, 35 12))', 0) ),
      (2, ST_Geometry ('linestring z (0 10 1, 0 0 3, 10 0 5)', 0) ),
      (3, ST_Geometry ('point empty', 1) ),
      (4, ST_Geometry ('multipolygon empty', 1) )
```

In the following SELECT statement, the ST_ToMultiLine function is used to return multilinestrings converted to ST_MultiLineString from the static type of ST_Geometry.

```
SELECT CAST( ST_AsText( ST_ToMultiLine(geometry) )
AS VARCHAR(130) ) LINES
FROM sample_geometries
```

Results:

LINES

MULTILINESTRING Z (0.00000000 10.00000000 1.00000000,


```

0.00000000 0.00000000 3.00000000,
10.00000000 0.00000000 5.00000000)
MULTILINESTRING EMPTY
MULTILINESTRING EMPTY

```

ST_ToMultiPoint function

The ST_ToMultiPoint function takes a geometry as an input parameter and converts it to a multipoint. The resulting multipoint is represented in the spatial reference system of the given geometry.

The given geometry must be empty, a point, or a multipoint. If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

```

►► db2gse.ST_ToMultiPoint(—geometry—) ◄◄

```

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry that is converted to a multipoint.

A geometry can be converted to a multipoint if it is empty, a point, or a multipoint. If the conversion cannot be performed, then an exception condition is raised (SQLSTATE 38SUD).

Return type

db2gse.ST_MultiPoint

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display

This example illustrates the use of the ST_ToMultiPoint function.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geometries
VALUES (1, ST_Geometry ('multipoint (0 0, 0 4)', 1) ),
      (2, ST_Geometry ('point (30 40)', 1) ),
      (3, ST_Geometry ('multipolygon empty', 1) )

```

In the following SELECT statement, the ST_ToMultiPoint function is used to return multipoints converted to ST_MultiPoint from the static type of ST_Geometry.

```

SELECT CAST( ST_AsText( ST_ToMultiPoint(geometry))
AS VARCHAR(62) ) MULTIPOINTS
FROM sample_geometries

```

Results:

MULTIPOINTS

```
-----  
MULTIPOINT ( 0.00000000 0.00000000, 0.00000000 4.00000000)  
MULTIPOINT ( 30.00000000 40.00000000)  
MULTIPOINT EMPTY
```

ST_ToMultiPolygon function

The ST_ToMultiPolygon function takes a geometry as an input parameter and converts it to a multipolygon. The resulting multipolygon is represented in the spatial reference system of the given geometry.

The given geometry must be empty, a polygon, or a multipolygon. If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_ToMultiPolygon—(—*geometry*—)—————►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry that is converted to a multipolygon.

A geometry can be converted to a multipolygon if it is empty, a polygon, or a multipolygon. If the conversion cannot be performed, then an exception condition is raised (SQLSTATE 38SUD).

Return type

db2gse.ST_MultiPolygon

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display

This example creates several geometries and then uses ST_ToMultiPolygon to return multipolygons.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse  
CREATE TABLE sample_geometries (id INTEGER, geometry ST_Geometry)
```

```
INSERT INTO sample_geometries  
VALUES (1, ST_Geometry ('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))', 1)),  
       (2, ST_Geometry ('point empty', 1)),  
       (3, ST_Geometry ('multipoint empty', 1))
```

In the following SELECT statement, the ST_ToMultiPolygon function is used to return multipolygons converted to ST_MultiPolygon from the static type of ST_Geometry.

```
SELECT CAST( ST_AsText( ST_ToMultiPolygon(geometry) )  
           AS VARCHAR(130) ) POLYGONS  
FROM sample_geometries
```

Results:

POLYGONS

```
MULTIPOLYGON (( 0.00000000 0.00000000, 5.00000000 0.00000000,  
                5.00000000 4.00000000, 0.00000000 4.00000000,  
                0.00000000 0.00000000))
```

MULTIPOLYGON EMPTY

MULTIPOLYGON EMPTY

ST_ToPoint function

The ST_ToPoint function takes a geometry as an input parameter and converts it to a point. The resulting point is represented in the spatial reference system of the given geometry.

The given geometry must be empty or a point. If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_ToPoint—(*—geometry—*)—◄◄

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry that is converted to a point.

A geometry can be converted to a point if it is empty or a point. If the conversion cannot be performed, then an exception condition is raised (SQLSTATE 38SUD).

Return type

db2gse.ST_Point

Example

This example creates three geometries in SAMPLE_GEOMETRIES and converts each to a point.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse  
CREATE TABLE sample_geometries (id INTEGER, geometry ST_Geometry)
```

```
INSERT INTO sample_geometries  
VALUES (1, ST_Geometry ('point (30 40)', 1) ),  
       (2, ST_Geometry ('linestring empty', 1) ),  
       (3, ST_Geometry ('multipolygon empty', 1) )
```

In the following SELECT statement, the ST_ToPoint function is used to return points converted to ST_Point from the static type of ST_Geometry.

```
SELECT CAST( ST_AsText( ST_ToPoint(geometry) ) AS VARCHAR(35) ) POINTS  
FROM sample_geometries
```

Results:

POINTS

POINT (30.00000000 40.00000000)
POINT EMPTY
POINT EMPTY

ST_ToPolygon function

The ST_ToPolygon takes a geometry as an input parameter and converts it to a polygon. The resulting polygon is represented in the spatial reference system of the given geometry.

The given geometry must be empty or a polygon. If the given geometry is null, then null is returned.

This function can also be called as a method.

Syntax

►►—db2gse.ST_ToPolygon—(—*geometry*—)—————►►

Parameter

geometry

A value of type ST_Geometry or one of its subtypes that represents the geometry that is converted to a polygon.

A geometry can be converted to a polygon if it is empty or a polygon. If the conversion cannot be performed, then an exception condition is raised (SQLSTATE 38SUD).

Return type

db2gse.ST_Polygon

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display

This example creates three geometries in SAMPLE_GEOMETRIES and converts each to a polygon.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries (id INTEGER, geometry ST_Geometry)
```

```
INSERT INTO sample_geometries
VALUES (1, ST_Geometry ('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))', 1) ),
      (2, ST_Geometry ('point empty', 1) ),
      (3, ST_Geometry ('multipolygon empty', 1) )
```

In the following SELECT statement, the ST_ToPolygon function is used to return polygons converted to ST_Polygon from the static type of ST_Geometry.

```
SELECT CAST( ST_AsText( ST_ToPolygon(geometry) ) AS VARCHAR(130) ) POLYGONS
FROM sample_geometries
```

Results:

POLYGONS

```
POLYGON (( 0.00000000 0.00000000, 5.00000000 0.00000000,  
          5.00000000 4.00000000,0.00000000 4.00000000,  
          0.00000000 0.00000000))
```

POLYGON EMPTY

POLYGON EMPTY

ST_Touches function

The ST_Touches function takes two geometries as input parameters and returns 1 if the given geometries spatially touch. Otherwise, 0 (zero) is returned.

Two geometries touch if the interiors of both geometries do not intersect, but the boundary of one of the geometries intersects with either the boundary or the interior of the other geometry.

If the second geometry is not represented in the same spatial reference system as the first geometry, it will be converted to the other spatial reference system.

If both of the given geometries are points or multipoints, or if any of the given geometries is null or empty, then null is returned.

Syntax

►►—db2gse.ST_Touches—(—*geometry1*—,—*geometry2*—)—►►

Parameters

geometry1

A value of type ST_Geometry or one of its subtypes that represents the geometry that is to be tested to touch *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that represents the geometry that is to be tested to touch *geometry1*.

Return type

INTEGER

Example

Several geometries are added to the SAMPLE_GEOMS table. The ST_Touches function is then used to determine which geometries touch each other.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse  
CREATE TABLE sample_geoms (id INTEGER, geometry ST_Geometry)
```

```
INSERT INTO sample_geoms  
VALUES (1, ST_Geometry('polygon ( (20 30, 30 30, 30 40, 20 40, 20 30) )' , 0) )
```

```
INSERT INTO sample_geoms  
VALUES (2, ST_Geometry('polygon ( (30 30, 30 50, 50 50, 50 30, 30 30) )' ,0) )
```

```
INSERT INTO sample_geoms  
VALUES (3, ST_Geometry('polygon ( (40 40, 40 60, 60 60, 60 40, 40 40) )' , 0) )
```

```
INSERT INTO sample_geoms
VALUES (4, ST_Geometry('linestring( 60 60, 70 70 )' , 0) )

INSERT INTO sample_geoms
VALUES (5, ST_Geometry('linestring( 30 30, 60 60 )' , 0) )

SELECT a.id, b.id, ST_Touches (a.geometry, b.geometry) TOUCHES
FROM sample_geoms a, sample_geoms b
WHERE b.id >= a.id
```

Results:

ID	ID	TOUCHES
	1	1
	1	2
	1	3
	1	4
	1	5
	2	2
	2	3
	2	4
	2	5
	3	3
	3	4
	3	5
	4	4
	4	5
	5	5

ST_Transform function

The ST_Transform function takes a geometry and a spatial reference system identifier as input parameters and transforms the geometry to be represented in the given spatial reference system. Projections and conversions between different coordinate systems are performed and the coordinates of the geometries are adjusted accordingly.

The geometry can be converted to the specified spatial reference system only if the geometry's current spatial reference system is based in the same geographic coordinate system as the specified spatial reference system. If either the geometry's current spatial reference system or the specified spatial reference system is based on a projected coordinate system, a reverse projection is performed to determine the geographic coordinate system that underlies the projected one.

If the given geometry is null, then null will be returned.

This function can also be called as a method.

Syntax

```
►►—db2gse.ST_Transform—(—geometry—,—srs_id—)—►►
```

Parameters

geometry
A value of type ST_Geometry or one of its subtypes that represents the geometry that is transformed to the spatial reference system identified by *srs_id*.

srs_id A value of type INTEGER that identifies the spatial reference system for the resulting geometry.

If the transformation to the specified spatial reference system cannot be performed because the current spatial reference system of *geometry* is not compatible with the spatial reference system identified by *srs_id*, then an exception condition is raised (SQLSTATE 38SUC).

If *srs_id* does not identify a spatial reference system listed in the catalog view DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS, then an exception condition is raised (SQLSTATE 38SU1).

Return type

db2gse.ST_Geometry

Examples

Example 1

The following example illustrates the use of ST_Transform to convert a geometry from one spatial reference system to another.

First, the state plane spatial reference system with an ID of 3 is created using a call to db2se.

```
db2se create_srs SAMP_DB
-srsId 3 -srsName z3101a -xOffset 0 -yOffset 0 -xScale 1 -yScale 1
- coordsysName NAD_1983_StatePlane_New_York_East_FIPS_3101_Feet
```

Then, points are added to:

- The SAMPLE_POINTS_SP table in state plane coordinates using that spatial reference system.
- The SAMPLE_POINTS_LL table using coordinates specified in latitude and longitude.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points_sp (id INTEGER, geometry ST_Point)
CREATE TABLE sample_points_ll (id INTEGER, geometry ST_Point)
```

```
INSERT INTO sample_points_sp
VALUES (12457, ST_Point('point ( 567176.0 1166411.0)', 3) )
```

```
INSERT INTO sample_points_sp
VALUES (12477, ST_Point('point ( 637948.0 1177640.0)', 3) )
```

```
INSERT INTO sample_points_ll
VALUES (12457, ST_Point('point ( -74.22371600 42.03498700)', 1) )
```

```
INSERT INTO sample_points_ll
VALUES (12477, ST_Point('point ( -73.96293200 42.06487900)', 1) )
```

Then the ST_Transform function is used to convert the geometries.

Example 2

This example converts points that are in latitude and longitude coordinates to state plane coordinates.

```
SELECT id, CAST( ST_AsText( ST_Transform( geometry, 3) )
AS VARCHAR(100) ) STATE_PLANE
FROM sample_points_ll
```

Results:

ID	STATE_PLANE
12457	POINT (567176.00000000 1166411.00000000)
12477	POINT (637948.00000000 1177640.00000000)

Example 3

This example converts points that are in state plane coordinates to latitude and longitude coordinates.

```
SELECT id, CAST( ST_AsText( ST_Transform( geometry, 1 ) )
AS VARCHAR(100) ) LAT_LONG
FROM sample_points_sp
```

Results:

ID	LAT_LONG
12457	POINT (-74.22371500 42.03498800)
12477	POINT (-73.96293100 42.06488000)

ST_Union function

The ST_Union function takes two geometries as input parameters and returns the geometry that is the union of the given geometries. The resulting geometry is represented in the spatial reference system of the first geometry.

Both geometries must be of the same dimension. If any of the two given geometries is null, null is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry and uses the same underlying datum, it will be converted to the other spatial reference system.

The resulting geometry is represented in the most appropriate spatial type. If it can be represented as a point, linestring, or polygon, then one of those types is used. Otherwise, the multipoint, multilinestring, or multipolygon type is used.

This function can also be called as a method.

Syntax

►►db2gse.ST_Union(—*geometry1*—,—*geometry2*—)◄◄

Parameters

geometry1

A value of type ST_Geometry or one of its subtypes that is combined with *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that is combined with *geometry1*.

Return type

db2gse.ST_Geometry

Examples

Example 1

The following SQL statements create and populate the SAMPLE_GEOMS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geoms (id INTEGER, geometry, ST_Geometry)
```

```
INSERT INTO sample_geoms
VALUES (1, ST_Geometry( 'polygon
((10 10, 10 20, 20 20, 20 10, 10 10) )', 0))
```

```
INSERT INTO sample_geoms
VALUES (2, ST_Geometry( 'polygon
((30 30, 30 50, 50 50, 50 30, 30 30) )', 0))
```

```
INSERT INTO sample_geoms
VALUES (3, ST_Geometry( 'polygon
((40 40, 40 60, 60 60, 60 40, 40 40) )', 0))
```

```
INSERT INTO sample_geoms
VALUES (4, ST_Geometry('linestring (70 70, 80 80)', 0))
```

```
INSERT INTO sample_geoms
VALUES (5, ST_Geometry('linestring (80 80, 100 70)', 0))
```

In the following examples, the results have been reformatted for readability. Your results will vary according to your display.

Example 2

This example finds the union of two disjoint polygons.

```
SELECT a.id, b.id, CAST ( ST_AsText( ST_Union( a.geometry, b.geometry) )
AS VARCHAR (350) ) UNION
FROM sample_geoms a, sample_geoms b
WHERE a.id = 1 AND b.id = 2
```

Results:

ID	ID	UNION
1	2	MULTIPOLYGON (((10.00000000 10.00000000, 20.00000000 10.00000000, 20.00000000 20.00000000, 10.00000000 20.00000000, 10.00000000 10.00000000)))

Example 3

This example finds the union of two intersecting polygons.

```
SELECT a.id, b.id, CAST ( ST_AsText( ST_Union(a.geometry, b.geometry))
AS VARCHAR (250)) UNION
FROM sample_geoms a, sample_geoms b
WHERE a.id = 2 AND b.id = 3
```

Results:

ID	ID	UNION
2	3	POLYGON ((30.00000000 30.00000000, 50.00000000 30.00000000, 50.00000000 40.00000000, 60.00000000 40.00000000, 60.00000000 60.00000000, 40.00000000 60.00000000, 40.00000000 50.00000000, 30.00000000 50.00000000, 30.00000000 30.00000000))

Example 4

Find the union of two linestrings.

```
SELECT a.id, b.id, CAST ( ST_AsText( ST_Union( a.geometry, b.geometry) )
      AS VARCHAR (250) ) UNION
FROM sample_geoms a, sample_geoms b
WHERE a.id = 4 AND b.id = 5
```

Results:

ID	ID	UNION
4	5	MULTILINESTRING((70.00000000 70.00000000,80.00000000 80.00000000), (80.00000000 80.00000000,100.00000000 70.00000000))

ST_Within function

Use the ST_Within function to determine whether one geometry is completely within another geometry.

Syntax

►►—db2gse.ST_Within—(—*geometry1*—,—*geometry2*—)——————►◄

Parameters

geometry1

A value of type ST_Geometry or one of its subtypes that is to be tested to be fully within *geometry2*.

geometry2

A value of type ST_Geometry or one of its subtypes that is to be tested to be fully within *geometry1*.

Return type

INTEGER

Usage

ST_Within takes two geometries as input parameters and returns 1 if the first geometry is completely within the second geometry. Otherwise, 0 (zero) is returned.

If any of the given geometries is null or is empty, null is returned.

If the second geometry is not represented in the same spatial reference system as the first geometry and uses the same underlying datum, it will be converted to the other spatial reference system.

ST_Within performs the same logical operation that ST_Contains performs with the parameters reversed. ST_Within returns the exact opposite result of ST_Contains.

The ST_Within function pattern matrix states that the interiors of both geometries must intersect, and that the interior or boundary of the primary geometry (*geometry a*) must not intersect the exterior of the secondary (*geometry b*). The asterisk (*) indicates that all other intersections do not matter.

Table 32. Matrix for ST_Within

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

Examples

Figure 20 on page 396 shows examples of ST_Within:

- A point geometry is within a multipoint geometry when its interior intersects one of the points in the second geometry.
- A multipoint geometry is within a multipoint geometry when the interiors of all points intersect the second geometry.
- A multipoint geometry is within a polygon geometry when all of the points are either on the boundary of the polygon or in the interior of the polygon.
- A point geometry is within a linestring geometry when all of the points are within the second geometry. In Figure 20 on page 396, the point is not within the linestring because its interior does not intersect the linestring; however, the multipoint geometry is within the linestring because all of its points intersect the interior of the linestring.
- A linestring geometry is within another linestring geometries when all of its points intersect the second geometry.
- A point geometry is not within a polygon geometry because its interior does not intersect the boundary or interior of the polygon.
- A linestring geometry is within a polygon geometry when all of its points intersect either the boundary or interior of the polygon.
- A polygon geometry is within a polygon geometry when all of its points intersect either the boundary or interior of the polygon.

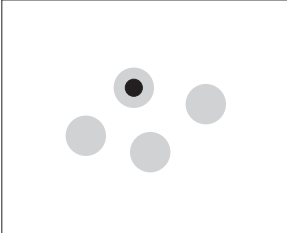
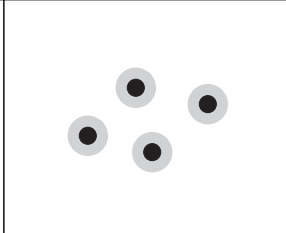
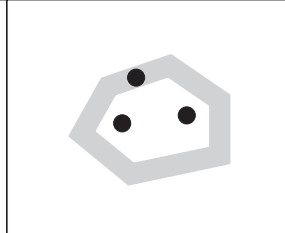
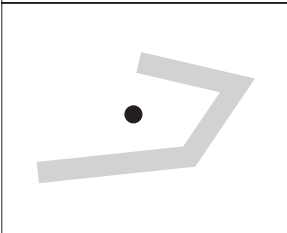
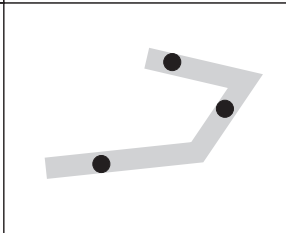
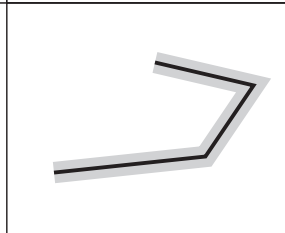
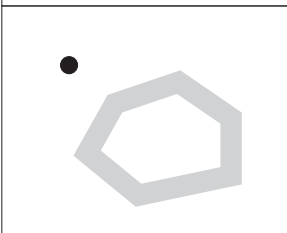
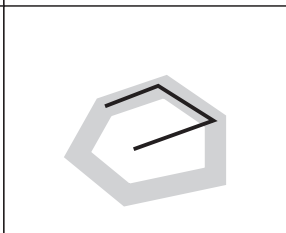
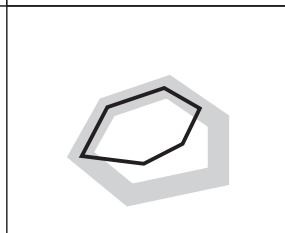
		
point / multipoint	multipoint / multipoint	multipoint / polygon
		
point / linestring	multipoint / linestring	linestring / linestring
		
point / polygon	linestring / polygon	polygon / polygon

Figure 20. ST_Within function

Example 1

This example illustrates use of the ST_Within function. Geometries are created and inserted into three tables, SAMPLE_POINTS, SAMPLE_LINES, and SAMPLE_POLYGONS.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)
CREATE TABLE sample_lines (id INTEGER, line ST_LineString)
CREATE TABLE sample_polygons (id INTEGER, geometry ST_Polygon)

INSERT INTO sample_points (id, geometry)
VALUES (1, ST_Point (10, 20, 1) ),
      (2, ST_Point ('point (41 41)', 1) )

INSERT INTO sample_lines (id, line)
VALUES (10, ST_LineString ('linestring (1 10, 3 12, 10 10)', 1) ),
      (20, ST_LineString ('linestring (50 10, 50 12, 45 10)', 1) )

INSERT INTO sample_polygons (id, geometry)
VALUES (100, ST_Polygon ('polygon (( 0 0, 0 40, 40 40, 40 0, 0 0))', 1) )
```

Example 2

This example finds points from the SAMPLE_POINTS table that are in the polygons in the SAMPLE_POLYGONS table.

```
SELECT a.id POINT_ID_WITHIN_POLYGONS
FROM sample_points_a, sample_polygons b
WHERE ST_Within( b.geometry, a.geometry) = 0
```

```
Results:
POINT_ID_WITHIN_POLYGONS
-----
2
```

Example 3

This example finds linestrings from the SAMPLE_LINES table that are in the polygons in the SAMPLE_POLYGONS table.

```
SELECT a.id LINE_ID_WITHIN_POLYGONS
FROM sample_lines a, sample_polygons b
WHERE ST_Within( b.geometry, a.geometry) = 0
```

```
Results:
LINE_ID_WITHIN_POLYGONS
-----
1
```

ST_WKBToSQL function

The ST_WKBToSQL function takes a well-known binary representation of a geometry and returns the corresponding geometry. The spatial reference system with the identifier 0 (zero) is used for the resulting geometry.

If the given well-known binary representation is null, then null is returned.

ST_WKBToSQL(*wkb*) gives the same result as ST_Geometry(*wkb*,0). Using the ST_Geometry function is recommended over using ST_WKBToSQL because of its flexibility: ST_Geometry takes additional forms of input as well as the well-known binary representation.

Syntax

```
►►—db2gse.ST_WKBToSQL—(—wkb—)—————►►
```

Parameter

wkb A value of type BLOB(2G) that contains the well-known binary representation of the resulting geometry.

Return type

db2gse.ST_Geometry

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates use of the ST_WKBToSQL function. First, geometries are stored in the SAMPLE_GEOMETRIES table in its GEOMETRY column. Then, their well-known binary representations are stored in the WKB column using the ST_AsBinary function in the UPDATE statement. Finally, the ST_WKBToSQL function is used to return the coordinates of the geometries in the WKB column.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries
(id INTEGER, geometry ST_Geometry, wkb BLOB(32K) )
```

```

INSERT INTO sample_geometries (id, geometry)
VALUES (10, ST_Point ( 'point (44 14)', 0 ) ),
      (11, ST_Point ( 'point (24 13)', 0 ) ),
      (12, ST_Polygon ('polygon ((50 20, 50 40, 70 30, 50 20))', 0 ) )
UPDATE sample_geometries AS temp_correlated
SET wkb = ST_AsBinary(geometry)
WHERE id = temp_correlated.id

```

Use this SELECT statement to see the geometries in the WKB column.

```

SELECT id, CAST( ST_AsText( ST_WKBToSQL(wkb) ) AS VARCHAR(120) ) GEOMETRIES
FROM sample_geometries

```

Results:

ID	GEOMETRIES
10	POINT (44.00000000 14.00000000)
11	POINT (24.00000000 13.00000000)
12	POLYGON ((50.00000000 20.00000000, 70.00000000 30.00000000, 50.00000000 40.00000000, 50.00000000 20.00000000))

ST_WKTToSQL function

The ST_WKTToSQL function takes a well-known text representation of a geometry and returns the corresponding geometry.

The spatial reference system with the identifier 0 (zero) is used for the resulting geometry.

If the given well-known text representation is null, then null is returned.

ST_WKTToSQL(*wkt*) gives the same result as ST_Geometry(*wkt*,0). The use of the ST_Geometry function gives you more flexibility that the ST_WKTToSQL function because ST_Geometry takes additional forms of input as well as the well-known text representation.

Syntax

►►—db2gse.ST_WKTToSQL—(—*wkt*—)—————►►

Parameter

wkt A value of type CLOB(2G) that contains the well-known text representation of the resulting geometry.

Return type

db2gse.ST_Geometry

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how ST_WKTToSQL can create and insert geometries using their well-known text representations.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_geometries (id INTEGER, geometry ST_Geometry)

INSERT INTO sample_geometries
VALUES (10, ST_WKTToSQL( 'point (44 14)' ) ),
      (11, ST_WKTToSQL( 'point (24 13)' ) ),
      (12, ST_WKTToSQL( 'polygon ((50 20, 50 40, 70 30, 50 20))' ) )

```

This SELECT statement returns the geometries that have been inserted.

```

SELECT id, CAST( ST_AsText(geometry) AS VARCHAR(120) ) GEOMETRIES
FROM sample_geometries

```

Results:

ID	GEOMETRIES
10	POINT (44.00000000 14.00000000)
11	POINT (24.00000000 13.00000000)
12	POLYGON ((50.00000000 20.00000000, 70.00000000 30.00000000, 50.00000000 40.00000000, 50.00000000 20.00000000))

ST_X function

The ST_X function takes a point as an input parameter and returns its X coordinate. You can optionally indicate an X coordinate as input parameter in addition to the point and the function returns the point itself with its X coordinate set to the given value.

If the given point is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

```

db2gse.ST_X(point [, x_coordinate ])

```

Parameters

point A value of type ST_Point for which the X coordinate is returned or modified.

x_coordinate
A value of type DOUBLE that represents the new X coordinate for *point*.

Return types

- DOUBLE, if *x_coordinate* is not specified
- db2gse.ST_Point, if *x_coordinate* is specified

Examples

Example 1

This example illustrates use of the ST_X function. Geometries are created and inserted into the SAMPLE_POINTS table.

```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)

INSERT INTO sample_points (id, geometry)

```

```
VALUES (1, ST_Point (2, 3, 32, 5, 1) ),
       (2, ST_Point (4, 5, 20, 4, 1) ),
       (3, ST_Point (3, 8, 23, 7, 1) )
```

Example 2

This example finds the X coordinates of the points in the table.

```
SELECT id, ST_X (geometry) X_COORD
FROM sample_points
```

Results:

ID	X_COORD
1	+2.000000000000000E+000
2	+4.000000000000000E+000
3	+3.000000000000000E+000

Example 3

This example returns a point with its X coordinate set to 40.

```
SELECT id, CAST( ST_AsText( ST_X (geometry, 40)) AS VARCHAR(60) )
      X_40
FROM sample_points
WHERE id=3
```

Results:

ID	X_40
3	POINT ZM (40.00000000 8.00000000 23.00000000 7.00000000)

ST_Y function

The ST_Y function takes a point as an input parameter and returns its Y coordinate. You can optionally indicate a Y coordinate as input parameter in addition to the point and the function returns the point itself with its Y coordinate set to the given value.

If the given point is null or is empty, then null is returned.

This function can also be called as a method.

Syntax

```
db2gse.ST_Y(—point— [, —y_coordinate—])
```

Parameters

point A value of type ST_Point for which the Y coordinate is returned or modified.

y_coordinate
A value of type DOUBLE that represents the new Y coordinate for *point*.

Return types

- DOUBLE, if *y_coordinate* is not specified
- db2gse.ST_Point, if *y_coordinate* is specified

Examples

Example 1

This example illustrates use of the ST_Y function. Geometries are created and inserted into the SAMPLE_POINTS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)

INSERT INTO sample_points (id, geometry)
VALUES (1, ST_Point (2, 3, 32, 5, 1) ),
       (2, ST_Point (4, 5, 20, 4, 1) ),
       (3, ST_Point (3, 8, 23, 7, 1) )
```

Example 2

This example finds the Y coordinates of the points in the table.

```
SELECT id, ST_Y (geometry) Y_COORD
FROM sample_points
```

Results:

ID	Y_COORD
1	+3.000000000000000E+000
2	+5.000000000000000E+000
3	+8.000000000000000E+000

Example 3

This example returns a point with its Y coordinate set to 40.

```
SELECT id, CAST( ST_AsText( ST_Y (geometry, 40)) AS VARCHAR(60) )
       Y_40
FROM sample_points
WHERE id=3
```

Results:

ID	Y_40
3	POINT ZM (3.00000000 40.00000000 23.00000000 7.00000000)

ST_Z function

The ST_Z function takes a point as an input parameter and returns its Y coordinate. You can optionally indicate a Z coordinate as input parameter in addition to the point and the function returns the point itself with its Y coordinate set to the given value.

ST_Z takes either:

- A point as an input parameter and returns its Z coordinate
- A point and a Z coordinate and returns the point itself with its Z coordinate set to the given value, even if the specified point has no existing Z coordinate.

If the specified Z coordinate is null, then the Z coordinate is removed from the point.

If the specified point is null or empty, then null is returned.

This function can also be called as a method.

Syntax

►► db2gse.ST_Z (*point* , *z_coordinate*) ►►

Parameters

point A value of type ST_Point for which the Z coordinate is returned or modified.

z_coordinate

A value of type DOUBLE that represents the new Z coordinate for *point*.

If *z_coordinate* is null, then the Z coordinate is removed from *point*.

Return types

- DOUBLE, if *z_coordinate* is not specified
- db2gse.ST_Point, if *z_coordinate* is specified

Examples

Example 1

This example illustrates use of the ST_Z function. Geometries are created and inserted into the SAMPLE_POINTS table.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)
```

```
INSERT INTO sample_points (id, geometry)
VALUES (1, ST_Point (2, 3, 32, 5, 1) ),
       (2, ST_Point (4, 5, 20, 4, 1) ),
       (3, ST_Point (3, 8, 23, 7, 1) )
```

Example 2

This example finds the Z coordinates of the points in the table.

```
SELECT id, ST_Z (geometry) Z_COORD
FROM sample_points
```

Results:

ID	Z_COORD
1	+3.200000000000000E+001
2	+2.000000000000000E+001
3	+2.300000000000000E+001

Example 3

This example returns a point with its Z coordinate set to 40.

```
SELECT id, CAST( ST_AsText( ST_Z (geometry, 40)) AS VARCHAR(60) )
       Z_40
FROM sample_points
WHERE id=3
```

Results:

ID	Z_40
3	POINT ZM (3.00000000 8.00000000 40.00000000 7.00000000)

Union aggregate functions

A union aggregate is the combination of the ST_BuildUnionAggr and ST_GetAggrResult functions. Use this combination to aggregate a column of geometries in a table to single geometry by constructing the union.

If all of the geometries to be combined in the union are null , then null is returned. If each of the geometries to be combined in the union are either null or are empty, then an empty geometry of type ST_Point is returned.

The ST_BuildUnionAggr function can also be called as a method.

Syntax

```
►►db2gse.ST_GetAggrResult(—————)—————►
►MAX(—db2sge.ST_BuildUnionAggr(—geometries—)—)—)—————►◄
```

Parameters

geometries

A column in a table that has a type of ST_Geometry or one of its subtypes and represents all the geometries that are to be combined into a union.

Return type

db2gse.ST_Geometry

Restrictions

You cannot construct the union aggregate of a spatial column in a table in any of the following situations:

- In partitioned database environments
- If a GROUP BY clause is used in the select
- If you use a function other than the DB2 aggregate function MAX

Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

This example illustrates how a union aggregate can be used to combine a set of points into multipoints. Several points are added to the SAMPLE_POINTS table. The ST_GetAggrResult and ST_BuildUnionAggr functions are used to construct the union of the points.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
CREATE TABLE sample_points (id INTEGER, geometry ST_Point)
```

```
INSERT INTO sample_points
VALUES (1, ST_Point (2, 3, 1) )
INSERT INTO sample_points
VALUES (2, ST_Point (4, 5, 1) )
INSERT INTO sample_points
VALUES (3, ST_Point (13, 15, 1) )
INSERT INTO sample_points
VALUES (4, ST_Point (12, 5, 1) )
```

```

INSERT INTO sample_points
VALUES (5, ST_Point (23, 2, 1) )
INSERT INTO sample_points
VALUES (6, ST_Point (11, 4, 1) )

SELECT CAST (ST_AsText(
    ST_GetAggrResult( MAX( ST_BuildUnionAggregate (geometry) ) ))
AS VARCHAR(160)) POINT_AGGREGATE
FROM sample_points

```

Results:

POINT_AGGREGATE

```

-----
MULTIPOINT ( 2.00000000 3.00000000, 4.00000000 5.00000000,
             11.00000000 4.00000000, 12.00000000 5.00000000,
             13.00000000 15.00000000, 23.00000000 2.00000000)

```

Chapter 19. Transform groups

Spatial Extender provides four transform groups that are used to transfer geometries between the DB2 server and a client application.

These transform groups accommodate the following data exchange formats:

- Well-known text (WKT) representation
- Well-known binary (WKB) representation
- ESRI shape representation
- Geography Markup Language (GML)

When data is retrieved from a table that contains a spatial column, the data from the spatial column is transformed to either a CLOB(2G) or a BLOB(2G) data type, depending on whether you indicated whether the transformed data was to be represented in binary or text format. You can also use the transform groups to transfer spatial data to the database.

To select which transform group is to be used when the data is transferred, use the SET CURRENT DEFAULT TRANSFORM GROUP statement to modify the DB2 special register CURRENT DEFAULT TRANSFORM GROUP. DB2 uses the value of this special register to determine which transform functions must be called to perform the necessary conversions.

Transform groups can simplify application programming. Instead of explicitly using conversion functions in the SQL statements, you can specify a transform group, which lets DB2 handle that task.

ST_WellKnownText transform group

Use the ST_WellKnownText transform group to transmit data to and from DB2 using the well-known text (WKT) representation.

When binding out a value from the database server to the client, the same function provided by ST_AsText() is used to convert a geometry to the WKT representation. When the well-known text representation of a geometry is transferred to the database server, the ST_Geometry(CLOB) function is implicitly used to perform the conversions to an ST_Geometry value. Using the transform group for binding in values to DB2 causes the geometries to be represented in the spatial reference system with the numeric identifier 0 (zero).

Examples

In the following examples, the lines of results have been reformatted for readability. The spacing in your results might vary according to your online display.

Example 1

The following SQL script shows how to use the ST_WellKnownText transform group to retrieve a geometry in its well-known text representation without using the explicit ST_AsText function.

```

CREATE TABLE transforms_sample (
    id    INTEGER,
    geom  db2gse.ST_Geometry)

INSERT
    INTO    transforms_sample
    VALUES (1, db2gse.ST_LineString('linestring
    (100 100, 200 100)', 0))

SET CURRENT DEFAULT TRANSFORM GROUP = ST_WellKnownText

SELECT id, geom
FROM    transforms_sample
WHERE   id = 1

```

Results:

```

ID    GEOM
---  -----
1    LINESTRING ( 100.000000000 100.000000000, 200.000000000 100.000000000)

```

Example 2

The following C code shows how to use the ST_WellKnownText transform group to insert geometries using the explicit ST_Geometry function for the host-variable wkt_buffer, which is of type CLOB and contains the well-known text representation of the point (10 10) that is to be inserted.

```

EXEC SQL BEGIN DECLARE SECTION;
    sqlint32 id = 0;
    SQL TYPE IS db2gse.ST_Geometry AS CLOB(1000) wkt_buffer;
EXEC SQL END DECLARE SECTION;

// set the transform group for all subsequent SQL statements
EXEC SQL
    SET CURRENT DEFAULT TRANSFORM GROUP = ST_WellKnownText;

id = 100;
strcpy(wkt_buffer.data, "point ( 10 10 )");
wkt_buffer.length = strlen(wkt_buffer.data);

// insert point using WKT into column of type ST_Geometry
EXEC SQL
    INSERT
    INTO    transforms_sample(id, geom)
    VALUES (:id, :wkt_buffer);

```

ST_WellKnownBinary transform group

Use the ST_WellKnownBinary transform group to transmit data to and from DB2 using the well-known binary (WKB) representation.

When binding out a value from the database server to the client, the same function provided by ST_AsBinary() is used to convert a geometry to the WKB representation. When the well-known binary representation of a geometry is transferred to the database server, the ST_Geometry(BLOB) function is used implicitly to perform the conversions to an ST_Geometry value. Using the transform group for binding in values to DB2 causes the geometries to be represented in the spatial reference system with the numeric identifier 0 (zero).

Example

In the following examples, the lines of results have been reformatted for readability. The spacing in your results might vary according to your online display.

Example 1

The following SQL script shows how to use the ST_WellKnownBinary transform group to retrieve a geometry in its well-known binary representation without using the explicit ST_AsBinary function.

```
CREATE TABLE transforms_sample (
  id INTEGER,
  geom db2gse.ST_Geometry)

INSERT
  INTO transforms_sample
  VALUES ( 1, db2gse.ST_Polygon('polygon ((10 10, 20 10, 20 20,
    10 20, 10 10))', 0))

SET CURRENT DEFAULT TRANSFORM GROUP = ST_WellKnownBinary

SELECT id, geom
  FROM transforms_sample
 WHERE id = 1
```

Results:

ID	GEOM
1	x'01030000000100000000500000000000000002440000000000024400000000000002440000000000003440000000000034400000000000034400000000000034400000000000002440000000000002440000000000002440'

Example 2

The following C code shows how to use the ST_WellKnownBinary transform group for inserting geometries using the explicit ST_Geometry function for the host-variable wkb_buffer, which is of type BLOB and contains the well-known binary representation of a geometry that is to be inserted.

```
EXEC SQL BEGIN DECLARE SECTION;
    sqlint32 id = 0;
    SQL TYPE IS db2gse.ST_Geometry AS BLOB(1000) wkb_buffer;
EXEC SQL END DECLARE SECTION;

// set the transform group for all subsequent SQL statements
EXEC SQL
    SET CURRENT DEFAULT TRANSFORM GROUP = ST_WellKnownBinary;

// initialize host variables
...
// insert geometry using WKB into column of type ST_Geometry

EXEC SQL
    INSERT
        INTO transforms_sample(id, geom)
        VALUES ( :id, :wkb_buffer );
```

ST_Shape transform group

Use the ST_Shape transform group to transmit data to and from DB2 using the ESRI shape representation.

When binding out a value from the database server to the client, the same function provided by `ST_AsShape()` is used to convert a geometry to its shape representation. When transferring the shape representation of a geometry to the database server, the `ST_Geometry(BLOB)` function is used implicitly to perform the conversions to an `ST_Geometry` value. Using the transform group for binding in values to DB2 causes the geometries to be represented in the spatial reference system with the numeric identifier 0 (zero).

Examples

In the following examples, the lines of results have been reformatted for readability. The spacing in your results might vary according to your online display.

Example 1

The following SQL script shows how the ST_Shape transform group can be used to retrieve a geometry in its shape representation without using the explicit ST_AsShape function.

```
CREATE TABLE transforms_sample(
  id INTEGER,
  geom db2gse.ST_Geometry)

INSERT
  INTO transforms_sample
  VALUES ( 1, db2gse.ST_Point(20.0, 30.0, 0) )

SET CURRENT DEFAULT TRANSFORM GROUP = ST_Shape

SELECT id, geom
  FROM transforms_sample
 WHERE id = 1
```

Results:

[illegible]

Example 2

The following C code shows how to use the ST_Shape transform group to insert geometries using the explicit ST_Geometry function for the host-variable shape_buffer, which is of type BLOB and contains the shape representation of a geometry that is to be inserted.

```
EXEC SQL BEGIN DECLARE SECTION;
    sqlint32 id = 0;
    SQL TYPE IS db2gse.ST_Geometry AS BLOB(1000) shape_buffer;
EXEC SQL END DECLARE SECTION;

// set the transform group for all subsequent SQL statements
EXEC SQL
    SET CURRENT DEFAULT TRANSFORM GROUP = ST_Shape;

// initialize host variables
...

SET CURRENT DEFAULT TRANSFORM GROUP = ST_Shape;
```



```
// insert geometry using shape representation
// into column of type ST_Geometry
EXEC SQL
    INSERT
    INTO transforms_sample(id, geom)
    VALUES ( :id, :shape_buffer );
```

ST_GML transform group

Use the ST_GML transform group to transmit data to and from DB2 using the geography markup language (GML).

When binding out a value from the database server to the client, the same function provided by ST_AsGML() is used to convert a geometry to its GML representation. When the GML representation of a geometry is transferred to the database server, the ST_Geometry(CLOB) function is used implicitly to perform the conversions to an ST_Geometry value. Using the transform group for binding in values to DB2 causes the geometries to be represented in the spatial reference system with the numeric identifier 0 (zero).

Examples

In the following examples, the lines of results have been reformatted for readability. The spacing in your results might vary according to your online display.

Example 1

The following SQL script shows how the ST_GML transform group can be used to retrieve a geometry in its GML representation without using the explicit ST_AsGML function.

```
CREATE TABLE transforms_sample (
    id INTEGER,
    geom db2gse.ST_Geometry)

INSERT
    INTO transforms_sample
    VALUES ( 1, db2gse.ST_Geometry('multipoint z (10 10
    3, 20 20 4, 15 20 30)', 0) )
    SET CURRENT DEFAULT TRANSFORM GROUP = ST_GML

SELECT id, geom
FROM transforms_sample
WHERE id = 1
```

Results:

ID	GEOM
1	<pre><gml:MultiPoint srsName=UNSPECIFIED><gml:PointMember> <gml:Point><gml:coord><gml:X>10</gml:X> <gml:Y>10</gml:Y><gml:Z>3</gml:Z> </gml:coord></gml:Point></gml:PointMember> <gml:PointMember><gml:Point><gml:coord> <gml:X>20</gml:X><gml:Y>20</gml:Y> <gml:Z>4</gml:Z></gml:coord></gml:Point> </gml:PointMember><gml:PointMember><gml:Point> <gml:coord><gml:X>15</gml:X><gml:Y>20 </gml:Y><gml:Z>30</gml:Z></gml:coord> </gml:Point></gml:PointMember></gml:MultiPoint></pre>

Example 2

The following C code shows how to use the ST_GML transform group for inserting geometries without using the explicit ST_Geometry function for the host-variable gml_buffer, which is of type CLOB and contains the GML representation of the point (20,20) that is to be inserted.

```
EXEC SQL BEGIN DECLARE SECTION;
    sqlint32 id = 0;
    SQL TYPE IS db2gse.ST_Geometry AS CLOB(1000) gml_buffer;
EXEC SQL END DECLARE SECTION;

// set the transform group for all subsequent SQL statements
EXEC SQL
    SET CURRENT DEFAULT TRANSFORM GROUP = ST_GML;
    id = 100;
    strcpy(gml_buffer.data, "<gml:point><gml:coord>"
        "<gml:X>20</gml:X> <gml:Y>20</gml:Y></gml:coord></gml:point>");

//initialize host variables
wkt_buffer.length = strlen(gml_buffer.data);

// insert point using WKT into column of type ST_Geometry
EXEC SQL
    INSERT
    INTO transforms_sample(id, geom)
    VALUES ( :id, :gml_buffer );
```

Chapter 20. Supported data formats

DB2 Spatial Extender provides industry standard spatial data formats that you can use.

The following four spatial data formats are described:

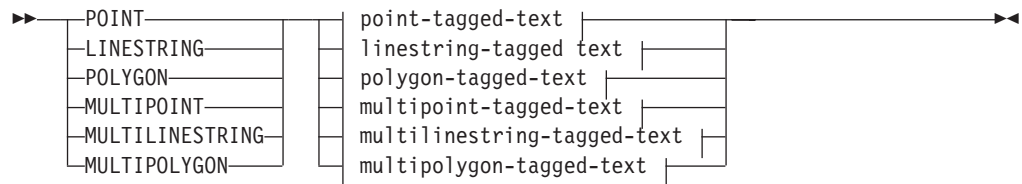
- Well-known text (WKT) representation
- Well-known binary (WKB) representation
- Shape representation
- Geography Markup Language (GML) representation

Well-known text (WKT) representation

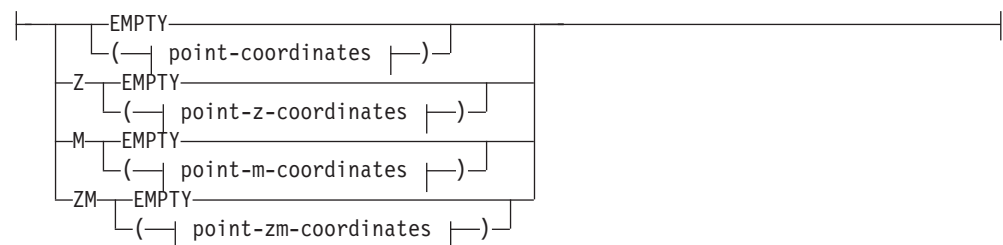
The OpenGIS Consortium "Simple Features for SQL" specification defines the well-known text representation to exchange geometry data in ASCII format. This representation is also referenced by the ISO "SQL/MM Part: 3 Spatial" standard.

See "Spatial functions that convert geometries to and from data exchange formats" for information on functions which accept and produce WKT data.

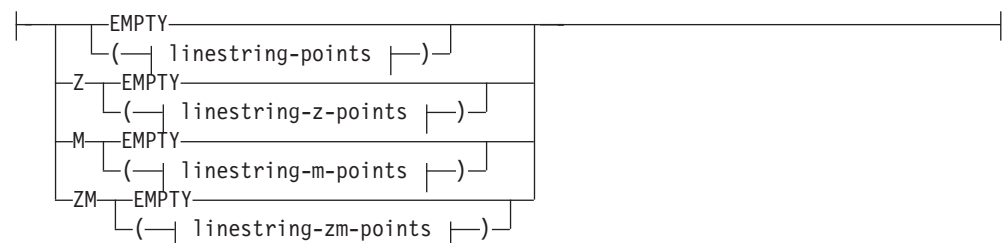
The well-known text representation of a geometry is defined as follows:



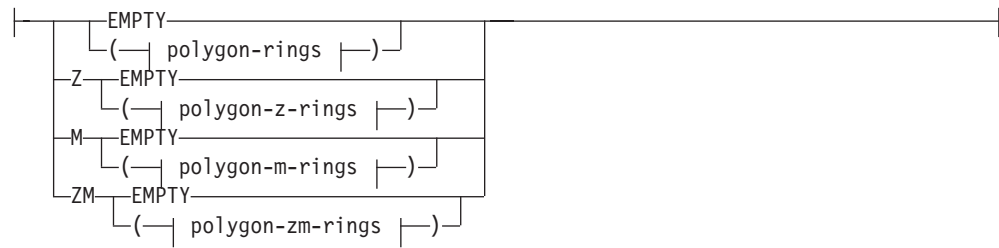
point-tagged-text:



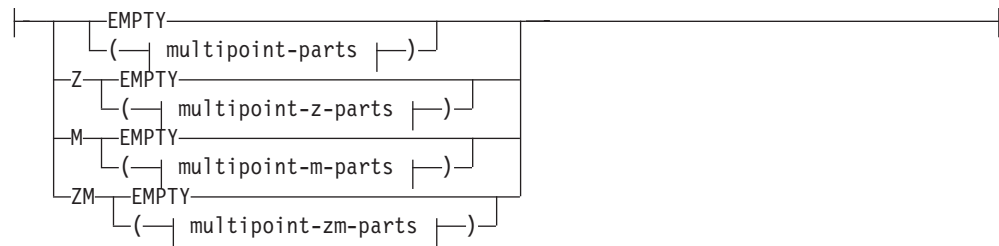
linestring-tagged-text:



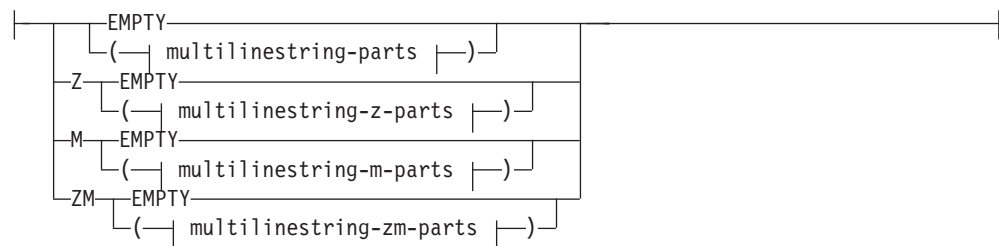
polygon-tagged-text:



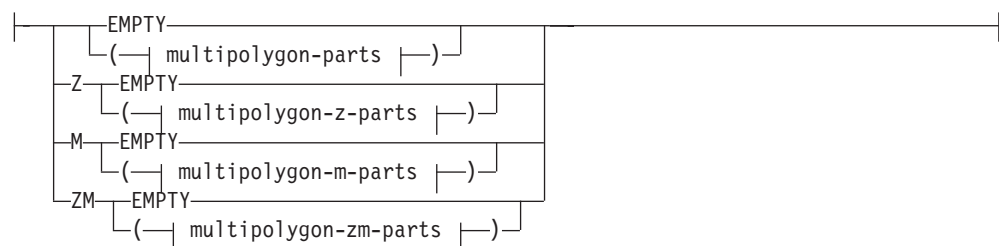
multipoint-tagged-text:



multilinestring-tagged-text:



multipolygon-tagged-text:



point-coordinates:



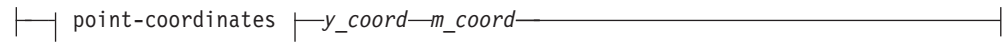
point-z-coordinates:



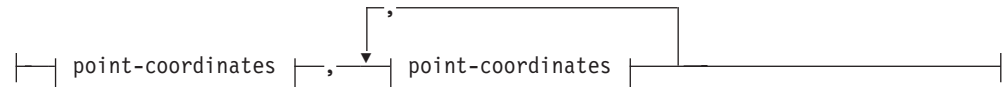
point-m-coordinates:



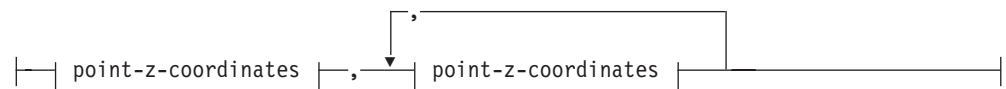
point-zm-coordinates:



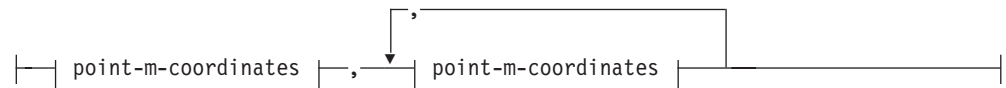
linestring-points:



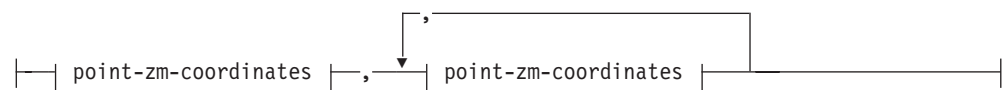
linestring-z-points:



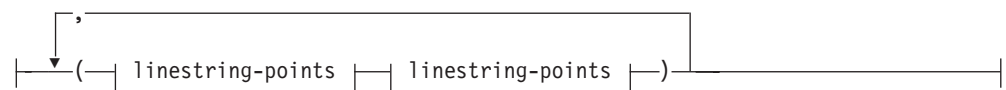
linestring-m-points:



linestring-zm-points:



polygon-rings:



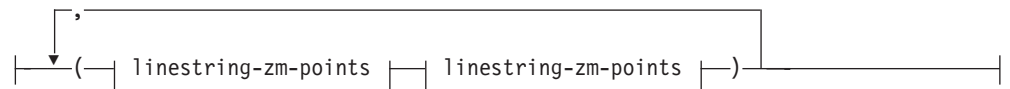
polygon-z-rings:



polygon-m-rings:



polygon-zm-rings:



multipoint-parts:



multipoint-z-parts:



multipoint-m-parts:



multipoint-zm-parts:



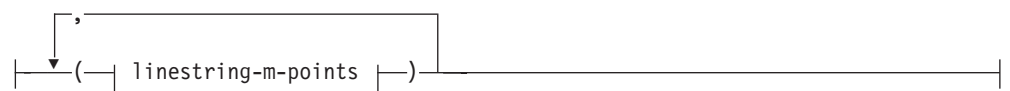
multilinestring-parts:



multilinestring-z-parts:



multilinestring-m-parts:



multilinestring-zm-parts:



multipolygon-parts:



multipolygon-z-parts:



multipolygon-m-parts:



multipolygon-zm-parts:



Parameters

x_coord

A numeric value (fixed, integer, or floating point), which represents the X coordinate of a point.

y_coord

A numeric value (fixed, integer, or floating point), which represents the Y coordinate of a point.

z_coord

A numeric value (fixed, integer, or floating point), which represents the Z coordinate of a point.

m_coord

A numeric value (fixed, integer, or floating point), which represents the M coordinate (measure) of a point.

If the geometry is empty, then the keyword EMPTY is to be specified instead of the coordinate list. The EMPTY keyword must not be embedded within the coordinate list

The following table provides some examples of possible text representations.

Table 33. Geometry types and their text representations

Geometry type	WKT representation	Comment
point	POINT EMPTY	empty point
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 M coordinate
point	POINT ZM(10.05 10.28 2.51 4.72)	point with Z coordinate and M coordinate
linestring	LINESTRING EMPTY	empty linestring
polygon	POLYGON ((10 10, 10 20, 20 20, 20 15, 10 10))	polygon
multipoint	MULTIPOINT Z(10 10 2, 20 20 3)	multipoint with Z coordinates
multilinestring	MULTILINESTRING M((310 30 1, 40 30 20, 50 20 10)(10 10 0, 20 20 1))	multilinestring with M coordinates
multipolygon	MULTIPOLYGON ZM(((1 1 1 1, 1 2 3 4, 2 2 5 6, 2 1 7 8, 1 1 1 1)))	multipolygon with Z coordinates and M coordinates

Well-known binary (WKB) representation

This section describes the well-known binary representation for geometries.

The OpenGIS Consortium "Simple Features for SQL" specification defines the well-known binary representation. This representation is also defined by the International Organization for Standardization (ISO) "SQL/MM Part: 3 Spatial" standard. See the related reference section at the end of this topic for information on functions that accept and produce the WKB.

The basic building block for well-known binary representations is the byte stream for a point, which consists of two double values. The byte streams for other geometries are built using the byte streams for geometries that are already defined.

The following example illustrates the basic building block for well-known binary representations.

```
// 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=wkbPoint
    Point     point;
};

WKBLineString {
    byte      byteOrder;
    uint32    wkbType;    // 2=wkbLineString
    uint32    numPoints;
    Point     points[numPoints];
};

WKBPolygon {
    byte      byteOrder;
    uint32    wkbType;    // 3=wkbPolygon
    uint32    numRings;
    LinearRing rings[numRings];
};

WKBMultiPoint {
    byte      byteOrder;
    uint32    wkbType;    // 4=wkbMultipoint
    uint32    num_wkbPoints;
    WKBPoint  WKBPoints[num_wkbPoints];
};

WKBMultiLineString {
    byte      byteOrder;
    uint32    wkbType;    // 5=wkbMultiLineString
    uint32    num_wkbLineStrings;
    WKBLineString WKBLineStrings[num_wkbLineStrings];
};

wkbMultiPolygon {
    byte      byteOrder;
    uint32    wkbType;    // 6=wkbMultiPolygon
    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 example of a geometry in well-known binary representation using NDR coding.

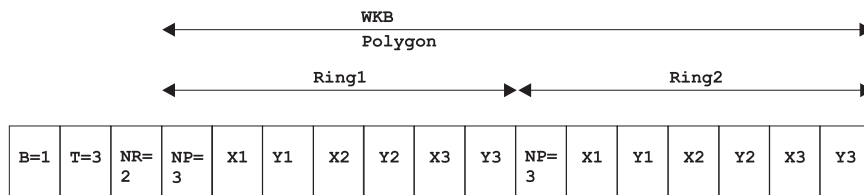


Figure 21. Geometry representation in NDR format. (B=1) of type polygon (T=3) with 2 linears (NR=2), where each ring has 3 points (NP=3).

Shape representation

Shape representation is a widely used industry standard defined by ESRI.

For a full description of shape representation, see the ESRI website at <http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf>.

Geography Markup Language (GML) representation

The Geography Markup Language (GML) is an XML encoding for geographic information defined by the OpenGIS Consortium "Geography Markup Language V2" specification.

DB2 Spatial Extender has several functions that generate geometries from representations in geography markup language(GML) representation.

For more details about this OpenGIS Consortium specification, see "OpenGIS Geography Markup Language (GML) Encoding Standard" at <http://www.opengeospatial.org/standards/gml>.

Chapter 21. Supported coordinate systems

DB2 Spatial Extender uses a specific coordinate systems syntax and supported coordinate system values to provide a standard textual representation for coordinate system information.

Coordinate systems syntax

The coordinate systems syntax is a string representation of such coordinate system.

The well-known text representation of spatial reference systems provides a standard textual representation for coordinate system information. The definitions of the well-known text representation are defined by the OGC "Simple Features for SQL" specification and the ISO SQL/MM Part 3: Spatial standard.

A coordinate 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 able 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 preceding note 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>
```

The type of coordinate system is identified by the keyword used:

PROJCS

A data set's coordinate system is identified by the PROJCS keyword if the data is in projected coordinates

GEOGCS

A data set's coordinate system is identified by the GEOGCS keyword if the data is in geographic coordinates

GEOCCS

A data set's coordinate system is identified by the GEOCCS keyword if the data is 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 are 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>
<inverse flattening> = <number>
<prime meridian> = PRIMEM["<name>", <longitude>]
<longitude> = <number>
```

The semi-major axis is measured in meters and must be greater than zero.

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

Use linear units that are supported by DB2 Spatial Extender.

Table 34. 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

Use angular units that are supported by DB2 Spatial Extender.

Table 35. Supported angular units

Unit	Valid range for latitude	Valid range for longitude	Conversion factor
Radian	-pi/2 and pi/2 radians (inclusive)	-pi and pi radians (inclusive)	1.0
Decimal Degree	-90 and 90 degrees (inclusive)	-180 and 180 degrees (inclusive)	pi/180
Decimal Minute	-5400 and 5400 minutes (inclusive)	-10800 and 10800 minutes (inclusive)	(pi/180)/60
Decimal Second	-324000 and 324000 seconds (inclusive)	-648000 and 648000 seconds (inclusive)	(pi/180)*3600
Gon	-100 and 100 gradians (inclusive)	-200 and 200 gradians (inclusive)	pi/200

Table 35. Supported angular units (continued)

Unit	Valid range for latitude	Valid range for longitude	Conversion factor
Grad	-100 and 100 gradians (inclusive)	-200 and 200 gradians (inclusive)	$\pi/200$

Supported spheroids

Use spheroids that are supported by DB2 Spatial Extender.

Table 36. Supported spheroids

Name	Semi-major axis	Inverse flattening
Airy 1830	6377563.396	299.3249646
Airy Modified 1849	6377340.189	299.3249646
Average Terrestrial System 1977	6378135.0	298.257
Australian National Spheroid	6378160.0	298.25
Bessel 1841	6377397.155	299.1528128
Bessel Modified	6377492.018	299.1528128
Bessel Namibia	6377483.865	299.1528128
Clarke 1858	6378293.639	294.260676369
Clarke 1866	6378206.4	294.9786982
Clarke 1866 (Michigan)	6378450.047	294.978684677
Clarke 1880	6378249.138	293.466307656
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 1922)	6378249.2	293.46598
Everest (1830 Definition)	6377299.36	300.8017
Everest 1830 Modified	6377304.063	300.8017
Everest Adjustment 1937	6377276.345	300.8017
Everest 1830 (1962 Definition)	6377301.243	300.8017255
Everest 1830 (1967 Definition)	6377298.556	300.8017
Everest 1830 (1975 Definition)	6377299.151	300.8017255
Everest 1969 Modified	6377295.664	300.8017

Table 36. Supported spheroids (continued)

Name	Semi-major axis	Inverse flattening
Fischer 1960	6378166.0	298.3
Fischer 1968	6378150 .0	298.3
Modified Fischer	6378155 .0	298.3
GEM 10C	6378137.0	298.257222101
GRS 1967	6378160.0	298.247167427
GRS 1967 Truncated	6378160.0	298.25
GRS 1980	6378137.0	298.257222101
Helmert 1906	6378200.0	298.3
Hough 1960	6378270.0	297.0
Indonesian National Spheroid	6378160.0	298.247
International 1924	6378388.0	297.0
International 1967	6378160.0	298.25
Krassowsky 1940	6378245.0	298.3
NWL 9D	6378145.0	298.25
NWL 10D	6378135.0	298.26
OSU 86F	6378136.2	298.25722
OSU 91A	6378136.3	298.25722
Plessis 1817	6376523.0	308.64
Sphere	6371000.0	0.0
Sphere (ArcInfo)	6370997.0	0.0
Struve 1860	6378298.3	294.73
Walbeck	6376896.0	302.78
War Office	6378300.0	296.0
WGS 1966	6378145.0	298.25
WGS 1972	6378135.0	298.26
WGS 1984	6378137.0	298.257223563

Supported prime meridians

Use prime meridians that are supported by DB2 Spatial Extender.

Table 37. 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
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

Use map projections that are supported by DB2 Spatial Extender.

Table 38. Cylindrical 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

Table 39. Conic projections

Name	Conic projection
Albers conic equal-area	Chamberlin trimetric
Bipolar oblique conformal conic	Two-point equidistant
Bonne	Hammer-Aitoff equal-area
Equidistant conic	Van der Grinten I
Lambert conformal conic	Miscellaneous
Polyconic	Alaska series E
Simple conic	Alaska Grid (Modified-Stereographic by Snyder)

Table 40. Map projection parameters

Parameter	Description
central_meridian	The line of longitude chosen as the origin of x-coordinates.
scale_factor	Scale_factor is 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 longitude 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.
longitude_of_origin	The longitude chosen as the origin of x-coordinates.
latitude_of_origin	The latitude chosen as the origin of y-coordinates.
false_easting	A value added to x-coordinates so that all x-coordinate values are positive.
false_northing	A value added to y-coordinates so that all y-coordinates are positive.
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.

Table 40. Map projection parameters (continued)

Parameter	Description
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.

Appendix A. Overview of the DB2 technical information

DB2 technical information is available in multiple formats that can be accessed in multiple ways.

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
 - Topics (Task, concept and reference topics)
 - Sample programs
 - Tutorials
- DB2 books
 - PDF files (downloadable)
 - PDF files (from the DB2 PDF DVD)
 - printed books
- Command-line help
 - Command help
 - Message help

Note: The DB2 Information Center topics are updated more frequently than either the PDF or the hardcopy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks® publications online at ibm.com. Access the DB2 Information Management software library site at <http://www.ibm.com/software/data/sw-library/>.

Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how to improve the DB2 documentation, send an email to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this email address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss. English and translated DB2 Version 10.1 manuals in PDF format can be downloaded from www.ibm.com/support/docview.wss?rs=71&uid=swg27009474.

Although the tables identify books available in print, the books might not be available in your country or region.

The form number increases each time a manual is updated. Ensure that you are reading the most recent version of the manuals, as listed below.

Note: The *DB2 Information Center* is updated more frequently than either the PDF or the hard-copy books.

Table 41. DB2 technical information

Name	Form Number	Available in print	Availability date
<i>Administrative API Reference</i>	SC27-5506-00	Yes	July 28, 2013
<i>Administrative Routines and Views</i>	SC27-5507-00	No	July 28, 2013
<i>Call Level Interface Guide and Reference Volume 1</i>	SC27-5511-00	Yes	July 28, 2013
<i>Call Level Interface Guide and Reference Volume 2</i>	SC27-5512-00	Yes	July 28, 2013
<i>Command Reference</i>	SC27-5508-00	Yes	July 28, 2013
<i>Database Administration Concepts and Configuration Reference</i>	SC27-4546-00	Yes	July 28, 2013
<i>Data Movement Utilities Guide and Reference</i>	SC27-5528-00	Yes	July 28, 2013
<i>Database Monitoring Guide and Reference</i>	SC27-4547-00	Yes	July 28, 2013
<i>Data Recovery and High Availability Guide and Reference</i>	SC27-5529-00	Yes	July 28, 2013
<i>Database Security Guide</i>	SC27-5530-00	Yes	July 28, 2013
<i>DB2 Workload Management Guide and Reference</i>	SC27-5520-00	Yes	July 28, 2013
<i>Developing ADO.NET and OLE DB Applications</i>	SC27-4549-00	Yes	July 28, 2013
<i>Developing Embedded SQL Applications</i>	SC27-4550-00	Yes	July 28, 2013
<i>Developing Java Applications</i>	SC27-5503-00	Yes	July 28, 2013
<i>Developing Perl, PHP, Python, and Ruby on Rails Applications</i>	SC27-5504-00	No	July 28, 2013
<i>Developing RDF Applications for IBM Data Servers</i>	SC27-5505-00	Yes	July 28, 2013
<i>Developing User-defined Routines (SQL and External)</i>	SC27-5501-00	Yes	July 28, 2013
<i>Getting Started with Database Application Development</i>	GI13-2084-00	Yes	July 28, 2013

Table 41. DB2 technical information (continued)

Name	Form Number	Available in print	Availability date
<i>Getting Started with DB2 Installation and Administration on Linux and Windows</i>	GI13-2085-00	Yes	July 28, 2013
<i>Globalization Guide</i>	SC27-5531-00	Yes	July 28, 2013
<i>Installing DB2 Servers</i>	GC27-5514-00	Yes	July 28, 2013
<i>Installing IBM Data Server Clients</i>	GC27-5515-00	No	July 28, 2013
<i>Message Reference Volume 1</i>	SC27-5523-00	No	July 28, 2013
<i>Message Reference Volume 2</i>	SC27-5524-00	No	July 28, 2013
<i>Net Search Extender Administration and User's Guide</i>	SC27-5526-00	No	July 28, 2013
<i>Partitioning and Clustering Guide</i>	SC27-5532-00	Yes	July 28, 2013
<i>pureXML Guide</i>	SC27-5521-00	Yes	July 28, 2013
<i>Spatial Extender User's Guide and Reference</i>	SC27-5525-00	No	July 28, 2013
<i>SQL Procedural Languages: Application Enablement and Support</i>	SC27-5502-00	Yes	July 28, 2013
<i>SQL Reference Volume 1</i>	SC27-5509-00	Yes	July 28, 2013
<i>SQL Reference Volume 2</i>	SC27-5510-00	Yes	July 28, 2013
<i>Text Search Guide</i>	SC27-5527-00	Yes	July 28, 2013
<i>Troubleshooting and Tuning Database Performance</i>	SC27-4548-00	Yes	July 28, 2013
<i>Upgrading to DB2 Version 10.5</i>	SC27-5513-00	Yes	July 28, 2013
<i>What's New for DB2 Version 10.5</i>	SC27-5519-00	Yes	July 28, 2013
<i>XQuery Reference</i>	SC27-5522-00	No	July 28, 2013

Table 42. DB2 Connect-specific technical information

Name	Form Number	Available in print	Availability date
<i>DB2 Connect Installing and Configuring DB2 Connect Personal Edition</i>	SC27-5516-00	Yes	July 28, 2013
<i>DB2 Connect Installing and Configuring DB2 Connect Servers</i>	SC27-5517-00	Yes	July 28, 2013
<i>DB2 Connect User's Guide</i>	SC27-5518-00	Yes	July 28, 2013

Displaying SQL state help from the command line processor

DB2 products return an SQLSTATE value for conditions that can be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

Procedure

To start SQL state help, open the command line processor and enter:

```
? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.

For example, ? 08003 displays help for the 08003 SQL state, and ? 08 displays help for the 08 class code.

Accessing different versions of the DB2 Information Center

Documentation for other versions of DB2 products is found in separate information centers on ibm.com[®].

About this task

For DB2 Version 10.1 topics, the *DB2 Information Center* URL is <http://pic.dhe.ibm.com/infocenter/db2luw/v10r1>.

For DB2 Version 9.8 topics, the *DB2 Information Center* URL is <http://pic.dhe.ibm.com/infocenter/db2luw/v9r8/>.

For DB2 Version 9.7 topics, the *DB2 Information Center* URL is <http://pic.dhe.ibm.com/infocenter/db2luw/v9r7/>.

For DB2 Version 9.5 topics, the *DB2 Information Center* URL is <http://publib.boulder.ibm.com/infocenter/db2luw/v9r5>.

Terms and conditions

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability: These terms and conditions are in addition to any terms of use for the IBM website.

Personal use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights: Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the previous instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Trademarks: IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.shtml

Appendix B. Notices

This information was developed for products and services offered in the U.S.A. Information about non-IBM products is based on information available at the time of first publication of this document and is subject to change.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information about 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 grant 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 character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country/region 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, changes, or both in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to websites not owned by IBM are provided for convenience only and do not in any manner serve as an endorsement of those

websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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.

Licensees 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 that has been exchanged, should contact:

IBM Canada Limited
U59/3600
3600 Steeles Avenue East
Markham, Ontario L3R 9Z7
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 document 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 contains sample application programs in source language, which illustrate 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample 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

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies

- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle, its affiliates, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Intel, Intel logo, Intel Inside, Intel Inside logo, Celeron, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
- Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

A

- administration notification log
 - details 105
- aggregate functions
 - spatial columns 238, 252, 253, 294, 295, 403
- ArcExplorer
 - using as interface 85
- automatic geocoding 57
- azimuthal projections 38

B

- batch geocoding 57
- batch mode
 - running geocoders 62

C

- catalog views
 - Spatial Extender 107
 - SPATIAL_REF_SYS 117
 - ST_COORDINATE_SYSTEMS 107
 - ST_GEOCODER_PARAMETERS 109
 - ST_GEOCODERS 111
 - ST_GEOCODING 111
 - ST_GEOCODING_PARAMETERS 112
 - ST_GEOMETRY_COLUMNS 108
 - ST_SIZINGS 114
 - ST_SPATIAL_REFERENCE_SYSTEMS 114
 - ST_UNITS_OF_MEASURE 117
- command line processor (CLP)
 - messages 102
 - Spatial Extender commands 119
- commands
 - DB2 Spatial Extender 119
 - db2se 119
 - db2trc
 - tracing DB2 Spatial Extender 104
- comparison functions
 - container relationships 229
 - DB2 Spatial Extender 227
 - DE-9IM pattern matrix string 231
 - geometry envelopes 229
 - identical geometries 230
 - intersections between geometries 229, 230
- conformal projections 38
- constructor functions
 - DB2 Spatial Extender 226
 - ESRI shape representation 224
 - Geography Markup Language (GML) representation 225
 - geometry values from coordinates 225
 - geometry values from data exchange formats 222
 - well-known binary representation 223
 - well-known text representation 222
- conversion
 - coordinate data into integers 44, 48
 - spatial data between coordinate systems 236
- coordinate systems
 - altering 120
 - creating 39, 124

- coordinate systems (*continued*)
 - deleting 132
 - overview 33, 419
 - ST_COORDINATE_SYSTEMS catalog view 107
 - ST_SPATIAL_REFERENCE_SYSTEMS catalog view 114
 - syntax 419
 - using existing coordinate systems 39
- coordinates
 - conversions 40, 44, 48
 - obtaining 232
 - overview 40
- CREATE INDEX statement
 - spatial grid index 74

D

- data formats
 - DB2 Spatial Extender 411
 - Geography Markup Language (GML) 418
 - shape representation 418
 - well-known binary (WKB) representation 416
 - well-known text (WKT) representation 411
- data types
 - DB2 Spatial Extender 53
 - multi-unit features 52
 - obtaining 232
 - single-unit features 52
- databases
 - DB2 Spatial Extender
 - enabling databases 29
 - resources 29
 - migrating
 - Spatial Extender 162
- DB2 Information Center
 - versions 430
- DB2 Spatial Extender
 - angular units 421
 - business data as source data 4
 - calling procedures 89
 - catalog views 107
 - CLP 13
 - commands
 - db2se alter_cs 120
 - db2se alter_srs 121
 - db2se create_cs 124
 - db2se create_srs 125
 - db2se disable_autogc 129
 - db2se disable_db 130
 - db2se drop_cs 132
 - db2se drop_srs 132
 - db2se enable_autogc 133
 - db2se enable_db 135
 - db2se export_shape 136
 - db2se import_shape 139
 - db2se migrate 162
 - db2se register_gc 146
 - db2se register_spatial_column 148
 - db2se remove_gc_setup 150
 - db2se restore_indexes 151
 - db2se run_gc 152
 - db2se save_indexes 152

DB2 Spatial Extender *(continued)*

commands *(continued)*

- db2se setup_gc 154
- db2se shape_info 157
- db2se unregister_gc 158
- db2se unregister_spatial_column 159
- db2se upgrade 160
- gseidx 81
- summary 119

constructor functions

- examples 226
- overview 221

coordinate systems 33

coordinate systems syntax 419

creating projects 15

data formats 411

data types

- all features 53
- multi-unit features 52
- single-unit features 52

functions

- see spatial functions 217

geocoders 57

getting started 19

header file 89

homogeneous collections

- multilinestrings 7
- multipoints 7
- multipolygons 7

installing 19, 21, 22, 24

interfaces 13

linear units 421

map projections 424

offset values 45

overview 1, 13

prime meridians 424

queries

- spatial functions 68
- spatial grid indexes 68

resources 29

sample program 91

scale factors 45

setting up 13, 19

setting up spatial resources

- databases 29
- project 33

spatial data types 51

spatial reference systems

- See spatial reference systems 40

spheroids 422

tracing problems 104

troubleshooting 97

units for offset values and scale factors 46

updating from 32-bit system to 64-bit system 28

upgrading

- overview 27
- process 27
- verifying installation 24
- writing applications 89

db2se commands

- alter_cs 120
- alter_srs 121
- create_cs 124
- create_srs 125
- disable_autogc 129
- disable_db 130
- drop_cs 132

db2se commands *(continued)*

- drop_srs 132
- enable_autogc 133
- enable_db 135
- export_shape 136
- import_shape 139
- invoking DB2 Spatial Extender commands 119
- migrate 162
- register_gc 146
- register_spatial_column 148
- remove_gc_setup 150
- restore_indexes 151
- run_gc 152
- save_indexes 152
- setup_gc 154
- shape_info 157
- unregister_gc 158
- unregister_spatial_column 159
- upgrade 160

db2trc command

- DB2 Spatial Extender problems 104

DE_HDN_SRS_1004 spatial reference system 43

DEFAULT_SRS spatial reference system 43

distance

- ST_Distance function 234, 268
- ST_DistanceToPoint function 271
- ST_PointAtDistance function 362

documentation

- overview 427
- PDF files 427
- printed 427
- terms and conditions of use 430

E

equal-area projections 38

equidistant projections 38

exports

- data
- shapefile 57

F

formulas

- geocoding 44, 48

function messages 101

functions

- generating spatial data 4
- spatial
- categories 221
- data-exchange format conversions 221

G

GCS_NORTH_AMERICAN_1927 coordinate system 43

GCS_NORTH_AMERICAN_1983 coordinate system 43

GCS_WGS_1984 coordinate system 43

GCSW_DEUTSCHE_HAUPTDRE IECKSNETZ coordinate system 43

geocoded columns

- disabling autogeocoding 129
- enabling autogeocoding 133

geocoders

- overview 57
- registering 30, 146

- geocoders (*continued*)
 - running
 - automatically 61
 - batch mode 62, 152
 - ST_GEOCODER_PARAMETERS catalog view 109
 - ST_GEOCODERS catalog view 111
 - ST_GEOCODING catalog view 111
 - ST_GEOCODING_PARAMETERS catalog view 112
 - ST_SIZINGS catalog view 114
 - unregistering 158
- geocoding
 - overview 57
 - removing setup 150
 - setting up 59, 154
- geographic coordinate system 33
- geographic features
 - comparison functions 227
 - overview 1
- Geographic Markup Language (GML)
 - data format 418
- geometries
 - client-server data transfer 405
 - generating new
 - conversion of one to another 235
 - modified forms 236
 - new space configurations 235
 - one from many 235
 - overview 235
 - new from existing geometry measures 236
 - overview 7
 - properties
 - overview 9
 - spatial data 5
 - spatial functions 231
 - spatial functions for data-type information 232
- geometry properties
 - boundaries 10
 - closed 11
 - coordinates 10
 - dimension 11
 - empty 11
 - exteriors 10
 - interiors 10
 - M coordinates 10
 - minimum bounding rectangle 11
 - non-simple 10
 - not empty 11
 - simple 10
 - types 9
 - X and Y coordinates 10
 - Z coordinates 10
- GET GEOMETRY command 81
- grid cell sizes
 - Spatial Extender 69
- grid indexes
 - overview 67
 - tuning 75
- grid levels
 - Spatial Extender 69
- grid sizes
 - spatial grid index 76
- gse_disable_autogc stored procedure 179
- gse_disable_db stored procedure 181
- gse_disable_sref stored procedure 184
- gse_enable_autogc stored procedure 185
- gse_enable_db stored procedure 187
- gse_enable_sref stored procedure 173

- gse_export_shape 189
- gse_import_shape stored procedure 192
- gse_register_gc stored procedure 199
- gse_register_layer stored procedure 203
- gse_run_gc stored procedure 207
- gse_unregist_gc stored procedure 213
- gseidx command 81

H

- hardware
 - requirements
 - Spatial Extender 20
- header files
 - DB2 Spatial Extender 89
- help
 - SQL statements 430

I

- imports
 - shape data 56
 - spatial data 5
- Index Advisor
 - determining appropriate grid size 69
 - GET GEOMETRY command to invoke 81
 - purpose 67, 75
- indexes
 - Index Advisor command 81
 - spatial functions 231, 234
 - spatial grid indexes
 - CREATE INDEX statement 74
 - overview 67
 - statistics 77
- installation
 - DB2 Spatial Extender
 - Linux 22
 - procedure 19
 - system requirements 20
 - UNIX 22
 - Windows 21

L

- linestrings 7
- logs
 - diagnostic 105

M

- MBR
 - details 11
- measure information
 - obtaining 232
- messages
 - DB2 Spatial Extender
 - CLP 102
 - functions 101
 - messages parts 97
 - migration information 102
 - shape information 102
 - stored procedures 99
- minimum bounding rectangle
 - details 11

- MQTs
 - spatial data 63
- multipliers to improve performance
 - processing coordinates 44, 48

N

- NAD27_ SRS_1002 (spatial reference system) 43
- NAD83_ SRS_1 (spatial reference system) 43
- notices 433

O

- offset values
 - overview 44, 48

P

- partitioned database environments
 - spatial data 63, 64
- performance
 - coordinate-data conversions 44, 48
- points 7
- polygons 7
- projected coordinate systems 33, 38
- properties of geometries
 - overview 9
 - spatial functions
 - boundary information 233
 - configuration information 233
 - coordinate and measure information 232
 - dimensional information 233
 - geometries within a geometry 233
 - spatial reference system 234

R

- registration
 - geocoders 30
 - spatial columns 54

S

- samples
 - DB2 Spatial Extender 91
- scale factors
 - overview 44, 48
- shape data
 - importing into table 56
- shape files
 - displaying information 157
 - exporting 136
 - importing 139
- shape representation 418
- shapefiles
 - exporting data 57
- software requirements
 - Spatial Extender 20
- spatial columns
 - creating 53
 - geocoding 57
 - populating 55
 - registering 54, 148
 - setting up 51
 - unregistering 159
- spatial columns (*continued*)
 - views 84
 - visualization tools 51
- spatial data
 - accessing 67
 - analyzing
 - functions 85
 - interfaces 85
 - overview 85
 - using indexes 86
 - creating projects 15
 - data formats 411
 - data types
 - all features 53
 - multi-unit features 52
 - single-unit features 52
 - derived from business data 4
 - exporting 55
 - functions 4
 - generating 85
 - geocoding 57
 - importing 5, 55
 - origin 4
 - overview 1, 5
 - partitioned databases 63, 64
 - retrieving
 - functions 85
 - interfaces 85
 - using indexes 86
 - spatial reference system identifier 11
 - ST_GEOMETRY_COLUMNS view 108
 - stored procedures 165
 - transferring from client to server 405
- spatial databases
 - disabling 130
 - enabling 29, 135
- Spatial Extender
 - See DB2 Spatial Extender 1
- spatial extent
 - overview 40
- spatial functions
 - by input type 219
 - categories 221
 - coding 217
 - comparison functions 227
 - comparisons of geometries
 - container relationships 229
 - DE-9IM pattern matrix string 231
 - geometry envelopes 229
 - identical geometries 230
 - intersections 229, 230
 - constructor functions 221
 - converting geometries between coordinate systems 236
 - data types 217
 - data-exchange format conversions
 - constructor functions 221
 - ESRI shape representation 224
 - Geography Markup Language (GML)
 - representation 225
 - geometry values from coordinates 225
 - geometry values from data exchange formats 222
 - well-known binary representation 223
 - well-known text representation 222
 - distance information 234
 - EnvelopesIntersect 236
 - examples 85

spatial functions (*continued*)

- generating new geometries from existing geometries
 - converting geometries 235
 - from existing geometry measures 236
 - modified forms of existing geometries 236
 - new space configurations 235
 - one from many 235
 - overview 235
- index information 234
- indexes 231
- MBR aggregate 238, 252, 294
- overview 217
- properties of geometries
 - boundary information 233
 - configuration information 233
 - coordinate and measure information 232
 - data-type information 232
 - dimensional information 233
 - geometries within a geometry 233
 - overview 231
 - spatial reference system 234
- queries 68
- spatial indexes 86
- ST_AppendPoint 240
- ST_Area 241
- ST_AsBinary 243
- ST_AsGML 244
- ST_AsShape 245
- ST_AsText 246
- ST_Boundary 248
- ST_Buffer 249
- ST_Centroid 255
- ST_ChangePoint 256
- ST_Contains 257
- ST_ConvexHull 260
- ST_CoordDim 262
- ST_Crosses 263
- ST_Difference 264
- ST_Dimension 266
- ST_Disjoint 267
- ST_Distance 268
- ST_DistanceToPoint 271
- ST_Endpoint 272
- ST_Envelope 273
- ST_EnvIntersects 274
- ST_EqualCoordsys 275
- ST_Equals 276
- ST_EqualSRS 278
- ST_ExteriorRing 279
- ST_FindMeasure 280
- ST_Generalize 281
- ST_GeomCollection 283
- ST_GeomCollFromTxt 285
- ST_GeomCollFromWKB 286
- ST_Geometry 287
- ST_GeometryN 289
- ST_GeometryType 290
- ST_GeomFromText 291
- ST_GeomFromWKB 292
- ST_GetIndexPars 296
- ST_InteriorRingN 299
- ST_Intersection 300
- ST_Intersects 301
- ST_Is3d 304
- ST_IsClosed 305
- ST_IsEmpty 306
- ST_IsMeasured 307

spatial functions (*continued*)

- ST_IsRing 308
- ST_IsSimple 309
- ST_IsValid 310
- ST_Length 311
- ST_LineFromText 313
- ST_LineFromWKB 314
- ST_LineString 315
- ST_LineStringN 317
- ST_LocateAlong 280
- ST_LocateBetween 327, 329
- ST_M 318
- ST_MaxM 319
- ST_MaxX 320
- ST_MaxY 322
- ST_MaxZ 323
- ST_MBR 325
- ST_MBRIntersects 326
- ST_MeasureBetween 327, 329
- ST_MidPoint 330
- ST_MinM 331
- ST_MinX 333
- ST_MinY 334
- ST_MinZ 335
- ST_MLineFromText 336
- ST_MLineFromWKB 338
- ST_MPointFromText 339
- ST_MPointFromWKB 340
- ST_MPolyFromText 342
- ST_MPolyFromWKB 343
- ST_MultiLineString 345
- ST_MultiPoint 346
- ST_MultiPolygon 348
- ST_NumGeometries 349
- ST_NumInteriorRing 350
- ST_NumLineStrings 351
- ST_NumPoints 352
- ST_NumPolygons 353
- ST_Overlaps 354
- ST_Perimeter 356
- ST_PerpPoints 357
- ST_Point 359
- ST_PointAtDistance 362
- ST_PointFromText 363
- ST_PointFromWKB 364
- ST_PointN 365
- ST_PointOnSurface 366
- ST_PolyFromText 367
- ST_PolyFromWKB 368
- ST_Polygon 369
- ST_PolygonN 372
- ST_Relate 373
- ST_RemovePoint 374
- ST_SRID 375, 376
- ST_SrsID 375, 376
- ST_SrsName 378
- ST_StartPoint 378
- ST_SymDifference 379
- ST_ToGeomColl 382
- ST_ToLineString 383
- ST_ToMultiLine 384
- ST_ToMultiPoint 385
- ST_ToMultiPolygon 386
- ST_ToPoint 387
- ST_ToPolygon 388
- ST_Touches 389
- ST_Transform 390

- spatial functions (*continued*)
 - ST_Union 392
 - ST_Within 394
 - ST_WKBTToSQL 397
 - ST_WKTTToSQL 398
 - ST_X 399
 - ST_Y 400
 - ST_Z 401
 - union aggregate 253, 295, 403
- spatial grid indexes
 - analyzing statistics 77
 - calculating grid sizes 76
 - CREATE INDEX statement 74
 - creating 73
 - functions that use 86
 - generating 67
 - grid levels and sizes 67, 69
 - Index Advisor command 81
 - queries 68
- spatial reference systems
 - changing 121
 - choosing existing spatial reference systems 42
 - coordinates 48
 - creating 42, 46, 49, 125, 173
 - measures 48
 - offset values 49
 - overview 40, 43
 - removing definition 132
 - scale factors 47
 - setting up 40
 - SPATIAL_REF_SYS catalog view 117
- spatial transform groups
 - ST_GML 409
 - ST_Shape 408
 - ST_WellKnownBinary 406
 - ST_WellKnownText 405
- SPATIAL_REF_SYS view 117
- SQL statements
 - help
 - displaying 430
- ST_ALTER_COORDSYS stored procedure 166
- ST_ALTER_SRS stored procedure 168
- ST_COORDINATE_SYSTEMS view 107
- ST_CREATE_COORDSYS stored procedure 171
- ST_CREATE_SRS stored procedure 173
- ST_DISABLE_AUTOGEOCODING stored procedure 179
- ST_DISABLE_DB stored procedure 181
- ST_Distance function 268
- ST_DistanceToPoint function 271
- ST_DROP_COORDSYS stored procedure 182
- ST_DROP_SRS procedure 184
- ST_ENABLE_AUTOGEOCODING stored procedure 185
- ST_ENABLE_DB stored procedure 187
- ST_EXPORT_SHAPE stored procedure 189
- ST_GEOCODER_PARAMETERS view 109
- ST_GEOCODERS view 111
- ST_GEOCODING view 111
- ST_GEOCODING_PARAMETERS view 112
- ST_Geometry values 218
- ST_GEOMETRY_COLUMNS view 108
- ST_IMPORT_SHAPE stored procedure 192
- ST_PointAtDistance function 362
- ST_REGISTER_GEOCODER stored procedure 199
- ST_REGISTER_SPATIAL_COLUMN stored procedure 203
- ST_REMOVE_GEOCODING_SETUP stored procedure 205
- ST_RUN_GEOCODING stored procedure 207
- ST_SETUP_GEOCODING stored procedure 210

- ST_SIZINGS view 114
- ST_SPATIAL_REFERENCE_SYSTEMS view 114
- ST_UNITS_OF_MEASURE catalog view 117
- ST_UNREGISTER_GEOCODER stored procedure 213
- ST_UNREGISTER_SPATIAL_COLUMN stored procedure 214
- stored procedures
 - DB2 Spatial Extender 99, 165
 - ST_ALTER_COORDSYS 166
 - ST_ALTER_SRS 168
 - ST_CREATE_COORDSYS 171
 - ST_CREATE_SRS 173
 - ST_DISABLE_AUTOGEOCODING 179
 - ST_DISABLE_DB 181
 - ST_DROP_COORDSYS 182
 - ST_DROP_SRS 184
 - ST_ENABLE_AUTOGEOCODING 185
 - ST_ENABLE_DB 187
 - ST_EXPORT_SHAPE 189
 - ST_IMPORT_SHAPE 192
 - ST_REGISTER_GEOCODER 199
 - ST_REGISTER_SPATIAL_COLUMN 203
 - ST_REMOVE_GEOCODING_SETUP 205
 - ST_RUN_GEOCODING 207
 - ST_SETUP_GEOCODING 210
 - ST_UNREGISTER_GEOCODER 213
 - ST_UNREGISTER_SPATIAL_COLUMN 214

T

- terms and conditions
 - publications 430
- transform groups
 - overview 405
- troubleshooting
 - administration notification log 105
 - DB2 Spatial Extender
 - messages 97, 101, 102
 - overview 97
 - stored procedures 99
 - shape information messages 102
- true-direction projections 38
- tuning
 - spatial grid indexes 75

U

- union aggregate functions 253, 295, 403
- units for offset values and scale factors 44, 48
- upgrades
 - databases enabled for Spatial Extender 160
 - DB2 Spatial Extender
 - overview 27
 - procedure 27
 - procedure (32-bit to 64-bit system) 28

V

- views
 - spatial columns 84

W

- well-known binary (WKB) representation, data format 416
- well-known text (WKT) representation, data format 411
- WGS84_SRS_1003 spatial reference system 43



Printed in USA

SC27-5525-00



Spine information:

IBM DB2 10.5 for Linux, UNIX, and Windows

Spatial Extender User's Guide and Reference

