

IBM<sup>®</sup> DB2<sup>®</sup> Spatial Extender



# User's Guide and Reference

*Version 8*



IBM<sup>®</sup> DB2<sup>®</sup> Spatial Extender



# User's Guide and Reference

*Version 8*

Before using this information and the product it supports, be sure to read the general information under *Notices*.

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 [www.ibm.com/shop/publications/order](http://www.ibm.com/shop/publications/order)
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at [www.ibm.com/planetwide](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 International Business Machines Corporation 1998 - 2002. All rights reserved.**

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

---

# Contents

---

## Part 1. Introduction. . . . . 1

### Chapter 1. About DB2 Spatial Extender . . . 3

|   |   |
|---|---|
| The purpose of DB2 Spatial Extender . . . . .   | 3 |
| Spatial data . . . . .  | 4 |
| How data represents geographic features. . . . .  | 4 |
| The nature of spatial data . . . . .  | 6 |
| Where spatial data comes from . . . . .   | 7 |
| How features, spatial information, spatial data,<br>and geometries fit together . . . . . | 8 |

### Chapter 2. About Geometries . . . . . 11

|  |    |
|--|----|
| Geometries . . . . .                         | 11 |
| Properties of geometries . . . . .           | 13 |
| Type . . . . .                               | 13 |
| Geometry coordinates. . . . .                | 14 |
| X and Y coordinates . . . . .                | 14 |
| Z coordinates . . . . .                      | 15 |
| M coordinates . . . . .                      | 15 |
| Interior, boundary, and exterior . . . . .   | 15 |
| Simple or non-simple . . . . .               | 15 |
| Empty or not empty . . . . .                 | 15 |
| Minimum bounding rectangle (MBR) . . . . .   | 15 |
| Dimension . . . . .                          | 15 |
| Spatial reference system identifier. . . . . | 16 |

### Chapter 3. How to use DB2 Spatial Extender . . . . . 17

|  |    |
|--|----|
| How to use DB2 Spatial Extender. . . . .   | 17 |
| Interfaces to DB2 Spatial Extender and<br>associated functionality . . . . .           | 17 |
| Tasks that you perform to set up DB2<br>Spatial Extender and create projects . . . . . | 17 |

---

## Part 2. Setting up DB2 Spatial Extender. . . . . 25

### Chapter 4. Getting started with DB2 Spatial Extender . . . . . 27

|   |    |
|---|----|
| Setting up and configuring Spatial<br>Extender—Steps. . . . . | 27 |
| Setting up and configuring Spatial<br>Extender . . . . .      | 27 |

|  |    |
|--|----|
| System requirements for installing Spatial<br>Extender . . . . .               | 28 |
| Installing DB2 Spatial Extender for<br>Windows . . . . .                       | 30 |
| Installing DB2 Spatial Extender for AIX . . . . .                              | 32 |
| Installing DB2 Spatial Extender for HP-UX . . . . .                            | 34 |
| Installing DB2 Spatial Extender for Solaris<br>Operating Environment . . . . . | 36 |
| Installing DB2 Spatial Extender for Linux<br>and Linux OS/390 . . . . .        | 38 |
| Creating the DB2 Spatial Extender instance<br>environment . . . . .            | 40 |
| Verifying the Spatial Extender installation . . . . .                          | 42 |
| Troubleshooting tips for the installation<br>sample program . . . . .          | 44 |
| Post-Installation considerations. . . . .                                      | 45 |
| Downloading ArcExplorer . . . . .  | 45 |
| Accessing geocoder reference data . . . . .                                    | 45 |
| CD-ROMs for DB2 Spatial Extender data<br>and maps . . . . .                    | 47 |

### Chapter 5. Migrating the Spatial Extender environment to DB2 Universal Database Version 8 . . . . . 49

|  |    |
|--|----|
| Migrating a spatially-enabled database . . . . .             | 49 |
| Migrating from a 32-bit to a 64-bit<br>environment . . . . . | 50 |
| Migration messages . . . . .                                 | 51 |

### Chapter 6. Setting up a database . . . . . 53

|   |    |
|---|----|
| Configuring a database to accommodate<br>spatial data . . . . . | 53 |
| Tuning the database configuration parameters . . . . .          | 53 |
| Tuning transaction log characteristics . . . . .                | 53 |
| Tuning the application heap size . . . . .                      | 55 |
| Tuning the application control heap size . . . . .              | 56 |

### Chapter 7. Setting up spatial resources for a database . . . . . 59

|  |    |
|--|----|
| How to set up resources in your database . . . . .             | 59 |
| Inventory of resources supplied for your<br>database . . . . . | 59 |
| Enabling a database for spatial operations . . . . .           | 60 |
| How to work with reference data . . . . .                      | 61 |
| Reference data . . . . .                                       | 61 |

## Table of contents

|  |            |  |            |
|--|------------|--|------------|
| Setting up access to reference data . . . . .                                    | 61         | Environments for performing spatial analysis   | 119        |
| Registering a geocoder . . . . .   | 62         | Examples of spatial function operations . . . . .  | 119        |
| <hr/>  |            | Rules for using grid indexes to optimize<br>spatial functions . . . . .                    | 121        |
| <b>Part 3. Creating projects that use<br/>spatial data . . . . .</b>             | <b>63</b>  | <b>Chapter 13. DB2 Spatial Extender<br/>commands . . . . .</b>                             | <b>123</b> |
| <b>Chapter 8. Setting up spatial resources for<br/>a project . . . . .</b>       | <b>65</b>  | Invoking commands for setting up DB2<br>Spatial Extender and developing projects . . . . . | 123        |
| How to use coordinate systems . . . . .  | 65         | <b>Chapter 14. Writing applications and<br/>using the sample program . . . . .</b>         | <b>131</b> |
| Coordinate systems . . . . .   | 65         | Writing applications for DB2 Spatial<br>Extender . . . . .                                 | 131        |
| Selecting or creating coordinate systems.  | 73         | Including the DB2 Spatial Extender header<br>file in spatial applications . . . . .        | 131        |
| How to set up spatial reference systems. . . . .                                 | 74         | Calling DB2 Spatial Extender stored<br>procedures from an application . . . . .            | 132        |
| About spatial reference systems . . . . .  | 74         | The DB2 Spatial Extender sample program  | 134        |
| Selecting or creating spatial reference<br>systems . . . . .                     | 76         | <b>Chapter 15. Identifying DB2 Spatial<br/>Extender problems . . . . .</b>                 | <b>143</b> |
| <b>Chapter 9. Setting up spatial columns . . . . .</b>                           | <b>83</b>  | How to interpret DB2 Spatial Extender<br>messages. . . . .                                 | 143        |
| Spatial columns . . . . .  | 83         | DB2 Spatial Extender stored procedure<br>output parameters . . . . .                       | 146        |
| Spatial columns with viewable content . . . . .                                  | 83         | DB2 Spatial Extender function messages . . . . .   | 148        |
| Spatial data types . . . . .   | 83         | DB2 Spatial Extender CLP messages . . . . .  | 150        |
| Creating spatial columns. . . . .  | 86         | DB2 Control Center messages. . . . .   | 153        |
| Setting up spatial columns for access by<br>visualization tools . . . . .        | 86         | Tracing DB2 Spatial Extender problems with<br>the db2trc command. . . . .                  | 154        |
| <b>Chapter 10. Populating spatial columns . . . . .</b>                          | <b>89</b>  | The db2diag.log utility . . . . .  | 155        |
| How to import and export spatial data . . . . .                                  | 89         | <b>Part 4. Reference material . . . . .</b>  | <b>157</b> |
| About importing and exporting spatial<br>data. . . . .                           | 89         | <b>Chapter 16. Stored procedures . . . . .</b>   | <b>159</b> |
| Importing spatial data . . . . .   | 91         | GSE_export_sde . . . . .   | 160        |
| Exporting spatial data. . . . .  | 94         | GSE_import_sde . . . . .   | 162        |
| How to use a geocoder . . . . .  | 96         | ST_alter_coordsys. . . . .   | 165        |
| Geocoders and geocoding . . . . .  | 96         | ST_alter_srs. . . . .  | 167        |
| Setting up geocoding operations . . . . .  | 98         | ST_create_coordsys . . . . .   | 172        |
| Setting up a geocoder to run<br>automatically . . . . .                          | 101        | ST_create_srs . . . . .  | 174        |
| Running a geocoder in batch mode . . . . .                                       | 102        | ST_disable_autogeocoding . . . . .   | 182        |
| <b>Chapter 11. Using indexes and views to<br/>access spatial data. . . . .</b>   | <b>105</b> | ST_disable_db . . . . .  | 184        |
| Spatial indexes. . . . .   | 105        | ST_drop_coordsys . . . . .   | 186        |
| How to create spatial grid indexes . . . . .                                     | 106        | ST_drop_srs. . . . .   | 187        |
| Creating spatial grid indexes . . . . .  | 107        | ST_enable_autogeocoding . . . . .  | 189        |
| General guidelines for getting the most<br>out of spatial grid indexes . . . . . | 110        | ST_enable_db . . . . .   | 191        |
| Using the Index Advisor . . . . .  | 111        | ST_export_shape . . . . .  | 194        |
| Using views to access spatial columns . . . . .                                  | 118        |  |            |
| <b>Chapter 12. Analyzing and Generating<br/>spatial information . . . . .</b>    | <b>119</b> |  |            |

|   |            |   |            |
|---|------------|---|------------|
| ST_import_shape . . . . .   | 198        | Functions for information about geometries within a geometry. . . . .                                   | 268        |
| ST_register_geocoder. . . . .   | 207        | Functions for information about rings, boundaries, envelopes, and minimum bounding rectangles . . . . . | 270        |
| ST_register_spatial_column . . . . .  | 212        | Functions for information about dimensions . . . . .  | 271        |
| ST_remove_geocoding_setup . . . . .   | 215        | Functions for information as to whether a geometry is closed, empty, or simple . . . . .                | 272        |
| ST_run_geocoding . . . . .  | 217        | Functions for information about spatial reference systems . . . . .                                     | 272        |
| ST_setup_geocoding . . . . .  | 220        | Spatial functions that generate new geometries . . . . .  | 273        |
| ST_unregister_geocoder. . . . .   | 224        | Functions for converting geometries into other geometries . . . . .                                     | 273        |
| ST_unregister_spatial_column. . . . .   | 226        | Functions for creating geometries that represent configurations of space . . . . .                      | 274        |
| <b>Chapter 17. Catalog views . . . . .</b>  | <b>229</b> | Functions for deriving individual geometries from multiple geometries . . . . .                         | 278        |
| The DB2GSE.ST_COORDINATE_SYSTEMS catalog view . . . . .                               | 229        | Functions for creating geometries based on measures . . . . .   | 279        |
| The DB2GSE.ST_GEOMETRY_COLUMNS catalog view . . . . .                                 | 231        | Functions for creating modified forms of geometries . . . . .   | 280        |
| The DB2GSE.ST_GEOCODER_PARAMETERS catalog view . . . . .                              | 232        | Miscellaneous spatial functions . . . . .   | 282        |
| The DB2GSE.ST_GEOCODERS catalog view . . . . .  | 233        | <b>Chapter 19. Spatial functions: syntax and parameters . . . . .</b>                                   | <b>285</b> |
| The DB2GSE.ST_GEOCODING catalog view . . . . .  | 234        | Spatial functions: considerations and associated data types . . . . .                                   | 285        |
| The DB2GSE.ST_GEOCODING_PARAMETERS catalog view . . . . .                             | 236        | Factors to consider . . . . .   | 285        |
| The DB2GSE.ST_SIZINGS catalog view . . . . .  | 238        | Treating values of ST_Geometry as values of a subtype . . . . .   | 286        |
| The DB2GSE.ST_SPATIAL_REFERENCE_SYSTEMS catalog view . . . . .                        | 238        | Spatial functions listed according to input type . . . . .  | 287        |
| The DB2GSE.ST_UNITS_OF_MEASURE catalog view . . . . .                                 | 242        | MBR Aggregate . . . . .   | 291        |
| <b>Chapter 18. Spatial functions: categories and uses . . . . .</b>                   | <b>243</b> | ST_AppendPoint . . . . .  | 292        |
| Spatial functions that convert geometries to and from data exchange formats . . . . . | 243        | ST_Area . . . . .   | 294        |
| Constructor functions in general . . . . .  | 243        | ST_AsBinary . . . . .   | 297        |
| Well-known text (WKT) representation . . . . .  | 248        | ST_AsGML . . . . .  | 298        |
| Well-known binary (WKB) representation . . . . .                                      | 249        | ST_AsShape. . . . .   | 300        |
| ESRI shape representation . . . . .   | 250        | ST_AsText . . . . .   | 301        |
| Geography Markup Language (GML) representation . . . . .                              | 251        | ST_Boundary . . . . .   | 302        |
| Functions that make comparisons . . . . .   | 252        | ST_Buffer . . . . .   | 304        |
| Comparison functions in general. . . . .  | 252        | ST_Centroid . . . . .   | 307        |
| Descriptions of functions . . . . .   | 254        | ST_ChangePoint . . . . .  | 308        |
| Differences between similar functions . . . . .                                       | 264        | ST_Contains . . . . .   | 310        |
| Functions that return information about properties of geometries . . . . .            | 266        | ST_ConvexHull . . . . .   | 312        |
| ST_GeometryType: A function for information about data types . . . . .                | 266        | ST_CoordDim . . . . .   | 314        |
| Functions for information about coordinates and measures . . . . .                    | 266        | ST_Crosses . . . . .  | 315        |
|   |            | ST_Difference . . . . .   | 317        |

## Table of contents

|   |     |                              |     |
|---|-----|------------------------------|-----|
| ST_Dimension . . . . .                        | 319 | ST_MLineFromText . . . . .   | 407 |
| ST_Disjoint . . . . .                         | 320 | ST_MLineFromWKB . . . . .    | 408 |
| ST_Edge_GC_USA . . . . .                      | 322 | ST_MPointFromText . . . . .  | 411 |
| ST_Distance. . . . .                          | 327 | ST_MPointFromWKB . . . . .   | 412 |
| ST_Endpoint . . . . .                         | 331 | ST_MPolyFromText . . . . .   | 414 |
| ST_Envelope . . . . .                         | 332 | ST_MPolyFromWKB . . . . .    | 416 |
| ST_EnvIntersects . . . . .                    | 333 | ST_MultiLineString . . . . . | 418 |
| ST_EqualCoordsys . . . . .                    | 335 | ST_MultiPoint . . . . .      | 420 |
| ST_Equals . . . . .                           | 336 | ST_MultiPolygon . . . . .    | 422 |
| ST_EqualSRS . . . . .                         | 338 | ST_NumGeometries . . . . .   | 424 |
| ST_ExteriorRing . . . . .                     | 339 | ST_NumInteriorRing. . . . .  | 425 |
| ST_FindMeasure or ST_LocateAlong . . . . .    | 341 | ST_NumLineStrings . . . . .  | 427 |
| ST_Generalize . . . . .                       | 343 | ST_NumPoints. . . . .        | 428 |
| ST_GeomCollection . . . . .                   | 345 | ST_NumPolygons. . . . .      | 429 |
| ST_GeomCollFromTxt . . . . .                  | 347 | ST_Overlaps . . . . .        | 430 |
| ST_GeomCollFromWKB. . . . .                   | 349 | ST_Perimeter . . . . .       | 432 |
| ST_Geometry . . . . .                         | 351 | ST_PerpPoints . . . . .      | 434 |
| ST_GeometryN . . . . .                        | 353 | ST_Point . . . . .           | 437 |
| ST_GeometryType . . . . .                     | 354 | ST_PointFromText. . . . .    | 440 |
| ST_GeomFromText . . . . .                     | 355 | ST_PointFromWKB . . . . .    | 442 |
| ST_GeomFromWKB . . . . .                      | 357 | ST_PointN . . . . .          | 443 |
| ST_GetIndexParams . . . . .                   | 358 | ST_PointOnSurface . . . . .  | 444 |
| ST_InteriorRingN . . . . .                    | 361 | ST_PolyFromText . . . . .    | 446 |
| ST_Intersection. . . . .                      | 363 | ST_PolyFromWKB . . . . .     | 447 |
| ST_Intersects . . . . .                       | 364 | ST_Polygon . . . . .         | 449 |
| ST_Is3d . . . . .                             | 366 | ST_PolygonN . . . . .        | 452 |
| ST_IsClosed. . . . .                          | 368 | ST_Relate . . . . .          | 453 |
| ST_IsEmpty. . . . .                           | 370 | ST_RemovePoint . . . . .     | 454 |
| ST_IsMeasured. . . . .                        | 371 | ST_SrsId, ST_SRID . . . . .  | 456 |
| ST_IsRing . . . . .                           | 372 | ST_SrsName . . . . .         | 458 |
| ST_IsSimple. . . . .                          | 373 | ST_StartPoint . . . . .      | 459 |
| ST_IsValid . . . . .                          | 375 | ST_SymDifference. . . . .    | 460 |
| ST_Length . . . . .                           | 376 | ST_ToGeomColl . . . . .      | 463 |
| ST_LineFromText . . . . .                     | 378 | ST_ToLineString . . . . .    | 464 |
| ST_LineFromWKB . . . . .                      | 379 | ST_ToMultiLine . . . . .     | 465 |
| ST_LineString . . . . .                       | 381 | ST_ToMultiPoint . . . . .    | 466 |
| ST_LineStringN . . . . .                      | 383 | ST_ToMultiPolygon . . . . .  | 468 |
| ST_M . . . . .                                | 384 | ST_ToPoint . . . . .         | 469 |
| ST_MaxM . . . . .                             | 386 | ST_ToPolygon . . . . .       | 470 |
| ST_MaxX . . . . .                             | 388 | ST_Touches . . . . .         | 471 |
| ST_MaxY . . . . .                             | 390 | ST_Transform . . . . .       | 473 |
| ST_MaxZ . . . . .                             | 392 | ST_Union . . . . .           | 475 |
| ST_MBR . . . . .                              | 393 | ST_Within . . . . .          | 478 |
| ST_MBRIntersects. . . . .                     | 395 | ST_WKBToSQL . . . . .        | 479 |
| ST_MeasureBetween, ST_LocateBetween . . . . . | 397 | ST_WKTTToSQL . . . . .       | 481 |
| ST_MidPoint . . . . .                         | 398 | ST_X . . . . .               | 482 |
| ST_MinM . . . . .                             | 400 | ST_Y . . . . .               | 484 |
| ST_MinX. . . . .                              | 401 | ST_Z . . . . .               | 485 |
| ST_MinY. . . . .                              | 403 | Union aggregate . . . . .    | 487 |
| ST_MinZ. . . . .                              | 405 |                              |     |



|   |            |   |            |
|---|------------|---|------------|
| <b>Chapter 20. Transform groups . . . . .</b>       | <b>489</b> | DB2GSE.SPATIAL_REF_SYS . . . . .                | 549        |
| Transform groups . . . . .                          | 489        | <b>Appendix C. Deprecated spatial functions</b> | <b>551</b> |
| ST_WellKnownText . . . . .                          | 489        | AsShape . . . . .                               | 552        |
| ST_WellKnownBinary . . . . .                        | 491        | EnvelopesIntersect . . . . .                    | 552        |
| ST_Shape . . . . .                                  | 492        | GeometryFromShape. . . . .                      | 553        |
| ST_GML . . . . .                                    | 493        | Is3d . . . . .                                  | 553        |
| <b>Chapter 21. Supported data formats . . . . .</b> | <b>497</b> | IsMeasured . . . . .                            | 553        |
| Well-known text (WKT) representation. . . . .       | 497        | LineFromShape . . . . .                         | 554        |
| Well-known binary (WKB) representation              | 503        | LocateAlong . . . . .                           | 554        |
| Shape representation. . . . .                       | 505        | LocateBetween . . . . .                         | 554        |
| Geography Markup Language (GML)                     |            | M . . . . .                                     | 555        |
| representation . . . . .                            | 505        | MLine FromShape . . . . .                       | 555        |
| <b>Chapter 22. Supported coordinate</b>             |            | MPointFromShape . . . . .                       | 555        |
| <b>systems . . . . .</b>                            | <b>507</b> | MPolyFromShape . . . . .                        | 556        |
| Supported coordinate systems . . . . .              | 507        | PointFromShape . . . . .                        | 556        |
| Coordinate systems syntax. . . . .                  | 507        | PolyFromShape . . . . .                         | 556        |
| Supported angular units . . . . .                   | 510        | ShapeToSQL . . . . .                            | 557        |
| Supported spheroids. . . . .                        | 510        | ST_GeomFromText . . . . .                       | 557        |
| Supported geodetic datums . . . . .                 | 512        | ST_GeomFromWKB . . . . .                        | 558        |
| Supported prime meridians . . . . .                 | 514        | ST_LineFromText . . . . .                       | 558        |
| Supported map projections. . . . .                  | 515        | ST_LineFromWKB . . . . .                        | 558        |
| <b>Appendix A. Deprecated stored</b>                |            | ST_MLineFromText . . . . .                      | 559        |
| <b>procedures . . . . .</b>                         | <b>519</b> | ST_MLineFromWKB . . . . .                       | 559        |
| db2gse.gse_enable_autogc . . . . .                  | 520        | ST_MPointFromText . . . . .                     | 559        |
| db2gse.gse_enable_db . . . . .                      | 522        | ST_MPointFromWKB . . . . .                      | 560        |
| db2gse.gse_enable_idx . . . . .                     | 523        | ST_MPolyFromText . . . . .                      | 560        |
| db2gse.gse_enable_sref . . . . .                    | 524        | ST_MPolyFromWKB . . . . .                       | 561        |
| db2gse.gse_export_shape . . . . .                   | 526        | ST_OrderingEquals . . . . .                     | 561        |
| db2gse.gse_disable_autogc. . . . .                  | 528        | ST_Point . . . . .                              | 561        |
| db2gse.gse_disable_db . . . . .                     | 529        | ST_PointFromText. . . . .                       | 562        |
| db2gse.gse_disable_sref . . . . .                   | 530        | ST_PolyFromText . . . . .                       | 562        |
| db2gse.gse_import_shape . . . . .                   | 531        | ST_PolyFromWKB . . . . .                        | 562        |
| db2gse.gse_register_gc . . . . .                    | 533        | ST_Transform . . . . .                          | 563        |
| db2gse.gse_register_layer . . . . .                 | 535        | ST_SymmetricDiff. . . . .                       | 563        |
| db2gse.gse_run_gc . . . . .                         | 541        | Z . . . . .                                     | 564        |
| db2gse.gse_unregist_gc . . . . .                    | 542        | <b>Notices . . . . .</b>                        | <b>565</b> |
| db2gse.gse_unregist_layer . . . . .                 | 543        | Trademarks . . . . .                            | 568        |
| <b>Appendix B. Deprecated catalog views</b>         | <b>547</b> | <b>Index . . . . .</b>                          | <b>571</b> |
| DB2GSE.COORD_REF_SYS . . . . .                      | 547        | <b>Contacting IBM . . . . .</b>                 | <b>575</b> |
| DB2GSE.GEOMETRY_COLUMNS . . . . .                   | 548        | Product information . . . . .                   | 575        |
| DB2GSE.SPATIAL_GEOCODER . . . . .                   | 548        |   |            |



---

## Part 1. Introduction



---

# Chapter 1. About DB2 Spatial Extender

This chapter introduces DB2 Spatial Extender by explaining its purpose, describing the data that it supports, and explaining how its underlying concepts fit together.

---

## The purpose of DB2 Spatial Extender

Use DB2<sup>®</sup> 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)

## About DB2 Spatial Extender

- 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.

---

## Spatial data

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

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

### How data represents geographic features

In DB2<sup>®</sup> 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 the cell of a relational table.) For example, consider office buildings and residences. In Figure 1 on page 5, 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—can be regarded as representing the customer’s residence.

### BRANCHES

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

### CUSTOMERS

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

*Figure 1. 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. A customer’s address is 9 Concourt Circle, San Jose CA 95141. DB2 Spatial Extender can translate these addresses into values that indicate where the branch and the customer’s home are located with respect to one another. Figure 2 on page 6 shows the BRANCHES and CUSTOMERS tables with new columns that are designated to contain such values.

## About DB2 Spatial Extender

### BRANCHES

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

### CUSTOMERS

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

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

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

Much spatial data is made up of coordinates. A *coordinate* is a number that denotes a position that is relative to a point of reference. For example, degrees of latitude are coordinates that denote positions relative to the equator. Degrees of longitude are coordinates that denote positions relative to the Greenwich meridian. Thus, on a map, the position of Yellowstone National Park is defined by 44.45 latitude degrees north of the equator and 110.40 degrees of longitude west of the Greenwich meridian. More precisely, these coordinates reference the center of Yellowstone National Park.

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

The simplest spatial data item consists of two coordinates that define the position of a single geographic feature. A more extensive spatial data item consists of several coordinates that define a linear path such as a road or river might form. A third kind consists of coordinates that define the 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

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 (as mentioned in “How data represents geographic features” on page 4). This process is called *geocoding*. To see the sequence involved, consider Figure 2 on page 6 as a “before” picture and Figure 3 as an “after” picture. Figure 2 on page 6 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     | STATE | ZIP   | LOCATION  |
|-----|-----------------|--------------------|----------|-------|-------|-----------|
| 937 | Airzone-Multern | 92467 Airzone Blvd | San Jose | CA    | 95141 | 1653 3094 |

#### CUSTOMERS

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

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

DB2 Spatial Extender uses a function, called a *geocoder*, to translate business data into spatial data.

### Using functions to generate spatial data

Spatial data can be generated not only by geocoders, but by other functions as well. Some of these functions, referred to as *constructors*, can generate spatial data from values that you provide as input. Others require existing spatial data as input. 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

## About DB2 Spatial Extender

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 shows the BRANCHES table with information that is supplied by ST\_Buffer.

### BRANCHES

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

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

In addition to ST\_Buffer, DB2 Spatial Extender provides several other functions that derive new spatial data from existing spatial data. For references to ST\_Buffer and these other functions, see **Related references** at the end of this section.

### Importing spatial data

A third way to obtain spatial data is to import it 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

This section summarizes several basic concepts that underlie the operations of DB2<sup>®</sup> Spatial Extender: geographic features, spatial information, spatial data, and geometries.

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 library 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* in the DB2 library, 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 numerical 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, dimension, and other properties

## Introduction

---

## Chapter 2. About Geometries

This chapter discusses entities of information, called geometries, that consist of coordinates and represent geographic features. The topics covered are:

- Geometries
- Properties of geometries

---

### Geometries

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<sup>®</sup> Spatial Extender, the *operational* definition of geometry is "a model of a geographic feature." 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". Six members of the hierarchy are instantiable; they can be defined with specific coordinate values and rendered visually as shown in the examples in the figure.

## About Geometries

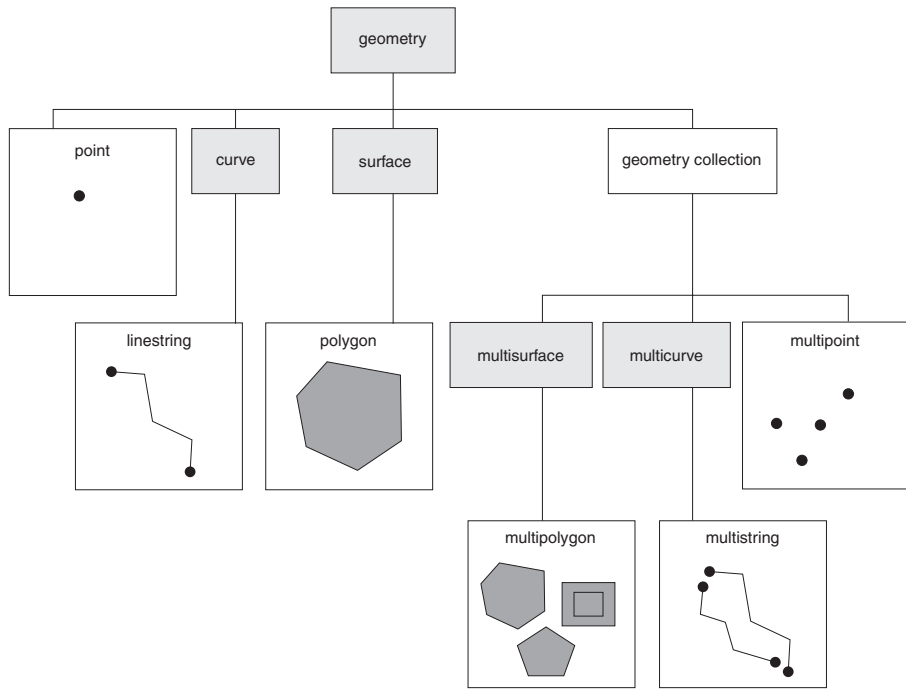


Figure 5. Hierarchy of geometries supported by DB2 Spatial Extender. Instantiable geometries in this figure include examples of how they might be rendered visually.

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

The base geometries include:

**Points** 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 large-scale 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

Linestrings represent linear geographic features (for example, streets, canals, and pipelines)

### Polygons

Polygons represent multisided geographic features (for example, welfare districts, forests, and wildlife habitats).

The homogeneous collections include:

### **Multipoints**

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**

Multilinestrings represent multipart features that are made up (for example, river systems and highway systems).

### **Multipolygons**

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.

### **Related concepts:**

- “Spatial data” on page 4

---

## **Properties of geometries**

This topic describes geometries’ properties . 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 a geometry’s associated spatial reference system

### **Type**

Each geometry belongs to a type in the hierarchy shown in the figure.

## About Geometries

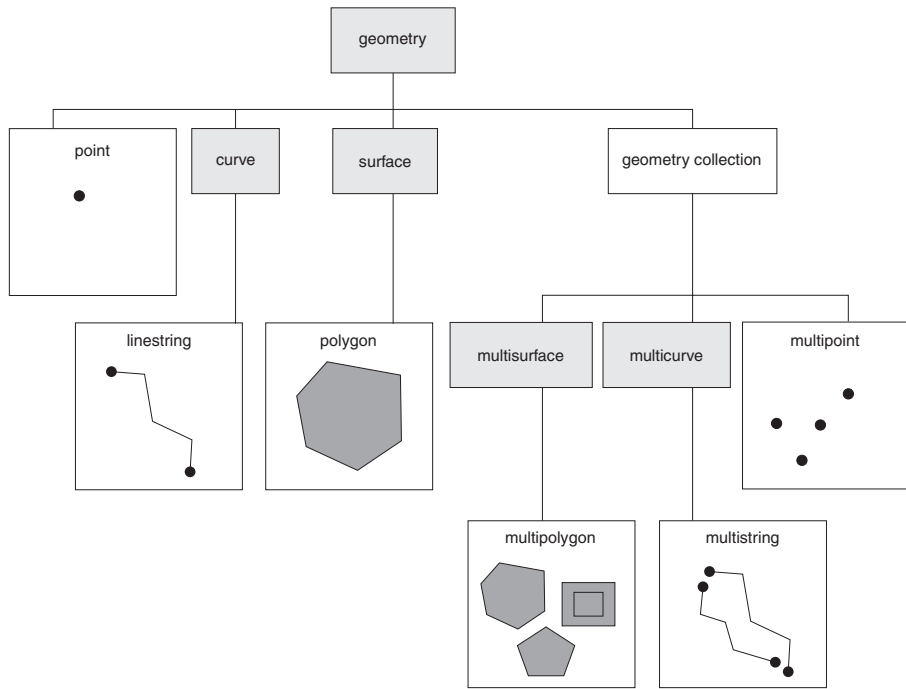


Figure 6. Hierarchy of geometries supported by DB2 Spatial Extender. Instantiable geometries in this figure include examples of how they might be rendered visually.

Seven types in the hierarchy—points, linestrings, polygons, geometry collections, multipoints, multilinestrings, and multipolygons—are instantiable. 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.

### Geometry coordinates

All geometries include at least one X coordinate and one Y coordinate, unless they are empty geometries in which 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 discuss:

- 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

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

## 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 GIS. If you want your application to process values that denote linear distances or mileposts, you can store these values along with the coordinates that define 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 interior, boundary, and exterior. The exterior of a geometry is all space not occupied by the geometry. The boundary of a geometry serves as the interface between its interior and exterior. The interior is the space occupied by the geometry.

## Simple or non-simple

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.

## Empty or not empty

A geometry is empty if it does not have any points. The envelope, boundary, interior, and exterior of an empty geometry are 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:

- 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

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

## About Geometries

- 1 Is empty
- 0 Has no length and an area of 0 (zero)
- 1 Has a length larger than 0 (zero) and an area of 0 (zero)
- 2 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 subtypes linestring and multilinestring have a dimension of one. They store road segments, branching river systems and any other features that are linear in nature.

Polygon and multipolygon 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

---

### How to use DB2 Spatial Extender

Support and use of DB2<sup>®</sup> Spatial Extender involves two main activities: setting up DB2 Spatial Extender and carrying out projects that use spatial data. This chapter introduces the interfaces through which you can perform these activities and the tasks that you execute in the course of performing them.

#### Interfaces to DB2 Spatial Extender and associated functionality

Several interfaces let you set up DB2 Spatial Extender and create projects that use spatial data. These interfaces are:

- The DB2 Control Center windows, notebooks, and menu choices that support DB2 Spatial Extender.
- A command line processor (CLP) provided by DB2 Spatial Extender. It is called the *db2se CLP*.
- Application programs that call DB2 Spatial Extender's stored procedures.

Other interfaces let you generate spatial information. They include:

- SQL queries that you submit from a DB2 CLP, from a query window in the DB2 Control Center, or from an application program.
- Visualization tools that render spatial information in graphical form. An example is ArcExplorer Java, which was created by the Environmental Systems Research Institute (ESRI) for IBM. ArcExplorer Java<sup>™</sup> can be downloaded from the DB2 Spatial Extender Web site.

#### Tasks that you perform to set up DB2 Spatial Extender and create projects

This section provides an overview of the tasks you perform to set up DB2 Spatial Extender and carry out projects that use spatial data. It includes a scenario that illustrates the tasks. The tasks fall into two categories:

- Setting up DB2 Spatial Extender
- Creating projects that use spatial data

##### Setting up DB2 Spatial Extender:

This section lists the tasks that you perform to set up DB2 Spatial Extender and uses a scenario to illustrate how a fictional company might approach each task.

## How to use DB2 Spatial Extender

### 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).

**Scenario:** The Safe Harbor Real Estate Insurance Company's information systems environment includes a DB2 Universal 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 Version 8 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.

2. Install DB2 Spatial Extender.

**Scenario:** The spatial administration team installs DB2 Spatial Extender on a UNIX machine in a DB2 environment.

3. If you have DB2 Spatial Extender Version 7, migrate your spatial data to DB2 Version 8.

**Scenario:** The Version 8 release is the first one that Safe Harbor has acquired. No migration is needed.

4. Configure your database to accommodate spatial data. You adjust configuration parameters to ensure that your database has enough memory and space for spatial functions, log files, and DB2 Spatial Extender applications.

**Scenario:** A member of the spatial administration team adjusts the UDF shared memory size, transaction log characteristics, application heap size, and application control heap size to values suited to DB2 Spatial Extender's requirements.

5. Set up spatial resources for your database. These resources include a system catalog, spatial data types, spatial functions, a geocoder, and other objects. The task of setting up these resources is referred to as *enabling the database for spatial operations*.

The geocoder supplied by DB2 Spatial Extender translates United States addresses into spatial data. It is called DB2SE\_USA\_GEOCODER.. Your organization and others can provide geocoders that translate addresses outside the United States and other kinds of data into spatial data.

**Scenario:** The spatial administration team sets up resources that will be required by the projects that it is planning.

- A member of the team issues a 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.
- Because Safe Harbor is starting to extend its business into Canada, the spatial administration team begins soliciting Canadian vendors for geocoders that translate Canadian addresses into spatial data. Safe Harbor does not expect to acquire such geocoders for a few months yet. Therefore the first locations on which it will gather data will be in the United States.

### Creating projects that use spatial data:

After you set up DB2 Spatial Extender, you are ready to undertake projects that use spatial data. This section lists the tasks involved in creating such a project and continues the scenario in which the Safe Harbor Real Estate Insurance Company seeks to integrate business and spatial data.

### 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).

**Scenario:** 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)

## How to use DB2 Spatial Extender

- This particular project will be concerned with customers and offices in the United States. Therefore, the spatial administration team decides to:
  - Use a coordinate system for the United States that DB2 Spatial Extender provides. It is called GCS\_NORTH\_AMERICAN\_1983.
  - Use DB2SE\_USA\_GEOCODER, because it is designed to geocode United States addresses.
- The spatial administration team decides what data is needed to meet the project's goals and what tables will contain this data.

### 2. Create a coordinate system if you need to do so.

**Scenario:** Having decided to use GCS\_NORTH\_AMERICAN\_1983, Safe Harbor can ignore this step.

### 3. 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.

**Scenario:** DB2 Spatial Extender provides a spatial reference system, NAD83\_SRS\_1, that is designed to be used with GCS\_NORTH\_AMERICAN\_1983. The spatial administration team decides to use NAD83\_SRS\_1.

### 4. 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.

**Scenario:** The spatial administration team defines columns to contain spatial data.

- The team adds a LOCATION column to the CUSTOMERS table. The table already contains customers' addresses. DB2SE\_USA\_GEOCODER will translate them into spatial data. Then DB2 will store this data in the LOCATION column.

- The team creates an OFFICE\_LOCATIONS table and an OFFICE\_SALES table to contain data that is now stored in the separate file system. 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.
5. Set up spatial columns for access by visualization tools, as needed. You do this 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.

**Scenario:** 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.

6. Populate spatial columns:

For a project that requires spatial data to be imported, import the data.

For a project that requires a geocoder:

- Set, in advance, the control information needed when a geocoder is invoked.
- As an option, set up the geocoder to run automatically each time a new address is added to the database, or an existing address is updated.

Run the geocoder in batch mode, as needed.

For a project that requires spatial data to be created by a spatial function, execute this function.

**Scenario:** The spatial administration team populates the CUSTOMER table’s LOCATION column, the OFFICE\_LOCATIONS table, the OFFICE\_SALES table, and a new HAZARD\_ZONES table:

- The team uses DB2SE\_USA\_GEOCODER to geocode addresses in the CUSTOMER table. The coordinates produced by the geocoding are inserted into the table’s LOCATION column.
- The team uses a utility to load office data from the filesystem into a file. Then the team imports this data to the new OFFICE\_LOCATIONS table.
- 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.

## How to use DB2 Spatial Extender

- 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.

**Scenario:** The spatial administration team creates indexes for the registered columns. It then creates a view that joins columns from the CUSTOMERS and HAZARD ZONES tables. Next, it registers the spatial columns in this view.

- Generate spatial information and related business information. Analyze the 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.

**Scenario:** The spatial analysis team runs queries to obtain information that will help it meet the original goals: to determine where to establish new branch offices, and to adjust premiums on the basis of customers' proximity to hazard areas.

### Related concepts:

- "How to create spatial grid indexes" on page 106

### Related tasks:

- "Setting up and configuring Spatial Extender" on page 27
- "Enabling a database for spatial operations" on page 60
- "Registering a geocoder" on page 62
- "Configuring a database to accommodate spatial data" on page 53
- "Importing shape data to a new or existing table" on page 91
- "Importing SDE transfer data to a new or existing table" on page 92
- "Setting up geocoding operations" on page 98
- "Setting up a geocoder to run automatically" on page 101
- "Running a geocoder in batch mode" on page 102
- "Exporting data to an SDE transfer file" on page 95
- "Selecting or creating coordinate systems" on page 73
- "Selecting or creating spatial reference systems" on page 76
- "Setting up spatial columns for access by visualization tools" on page 86
- "Creating spatial columns" on page 86
- "Calling DB2 Spatial Extender stored procedures from an application" on page 132



- “Including the DB2 Spatial Extender header file in spatial applications” on page 131

**Related reference:**

- “Invoking commands for setting up DB2 Spatial Extender and developing projects” on page 123

## How to use DB2 Spatial Extender

---

## Part 2. Setting up DB2 Spatial Extender



---

## Chapter 4. Getting started with DB2 Spatial Extender

This chapter provides instructions for installing and configuring Spatial Extender for AIX, HP-UX, Windows NT, Window 2000, Linux, Linux for OS/390, and Solaris Operating Environments. This chapter also explains how to trouble shoot some of the installation and configuration problems that you might encounter as you invoke Spatial Extender.

---

### Setting up and configuring Spatial Extender—Steps

This section contains details for the following topics:

- System requirements for installing Spatial Extender
- Installation instructions for UNIX and Windows platforms
- Explanation on how to create a DB2 Spatial Extender instance environment
- Verifying the Spatial Extender installation
- Troubleshooting tips for the installation sample program

### Setting up and configuring Spatial Extender

A DB2 Spatial Extender system consists of DB2 Universal Database, DB2 Spatial Extender, and, for most applications, a geobrowser. A geobrowser is not required but is useful for visually rendering the results of spatial queries, generally in the form of maps. Databases enabled for spatial operations are located on the server. You can use client applications to access spatial data through the DB2 Spatial Extender stored procedures and spatial queries. You can also configure DB2 Spatial Extender in a stand-alone environment, which is a configuration where both the client and server reside on the same machine. In both client-server and stand-alone configurations, you can view spatial data with a geobrowser such as ArcExplorer for DB2 or ESRI's ArcGIS tool suites running with ArcSDE.

You can download a free copy of ArcExplorer for DB2 from IBM's DB2 Spatial Extender Web site at: <http://www.ibm.com/software/data/spatial/>.

DB2 Spatial Extender can be installed on: Windows, AIX , HP-UX, Solaris Operating Environment, Linux, and Linux for OS/390.

#### Prerequisites:

Before you set up DB2 Spatial Extender, you must have DB2 software installed and configured on the client and the server.

## Getting started

### Procedure:

To set up Spatial Extender:

1. Ensure that your system meets all software requirements.
2. Install Spatial Extender. The steps vary depending on your operating system:
  - Windows
  - AIX
  - HP-UX
  - Solaris Operating Environment
  - Linux and Linux for OS/390Linux and Linux for OS/390
3. For UNIX platforms:Create a DB2 Spatial Extender instance environment.
4. Verify the installation.Verify the installation.
5. If necessary, see the troubleshooting tips and take appropriate actions to correct any problems.

### Related concepts:

- “System requirements for installing Spatial Extender” on page 28

### Related tasks:

- “Installing DB2 Spatial Extender for Windows” on page 30
- “Installing DB2 Spatial Extender for AIX” on page 32
- “Installing DB2 Spatial Extender for HP-UX” on page 34
- “Installing DB2 Spatial Extender for Solaris Operating Environment” on page 36
- “Installing DB2 Spatial Extender for Linux and Linux OS/390” on page 38
- “Creating the DB2 Spatial Extender instance environment” on page 40
- “Verifying the Spatial Extender installation” on page 42
- “Troubleshooting tips for the installation sample program” on page 44
- “Downloading ArcExplorer” on page 45

### Related reference:

- “CD-ROMs for DB2 Spatial Extender data and maps” on page 47

## System requirements for installing Spatial Extender

DB2<sup>®</sup> Spatial Extender can be installed on Windows, AIX, HP-UX, Solaris Operating Environment, Linux, and Linux for OS/390.

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

**Software requirements:**

To install Spatial Extender, you must have the following DB2 software installed and configured on the server:

**Server software**

DB2 Universal Database™ Enterprise Server Edition Version 8.1, DB2 Universal Database Workgroup Server Edition Version 8.1, or DB2 Universal Database Personal Edition Version 8.1. One of these products must be installed on your system *before* you install DB2 Spatial Extender. If you plan to use the DB2 Control Center, create and configure the DB2 Administration Server (DAS). For more information on creating and configuring DAS, see the *IBM® DB2 Universal Database Administration Guide: Implementation*.

**Note:** Although you can use DB2 Spatial Extender with the DB2 Universal Database Enterprise Server Edition, the spatial index cannot be partitioned across multiple nodes like in the massive parallel processing (MPP) environment.

**Note:** The spatial client is included in the administration and application development client.

**Disk space requirements:**

To install Spatial Extender, your system must meet the disk space requirements listed in the following table.

*Table 1. Disk space requirements for DB2 Spatial Extender*

| DB2 Spatial Extender software   | Disk space  |
|---|---|
| Server software for DB2 Spatial Extender:   | 594 MB total disk space:  |
| <ul style="list-style-type: none"> <li>• Spatial Extender server library code, sample geocoder reference data, and documentation</li> <li>• Optional and available on a separate CD-ROM: geocoder reference data (United States)</li> </ul> | <ul style="list-style-type: none"> <li>• 31 MB (DB2 Spatial Extender server library code, sample geocoder reference data, and documentation)</li> <li>• 563 MB (United States geocoder reference data)</li> </ul> |

Table 1 specifies the disk space required when you install DB2 Universal Database and DB2 Spatial Extender in a typical installation for Windows® or with pre-selected components in AIX, HP-UX, Solaris Operating Environment, Linux, and Linux 390. If you are installing DB2 Spatial Extender or have installed DB2 Universal Database with a different installation type, your disk space calculations will differ.

## Getting started

When your system meets all the software and disk space requirements, you can install Spatial Extender.

### Installing DB2 Spatial Extender for Windows

This task is part of the larger task of Setting up DB2 Spatial Extender.

You can install DB2 Spatial Extender on Windows operating systems by using the DB2 Setup wizard or a response file.

Recommendation: use the DB2 Setup wizard to install Spatial Extender. The setup wizard provides an easy-to-use graphical interface with installation help, automated user and group creation, protocol configuration, and instance creation.

If you are using the DB2 Setup wizard to install Spatial Extender, you can click **Cancel** at any point during the installation to exit the process.

#### Prerequisites:

Before you install the DB2 Spatial Extender product, you must have a DB2 Version 8 server product installed.

**Note:** The Spatial Extender client and sample components are available in the DB2 client and server installs, so you do not need to have a DB2 server if you just need Spatial client functionality.

#### Procedure:

To install Spatial Extender for Windows using the DB2 Setup wizard:

1. Insert the Spatial Extender CD-ROM into the CD-ROM drive. The DB2 Setup Launchpad, an interface from which you can install DB2 Spatial Extender, opens.
2. Click **Install products**.
3. Select DB2 Spatial Extender as the product you want to install and click **NEXT**. The DB2 Setup wizard launches. Click **NEXT**. Use the DB2 Setup wizard to guide you through setup, and through the remaining installation steps. At any time during the installation, you can click **Help** to launch the online installation help.

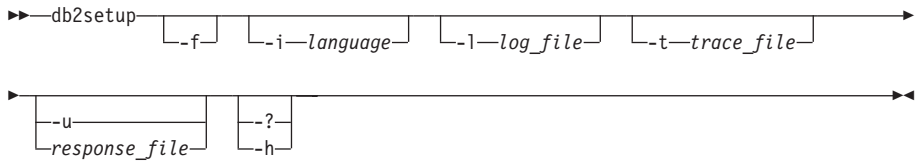
To install Spatial Extender for Windows using the response file:

1. Log on to the system with a user account that you want to use to perform the installation.
2. Insert the Spatial Extender CD-ROM. See the *DB2 Installation and Configuration Supplement* for more information.



3. Run the setup program by issuing the following from a command prompt:

**db2setup command**



Where:

- f** Forces any DB2 processes to stop before installing.
- i (language)**  
Is the two letter language code of the language in which to perform the installation.
- l (log\_file )**  
Is the full path and file name of the log file to use.
- t (trace\_file)**  
Generates a fully qualified file with install trace information.
- u (response\_file)**  
Specifies the fully qualified response file name. If you changed and renamed the sample response file that is provided, make sure that this parameter matches the new name. This parameter is required. The response file is located at `db2\Windows\samples\db2gse.rsp` on your DB2 Spatial Extender installation CD.
- , -h** Generates usage information.

4. After the installation is complete, check the messages in the log file

**Related concepts:**

- “System requirements for installing Spatial Extender” on page 28

**Related tasks:**

- “Creating a response file on Windows” in the *Installation and Configuration Supplement*
- “Setting up and configuring Spatial Extender” on page 27
- “Verifying the Spatial Extender installation” on page 42
- “Troubleshooting tips for the installation sample program” on page 44

## Getting started

### Installing DB2 Spatial Extender for AIX

You can install DB2 Spatial Extender for AIX by using the DB2 Setup wizard, by using the `db2_install` script, or by using the System Management Interface Tool (SMIT).

**Recommendation:** use the DB2 Setup wizard to install Spatial Extender. The Setup wizard provides an easy-to-use graphical interface with installation help, automated user and group creation, protocol configuration, and instance creation. If you choose not to use the wizard, you can install Spatial Extender by using the `db2_install` script or by using AIX's System Management Interface Tool (SMIT). Using SMIT to install Spatial Extender is only recommended for advanced users in a situations where greater manual control over the setup process is required.

#### Prerequisites:

Before you install Spatial Extender on AIX:

- Ensure your system meets all software, memory, and disk space requirements.
- Update the kernel configuration parameters and restart the system for all DB2 clients and servers on AIX.
- You must have a DB2 Version 8 server product installed if you are installing in a server or standalone environment.

**Note:** The DB2 Spatial Extender client and sample components are available in the DB2 client and server. If you just need spatial client functionality, you do not need to install DB2 Spatial Extender.

- You must have root authority.

#### Procedure:

To install Spatial Extender using the DB2 Setup wizard:

1. Log in as a user with root authority.
2. Insert and mount the Spatial Extender CD-ROM. The DB2 Setup Launchpad, an interface from which you can install DB2 Spatial Extender, opens. For information on how to mount a CD-ROM, see the *DB2 Installation and Configuration Supplement*.
3. Select DB2 Spatial Extender as the product you want to install and click **NEXT**.
4. The DB2 Setup wizard window opens. Use the DB2 Setup wizard to guide you through setup, and through the remaining installation steps. At any time during the installation, you can click **Help** to launch the online installation help.

To install DB2 Spatial Extender using the `db2_install` script:

1. Log in as a user with root authority.
2. Insert and mount the appropriate CD-ROM.
3. Enter the `./db2_install` command to start the `db2_install` script. The `db2_install` script can be found in the root directory on your DB2 Version 8 product CD-ROM. The `db2_install` script prompts you for the product keyword.
4. Type **DB2.GSE** to install DB2 Spatial Extender.

To install DB2 Spatial Extender using the System Management Interface Tool (SMIT):

1. Log in as a user with root authority.
2. Insert and mount the Spatial Extender CD-ROM.
3. Enter the `smit install_latest` command.
4. Type `/cdrom/db2` in the **INPUT device/directory** for the software field.
5. Click **DO** or press Enter to verify that the installation directory exists.
6. In the **Software to install** field, identify whether the client or server components are to be installed.

**Note:** Refer to the `ComponentList.htm` file on DB2 Spatial Extender CD for a complete list of the components that you should install for DB2 Spatial Extender.

7. Click **DO** or press Enter. You are prompted to confirm the installation parameters.
8. Press Enter to confirm.
9. Log out.

When the installation is complete, Spatial Extender will be installed in the `/usr/opt/db2_08_01` directory along with your other DB2 products.

After you install Spatial Extender, create your DB2 instance environment if you did not already do so, and then verify the installation.

### Related concepts:

- “System requirements for installing Spatial Extender” on page 28

### Related tasks:

- “Installing a DB2 product on AIX using SMIT” in the *Installation and Configuration Supplement*
- “Mounting the CD-ROM on AIX” in the *Installation and Configuration Supplement*

## Getting started

- “Installing a DB2 product using the db2\_install script” in the *Installation and Configuration Supplement*
- “Verifying the Spatial Extender installation” on page 42
- “Troubleshooting tips for the installation sample program” on page 44

### Installing DB2 Spatial Extender for HP-UX

You can install Spatial Extender using the DB2® Setup wizard, by using the db2\_install script, or by using the **swinstall** command.

Recommendation: use the DB2 Setup wizard to install Spatial Extender. The setup wizard provides an easy-to-use graphical interface with installation help, automated user and group creation, protocol configuration, and instance creation. If you choose not to use the wizard, you can install Spatial Extender for HP-UX by using the db2\_install script or by using the **swinstall** command. Using the HP-UX **swinstall** command to install Spatial Extender is only recommended for advanced users in situations where greater manual control over the setup process is required.

#### Prerequisites:

Before you install the DB2 Spatial Extender product for HP-UX:

- Ensure your system meets all hardware, software, and memory requirements.
- You must have a DB2 Version 8 server product installed.

**Note:** The DB2 Spatial Extender client and sample components are available in the DB2 client and server. If you just need spatial client functionality, you do not need to install DB2 Spatial Extender.

- Update the kernel configuration parameters and restart the system for all DB2 clients and servers on HP-UX
- You must have root authority.

#### Procedure:

To install Spatial Extender for HP-UX using the DB2 Setup wizard:

1. Insert and mount the DB2 Spatial Extender CD-ROM. The DB2 Setup Launchpad, an interface from which you can install DB2 Spatial Extender, opens.
2. Select DB2 Spatial Extender as the product you want to install and click **NEXT**. The DB2 Setup wizard launches. Click **NEXT**. Use the DB2 Setup wizard to guide you through setup, and through the remaining installation steps. At any time during the installation, you can click **Help** to launch the online installation help.

To install Spatial Extender for HP-UX using the `db2_install` script:

1. Log in as a user with root authority.
2. Insert and mount the appropriate CD-ROM.
3. Enter the `./db2_install` command to start the `db2_install` script. The `db2_install` script can be found in the root directory on your DB2 Version 8 product CD-ROM. The `db2_install` script prompts you for the product keyword.
4. Type **DB2.GSE** to install DB2 Spatial Extender.

To install Spatial Extender for HP-UX using the `swinstall` command:

1. Log in as a user with root authority.
2. Insert and mount the Spatial Extender CD-ROM.
3. Run the `swinstall` program using the following command:  

```
swinstall -x autoselect_dependencies=true
```

This command opens the Software Selection window and the Specify Source window. If necessary, change the value in the **Source host name** field in the Specify Source window.

4. In the **Source Depot Path field** enter `/cdrom/db2/hpux` where `/cdrom` represents the CD-ROM mount directory.
5. Click **OK** to return to the Software Selection window. The Software Selection window contains a list of available software to install.
6. Select the products you are licensed to install.
7. Select **Mark for Install** from the **Actions** menu to choose the product to be installed. A message appears:  
In addition to the software you just marked, other software was automatically marked to resolve dependencies. This message will not appear again.
8. Select **OK**.
9. Select **Install (analysis)** from the **Actions** menu to begin installing the product and to open the Install Analysis window.
10. Select **OK** in the Install Analysis window when the **Status** field displays a Ready message.
11. Select **Yes** in the Confirmation windows to confirm that you want to install the software.

View the Install window to read processing data while the software is being installed, until the **Status** field indicates Ready and the Note window opens. The `swinstall` program loads the file set, and runs the control scripts for the file set.

12. Select **Exit** from the **File** menu to exit from `swinstall`.

## Getting started

When the installation is complete, Spatial Extender will be installed in the /opt/IBM/db2/V8.1 directory along with your other DB2 products.

After you install Spatial Extender, create your DB2 instance environment if you did not already do so, and then verify the installation.

### Related concepts:

- “System requirements for installing Spatial Extender” on page 28

### Related tasks:

- “Installing a DB2 product on HP-UX using swinstall” in the *Installation and Configuration Supplement*
- “Mounting the CD-ROM on HP-UX” in the *Installation and Configuration Supplement*
- “Installing a DB2 product using the db2\_install script” in the *Installation and Configuration Supplement*
- “Verifying the Spatial Extender installation” on page 42
- “Troubleshooting tips for the installation sample program” on page 44

## Installing DB2 Spatial Extender for Solaris Operating Environment

You can install Spatial Extender using the DB2<sup>®</sup> Setup wizard, by using the db2\_install script, or by using the **pkgadd** command.

Recommendation: use the DB2 Setup wizard to install DB2 Spatial Extender. The setup wizard provides an easy-to-use graphical interface with installation help, automated user and group creation, protocol configuration, and instance creation. If you choose not to use the wizard, you can install Spatial Extender using the db2\_install script or by using the Solaris Operating Environment **pkgadd** command. Using the Solaris Operating Environment **pkgadd** command is only recommended for advanced users in situations where greater manual control over the setup process is required.

DB2 Spatial Extender is made up of different functions and components that are referred to as packages in the Solaris environment. When you install Spatial Extender using the **pkgadd** command, you must install each required package and each associated package for the optional functions that you want to use. The ComponentList.htm file on your DB2 Spatial Extender CD has a complete list of the packages that you should install for DB2 Spatial Extender. The ComponentList.htm file is located in /cdrom/db2/solaris where /cdrom is the mount point for your DB2 Spatial Extender CD-ROM.

### Prerequisites:

Before you install the DB2 Spatial Extender product for Solaris Operating Environments:

- Ensure your system meets all hardware, software, and memory requirements.
- You must have a DB2 Version 8 server product installed if you are installing in a server or stand-alone environment.

**Note:** The DB2 Spatial Extender client and sample components are available in the DB2 client and server. If you just need spatial client functionality, you do not need to install DB2 Spatial Extender.

- Update the kernel configuration parameters and restart the system for all DB2 clients and servers on Solaris.
- You must have root authority.

### Procedure:

To install DB2 Spatial Extender for Solaris Operating Environments using the DB2 Setup wizard:

1. Log in as a user with root authority.
2. Insert and mount your DB2 Spatial Extender CD-ROM. The DB2 Setup Launchpad, an interface from which you can install DB2 Spatial Extender opens. For information on how to mount a CD-ROM, see *DB2 for UNIX Quick Beginnings*.
3. Select **Spatial Extender** as the product you want to install and click **NEXT**.
4. The DB2 Setup wizard launches. Use the DB2 Setup wizard to guide you through setup, and through the remaining installation steps. At any time during the installation, you can click **HELP** to launch the online installation help.

To install DB2 Spatial Extender using the `db2_install` script:

1. Log in as a user with root authority.
2. Insert and mount the appropriate CD-ROM.
3. Enter the `./db2_install` command to start the `db2_install` script. The `db2_install` script can be found in the root directory on your DB2 Version 8 product CD-ROM. The `db2_install` script prompts you for the product keyword.
4. Type **DB2.GSE** to install DB2 Spatial Extender.

To install DB2 Spatial Extender for Solaris using the `pkgadd` command:

1. Log in as a user with root authority.
2. Insert and mount the DB2 Spatial Extender CD-ROM.

## Getting started

3. Identify the required packages and optional packages that you want to install. See the ComponentList.htm file on your CD for a complete list of the components that you should install for DB2 Spatial Extender.
4. Run the **pkgadd** command for each package that you want to install by typing:

```
pkgadd package_name
```

Where *package\_name* is the package that you want to install.

For example, if you want to install the Spatial Extender Base Server Support, you would need to install the db2gssg81 package by entering the following command:

```
pkgadd db2gssg81
```

When the installation is complete your Spatial Extender software will be installed in the /opt/IBM/db2/V8.1 directory.

After you install Spatial Extender, create your DB2 instance environment if you did not already do so, and then verify the installation.

### Related concepts:

- “System requirements for installing Spatial Extender” on page 28

### Related tasks:

- “Installing a DB2 product on Solaris using pkgadd” in the *Installation and Configuration Supplement*
- “Mounting the CD-ROM on Solaris” in the *Installation and Configuration Supplement*
- “Installing a DB2 product using the db2\_install script” in the *Installation and Configuration Supplement*
- “Verifying the Spatial Extender installation” on page 42
- “Troubleshooting tips for the installation sample program” on page 44

## Installing DB2 Spatial Extender for Linux and Linux OS/390

You can install DB2 Spatial Extender for Linux and Linux OS/390 by using the DB2 Setup wizard, by using the db2\_install script, or by using the **rpm** command.

Recommendation: use the DB2 Setup wizard to install Spatial Extender. The setup wizard provides an easy-to-use graphical interface with installation help, automated user and group creation, protocol configuration, and instance creation. If you choose not to use the wizard, you can install Spatial Extender by using the db2\_install script or by using the **rpm** command. Using the



Linux **rpm** command to install Spatial Extender is only recommended for advanced users in situations where greater manual control over the setup process is required.

### Prerequisites:

Before you install the DB2 Spatial Extender product for Linux and Linux OS/390:

- Ensure your system meets all hardware, software, and memory requirements.
- You must have a DB2 Version 8 server product installed if you are installing in a server or stand alone environment.

**Note:** The DB2 Spatial Extender client and sample components are available in the DB2 client and server. If you just need spatial client functionality, you do not need to install DB2 Spatial Extender.

- Update the kernel configuration parameters and restart the system for all DB2 clients and servers on HP-UX
- You must have root authority.

### Procedure:

To install DB2 Spatial Extender using the DB2 Setup wizard:

1. Log in as a user with root authority.
2. Insert and mount your DB2 Spatial Extender CD-ROM. The DB2 Setup Launchpad, an interface from which you can install DB2 Spatial Extender opens. For information on how to mount a CD-ROM, see the *DB2 Installation and Configuration Supplement*.
3. Click **Install Products**.
4. Select **Spatial Extender** as the product you want to install and click **NEXT**.
5. The DB2 Setup wizard window opens. Use the DB2 Setup wizard to guide you through setup, and through the remaining installation steps. At any time during the installation, you can click **Help** to launch the online installation help.

To install DB2 Spatial Extender using the db2\_install script:

1. Log in as a user with root authority.
2. Insert and mount the appropriate CD-ROM.
3. Enter the **./db2\_install** command to start the db2\_install script. The db2\_install script can be found in the root directory on your DB2 Version 8 product CD-ROM. The db2\_install script prompts you for the product keyword.
4. Type **DB2.GSE** to install DB2 Spatial Extender.

## Getting started

To install Spatial Extender for Linux using the **rpm** command:

1. Log in as a user with root authority.
2. Enable your system for DB2 for Linux installation. See the *DB2 Installation and Configuration Supplement*
3. Insert and mount the DB2 Spatial Extender CD-ROM.
4. Identify the required packages and optional packages you want to install .

**Note:** Refer to the ComponentList.htm file on your DB2 Spatial Extender CD for a complete list of the components that you should install for DB2 Spatial Extender.

5. Run the **rpm** command for each package you want to install:

```
rpm -ivh package_name
```

For example, if you want to install the server you would need to install the IBM\_db2gssg81-8.1.0-0.i386.rpm package by entering the following command:

```
rpm -IBM_db2gssg81-8.1.0-0.i386.rpm
```

When the installation is complete, Spatial Extender will be installed in the /opt/IBM/db2/V8.1 directory along with your other DB2 products.

After you install Spatial Extender, create your DB2 instance environment if you did not already do so, and then verify the installation.

### Related concepts:

- “System requirements for installing Spatial Extender” on page 28

### Related tasks:

- “Installing a DB2 product on Linux using rpm” in the *Installation and Configuration Supplement*
- “Mounting the CD-ROM on Linux” in the *Installation and Configuration Supplement*
- “Installing a DB2 product using the db2\_install script” in the *Installation and Configuration Supplement*
- “Verifying the Spatial Extender installation” on page 42
- “Troubleshooting tips for the installation sample program” on page 44

## Creating the DB2 Spatial Extender instance environment

This task is part of the larger task of setting up Spatial Extender.

This section is only applicable for UNIX platforms.

DB2 Spatial Extender can be used with any DB2 instance that is created after installing the Spatial Extender code.

The **db2icrt** command is used to create new DB2 instances. All new DB2 instances that you create after installing DB2 Spatial Extender include DB2 Spatial Extender in the instance environment.

DB2 instances created before you install Spatial Extender do not include DB2 Spatial Extender in their instance environments. To update existing DB2 instances, use the **db2iupdt** command. If you are using the DB2 Control Center and created an instance for the DB2 Administration server prior to installing DB2 Spatial Extender, then you must update this instance.

### Procedure:

To update an instance using the **db2iupdt** command:

1. Log in as a user with root authority.
2. Run the following command:

```
DB2DIR/instance/db2iupdt -a AuthType -u FencedID InstName
```

Where:

#### **DB2DIR**

the DB2 installation directory.

- On AIX, the DB2 installation directory is /usr/opt/db2\_08\_01
- On all other UNIX-based operating systems, the installation directory is /opt/IBM/db2/V8.1

#### **-a AuthType**

Represents the authentication type for the instance. AuthType can be one of SERVER, CLIENT, DCS, SERVER\_ENCRYPT, DCS\_ENCRYPT. SERVER is the default. This parameter is optional.

#### **-u FencedID**

Represents the name of the user under which fenced user defined functions (UDFs) and fenced stored procedures will run. This flag is not required if you are creating an instance on a DB2 client. Specify the name of the fenced user you created.

#### **InstName**

Represents the name of instance. The name of the instance must be the same as the name of the instance owning user. Specify the name of the instance owning user you created. The instance will be created in the instance owning user's home directory.

To create an instance using **db2icrt**:

1. Log in as user with root authority.

## Getting started

2. Run the following command:

```
DB2DIR/instance/db2icrt -a AuthType -u FencedID InstName
```

Where:

### **DB2DIR**

the DB2 installation directory.

- On AIX, the DB2 installation directory is `/usr/opt/db2_08_01`
- On all other UNIX-based operating systems, the installation directory is `/opt/IBM/db2/V8.1`

### **-a AuthType**

Represents the authentication type for the instance. AuthType can be one of SERVER, CLIENT, DCS, SERVER\_ENCRYPT, DCS\_ENCRYPT. SERVER is the default. This parameter is optional.

### **-u FencedID**

Represents the name of the user under which fenced user defined functions (UDFs) and fenced stored procedures will run. This flag is not required if you are creating an instance on a DB2 client. Specify the name of the fenced user you created.

### **InstName**

Represents the name of instance. The name of the instance must be the same as the name of the instance owning user. Specify the name of the instance owning user you created. The instance will be created in the instance owning user's home directory.

For example, if you are using server authentication, your fenced user is `db2fenc1`, and your instance owning user is `db2inst1`, use the following command to create an instance on an AIX system:

```
/usr/opt/db2_08_01/instance/db2icrt -a server -u db2fenc1 db2inst1
```

After you create an instance you may want to configure notification for health monitoring. This task can be performed using the Health Center or CLP. See the *DB2 Installation and Configuration Supplement* for more information.

## Verifying the Spatial Extender installation

This task is part of a larger task of setting up and configuring Spatial Extender.

After you install DB2 Spatial Extender, you can create a database and run the installation check program to verify that DB2 Spatial Extender is installed and configured correctly.

You can verify the installation by using the DB2 Spatial Extender sample program `runGseDemo`. The following verification steps apply to Windows, AIX, HP-UX, Solaris Operating Environments, Linux, and Linux 390, Operating systems.

### Prerequisites:

For UNIX (AIX, HP-UX, Solaris Operating Environments, Linux, and Linux 390) installations, check that you established the DB2 Spatial Extender instance environment before you run the installation check program. See the *DB2 Installation and Configuration Supplement* for information on how to run the `db2ilist` program to check your instances.

### Procedure:

To verify the installation:

1. For UNIX only: Log on as the instance owner.

2. Create a database. For example, type:

```
db2 create database mydb
```

where *mydb* is the database name.

3. Locate the installation check program.

- a. For UNIX operating systems type:

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

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

- b. For Windows type:

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

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

4. Run the installation check program. For example type:

```
runGseDemo mydb userID password
```

where *mydb* is the database name.

If you receive error messages during the verification process, you need to troubleshoot the installation.

### Related tasks:

- "Troubleshooting tips for the installation sample program" on page 44
- "Installing DB2 Spatial Extender for Windows" on page 30
- "Installing DB2 Spatial Extender for AIX" on page 32

## Getting started

- “Installing DB2 Spatial Extender for HP-UX” on page 34
- “Installing DB2 Spatial Extender for Solaris Operating Environment” on page 36
- “Installing DB2 Spatial Extender for Linux and Linux OS/390” on page 38

### Troubleshooting tips for the installation sample program

The runGseDemo sample program is designed to surface problems with your installation. During the installation verification, you might receive error messages that can help you diagnose specific system problems. Most of the error messages are caused by a small number of typical problems. To avoid these errors, ensure that you do the following task each time that you run the installation check program:

- Be sure that you installed the DB2 Spatial Extender product in the appropriate environments.
- Use a new database that does not have any spatial operations associated with it.
- Increase the database configuration parameter value for application heap size.

The remainder of this section lists the typical problem areas to check when you troubleshoot.

#### Database is already spatially enabled

Check that the database for which you are verifying the installation is new and has no spatial operations associated with it; if it does, the sample program will fail.

You will receive the following error message if the database you are running the sample program against is already spatially enabled:

```
Enabling database logtst...  
Returning from ENABLE_DB:  
Return code = -14  
Return message text =  
GSE0014E The database has already been enabled for spatial operations.
```

To fix this problem, drop the database and repeat the steps in Verifying the Spatial Extender installation.

#### Database manager configuration parameter value for application heap size

If APPLHEAPSZ is not set at adequate value, you'll get this error message while enabling database for spatial operations:

```
GSE0213N A bind operation failed.  
SQLERROR = "SQL0001N Binding or precompilation  
did not complete successfully.  
SQLSTATE=00000".SQLSTATE=57011
```

To increase the database configuration parameter value for the application heap size, type:

```
db2 update db cfg for database_name using APPLHEAPSZ 2048
```

If 2048 is inadequate, increase the APPLHEAPSZ parameter in increments of 256.

---

### Post-Installation considerations

After you install Spatial Extender, consider the following:

- Downloading ArcExplorer
- Accessing geocoder reference data

#### Downloading ArcExplorer

IBM provides a browser, produced by Environmental Systems Research Institute (ESRI) for IBM, that can directly produce visual results of queries of DB2 Spatial Extender data without requiring an intermediate data server. This browser is ArcExplorer for DB2. You can download a free copy of ArcExplorer from IBM's Spatial Extender Web site at <http://www.ibm.com/software/data/spatial/>.

For more information on installing and using ArcExplorer, see the *Using ArcExplorer* book, which is also available as part of the ArcExplorer product download on the DB2 Spatial Extender Web site.

**Important:** DB2 Universal Database Version 8.1 is shipped with IBM Java Development Kit (JDK) Version 1.3.1. When you install JRE ArcExplorer, place it in a separate directory from DB2. Remember to set the CLASSPATH environment variable.

#### Related tasks:

- "Setting up and configuring Spatial Extender" on page 27

#### Accessing geocoder reference data

The geocoder reference data on the DB2 Spatial Extender Geocoder Reference Data CD-ROM is created specifically to work with a geocoder supplied by Spatial Extender. It is composed of USA base map street network data that the DB2SE\_USA geocoder uses to determine the coordinates of addresses in a spatially enabled database. This base-map data is collectively called *reference data*. The DB2SE\_USA geocoder takes address data (non-spatial) in your database, compares and matches it with the reference data, and converts it into coordinates that can be stored by DB2 Spatial Extender. This process is called *geocoding*.

## Getting started

The reference data provided on the CD-ROM includes the EDGELocator.loc file. The EDGELocator.loc file is used by the DB2SE\_USA geocoder to locate specific reference data. For example, if you are geocoding addresses in California, Kentucky, and Oregon, the DB2SE\_USA geocoder uses the locator file on the CD-ROM to determine the address locations .

### Procedure:

You can access the geocoder data directly from the CD-ROM, or you can copy the data to your hard drive. To copy geocoder data files from the CD-ROM to your DB2 Spatial Extender server environment, perform the steps explained in this section.

For UNIX operating systems:

1. Mount the CD-ROM. For information on how to mount a CD-ROM, see *DB2 for UNIX Quick Beginnings*.
2. Log in at the target server machine as a user with root authority.
3. Type the following command:

- For AIX:

```
cp /cdrom/db2/* /usr/opt/db2_08_01/gse/refdata/
```

- For all other UNIX platforms:

```
cp /cdrom/db2/* /opt/IBM/db2/V8.1/gse/refdata/
```

**Note:** You can copy geocoder data files into any directory on your local drive. If you choose to copy the files into a directory that you specify , you must modify the locator file to point to the new location.

4. Log out.

For Windows, you can use either the Command window or Windows Explorer.

To use the Command window to access geocoder data:

1. Click **Start** -> **Program** -> **IBM DB2** -> **Command Window**.
2. Type the following command:

```
copy d:\db2\* %db2path%\gse\refdata
```

where *d*: is the letter that corresponds to your CD-ROM drive.

**Note:** You can copy geocoder data files into any directory on your local drive. If you choose to copy the files into a directory that you specify , you must modify the locator file to point to the new location.



To use the Windows Explorer to access geocoder data:

Copy all the files from *d:\db2* to *c:\Program Files\IBM\splib\gse\refdata*, where *d:* is the CD-ROM drive and *c:\Program Files\IBM\splib* is the directory where DB2 is installed.

**Related tasks:**

- “Setting up and configuring Spatial Extender” on page 27

## CD-ROMs for DB2 Spatial Extender data and maps

DB2 Spatial Extender is shipped with seven data and maps CD-ROMs.

The data and maps information, labeled DB2 Spatial Extender Data and Maps 1 – 7, is provided on seven CD-ROMs. The following table provides a summary of the data located on each CD-ROM.

*Table 2. Data and maps CD-ROM information*

| Data and Maps CD-ROM | Type of map data summary         |
|----------------------|----------------------------------|
| CD-ROM 1             | Europe and World                 |
| CD-ROM 2             | Canada, Mexico and United States |
| CD-ROM 3             | United States                    |
| CD-ROM 4             | United States (western region)   |
| CD-ROM 5             | United States (central region)   |
| CD-ROM 6             | United States (eastern region)   |
| CD-ROM 7             | United States (southern region)  |

For a detailed description of the data provided by ESRI, see the ESRI help file, *esridata.hlp*, located on the DB2 Spatial Extender Data and Maps CD-ROM.

- For Windows, view the help file in *x:esridata.hlp*, where *x:* is the CD-ROM drive.
- For UNIX operating systems, view or print the help file located on the CD-ROM in */cdrom/esridata.hlp*, where */cdrom* is your mount point.

## Getting started

---

## Chapter 5. Migrating the Spatial Extender environment to DB2 Universal Database Version 8

This section explains how you migrate DB2 Spatial Extender from Version 7 to Version 8. It also explains how you use the migration utility to migrate from a 32-bit environment to a 64-bit environment.

---

### Migrating a spatially-enabled database

If you have been using DB2 Spatial Extender Version 7, you must complete the following steps before using an existing spatially-enabled database with DB2 Spatial Extender Version 8. This topic describes the steps required to migrate spatially-enabled databases from a previous version of DB2 Spatial Extender.

#### Prerequisites:

- Terminate all connections to the database before you run the migration utility.
- Before you start the migration process, ensure that your system meets the installation requirements for DB2 Spatial Extender Version 8.
- To backup a database, you must have SYSADM, SYSCTRL, or SYSMANT authority for the database.
- To migrate a database, you must have SYSADM authority.

#### Procedure:

To migrate the DB2 Spatial Extender environment:

1. Backup your Version 7 database. For information on how to backup your database, see *DB2 Installation and Configuration Supplement*.
2. Install DB2 Universal Database Version 8 and DB2 Spatial Extender Version 8.
3. Migrate your DB2 instance and databases from Version 7 to Version 8. For more information on how to migrate your DB2 instance and databases, see *DB2 Installation and Configuration Supplement*
4. Migrate a spatially-enabled database from Version 7 to Version 8 using the Spatial Extender migration utility.
  - a. From an operating system command prompt, type the following command:

# Migrating

## db2se migrate database command

```
► db2se migrate database_name
└─ user_id ─ password ─┘
└─ table_creation_parameters ─┘
└─ force_value ─┘ └─ messages_filename ─┘
```

Where:

*-database\_name*

The name of the database to be migrated.

*-user\_id*

The database user ID which has either SYSADM or DBADM authority on the database that is being migrated.

*-password*

Your user password.

*-table\_creation\_parameters*

The parameters to be used in the creation of the Spatial Extender catalog tables.

*-force\_value*

- 0: Attempt migration but stop if any application-defined objects such as views, functions, or triggers have been based on Spatial Extender objects.
- 1: Saves and restores spatial indexes.
- 2: Automatically saves and restores application-defined objects. Saves spatial index information but does not automatically restore spatial indexes.

*-messages\_filename*

The file name containing the report of migration actions. The *messages\_file* must be a fully qualified file name on the server.

## Migrating from a 32-bit to a 64-bit environment

If you have spatial indexes that were created in a 32-bit environment and you want to migrate to a 64 bit environment, complete the following steps.

**Note:** 64-bit Windows and Linux servers are supported for testing efforts

1. Backup your database.
2. Save the spatial indexes that are defined by typing the following from an operating system command prompt:

## db2se save\_indexes command

```

▶▶ db2se—save_indexes—database_name—————▶
    └───userId—user_id— -pw—password──┘
▶
└───messagesFile—messages_filename──┘▶▶

```

Where:

*-database\_name*

The name of the database to be migrated.

*-user\_id*

The database user ID which has either SYSADM or DBADM authority on the database that is being migrated.

*-password*

Your user password.

*-messages\_filename*

The file name containing the report of migration actions. The messages\_file must be a fully qualified file name on the server.

3. Migrate your V8 database from a 32-bit to a 64-bit environment. For more information, on switching from a 32-bit environment to a 64-bit environment, see *Quick Beginnings for DB2*.
4. Restore the spatial indexes. In an operating system prompt, type the following command:

## db2se restore\_indexes command

```

▶▶ db2se—restore_indexes—database_name—————▶
    └───userId—user_id──┘
▶
└───messagesFile—messages_filename──┘▶▶

```

*-database\_name*

The name of the database to be migrated.

*-user\_id*

The database user ID which has either SYSADM or DBADM authority on the database that is being migrated.

*-messages\_filename*

The file name containing the report of migration actions. The messages\_file must be a fully qualified file name on the server.

## Migration messages

If the migration is successful, the following message displays:

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

## Migrating

If the migration is not successful, the following message displays:

GSE9002N An error occurred during an attempt to perform Spatial Extender database migration.

**Note:** The following errors may occur during migration:

- Database is not currently spatially enabled.
- Database is not a Version 7 spatially-enabled database
- Database is already a Version 8 spatially-enabled database
- Database name is not valid
- Other connections to the database exist. Can not be run.
- Spatial catalog is not consistent.
- User is not authorized
- Password is not valid
- Some user objects could not be migrated

Be sure to check the messages files for details on the error(s) you may receive. The message file also contains useful information such as indexes, views, and the geocoding setup that was migrated.

### Related tasks:

- “Backing up databases before DB2 migration” in the *Quick Beginnings for DB2 Servers*
- “Migrating databases” in the *Quick Beginnings for DB2 Servers*
- “Migrating instances (UNIX)” in the *Quick Beginnings for DB2 Servers*
- “Migrating DB2 (Windows)” in the *Quick Beginnings for DB2 Servers*
- “Migrating DB2 (UNIX)” in the *Quick Beginnings for DB2 Servers*
- “Migrating DB2 Personal Edition (Windows)” in the *Quick Beginnings for DB2 Personal Edition*
- “Migrating DB2 Personal Edition (Linux)” in the *Quick Beginnings for DB2 Personal Edition*
- “Migrating databases on DB2 Personal Edition (Windows)” in the *Quick Beginnings for DB2 Personal Edition*
- “Migrating instances and databases on DB2 Personal Edition (Linux)” in the *Quick Beginnings for DB2 Personal Edition*

---

## Chapter 6. Setting up a database

This chapter discusses how to configure a database to accommodate spatial data.

---

### Configuring a database to accommodate spatial data

DB2 Spatial Extender, which runs in the DB2 Universal Database environment, works with most default DB2 configuration values. However, several configuration parameters affect spatial operations. You must tune these parameters so that your spatial applications perform as efficiently as possible. In certain cases, choosing a value other than the default value is required for spatial operations. In other cases, doing so is recommended, depending on your applications and your overall DB2 environment. This chapter identifies the DB2 configuration parameters that influence the operations of DB2 Spatial Extender.

The following sections explain how to tune the DB2 database manager and database configuration parameters that affect DB2 Spatial Extender operations.

---

### Tuning the database configuration parameters

Several database configuration parameters affect spatial applications. To modify any database configuration parameter, you must be connected to the database. When you modify the values of these parameters for a database, the change affects only that database. The following sections explain how to tune the parameters for spatial applications:

- “Tuning transaction log characteristics”
- “Tuning the application heap size” on page 55
- “Tuning the application control heap size” on page 56

#### Tuning transaction log characteristics

Before you enable a database for spatial operations, ensure that you have enough transaction log capacity. The default values for the transaction log configuration parameters do not provide sufficient transaction log capacity if your plans include:

- Enabling a database for spatial operations in a Windows environment
- Using the ST\_import\_shape stored procedure to import from shape files
- Using geocoding with a large commit scope
- Running concurrent transactions

## Setting up a database

If your plans include any of these uses now or in the future, you need to increase the capacity of your transaction log for the database by increasing one or more of the transaction log configuration parameters. Otherwise, you can use the default characteristics. In this case, proceed to “Tuning the application heap size” on page 55.

**Recommendation:** Refer to the following table for the recommended minimum values for the three transaction log configuration parameters.

*Table 3. Recommended minimum values for transaction configuration parameters*

| Parameter  | Description   | Default value | Recommended minimum value |
|------------|---|---------------|---------------------------|
| LOGFILSZ   | Specifies the log file size as a number of 4-KB blocks                                | 1000          | 1000                      |
| LOGPRIMARY | Specifies how many primary log files are to be preallocated to the recovery log files | 3             | 10                        |
| LOGSECOND  | Specifies the number of secondary log files   | 2             | 2                         |

If the capacity of your transaction log is inadequate, the following error message is issued when you try to enable a database for spatial operations:  
GSE0010N Not enough log space is available to DB2.

### Procedure:

To increase the value of one or more configuration parameters:

1. Find the current value for the LOGFILSZ, LOGPRIMARY, and LOGSECOND parameters by reviewing the output from the GET DATABASE CONFIGURATION command or from the **Configure Database** window of the DB2 Control Center.
2. Decide whether to change one, two, or three of the values as indicated in the table above.
3. Change each value that you want to modify. You can change the values by issuing one or more of the following commands, where *db\_name* identifies your database:

```
UPDATE DATABASE CONFIGURATION FOR db_name USING LOGFILSZ 1000
```



```
UPDATE DATABASE CONFIGURATION FOR db_name USING LOGPRIMARY 10
```

```
UPDATE DATABASE CONFIGURATION FOR db_name USING LOGSECOND 2
```

If the only parameter that you change is LOGSECOND, the change takes effect immediately. In this case, proceed to “Tuning the application heap size”.

4. If you change the LOGFILSIZ or LOGPRIMARY parameter, or both:
  - a. Disconnect all applications from the database.
  - b. If the database was explicitly activated, deactivate the database.

The changes to the LOGFILSIZ or LOGPRIMARY parameters, or both, take effect the next time either the database is activated or a connection to the database is established.

### Tuning the application heap size

You use the database configuration parameter APPLHEAPSZ to specify the size of the application heap (in number of 4-KB pages). This parameter defines the number of private memory pages that are available for use by the database manager on behalf of a specific agent or subagent. The heap is allocated when an agent or subagent is initialized for an application. The allocated amount is the minimum amount that is needed to process the request to the agent or subagent. As the agent or subagent requires more heap space to process larger SQL statements, the database manager allocates memory as needed, up to the maximum that is specified on this parameter. The application heap is allocated out of the agent’s private memory.

The default value for the APPLHEAPSZ parameter is 128 (4-KB pages). When you run the ST\_enable\_db stored procedure, this value must be at least 2048.

**Recommendation:** For most DB2 Spatial Extender applications, especially those that import from or export to shape files, use an APPLHEAPSZ parameter value of at least 2048.

If the APPLHEAPSZ is set to an inadequate value, the following error message is issued when you try to enable a database for spatial operations:

```
GSE0009N Not enough space is available in DB2's application heap.
```

```
GSE0213N A bind operation failed. SQLERROR = "SQL0001N Binding or precompilation did not complete successfully. SQLSTATE=00000".
```

#### Procedure:

To change the application heap size:

1. Find the current value for the APPLHEAPSZ parameter by reviewing the output from the GET DATABASE CONFIGURATION command or from the **Configure Database** window of the DB2 Control Center.

## Setting up a database

2. Change the value to the recommended value of 2048 or to a larger value. You can change the value to 2048 by issuing the following command, where *db\_name* identifies your database:  

```
UPDATE DATABASE CONFIGURATION FOR db_name USING APPLHEAPSZ 2048
```
3. Disconnect all applications from the database.
4. If the database was explicitly activated, deactivate the database.

The change takes effect the next time either the database is activated or a connection to the database is established.

## Tuning the application control heap size

All DB2 Spatial Extender applications, especially those that import from or export to shape files, can benefit from using the recommended value for the application control heap size. You specify this characteristic with the `APP_CTL_HEAP_SZ` parameter. This parameter specifies the maximum size, in 4-KB pages, for the application control shared memory. Application control heaps are allocated from this shared memory. One application control heap is allocated for each application at the database where the application is active (or, in the case of a partitioned database system, at each database partition where the application is active). The heap is allocated during connect processing by the first agent that receives a request for the application at the database (or at the database partition). The heap is used for sharing information between agents that work on behalf of the same application. (In a partitioned database environment, the sharing occurs at the database partition level; sharing does not occur across database partitions.) The default value for the `APP_CTL_HEAP_SZ` parameter is 128.

**Recommendation:** For most DB2 Spatial Extender applications, use an `APP_CTL_HEAP_SZ` parameter value of at least 1024 (4-KB pages).

If the `APP_CTL_HEAP_SZ` is set to an inadequate value, the following error message is issued when you import data into a database from shape files:

```
GSE0214N An INSERT statement failed. SQLERROR = "SQL0973N Not enough storage is available in the "APP_CTL_HEAP" heap to process the statement.
```

### Procedure:

To change the application control heap size:

1. Find the current value for the `APP_CTL_HEAP_SZ` parameter by reviewing the output from the `GET DATABASE CONFIGURATION` command or from the **Configure Database** window of the DB2 Control Center.
2. Change the value to the recommended value of 1024 (4-KB pages) or to a larger value. You can issue the following command, where *db\_name* identifies your database:

## Setting up a database

```
UPDATE DATABASE CONFIGURATION FOR db_name USING APP_CTL_HEAP_SZ 1024
```

3. Disconnect all applications from the database.
4. If the database was explicitly activated, deactivate the database.

The change takes effect the next time either the database is activated or a connection to the database is established.

## Setting up a database

---

## Chapter 7. 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. These resources include:

- Objects provided by Spatial Extender to support spatial operations; for example, stored procedures to administrate a database, spatial data types, and spatial utilities for geocoding and importing or exporting spatial data.
- Reference data: Ranges of addresses that DB2SE\_USA\_GEOCODER uses to convert individual addresses to coordinates.
- Any geocoders that users or vendors provide.

This chapter describes these resources and introduces the tasks through which you make them available: enabling your database for spatial operations, setting up access to reference data, and registering non-default geocoders.

---

### How to set up resources in your database

The first task that you perform after setting up your database to accommodate spatial data is to render the database capable of supporting spatial operations—operations such as populating tables with spatial data and processing spatial queries. This task involves loading the database with certain resources supplied by DB2 Spatial Extender. This section describes these resources and outlines the task.

#### Inventory of resources supplied for your database

To enable a database to support spatial operations, DB2<sup>®</sup> Spatial Extender provides the database with the following resources:

- Stored procedures. When you request a spatial operation—for example, when you issue a command to import spatial data—DB2 Spatial Extender invokes 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 may require that the spatial column be registered in the catalog.
- A spatial grid index. It lets you to define grid indexes on spatial columns.

## Setting up spatial resources for a database

- Spatial functions. You use these to work with spatial data in a number of ways; for example, to determine 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

The task of having DB2 Spatial Extender supply a database with resources for creating spatial columns and manipulating spatial data is generally referred to as “enabling the database for spatial operations”.

#### Prerequisite:

Before you enable a database for spatial operations, your user ID must have either SYSADM or DBADM authority on the database.

#### Restrictions:

You can only use data types created by the enable\_db command.

#### Procedure:

You can enable a database for spatial operations in any of the following ways:

- Use the Enable Database window from the DB2 Spatial Extender menu option. The menu option is available from the database object of the DB2 Control Center.
- Issue the **db2se enable\_db** command.
- Run an application that calls the db2gse!ST\_enable\_db stored procedure.

#### Note:

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

#### Related concepts:

- “Inventory of resources supplied for your database” on page 59

#### Related tasks:

## Setting up spatial resources for a database

- “Writing applications for DB2 Spatial Extender” on page 131

### Related reference:

- “Invoking commands for setting up DB2 Spatial Extender and developing projects” on page 123
- “ST\_enable\_db” on page 191

---

## How to work with reference data

This section explains what reference data is and states what you need to do in order to access it.

### Reference data

Reference data is range of addresses that DB2SE\_USA\_GEOCODER uses to convert individual addresses into coordinates. This data consists of ranges of the most recent addresses that the United States Census Bureau has collected. When DB2SE\_USA\_GEOCODER reads an address from the database, it searches the reference data for:

- Names of certain streets within the area designated by the address’s zip code. The geocoder looks for names that match the name of the street in the address to a specified degree, or to a degree higher than the specified one; for example, 80 percent or higher.
- The address range that corresponds to the address number.

If a match is found and does not have the requested score, the geocoder returns the coordinates of the address it has read. If a match is not found or does not have the requested score, the geocoder returns a null.

An advanced configuration file called the *locator file* can be used to further influence the processing performed by the geocoder, DB2SE\_USA\_GEOCODER. The default configuration provided by DB2® Spatial Extender usually does not need to be changed in this file.

### Setting up access to reference data

The reference data for DB2SE\_USA\_GEOCODER is on one of the CD-ROMs on which Spatial Extender is shipped. This section describes how to prepare to access it.

#### Procedure:

To prepare to access the default geocoder’s reference data:

1. Decide whether to keep the reference data on the CD-ROM or to store it on your hard drive. If you keep it on the CD-ROM, then you save the space (about 700 megabytes’ worth) that it would occupy on the hard

## Setting up spatial resources for a database

drive. If you store it on hard drive, you will be able to retrieve it faster than you can retrieve it from the CD-ROM.

2. If you want to store the reference data on your hard drive:
  - a. Verify that the hard drive has enough space to contain the data (about 700 megabytes).
  - b. Copy the data to the hard drive. For instructions, see the README that accompanies the reference data.
  - c. Determine whether the copy was successful: To verify on UNIX that the data was loaded properly, look for it in the `$DB2INSTANCE/sqlib/gse/refdata/` directory. To verify on Windows NT that the data was loaded properly, look for it in the `%DB2PATH%\gse\refdata\` directory.
3. Tell DB2SE\_USA\_GEOCODER the name and location of the locator file and the base map. You do this by setting DB2SE\_USA\_GEOCODER's `base_map` and `locator_file` parameters to the appropriate values. For more information, see your database administrator or contact your IBM representative.

### Registering a geocoder

DB2SE\_USA\_GEOCODER is registered to DB2 Spatial Extender automatically when a database is enabled for spatial operations. Before other geocoders can be used, they also must be registered.

#### Prerequisite:

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

#### Procedure:

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

- Register it from the Register Geocoder window of the DB2 Control Center.
- Issue the `db2se register_gc` command.
- Run an application that calls the `db2gse.ST_register_geocoder` stored procedure.

#### Related concepts:

- “Geocoders and geocoding” on page 96



---

## **Part 3. Creating projects that use spatial data**



---

## Chapter 8. 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. This chapter:

- Discusses the nature of coordinate systems and tells how to create them
- Explains what spatial reference systems are and tells how to create them

---

### How to use coordinate systems

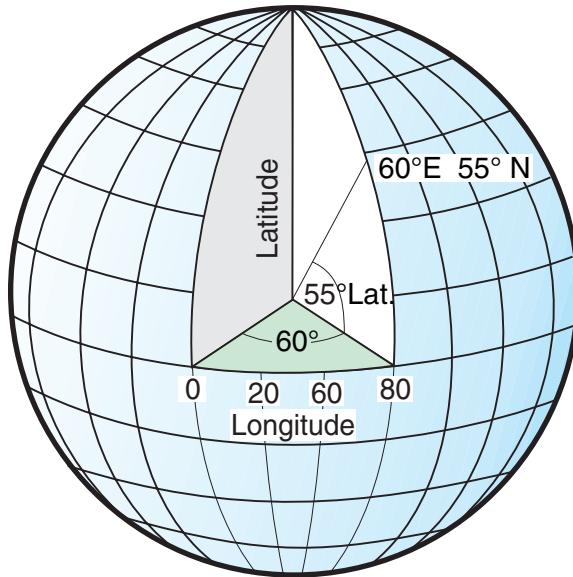
When you plan a project that uses spatial data, you need to determine whether the data should be based on one of the coordinate systems that are registered to the Spatial Extender catalog. If none of these coordinate systems meet your requirements, you can create one that does. This discussion explains 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 conceptual tool 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<sup>®</sup> Spatial Extender supports two types of coordinate systems, geographic and projected, to determine the location of a geographic feature. A *geographic coordinate system* is a grid system that uses a three-dimensional spherical surface to determine locations for a given area on the earth. Any location on earth can be referenced by a point where the grid's horizontal and vertical lines intersect. The values for the points can be measured in radians; in degrees, minutes, and seconds (DMS); or in decimal degrees.

Figure 7 on page 66 shows a geographic coordinate system.

## Setting up spatial resources for a project

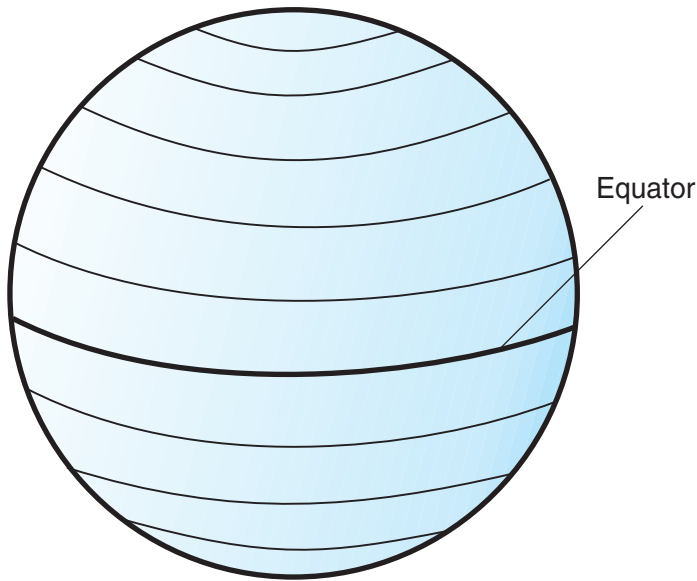


### Longitude and Latitude Values

Figure 7. A geographic coordinate system

The horizontal lines are called *latitude lines*. They are equidistant and parallel to one another, and form concentric circles around the earth. The *equator* is the largest latitude line and divides the earth in half. It is equal in distance from each of the poles, and the value of this line is zero degrees. The values north of the equator have positive measurements ranging from 0 to +90 degrees, while the values south of the equator have negative measurements ranging from 0 to -90 degrees.

Figure 8 on page 67 illustrates latitude lines.



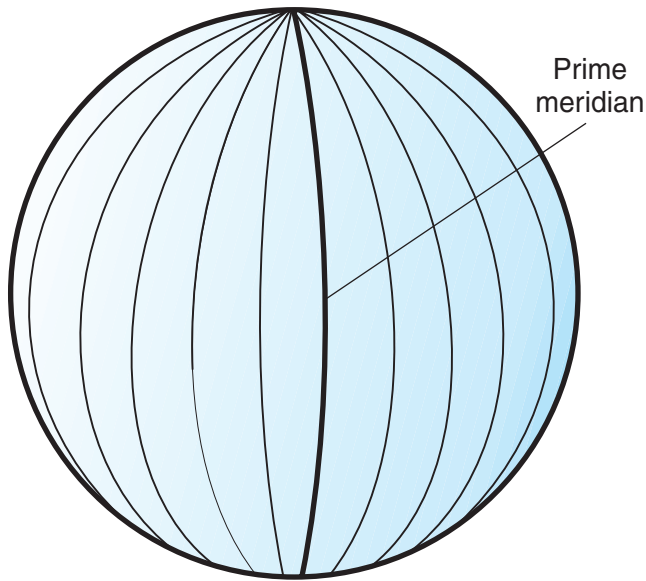
## Parallels (Lines of latitude)

Figure 8. Latitude lines

The vertical lines are called *longitude lines*, and are often referred to as *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, there are also longitude lines that pass through Bern, Bogota, and Paris that can be used as the prime meridian. The values east of the prime meridian have positive measurements ranging from 0 to +180 degrees, while the values west of the prime meridian have negative measurements ranging from 0 to -180 degrees.

Figure 9 on page 68 illustrates longitude lines.

## Setting up spatial resources for a project



### Meridians (Lines of longitude)

Figure 9. Longitude lines

The latitude and longitude lines encompass the globe to form a gridded network 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 of one degree latitude is approximately equal to one degree longitude. Because the longitude lines converge at the poles, the distance between each meridian is different at every parallel. Therefore, as you move closer to the poles, the difference between one degree latitude will be much greater than 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 km; 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 distances of the points cannot be measured accurately by using angular units of measure.

Figure 10 on page 69 shows the different dimensions between locations on the graticule.

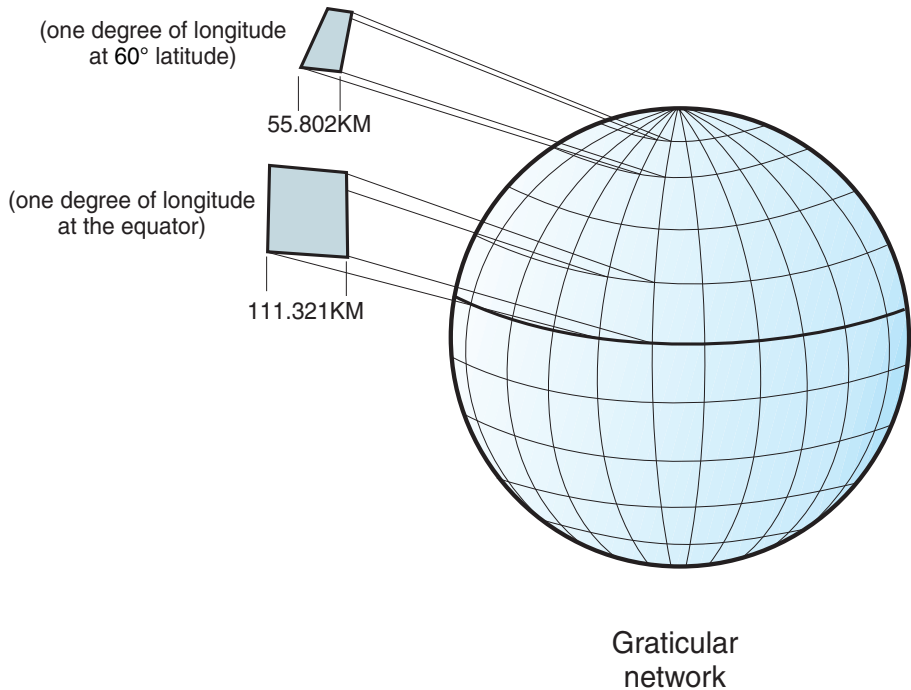
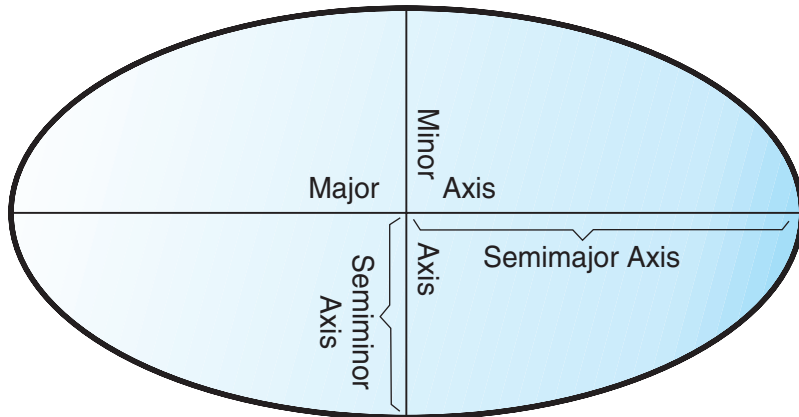
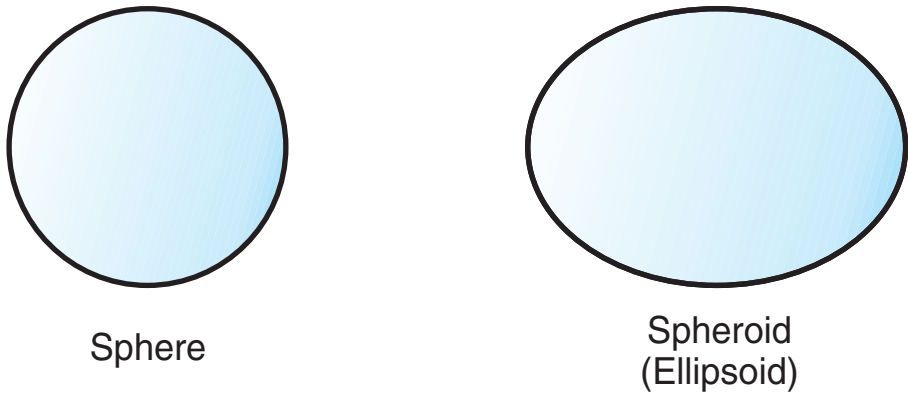


Figure 10. Different dimensions between locations on the graticule

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 large-scale maps. A *spheroid* is an ellipsoid, that is based on an ellipse, whereas a sphere is based on a circle. A sphere can be used for small-scale maps, because the differences between the two representations of the earth are virtually undetectable on a map of a small area. 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.

Figure 11 on page 70 shows the sphere and spheroid approximations of the earth.

## Setting up spatial resources for a project



### The major and minor axes of an ellipse

Figure 11. 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. Datums are especially helpful in analyzing data in a local area. A local datum aligns its spheroid to closely fit the earth's surface in a particular area. Therefore, the coordinate system's measurements will not be accurate if they are used with an area other than the one they were designed for.



## Setting up spatial resources for a project

Figure 12 shows how different datums align with the earth's surface.

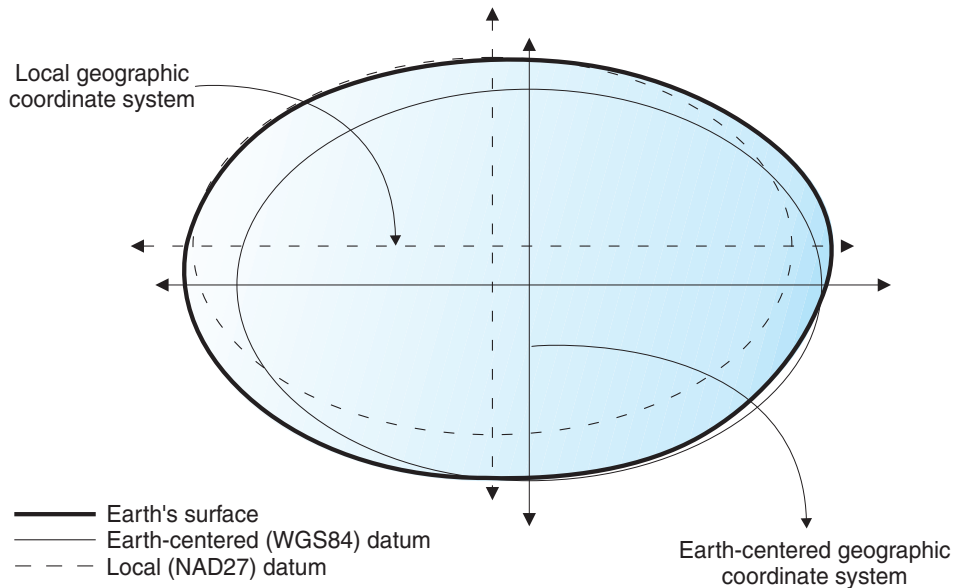


Figure 12. Datum alignments

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".

A *projected coordinate system* is a flat, two-dimensional grid 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 can be in terms of units such as feet or meters. The latitude and longitude coordinates are converted to x, y coordinates on the flat projection. The x coordinate represents the horizontal location of a point, and the y coordinate represents the vertical location of a point. The horizontal line in the center of the grid is referred to as the x axis, and the vertical line in the center of the grid is referred to as the y axis. The intersection of these two axes is the origin and has a coordinate of (0,0). The values above the x axis are positive, and the values below the x axis are negative. The horizontal lines of the grid are equidistant and are parallel to the x axis. The values to the right of the y axis are positive, and the values to the left of the y axis are negative. The vertical lines of the grid are equidistant and parallel to the y axis.

## Setting up spatial resources for a project

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 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 *Mecator* 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.

### Selecting or creating coordinate systems

After you enable a database for spatial operations, you are ready to plan projects that use spatial data. A first step in planning a project is to determine what coordinate system to use. Your options are as follows:

- You can use a coordinate system that was shipped with DB2 Spatial Extender or one that was created by a user. Over 2000 coordinate systems are shipped with DB2 Spatial Extender. Among them are:
  - 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.
  - 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 “Maps and Data” CDs that are shipped with DB2 Spatial Extender.
    - When you plan to use the geocoder shipped with DB2 Spatial Extender to geocode addresses within the United States

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.

- You can create a coordinate system.

#### Prerequisites:

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

#### Procedure:

You can create a coordinate system in any of the following ways:

- Create it from the Create Coordinate System window of the DB2 Control Center.
- Issue the **db2se create\_cs** command from the db2se command line processor.

## Setting up spatial resources for a project

- Run an application that invokes the `db2se.ST_create_coordsys` stored procedure.

### Related concepts:

- “Coordinate systems” on page 65

### Related tasks:

- “Writing applications for DB2 Spatial Extender” on page 131

### Related reference:

- “Invoking commands for setting up DB2 Spatial Extender and developing projects” on page 123
- “`ST_create_coordsys`” on page 172

---

## 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.

### About spatial reference systems

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

- Coordinates that define the maximum possible extent of space that is referenced by a given range of coordinates.
- The name of the coordinate system from which the coordinates are derived.
- 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 first and third subsets of parameter values cited in the preceding list: those that define a maximum extent of space and those that are used as conversion factors.

### 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

## Setting up spatial resources for a project

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 planes 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 planes 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 parameter values in a spatial reference system are used to effect the conversions described in the preceding paragraph. One type of value, called an *offset*, is subtracted from each negative coordinate, leaving a positive value as a remainder. Each decimal coordinate is multiplied by another type of value, called a *scale factor*, resulting 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 non-decimal 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.)

The aforementioned 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.

## Setting up spatial resources for a project

### Selecting or creating spatial reference systems

After you determine what coordinate system to use, you are ready to provide a spatial reference system for your data. Your options are as follows:

- You can use a spatial reference system that was shipped with DB2 Spatial Extender or one that was created by a user. Some of the spatial reference systems shipped with DB2 Spatial Extender are:
  - A default spatial reference system. Its name is “DEFAULT\_SRS.” Use this system when you use the Unspecified coordinate system.
  - A spatial reference system that you can use when you use the coordinate system that DB2 Spatial Extender provides for addresses in the United States (GCS\_NORTH\_AMERICAN\_1983). This spatial reference system’s name is “NAD83\_SRS\_1” and its numeric identifier is 1. It is advisable to use NAD83\_SRS\_1 when you want to use the geocoder or the United States sample data that are shipped with DB2 Spatial Extender.

To find out more about these spatial reference systems, and to determine if spatial reference systems are available to you, consult the DB2SE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view.

- You can create a spatial reference system. There are two ways to do this:
  - Determine and specify the extent of the geographical area that your spatial data defines, as well as the degree of accuracy that you want the data to have. If you are new to spatial reference systems, you might find this method to be the faster of the two.
  - Determine and specify all the factors that DB2 Spatial Extender requires to convert negative values and decimal values to positive values and integers. You might prefer this method if you are accustomed to creating spatial reference systems. However, this section offers a “fast-track” approach that might appeal to all users. The following section describes this approach. The sections that follow the next one describe the longer methods.

**Tip:** Use the longer methods when the largest coordinate multiplied by the scale factor is greater than the largest positive integer or when you need to follow strict criteria for accuracy and precision.

#### Prerequisites:

There are no prerequisites for using an existing spatial reference system or for creating one.

#### Restrictions:

## Setting up spatial resources for a project

When you create a spatial reference system, select an offset and scale factor that, when applied against spatial data, result in integers that are less than twice the value of the largest possible integer in your original spatial data.

### **Fast-track procedure for creating a spatial reference system:**

If you are working with decimal-degrees, the following values will support the full range of latitude-longitude coordinates and preserve 6 decimal positions, equivalent to approximately 100 cm. Note that all the data on the DB2 Spatial Extender sample data CDs is in decimal-degrees.

```
xOffset = -180  
yOffset = -90  
xyScale = 1000000  
zOffset = 0  
zScale = 1  
mOffset = 0
```

In addition to the values just listed, you need to specify the coordinate system for the data that you are working with.

If you are working with positive coordinates, all offset values can be set to 0. If the coordinates are integers, the scale factors can be set to 1. If the coordinates are decimal values, the scale factor should be set 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.

### **Procedure for creating a spatial reference system by determining and specifying the extent of geographical area:**

A description of this procedure follows. The description begins with an overview of the steps in the procedure and concludes with details of each step.

#### *Overview of steps:*

1. Determine the highest coordinate and the lowest coordinate among the coordinates of the locations that you want to represent. If your data is to include measures, determine the highest and lowest of these measures.
2. Optional but recommended: Indicate to DB2 Spatial Extender that the extent that encompasses the locations that you are concerned with is larger than it actually is. Consequently, after you populate a spatial column with the coordinates for these locations, you can update the column with coordinates for locations of new features as they are added to the outer reaches of the extent. Otherwise, if you want to update the column with

## Setting up spatial resources for a project

coordinates that lie beyond the extent defined by the spatial reference system, you need to supply a spatial reference system that defines a larger extent. Replacing one spatial reference system with another can be a complex and time-consuming task.

3. If any of the coordinates that define the locations that you are representing are decimal numbers, or if any measures associated with these coordinates are decimal numbers, determine what *scale factors* to use. These are values that, when multiplied by decimal coordinates and measures, yield integers whose precision is at least the same as that of the coordinates and measures.
4. Create the spatial reference system that you want.

### *Details of steps:*

1. To determine the highest and lowest coordinates of the locations that you are interested in, and to determine the highest and lowest measures to store with your spatial data, answer the following questions
  - Is the easternmost location in your domain defined by a single pair or by multiple pairs of X and Y coordinates? If it is defined by a single pair, what is the X coordinate? If it is defined by multiple pairs, what is the easternmost X coordinate in these pairs? (The answer to both questions is the highest X coordinate.) For example, if you are representing oil wells, and each one is defined by a pair of X and Y coordinates, what X coordinate indicates the location of the oil well that is furthest east?
  - Is the westernmost location in your domain defined by a single pair or by multiple pairs of X and Y coordinates? If it is defined by a single pair, what is the X coordinate? If it is defined by multiple pairs, what is the westernmost X coordinate in these pairs? (The answer to both questions is the lowest X coordinate. If the location lies to the west of the point of origin, this coordinate will be a negative value.)
  - Is the northernmost location in your domain defined by a single pair or by multiple pairs of X and Y coordinates? If it is defined by a single pair, what is the Y coordinate? If it is defined by multiple pairs, what is the northernmost Y coordinate in these pairs? (The answer to both questions is the highest Y coordinate.)
  - Is the southernmost location in your domain defined by a single pair or by multiple pairs of X and Y coordinates? If it is defined by a single pair, what is the Y coordinate? If it is defined by multiple pairs, what is the southernmost Y coordinate in these pairs? (The answer to both questions is the lowest Y coordinate. If the location lies to the south of the point of origin, this coordinate will be a negative value.)
  - Are the heights and depths of the locations defined by Z coordinates? If so, what Z coordinate defines the greatest of the heights? What Z coordinate defines the greatest of the depths?



## Setting up spatial resources for a project

- If you are going to include measures in your spatial data, which measure has the highest numerical value? Which has the lowest?
2. If you want to indicate to DB2 Spatial Extender that the domain that encompasses the locations that you are concerned with is larger than it actually is, proceed as follows:

To 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 in this discussion 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 define 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 data's extent.

3. If any of the coordinates that define the locations that you are representing are decimal numbers, or if any measures associated with these coordinates are decimal numbers, determine what scale factors to use.

Specify a scale factor that, when multiplied by a decimal X coordinate or a decimal Y coordinate, yields a 32-bit integer. It is advisable to make this scale factor a factor of 10: 10 to the first power (10), 10 to the second power (100), 10 to the third power (1000), or, if necessary, a larger factor. To decide what factor of 10 the scale factor should be:

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

To avoid this problem, select a factor of 10 that, when multiplied by the original coordinate, yields a decimal number that DB2 Spatial Extender can truncate to a storable integer, with minimum loss of

## Setting up spatial resources for a project

precision. In this case, you could select 10 to the fourth power (10000). Multiplying 10000 by 10006.789876 yields 100067898.76. DB2 Spatial Extender would truncate this number to 100067898, reducing its accuracy by a virtually insignificant amount.

4. Create the spatial reference system that you want. As you do so, you will specify augmented values and scale factors that you determined in the preceding steps.

You can create a spatial reference system in any of the following ways:

- Create it from the – window of the Control Center.
- Issue the **db2se create\_srs** command from the db2se command line processor.
- Run an application that invokes the db2se.ST\_create\_srs stored procedure.

For information about how to proceed, consult the sources that are listed under “Related tasks” at the end of this discussion.

### Full procedure for creating a spatial reference system by determining and specifying all conversion factors:

1. Determine the lowest negative 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. For guidelines on performing this step, see the methods for determining lowest coordinates and measures in step 1 on page 78
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. Consequently, after you write data about these locations to a spatial column, you can augment the column with data about locations of new features as they are added to outer reaches of the domain, without your having to replace your spatial reference system with another one.

To perform this step, proceed as follows: To 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, as noted in 77, is referred to in this discussion 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 define 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. Determine what *offset factors* (or *offsets*, for short) to use. These are values that, when subtracted from negative coordinates and measures, leave positive numbers.

## Setting up spatial resources for a project

To perform this step, proceed as follows:

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

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

- Similarly, specify offsets that leave 0 when subtracted from the augmented Y value, the augmented Z value, and the augmented measure.
4. If any of the coordinates for the locations that you are representing are decimal numbers, determine what scale factors to use. For guidelines on performing this step, see step 3 on page 79.
  5. Define the spatial reference system that you want to DB2 Spatial Extender. As you do so, you will specify the augmented values, offsets, and scale factors that you arrived at in preceding steps.

You can create a spatial reference system in any of the following ways:

- Create it from the Create Spatial Reference System window of the DB2 Control Center.
- Issue the **db2se create\_srs** command from the db2se command line processor.
- Run an application that invokes the db2se.ST\_create\_srs stored procedure.

For information about how to proceed, consult the sources that are listed under “Related tasks” at the end of this discussion.

### Note about selecting scale factors that preserve precision:

When you invoke a spatial function that takes as input a decimal coordinate or measure and the identifier of a spatial reference system, the function multiplies the decimal coordinate or measure by a scale factor within the system. The result is an integer that DB2 Spatial Extender stores. The scale factor needs to be large enough to ensure that the precision of this integer is the same as the precision of the decimal coordinate.

For example, suppose that the ST\_Point function is given input that consists of an X coordinate 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

## Setting up spatial resources for a project

Y coordinates. If this scale factor is 10, the resulting integers that DB2 Spatial Extender stores will be 100 and 200, respectively. Because the precision of these integers (3) is less than the precision of the coordinates (4), DB2 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.

### **Related concepts:**

- “About spatial reference systems” on page 74

### **Related tasks:**

- “Writing applications for DB2 Spatial Extender” on page 131

### **Related reference:**

- “Invoking commands for setting up DB2 Spatial Extender and developing projects” on page 123
- “ST\_create\_srs” on page 174

---

## Chapter 9. Setting up spatial columns

In preparing to obtain spatial data for a project, you not only choose or create a coordinate system and spatial reference system; you also provide one or more table columns to contain the data. This chapter:

- Notes that results of queries of the columns can be rendered graphically, and provides guidelines for choosing data types for the columns
- Describes the task of providing the columns
- Describes the task of making the columns accessible to tools that can display their content in graphical form

---

### Spatial columns

#### Spatial columns with viewable content

When you use a visualization tool, such as ArcExplorer, to query a spatial column, the tool will return 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 to register the column with a spatial reference system.

#### Spatial data types

When you enable a database for spatial operations, DB2<sup>®</sup> Spatial Extender supplies the database with a hierarchy of structured data types. Figure 13 on page 84 presents this hierarchy. In this figure, the instantiable types have a white background; the uninstantiable types have a shaded background.

## Setting up spatial columns

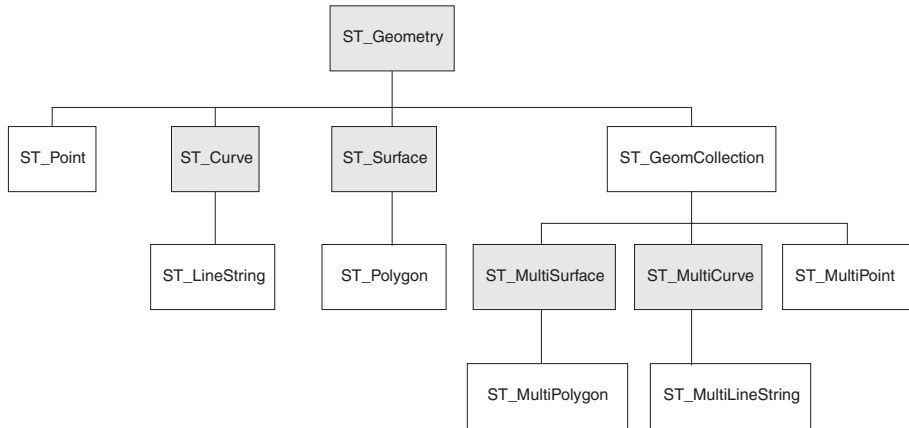


Figure 13. Hierarchy of spatial data types. Data types named in white boxes are instantiable. Data types named in shaded boxes are not instantiable.

The hierarchy in Figure 13 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 values--an X coordinate and a Y coordinate--that define such an intersection. The X coordinate indicates where the intersection lies on the east-west line; the Y coordinate indicates where the intersection lies on the north-south line.
- Use `ST_LineString` for coordinates that define the space that is occupied by linear features; for example, streets, canals, and pipelines.
- Use `ST_Polygon` when you want to indicate the extent of space covered by a multi-sided feature; for example, a 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. On the other hand, 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 kind of data items that columns of the other data types can contain.

### Attention:

If you plan to use the supplied geocoder, `DB2SE_USA_GEOCODER`, to produce data for a spatial column, the column must be of type `ST_Point` or `ST_Geometry`. Certain visualization tools, however, do not

## Setting up spatial columns

support ST\_Geometry columns, but only columns to which a proper subtype of ST\_Geometry has been assigned.

### Related tasks:

- “Setting up spatial columns for access by visualization tools” on page 86
- “Creating spatial columns” on page 86

---

## Creating spatial columns

There are several ways to provide your database with spatial columns:

- Use DB2’s CREATE TABLE statement to create a table and to include a spatial column within that table.
- Use DB2’s ALTER TABLE 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.
- If you are importing spatial data from an SDE transfer file, use DB2 Spatial Extender to create a table, to provide this table with a column to hold the data, and to make the column accessible to visualization tools.

For information about how to perform these operations, consult the sources that are listed under “Related tasks” at the end of this discussion.

### Related tasks:

- “Writing applications for DB2 Spatial Extender” on page 131
- “Setting up spatial columns for access by visualization tools” on page 86

### Related reference:

- “Invoking commands for setting up DB2 Spatial Extender and developing projects” on page 123

---

## Setting up spatial columns for access by visualization tools

If you want certain visualization tools—for example, ArcExplorer—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.

### Prerequisites:



Before you register a spatial column, your user ID must hold one of the following forms of authorization:

- SYSADM or DBADM authority on the database in which the table that contains the column resides
- The CONTROL or ALTER privilege on this table.

If you are using the db2se command line processor or an application program to import data from an SDE transfer file, you can have DB2 Spatial Extender automatically create and register a column to hold the data. In that case, your user ID must hold SYSADM or DBADM authority on the database.

### Procedure:

You can register a spatial column in any of the following ways:

- Use the Spatial Columns and Select Spatial Reference System windows of the DB2 Control Center to register the column.
- Issue the **db2se register\_spatial\_column** command.
- Run an application that invokes the db2gse.ST\_register\_spatial\_column stored procedure.
- If you want to import spatial data from an SDE transfer file, you can use the Import Spatial Data window of the Control Center, the **import\_sde** command, or the db2gse.ST\_import\_sde stored procedure to create a table with a spatial column, to register this column, and to import the data into the column.

### Related tasks:

- “Writing applications for DB2 Spatial Extender” on page 131
- “Creating spatial columns” on page 86

### Related reference:

- “Invoking commands for setting up DB2 Spatial Extender and developing projects” on page 123
- “ST\_register\_spatial\_column” on page 212

## Setting up spatial columns

---

## Chapter 10. Populating spatial columns

After you create spatial columns, and register the ones to be accessed by these visualization tools, you are ready to populate the columns with spatial data. There are three ways to supply the data: import it; use a geocoder to derive it from business data; or use spatial functions to create it or to derive it from business data or other spatial data. This chapter:

- Discusses the concept and task of importing spatial data to your database, and the concept and task of exporting spatial data to files that applications can use.
- Discusses geocoding and introduces the tasks of setting up geocoding operations, setting up geocoders to run automatically, and running geocoders in batch mode.

---

### How to import and export spatial data

This section discusses the concept of importing and exporting data, and introduces the following tasks:

- Importing spatial data to a new table, or to an existing table or view
- Exporting spatial data to files that applications can use

### About importing and exporting spatial data

You can use DB2<sup>®</sup> 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

## Populating spatial columns

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.

### Shapefiles and SDE transfer files:

DB2 Spatial Extender supports two types of data exchange files: shapefiles and SDE transfer files. The term *shapefile* 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*.

Shapefiles 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 shapefiles as well.

*SDE transfer files* are often used for importing data that originates in ESRI databases. Each file includes spatial data, a spatial reference system for this data, and business data. The spatial data, whose format is proprietary to ESRI, is intended for a table column that has been registered to the DB2 Spatial Extender catalog. The business data is targeted for other columns in the table to which the registered column belongs.

## Importing spatial data

This section provides an overview of the tasks of importing shape data and SDE transfer data to your database. The section includes cross-references to specifics that you need to know (for example, processes and parameters) in order to perform these tasks.

### 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. More specifically, you can:

- Import the shape data to a spatial column in an existing table, an existing updatable 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

### Prerequisites:

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

- SYSADM or DBADM authority on the database that contains the table or view
- CONTROL privilege on the table or view
- The INSERT privilege on the table or view
- The SELECT privilege on the table or view (required only if the table includes an idColumn that is not an IDENTITY column)
- 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 forms of authorization:

- SYSADM, DBADM, or CREATETAB authority on the database that contains the table
- One of the following permissions:
  - CREATEIN privilege on the schema to which the table belongs (required when the schema already exists)
  - IMPLICIT\_SCHEMA authority on the database that contains the table (required when the schema specified for the table does not actually exist)
- Privileges to access the directories to which input files and error files belong

## Populating spatial columns

- Read privileges on the input files and write privileges on the error files

### Procedure:

You can import shape data in any of the following ways:

- Use the Import Shape Data window of the DB2 Control Center.
- Issue the **db2se import\_shape** command.
- Run an application that calls the db2gse.ST\_import\_shape stored procedure.

### Recommendation:

You can enhance the performance of import processing by exploiting features available in DB2. For example, when you import data to an existing table or to a table that you create, define the table as NOT LOGGED INITIALLY by specifying the appropriate table creation parameters.

### Related concepts:

- “About importing and exporting spatial data” on page 89

### Related tasks:

- “Importing SDE transfer data to a new or existing table” on page 92
- “Writing applications for DB2 Spatial Extender” on page 131

### Related reference:

- “Invoking commands for setting up DB2 Spatial Extender and developing projects” on page 123
- “ST\_import\_shape” on page 198

## Importing SDE transfer data to a new or existing table

You can import SDE transfer data to an existing table, or you can create a table and import SDE transfer data to it in a single operation. More specifically, you can:

- Import SDE transfer data to an existing table that includes a spatial column that is already registered to the DB2 Spatial Extender catalog. The transfer data can include spatial data for the column and business data for other columns in the table.
- Automatically create a table that has a spatial column, register this column to the catalog, and import SDE transfer data to this column as well as to the table’s other columns.

### Prerequisites:

Before you import data to a column in an existing table or view, your user ID must hold one of the following forms of authorization:

- SYSADM or DBADM authority on the database that contains the table or view
- CONTROL privilege on the table or view
- Both the INSERT and SELECT privileges on the table or view

Before you initiate the operation to create a table automatically and import shape data to it, your user ID must hold the following forms of authorization:

- Either SYSADM, DBADM, or CREATETAB authority on the database that contains the table
- One of the following permissions:
  - CREATEIN privilege on the schema to which the table belongs (required when the schema already exists)
  - IMPLICIT\_SCHEMA authority on the database that contains the table (required when the schema specified for the table does not actually exist)

### **Procedure:**

You can import SDE transfer data in any of the following ways:

- Use the Import window of the DB2 Control Center.
- Issue the **db2se import\_sde** command.
- Run an application that calls the db2gse.GSE\_import\_sde stored procedure.

For information about how to perform these actions, consult the sources that are listed under “Related tasks” at the end of this discussion.

### **Related concepts:**

- “About importing and exporting spatial data” on page 89

### **Related tasks:**

- “Importing shape data to a new or existing table” on page 91
- “Writing applications for DB2 Spatial Extender” on page 131

### **Related reference:**

- “Invoking commands for setting up DB2 Spatial Extender and developing projects” on page 123
- “GSE\_import\_sde” on page 162

## Populating spatial columns

### Exporting spatial data

This section provides an overview of the tasks of exporting spatial data to shape and SDE transfer files. The section includes cross-references to specifics that you need to know (for example, processes and parameters) in order to perform these tasks.

#### Exporting data to a shapefile

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

#### Prerequisites:

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.

#### Procedure:

You can export data to a shapefile in any of the following ways:

- Initiate the export from the Export Shape File window of the DB2 Control Center.
- Issue the **db2se export\_shape** command from the db2se command line processor.
- Run an application that calls the db2gse.ST\_export\_shape stored procedure.

For information about how to perform these actions, consult the sources that are listed under "Related tasks" at the end of this discussion.



### Exporting data to an SDE transfer file

You can export a table that contains spatial data to an SDE transfer file. The table cannot contain more than one spatial column. Moreover, this column must be registered to the DB2 Spatial Extender catalog. If the table contains business data, this data will be exported along with the spatial data. You can export either all rows in the table or a subset of rows. To export a subset, specify a WHERE clause that identifies the subset.

#### Prerequisites:

Before you can export data to an SDE transfer file, your user ID must hold the following permissions:

- Either SYSADM OR DBADM authority.
- The SELECT privilege on the table that is to be exported.
- The privilege to write to the directory where the file to which you will be exporting data resides

#### Restrictions:

- You can export only one spatial column in each export operation.
- The columns that you export must have data types that the SDE format supports.
- The table must contain exactly one spatial column.
- This column must be registered to the DB2 Spatial Extender catalog.
- You cannot append to existing SDE files.

#### Procedure:

You can export spatial and business data to an SDE transfer file in any of the following ways:

- Use the Export SDE Files window of the DB2 Control Center.
- Issue the **db2se export\_sde** command.
- Run an application that calls the db2gse.GSE\_export\_sde stored procedure.

#### Related concepts:

- “About importing and exporting spatial data” on page 89

#### Related tasks:

- “Writing applications for DB2 Spatial Extender” on page 131

#### Related reference:

- “Invoking commands for setting up DB2 Spatial Extender and developing projects” on page 123

## Populating spatial columns

- “GSE\_export\_sde” on page 160

---

### How to use a geocoder

This section discusses the concept of geocoding and introduces the following tasks:

- Defining the work that you want a geocoder to do; for example, specifying how many records the geocoder should process before a commit is issued
- Setting up a geocoder to geocode data as soon as the data is added to, or updated in, a table.
- Running a 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<sup>®</sup> 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. For example, the geocoder that is shipped with DB2 Spatial Extender, DB2SE\_USA\_GEOCODER, translates United States addresses into ST\_Point data. DB2 Spatial Extender can support vendor-supplied and user-supplied geocoders as well; and their input and output need not be like that of DB2SE\_USA\_GEOCODER. To illustrate: 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. For example, in batch mode, DB2SE\_USA\_GEOCODER attempts to translate all the addresses in 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. For example, DB2SE\_USA\_GEOCODER searches 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 geocoding process that DB2SE\_USA\_GEOCODER participates in is as follows:

1. DB2SE\_USA\_GEOCODER performs operations that it has been designed to carry out:
  - a. DB2SE\_USA\_GEOCODER parses each address that it receives as input.
  - b. DB2SE\_USA\_GEOCODER searches the reference data for street names that, to a certain degree, resemble the street name in the parsed address. It confines its search to streets within the area designated by the address's zip code.
  - c. If the search is successful, DB2SE\_USA\_GEOCODER determines whether any address on the streets it has found match the parsed address to a certain degree.
  - d. If DB2SE\_USA\_GEOCODER finds a match, it geocodes the parsed address. Otherwise, it returns a null.
2. If DB2SE\_USA\_GEOCODER geocodes the parsed address, DB2 puts the resulting coordinates in a designated spatial column.
3. If DB2SE\_USA\_GEOCODER is geocoding in batch mode, DB2 Spatial Extender issues a commit either (a) every time DB2SE\_USA\_GEOCODER finishes processing a certain number of input records or (b) after DB2SE\_USA\_GEOCODER finishes processing all of its input.

### The user's tasks:

## Populating spatial columns

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 needs to be done when a geocoder is invoked. For example, you can specify:

- 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, DB2SE\_USA\_GEOCODER can 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 described in the foregoing 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.

### Prerequisites:

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

- SYSADM or DBADM authority on the database that contains the tables that the geocoder will operate on
- The SELECT privilege and the CONTROL or UPDATE privilege on each table that the geocoder operates on

### Procedure:

You can set up geocoding operations in any of the following ways:

- Invoke it from the Set Up Geocoding window of the DB2 Control Center.
- Issue the **db2se setup\_gc** command.
- Run an application that calls the db2gse.ST\_setup\_geocoding stored procedure.

For information about how to perform these actions, consult the sources that are listed under “Related tasks” at the end of this discussion.

### Recommendations:

- When DB2SE\_USA\_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

## Populating spatial columns

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 scenario:
    - 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.
- Other scenarios:
- 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.

### Prerequisites:

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 forms of authorization:
  - SYSADM or DBADM authority on the database that contains the table on which triggers to invoke the geocoder will be defined
  - One or more privileges on this table:
    - The CONTROL privilege.
    - If you do not have the CONTROL privilege, you need the ALTER, SELECT, and UPDATE privileges.
  - The privileges required to create triggers on this table.

### Procedure:

There are three ways to set up automatic geocoding:

- Do so from either the Set Up Geocoding window or the Geocoding window of the DB2 Control Center.
- Issue the **db2se enable\_autogc** command.
- Run an application that calls the `db2gse.ST_enable_autogeocoding` stored procedure.

For information about how to perform these actions, consult the sources that are listed under “Related tasks” at the end of this discussion.

### Recommendations:

- 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

## Populating spatial columns

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.

### Running a geocoder in batch mode

You can invoke a geocoder to run in batch mode; that is, to attempt, in a single operation, to translate multiple records into spatial data that is to go into a specific column.

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.

#### Prerequisites:

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

- SYSADM or DBADM authority on the database that contains the table whose data is to be geocoded
- The CONTROL or 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.



### Restrictions:

### Procedure:

You can invoke a geocoder to run in batch mode in any of the following ways:

- Invoke it from the Run Geocoding window of the DB2 Control Center.
- Issue the **db2se run\_gc** command.
- Run an application that calls the `db2gse.ST_run_geocoding` stored procedure.

## Populating spatial columns

---

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

Before you query spatial columns, you can create indexes and views that will facilitate access to them. This chapter:

- Describes the nature of the indexes that Spatial Extender uses to expedite access to spatial data
- Explains how to create such indexes
- Explains how to use views to access spatial data

---

### Spatial indexes

DB2<sup>®</sup> Spatial Extender's indexing technology utilizes *grid indexing*, which is designed to index multi-dimensional spatial data, to index spatial columns. DB2 Spatial Extender provides a grid index that is optimized for two dimensional data, however higher dimensional geometries can be indexed as well. The index is created on the X and Y dimensions.

A spatial grid index is generated using the minimum bounding rectangle (MBR) of a geometry. For most geometries, the MBR is a rectangle that surrounds the geometries. Wherever possible, its four sides coincide with the geometry's boundary. The MBR for a horizontal or vertical linestring is a degenerated rectangle that is collapsed into the linestring itself. The MBR of a point is a degenerated rectangle that is collapsed into the point (the original point itself). An MBR is itself a geometry; it is represented by the minimum and maximum X and Y coordinates of the geometry that it surrounds or collapses to.

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 intersection is recorded as the internal identifier of the row that contains the geometry and the minimum X and Y coordinates of the grid cell intersected. DB2 Spatial Extender supports 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. 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 indexes and views

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.

Each grid level should be at least three times larger than the previous level. If a feature intersects four or more grid cells, the feature is promoted to the next larger level. In general, the larger features will be indexed at the larger levels.

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 10.0e0, 100.0e0, and 1000.0e0 will first intersect each geometry with the level 10.0e0 grid. If a geometry intersects with more than four grid cells of size 10.0e0, it is promoted and intersected with the level 100.0e0 grid. If more than four intersections result at the 100.0e0 level, the geometry is promoted to the 1000.0e0 level. At the 1000.0e0 level, the intersections must be entered into the spatial index because this is the highest possible level.

For more information about spatial indexes, see the Spatial indexes DB2 topic.

---

## How to create spatial grid indexes

Good query performance is related to having efficient indexes defined on the columns of the base tables in a database. The performance of the query is directly related to how quickly values in the column can be found during the query.

Spatial queries are typically queries that involve two or more dimensions. For example, in a spatial query you might want to know if a point is included within an area (polygon). Due to the multidimensional nature of spatial queries, the DB2® native B-tree indexing is inefficient for these queries. For this reason, DB2 Spatial Extender provides an index based on a grid called a spatial grid index. If you create a spatial grid index on the spatial column, the index entries can be scanned much more quickly than can all the values in the column itself. Then the performance of the query is improved.

When you create a spatial grid index, you give it: a name, the name of the spatial column on which it is to be defined, and the grid sizes. For detailed information on the relationship between spatial grid indexes and grid sizes, please see Spatial indexes. The grid sizes define the resolution with which the space is partitioned. The combination of the three grid sizes

works to minimize the size of the grid cells while also minimizing the size of the spatial grid index, by minimizing the total number of index entries.

Knowledge of the relative geometry size distribution of the data in the spatial column is important information to use when determining the optimum grid size and number of grid levels to use. DB2 Spatial Extender provides a utility, called the Index Advisor that will analyze the spatial column data and suggest appropriate grid sizes.

The following sections in this topic describe:

- The steps for creating a spatial grid index
- General guidelines for how to get the most out of your spatial grid indexes

The Index Advisor is described in Using the index advisor.

### Creating spatial grid indexes

There are several ways to create a spatial grid index:

- Using the Control Center.
- Using the SQL CREATE INDEX statement with the `db2gse.spatial_index` index extension in the EXTEND USING clause.
- Using a GIS tool that works with DB2 Spatial Extender. If you use such a tool to create the index, the tool will issue the same SQL CREATE INDEX statement as described before.

This section presents the steps for the first two methods. For information on using a GIS tool to create a spatial grid index, see the documentation that comes with that tool.

#### Prerequisites:

Before you create a spatial grid index, your user ID must hold the authorizations that are needed for the DB2 SQL CREATE INDEX statement. The user ID must include at least one of the following:

- SYSADM or DBADM authority on the database where the table that has the column resides
- One of the following items:
  - CONTROL privilege on the table
  - INDEX privilege on the table

And one of the following items:

- IMPLICIT\_SCHEMA authority on the database, if the implicit or explicit schema of the index does not exist
- CREATEIN privilege on the schema, if the schema name of the index refers to an existing schema

## Using indexes and views

For a complete description of the DB2 SQL CREATE INDEX statement, see CREATE INDEX statement.

Before you create a spatial grid index on a column, it is recommended that you 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. For more information on using the Index Advisor, see the **Related tasks** section at the end of this topic.

Before you create a spatial grid index, you must know the values you want to specify for the fully qualified spatial grid index name and the three grid sizes that the index will use.

### Restrictions:

The same restrictions for creating indexes using the CREATE INDEX statement are in effect when you create a spatial grid index: the column on which you create an index must be a base table column, not a view column or a nickname column. DB2 will resolve aliases in the process.

### Procedure:

To do this task using the Control Center:

1. In the Control Center, right-click the table that has the spatial column that you want a spatial grid index on and select **Spatial Extender**→**Spatial Indexes** from the pop-up menu. The Spatial Indexes window opens.
2. Follow the instructions in the online help for the Spatial Indexes window. You can display those instructions by clicking on the **Help** button in the Spatial Indexes window.

To do this task using the SQL CREATE INDEX statement:

Issue the CREATE INDEX command using the EXTEND USING clause and the db2gse.spatial\_index grid index extension. The command is:

```
▶▶ CREATE INDEX index_schema.index_name ON table_schema.table_name (column_name) EXTEND USING  
▶▶ db2gse.spatial_index (finest_grid_size, middle_grid_size  
▶▶ , coarsest_grid_size)
```

Where:

*index\_schema.*

Name of the schema to which the index that you are creating is to belong. If you do not specify a name, DB2 uses the schema name that is stored in the CURRENT SCHEMA special register.

*index\_name*

Unqualified name of the grid index that you are creating.

*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 equal 0 (zero).
- *coarsest\_grid\_size* must either be larger than *middle\_grid\_size* or equal 0 (zero).

For example, for table BRANCHES that has a spatial column LOCATION, the spatial grid index LOCIDX is created on that column:

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

Whether you create the spatial grid index using the Control Center or 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 on their values, an error condition is raised at the times as described in these situations:

- If there are no non-null geometries in the spatial column, the spatial grid index creation process will succeed. The validity of the grid sizes is verified when a non-null geometry is inserted or updated in that spatial column. If the specified grid sizes are not valid, an error condition is raised when the non-null geometry is inserted or updated.

## Using indexes and views

- If there are non-null geometries in the spatial column, the validity of the grid sizes is verified during the spatial grid index creation process. If the specified grid sizes are not valid, the process will fail during the indexing of the existing geometries. The error condition is raised immediately, and the spatial grid index is not created.

### General guidelines for getting the most out of spatial grid indexes

It is recommended that you use the Index Advisor for determining appropriate grid sizes for your spatial grid indexes. Because you can use the Index Advisor to:

- Analyze an existing spatial grid index
- Analyze the geometries in a spatial column for which you want to create a spatial grid index

It is the best way for you to tune your spatial grid indexes and make your spatial queries most efficient.

Besides following the recommendations from the Index Advisor, you can use the tips presented in these sections for fine tuning your spatial grid indexes. 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 grid size tends to result in a large number of entries in the index. Try to strike a balance between the two.

### Tips on selecting the number of grid levels

Here are some tips on selecting the number of grid levels for a spatial grid index:

- If the values in the spatial column are about the same relative size, use a single grid level. You can have up to three grid levels. However, for each grid level in a spatial grid index, a separate index search will be performed during a spatial query. Therefore, if you have more grid levels, your query will be less efficient.

Typically, a spatial column will not contain geometries of the same relative size. However, usually geometries in a spatial column can be grouped according to size. You would 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.

- When you use more than one grid level, make the size of each grid level at least three times larger than the size of the preceding grid level. For example, if your first (finest) grid level has a size of 10.0, make the next one at least 30.0.



### Tips on selecting grid cell sizes

Here are some tips on selecting the grid cell sizes for each grid level of your spatial grid index:

- Make the grid cell size for each grid level at least three times the size of the cells of the preceding grid level.
- If you are planning to do an initial load of data into the column, you should create the spatial grid index after completing 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 have to be maintained during the load process.

### Related tasks:

- “Using the Index Advisor” on page 111

### Related reference:

- “CREATE INDEX statement” in the *SQL Reference, Volume 2*

---

## Using the Index Advisor

DB2 Spatial Extender provides a utility, called the *Index Advisor*, that lets you:

- Create a simulated grid index and tune this index until it becomes a satisfactory model for a real grid index.
- Determine whether to retain or replace an existing grid index.

This section explains how to use the Index Advisor and documents the syntax and parameters of the command that invokes it.

### Prerequisites:

Before you can access the data that you are indexing:

- If you want to use the ANALYZE clause to analyze a subset of the rows only, you must have a USER TEMPORARY table space available. Set the page size of this table space to at least 8 KB.
- Your user ID must hold the SELECT privilege on this table.

### Procedure for creating and tuning a simulated grid index:

To create and tune a simulated index:

1. Prepare to initiate the command that invokes the Index Advisor.
2. Ask the Index Advisor to recommend grid cell sizes for the index that you want to create.

## Using indexes and views

3. Have the Index Advisor collect statistics based on the recommended grid cell sizes.
4. Determine how well the recommended grid cell sizes facilitate retrieval.
5. If the statistics are not satisfactory, improve them by repeating steps 2, 3, and 4.

Details follow.

1. Prepare to initiate the command that invokes the Index Advisor:
  - a. From an operating system prompt, type `gseidx CONNECT TO mydb<`, where *mydb* is the name of the database that contains the table with the column whose data you want to index.
  - b. If you want to use another user ID to connect, specify this ID and its password. Precede the ID with the keyword `USER` and precede the password with the keyword `USING`. For example:  

```
gseidx CONNECT TO mydb USER user ID USING ppassword
```
2. Ask the Index Advisor to recommend grid cell sizes for the index that you want to create:
  - a. Type the command that invokes the Index Advisor. You can either:
    - Simply request grid cell sizes. To do so specify the `ADVISE` keyword.
    - Request grid cell sizes that are based on the data that your queries typically access. To do this, determine the overall extent of the data in the column for which you want to create an index. Next, determine what percentage of this extent is typically accessed by your queries. Then specify the `ADVISE QUERY BOX` keywords and set the *query-box-size* parameter to this percentage.
  - b. Run the command. The Index Advisor returns recommended grid cell sizes.
3. Have the Index Advisor collect statistics based on the recommended grid cell sizes.
  - a. Type the command again. This time specify keywords to request statistics based on the grid cell sizes that the Index Advisor recommended. You can ask for statistics on either all the data indexed or a subset of this data.
    - To obtain statistics on all indexed data, specify the `simulated-index` clause. In this clause, specify the grid cell size or sizes that were returned in step 2b.
    - To obtain statistics on indexed data in a subset of rows, specify:
      - The `simulated-index` clause. In this clause, specify the grid cell size or sizes that were returned in step 2b.

- The ANALYZE keyword and its parameters. These command elements let you determine either the number or percentage of rows that the Advisor should analyze to obtain statistics.

**Tip:** If the table includes over 1000 rows, and its data is distributed more or less uniformly, request an analysis of 1000 rows. If the data is not distributed uniformly, request an analysis of more than 1000 rows.

- b. Run the command. The Index Advisor returns statistics on the simulated index.
4. Determine how well the recommended grid cell sizes facilitate retrieval. You can do this by assessing the statistics returned in step 3b.

**Tips:**

- The statistic, “Index Entry/Geometry ratio”, should be 1 to 4
  - The statistic, “Number of Index Entries”, should be close or equal to the statistic, “Number of Geometries”.
5. If the statistics are not satisfactory, improve them by repeating steps 2, 3, and 4.

**Tip:** If the Index Advisor analyzed a subset of rows, and if the statistics are satisfactory, repeat steps 2, 3, and 4 multiple times, increasing the number of rows in the analysis each time. Continue to do this until the statistics are satisfactory with each repetition.

### **Procedure for determining whether to retain an existing grid index:**

Statistics on an existing grid index can tell you whether it is efficient, or whether it should be replaced by a more efficient index. To obtain these statistics and, if necessary, to replace the index:

1. Prepare to initiate the command that invokes the Index Advisor.
2. Have the Index Advisor collect statistics based on the grid cell sizes of the existing index.
3. Determine how well these grid cell sizes facilitate retrieval.
4. If the statistics are not satisfactory:
  - a. Create a simulated grid index and tune this index until it becomes a satisfactory model for a real grid index.
  - b. Drop the existing index and replace it with an index that matches the simulated one.

Details follow.

1. Prepare to initiate the command that invokes the Index Advisor:

## Using indexes and views

- a. From an operating system prompt, type `gseidx connect to mydb`, where *mydb* is the name of the database that contains the table with the column for which you want to provide or test an index.
  - b. If you want to connect to the database under a different user ID to connect, specify this ID and its password. Precede the ID with the keyword `USER` and precede the password with the keyword `USING`. For example:  

```
gseidx CONNECT TO mydb USER user ID USING password
```
2. Have the Index Advisor collect statistics based on the grid cell sizes of the existing index. You can ask for statistics on either all indexed data or a subset of this data.
    - To obtain statistics on all indexed data, type the **GET GEOMETRY STATISTICS | INFORMATION** command and specify its existing-index clause. Include the `DETAIL` keyword.
    - To obtain statistics on indexed data in a subset of rows, type the **GET GEOMETRY STATISTICS | INFORMATION** command and specify:
      - Its existing-index clause. Include the `DETAIL` keyword.
      - The `ANALYZE` keyword and its parameters. These command elements let you determine either the number or percentage of rows that the Index Advisor is to analyze to obtain statistics.

**Tip:** If the table includes over 1000 rows, and its data is distributed more or less uniformly, request an analysis of 1000 rows. If the data is not distributed uniformly, request an analysis of more than 1000 rows.

Run the command. The Index Advisor returns statistics on the existing index.

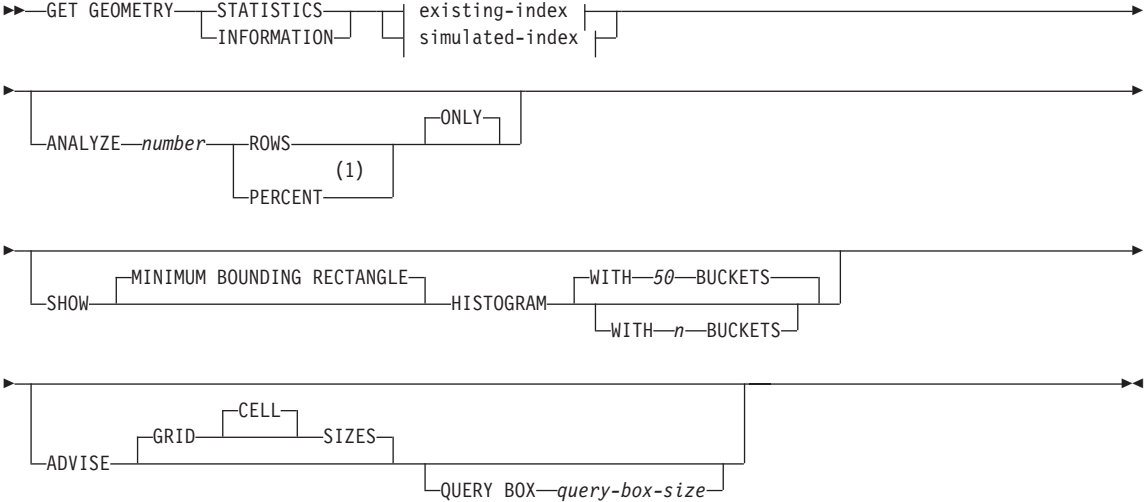
3. Determine how well the grid cell sizes of the existing index facilitate retrieval. You can do this by assessing the statistics returned in step 2.

### Tips:

- The statistic, “Index Entry/Geometry ratio”, should be 1 to 4
  - The statistic, “Number of Index Entries”, should be close or equal to the statistic, “Number of Geometries”.
4. If the statistics are not satisfactory:
    - a. Create a simulated grid index over the column for which the existing index is defined. Tune this simulated index until it becomes a satisfactory model for a real grid index.
    - b. Drop the existing index and replace it with an index that matches the simulated one.

**Syntax and parameters for the GET GEOMETRY STATISTICS | INFORMATION command:**

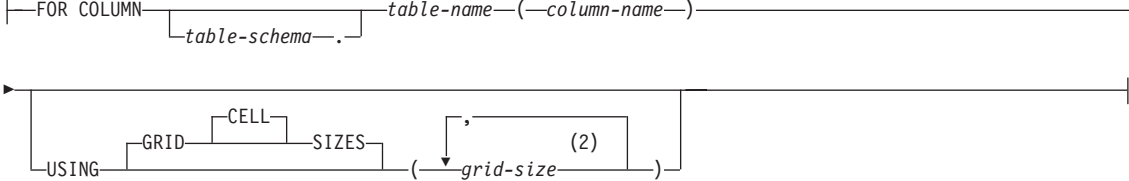
This section documents the syntax and parameters of the **GET GEOMETRY STATISTICS | INFORMATION** command. Figure 14 shows the syntax.



**existing-index:**



**simulated-index:**



**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.

Figure 14. Syntax of GET GEOMETRY STATISTICS | INFORMATION command

## Using indexes and views

The following text explains the keywords and parameters of the GET GEOMETRY STATISTICS | INFORMATION command.

### **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.
- Histogram showing relative frequency of geometries from cell to cell

### **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**

Instruction to gather statistics on data in a subset of table rows. Specify the approximate quantity or approximate percentage of the rows to be included in this subset.

### **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**

Refers 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. The default is to use 50 buckets.

### **ADVISE GRID CELL SIZES**

Computes close-to-optimal grid cell sizes.

### **QUERY BOX** *query-box-size*

Computes a close-to-optimal query box size. A *query box* is that percentage of the whole extent of the data in a column that is accessed by typical queries. If the typical queries access 1% of the data in the column, specify 0.01. If it accesses 0.03%, specify 0.0003.

### **Examples:**

The following example is a request to return detailed information about an existing grid index whose fully-qualified name is **TOOLBOX.GRIDIRON**:

```
GET GEOMETRY STATISTICS  
FOR INDEX TOOLBOX.GRIDIRON  
DETAIL
```

The following example is a request to return detailed information about a simulated index for column **POINT** in table **NEWTON.EXPPPOINT8**. Two grids are defined for this index, one with a cell size of .5 and the other with a cell size of 10. The Index Advisor is to base the information that it returns on an analysis of 1000 rows in column **POINT**.

```
GET GEOMETRY STATISTICS  
FOR COLUMN NEWTON.EXPPPOINT8(POINT)  
USING GRID SIZES (.5, 10)  
ANALYZE 1000 ROWS ONLY
```

### 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. If you have a table that has a spatial column and you want a view to use it, use the following sources of information.

**Related tasks:**

- “Creating a view” in the *Administration Guide: Implementation*

**Related reference:**

- “CREATE VIEW statement” in the *SQL Reference, Volume 2*



---

## Chapter 12. Analyzing and Generating spatial information

After you populate spatial columns, you are ready to query them. This chapter:

- Describes the environments in which you can submit queries
- Provides examples of the various types of spatial functions that you can invoke in a query
- Provides guidelines on using spatial functions in conjunction with spatial indexes

---

### Environments for performing spatial analysis

You can perform spatial analysis by using SQL and spatial functions in the following programming environments:

- Interactive SQL statements.  
You can enter interactive SQL statements from the DB2® Command Center, the DB2 Command Window, or the DB2 command line processor.
- Application programs in all languages supported by DB2.

---

### Examples of spatial function operations

DB2 Spatial Extender provides functions that perform various operations on spatial data. Generally speaking, these functions can be categorized according to the type of operation that they perform. Table 4 lists these categories, along with examples. The text following Table 4 shows coding for these examples.

*Table 4. Spatial functions and operations*

| <b>Category of function</b>                            | <b>Example of operation</b>   |
|--|---|
| 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. |

## Generating and analyzing spatial information

### 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) )
```

---

### Rules for using grid indexes to optimize spatial functions

A specialized group of spatial functions can improve query performance by exploiting a spatial index. 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 (zero). If the comparison cannot be performed, the function can return a null.

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.

These functions, called *comparison functions* in the documentation of DB2 Spatial Extender, are:

- `ST_Contains`
- `ST_Crosses`
- `ST_Disjoint`
- `ST_EnvIntersect`
- `ST_Equals`
- `ST_Intersects`
- `ST_MBRIntersects`
- `ST_Overlaps`
- `ST_Relate`
- `ST_Touches`
- `ST_Within`

Because of the time and memory required to execute a function, such execution can involve considerable processing. Furthermore, the more complex the geometries to be compared, the more complex and time-intensive the comparison will be. The specialized functions listed above can complete their operations more quickly if they can use a spatial grid index to locate geometries. To enable such a function to use a spatial grid 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 grid index cannot be used.

## Generating and analyzing spatial information

- The operator used in the predicate that compares the result of the function with another expression must be an equals sign.
- The expression on the right of the predicate must be the constant 1 (one).
- The function must be the expression on left of the predicate. For example:  
WHERE db2gse.ST\_Contains(s.sales\_zone,ST\_Point(121.8,37.3, 1)) = 1
- The operation must involve a search in a spatial column on which a spatial grid index is defined.

Table 5 shows correct and incorrect ways of creating spatial queries to utilize a spatial grid index.

*Table 5. Demonstration of how spatial functions can adhere to and violate rules for utilizing a spatial grid index.*

| <b>Queries that reference spatial functions</b>  | <b>Whether queries violate rules</b>   |
|--|--|
| <pre>SELECT * FROM stores AS s WHERE db2gse.ST_Contains(s.sales_zone,   ST_Point(-121.8,37.3, 1)) = 1</pre>  | No condition is violated in this example.  |
| <pre>SELECT * FROM stores AS s WHERE db2gse.ST_Length(s.location) &gt; 10</pre>  | The spatial function ST_Length does not compare geometries and cannot utilize a spatial grid index.    |
| <pre>SELECT * FROM stores AS s WHERE 1=db2gse.ST_Within(s.location,:BayArea)</pre>   | The function must be an expression on the left side of the predicate.                                  |
| <pre>SELECT * FROM stores AS s WHERE db2gse.ST_Contains(s.sales_zone,   ST_Point(-121.8,37.3, 1)) &lt;&gt; 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 grid index exists on either of the arguments for the function, so no index can be utilized. |

---

## Chapter 13. DB2 Spatial Extender commands

This chapter explains the commands used to set up DB2 Spatial Extender. It also explains how you use these commands to develop projects.

---

### Invoking commands for setting up DB2 Spatial Extender and developing projects

You can use a 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, consult Table 6. for the associated stored procedure topic for the command. For example, the **`db2se create_srs`** command requires the same authorities as the `db2.ST_create_srs` stored procedure.

**Exception:** The **`db2se shape_info`** command does not call a stored procedure. Rather, it displays information about the contents of shape files.

#### Procedure:

Enter `db2se` commands from an operating system prompt.

To find out what subcommands and parameters you can specify:

- Type `db2se` or `db2se -h`; then press enter. A list of `db2se` subcommands is displayed.
- Type `db2se` and a subcommand, or `db2se` and a subcommand followed by `-h`. 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.

**Important:** For your convenience, command syntax can be retrieved interactively on your monitor; you do not need to look up the syntax elsewhere.

## Commands

To issue a db2se command, type db2se. Then type a subcommand, followed by the parameters and parameter values that the subcommand requires.

Finally, press enter. Be aware that:

- 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 ID and password if you want to connect to the database as a user other than yourself. Always precede the ID with the parameter `userId` and precede the password with the parameter `pw`.

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"`.

**Note :** you might have to escape the quotation marks to ensure they are not interpreted by the system prompt (shell), for example, specify:

```
\ "myTable\ "
```

```
\ "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; for example:

```
"select * from newtable"
```

When the db2se command is executed, the stored procedure that corresponds to the command will be invoked, and the operation that you requested will be performed.

### Overview of db2se commands:

The following table indicates what db2se commands to issue to perform the tasks involved in setting up Spatial Extender and creating projects that use spatial data. This table also provides examples of db2se commands and refers you to information about authorizations and command-specific parameters. To the right of the task, in the second column, you will see a link or reference to information about a stored procedure. This stored procedure is called when the command is issued. Authorization to use the stored procedure is the same as the authorization to use the command; also, the command and stored procedure share the same parameters. To find out the authorization and the meanings of the parameters, click the link or turn to the section identified by the reference.

Table 6. db2se commands indexed by task

| Task                                   | Command and example  |
|--|--|
| Create a coordinate system.            | <p><b>db2se create_cs</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_create_coordsys stored procedure.</p> <p>The following example creates a coordinate system named mycoordsys.</p> <pre>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]]</pre> |
| Create a spatial reference system.     | <p><b>db2se create_srs</b></p> <p>Command-specific parameters are the same as those for the db2gse.ST_create_srs stored procedure. No authorization is required.</p> <p>The following example creates a spatial reference system named mysrs.</p> <pre>db2se create_srs mydb -srsName \"mysrs\" -srsID 100 -xScale 10 -coordsysName \"GCS_North_American_1983\"</pre>  |
| Drop a spatial reference system.       | <p><b>db2se drop_srs</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_drop_srs stored procedure.</p> <p>The following example drops a spatial reference system named mysrs.</p> <pre>db2se drop_srs mydb -srsName \"mysrs\"</pre>  |
| Delete a coordinate system definition. | <p><b>db2se drop_cs</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_drop_coordsys stored procedure.</p> <p>The following example drops a coordinate system named mycoordsys.</p> <pre>db2se drop_cs mydb -coordsysName \"mycoordsys\"</pre>   |

## Commands

Table 6. db2se commands indexed by task (continued)

| Task   | Command and example   |
|--|---|
| Disable a setup to geocode data automatically. | <p><b>db2se disable_autogc</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_disable_autogeocoding stored procedure.</p> <p>The following example disables the automatic geocoding for a geocoded column named my column in table mytable:</p> <pre>db2se disable_autogc mydb -tableName \mytable\ -columnName \mycolumn\</pre>  |
| Enable a database for spatial operations.      | <p><b>db2se enable_db</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_enable_db stored procedure.</p> <p>The following example enables a database named mydb for spatial operations.</p> <pre>db2se enable_db mydb</pre>   |
| Export data to an SDE transfer file.           | <p><b>db2se export_sde</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.GSE_export_sde stored procedure.</p> <p>The following example exports data from table mySDEtable, which contains spatial column mySpatialcolumn, to an SDE transfer file named mysdefile.</p> <pre>db2se export_sde mydb -tableName \mySDEtable\ -columnName \mySpatialcolumn\ -fileName /home/myaccount/mysdefile</pre> |
| Export data to shape files.                    | <p><b>db2se export_shape</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_export_shape stored procedure.</p> <p>The following example exports a spatial column named mycolumn and its associated table mytable to a shape file named myshapefile.</p> <pre>db2se export_shape mydb -fileName /home/myaccount/myshapefile -selectStatement "select * from mytable"</pre>                       |



Table 6. db2se commands indexed by task (continued)

| Task                         | Command and example   |
|------------------------------|---|
| Import an SDE transfer file. | <p><b>db2se import_sde</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.GSE_import_sde stored procedure.</p> <p>The following example imports an SDE transfer file named mysdefile to table mySDEtable, which contains a spatial column named mySpatialcolumn. A commit is to be issued for every ten records.</p> <pre>db2se import_sde mydb -tableName \"mysdetable\" -columnName \"mySpatialcolumn\" -fileName /home/myaccount/\"mysdefile\" -commitScope 10</pre>  |
| Import shapshapeefiles.      | <p><b>db2se import_shape</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_import_shape stored procedure.</p> <p>The following command imports a shapefile named myfile to a table named mytable. During the import, the spatial data in myfile is inserted into a mytable column named mycolumn.</p> <pre>db2se import_shape mydb -fileName \"myfile\" -srsName NAD83_SRS_1 -tableName \"mytable\" -spatialColumnName \"mycolumn\"</pre>  |
| Register a geocoder.         | <p><b>db2se register_gc</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_register_geocoder stored procedure.</p> <p>The following example registers a geocoder that is named mygeocoder and that is implemented by a function named myschema.myfunction.</p> <pre>db2se register_gc mydb -geocoderName \"mygeocoder\" -functionSchema \"myschema\" -functionName \"myfnctn\" -defaultParameterValues \"1, 'string',,cast(null as varchar(50))\" -vendor myvendor -description \"myvendor geocoder returning well-known text\"</pre> |

## Commands

Table 6. db2se commands indexed by task (continued)

| Task  | Command and example  |
|---|--|
| Register a spatial column.  | <p><b>db2se register_spatial_column</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_register_spatial_column stored procedure.</p> <p>The following example registers a spatial column named mycolumn in table mytable with spatial reference system USA_SRS_1.</p> <pre>db2se register_spatial_column mydb -tableName \"mytable\" -columnName \"mycolumn\" -srsName USA_SRS_1</pre> |
| Remove the resources that enable a database for spatial operations. | <p><b>db2se disable_db</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_disable_db stored procedure.</p> <p>The following example removes the resources that enable database mydb for spatial operations.</p> <pre>db2se disable_db mydb</pre>   |
| Remove a set-up for geocoding operations.                           | <p><b>db2se remove_gc_setup</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_remove_gc_setup stored procedure.</p> <p>The following example removes a set-up for geocoding operations that apply to a spatial column named mycolumn in table mytable.</p> <pre>db2se remove_geocoding_setup mydb -tableName \"mytable\" -columnName \"mycolumn\"</pre>                               |
| Run a geocoder in batch mode.                                       | <p><b>db2se run_gc</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_run_gc stored procedure.</p> <p>The following example runs a geocoder in batch mode to populate a column named mycolumn in a table named mytable.</p> <pre>db2se run_gc mydb -tableName \"mytable\" -columnName \"mycolumn\"</pre>   |

Table 6. db2se commands indexed by task (continued)

| Task                                    | Command and example   |
|---|---|
| Set up a geocoder to run automatically. | <p><b>db2se enable_autogeocoding</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_enable_autogeocoding stored procedure.</p> <p>The following example sets up automatic geocoding for a column named mycolumn in table mytable</p> <pre>db2se enable_autogeocoding mydb -tableName  \"mytable\" -columnName \"mycolumn\"</pre>  |
| Set up geocoding operations.            | <p><b>db2se setup_gc</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_setup_geocoding stored procedure.</p> <p>The following example sets up geocoding operations to populate a spatial column named mycolumn in table mytable.</p> <pre>db2se setup_gc mydb -tableName \"mytable\"  -columnName \"mycolumn\" -geocoderName  \"db2se_USA_GEOCODER\" -parameterValues  \"address,city,state,zip,2,90,70,20,1.1,'meter',4..\"  -autogeocodingColumns address,city,state,zip  commitScope 10</pre> |
| Show the contents of a shape file.      | <p><b>db2se shape_info</b></p> <p>To use this command, you must:</p> <ul style="list-style-type: none"> <li>• Have permission to read the file that the command references</li> <li>• Be able to connect to the database that contains this file if the database parameter is specified to look for compatible coordinates</li> </ul> <p>The following example shows the contents of a shape file named myfile.</p> <pre>db2se shape_info mydb -fileName \"myfile\"</pre>   |
| Unregister a geocoder.                  | <p><b>db2se unregister_gc</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_unregister_geocoder stored procedure.</p> <p>The following example unregisters a geocoder named mygeocoder.</p> <pre>db2se unregister_gc mydb -geocoderName \"mygeoco  der\"</pre>   |

## Commands

Table 6. db2se commands indexed by task (continued)

| Task  | Command and example  |
|---|--|
| Unregister a spatial column.                  | <p><b>db2se unregister_spatial_column</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_unregister_spatial_column stored procedure.</p> <p>The following example unregisters a spatial column named mycolumn in table mytable.</p> <pre>db2se unregister_spatial_column mydb -tableName  \"mytable\" -columnName \"mycolumn\"</pre>                         |
| Update a coordinate system definition.        | <p><b>db2se alter_cs</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_alter_coordsys stored procedure.</p> <p>The following example updates the definition of a coordinate system named mycoordsys with a new organization name.</p> <pre>db2se alter_cs mydb -coordsysName \"mycoordsys\"  -organization myNeworganizationb -tableName  \"mytable\"</pre> |
| Update a spatial reference system definition. | <p><b>db2se alter_srs</b></p> <p>Command-specific parameters and required authorizations are the same as those for the db2gse.ST_alter_srs stored procedure.</p> <p>The following example alters a spatial reference system named mysrs with a different xOffset and description.</p> <pre>db2se alter_srs mydb -srsName \"mysrs\"  -xOffset 35 -description \"This is my  own spatial reference system.\"</pre>     |

---

## Chapter 14. Writing applications and using the sample program

This chapter explains how you write Spatial Extender applications.

---

### Writing applications for DB2 Spatial Extender

If you plan to write application programs that invoke DB2 Spatial Extender stored procedures or functions, read the following task and reference information.

**Related concepts:**

- “The DB2 Spatial Extender sample program” on page 134

**Related tasks:**

- “Calling DB2 Spatial Extender stored procedures from an application” on page 132
- “Including the DB2 Spatial Extender header file in spatial applications” on page 131

---

### Including the DB2 Spatial Extender header file in spatial applications

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

**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 ensure that your DB2 Spatial Extender applications can use the necessary definitions in this header file:

1. 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 DB2 Universal Database is installed.

## Writing applications and using the sample program

2. Ensure that the path of the include directory is specified in your makefile with the compilation option.

---

### Calling DB2 Spatial Extender stored procedures from an application

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

#### Procedure:

To call DB2 Spatial Extender stored procedures, take the following actions:

- To call the ST\_enable\_db stored procedure, which enables 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 stored procedure is the only one in which you need to include an exclamation mark in the call (that is, db2gse!).

- To call any other DB2 Spatial Extender stored procedure, 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:

```
CALL db2gse.spatial_procedure_name
```

Notice that no exclamation mark is included in the preceding call.

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

Table 7.

| Stored procedure   | Description  |
|--------------------|--|
| GSE_export_sde     | Exports a spatial column and its associated table to an SDE transfer file. |
| GSE_import_sde     | Imports an SDE transfer file to a database.                                |
| 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.                               |

*Table 7. (continued)*

|                              |  |
|------------------------------|--|
| 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.               |
| ST_unregister_geocoder       | Unregisters a geocoder other than DB2SE_USA_GEOCODER.  |
| ST_unregister_spatial_column | Removes the registration of a spatial column.  |

### The DB2 Spatial Extender sample program

The DB2<sup>®</sup> 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 steps to verify the DB2 Spatial Extender installation are discussed in a separate topic.

- On UNIX, you can locate the `runGseDemo` program in the following path:  
`$HOME/sql1lib/samples/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\sql1lib\samples\spatial`

where `c:\Program Files\IBM\sql1lib` 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 will enable a database for spatial operations and perform spatial analysis on data in that database. This database will contain tables with customer and flood zone information. From this information you can 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

There are additional sample programs available through the DB2 Spatial Extender web site: [www.ibm.com/software/data/spatial/](http://www.ibm.com/software/data/spatial/)

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 the related links at the end of this section.



## Writing applications and using the sample program

The following table shows the sample program steps, the associated stored procedures, and a description of each step. In each step you will perform an action and in many cases then 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.

The C functions that invoke the stored procedures are displayed in parentheses. For more information about the stored procedures, see the related links at the end of this section.

*Table 8. DB2 Spatial Extender sample program steps*

| Steps                                  | Action and description   |
|--|--|
| Enable or disable the spatial database | <ul style="list-style-type: none"><li>• Enable the spatial database (stEnableDB)<br/>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.</li><li>• Disable the spatial database (stDisableDB)<br/>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.</li><li>• Enable the spatial database (stEnableDB)<br/>Same as above.</li></ul> |
| Create or drop a coordinate system     | <ul style="list-style-type: none"><li>• Create a coordinate system named NORTH_AMERICAN_TEST (stCreateCS)<br/>This step creates a new coordinate system in the database.</li><li>• Drop the coordinate system named NORTH_AMERICAN_TEST (stDropCS)<br/>This step drops the coordinate system NORTH_AMERICAN_TEST from the database.</li><li>• Create a coordinate system named NORTH_AMERICAN (stCreateCS)<br/>This step creates a new coordinate system, NORTH_AMERICAN, which will be used by the spatial reference system created in the next step.</li></ul>   |

## Writing applications and using the sample program

Table 8. DB2 Spatial Extender sample program steps (continued)

| Steps                                     | Action and description  |
|---|---|
| Create or drop a spatial reference system | <ul style="list-style-type: none"><li>• Create the spatial reference system named SRSDEMOTEST (stCreateSRS)<br/>This step defines a new spatial reference system (SRS) that is used to interpret the SRS. 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 CUSTOMERS table column.</li><li>• Drop the SRS named SRSDEMOTEST (stDropSRS)<br/>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.</li><li>• Create the SRS named SRSDEMO1 (stCreateSRS)</li></ul>   |
| Create and populate the spatial tables    | <ul style="list-style-type: none"><li>• Create the CUSTOMERS table (sqlSetupTables)</li><li>• Populate the CUSTOMERS table (sqlSetupTables)<br/>The CUSTOMERS table represents business data that has been stored in the database for several years.</li><li>• Alter the CUSTOMERS table by adding the LOCATION column (sqlSetupTables)<br/>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.</li><li>• Create the OFFICES table (sqlSetupTables)<br/>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.</li></ul> |

## Writing applications and using the sample program

Table 8. DB2 Spatial Extender sample program steps (continued)

| Steps                               | Action and description  |
|-------------------------------------|---|
| Populate the columns                | <ul style="list-style-type: none"><li>• Geocode the addresses data for the LOCATION column of CUSTOMERS table with the geocoder named DB2SE_USA_GEOCODER (stRunGC)<br/>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.</li><li>• Load the previously created OFFICES table from the SHAPE file using SRS SRSDEMO1 (stImportShape)<br/>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.</li><li>• Create and load the FLOODZONES table from the SHAPE file using SRS SRSDEMO1 (stImportShape)<br/>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.</li><li>• Create and load the REGIONS table from the SHAPE file using SRS SRSDEMO1 (stImportShape)</li></ul> |
| Register or unregister the geocoder | <ul style="list-style-type: none"><li>• Register the geocoder named SAMPLEGC2 (stRegisterGC)</li><li>• Unregister the geocoder named SAMPLEGC2 (stUnregisterGC)</li><li>• Register the geocoder SAMPLEGC (stRegisterGC)</li></ul> <p>These steps register and unregister the geocoder named SAMPLEGC2 and then create a new geocoder, SAMPLEGC, to use in the sample program.</p>   |

## Writing applications and using the sample program

Table 8. DB2 Spatial Extender sample program steps (continued)

| Steps  | Action and description   |
|--|--|
| Create spatial indexes                                       | <ul style="list-style-type: none"><li>• Create the spatial grid index for the LOCATION column of the CUSTOMERS table (sqlCreateIdx)</li><li>• Drop the spatial grid index for the LOCATION column of the CUSTOMERS table (sqlDropIdx)</li><li>• Create the spatial grid index for the LOCATION column of the CUSTOMERS table (sqlCreateIdx)</li><li>• Create the spatial grid index for the LOCATION column of the OFFICES table (sqlCreateIdx)</li><li>• Create the spatial grid index for the LOCATION column of the FLOODZONE table (sqlCreateIdx)</li><li>• Create the spatial grid index for the LOCATION column of the REGIONS table (sqlCreateIdx)</li></ul> <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"><li>• Set up geocoding for the LOCATION column of the CUSTOMERS table with geocoder DB2SE_USA_GEOCODER (stSetupGeocoding)<br/>This step associates the LOCATION column of the CUSTOMERS table with geocoder DB2SE_USA_GEOCODER and sets up the corresponding geocoding parameter values.</li><li>• Enable automatic geocoding for the LOCATION column of the CUSTOMERS table (stEnableAutoGC)<br/>This step turns on the automatic invocation of the geocoder. Using automatic geocoding causes the LOCATION, ADDRESS, CITY, STATE, and ZIP columns of the CUSTOMERS table to be synchronized with each other for subsequent insert and update operations.</li></ul>   |
| Insert, update, and delete operations on the CUSTOMERS table | <ul style="list-style-type: none"><li>• Insert some records with a different street (sqlInsDelUpd)</li><li>• Update some records with a new address (sqlInsDelUpd)</li><li>• Delete all records from the table (sqlInsDelUpd)</li></ul> <p>These steps demonstrate an insert, update, and delete operations on the ADDRESS, CITY, STATE, and ZIP columns of the CUSTOMERS table. After the automatic geocoding is enabled, data that is inserted or updated in these columns are automatically geocoded into the LOCATION column. This process was enabled in the previous step.</p>   |

## Writing applications and using the sample program

Table 8. DB2 Spatial Extender sample program steps (continued)

| Steps   | Action and description  |
|---|---|
| Disable automatic geocoding                               | <ul style="list-style-type: none"><li>• Disable automatic geocoding for the LOCATION column in the CUSTOMERS table (stDisableAutoGC)</li><li>• Remove the geocoding setup for the LOCATION column of the CUSTOMERS table (stRemoveGeocodingSetup)</li><li>• Drop the spatial index for the LOCATION column of the CUSTOMERS table (sqlDropIdx)</li></ul> <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> <p><b>Recommendation:</b> 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>   |
| Re-geocode the CUSTOMERS table                            | <ul style="list-style-type: none"><li>• Geocode the LOCATION column of the CUSTOMERS table again with a lower precision level: 90% instead of 100% (stRunGC)</li><li>• Re-create the spatial index for the LOCATION column of the CUSTOMERS table (sqlCreateIdx)</li><li>• Re-enable automatic geocoding with a lower precision level: 90% instead of 100% (stEnableAutoGC)</li></ul> <p>These steps run the geocoder in batch mode, re-create the spatial index, and re-enable the automatic geocoding with a new precision level. This action is recommended when a spatial administrator notices a high failure rate in the geocoding process. If the precision level is set to 100%, it might fail to geocode an address because it cannot find a matching address in the reference data. By reducing the precision level, the geocoder might be more successful in finding matching data. After the table is re-geocoded in batch mode, the automatic geocoding is re-enabled and the spatial index is re-created. This facilitates the incremental maintenance of the spatial index and the spatial column for subsequent insert and update operations.</p> |
| Create a view and register the spatial column in the view | <ul style="list-style-type: none"><li>• Create a view, HIGHRISKCUSTOMERS, based on the join of the CUSTOMERS table and the FLOODZONE table (sqlCreateView)</li><li>• Register the view's spatial column (stRegisterSpatialColumn)</li></ul> <p>These steps create a view and register its spatial column.</p>   |

## Writing applications and using the sample program

Table 8. DB2 Spatial Extender sample program steps (continued)

| Steps                                | Action and description   |
|--------------------------------------|--|
| Perform spatial analysis             | <ul style="list-style-type: none"><li>• Find the number of customers served by each region (ST_Within)</li><li>• 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)</li><li>• For each region, find the average income and premium of each customer (ST_Within)</li><li>• Find the number of flood zones that each office zone overlaps (ST_Overlaps)</li><li>• Find the nearest office from a specific customer location assuming that the office is located in the centroid of the office zone (ST_Distance)</li><li>• Find the customers whose location is close to the boundary of a specific flood zone (ST_Buffer, ST_Intersects)</li><li>• Find those high-risk customers within a specified distance from a specific office (ST_Within)</li></ul> <p>All of these steps use the <code>sqlRunSpatialQueries</code> stored procedure.</p> <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> |
| Export spatial data into SHAPE files | <ul style="list-style-type: none"><li>• Export the <code>HIGHRISKCUSTOMERS</code> view to SHAPE files (<code>stExportShape</code>)</li></ul> <p>This step shows an example of exporting the <code>HIGHRISKCUSTOMERS</code> 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).</p> <p>This step is included in the <code>runGseDemo.c</code> program but is commented out for reference only. You can modify the sample program to specify the location for the export SHAPE file, and rerun the sample program.</p>  |

## Writing applications and using the sample program

Table 8. DB2 Spatial Extender sample program steps (continued)

| Steps                       | Action and description   |
|-----------------------------|--|
| Export and import SDE files | <ul style="list-style-type: none"><li>• Export the CUSTOMERS table to an SDE transfer file (gseExportSDE)</li><li>• Import data from the newly exported SDE transfer file (gseImportSDE)</li></ul> <p>These steps show examples of exporting and importing SDE transfer files.</p> <p>These steps are including in the runGseDemo.c program but are commented out for reference only. You can modify the sample program to specify the location for the export SHAPE file, and rerun the sample program.</p> |

### Related tasks:

- “Verifying the Spatial Extender installation” on page 42
- “Troubleshooting tips for the installation sample program” on page 44
- “Writing applications for DB2 Spatial Extender” on page 131
- “Calling DB2 Spatial Extender stored procedures from an application” on page 132
- “Including the DB2 Spatial Extender header file in spatial applications” on page 131

## Writing applications and using the sample program



---

## Chapter 15. Identifying DB2 Spatial Extender problems

If you encounter a problem working with DB2 Spatial Extender, you need to 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 `db2diag.log` file, which records diagnostic information about errors. The information in this file is intended primarily for an IBM customer support representative.
- 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.

This chapter discusses each of these approaches.

---

### How to interpret DB2 Spatial Extender messages

You can work with DB2<sup>®</sup> Spatial Extender through four different interfaces:

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

All interfaces return DB2 Spatial Extender messages to help you determine whether the spatial operation that you requested completed successfully or resulted in an error.

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

GSE0000I: The operation was completed successfully.

## Identifying problems

Table 9. 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:<br><br>C      Critical error messages<br>N      Non-critical error messages<br>W      Warning messages<br>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 "? GSEnnn" | more
```

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

### Related concepts:

- “DB2 Spatial Extender stored procedure output parameters” on page 146
- “DB2 Spatial Extender function messages” on page 148
- “DB2 Spatial Extender CLP messages” on page 150
- “DB2 Control Center messages” on page 153
- “The db2diag.log utility” on page 155

### Related tasks:

- “Tracing DB2 Spatial Extender problems with the db2trc command” on page 154

### Related reference:

- “GSE messages” in the *Message Reference: Volume 1*

---

### DB2 Spatial Extender stored procedure output parameters

DB2<sup>®</sup> Spatial Extender stored procedures are invoked *implicitly* when you enable and use Spatial Extender from the DB2 Control Center or when you use the DB2 Spatial Extender CLP (db2se). You can invoke stored procedures *explicitly* in an application program or from the DB2 command line.

This topic describes how to diagnose problems when stored procedures are invoked explicitly in application programs or from the DB2 command line. To diagnose stored procedures invoked implicitly, you use the messages returned by the DB2 Spatial Extender CLP or the messages returned by the DB2 Control Center. These messages are discussed in separate topics.

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 10. 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 is comprised 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](#).

### 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` stored procedure. The example below 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
```

## Identifying problems

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
               numerical 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
```

### Related concepts:

- “How to interpret DB2 Spatial Extender messages” on page 143
- “DB2 Spatial Extender function messages” on page 148
- “DB2 Spatial Extender CLP messages” on page 150
- “DB2 Control Center messages” on page 153

### Related reference:

- “GSE messages” in the *Message Reference: Volume 1*

---

## DB2 Spatial Extender function messages

The messages returned by DB2<sup>®</sup> 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. For example:

- 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 11. 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.<br><br>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 | The DB2 Spatial Extender message explanation.   |
| SQLSTATE=38SSL        | An SQLSTATE code that further identifies the error. An SQLSTATE code is returned for each statement or row. <ul style="list-style-type: none"> <li>The SQLSTATE codes for Spatial Extender function errors are 38Sxx, where each x is a character letter or number.</li> <li>The SQLSTATE codes for Spatial Extender function warnings are 01HSx, where the x is a character letter or number.</li> </ul> |

### 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 below:

```
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
```

## Identifying problems

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 the above 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.

### Related concepts:

- “How to interpret DB2 Spatial Extender messages” on page 143
- “DB2 Spatial Extender stored procedure output parameters” on page 146
- “DB2 Spatial Extender CLP messages” on page 150
- “DB2 Control Center messages” on page 153

### Related reference:

- “GSE messages” in the *Message Reference: Volume 1*

---

## DB2 Spatial Extender CLP messages

The DB2<sup>®</sup> Spatial Extender CLP (db2se) 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.
- Migration operations.
- Import and export shape operations to and from the client.

### Examples of stored procedure messages 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 is comprised 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 shape information messages 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.
```

## Identifying problems

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 migration messages returned by the Spatial Extender CLP:

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

Suppose you invoke the migration of the database mydb using the command `db2se migrate mydb -messagesFile /tmp/migrate.msg`. The message text returned by the Spatial Extender CLP is:

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

### Related concepts:

- “How to interpret DB2 Spatial Extender messages” on page 143
- “DB2 Spatial Extender stored procedure output parameters” on page 146
- “DB2 Spatial Extender function messages” on page 148
- “DB2 Control Center messages” on page 153

### Related reference:

- “GSE messages” in the *Message Reference: Volume 1*

## DB2 Control Center messages

When you work with DB2<sup>®</sup> Spatial Extender through the DB2 Control Center, messages will appear in the DB2 Message window. Most of the messages that you will encounter will be DB2 Spatial Extender messages. Occasionally, you will receive an SQL message. The SQL messages are returned when an error involves licensing, locking, or when a DAS service is not available. The following sections provide examples of how DB2 Spatial Extender messages and SQL messages will appear in the DB2 Control Center.

### DB2 Spatial Extender messages:

When you receive a DB2 Spatial Extender message through the Control Center, the entire message text appears in the text area of DB2 Message window, for example:

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

### SQL messages:

When you receive an SQL message through the Control Center that pertains to DB2 Spatial Extender:

- The message identifier, message number, and message type appear on the left side of the DB2 Message window, for example: SQL0612N.
- The message text appears in the text area of the DB2 Message window.

The message text that appears in the DB2 Message window might contain the SQL message text and the SQLSTATE, or it might contain the message text and the detailed explanation and user response.

An example of an SQL message that contains the SQL message text and the SQLSTATE is:

```
[IBM][CLI Driver][DB2/NT] SQL0612N "<name>" is a duplicate name. SQLSTATE=42711
```

An example of an SQL message that contains the message text and the detailed explanation and user response is:

```
SQL8008N
The product "DB2 Spatial Extender" does not have a valid
license key installed and the evaluation period has expired.
```

Explanation:

A valid license key could not be found and the evaluation period has expired.

User Response:

## Identifying problems

Install a license key for the fully entitled version of the product. You can obtain a license key for the product by contacting your IBM® representative or authorized dealer.

### Related concepts:

- “How to interpret DB2 Spatial Extender messages” on page 143
- “DB2 Spatial Extender stored procedure output parameters” on page 146
- “DB2 Spatial Extender function messages” on page 148
- “DB2 Spatial Extender CLP messages” on page 150

### Related reference:

- “GSE messages” in the *Message Reference: Volume 1*

---

## 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. The DB2 trace facility is activated by the **db2trc** system 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.

On UNIX operating systems, you must have SYSADM, SYSCTRL, or SYSMANT authorization to trace a DB2 instance.

On Windows operating systems, no special authorization is required.

### Procedure:

To trace the DB2 Spatial Extender events to memory, you follow these basic steps:

1. Shut down all other applications.
2. Turn the trace on. The DB2 Support technical support representative will provide you with the specific parameters for this step. The basic command is:  

```
db2trc on
```

**Restriction:** The **db2trc** command must be entered at a 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.

You can trace to memory or to a file. The preferred method for tracing is to trace to memory. If the problem being re-created 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, 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
```

You should dump the trace immediately after the problem occurs.

5. Turn the trace off, 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
```

### Related concepts:

- “DB2 trace (db2trc)” in the *Troubleshooting Guide*
- “How to interpret DB2 Spatial Extender messages” on page 143
- “The db2diag.log utility” on page 155

### Related reference:

- “GSE messages” in the *Message Reference: Volume 1*

---

## The db2diag.log utility

Diagnostic information about errors is recorded in the db2diag.log text file. This information is used for problem determination and is intended for DB2® technical support.

## Identifying problems

The db2diag.log and administration notification log are files that contain text information logged by DB2 as well as DB2 Spatial Extender. They are located in the directory specified by the DIAGPATH database manager configuration parameter. On Windows® NT, Windows 2000, and Windows XP systems, the DB2 administration notification log is found in the event log and can be reviewed through the Windows Event Viewer.

The information that DB2 records in the administration logs 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 logs grow continuously. When they get too large, back them up and then erase the file. A new set of files is generated automatically the next time they are required by the system.

### **Related concepts:**

- "Interpreting the administration logs" in the *Troubleshooting Guide*
- "How to interpret DB2 Spatial Extender messages" on page 143

### **Related tasks:**

- "Tracing DB2 Spatial Extender problems with the db2trc command" on page 154

### **Related reference:**

- "GSE messages" in the *Message Reference: Volume 1*

---

## Part 4. Reference material





---

## Chapter 16. Stored procedures

This section provides reference information about the stored procedures that you can use to set up DB2 Spatial Extender and create projects that use spatial data. When you set up DB2 Spatial Extender or create projects from the DB2 Control Center or the DB2 command line processor, you invoke these stored procedures implicitly. For example, when you click **OK** from a DB2 Spatial Extender window in the DB2 Control Center, DB2 calls the stored procedure that is associated with that window.

Alternatively, you can invoke the stored procedures explicitly in an application program.

Before invoking most DB2 Spatial Extender stored procedures on a database, you must enable that database for spatial operations by invoking the `ST_enable_db` stored procedure, either directly or by using the DB2 Control Center. (You can read about invoking this stored procedure in the topic about `ST_enable_db`, later in this section.)

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:

- `GSE_export_sde`
- `GSE_import_sde`
- `ST_alter_coordsys`
- `ST_alter_srs`
- `ST_create_coordsys`
- `ST_create_srs`
- `ST_disable_autogeocoding`
- `ST_disable_db`
- `ST_drop_coordsys`
- `ST_drop_srs`
- `ST_enable_autogeocoding`
- `ST_enable_db`
- `ST_export_shape`
- `ST_import_shape`

## Stored procedures

- ST\_register\_geocoder
- ST\_register\_spatial\_column
- ST\_remove\_geocoding\_setup
- ST\_run\_geocoding
- ST\_setup\_geocoding
- ST\_unregister\_geocoder
- ST\_unregister\_spatial\_column

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

---

### GSE\_export\_sde

Use this stored procedure to export a spatial column and its associated table to an SDE transfer file.

#### Restrictions:

- Exactly one spatial column must exist in the table or view.
- The spatial column must be registered.
- You cannot append to existing SDE files.

#### Authorization:

The user ID under which the stored procedure is invoked must have either SYSADM or DBADM authority. In addition, this user ID under must hold the SELECT privilege on the table that is to be exported.

#### Syntax:

```
► db2gse.GSE_export_sde—(—table_schema—, —table_name—, —column_name—, —  
                                  —null—)  
► —file_name—, —where_clause—)  
                                  —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 you are exporting. 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 registered spatial column that you are exporting. 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).

#### *file\_name*

Names the SDE transfer file to which the specified spatial column and its associated table are to be exported. You must specify a non-null value for this parameter.

The data type of this parameter is VARCHAR(256).

#### *where\_clause*

Specifies the body of the SQL WHERE clause, which defines a restriction on the set of records that are to be exported. 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.

If this parameter is specified, the value can reference any attribute column in the table that you are exporting.

The data type of this parameter is VARCHAR(1024).

### **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

## GSE\_export\_sde

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 GSE\_export\_sde stored procedure. This example uses a DB2 CALL command to export data from a table named CUSTOMERS to SDE files:

```
call db2gse.GSE_export_sde(NULL, 'CUSTOMERS', 'LOCATION', '/tmp/export_sde_file',  
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.

### **Related reference:**

- “GSE\_import\_sde” on page 162

---

## GSE\_import\_sde

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

- If the SDE transfer file is targeted for an existing table that has a registered spatial column, DB2 Spatial Extender loads the table with the file’s data.
- Otherwise, DB2 Spatial Extender creates a table that has a spatial column, registers this column, and loads the spatial column and the table’s other columns with the file’s data.

The spatial reference system that is specified in the SDE transfer file is compared with the spatial reference systems that are registered to DB2 Spatial Extender. If the specified system matches a registered system, all data values in the transfer data, when loaded, are modified in the way that the registered system specifies. If the specified system matches none of the registered systems, DB2 Spatial Extender creates a new spatial reference system to specify the modifications.

**Authorization:**

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

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

When the table to which you want to import data must be created, the user ID under which this stored procedure is invoked must hold either SYSADM or DBADM authority on the database that contains the table that is to be created.

**Syntax:**

```

▶▶ db2gse.GSE_import_sde—(—table_schema—, —table_name—, —column_name—, —————▶
└──┬───┘
  null
)
▶—file_name—, —commit_scope—)—————▶▶
└──┬───┘
  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 into which the SDE transfer data 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 of this parameter is VARCHAR(128) or, if you enclose the value in double quotation marks, VARCHAR(130).

*column\_name*

Names the registered column into which the SDE transfer file's spatial data is to be loaded. You must specify a non-null value for this parameter.

## GSE\_import\_sde

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).

### *file\_name*

Names the SDE transfer file that is to be imported. You must specify a non-null value for this parameter.

The data type of this parameter is VARCHAR(256).

### *commit\_scope*

Specifies the number of records that are to be imported before a COMMIT is issued. 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 no records are committed.

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 GSE\_import\_sde stored procedure. This example uses a DB2 CALL command to import an SDE file named tmp/customerSDE into a table named CUSTOMERS. This CALL command specifies that a COMMIT is to be performed after every 5 records are imported:

```
call db2gse.GSE_import_sde(NULL, 'CUSTOMERS', 'LOCATION',  
    '/tmp/customerSde', 5, ?,?)
```

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.

**Related reference:**

- “GSE\_export\_sde” on page 160

## ST\_alter\_coordsys

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 either SYSADM or DBADM authority.

**Syntax:**

```
▶▶ db2gse.ST_alter_coordsys (—coordsys_name—, —definition—, —————▶
                                [null]
▶ —organization—, —organization_coordsys_id—, —description—) —————▶▶
  [null]          [null]          [null]
```

**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

## ST\_alter\_coordsys

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.

**Related reference:**

- “ST\_create\_coordsys” on page 172
- “ST\_drop\_coordsys” on page 186

## ST\_alter\_srs

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.

## ST\_alter\_srs

### Authorization:

The user ID under which the stored procedure is invoked must have either SYSADM or DBADM authority.

### Syntax:

```
db2gse.ST_alter_srs—(—srs_name—, —srs_id—, —x_offset—, —x_scale—, —y_offset—, —y_scale—, —z_offset—, —z_scale—, —m_offset—, —m_scale—, —coordsys_name—, —description—)
```

The diagram shows the syntax for the `ST_alter_srs` stored procedure. The parameters are: `srs_name`, `srs_id`, `x_offset`, `x_scale`, `y_offset`, `y_scale`, `z_offset`, `z_scale`, `m_offset`, `m_scale`, `coordsys_name`, and `description`. Each parameter name is enclosed in a box with the word "null" underneath it, indicating that the parameter is optional. The entire syntax is enclosed in a large arrow pointing to the right.

### 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.

## ST\_alter\_srs

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.

**Related reference:**

- “ST\_drop\_srs” on page 187

## ST\_alter\_srs

- “ST\_create\_srs” on page 174

---

## ST\_create\_coordsys

Use this stored procedure to store information in the database about 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 either SYSADM or DBADM authority.

### Syntax:

```
db2gse.ST_create_coordsys(coordsys_name, definition, organization  
                           organization_coordsys_id, description)
```

The parameters *organization*, *organization\_coordsys\_id*, and *description* are optional and can be null.

### 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:**

## ST\_create\_coordsys

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.

### Related reference:

- “ST\_drop\_srs” on page 187
- “ST\_alter\_srs” on page 167

---

## ST\_create\_srs

Use these stored procedures 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.

This stored procedure replaces db2gse.gse\_enable\_sref.

#### Authorization:

None required.

#### Syntax:

##### With conversion factors (version 1):

```
▶ db2gse.ST_create_srs(—srs_name—, —srs_id—, —x_offset—, —x_scale—, —
  [null]
▶ —y_offset—, —y_scale—, —z_offset—, —z_scale—, —m_offset—
  [null] [null] [null] [null] [null]
▶ —, —m_scale—, —coordsys_name—, —description—)
  [null] [null]
```

##### With maximum possible extend (version 2):

```
▶ db2gse.ST_create_srs(—srs_name—, —srs_id—, —x_min—, —x_max—, —x_scale—, —
▶ —, —y_min—, —y_max—, —y_scale—, —z_min—
  [null]
▶ —, —z_max—, —z_scale—, —m_min—, —m_max—, —m_scale—, —coordsys_name—
  [null] [null]
▶ —, —description—)
  [null]
```

#### Parameter descriptions:

##### With conversion factors (version 1):

###### *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).

## ST\_create\_srs

### *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\_offset*

Specifies the offset for all X coordinates of geometries that are represented in this spatial reference system. 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.) 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.

### *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. Either the *x\_offset* value is specified explicitly, or a default *x\_offset* value of 0 is used. You must specify a non-null value for this parameter.

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. 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.

## ST\_create\_srs

### *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 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 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.

## ST\_create\_srs

### *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 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 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:**

## ST\_create\_srs

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

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.

This stored procedure replaces db2gse.gse\_disable\_autogc.

### Authorization:

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

- SYSADM or DBADM 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—,——————►  
                                  —null—)
```



►—*column\_name*—)—————▶

### 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

## ST\_disable\_autogeocoding

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.

### Related reference:

- “ST\_enable\_autogeocoding” on page 189
- “ST\_setup\_geocoding” on page 220

---

## ST\_disable\_db

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

This stored procedure replaces db2gse.gse\_disable\_db.

### Authorization:

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

### Syntax:

```
►►—db2gse.ST_disable_db—(—force—)—————►►
                        └—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:

## ST\_disable\_db

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.

### Related reference:

- “ST\_alter\_coordsys” on page 165
- “ST\_create\_coordsys” on page 172

---

## ST\_drop\_coordsys

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 removed 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 either SYSADM or DBADM authority.

### Syntax:

```
►►—db2gse.ST_drop_coordsys—(—coordsys_name—)—————►►
```

### 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**

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.

This stored procedure replaces db2gse.gse\_disable\_sref.

## ST\_drop\_srs

### Authorization:

The user ID under which the stored procedure is invoked must have either SYSADM or DBADM authority.

### Syntax:

```
►—db2gse.ST_drop_srs—(—srs_name—)—————►
```

### 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).

### Example:

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.

**Related reference:**

- “ST\_create\_srs” on page 174
- “ST\_alter\_srs” on page 167

---

## ST\_enable\_autogeocoding

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 invoking the ST\_setup\_geocoding stored 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.

This stored procedure replaces db2gse.gse\_enable\_autogc.

**Authorization:**

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

- SYSADM or 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 SYSADM or 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

## ST\_enable\_autogeocoding

- 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—, —————►  
                                  |  
                                  —null—  
► —column_name—)—————►
```

### 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.

### **Related reference:**

- “ST\_setup\_geocoding” on page 220

---

## ST\_enable\_db

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.

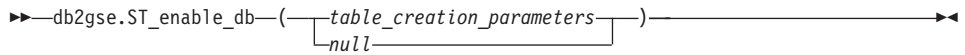
This stored procedure replaces db2gse.gse\_enable\_db.

### **Authorization:**

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

### **Syntax:**

## ST\_enable\_db



### 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);

```

### Related reference:

- "ST\_disable\_db" on page 184

## ST\_export\_shape

---

### ST\_export\_shape

Use this stored procedure to export a spatial column and its associated table to a shape file.

This stored procedure replaces db2gse.gse\_export\_shape.

#### 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 , output_column_names , select_statement , messages_file )
```

The syntax diagram shows the following structure:

- db2gse.ST\_export\_shape (
- file\_name* ,
- append\_flag* (with a *null* alternative),
- output\_column\_names* (with a *null* alternative),
- select\_statement* ,
- messages\_file* (with a *null* alternative)
- )

#### 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” on page 196 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

## ST\_export\_shape

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 12. 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\_export\_shape

### Related reference:

- “ST\_import\_shape” on page 198

---

## ST\_import\_shape

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.

This stored procedure replaces db2gse.gse\_import\_shape.

### 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:
  - SYSADM or DBADM
  - 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:
  - SYSADM or 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—, —srs_name—, —
      |null|
▶ —table_schema—, —table_name—, —table_attr_columns—, —
      |null|
▶ —create_table_flag—, —table_creation_parameters—, —spatial_column—
      |null|
▶ , —type_schema—, —type_name—, —inline_length—, —id_column—, —
      |null|
▶ —id_column_is_identity—, —restart_count—, —commit_scope—, —
      |null|
▶ —exception_file—, —messages_file—
      |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” on page 206 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)
```

## ST\_import\_shape

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” on page 206 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.

## ST\_import\_shape

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 )
```

## ST\_import\_shape

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 no records are committed.

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” on page 206 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

## ST\_import\_shape

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 13. Relationship between DB2 data types and dBASE attribute data types

| .dbf type | .dbf length <sup>b</sup> (See note) | .dbf decimals <sup>b</sup> (See note) | 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,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:

- *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
- *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,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.

### Related reference:

- “ST\_export\_shape” on page 194

---

## ST\_register\_geocoder

Use this stored procedure to register a geocoder other than the DB2SE\_USA\_GEOCODER geocoder, which is shipped with DB2 Spatial Extender. The DB2SE\_USA\_GEOCODER geocoder is registered by DB2 Spatial Extender when the database is enabled.

**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

## ST\_register\_geocoder

- The values to use with the ST\_create\_srs parameters 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.

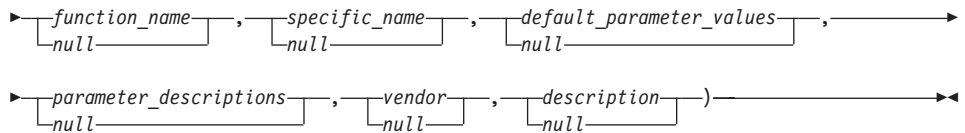
This stored procedure replaces db2gse.gse\_register\_gc.

### Authorization:

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

### Syntax:

```
►► db2gse.ST_register_geocoder—(—geocoder_name—, —function_schema—, —————→  
                                        └—null—┘
```



### 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.

## ST\_register\_geocoder

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.

**Recommendation:** Include the following information:

- Coordinate system name if spatial data, such as well-known text (WKT) or well-known binary (WKB), is to be returned
- Spatial reference system, if ST\_Geometry or any of its subtypes are to be returned
- Name of the geographical area to which this geocoder applies
- Any other information about the geocoder that users should know

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.

## ST\_register\_geocoder

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.

### Related reference:

- “ST\_unregister\_geocoder” on page 224

---

## ST\_register\_spatial\_column

Use this stored procedure to register a spatial column and to associate a spatial reference system (SRS) with it. When this stored procedure is processed, information about the spatial column that is being registered is added to 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.

This stored procedure replaces `db2gse.gse_register_layer`.

### Authorization:

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

- SYSADM or DBADM authority on the database that contains the table to which the spatial column that is being registered belongs
- CONTROL or ALTER privilege on this table

### Syntax:

```

▶—db2gse.ST_register_spatial_column—(—table_schema—, —table_name—, —————▶
                                     └──┬──┘
                                     └──┴──┘
                                     null
▶—column_name—, —srs_name—)—————▶

```

### 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).

## ST\_register\_spatial\_column

### *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).

### **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.

### **Related reference:**

- “ST\_unregister\_spatial\_column” on page 226



## ST\_remove\_geocoding\_setup

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:

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

### Syntax:

```
►► db2gse.ST_remove_geocoding_setup ( ( table_schema | null ), table_name , column_name )
```

### 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.

## ST\_remove\_geocoding\_setup

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.

### **Related reference:**

- “ST\_setup\_geocoding” on page 220

---

**ST\_run\_geocoding**

Use this stored procedure to run a geocoder in batch mode on a geocoded column.

This stored procedure replaces db2gse.gse\_run\_gc.

**Authorization:**

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

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

**Syntax:**

```

▶ db2gse.ST_run_geocoding( ( table_schema , table_name , column_name ,
                             null
▶ geocoder_name , parameter_values , where_clause ,
                             null
▶ commit_scope )
                             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.

## ST\_run\_geocoding

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 no COMMIT is performed.

The data type of this parameter is INTEGER.

### Output parameters:

## ST\_run\_geocoding

### *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 DB2SE\_USA\_GEOCODER 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',  
    'DB2SE_USA_GEOCODER',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.

### **Related reference:**

- “ST\_setup\_geocoding” on page 220

---

## ST\_setup\_geocoding

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 that is set up here is recorded in 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 stored 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:

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

### Syntax:

```

▶▶ db2gse.ST_setup_geocoding—(
    [table_schema] , [table_name] ,
    [column_name] , [geocoder_name] , [parameter_values] ,
    [autogeocoding_columns] , [where_clause] , [commit_scope] )

```

Diagram illustrating the syntax of the ST\_setup\_geocoding stored procedure. The procedure signature is: db2gse.ST\_setup\_geocoding—( [table\_schema] , [table\_name] , [column\_name] , [geocoder\_name] , [parameter\_values] , [autogeocoding\_columns] , [where\_clause] , [commit\_scope] ). Brackets indicate that each parameter is optional and 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 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

## ST\_setup\_geocoding

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:**

## ST\_setup\_geocoding

### *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 DB2SE\_USA\_GEOCODER:

```
call db2gse.ST_setup_geocoding(NULL, 'CUSTOMERS', 'LOCATION',
'DB2SE_USA_GEOCODER', 'ADDRESS,CITY,STATE,ZIP,1,100,80,,, "$HOME/sql1lib/
gse/refdata/ky.edg", "$HOME/sql1lib/samples/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

### Related reference:

- “ST\_unregister\_geocoder” on page 224
- “ST\_remove\_geocoding\_setup” on page 215

---

## ST\_unregister\_geocoder

Use this stored procedure to unregister a geocoder other than the DB2SE\_USA\_GEOCODER geocoder, which is shipped with DB2 Spatial Extender.

**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.

This stored procedure replaces db2gse.gse\_unregister\_gc.

### Authorization:

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

### Syntax:

►—db2gse.ST\_unregister\_geocoder—(*—geocoder\_name—*)—————►

### 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).

## ST\_unregister\_geocoder

### 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.

### Related reference:

- “ST\_register\_geocoder” on page 207
- “ST\_setup\_geocoding” on page 220

---

## ST\_unregister\_spatial\_column

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.

This stored procedure replaces db2gse.gse\_unregister\_layer.

### Authorization:

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

- SYSADM or DBADM authority
- CONTROL or ALTER privilege on this table

### Syntax:

```
▶—db2gse.ST_unregister_spatial_column—(—

|                     |
|---------------------|
| <i>table_schema</i> |
| null                |

—,—table_name—,—  
▶—column_name—)▶▶
```

### 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.

## ST\_unregister\_spatial\_column

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.

### Related reference:

- “ST\_register\_spatial\_column” on page 212

---

## Chapter 17. Catalog views

Spatial Extender's catalog views contain information about:

**"The DB2GSE.ST\_COORDINATE\_SYSTEMS catalog view"**

Coordinate systems that you can use

**"The DB2GSE.ST\_GEOMETRY\_COLUMNS catalog view" on page 231**

Spatial columns that you can populate or update.

**"The DB2GSE.ST\_GEOCODERS catalog view" on page 233 and "The DB2GSE.ST\_GEOCODING\_PARAMETERS catalog view" on page 236**

Geocoders that you can use

**"The DB2GSE.ST\_GEOCODING catalog view" on page 234 and "The DB2GSE.ST\_GEOCODING\_PARAMETERS catalog view" on page 236**

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

**"The DB2GSE.ST\_SIZINGS catalog view" on page 238**

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

**"The DB2GSE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view" on page 238**

Spatial reference systems that you can use.

**"The DB2GSE.ST\_UNITS\_OF\_MEASURE catalog view" on page 242**

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

---

### The DB2GSE.ST\_COORDINATE\_SYSTEMS catalog view

When you enable a database for spatial operations, multiple coordinate systems are automatically registered in the DB2 Spatial Extender catalog. When users define additional coordinate systems to the database, these coordinate systems are also automatically registered in the catalog. To retrieve information about registered coordinate systems, query the DB2GSE.ST\_COORDINATE\_SYSTEMS catalog view. For a description of columns in this view, see the following table.

## DB2GSE.ST\_COORDINATE\_SYSTEMS catalog view

Table 14. 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.   |
| COORDSYS_TYPE            | VARCHAR(128)  | No        | Type of this coordinate system:<br><b>PROJECTED</b><br>Two-dimensional.<br><b>GEOGRAPHIC</b><br>Three-dimensional. Uses X and Y coordinates.<br><b>GEOCENTRIC</b><br>Three-dimensional. Uses X, Y, and Z coordinates.<br><b>UNSPECIFIED</b><br>Abstract or non-real world coordinate system.<br>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.<br><br>This column is null if the ORGANIZATION_COORDSYS_ID column is null.  |
| ORGANIZATION_COORDSYS_ID | INTEGER       | Yes       | 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.<br><br>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.  |



---

**The 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's name and numeric identifier. For additional information about spatial columns, query DB2's SYSCAT.COLUMN catalog view.

For a description of DB2GSE.ST\_GEOMETRY\_COLUMNS, see the following table.

*Table 15. Columns in the DB2GSE.ST\_GEOMETRY\_COLUMNS catalog view*

| <b>Name</b>  | <b>Data type</b> | <b>Nullable?</b> | <b>Content</b>   |
|--------------|------------------|------------------|--|
| 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.<br><br>The combination of TABLE_SCHEMA, TABLE_NAME, and COLUMN_NAME uniquely identifies the column.   |
| TYPE_SCHEMA  | VARCHAR(128)     | No               | 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. |

---

### The DB2GSE.ST\_GEOCODER\_PARAMETERS catalog view

When you enable a database for spatial operations, information about the parameters of the supplied geocoder, DB2GSE\_USA\_GEOCODER, is automatically recorded in the DB2 Spatial Extender catalog. If you register additional geocoders, information about their parameters is also recorded in the catalog. To retrieve information about a geocoders' parameters from the catalog, query the DB2GSE.ST\_GEOCODER\_PARAMETERS catalog view. For a description of columns in this view, see the following table.

For more information about geocoders' parameters, query DB2's SYSCAT.ROUTINEPARMS catalog view. For a description of this view, see the *SQL Reference*.

Table 16. 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.<br><br>The combined values in the GEOCODER_NAME and ORDINAL columns uniquely identify this parameter.<br><br>A record in DB2's 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.<br><br>The content of the PARAMETER_NAME column is obtained from the DB2 catalog.   |

## DB2GSE.ST\_GEOCODER\_PARAMETERS catalog view

Table 16. Columns in the DB2GSE.ST\_GEOCODER\_PARAMETERS (continued)

| Name              | Data type     | Nullable? | Content  |
|-------------------|---------------|-----------|--|
| 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       | <p>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.</p> <p>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 to the ST_run_geocoding stored procedure. If either corresponding value differs from the default value, the corresponding value will override the default value.</p> |
| DESCRIPTION       | VARCHAR(256)  | Yes       | Description of the parameter indicating its application.   |

### The DB2GSE.ST\_GEOCODERS catalog view

When you enable a database for spatial operations, the supplied geocoder, DB2GSE\_USA\_GEOCODER, is automatically registered in the DB2 Spatial Extender catalog. 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 a description of columns in this view, see the following table.

For information about geocoders' parameters, query DB2 Spatial Extender's DB2GSE.ST\_GEOCODER\_PARAMETERS catalog view and DB2's SYSCAT.ROUTINEPARMS catalog view. For information about functions that are used as geocoders, query DB2's SYSCAT.ROUTINES catalog view.

## DB2GSE.ST\_GEOCODERS catalog view

Table 17. 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.<br><br>The combined values of FUNCTION_SCHEMA and SPECIFIC_NAME uniquely identify the function that is being used as this geocoder. |
| RETURN_TYPE_SCHEMA | VARCHAR(128) | No        | Name of the schema to which the data type of this geocoder's output parameter belongs. This name is obtained from the DB2 catalog.   |
| RETURN_TYPE_NAME   | VARCHAR(128) | No        | Unqualified name of the data type of this geocoder's output parameter. This name is obtained from the DB2 catalog.   |
| VENDOR             | VARCHAR(256) | Yes       | Name of the vendor that created this geocoder.   |
| DESCRIPTION        | VARCHAR(256) | Yes       | Description of the geocoder that indicates its application.  |

## The 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 the following table, 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, set-ups for the supplied geocoder, DB2GSE\_USA\_GEOCODER, include 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 18. 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.<br><br>The combined values in the TABLE_SCHEMA, TABLE_NAME, and COLUMN_NAME columns uniquely identify the spatial column.                            |
| GEOCODER_NAME  | VARCHAR(128)   | No        | 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:<br><b>BATCH</b> Only batch geocoding is enabled.<br><b>AUTO</b> Automatic geocoding is set up and activated.<br><b>INVALID</b> 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.  |

---

**The DB2GSE.ST\_GEOCODING\_PARAMETERS catalog view**

When you set up geocoding operations for a particular geocoder, geocoder-specific aspects of the settings are automatically recorded in the Spatial Extender catalog. For example, an operation specific to the supplied geocoder, DB2GSE\_USA\_GEOCODER, is to compare addresses given as input to reference data, and to geocode the former if they match the latter to a specified degree, or to a degree higher than the specified one. When you set up operations for this geocoder, you specify what this 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 19. 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.<br><br>The combined values in the TABLE_SCHEMA, TABLE_NAME, and COLUMN_NAME columns uniquely identify this spatial column. |

## DB2GSE.ST\_GEOCODING\_PARAMETERS catalog view

Table 19. Columns in the DB2GSE.ST\_GEOCODING\_PARAMETERS catalog view (continued)

| Name            | Data type     | Nullable? | Content  |
|-----------------|---------------|-----------|--|
| ORDINAL         | SMALLINT      | No        | <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 DB2's 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

---

### The DB2GSE.ST\_SIZINGS catalog view

Use the DB2GSE.ST\_SIZINGS catalog view to retrieve:

- 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 the view, see the following table.

Table 20. 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:<br><br><b>A numeric value other than 0</b><br>The allowable maximum length of values assigned to this variable.<br><br><b>0</b><br>Either any length is allowed, or the allowable length cannot be determined.<br><br><b>NULL</b><br>Spatial Extender does not support this variable. |
| DESCRIPTION     | VARCHAR(128) | Yes       | Description of this variable.   |

---

### The DB2GSE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view

When you enable a database for spatial operations, two default spatial reference systems—one to use when you do not specify what coordinate system your spatial data should derive from, and one to use with the GCS\_NORTH\_AMERICAN\_1983 coordinate system—are automatically registered in the Spatial Extender catalog. When users create additional spatial reference systems, these systems are also automatically registered in the catalog. To retrieve information about registered spatial reference systems, query the DB2GSE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view.



## DB2GSE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view

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 21. 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        | Numerical identifier of the spatial reference system. Each spatial reference system has a unique numerical identifier.<br><br>Spatial functions reference spatial reference systems by their numerical identifiers rather than by their names.  |
| X_OFFSET | DOUBLE       | No        | 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. |

## DB2GSE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view

Table 21. Columns in the DB2GSE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view (continued)

| Name     | Data type | Nullable? | Content   |
|----------|-----------|-----------|---|
| 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.  |
| 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.   |

## DB2GSE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view

Table 21. Columns in the DB2GSE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view (continued)

| Name                     | Data type     | Nullable? | Content  |
|--------------------------|---------------|-----------|--|
| 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_COORSYS_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

---

### The DB2GSE.ST\_UNITS\_OF\_MEASURE catalog view

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. To find out what units of measure you can choose from, consult the DB2GSE.ST\_UNITS\_OF\_MEASURE catalog view.

Table 22. 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:<br><b>LINEAR</b><br>The unit of measure is linear.<br><b>ANGULAR</b><br>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.<br><br>The base unit itself has a conversion factor of 1.0. |
| DESCRIPTION       | VARCHAR(256) | Yes       | Description of the unit of measure.  |

---

## Chapter 18. Spatial functions: categories and uses

This chapter introduces all the spatial functions, organizing them by category:

- Functions that convert geometries to and from data exchange formats
- Functions that make comparisons
- Functions that return information about properties of geometries
- Functions that derive new geometries from existing ones
- Miscellaneous functions

---

### Spatial functions that convert geometries to and from data exchange formats

DB2 Spatial Extender provides spatial functions that convert geometries to and from the following data exchange formats:

- 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*. The next section describes these functions in general. Each successive section focuses on the constructor functions for one of the data exchange formats.

#### Constructor functions in general

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:

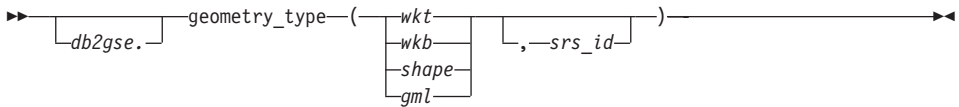
- The SQL for calling functions that operate on data exchange formats, and the type of geometry returned by these functions
- The SQL for calling a function that creates points from X and Y. coordinates, and the type of geometry returned by this function
- Examples of code and result sets

#### Functions that operate on data exchange formats

This section provides the syntax for calling functions that operate on data exchange formats, describes the functions' input parameters, and identifies the type of geometry that these functions return.

## Spatial functions

### Syntax:



### Parameters and other elements of syntax:

*db2gse* Name of the schema to which the spatial data types supplied by DB2 Spatial Extender belong.

*geometry\_type*

One of the following constructor functions:

- ST\_Point
- ST\_LineString
- ST\_Polygon
- ST\_MultiPoint
- ST\_MultiLineString
- ST\_MultiPolygon
- ST\_GeomCollection
- ST\_Geometry

*wkt* A value of type CLOB(2G) that contains the well-known text representation of the geometry.

*wkb* A value of type BLOB(2G) that contains the well-known binary representation of the geometry.

*shape* A value of type BLOB(2G) that contains the ESRI shape representation of the geometry.

*gml* A value of type CLOB(2G) that contains the Geography Markup Language (GML) representation of the 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, then the spatial reference system with the numeric identifier 0 (zero) is used.

### Return Type:

*geometry\_type*

If *geometry\_type* is *ST\_Geometry*, the dynamic type of the returned geometry type corresponds to the geometry indicated by the input value.

If *geometry\_type* is any other type, the dynamic type of the returned geometry type corresponds to the function name. If the geometry indicated by the input value does not match the function name or the name of one of its subtypes, an error is returned.

### A function that creates geometries from coordinates

The `ST_Point` function creates geometries not only from data exchange formats but also from numeric coordinate values—a very useful capability if your location data is already stored in your database. This section provides the syntax for calling `ST_Point`, an explanation of its parameters, and information about the type of geometry that it returns.

#### Syntax:

```
db2gse.ST_Point(—| coordinates | [,—srs_id—])
```

#### coordinates:

```
|—x_coordinate—,—y_coordinate—| [,—z_coordinate—] [,—m_coordinate—]
```

#### Parameters:

##### *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.

##### *srs\_id*

A value of type `INTEGER` that identifies the spatial reference system for the resulting point.

## Spatial functions

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

### Examples

This section provides examples of code for invoking constructors, code for creating tables to contain constructors' output, code for retrieving the output, and the output itself.

The following example will insert 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 using the 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 we know the spatial column may only contain ST\_Point values, we could use the following example which inserts two points. Attempting to insert a linestring or any other type which is not a point would result 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"
```

Most often, data will be inserted by an application program reading from a file or web interface.

The following example uses embedded SQL and assumes that the application fills the data areas with the appropriate values. For a reference to further details on the use of embedded SQL, see **Related reference** at the end of this section.

```
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 example Java code uses JDBC to insert point geometries using X, Y numeric coordinate values and using WKT to specify the geometries. For a reference to further details on using the JDBC interface, see **Related reference** at the end of this section.

```
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),
```

## Spatial functions

```
        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();
```

### Well-known text (WKT) representation

Text representations are CLOB values representing character ASCII strings. They permit geometries to be exchanged in ASCII text form.

The function **ST\_AsText** is provided to convert 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 ( 30.00000000 40.00000000)
 200   LINESTRING ( 50.00000000 50.00000000, 100.00000000 100.00000000)
```

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, the **ST\_WellKnownText** transform group can be used to implicitly convert geometries to their well-known text representation when binding them out. The following example code illustrates 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;
```

Note that no spatial function is used in the `SELECT` statement to convert the geometry. For a reference to further details on the use of transform groups, see **Related reference** at the end of this section.

In addition to the functions discussed 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 conform to the OGC “Simple Features For SQL” specification and the ISO SQL/MM Part 3: Spatial standard. These other functions are:

- **ST\_WKTToSQL**
- **ST\_GeomFromText**
- **ST\_GeomCollFromTxt**
- **ST\_PointFromText**
- **ST\_LineFromText**
- **ST\_PolyFromText**
- **ST\_MPointFromText**
- **ST\_MLineFromText**
- **ST\_MPolyFromText**

### Well-known binary (WKB) representation

The WKB representation consists of binary data structures that must be BLOB values representing 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 function **ST\_AsBinary** is provided to convert 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;
```

## Spatial functions

Alternatively, the `ST_WellKnownBinary` transform group can be used to implicitly convert geometries to their well-known binary representation when binding them out. The following example code illustrates how to use the 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;
```

Note that no spatial function is used in the `SELECT` statement to convert the geometry. For a reference to further details on the use of transform groups, see **Related reference** at the end of this section.

In addition to the functions discussed 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 conform to the OGC “Simple Features For SQL” specification and the ISO SQL/MM Part 3: Spatial standard. These other functions are:

- **ST\_WKBToSQL**
- **ST\_GeomFromWKB**
- **ST\_GeomCollFromWKB**
- **ST\_PointFromWKB**
- **ST\_LineFromWKB**
- **ST\_PolyFromWKB**
- **ST\_MPointFromWKB**
- **ST\_MLineFromWKB**
- **ST\_MPolyFromWKB**

### ESRI shape representation

The ESRI Shape representation consists of binary data structures that must be managed by an application program written in a supported language.

The function **ST\_AsShape** is provided to convert 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, the ST\_Shape transform group can be used to implicitly convert geometries to their shape representation when binding them out. The following example code illustrates 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;
```

Note that no spatial function is used in the SELECT statement to convert the geometry.

### Geography Markup Language (GML) representation

GML representations are ASCII strings. They permit geometries to be exchanged in ASCII text form.

The function **ST\_AsGML** is provided to convert 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 following example code illustrates how to use the transform group. In the following example, the results have been reformatted for readability. The spacing in your results will 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">
```

## Spatial functions

```
<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, the ST\_GML transform group can be used 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> |

Note that no spatial function is used in the select statement to convert the geometry.

### Related reference:

- “Transform groups” on page 489

---

## Functions that make comparisons

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. This section refers to these functions as *comparison functions*. It discusses their general nature, describes them individually, and points out differences between functions that are similar.

### Comparison functions in general

DB2 Spatial Extender’s 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, or null 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 interior, boundary and exterior, with consideration for the dimension of the resulting intersections.

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

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

|                 | <b>Interior</b>        | <b>Boundary</b>        | <b>Exterior</b>        |
|-----------------|------------------------|------------------------|------------------------|
| <b>Interior</b> | $\dim(I(a) \cap I(b))$ | $\dim(I(a) \cap B(b))$ | $\dim(I(a) \cap E(b))$ |
| <b>Boundary</b> | $\dim(B(a) \cap I(b))$ | $\dim(B(a) \cap B(b))$ | $\dim(B(a) \cap E(b))$ |
| <b>Exterior</b> | $\dim(E(a) \cap I(b))$ | $\dim(E(a) \cap B(b))$ | $\dim(E(a) \cap E(b))$ |

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.

## Spatial functions

For example, the following pattern matrix for the ST\_Within function includes the values T, F, and \*.

*Table 23. Matrix for ST\_Within.* The pattern matrix of the ST\_Within function for geometry combinations.

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

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. DE-9IM is collaboration of four authors, Clementini, Eliseo, Di Felice, and van Osstrom. They published the model in "A Small Set of Formal Topological Relationships Suitable for End-User Interaction," D. Abel and B.C. Ooi (Ed.), *Advances in Spatial Database—Third International Symposium. SSD '93*. LNCS 692. Pp. 277-295. The 9 Intersection model by 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*.

### Descriptions of functions

The comparison functions are:

- ST\_Contains
- ST\_Crosses
- ST\_Disjoint
- ST\_EnvIntersects
- ST\_EqualCoordsys
- ST\_Equals
- ST\_EqualSRS
- ST\_Intersects
- ST\_MBRIntersects
- ST\_Overlaps
- ST\_Relate



- ST\_Touches
- ST\_Within

### ST\_Contains

ST\_Contains returns a value of 1 (one) if the second geometry is completely contained by the first geometry. The ST\_Contains function returns the exact opposite result of the ST\_Within function.



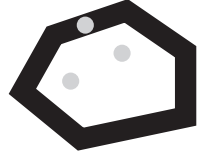






|   |   |  |
|---|---|--|
|    |    |    |
| multipoint / point  | multipoint / multipoint   | polygon / multipoint   |
|    |    |    |
| linestring / point  | linestring / multipoint   | linestring / linestring  |
|  |  |  |
| polygon / point   | polygon / linestring  | polygon / polygon  |

Figure 15. ST\_Contains. The dark geometries represent geometry a and the gray geometries represent geometry b. In all cases, geometry a contains geometry b completely.

The pattern matrix of the ST\_Contains function states that the interiors of both geometries must intersect and that the interior and boundary of the secondary (geometry b) must not intersect the exterior of the primary (geometry a).

## Spatial functions

Table 24. Matrix for ST\_Contains

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

### ST\_Crosses

ST\_Crosses takes two geometries and returns a value of 1 (one) if:

- The intersection results in a geometry whose dimension is less than the maximum dimension of the source geometries.
- The intersection set is interior to both source geometries.

ST\_Crosses returns a null if the first geometry is a surface or multisurface or if the second geometry is a point or multipoint. For all other combinations, ST\_Crosses returns either a value of 1 (one: yes, the two geometries cross) or a value of 0 (zero: no, they do not cross).

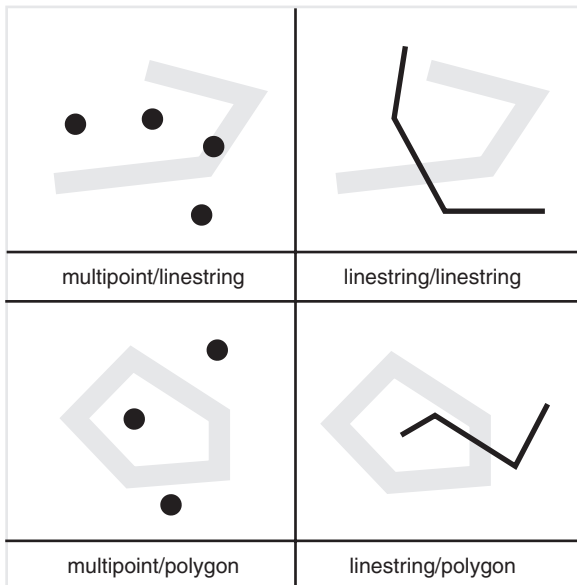


Figure 16. ST\_Crosses. The dark geometries represent geometry *a*; the gray geometries represent geometry *b*. In all cases, geometry *b* crosses geometry *a*.

The pattern matrix in Table 25 on page 257 applies if the first geometry is a point or multipoint, or if the first geometry is a curve or multicurve, and the second geometry is a surface. The matrix states that the interiors must intersect and that the interior of the primary (geometry *a*) must intersect the exterior of the secondary (geometry *b*).

Table 25. Matrix for ST\_Crosses (1)

|   |          | b        |          |          |
|---|----------|----------|----------|----------|
|   |          | Interior | Boundary | Exterior |
| a | Interior | T        | *        | T        |
|   | Boundary | *        | *        | *        |
|   | Exterior | *        | *        | *        |

The pattern matrix in Table 26 applies if the first and second geometries are both curves or multicurves. The matrix states that the dimension of the intersection of the interiors must be 0 (intersect at a point). If the dimension of this intersection is 1 (intersect at a linestring), the ST\_Crosses function returns a value of 0 (no, the geometries do not cross); however, the ST\_Overlaps function returns a value of 1 (yes, the geometries overlap).

Table 26. Matrix for ST\_Crosses (2)

|   |          | b        |          |          |
|---|----------|----------|----------|----------|
|   |          | Interior | Boundary | Exterior |
| a | Interior | 0        | *        | *        |
|   | Boundary | *        | *        | *        |
|   | Exterior | *        | *        | *        |

### ST\_Disjoint

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.

## Spatial functions

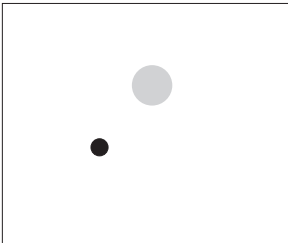
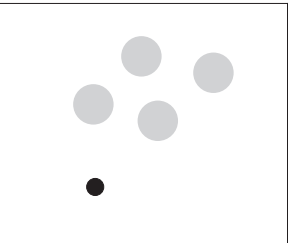
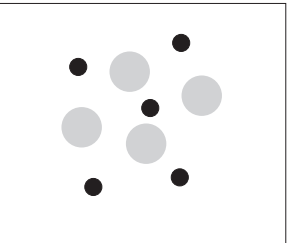
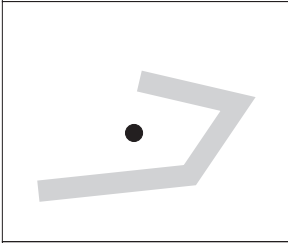
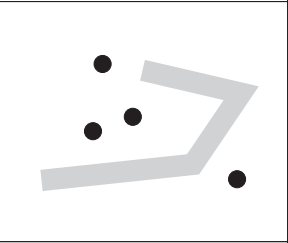
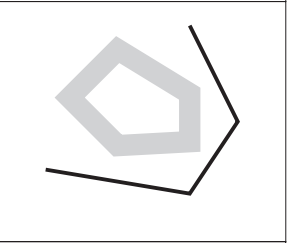
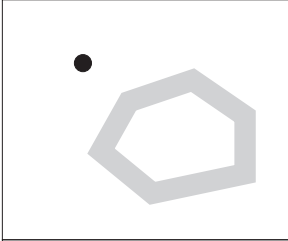
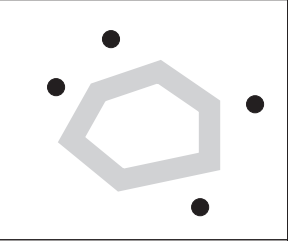
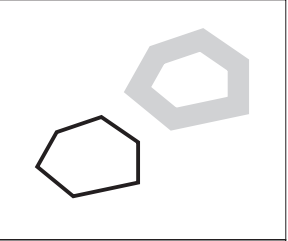
|   |   |  |
|---|---|--|
|  |  |  |
| point / point   | point / multipoint  | multipoint / multipoint  |
|  |  |  |
| point / linestring  | multistring / linestring  | polygon / linestring   |
|  |  |  |
| point / polygon   | multipoint / multipolygon   | polygon / polygon  |

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.

Table 27. Matrix for *ST\_Disjoint*. This matrix simply states that neither the interiors nor the boundaries of either geometry intersect.

|   |          | b        |          |          |
|---|----------|----------|----------|----------|
|   |          | Interior | Boundary | Exterior |
| a | Interior | F        | F        | *        |
|   | Boundary | F        | F        | *        |
|   | Exterior | *        | *        | *        |

### **ST\_EnvIntersects**

*ST\_EnvIntersect* returns a value of 1 (one) if the envelopes of two geometries intersect. It is a convenience function that efficiently implements *ST\_Intersects* (*ST\_Envelope*(g1), *ST\_Envelope*(g2)).

**ST\_EqualCoordsys**

ST\_EqualCoordsys returns a value of 1 (one) if two coordinate system definitions are identical. In comparing the definitions, ST\_EqualCoordsys disregards differences in case, spaces, parentheses, and representation of floating point numbers.

**ST\_Equals**

ST\_Equals returns a value of 1 (one) if two geometries are identical. The order of the points used to define the geometries is not relevant to the test of equality.






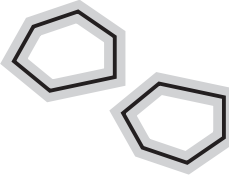
|   |   |
|---|---|
|    |    |
| <p>point / point</p>  | <p>multipoint / multipoint</p>  |
|    |    |
| <p>linestring /linestring</p>   | <p>multistring /multistring</p>   |
|  |  |
| <p>polygon / polygon</p>  | <p>multipolygon / multipolygon</p>  |

Figure 18. ST\_Equals. The dark geometries represent geometry a; the gray geometries represent geometry b. In all cases, geometry a is equal to geometry b.

## Spatial functions

*Table 28. Matrix for equality.* The DE-9IM pattern matrix for equality ensures that the interiors intersect and that no part interior or boundary of either geometry intersects the exterior of the other.

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

### ST\_EqualSRS

ST\_EqualSRS returns a value of 1 (one) if two spatial reference systems are identical, provided that the numeric identifier of either or both systems is not null.

### ST\_Intersects

ST\_Intersects returns a value of 1 (one) if the intersection does not result in an empty set. 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 29. Matrix for ST\_Intersects (1).* The ST\_Intersects function returns 1 (one) if the interiors of both geometries intersect.

|   |          | b        |          |          |
|---|----------|----------|----------|----------|
|   |          | Interior | Boundary | Exterior |
| a | Interior | T        | *        | *        |
|   | Boundary | *        | *        | *        |
|   | Exterior | *        | *        | *        |

*Table 30. 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.

|   |          | b        |          |          |
|---|----------|----------|----------|----------|
|   |          | Interior | Boundary | Exterior |
| a | Interior | *        | T        | *        |
|   | Boundary | *        | *        | *        |
|   | Exterior | *        | *        | *        |

*Table 31. 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.

|   |          | b        |          |          |
|---|----------|----------|----------|----------|
|   |          | Interior | Boundary | Exterior |
| a | Interior | *        | *        | *        |
|   | Boundary | T        | *        | *        |
|   | Exterior | *        | *        | *        |

Table 32. Matrix for ST\_Intersects (4). The ST\_Intersects function returns 1 (one) if the boundaries of either geometry intersect.

|   |          | b        |          |          |
|---|----------|----------|----------|----------|
|   |          | Interior | Boundary | Exterior |
| a | Interior | *        | *        | *        |
|   | Boundary | *        | T        | *        |
|   | Exterior | *        | *        | *        |

**ST\_MBRIntersects**

ST\_MBRIntersects returns a value of 1 (one) if the minimum bounding rectangles (MBRs) of two geometries intersect.

**ST\_Overlaps**

ST\_Overlaps compares two geometries of the same dimension. It returns a value of 1 (one) if their intersection set results in a geometry different from both, but that has the same dimension.

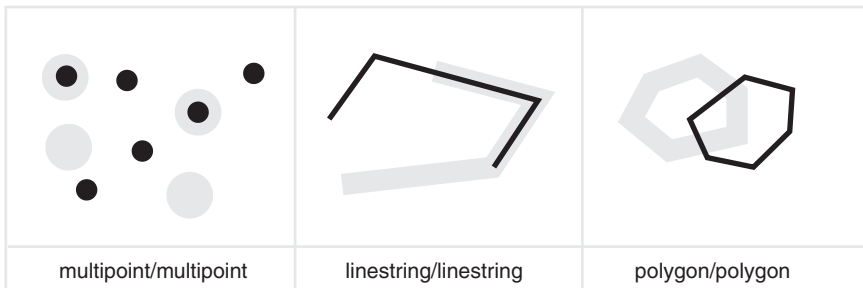


Figure 19. ST\_Overlaps. The dark geometries represent geometry a; the gray geometries represent geometry b. In all cases, both geometries have the same dimension, and one overlaps the other partway. The area of overlap is a new geometry; it has the same dimension as geometries a and b.

The pattern matrix in Table 33 applies if the first and second geometries are both either points, multipoints, surfaces, or multisurfaces. ST\_Overlaps returns a value of 1 if the interior of each geometry intersects the other geometry’s interior and exterior.

Table 33. Matrix for ST\_Overlaps (1)

|   |          | b        |          |          |
|---|----------|----------|----------|----------|
|   |          | Interior | Boundary | Exterior |
| a | Interior | T        | *        | T        |
|   | Boundary | *        | *        | *        |
|   | Exterior | T        | *        | *        |

The pattern matrix in Table 34 on page 262 applies if the first and second geometries are both curves or multicurves. In this case the intersection of the

## Spatial functions

geometries must result in a geometry that has a dimension of 1 (another curve). If the dimension of the intersection of the interiors is 1, `ST_Overlaps` returns a value of 0 (no, the geometries do not overlap); however the `ST_Crosses` function would return a value of 1 (yes, the geometries cross).

Table 34. Matrix for `ST_Overlaps` (2)

|   |          | b        |          |          |
|---|----------|----------|----------|----------|
|   |          | Interior | Boundary | Exterior |
| a | Interior | 1        | *        | T        |
|   | Boundary | *        | *        | *        |
|   | Exterior | T        | *        | *        |

### ST\_Relate

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).

### ST\_Touches

`ST_Touches` returns a value of 1 (one) if all the points common to both geometries can be found only on the boundaries. The interiors of the geometries must not intersect one another. At least one geometry must be a curve, surface, multicurve, or multisurface.

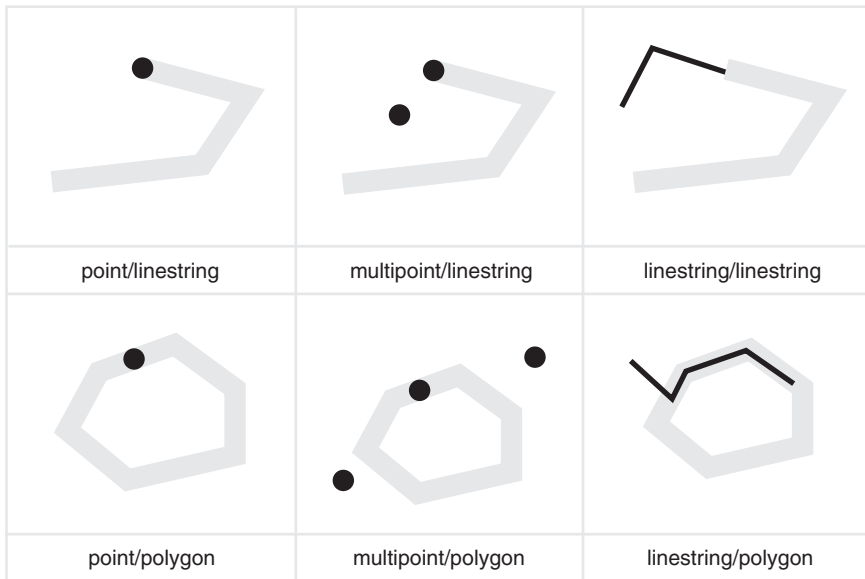


Figure 20. `ST_Touches`. The dark geometries represent geometry *a*; the gray geometries represent geometry *b*. In all cases, the boundary of geometry *b* intersects geometry *a*. The interior of geometry *b* remains separate from geometry *a*.



The pattern matrices show that the `ST_Touches` function returns 1 (one) when the interiors of the geometry do not intersect, and the boundary of either geometry intersects the other's interior or its boundary.

Table 35. Matrix for `ST_Touches` (1)

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

Table 36. Matrix for `ST_Touches` (2)

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

Table 37. Matrix for `ST_Touches` (3)

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

### **ST\_Within**

`ST_Within` returns a value of 1 (one) if the first geometry is completely within the second geometry. `ST_Within` returns the exact opposite result of `ST_Contains`.

## Spatial functions

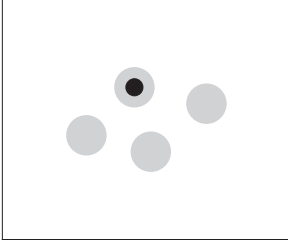
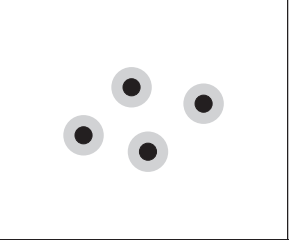
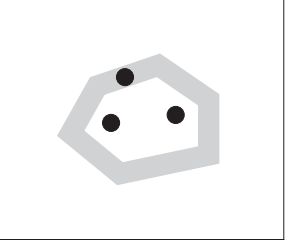
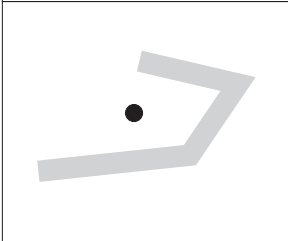
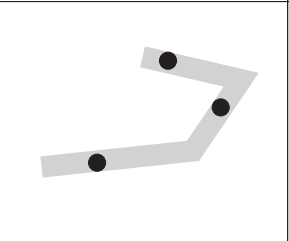

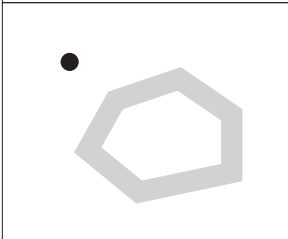
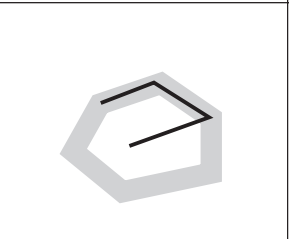
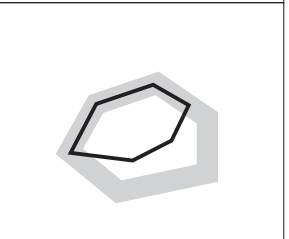
|   |   |  |
|---|---|--|
|  |  |  |
| point / multipoint  | multipoint / multipoint   | multipoint / polygon   |
|  |  |  |
| point / linestring  | multipoint / linestring   | linestring / linestring  |
|  |  |  |
| point / polygon   | linestring / polygon  | polygon / polygon  |

Figure 21. *ST\_Within*

The *ST\_Within* function pattern matrix states that the interiors of both geometries must intersect, and that the interior and boundary of the primary geometry (geometry *a*) must not intersect the exterior of the secondary (geometry *b*).

Table 38. Matrix for *ST\_Within*

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

### Differences between similar functions

This section will briefly describe the differences between:

- *ST\_Contains* and *ST\_Within*

- `ST_Intersects`, `ST_Crosses` and `ST_Overlaps`
- `ST_EnvIntersects` and `ST_MBRIntersects`

`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; in other words, whether the first contains the second. `ST_Within` determines whether the first geometry is completely inside the second; that is, whether the first is within the second.

`ST_Intersects`, `ST_Crosses`, `ST_Overlaps`, and `ST_Touches` all determine whether one geometry intersects another. They 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 partway, 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.

`ST_EnvIntersects` and `ST_MBRIntersects` are similar in that they 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.

## Spatial functions

More recently, the smallest rectangular area into which a geometry can fit has been 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 return information about properties of geometries

This section introduces spatial functions that return information about properties of geometries. This information 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

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 another category. For information about this other category, which is called “Spatial functions that generate new geometries”, see the appropriate link or cross-reference at the end of this section.

#### **ST\_GeometryType: A function for information about data types**

`ST_GeometryType` takes a geometry as input parameter and returns the fully qualified type name of the dynamic type of that geometry.

#### **Functions for information about coordinates and measures**

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

### **ST\_CoordDim**

ST\_CoordDim returns a value that denotes what types of coordinates a geometry has, and whether the geometry also contains any measures. This value is called a *coordinate dimension*. (Note that a coordinate dimension is not the same thing as the property referred to as *dimension*. The latter indicates whether a geometry has breadth or length, not whether it contains coordinates of a specific type or measures.)

### **ST\_IsMeasured**

ST\_IsMeasured takes a geometry as an input parameter and returns 1 if the given geometry has M coordinates (measures). Otherwise 0 (zero) is returned.

### **ST\_IsValid**

ST\_IsValid 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.

### **ST\_Is3D**

ST\_Is3d takes a geometry as an input parameter and returns 1 if the given geometry has Z coordinates. Otherwise, 0 (zero) is returned.

### **ST\_M**

If a measure is stored with a given point, ST\_M can take the point as an input parameter and return the measure.

## Spatial functions

### **ST\_MaxM**

ST\_MaxM takes a geometry as an input parameter and returns its maximum measure.

### **ST\_MaxX**

ST\_MaxX takes a geometry as an input parameter and returns its maximum X coordinate.

### **ST\_MaxY**

ST\_MaxY takes a geometry as an input parameter and returns its maximum Y coordinate.

### **ST\_MaxZ**

ST\_MaxZ takes a geometry as an input parameter and returns its minimum Z coordinate.

### **ST\_MinM**

ST\_MinM takes a geometry as an input parameter and returns its minimum measure.

### **ST\_MinX**

ST\_MinX takes a geometry as an input parameter and returns its minimum X coordinate.

### **ST\_MinY**

ST\_MinY takes a geometry as an input parameter and returns its minimum Y coordinate.

### **ST\_MinZ**

ST\_MinZ takes a geometry as an input parameter and returns its minimum Z coordinate.

### **ST\_X**

ST\_X can take point as an input parameter and return the point's X coordinate.

### **ST\_Y**

ST\_Y can take point as an input parameter and return the point's Y coordinate.

### **ST\_Z**

If a Z coordinate is stored with a given point, ST\_Z can take the point as an input parameter and return the Z coordinate.

## **Functions for information about geometries within a geometry**

The following functions return information about geometries within a geometry. For example, 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

### **ST\_Centroid**

ST\_Centroid 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.

### **ST\_EndPoint**

ST\_Endpoint takes a curve as an input parameter and returns the point that is the last point of the curve.

### **ST\_GeometryN**

ST\_GeometryN takes a geometry collection and an index as input parameters and returns the geometry in the collection that is identified by the index.

### **ST\_LineStringN**

ST\_LineStringN takes a multilinestring and an index as input parameters and returns the linestring that is identified by the index.

### **ST\_MidPoint**

ST\_MidPoint 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.

### **ST\_NumGeometries**

ST\_NumGeometries takes a geometry collection as an input parameter and returns the number of geometries in the collection.

### **ST\_NumLineStrings**

ST\_NumLineStrings takes a multilinestring as an input parameter and returns the number of linestrings that it contains.

## Spatial functions

### **ST\_NumPoints**

ST\_NumPoints 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.

### **ST\_NumPolygons**

ST\_NumPolygons takes a multipolygon as an input parameter and returns the number of polygons that it contains.

### **ST\_PointN**

ST\_PointN 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.

### **ST\_PolygonN**

ST\_PolygonN takes a multipolygon and an index as input parameters and returns the polygon that is identified by the index.

### **ST\_StartPoint**

ST\_StartPoint takes a curve as an input parameter and returns the point that is the first point of the curve.

## **Functions for information about rings, boundaries, envelopes, and minimum bounding rectangles**

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

### **ST\_Boundary**

ST\_Boundary takes a geometry as an input parameter and returns its boundary as a new geometry.



### **ST\_Envelope**

ST\_Envelope 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.

### **ST\_EnvIntersects**

ST\_EnvIntersects takes two geometries as input parameters and returns 1 if the envelopes of two geometries intersect. Otherwise, 0 (zero) is returned.

### **ST\_ExteriorRing**

ST\_ExteriorRing takes a polygon as an input parameter and returns its exterior ring as a curve.

### **ST\_InteriorRingN**

ST\_InteriorRingN 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 and not by any geometric orientation.

### **ST\_MBR**

ST\_MBR takes a geometry as an input parameter and returns its minimum bounding rectangle.

### **ST\_MBRIntersects**

ST\_MBRIntersects returns a value of 1 (one) if the minimum bounding rectangles (MBRs) of two geometries intersect.

### **ST\_NumInteriorRing**

ST\_NumInteriorRing takes a polygon as an input parameter and returns the number of its interior rings.

### **ST\_Perimeter**

ST\_Perimeter 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 given units.

## **Functions for information about dimensions**

The following functions return information about the dimension of a geometry. For example, ST\_Area reports how much area a given geometry covers.

These functions are:

- ST\_Area
- ST\_Dimension
- ST\_Length

## Spatial functions

### **ST\_Area**

ST\_Area takes a geometry and, optionally, a unit as input parameters and returns the area covered by the given geometry in the given unit of measure.

### **ST\_Dimension**

ST\_Dimension takes a geometry as an input parameter and returns its dimension.

### **ST\_Length**

ST\_Length takes a curve or multicurve and, optionally, a unit as input parameters and returns the length of the given curve or multicurve in the given unit of measure.

## **Functions for information as to whether a geometry is closed, empty, or simple**

The following functions indicate:

- Whether a given curve or multicurve is closed (that is, whether the start point and end point of the curve or multicurve 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

### **ST\_IsClosed**

ST\_IsClosed 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.

### **ST\_IsEmpty**

ST\_IsEmpty takes a geometry as an input parameter and returns 1 if the given geometry is empty. Otherwise 0 (zero) is returned.

### **ST\_IsSimple**

ST\_IsSimple takes a geometry as an input parameter and returns 1 if the given geometry is simple. Otherwise, 0 (zero) is returned.

## **Functions for information about spatial reference systems**

The following functions return values that identify the spatial reference system that has been associated with the geometry. In addition, the function ST\_SrsID 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

### **ST\_SrsId (also called ST\_SRID)**

ST\_SrsId (or ST\_SRID) 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.

### **ST\_SrsName**

ST\_SrsName takes a geometry as an input parameter and returns the name of the spatial reference system in which the given geometry is represented.

---

## **Spatial functions that generate new 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 for converting geometries into other geometries**

The following functions can convert geometries of a super type 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.

These functions are:

- ST\_Polygon
- ST\_ToGeomColl
- ST\_ToLineString
- ST\_ToMultiLine

## Spatial functions

- ST\_ToMultiPoint
- ST\_ToMultiPolygon
- ST\_ToPoint
- ST\_ToPolygon

### **ST\_Polygon**

ST\_Polygon can construct a polygon from a closed linestring. The linestring will define the exterior ring of the polygon.

### **ST\_ToGeomColl**

ST\_ToGeomColl takes a geometry as an input parameter and converts it to a geometry collection.

### **ST\_ToLineString**

ST\_ToLineString takes a geometry as an input parameter and converts it to a linestring.

### **ST\_ToMultiLine**

ST\_ToMultiLine takes a geometry as an input parameter and converts it to a multilinestring.

### **ST\_ToMultiPoint**

ST\_ToMultiPoint takes a geometry as an input parameter and converts it to a multipoint.

### **ST\_ToMultiPolygon**

ST\_ToMultiPolygon takes a geometry as an input parameter and converts it to a multipolygon.

### **ST\_ToPoint**

ST\_ToPoint takes a geometry as an input parameter and converts it to a point.

### **ST\_ToPolygon**

ST\_ToPolygon takes a geometry as an input parameter and converts it to a polygon.

## **Functions for creating geometries that represent configurations of space**

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
- 

### ST\_Buffer

The ST\_Buffer function can generate a new geometry that extends outward from an existing geometry by a specified radius. The new geometry will be a surface when the existing geometry is buffered or whenever the elements of a collection are so close that the buffers around the single elements of the collection overlap. However, when the buffers are separate, individual buffer surfaces result, in which case ST\_Buffer returns a multisurface.

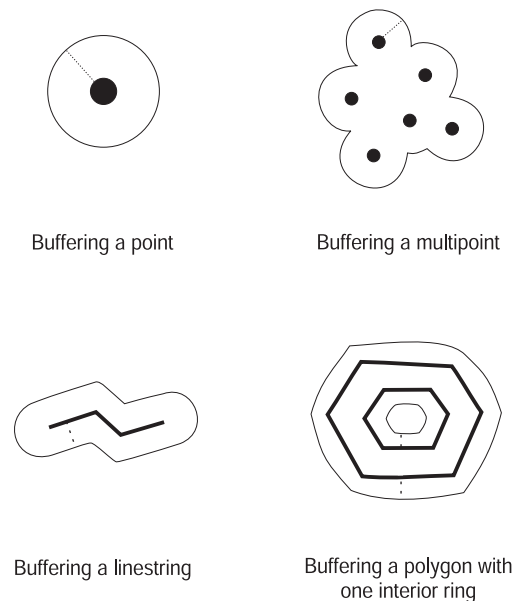


Figure 22. ST\_Buffer

The ST\_Buffer function accepts both positive and negative distance; however, only geometries with a dimension of two (surfaces and multisurfaces) apply a negative buffer. The absolute value of the buffer distance is used whenever the dimension of the source geometry is less than 2 (all geometries that are not surfaces or multisurfaces).

In general, for exterior rings, positive buffer distances generate surface rings that are away from the center of the source geometry; negative buffer distances generate surface or multisurface rings toward the center. For interior

## Spatial functions

rings of a surface or multisurface, a positive buffer distance generates a buffer ring toward the center, and a negative buffer distance generates a buffer ring away from the center.

The buffering process merges surfaces that overlap. Negative distances greater than one half the maximum interior width of a polygon result in an empty geometry.

### **ST\_ConvexHull**

The `ST_ConvexHull` function returns the convex hull of any geometry that has at least three vertices forming a convex. (*Vertices* are the pairs of X and Y coordinates within geometries. A *convex hull* is the smallest convex polygon that can be formed by all vertices within a given set of vertices.)

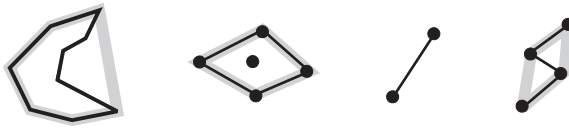


Figure 23. `ST_ConvexHull`

### **ST\_Difference**

`ST_Difference` takes two geometries of the same dimension as input. The `ST_Difference` function returns that portion of the first geometry that is not intersected by the second geometry. This operation is the spatial equivalent of the logical AND NOT. The portion of geometry returned by `ST_Difference` is itself a geometry—a collection that has the same dimension as the geometries taken as input. If these two geometries are equal—that is, if they occupy the same space—the returned geometry is empty.

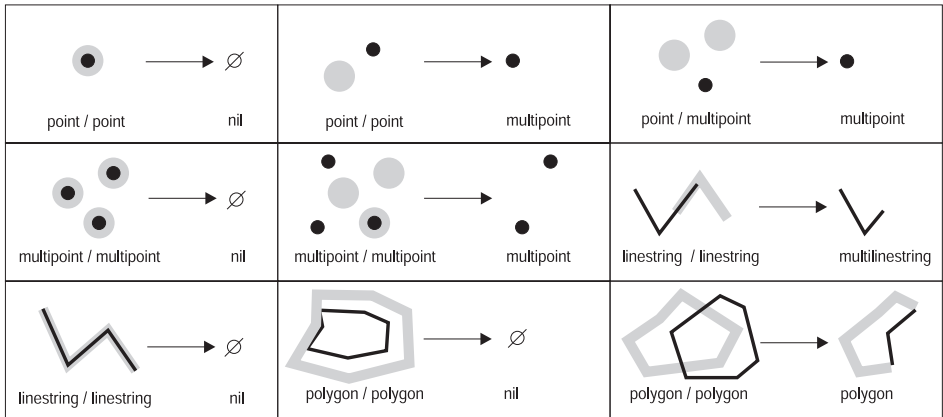
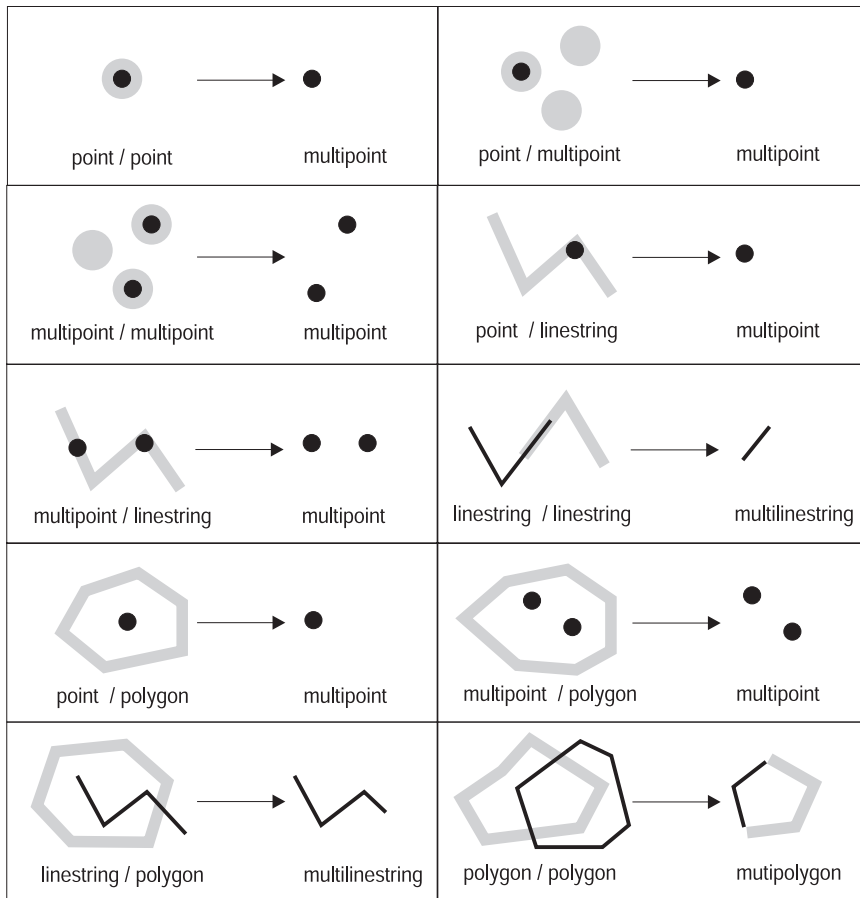


Figure 24. *ST\_Difference*. To the left of each arrow are two geometries that are given to *ST\_Difference* as input. To the right of each arrow is the output of *ST\_Difference*. If part of the first geometry is intersected by the second, the output is that part of the first geometry that is not intersected. If the geometries given as input are equal, the output is an empty geometry (denoted by the term *nil*).

### ST\_Intersection

The *ST\_Intersection* function returns a set of points, represented as a geometry, which define the intersection of two given geometries. If the geometries given to *ST\_Intersection* as input do not intersect, or if they do, and the dimension of their intersection is less than the geometries' dimensions, *ST\_Intersection* returns an empty geometry.

## Spatial functions



*Figure 25. ST\_Intersection.* To the left of each arrow are two intersecting geometries that are given to `ST_Intersection` as input. To the right of each arrow is the output of `ST_Intersection`—a geometry that represents the intersection created by the geometries at the left.

### **ST\_SymDifference**

The `ST_SymDifference` function returns the symmetric difference (the spatial equivalent of the logical XOR operation) of two intersecting geometries that have the same dimension. If these geometries are equal, `ST_SymDifference` returns an empty geometry. If they are not equal, then a portion of one or both of them will lie outside the area of intersection.

### **Functions for deriving individual geometries from multiple geometries**

The following functions derive individual geometries from multiple geometries. For example, `ST_Union` combines two geometries into a single geometry.



These functions are:

- MBR Aggregate
- ST\_Union
- Union Aggregate

## MBR Aggregate

The combination of the functions ST\_BuildMBRAggr and ST\_GetAggrResult aggregates a column of geometries in a selected column to a single geometry by constructing 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.

## ST\_Union

The ST\_Union function returns the union set of two geometries. This operation is the spatial equivalent of the logical OR. The two geometries must be of the same dimension. ST\_Union always returns the result as a collection.

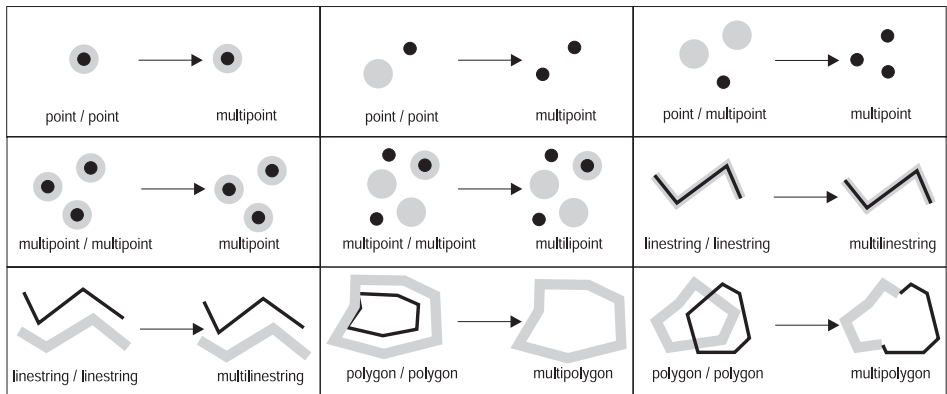


Figure 26. ST\_Union

## Union Aggregate

A union aggregate is the combination of the ST\_BuildUnionAggr and ST\_GetAggrResult functions. This combination aggregates a column of geometries in a table to single geometry by constructing the union.

## Functions for creating geometries based on measures

The following functions can create geometries whose points are associated with a specific measure or with a specific sequence of two measures. For example, suppose that measures ranging from a value of 4 to a value of 8 are stored with the points in a multicurve. If you want to know with which points a measure with a value of 7 is stored, you could use the FindMeasure function to return those points within a single multipoint.

These functions are:

## Spatial functions

- ST\_FindMeasure (also called ST\_LocateAlong)
- ST\_MeasureBetween (also called ST\_LocateBetween)

### ST\_FindMeasure or ST\_LocateAlong

ST\_FindMeasure or ST\_LocateAlong 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.

### ST\_MeasureBetween (also called ST\_LocateBetween)

ST\_MeasureBetween or ST\_LocateBetween takes 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.

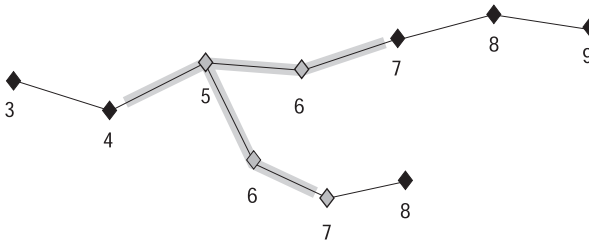


Figure 27. *LocateBetween*

## Functions for creating modified forms of geometries

The following functions create modified forms of existing geometries. For example, the ST\_AppendPoint function creates 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`

### **ST\_AppendPoint**

`ST_AppendPoint` takes a curve and a point as input parameters and extends the curve by the given point.

### **ST\_ChangePoint**

`ST_ChangePoint` 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.

### **ST\_Generalize**

`ST_Generalize` 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-Decker 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.

### **ST\_M**

If a given point is not associated with a measure, `ST_M` can provide a measure to be stored with the point. If the point has an associated measure, `ST_M` can replace this measure with another one.

### **ST\_PerpPoints**

`ST_PerpPoints` 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.

### **ST\_RemovePoint**

`ST_RemovePoint` 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.

### **ST\_X**

`ST_X` can replace a point's X coordinate with another X coordinate.

### **ST\_Y**

`ST_Y` can replace a point's Y coordinate with another Y coordinate.

## Spatial functions

### ST\_Z

If a given point has no Z coordinate, ST\_Z can add a Z coordinate to the point. If the point does have a Z coordinate, ST\_Z can replace this coordinate with another Z coordinate.

---

## Miscellaneous spatial functions

Three spatial functions do not fit well in the categories that the other spatial functions belong to. They do not convert geometries' formats, compare geometries, return information about geometries' properties, or create geometries. They are:

### ST\_Distance:

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 28 illustrates this information.



*Figure 28. 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.*

### ST\_GetIndexParams:

`ST_GetIndexParms` 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.

### **ST\_Transform:**

`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.

## Reference material

---

## Chapter 19. Spatial functions: syntax and parameters

This section introduces the spatial functions described in the following sections. It discusses certain factors that are common to all or most spatial functions. The functions are documented here in alphabetical order.

---

### Spatial functions: considerations and associated data types

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

#### Factors to consider

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

## Considerations for spatial functions

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
```

### 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® attempts to pass geometries returned by `ST_Union` to `ST_PointOnSurface`, DB2 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 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 raises a run-time exception.

If a function returns a geometry of a super type, the `TREAT` operator generally can tell DB2 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 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 listed according to input type

Table 39 on page 288 lists the 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 39 on page 288 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.

## Considerations for spatial functions

Table 39. Spatial functions listed according to input type

| Data type of input parameter | Function   |
|------------------------------|--|
| ST_Geometry                  | ST_Area<br>ST_AsBinary<br>ST_AsGML<br>ST_AsShape<br>ST_AsText<br>ST_Boundary<br>ST_Buffer<br>ST_BuildMBrAggr<br>ST_BuildUnionAggr<br>ST_Centroid<br>ST_Contains<br>ST_ConvexHull<br>ST_CoordDim<br>ST_Crosses<br>ST_Difference<br>ST_Dimension<br>ST_Disjoint<br>ST_Distance<br>ST_Envelope<br>ST_EnvIntersects<br>ST_Equals<br>ST_FindMeasure or ST_LocateAlong<br>ST_Generalize<br>ST_GeometryType |

## Considerations for spatial functions

Table 39. Spatial functions listed according to input type (continued)

| Data type of input parameter | Function   |
|------------------------------|--|
| ST_Geometry, continued       | ST_Intersection<br>ST_Intersects<br>ST_Is3D<br>ST_IsEmpty<br>ST_IsMeasured<br>ST_IsSimple<br>ST_IsValid<br>ST_MaxM<br>ST_MaxX<br>ST_MaxY<br>ST_MaxZ<br>ST_MBR<br>ST_MBRIntersects<br>ST_MeasureBetween or ST_LocateBetween<br>ST_MinM<br>ST_MinX<br>ST_MinY<br>ST_MinZ<br>ST_NumPoints<br>ST_Overlaps<br>ST_Relate<br>ST_SRID or ST_SrsId<br>ST_SrsName<br>ST_SymDifference<br>ST_SymmetricDiff<br>ST_ToGeomColl<br>ST_ToLineString<br>ST_ToMultiLine<br>ST_ToMultiPoint<br>ST_ToMultiPolygon<br>ST_ToPoint<br>ST_ToPolygon<br>ST_Touches<br>ST_Transform<br>ST_Union<br>ST_Within |
| ST_Point                     | ST_M<br>ST_X<br>ST_Y<br>ST_Z   |

## Considerations for spatial functions

Table 39. Spatial functions listed according to input type (continued)

| Data type of input parameter | Function  |
|------------------------------|---|
| ST_Curve                     | ST_AppendPoint<br>ST_ChangePoint<br>ST_EndPoint<br>ST_IsClosed<br>ST_IsRing<br>ST_Length<br>ST_MidPoint<br>ST_PerpPoints<br>ST_RemovePoint<br>ST_StartPoint |
| ST_LineString                | ST_PointN<br>ST_Polygon   |
| ST_Surface                   | ST_Perimeter<br>ST_PointOnSurface   |
| ST_GeomCollection            | ST_GeometryN<br>ST_NumGeometries  |
| ST_MultiPoint                | ST_PointN   |
| ST_MultiCurve                | ST_IsClosed<br>ST_Length<br>ST_PerpPoints   |
| ST_MultiLineString           | ST_LineStringN<br>ST_NumLineStrings<br>ST_Polygon   |
| ST_MultiSurface              | ST_Perimeter<br>ST_PointOnSurface   |
| ST_MultiPolygon              | ST_NumPolygons<br>ST_PolygonN   |

### Related reference:

- “ST\_Boundary” on page 302
- “ST\_Area” on page 294
- “ST\_PerpPoints” on page 434
- “ST\_Point” on page 437
- “ST\_PointOnSurface” on page 444
- “ST\_Relate” on page 453
- “ST\_Union” on page 475

## MBR Aggregate

The combination of the functions `ST_BuildMBRAggr` and `ST_GetAggrResult` aggregates a column of geometries in a selected column to a single geometry by constructing 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.

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 an MPP environment.
- If `GROUP BY` clause is used in the full-select.
- If you use a function other than the DB2 aggregate function `MAX`.

### Example:

## MBR Aggregate

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

`ST_AppendPoint` 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 example, 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:

## ST\_AppendPoint

```
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

ST\_Area takes a geometry and, optionally, a unit as input parameters and returns the area covered by the given geometry in the 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, then null is returned.

This function can also be called as a method.

### Syntax:

```
db2gse.ST_Area(geometry [unit])
```

### Parameter:

*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.

If the geometry is in an unspecified coordinate system and the *unit* parameter is specified, or if the geometry is in a geographic coordinate system and a linear unit is specified, then an error is returned (SQLSTATE 38SU4).

### 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 selects the sales region ID and area:

```
SELECT id, ST_Area(geometry) AS area
FROM   sample_polygons
```

#### Results:

| ID | AREA                     |
|----|--------------------------|
| 1  | +1.0000000000000000E+002 |
| 2  | +2.0000000000000000E+002 |
| 3  | +2.5000000000000000E+001 |

#### Example 2:

## ST\_Area

The following SELECT statement selects 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  | +1.000000000000000E+002 | +9.29034116132748E+000 | +3.58702077598427E-006 |
| 2  | +2.000000000000000E+002 | +1.85806823226550E+001 | +7.17404155196855E-006 |
| 3  | +2.500000000000000E+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 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
```

The following code adds 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  | +7.946987880000000E+008 | +7.38302286101346E+007 | +2.85060106320552E+001 |

### Related reference:

- “The DB2GSE.ST\_UNITS\_OF\_MEASURE catalog view” on page 242

---

## ST\_AsBinary

ST\_AsBinary 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))
```

## ST\_AsBinary

```
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'0101000000000000000000000024400000000000003440' |

### Related reference:

- “Well-known binary (WKB) representation” on page 503

---

## ST\_AsGML

ST\_AsGML 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)
```

### Parameter:

*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 representation of the geometries. The geometry is converted to a GML fragment by the ST\_AsGML function.

```
SELECT id, cast(ST_AsGML(geometry) AS varchar(110)) AS gml_fragment
FROM   sample_points
WHERE  id = 1100
```

**Results:**

The SELECT statement returns the following result set:

| ID   | GML_FRAGMENT  |
|------|---|
| 1100 | <gml:Point srsName";EPSG:4269";><gml:coord><br><gml:X>10</gml:X><gml:Y>20</gml:Y><br></gml:coord></gml:Point> |

**Related reference:**

- “Spatial functions that convert geometries to and from data exchange formats” on page 243
- “Geography Markup Language (GML) representation” on page 505

---

### ST\_AsShape

ST\_AsShape 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.

#### Syntax:

►—db2gse.ST\_AsShape—(—*geometry*—)—————►

#### Parameter:

*geometry*

A value of type ST\_Geometry or one of its subtypes to be converted to the corresponding ESRI shape representation.

#### Return type:

BLOB(2G)

#### Example:

The following code fragment illustrates how to use the ST\_AsShape function to convert the points in the geometry column of the SAMPLE\_POINTS table into shape binary representation in the shape BLOB column. This example populates the shape column from the geometry column. The shape binary representation is used to display the geometries in geobrowsers, which require geometries to comply with the ESRI shapefile format, or to construct the geometries for the \*.SHP file of the shape file.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE SAMPLE_POINTS (id integer, geometry ST_POINT, shape BLOB(32K))
```

```
INSERT INTO SAMPLE_POINTS (id, geometry)
VALUES
    (1100, ST_Point(10, 20, 1))
```

```
INSERT INTO sample_points(id, shape)
VALUES (2222,
    (SELECT ST_AsShape(geometry)
     FROM sample_points
     WHERE id = 1100))
```

```
SELECT id, substr(ST_AsShape(geometry), 1, 20) AS shape
FROM sample_points
WHERE id = 1100
```

Returns:

ID      SHAPE

```
-----
1100 x'01000000000000000000000024400000000000003440'
```

**Related reference:**

- “Shape representation” on page 505

## ST\_AsText

ST\_AsText takes a geometry as an input parameter and returns its well-known text representation.

If the given geometry is null, then null is returned.

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)
```

## ST\_AsText

```
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 | LINSTRING ( 200.00000000 100.00000000,<br>210.00000000 130.00000000, 220.00000000<br>140.00000000)   |
| 3  | st_polygon    | POLYGON (( 110.00000000 120.00000000,<br>130.00000000 120.00000000, 130.00000000<br>140.00000000, 110.00000000140.00000000,<br>110.00000000 120.00000000)) |

### Related reference:

- “Well-known text (WKT) representation” on page 497

---

## ST\_Boundary

ST\_Boundary 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

## ST\_Boundary

```
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

ST\_Buffer 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, then 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 used.
- If *geometry* is in a geographic coordinate system, the angular unit associated with this coordinate system is used.

If the geometry is in an unspecified coordinate system and the *unit* parameter is specified, or if the geometry is in a geographic coordinate system and a linear unit is specified, then an error is returned (SQLSTATE 38SU4).

### 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 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))
```

## ST\_Buffer

### Example 1:

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,<br>59.00000000 55.00000000, 54.00000000 59.00000000, 49.00000000<br>60.00000000, 44.00000000 58.00000000, 41.00000000 53.00000000,<br>40.00000000 48.00000000,42.00000000 43.00000000, 47.00000000<br>41.00000000, 52.00000000 40.00000000, 57.00000000 42.00000000,<br>60.00000000 50.00000000))  |
| 2  | st_linestring   | POLYGON (( 230.00000000<br>140.00000000, 229.00000000 145.00000000, 224.00000000<br>149.00000000, 219.00000000 150.00000000, 213.00000000 147.00000000,<br>203.00000000 137.00000000, 201.00000000 133.00000000, 191.00000000<br>103.00000000, 191.00000000 99.00000000, 192.00000000 95.00000000,<br>196.00000000 91.00000000, 200.00000000 91.00000000,204.00000000<br>91.00000000, 209.00000000 97.00000000, 218.00000000 124.00000000,<br>227.00000000 133.00000000, 230.00000000 140.00000000)) |
| 3  | st_polygon      | POLYGON (( 140.00000000 120.00000000,<br>140.00000000 140.00000000, 139.00000000 145.00000000, 130.00000000<br>150.00000000, 110.00000000 150.00000000, 105.00000000 149.00000000,<br>100.00000000 140.00000000,100.00000000 120.00000000, 101.00000000<br>115.00000000, 110.00000000 110.00000000,130.00000000 110.00000000,<br>135.00000000 111.00000000, 140.00000000 120.00000000))  |
| 4  | st_multipolygon | POLYGON (( 55.00000000 30.00000000,<br>55.00000000 40.00000000, 54.00000000 45.00000000, 45.00000000<br>50.00000000, 30.00000000 50.00000000, 25.00000000 49.00000000,<br>20.00000000 40.00000000, 20.00000000 30.00000000, 21.00000000<br>25.00000000, 30.00000000 20.00000000, 45.00000000 20.00000000,<br>50.00000000 21.00000000, 55.00000000 30.00000000))  |

### Example 2:

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 125.00000000)) |

### Example 3:

The following SELECT statement illustrates the result of applying a buffer with the unit parameter specified.

```
SELECT id, spatial_type,
       cast(ST_AsText(ST_Buffer(geometry, 10, 'METER')) AS varchar(680))
       AS buffer_10_meter
FROM   sample_geometries
WHERE  id = 3
```

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)) |

### Related reference:

- “The DB2GSE.ST\_UNITS\_OF\_MEASURE catalog view” on page 242

---

## ST\_Centroid

ST\_Centroid 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:

## ST\_Centroid

►—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

ST\_ChangePoint 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:**

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) )
```

## ST\_ChangePoint

```
INSERT INTO sample_lines VALUES
(2, ST_LineString('linestring z (0 0 4, 5 5 5, 10 10 6, 5 5 7)', 0) )
```

### Example 1:

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
```

Results:

NEW

```
-----
LINESTRING ( 10.00000000 10.00000000, 6.00000000 6.00000000, 0.00000000
0.00000000, 10.00000000 0.00000000, 6.00000000 6.00000000, 0.00000000
10.00000000)
```

### Example 2:

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

ST\_Contains takes two geometries as input parameter and returns 1 if the first geometry completely contains the second; otherwise it returns 0 (zero) to indicate that the first geometry does not completely contain the second.

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 is returned.

### 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

**Examples:**

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 1:**

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
```

## ST\_Contains

Results:

| POLYGON_ID | CONTAINS         | POINT_ID |
|------------|------------------|----------|
| 100        | does contain     | 1        |
| 100        | does not contain | 2        |

### Example 2:

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      |

### Related reference:

- “ST\_Within” on page 478

---

## ST\_ConvexHull

ST\_ConvexHull 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 above 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,<br>20.00000000 20.00000000, 30.00000000<br>30.00000000, 30.00000000 50.00000000,<br>20.00000000 40.00000000)) |

## ST\_ConvexHull

```
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

ST\_CoordDim 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

ST\_Crosses 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

## ST\_Crosses

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  | 0       |
| 3  | 1  | 1       |
| 1  | 2  | -       |
| 2  | 2  | 0       |
| 3  | 2  | 1       |

|   |   |   |
|---|---|---|
| 1 | 3 | - |
| 2 | 3 | 1 |
| 3 | 3 | 0 |

**Related reference:**

- “Functions that make comparisons” on page 252

---

**ST\_Difference**

ST\_Difference takes two geometries as input parameters and returns the geometry that is the difference of the given geometries. The difference of the two geometries is that part of the first geometry that does not overlap with the second 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 any of the two given geometries is null, or if the two given geometries are not of the same dimension, then 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 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\_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*.

## ST\_Difference

### Return type:

db2gse.ST\_Geometry

### Examples:

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_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<br>10.00000000, 20.00000000 20.00000000,<br>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**

ST\_Dimension 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

## ST\_Dimension

### 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

ST\_Disjoint 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:**

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 1:**

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        |

## ST\_Disjoint

|   |   |   |
|---|---|---|
| 1 | 2 | 0 |
| 1 | 3 | 1 |
| 1 | 4 | 1 |
| 1 | 5 | 0 |

### Example 2:

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 3:

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        |

### Related reference:

- “Functions that make comparisons” on page 252

---

## ST\_Edge\_GC\_USA

ST\_Edge\_GC\_USA is the function that implements the DB2SE\_USA\_GEOCODER which geocodes addresses located in the United States of America into points. The addresses are compared (matched) against EDGE files, which are provided on the geocoder data CD.

The function takes the street number and name, the city name, the state, the zip code, and the spatial reference system identifier for the resulting point as input parameters and returns an ST\_Point value. Additionally, several configuration parameters that influence the geocoding process can be specified.

### Syntax:

```
► db2gse.ST_Edge_GC_USA(—street—,—city—,—state—,—zip—,—srs_id—,——————►
► —spelling_sens—,—min_match_score—,—side_offset—,—side_offset_units—,—end_offset—,—————►
► —base_map—,—locator_file—)——————►
```

### Parameter:

- street* A value of type VARCHAR(128) that contains the street number and name of the address to be geocoded.
- This value must not be null.
- city* A value of type VARCHAR(128) that contains the name of the city of the address to be geocoded.
- This value can be null if the *zip* parameter is specified.
- state* A value of type VARCHAR(128) that contains the name of the state of the address to be geocoded. The state can be abbreviated or spelled out.
- This value can be null if the *zip* parameter is specified.
- zip* A value of type VARCHAR(10) that contains the zip code of the address to be geocoded. The zip code can be given as 5 digits or in the 5+4 notation.
- This value can be null if the *city* and *state* parameters are specified.
- srs\_id* A value of type INTEGER that contains the numeric identifier of the spatial reference system for the resulting point. The value must identify an existing spatial reference system, that uses a projected coordinate system based on the geographic coordinate system GCS\_NORTH\_AMERICAN\_1983, or an existing spatial reference system that uses the geographic coordinate system itself, GCS\_NORTH\_AMERICAN\_1983.
- 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).
- spelling\_sens* A value of type INTEGER that specifies the spelling sensitivity that should be applied to the given address. The value must be in the range from 0 (zero) to 100. The higher this value is, the more strict the

geocoder will be regarding differences in the spelling of the given address. Deviations result in a higher penalty that will be applied to the final score of the match.

If the spelling sensitivity is set too high, fewer addresses might be geocoded successfully, and a null will be returned instead. If the spelling sensitivity is set too low, a more unmatching addresses might be considered correct matches due to the accepted level of difference in the spelling of the addresses. **Recommendation:** Set this value to 60.

If this value is null, the spelling sensitivity will be derived from the locator file. If it is not specified in the locator file, a spelling sensitivity of 60 is used.

### *min\_match\_score*

A value of type INTEGER that contains the minimum score value that a point must have to be considered a match for the given address. The minimum score value must be in the range from 0 (zero) to 100. If the score of the point is lower than the *min\_match\_score* value, null is returned instead of the point, and the address is not geocoded.

Different factors like the quality of the base map, the spelling sensitivity, or the accuracy if the address influence the score of a point. **Recommendation:** Set this value to 80.

If this value is null, the minimum match score will be derived from the locator file. If it is not specified in the locator file, a minimum score value of 80 is used.

### *side\_offset*

A value of type DOUBLE that specifies how far a resulting point is to be placed off the center of the street. The value must be larger than or equal to 0 (zero). The *side\_offset\_unit* parameter identifies the units that are used to measure the side offset.

If this value is null, the side offset will be derived from the locator file. If it is not specified in the locator file, a side offset of 0.0 is used.

### *side\_offset\_units*

A value of type VARCHAR(128) that contains the units in which the *side\_offset* parameter is measured. The value must be one of the following units:

- Inches
- Points
- Feet
- Yards
- Miles

- Nautical miles
- Millimeters
- Centimeters
- Meters
- Kilometers
- Decimal degrees
- Projected meters
- Reference data units

If this value is null, the side offset units will be derived from the locator file. If it is not specified in the locator file, the side offset will be measured in feet.

#### *end\_offset*

A value of type INTEGER that indicates how far a point that would be exactly at the end of a street segment should be placed in the segment instead. The value must be larger than or equal to 0 (zero). This parameter is used to avoid placing resulting points in the middle of a street at intersections. The end offset is measured in points (the smallest possible resolution) on the base map.

If this value is null, the end offset will be derived from the locator file. If it is not specified in the locator file, an end offset of 3 is used.

#### *base\_map*

A value of type VARCHAR(256) that contains the fully qualified path, including the base name, to the base map (.edg) file. The base map file is used by the geocoder to match the given addresses against. The base maps supplied by the DB2 Spatial Extender should be used. You can use this parameter if you placed the base maps in a different directory.

If this value is null, the path to the base map will be derived from the locator file. If it is not specified in the locator file, the base map will be searched for in the sqllib directory of the current instance, in the gse/refdata subdirectory. The base name of the file searched for is usa.edg.

#### *locator\_file*

A value of type VARCHAR(256) that contains the fully qualified path, including the base name, to the locator file that contains additional configuration parameters for the geocoder. The locator file supplied by the DB2 Spatial Extender should be used.

## ST\_Edge\_GC\_USA

If this value is null, the locator file will be searched for in the sqllib directory of the current instance, in the gse/cfg/geocoder subdirectory. The base name of the file searched for is EDGELocator.loc.

### Return type:

db2gse.ST\_Point

### Examples:

#### Example 1:

The following code creates a table SAMPLE\_GEOCODING and inserts two addresses that are subsequently geocoded. The minimum match score will be set to 50 for the given addresses, and the spatial reference system for the resulting points is 1.

```
SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse
```

```
CREATE TABLE sample_geocoding (  
  street VARCHAR(128),  
  city   VARCHAR(128),  
  state  VARCHAR(128),  
  zip    VARCHAR(5) )
```

```
INSERT INTO geocoding(street, city, state, zip)  
VALUES ('1212 New York Ave NW', 'Washington', 'DC', '20005'),  
('100 First North Street', 'San Jose', 'CA', NULL)
```

```
SELECT VARCHAR(ST_AsText(ST_Edge_GC_USA(street, city, state, zip, 1,  
  CAST(NULL AS INTEGER), 50, CAST(NULL AS DOUBLE),  
  CAST(NULL AS VARCHAR(128)), CAST(NULL AS INTEGER),  
  CAST(NULL AS VARCHAR(256)), CAST(NULL AS VARCHAR(256))))), 50)  
FROM sample_geocoding
```

### Results:

```
1  
-----  
POINT ( -77.02829300 38.90049000)  
POINT ( -121.94507200 37.28766700)
```

#### Example 2:

In this example, a spatial reference system is created that uses a projected coordinate system. To simplify the interface of the geocoding function, a user-defined function is created to wrap the ST\_Edge\_GC\_USA function.

```
db2se create_srs <db_name> -srsName CALIFORNIA -srsId 101 -xScale 1  
-coordsysName NAD_1983_STATEPLANE_CALIFORNIA_I_FIPS_0401
```



```

SET CURRENT FUNCTION PATH = CURRENT FUNCTION PATH, db2gse

CREATE FUNCTION California_GC (
  street VARCHAR(128), city VARCHAR(128), zip VARCHAR(10))
  RETURNS db2gse.ST_Point
  LANGUAGE SQL
  RETURN db2gse.ST_Edge_GC_USA(street, city, 'CA', zip, 101,
    CAST(NULL AS INTEGER), CAST(NULL AS INTEGER),
    CAST(NULL AS DOUBLE), CAST(NULL AS VARCHAR(128)),
    CAST(NULL AS INTEGER), CAST(NULL AS VARCHAR(256)))

CREATE TABLE sample_geocoding (
  street VARCHAR(128),
  city VARCHAR(128),
  state VARCHAR(128),
  zip VARCHAR(5) )

INSERT INTO geocoding(street, city, state, zip)
VALUES ('100 First North Street', 'San Jose', 'CA', NULL)

SELECT VARCHAR(ST_AsText(California_GC(street, city, zip)), 50)
FROM sample_geocoding

```

Results:

```

1
-----
POINT ( 2004879.000000000 272723.000000000)

NetBIOS

```

**Note:** The values of the X and Y coordinates of the point are different than in the first example because a different spatial reference system is used.

---

## ST\_Distance

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.

This function can also be called as a method.

**Syntax:**

## ST\_Distance

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 used.
- If *geometry1* is in a geographic coordinate system, the angular unit associated with this coordinate system is used.

If *geometry1* is in an unspecified coordinate system and the *unit* parameter is specified, or if *geometry1* is in a geographic coordinate system and a linear unit is specified, then an error is returned (SQLSTATE 38SU4).

### Return type:

DOUBLE

### Examples:

The following code creates and populates 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 1:

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 2:

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)
```

## ST\_Distance

```
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      | SG1_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 3:

The following SELECT statement calculates the distance in kilometers between the various geometries.

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, 'KILOMETER')
           AS DECIMAL(10, 4)) AS distance
FROM sample_geometries1 sg1, sample_geometries2 sg2
ORDER BY sg1.id
```

Results:

| SG1_ID | SG1_TYPE      | SG1_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 |

### Related reference:

- “Functions that make comparisons” on page 252

---

**ST\_Endpoint**

ST\_Endpoint 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\_Endpoint

### Related reference:

- “ST\_PointN” on page 443

---

## ST\_Envelope

ST\_Envelope 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)) |

#### Related reference:

- “ST\_MBR” on page 393

---

## ST\_EnvIntersects

ST\_EnvIntersects 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.

## ST\_EnvIntersects

### Syntax:

►►db2gse.ST\_EnvIntersect(—*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

ST\_EqualCoordsys 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:

## ST\_EqualCoordsys

1  
-----  
0

### Related reference:

- “The DB2GSE.ST\_COORDINATE\_SYSTEMS catalog view” on page 229

---

## ST\_Equals

ST\_Equals 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\_Equal 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
```

## ST\_Equals

```
(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      |

### Related reference:

- “Functions that make comparisons” on page 252

---

## ST\_EqualsSRS

ST\_EqualsSRS 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_EqualsSRS(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

```

#### Related reference:

- “The DB2GSE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view” on page 238

---

## ST\_ExteriorRing

ST\_ExteriorRing 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:

## ST\_ExteriorRing

*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<br>120.00000000, 90.00000000 150.00000000, 40.00000000 150.00000000,<br>40.00000000 120.00000000) |
| 2  | LINESTRING ( 10.00000000 10.00000000, 50.00000000<br>10.00000000, 10.00000000 30.00000000, 10.00000000 10.00000000)                                  |

### Related reference:

- “ST\_Boundary” on page 302

---

**ST\_FindMeasure or ST\_LocateAlong**

ST\_FindMeasure or ST\_LocateAlong 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 | db2gse.ST_LocateAlong] (—geometry—, —measure—)

```

**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:**

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.

## ST\_FindMeasure or ST\_LocateAlong

```
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 1:

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 2:

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)
```

**Related reference:**



- “ST\_MeasureBetween, ST\_LocateBetween” on page 397

---

## ST\_Generalize

ST\_Generalize 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-Decker 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, then an empty geometry of type ST\_Point is returned. If the given geometry or the threshold is null, then 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 larger 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 lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

## ST\_Generalize

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 1:

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 2:

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

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:

```

db2gse.ST_GeomCollection(
  (
    wkt
    wkb
    shape
    gml
  )
  [, srs_id]
)

```

### 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:

## ST\_GeomCollection

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)

```

**Related reference:**

- “Well-known text (WKT) representation” on page 497
- “Well-known binary (WKB) representation” on page 503
- “Shape representation” on page 505
- “Geography Markup Language (GML) representation” on page 505

---

**ST\_GeomCollFromTxt**

ST\_GeomCollFromTxt 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.

## ST\_GeomCollFromTxt

*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 |



## ST\_GeomCollFromWKB

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<br>3.00000000, 5.00000000 6.00000000)   |
| 4022 | MULTILINESTRING (( 33.00000000 2.00000000,<br>34.00000000 3.00000000, 35.00000000 6.00000000), ( 28.00000000<br>4.00000000, 29.00000000 5.00000000, 31.00000000 8.00000000,<br>43.00000000 12.00000000)) |

### Related reference:

- “Well-known binary (WKB) representation” on page 503



---

**ST\_Geometry**

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:**

```

▶▶ db2gse.ST_Geometry ( ( wkt
                        | wkb
                        | shape
                        | gml
                        ) [ , srs_id ] )

```

**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).

## ST\_Geometry

### 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 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\_WKTTToSql 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,<br>35.00000000 6.00000000)                    |
| 7003 | POLYGON (( 3.00000000 3.00000000, 5.00000000 3.00000000,<br>4.00000000 6.00000000, 3.00000000 3.00000000)) |
| 7004 | POINT ( 50.00000000 60.00000000)   |

### Related reference:

- “Well-known text (WKT) representation” on page 497

---

**ST\_GeometryN**

ST\_GeometryN 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),
```

## ST\_GeometryN

```
(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))
```

**Related reference:**

- “ST\_NumGeometries” on page 424

---

## ST\_GeometryType

ST\_GeometryType 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_MULTIPPOINT" |

---

## ST\_GeomFromText

ST\_GeomFromText 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.

## ST\_GeomFromText

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,<br>35.00000000 6.00000000)                    |
| 1253 | POLYGON (( 3.00000000 3.00000000, 5.00000000 3.00000000,<br>4.00000000 6.00000000, 3.00000000 3.00000000)) |

### Related reference:

- “Well-known text (WKT) representation” on page 497

---

**ST\_GeomFromWKB**

ST\_GeomFromWKB 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.

## ST\_GeomFromWKB

```
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<br>3.00000000, 35.00000000 6.00000000)                    |
| 1903 | POLYGON (( 3.00000000 3.00000000, 5.00000000 3.00000000,<br>4.00000000 6.00000000, 3.00000000 3.00000000)) |

### Related reference:

- “Well-known binary (WKB) representation” on page 503

---

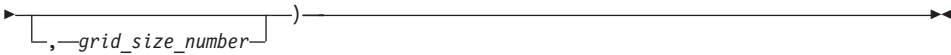
## ST\_GetIndexParms

ST\_GetIndexParms 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 ) )
```





**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 parameter is 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).

## ST\_GetIndexParms

### 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:

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 1:

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 2:

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 |

### Related concepts:

- “Spatial indexes” on page 105

---

## ST\_InteriorRingN

ST\_InteriorRingN 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 and not by any geometric orientation.

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:

## ST\_InteriorRingN

►—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) |

### Related reference:

- “ST\_ExteriorRing” on page 339
- “ST\_NumInteriorRing” on page 425

---

**ST\_Intersection**

ST\_Intersection takes two geometries as input parameters and returns the geometry that is the intersection of the two given geometries. The intersection is the part of the first geometry that is also part of the second geometry. The resulting geometry is represented in the spatial reference system of the first 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 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 any of the two given geometries is null, then null is returned.

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

**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 different geometries and then determines the intersection (if any) with the first one.

## ST\_Intersection

```
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<br>30.00000000, 50.00000000 50.00000000, 30.00000000 50.00000000, 30.00000000<br>30.00000000)) |
|       | 1     | 2 LINESTRING ( 30.00000000 40.00000000, 30.00000000<br>30.00000000)  |
|       | 1     | 3 POLYGON (( 40.00000000 40.00000000, 50.00000000<br>40.00000000, 50.00000000 50.00000000, 40.00000000 50.00000000, 40.00000000<br>40.00000000)) |
|       | 1     | 4 POINT EMPTY  |
|       | 1     | 5 LINESTRING ( 30.00000000 30.00000000, 50.00000000<br>50.00000000)  |

5 record(s) selected.

---

## ST\_Intersects

ST\_Intersects takes two geometries as input parameters and returns 1 if the given geometries intersect. If the geometries do not 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 is returned.

**Syntax:**

```
►►—db2gse.ST_Intersects—(—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 intersection with *geometry2*.

*geometry2*

A value of type ST\_Geometry or one of its subtypes that represents the geometry that is to be tested for intersection with *geometry1*.

**Return type:**

INTEGER

**Examples:**

The following code creates and populates 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.

## ST\_Intersects

```
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 |

### Related reference:

- “Functions that make comparisons” on page 252

---

## ST\_Is3D

ST\_Is3D 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

ST\_IsClosed 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:

## ST\_IsClosed

| ID | IS_CLOSED |
|----|-----------|
| 6  | 0         |
| 7  | 1         |

---

## ST\_IsEmpty

ST\_IsEmpty takes a geometry as an input parameter and returns 1 if the given geometry is empty. Otherwise 0 (zero) is returned. A geometry is empty if it does not have any points that define it.

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_IsMeasured(geometry) Is_Measured
FROM sample_geoms
```

Results:

| ID | IS_MEASURED |
|----|-------------|
| 1  | 0           |
| 2  | 0           |
| 3  | 1           |
| 4  | 0           |
| 5  | 1           |

---

## ST\_IsMeasured

ST\_IsMeasured takes a geometry as an input parameter and returns 1 if the given geometry has M coordinates (measures). 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_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.

## ST\_IsMeasured

```
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

ST\_IsRing 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       |

**Related reference:**

- “ST\_IsClosed” on page 368
- “ST\_IsSimple” on page 373

---

**ST\_IsSimple**

ST\_IsSimple takes a geometry as an input parameter and returns 1 if the given geometry is simple. Otherwise, 0 (zero) is returned.

## ST\_IsSimple

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

ST\_IsValid 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:

## ST\_IsValid

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

ST\_Length takes a curve or multcurve and, optionally, a unit as input parameters and returns the length of the given curve or multcurve in the given unit of measure.

If the given curve or multcurve is null or is empty, then 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 used.
- If *curve* is in a geographic coordinate system, the angular unit associated with this coordinate system is used.

If *curve* is in an unspecified coordinate system and *unit* is given, or if the geometry is in a geographic coordinate system and a linear unit is specified, then an error is returned (SQLSTATE 38SU4).

### Return type:

DOUBLE

### Examples:

The following code creates a table SAMPLE\_GEOMETRIES and inserts 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 1:

The following SELECT statement calculates the length of the line in the SAMPLE\_GEOMTRIES table.

## ST\_Length

```
SELECT id, spatial_type, cast(ST_Length(geometry..ST_ToLineString)
    AS DECIMAL(7, 2)) AS "Line Length"
FROM   sample_geometries
WHERE  id = 1110
```

Results:

| ID   | SPATIAL_TYPE  | Line Length |
|------|---------------|-------------|
| 1110 | ST_LineString | 10.00       |

### Example 2:

The following SELECT statement calculates the length of the multiline in the SAMPLE\_GEOMETRIES table.

```
SELECT id, spatial_type, ST_Length(ST_ToMultiLine(geometry))
    AS multiline_length
FROM   sample_geometries
WHERE  id = 1111
```

Results:

| ID   | SPATIAL_TYPE       | MULTILINE_LENGTH       |
|------|--------------------|------------------------|
| 1111 | ST_MultiLineString | +2.76437123387202E+001 |

---

## ST\_LineFromText

ST\_LineFromText takes a well-known text representation of a linestring and, optionally, a spatial reference system identifier as input parameters and returns the corresponding linestring.

If the given well-known text representation is null, then null is returned.

The preferred version for this functionality is ST\_LineString.

### Syntax:

```
db2gse.ST_LineFromText(wkt [, srs_id])
```

### Parameter:

*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).

**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\_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 ( 850.00000000 250.00000000, 850.00000000 850.00000000) |
| 1111 | LINESTRING EMPTY   |

**Related reference:**

- “Well-known text (WKT) representation” on page 497

---

## ST\_LineFromWKB

ST\_LineFromWKB 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.

## ST\_LineFromWKB

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)
```

**Related reference:**

- “Well-known binary (WKB) representation” on page 503

---

## ST\_LineString

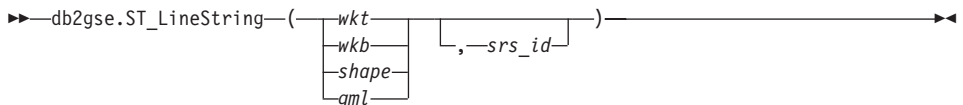
ST\_LineString constructs a linestring from one of the following inputs:

- A well-known text representation
- A well-known binary representation
- An ESRI shape representation
- A representation in the geography markup language (GML)

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:**



**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.

## ST\_LineString

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)   |

### Related reference:

- “Spatial functions that convert geometries to and from data exchange formats” on page 243
- “Well-known text (WKT) representation” on page 497



---

## ST\_LineStringN

ST\_LineStringN 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),
```

## ST\_LineStringN

```
(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<br>5.00000000, 31.00000000 8.00000000, 43.00000000 12.00000000) |
| 1111 | LINESTRING ( 58.00000000 4.00000000, 59.00000000<br>5.00000000, 61.00000000 8.00000000)                          |

**Related reference:**

- “ST\_NumLineStrings” on page 427

---

## ST\_M

ST\_M can either:

- Take a point as an input parameter and return its M (measure) coordinate
- Take a point and an M coordinate and return the point itself with its M coordinate set to the given measure, even if the specified point has no existing M coordinate.

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:**

These examples illustrate 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 1:**

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  | +5.00000000000000E+000 |
| 2  | +4.00000000000000E+000 |
| 3  | +7.00000000000000E+000 |

**Example 2:**

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
```

## ST\_M

Results:

```
ID          M_COORD_40
-----
          3 POINT ZM (3.00000000 8.00000000 23.00000000 40.00000000)
```

### Related reference:

- “ST\_X” on page 482
- “ST\_Y” on page 484
- “ST\_Z” on page 485

---

## ST\_MaxM

ST\_MaxM 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:

These examples illustrate 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 1:**

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 2:**

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            |

**Related concepts:**

- “ST\_MaxX” on page 388

**Related reference:**

- “ST\_MaxY” on page 390
- “ST\_MaxZ” on page 392
- “ST\_MinM” on page 400

ST\_MaxX 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:

These examples illustrate 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 1:**

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 2:**

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 3:**

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:

## ST\_MaxX

| MIN_X | MIN_Y | MIN_Z | MIN_M | MAX_X | MAX_Y | MAX_Z | MAX_M |
|-------|-------|-------|-------|-------|-------|-------|-------|
| 0     | 0     | 10    | 3     | 120   | 140   | 40    | 16    |

### Related reference:

- “ST\_MaxM” on page 386
- “ST\_MaxY” on page 390
- “ST\_MaxZ” on page 392
- “ST\_MinX” on page 401

---

## ST\_MaxY

ST\_MaxY 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:

These examples illustrate 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 1:**

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 2:**

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           |

**Related concepts:**

- “ST\_MaxX” on page 388

**Related reference:**

- “ST\_MaxM” on page 386
- “ST\_MaxZ” on page 392
- “ST\_MinY” on page 403

ST\_MaxZ 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:

These examples illustrate 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 1:**

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 2:**

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            |

**Related concepts:**

- “ST\_MaxX” on page 388

**Related reference:**

- “ST\_MaxM” on page 386
- “ST\_MaxY” on page 390
- “ST\_MinZ” on page 405

---

**ST\_MBR**

ST\_MBR 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, then the horizontal or vertical linestring itself is returned. Otherwise, the minimum bounding rectangle of the geometry is returned as a polygon. If the given geometry is null or is empty, then null is returned.

## ST\_MBR

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,<br>15.00000000 11.00000000, 5.00000000 11.00000000,<br>5.00000000 5.00000000))        |
| 2  | POLYGON (( 20.00000000 30.00000000, 30.00000000 30.00000000,<br>30.00000000 35.00000000, 20.00000000 35.00000000,<br>20.00000000 30.00000000 )) |

### Related reference:

- “ST\_Envelope” on page 332
- “ST\_MBRIntersects” on page 395

---

## ST\_MBRIntersects

ST\_MBRIntersects 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:

These examples illustrate 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
```

## ST\_MBRIntersects

```
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 1:

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 2:

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              |

### Related reference:

- “ST\_EnvIntersects” on page 333

- “ST\_MBR” on page 393

---

## ST\_MeasureBetween, ST\_LocateBetween

ST\_MeasureBetween or ST\_LocateBetween takes 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.

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.

Both functions can also be called as methods.

### Syntax:

```

▶—┬──db2gse.ST_MeasureBetween──┬──(—geometry—,—startMeasure—,—endMeasure—)──▶
  └──db2gse.ST_LocateBetween──┘

```

### 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:

## ST\_MeasureBetween and ST\_LocateBetween

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,<br>3.00000000 3.00000000 6.00000000,<br>4.00000000 4.00000000 6.00000000,<br>5.00000000 5.00000000 6.00000000) |

---

## ST\_MidPoint

ST\_MidPoint 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) |

## ST\_MidPoint

```
2 POINT ( 3.00000000 3.45981800)
3 POINT ( 5.00000000 0.00000000)
4 POINT ( 7.50000000 20.00000000)
```

---

## ST\_MinM

ST\_MinM 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:

These examples illustrate 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 1:**

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 2:**

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             |

**Related reference:**

- “ST\_MaxM” on page 386
- “ST\_MinX” on page 401
- “ST\_MinY” on page 403
- “ST\_MinZ” on page 405

---

**ST\_MinX**

ST\_MinX 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.

## ST\_MinX

### 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:

These examples illustrate 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 1:

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 2:**

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
```

**Related concepts:**

- “ST\_MaxX” on page 388

**Related reference:**

- “ST\_MinM” on page 400
- “ST\_MinY” on page 403
- “ST\_MinZ” on page 405

**ST\_MinY**

ST\_MinY 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:**

## ST\_MinY

DOUBLE

### Examples:

These examples illustrate 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 1:

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 2:

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
```

**Related reference:**

- “ST\_MaxY” on page 390
- “ST\_MinM” on page 400
- “ST\_MinX” on page 401
- “ST\_MinZ” on page 405

---

**ST\_MinZ**

ST\_MinZ 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:**

These examples illustrate 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) )
```

## ST\_MinZ

```
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 1:

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 2:

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            |

### Related reference:

- “ST\_MaxZ” on page 392
- “ST\_MinM” on page 400
- “ST\_MinX” on page 401
- “ST\_MinY” on page 403



---

**ST\_MLineFromText**

ST\_MLineFromText 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

## ST\_MLineFromText

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 )) |

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “ST\_MLineFromWKB” on page 408
- “ST\_MultiLineString” on page 418

---

## ST\_MLineFromWKB

ST\_MLineFromWKB 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)

## ST\_MLineFromWKB

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,<br>65.00000000 6.00000000),<br>( 58.00000000 4.00000000, 59.00000000 5.00000000,<br>61.00000000 8.00000000),<br>( 69.00000000 3.00000000, 67.00000000 4.00000000,<br>66.00000000 7.00000000, 68.00000000 9.00000000 )) |

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “ST\_MLineFromText” on page 407
- “ST\_MultiLineString” on page 418
- “Well-known binary (WKB) representation” on page 503

---

**ST\_MPointFromText**

ST\_MPointFromText takes a well-known text representation of a multipoint and, optionally, a spatial reference 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).

## ST\_MPointFromText

```
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) |

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “ST\_MPointFromWKB” on page 412
- “ST\_MultiPoint” on page 420
- “Well-known text (WKT) representation” on page 497

---

## ST\_MPointFromWKB

ST\_MPointFromWKB 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.

## ST\_MPointFromWKB

```
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<br>16.00000000 24.00000000 13.00000000) |

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “ST\_MPointFromText” on page 411
- “ST\_MultiPoint” on page 420
- “Well-known binary (WKB) representation” on page 503
- “ST\_Point” on page 437

---

## ST\_MPolyFromText

ST\_MPolyFromText 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,
```

## ST\_MPolyFromText

```
1.00000000 28.0000000, 8.00000000 24.00000000)),  
    ( 3.00000000 3.00000000, 5.00000000 3.00000000,  
4.00000000 6.00000000, 3.00000000 3.00000000)))
```

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “ST\_MPolyFromWKB” on page 416
- “ST\_MultiPolygon” on page 422
- “Well-known text (WKT) representation” on page 497

---

## ST\_MPolyFromWKB

ST\_MPolyFromWKB 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
```

## ST\_MPolyFromWKB

```
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 )))
```

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “ST\_MPolyFromText” on page 414
- “ST\_MultiPolygon” on page 422
- “Well-known binary (WKB) representation” on page 503
- “ST\_Polygon” on page 449

---

## ST\_MultiLineString

ST\_MultiLineString constructs a multilinestring from one of the following inputs:

- 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:

```
►► db2gse.ST_MultiLineString ( ( 

|              |
|--------------|
| <i>wkt</i>   |
| <i>wkb</i>   |
| <i>gml</i>   |
| <i>shape</i> |

 [, srs_id] ) ►►
```

### 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:

## ST\_MultiLineString

```
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,<br>35.00000000 6.00000000),<br>( 28.00000000 4.00000000, 29.00000000 5.00000000,<br>31.00000000 8.00000000, 43.00000000 12.00000000),<br>( 39.00000000 3.00000000, 37.00000000 4.00000000,<br>36.00000000 7.00000000 ) ) |

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “Well-known text (WKT) representation” on page 497
- “Well-known binary (WKB) representation” on page 503
- “Shape representation” on page 505
- “Geography Markup Language (GML) representation” on page 505

---

## ST\_MultiPoint

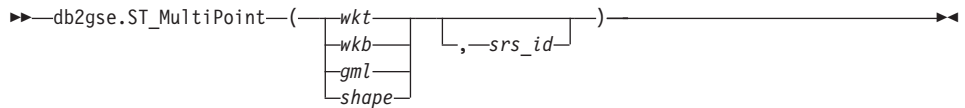
ST\_MultiPoint constructs a multipoint from one of the following inputs:

- 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:

**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))
```

## ST\_MultiPoint

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<br>3.00000000, 5.00000000 6.00000000) |

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “Well-known text (WKT) representation” on page 497
- “Well-known binary (WKB) representation” on page 503
- “Shape representation” on page 505
- “Geography Markup Language (GML) representation” on page 505

---

## ST\_MultiPolygon

ST\_MultiPolygon constructs a multipolygon from one of the following inputs:

- 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:

```
db2gse.ST_MultiPolygon(wkt | wkb | shape | gml [, srs_id])
```

### 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) )
```

## ST\_MultiPolygon

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))) |

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “Well-known text (WKT) representation” on page 497
- “Well-known binary (WKB) representation” on page 503
- “Shape representation” on page 505
- “Geography Markup Language (GML) representation” on page 505

---

## ST\_NumGeometries

ST\_NumGeometries 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                 |

**Related reference:**

- “ST\_GeometryN” on page 353

---

**ST\_NumInteriorRing**

ST\_NumInteriorRing 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:**

## ST\_NumInteriorRing

►—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         |

### Related reference:

- “ST\_InteriorRingN” on page 361

---

## ST\_NumLineStrings

ST\_NumLineStrings 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\_NumLineStrings

### Related reference:

- “ST\_LineStringN” on page 383

---

## ST\_NumPoints

ST\_NumPoints 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          |

**Related reference:**

- “ST\_PointN” on page 443

## ST\_NumPolygons

ST\_NumPolygons 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\_NumPolygons

```
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_polys
VALUES (2,
        ST_MultiPolygon ('multipolygon empty', 1) )

INSERT INTO sample_polys
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          |

### Related reference:

- “ST\_PolygonN” on page 452

---

## ST\_Overlaps

ST\_Overlaps takes two geometries as input parameters and returns 1 if the intersection of the geometries results in a geometry of the same dimension but is not equal to either of the given geometries. Otherwise, 0 (zero) is returned.

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:

These examples illustrate 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 60, 0 50))', 1) )
```

### Example 1:

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 2:

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'
```

## ST\_Overlaps

```
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 3:

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 |

### Related reference:

- “Rules for using grid indexes to optimize spatial functions” on page 121

---

## ST\_Perimeter

ST\_Perimeter 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 given units.

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_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 used.
- If *surface* is in a geographic coordinate system, the angular unit associated with this coordinate system is used.

If the surface is in an unspecified coordinate system and the *unit* parameter is specified, or if the surface is in a geographic coordinate system and a linear unit is specified, then an exception condition is raised (SQLSTATE 38SU4).

### Return type:

DOUBLE

### Examples:

These examples illustrate 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))
```

## ST\_Perimeter

### Example 1:

This example lists the ID and perimeter of the polygon.

```
SELECT id, ST_Perimeter (geometry) AS PERIMETER
FROM sample_polys
```

Results:

| ID | PERIMETER               |
|----|-------------------------|
| 1  | +1.800000000000000E+001 |

### Example 2:

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

ST\_PerpPoints takes a curve or multcurve and a point as input parameters and returns the perpendicular projection of the given point on the curve or multcurve. 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 multcurve has Z or M coordinates, the Z or M coordinate of the resulting points are computed by interpolation on the given curve or multcurve.

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:**

These examples illustrate 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 1:**

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 2:**

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:**

## ST\_PerpPoints

```
PERP
-----
MULTIPOINT ( 0.00000000 5.00000000, 5.00000000 0.00000000, 10.00000000 5.00000000 )
```

### Example 3:

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 4:

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 5:

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**

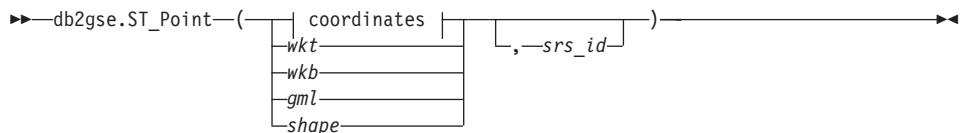
ST\_Point constructs a point from one of the following sets of input:

- 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)

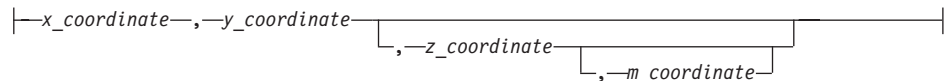
An optional spatial reference system identifier can be specified to indicate the spatial reference system that the resulting point is in.

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.

## ST\_Point

*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 ( 120.00000000 358.00000000 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.

## ST\_Point

```
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 ) |

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “ST\_Is3d” on page 366
- “ST\_IsMeasured” on page 371
- “Well-known text (WKT) representation” on page 497
- “Well-known binary (WKB) representation” on page 503
- “Shape representation” on page 505
- “Geography Markup Language (GML) representation” on page 505

---

## ST\_PointFromText

ST\_PointFromText 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)
```

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “ST\_Point” on page 437
- “ST\_PointFromWKB” on page 442

## ST\_PointFromWKB

---

## ST\_PointFromWKB

ST\_PointFromWKB 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) |

#### Related concepts:

- “Spatial data” on page 4

#### Related reference:

- “ST\_Point” on page 437
- “ST\_PointFromText” on page 440

---

## ST\_PointN

ST\_PointN 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.

## ST\_PointN

### 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 (5.00000000 5.00000000) |

### Related reference:

- “ST\_Endpoint” on page 331
- “ST\_NumPoints” on page 428
- “ST\_StartPoint” on page 459

---

## ST\_PointOnSurface

ST\_PointOnSurface 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

---

### ST\_PolyFromText

ST\_PolyFromText 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).





## ST\_PolyFromWKB

*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))

```

**Related concepts:**

- “Spatial data” on page 4

**Related reference:**

- “ST\_PolyFromText” on page 446
- “ST\_Polygon” on page 449

---

## ST\_Polygon

ST\_Polygon constructs a polygon from one of the following inputs:

- 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:**

```

▶▶ db2gse.ST_Polygon—( linestring [ wkt | wkb | shape | gml ] [ , srs_id ] )

```

**Parameters:**

## ST\_Polygon

### *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  |
|------|---|
| 1100 | POLYGON (( 10.00000000 20.00000000, 20.00000000 30.00000000<br>10.00000000 40.00000000, 10.00000000 20.00000000))   |
| 1101 | POLYGON (( 110.00000000 120.00000000, 120.00000000 130.00000000<br>110.00000000 140.00000000, 110.00000000 120.00000000))   |
| 1102 | POLYGON (( 110.00000000 120.00000000, 130.00000000 120.00000000<br>130.00000000 140.00000000, 110.00000000 140.00000000<br>110.00000000 120.00000000),<br>( 115.00000000 125.00000000, 115.00000000 135.00000000<br>125.00000000 135.00000000, 125.00000000 135.00000000<br>115.00000000 125.00000000)) |

**Related concepts:**

- “Spatial data” on page 4

**Related reference:**

## ST\_Polygon

- “Well-known text (WKT) representation” on page 497
- “Well-known binary (WKB) representation” on page 503
- “Shape representation” on page 505
- “Geography Markup Language (GML) representation” on page 505

---

## ST\_PolygonN

ST\_PolygonN 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,<br>1.00000000 28.00000000, 8.00000000 24.00000000)) |

**Related reference:**

- “ST\_NumPolygons” on page 429

---

## ST\_Relate

ST\_Relate takes two geometries and a Dimensionally Extended 9 Intersection Model (DE-9IM) matrix as input parameters and returns 1 if the given geometries meet the conditions specified by the matrix. Otherwise, 0 (zero) is returned.

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:**

## ST\_Relate

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***T**') "Overlaps ",
       ST_Relate(a.geometry, b.geometry, 'T*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***FF2') "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      |

### Related reference:

- “Functions that make comparisons” on page 252

---

## ST\_RemovePoint

ST\_RemovePoint 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:

In the following example, two linestrings are added to the SAMPLE\_LINES table. These linestrings are used in the examples below.

```
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 1:

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)
```

## ST\_RemovePoint

### Example 2:

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, ST\_SRID

ST\_SrsId (or ST\_SRID) 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] )
```

db2gse.ST\_SRID

### 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_SRID (geometry) SRSID
FROM sample_points
```

Results:

| ID | SRSID |
|----|-------|
| 1  | 0     |
| 2  | 1     |

**Related reference:**

- “ST\_Transform” on page 473

## ST\_SrsName

---

### ST\_SrsName

ST\_SrsName 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 |

#### Related reference:

- “ST\_SrsId, ST\_SRID” on page 456

---

## ST\_StartPoint

ST\_StartPoint 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:

## ST\_StartPoint

| ID | START_POINT                                 |
|----|---|
| 1  | POINT ( 10.00000000 10.00000000)            |
| 2  | POINT Z ( 0.00000000 0.00000000 4.00000000) |

### Related reference:

- “ST\_Endpoint” on page 331
- “ST\_PointN” on page 443

---

## ST\_SymDifference

ST\_SymDifference takes two geometries as input parameters and returns the geometry that is the symmetrical difference of the two given geometries. The symmetrical difference is the nonintersecting part of the two given geometries. The resulting geometry is represented in the spatial reference system of the first 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 the given geometries are equal, an empty geometry of type ST\_Point is returned. If any of the two given geometries is null, or if they have different dimensions, 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:**

These examples illustrate 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 lines of results have been reformatted for readability. The spacing in your results will vary according to your online display.

**Example 1:**

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:

## ST\_SymDifference

```
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 2:

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 3:

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))
```

### Related reference:

- “ST\_Difference” on page 317



---

**ST\_ToGeomColl**

ST\_ToGeomColl 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))
```

## ST\_ToGeomColl

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<br>3.00000000, 4.00000000 6.00000000,<br>3.00000000 3.00000000))) |
| 2  | MULTIPOINT ( 1.00000000 2.00000000)  |

---

## ST\_ToLineString

ST\_ToLineString 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

ST\_ToMultiLine 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).

## ST\_ToMultiLine

### 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

ST\_ToMultiPoint 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

---

### ST\_ToMultiPolygon

ST\_ToMultiPolygon 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

ST\_ToPoint 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.

## ST\_ToPoint

```
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 ( 30.000000000 40.000000000)
POINT EMPTY
POINT EMPTY
```

---

## ST\_ToPolygon

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

ST\_Touches 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.

## ST\_Touches

### 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  | 0       |
| 1  | 2  | 1       |
| 1  | 3  | 0       |
| 1  | 4  | 0       |
| 1  | 5  | 1       |
| 2  | 2  | 0       |
| 2  | 3  | 0       |
| 2  | 4  | 0       |

|   |   |   |
|---|---|---|
| 2 | 5 | 1 |
| 3 | 3 | 0 |
| 3 | 4 | 1 |
| 3 | 5 | 1 |
| 4 | 4 | 0 |
| 4 | 5 | 1 |
| 5 | 5 | 0 |

**Related reference:**

- “Rules for using grid indexes to optimize spatial functions” on page 121

---

**ST\_Transform**

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.

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*

## ST\_Transform

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:

The following examples illustrate 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 1:

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_11
```

Results:

| ID    | STATE_PLANE                                 |
|-------|---|
| 12457 | POINT ( 567176.000000000 1166411.000000000) |
| 12477 | POINT ( 637948.000000000 1177640.000000000) |

### Example 2:

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) |

### Related reference:

- “The DB2GSE.ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view” on page 238

---

## ST\_Union

ST\_Union 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.

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.

## ST\_Union

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:

The following examples illustrate the use of the ST\_Union function. First geometries are created and inserted into the SAMPLE\_GEOMS table and then ST\_Union is used with those geometries.

```
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 lines of results have been reformatted here for readability. The spacing in your results will vary according to your online display.

### Example 1:

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<br>10.00000000, 20.00000000 20.00000000), 10.00000000<br>20.00000000, 10.00000000 10.00000000))<br>( ( 30.00000000 30.00000000, 50.00000000<br>30.00000000,50.00000000 50.00000000, 30.00000000<br>50.00000000,30.00000000 30.00000000)) |

### Example 2:

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<br>30.00000000,50.00000000 40.00000000, 60.00000000<br>40.00000000,60.00000000 60.00000000, 40.00000000<br>60.00000000 40.00000000 50.00000000, 30.00000000<br>50.00000000, 30.00000000 30.00000000)) |

### Example 3:

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),<br>( 80.00000000 80.00000000, 100.00000000 70.00000000)) |

---

### ST\_Within

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 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 is returned.

ST\_Within performs the same logical operation that ST\_Contains performs with the parameters reversed.

#### 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

#### Examples:

These examples illustrate 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 1:**

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 2:**

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
```

**Related reference:**

- “ST\_Contains” on page 310

---

**ST\_WKBTtoSQL**

ST\_WKBTtoSQL 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\_WKBTtoSQL(*wkb*) gives the same result as ST\_Geometry(*wkb*,0). Using the ST\_Geometry function is recommended over using ST\_WKBTtoSQL because of its flexibility: ST\_Geometry takes additional forms of input as well as the well-known binary representation.

## ST\_WKBTToSQL

### Syntax:

►—db2gse.ST\_WKBTToSQL—(—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\_WKBTToSQL 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\_WKBTToSQL 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_WKBTToSQL(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,<br>50.00000000 40.00000000, 50.00000000 20.00000000)) |

**Related concepts:**

- “Spatial data” on page 4

**Related reference:**

- “ST\_Geometry” on page 351
- “ST\_WKTTtoSQL” on page 481

---

**ST\_WKTTtoSQL**

ST\_WKTTtoSQL 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\_WKTTtoSQL(*wkt*) gives the same result as ST\_Geometry(*wkt*,0). Using the ST\_Geometry function is recommended over using ST\_WKTTtoSQL because of its flexibility: ST\_Geometry takes additional forms of input as well as the well-known text representation.

**Syntax:**

►►—db2gse.ST\_WKTTtoSQL—(—*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\_WKTTtoSQL 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
```

## ST\_WKTToSQL

```
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,<br>50.00000000 40.00000000, 50.00000000 20.00000000)) |

### Related concepts:

- “Spatial data” on page 4

### Related reference:

- “ST\_Geometry” on page 351
- “ST\_WKBToSQL” on page 479

---

## ST\_X

ST\_X takes either:

- A point as an input parameter and returns its X coordinate
- A point and an X coordinate and 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:**

These examples illustrate 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 1:**

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  | +2.0000000000000000E+000 |
| 2  | +4.0000000000000000E+000 |
| 3  | +3.0000000000000000E+000 |

**Example 2:**

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) |

**Related reference:**

- “ST\_M” on page 384
- “ST\_Y” on page 484
- “ST\_Z” on page 485

ST\_Y takes either:

- A point as an input parameter and returns its Y coordinate
- A point and a Y coordinate and 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:

These examples illustrate 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 1:

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 2:

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)

```

### Related reference:

- “ST\_M” on page 384
- “ST\_X” on page 482
- “ST\_Z” on page 485

---

## ST\_Z

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:

## ST\_Z

*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:

These examples illustrate 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 1:

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 2:

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)
```

**Related reference:**

- “ST\_M” on page 384
- “ST\_X” on page 482
- “ST\_Y” on page 484

---

## Union aggregate

A union aggregate is the combination of the `ST_BuildUnionAggr` and `ST_GetAggrResult` functions. This combination aggregates 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 massively parallel processing (MPP) environments
- If a `GROUP BY` clause is used in the select
- If you use a function other than the DB2 aggregate function `MAX`

## Union aggregate

### 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 20. Transform groups

---

### Transform groups

Spatial Extender provides four transform groups that can be used with the following data exchange formats while geometries are transferred between the DB2 server and a client application.

- Well-known text representation (WKT)
- Well-known binary representation (WKB)
- ESRI shape representation
- Geography Markup Language (GML)

When retrieving data from a table that contains a spatial column, the data from the spatial column will be transformed to either a CLOB(2G) or a BLOB(2G), depending on whether a binary or textual representation was chosen. Also, when transferring spatial data in one of the listed data exchange formats to the database, the transform groups can be used.

To select which transform group is to be used when the data is transferred, the SET CURRENT DEFAULT TRANSFORM GROUP statement must be used to modify the DB2 special register CURRENT DEFAULT TRANSFORM GROUP. DB2 will use the value of the 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 take care of that task.

#### **ST\_WellKnownText**

The ST\_WellKnownText transform group can be used 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 functionality provided by ST\_AsText() is used to convert a geometry to WKT. When transferring the well-known text representation of a geometry 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).

#### **Example:**

## ST\_WellKnownText

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:

The following SQL script shows how the ST\_WellKnownText transform group can be used 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 ( 100.00000000 100.00000000, 200.00000000 100.00000000)
```

### Example 2:

The following C code illustrates how to use the ST\_WellKnownText transform group for inserting geometries using 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

The ST\_WellKnownBinary transform group can be used 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 functionality provided by ST\_AsBinary() is used to convert a geometry to WKB. When transferring the well-known binary 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).

### 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:

The following SQL script shows how the ST\_WellKnownBinary transform group can be used 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'01030000000010000000050000000000000000000000244000 |

## ST\_WellKnownBinary

```
0000000000244000000000000024400000000000003440  
000000000000344000000000000034400000000000034  
40000000000000244000000000000244000000000000  
2440'
```

### Example 2:

The following C code illustrates 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

The ST\_Shape transform group can be used to transmit data to and from DB2 using the shape representation.

When binding out a value from the database server to the client, the same functionality 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 causing 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 will vary according to your online display.

### Example 1:



## ST\_GML

When binding out a value from the database server to the client, the same functionality provided by ST\_AsGML() is used to convert a geometry to its GML representation. When transferring the GML representation of a geometry 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 causing 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 will 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 <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>
```

#### Example 2:



The following C code illustrates how to use the ST\_GML transform group for inserting geometries without the need to use 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 );
```

**ST\_GML**

## Chapter 21. Supported data formats

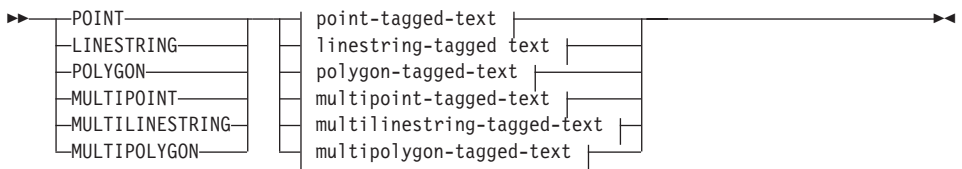
This chapter describes the industry standard spatial data formats that can be used with DB2 Spatial Extender. See “Spatial functions that convert geometries to and from data exchange formats” on page 243 for information on functions which accept and produce these formats. See “About importing and exporting spatial data” on page 89 for information on importing and exporting files containing these formats. 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

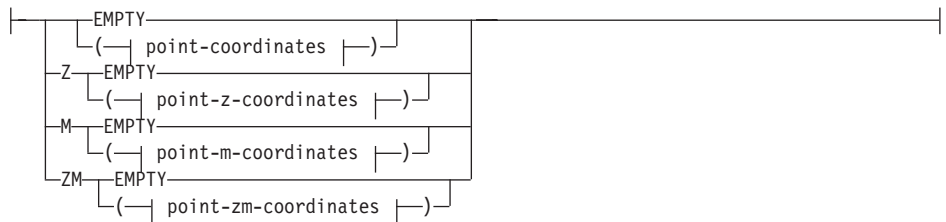
### Well-known text (WKT) representation

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:

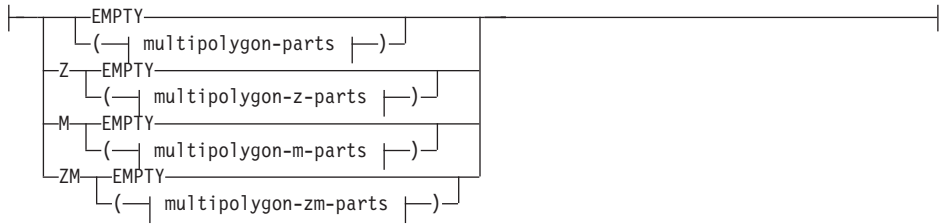


#### point-tagged-text:





## Well-known text (WKT) representation



### point-coordinates:



### point-z-coordinates:



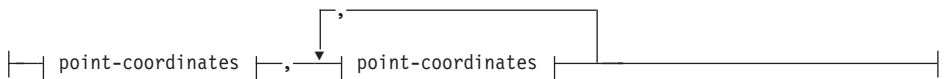
### point-m-coordinates:



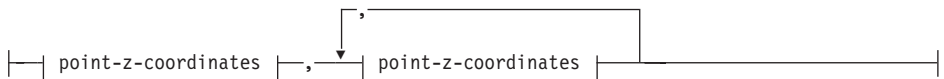
### point-zm-coordinates:



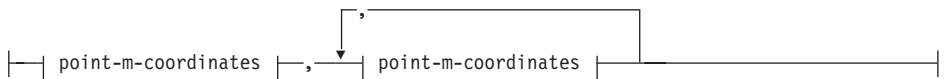
### linestring-points:



### linestring-z-points:

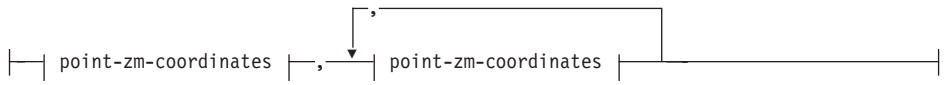


### linestring-m-points:

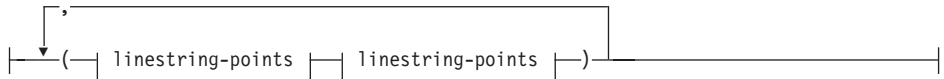


### linestring-zm-points:

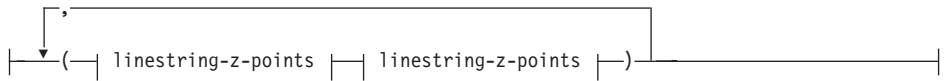
## Well-known text (WKT) representation



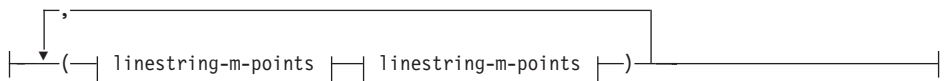
### **polygon-rings:**



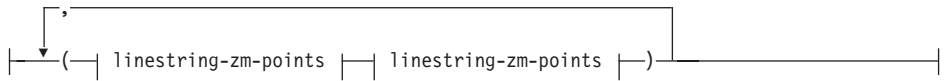
### **polygon-z-rings:**



### **polygon-m-rings:**



### **polygon-zm-rings:**



### **multipoint-parts:**



### **multipoint-z-parts:**



### **multipoint-m-parts:**



**multipoint-zm-parts:**



**multilineestring-parts:**



**multilineestring-z-parts:**



**multilineestring-m-parts:**



**multilineestring-zm-parts:**



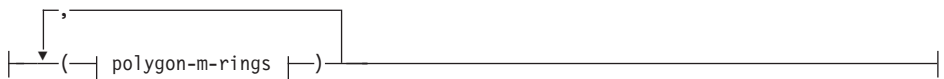
**multipolygon-parts:**



**multipolygon-z-parts:**



**multipolygon-m-parts:**



## Well-known text (WKT) representation

### multipolygon-zm-parts:



### Parameters:

*x\_coord*

A numerical value (fixed, integer, or floating point), which represents the X coordinate of a point.

*y\_coord*

A numerical value (fixed, integer, or floating point), which represents the Y coordinate of a point.

*z\_coord*

A numerical value (fixed, integer, or floating point), which represents the Z coordinate of a point.

*m\_coord*

A numerical 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 40. 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                                  |



Table 40. Geometry types and their text representations (continued)

| Geometry type   | WKT representation   | Comment   |
|-----------------|--|---|
| 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 |

### Related reference:

- “Spatial functions that convert geometries to and from data exchange formats” on page 243

## 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;
```

## Well-known binary (WKB) representation

```
    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;
```

## Well-known binary (WKB) representation

```
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.

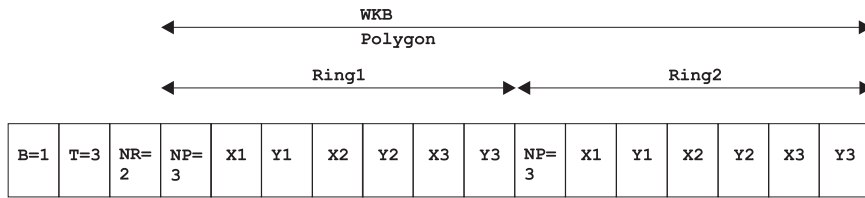


Figure 29. Geometry representation in NDR format. (B=1) of type polygon (T=3) with 2 linears (NR=2), where each ring has 3 points (NP=3).

### Related reference:

- “Spatial functions that convert geometries to and from data exchange formats” on page 243

---

## 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>.

The “Spatial functions that convert geometries to and from data exchange formats” topic in the related link section below explains the spatial functions that accept and produce shape data format.

### Related reference:

- “Spatial functions that convert geometries to and from data exchange formats” on page 243

---

## Geography Markup Language (GML) representation

DB2 Spatial Extender has several functions that generate geometries from representations in geography markup language(GML) representation.

## GML representation

See "Spatial functions that convert geometries to and from data exchange formats" in the related link section below for a detailed description of the functions provided by DB2 Spatial Extender that convert geometry values to and from GML representation.

The Geography Markup Language (GML) is an XML encoding for geographic information defined by the OpenGIS Consortium "Geography Markup Language V2" specification. This OpenGIS Consortium specification can be found at <http://www.opengis.org/techno/implementation.htm>.

### **Related reference:**

- "Spatial functions that convert geometries to and from data exchange formats" on page 243

---

## Chapter 22. Supported coordinate systems

This chapter provides reference information about the coordinate values used to interpret spatial data. The following topics are covered:

- Overview of coordinate systems
- Supported linear units
- Supported angular units
- Supported spheroids
- Supported geodetic datums
- Supported prime meridians
- Supported map projections

---

### Supported coordinate systems

This topic provides an explanation of coordinate systems syntax and lists the coordinate system values that are supported by DB2 Spatial Extender.

#### Coordinate systems syntax

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 modeled after the Petrotechnical Open Software Corporation/European Professional Surveyors Group (POSC/EPSG) coordinate system data model.

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 note above regarding the use of brackets):

```
<coordinate system> = <projected cs> |  
<geographic cs> | <geocentric cs>  
<projected cs> = PROJCS["<name>",  
<geographic cs>, <projection>, {<parameter>},]*
```

## Supported coordinate systems

```
<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

*Table 41. 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     |

## Supported coordinate systems

Table 41. Supported linear units (continued)

| Unit          | Conversion factor |
|---------------|-------------------|
| Chain (Sears) | 792/39.370147     |
| Yard (Indian) | 36/39.370141      |
| Yard (Sears)  | 36/39.370147      |
| Fathom        | 1.8288            |
| Nautical Mile | 1852.0            |

## Supported angular units

Table 42. Supported angular units

| Unit           | Conversion factor |
|----------------|-------------------|
| Radian         | 1.0               |
| Decimal Degree | $\pi/180$         |
| Decimal Minute | $(\pi/180)/60$    |
| Decimal Second | $(\pi/180)/36000$ |
| Gon            | $\pi/200$         |
| Grad           | $\pi/200$         |

## Supported spheroids

Table 43. Supported spheroids

| Name                   | Semi-major axis | Inverse flattening |
|------------------------|-----------------|--------------------|
| Airy                   | 6377563.396     | 299.3249646        |
| Modified Airy          | 6377340.189     | 299.3249646        |
| Australian             | 6378160         | 298.25             |
| Bessel                 | 6377397.155     | 299.1528128        |
| Modified Bessel        | 6377492.018     | 299.1528128        |
| Bessel (Namibia)       | 6377483.865     | 299.1528128        |
| Clarke 1866            | 6378206.4       | 294.9786982        |
| Clarke 1866 (Michigan) | 6378693.704     | 294.978684677      |
| Clarke 1880            | 6378249.145     | 293.465            |
| Clarke 1880 (Arc)      | 6378249.145     | 293.466307656      |
| Clarke 1880 (Benoit)   | 6378300.79      | 293.466234571      |
| Clarke 1880 (IGN)      | 6378249.2       | 293.46602          |
| Clarke 1880 (RGS)      | 6378249.145     | 293.465            |



Table 43. Supported spheroids (continued)

| Name                        | Semi-major axis | Inverse flattening |
|-----------------------------|-----------------|--------------------|
| Clarke 1880 (SGA)           | 6378249.2       | 293.46598          |
| Everest 1830                | 6377276.345     | 300.8017           |
| Everest 1975                | 6377301.243     | 300.8017           |
| Everest (Sarawak and Sabah) | 6377298.556     | 300.8017           |
| Modified Everest 1948       | 6377304.063     | 300.8017           |
| Fischer 1960                | 6378166         | 298.3              |
| Fischer 1968                | 6378150         | 298.3              |
| Modified Fischer (1960)     | 6378155         | 298.3              |
| GEM10C                      | 6378137         | 298.257222101      |
| GRS 1980                    | 6378137         | 298.257222101      |
| Hayford 1909                | 6378388         | 297.0              |
| Helmert 1906                | 6378200         | 298.3              |
| Hough                       | 6378270         | 297.0              |
| International 1909          | 6378388         | 297.0              |
| International 1924          | 6378388         | 297.0              |
| New International 1967      | 6378157.5       | 298.2496           |
| Krasovsky                   | 6378245         | 298.3              |
| Mercury 1960                | 6378166         | 298.3              |
| Modified Mercury 1968       | 6378150         | 298.3              |
| NWL9D                       | 6378145         | 298.25             |
| OSU_86F                     | 6378136.2       | 298.25722          |
| OSU_91A                     | 6378136.3       | 298.25722          |
| Plessis 1817                | 6376523         | 308.64             |
| South American 1969         | 6378160         | 298.25             |
| Southeast Asia              | 6378155         | 298.3              |
| Sphere (radius = 1.0)       | 1               | 0                  |
| Sphere (radius = 6371000 m) | 6371000         | 0                  |
| Sphere (radius = 6370997 m) | 6370997         | 0                  |
| Struve 1860                 | 6378297         | 294.73             |
| Walbeck                     | 6376896         | 302.78             |
| War Office                  | 6378300.583     | 296                |

## Supported coordinate systems

*Table 43. Supported spheroids (continued)*

| Name     | Semi-major axis | Inverse flattening |
|----------|-----------------|--------------------|
| WGS 1960 | 6378165         | 298.3              |
| WGS 1966 | 6378145         | 298.25             |
| WGS 1972 | 6378135         | 298.26             |
| WGS 1984 | 6378137         | 298.257223563      |

## Supported geodetic datums

*Table 44. Supported geodetic datums*

|                                  |                                 |
|----------------------------------|---------------------------------|
| Adindan                          | Lisbon                          |
| Afgooye                          | Loma Quintana                   |
| Agadez                           | Lome                            |
| Australian Geodetic Datum 1966   | Luzon 1911                      |
| Australian Geodetic Datum 1984   | Mahe 1971                       |
| Ain el Abd 1970                  | Makassar                        |
| Amersfoort                       | Malongo 1987                    |
| Aratu                            | Manoca                          |
| Arc 1950                         | Massawa                         |
| Arc 1960                         | Merchich                        |
| Ancienne Triangulation Francaise | Militar-Geographische Institute |
| Barbados                         | Mhast                           |
| Batavia                          | Minna                           |
| Beduaram                         | Monte Mario                     |
| Beijing 1954                     | M'poraloko                      |
| Reseau National Belge 1950       | NAD Michigan                    |
| Reseau National Belge 1972       | North American Datum 1927       |
| Bermuda 1957                     | North American Datum 1983       |
| Bern 1898                        | Nahrwan 1967                    |
| Bern 1938                        | Naparima 1972                   |
| Ancienne Triangulation Francaise | Militar-Geographische Institute |
| Barbados                         | Mhast                           |
| Batavia                          | Minna                           |
| Beduaram                         | Monte Mario                     |
| Beijing 1954                     | M'poraloko                      |

Table 44. Supported geodetic datums (continued)

|                                  |                                       |
|----------------------------------|---------------------------------------|
| Reseau National Belge 1950       | NAD Michigan                          |
| Reseau National Belge 1972       | North American Datum 1927             |
| Bermuda 1957                     | North American Datum 1983             |
| Bern 1898                        | Nahrwan 1967                          |
| Bern 1938                        | Naparima 1972                         |
| Ancienne Triangulation Francaise | Militar-Geographische Institute       |
| Barbados                         | Mhast                                 |
| Batavia                          | Minna                                 |
| Beduaram                         | Monte Mario                           |
| Beijing 1954                     | M'poraloko                            |
| Reseau National Belge 1950       | NAD Michigan                          |
| Reseau National Belge 1972       | North American Datum 1927             |
| Bermuda 1957                     | North American Datum 1983             |
| Bern 1898                        | Nahrwan 1967                          |
| Bern 1938                        | Naparima 1972                         |
| Bogota                           | Nord de Guerre                        |
| Bukit Rimpah                     | NGO 1948                              |
| Camacupa                         | Nord Sahara 1959                      |
| Campo Inchauspe                  | NSWC 9Z-2                             |
| Cape                             | Nouvelle Triangulation Francaise      |
| Carthage                         | New Zealand Geodetic Datum 1949       |
| Chua                             | OS (SN) 1980                          |
| Conakry 1905                     | OSGB 1936                             |
| Corrego Alegre                   | OSGB 1970 (SN)                        |
| Cote d'Ivoire                    | Padang 1884                           |
| Datum 73                         | Palestine 1923                        |
| Deir ez Zor                      | Pointe Noire                          |
| Deutsche Hauptdreiecksnetz       | Provisional South American Datum 1956 |
| Douala                           | Pulkovo 1942                          |
| European Datum 1950              | Qatar                                 |
| European Datum 1987              | Qatar 1948                            |
| Egypt 1907                       | Qornoq                                |
| European Reference System 1989   | RT38                                  |

## Supported coordinate systems

*Table 44. Supported geodetic datums (continued)*

|                                    |                                      |
|------------------------------------|--------------------------------------|
| Fahud                              | South American Datum 1969            |
| Gandajika 1970                     | Sapper Hill 1943                     |
| Garoua                             | Schwarzeck                           |
| Geocentric Datum of Australia 1994 | Segora                               |
| Guyane Francaise                   | Serindung                            |
| Herat North                        | Stockholm 1938                       |
| Hito XVIII 1963                    | Sudan                                |
| Hu Tzu                             | Shan Tananarive 1925                 |
| Hungarian Datum 1972               | Timbalai 1948                        |
| Indian 1954                        | TM65                                 |
| Indian 1975                        | TM75                                 |
| Indonesian Datum 1974              | Tokyo                                |
| Jamaica 1875                       | Trinidad 1903                        |
| Jamaica 1969                       | Trucial Coast 1948                   |
| Kalianpur                          | Voirol 1875                          |
| Kandawala                          | Voirol Unifie 1960                   |
| Kertau                             | WGS 1972                             |
| Kuwait Oil Company                 | WGS 1972 Transit Broadcast Ephemeris |
| La Canoa                           | WGS 1984                             |
| Lake                               | Yacare                               |
| Leigon                             | Yoff                                 |
| Liberia 1964                       | Zanderij                             |

## Supported prime meridians

*Table 45. Supported prime meridians*

| <b>Location</b> | <b>Coordinates</b> |
|-----------------|--------------------|
| 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    |

Table 45. Supported prime meridians (continued)

| Location  | Coordinates      |
|-----------|------------------|
| Madrid    | 3° 41' 16.58" W  |
| Paris     | 2° 20' 14.025" E |
| Rome      | 12° 27' 8.4" E   |
| Stockholm | 18° 3' 29" E     |

## Supported map projections

Table 46. 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 47. Conic projections

|                                 |  |
|---------------------------------|--|
| Albers conic equal-area         | Chamberlin trimetric                           |
| Bipolar oblique conformal conic | Two-point equidistant                          |
| Bonne                           | Hammer-Aitoff equal-area                       |
| Equidistant conic               | Van der Grinten I                              |
| Lambert conformal conic         | Miscellaneous                                  |
| Polyconic                       | Alaska series E                                |
| Simple conic                    | Alaska Grid (Modified-Stereographic by Snyder) |

## Supported coordinate systems

Table 48. 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.                                 |
| 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.   |

Table 48. Map projection parameters (continued)

| Parameter                | Description  |
|--------------------------|--|
| 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.   |

## Reference material



---

## Appendix A. Deprecated stored procedures

This topic outlines the deprecated stored procedures.

**Note:** Recommendation: write all new applications using the stored procedures defined in DB2 Spatial Extender Version 8 and update current applications to use the stored procedures defined in Version 8.

The deprecated stored procedures carried out the tasks summarized in the table below.

*Table 49. Deprecated stored procedures*

| Stored procedure name     | Stored procedure task   |
|---------------------------|---|
| db2gse.gse_enable_autogc  | Enabled a geocoder to automatically keep spatial columns synchronized with their corresponding attribute columns                    |
| db2gse.gse_enable_db      | Enabled a database to support spatial operations  |
| db2gse.gse_enable_idx     | Created an index for a spatial column   |
| db2gse.gse_enable_sref    | Created a spatial reference system  |
| db2gse.gse_export_shape   | Exported a layer and its associated table to a shape file   |
| db2gse.gse_disable_autogc | Disabled a geocoder so that it could not automatically keep spatial columns synchronized with their corresponding attribute columns |
| db2gse.gse_disable_db     | Disabled support for spatial operations in a database   |
| db2gse.gse_disable_sref   | Dropped a spatial reference system  |
| db2gse.gse_import_shape   | Imported a layer and its associated table from an ESRI_SDE transfer file  |
| db2gse.gse_register_gc    | Registered a geocoder other than the default geocoder   |
| db2gse.gse_register_layer | Registered a spatial column as a layer  |
| db2gse.gse_run_gc         | Running a geocoder in batch mode  |
| db2gse.gse_unregist_gc    | Unregistered a geocoder other than the default geocoder   |
| db2gse.gse_unregist_layer | Unregistered a layer  |

## Deprecated stored procedures

---

### db2gse.gse\_enable\_autogc

Use this stored procedure to:

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

#### Authorization:

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

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

#### Parameters:

*Table 50. Input parameters for the db2gse.gse\_enable\_autogc stored procedure.*

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

---

*Table 50. Input parameters for the db2gse.gse\_enable\_autogc stored procedure. (continued)*

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

## Deprecated stored procedures

Table 50. Input parameters for the `db2gse.gse_enable_autogc` stored procedure. (continued)

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

### Results:

Table 51. Output parameters for the `db2gse.gse_enable_autogc` stored procedure.

| Name                 | Data type                  | Description  |
|----------------------|----------------------------|--|
| <code>msgCode</code> | <code>INTEGER</code>       | Code associated with the messages that the caller of this stored procedure can return. |
| <code>msgText</code> | <code>VARCHAR(1024)</code> | Complete error message, as constructed at the server.                                  |

## `db2gse.gse_enable_db`

Use this stored procedure to supply a database with the resources that it needs to store spatial data and to support operations. These resources include spatial data types, a spatial index type, catalog tables and views, supplied functions, and other stored procedures. The external library and function name for this stored procedure is `db2gse.gse_enable_db`.

### Authorization:

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

### Results:

Table 52. Output parameters for the `db2gse.gse_enable_db` stored procedure.

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

**db2gse.gse\_enable\_idx**

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

**Authorization:**

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

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

**Parameters:**

*Table 53. Input parameters for the db2gse.gse\_enable\_idx stored procedure.*

| Name        | Data type    | Description  |
|-------------|--------------|--|
| layerSchema | VARCHAR(30)  | Name of the schema to which the table specified in the layerTable parameter belongs.<br><br>This parameter is nullable.<br><br>You must supply a value for this parameter. The parameter can be a NULL value.                    |
| layerTable  | VARCHAR(128) | Name of the table on which the index that you are creating is to be defined.<br><br>This parameter is not nullable.  |
| layerColumn | VARCHAR(128) | Name of the spatially enabled column that is to be searched with the aid of the index that you are creating.<br><br>This parameter is not nullable.  |
| indexName   | VARCHAR(128) | Name of the index that is to be created.<br><br>This parameter is not nullable.<br><br>Do not specify a schema name. DB2 Spatial Extender automatically assigns the index to the schema referenced by the layerSchema parameter. |
| gridSize1   | DOUBLE       | Number that indicates what the granularity of the finest index grid should be.<br><br>This parameter is not nullable.  |

## Deprecated stored procedures

Table 53. Input parameters for the `db2gse.gse_enable_idx` stored procedure. (continued)

| Name                   | Data type | Description   |
|------------------------|-----------|---|
| <code>gridSize2</code> | DOUBLE    | Number that denotes either (1) that there is to be no second grid for this index or (2) what the granularity of the second grid should be.<br><br>This parameter is nullable.<br><br>If there is to be no second grid, specify 0. If you want a second grid, it must be less granular than the grid denoted by <code>gridSize1</code> . |
| <code>gridSize3</code> | DOUBLE    | Number that denotes either (1) that there is to be no third grid for this index or (2) what the granularity of the third grid should be.<br><br>This parameter is nullable.<br><br>If there is to be no third grid, specify 0. If you want a third grid, it must be less granular than the grid denoted by <code>gridSize2</code> .     |

### Results:

Table 54. Output parameters for the `db2gse.gse_enable_idx` stored procedure.

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

## `db2gse.gse_enable_sref`

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

### Authorization:

None required.

### Parameters:

*Table 55. Input parameters for the db2gse.gse\_enable\_sref stored procedure.*

| Name    | Data type   | Description   |
|---------|-------------|---|
| srId    | INTEGER     | <p>A numeric identifier for the spatial reference system.</p> <p>This identifier must be unique within your spatially-enabled database.</p> <p>This parameter is not nullable.</p>            |
| srName  | VARCHAR(64) | <p>Short description of the spatial reference system.</p> <p>This description must be unique within your spatially-enabled database.</p> <p>This parameter is not nullable.</p>               |
| falsex  | DOUBLE      | <p>A number that, when subtracted from a negative X coordinate value, leaves a non-negative number (that is, a positive number or a zero).</p> <p>This parameter is not nullable.</p>         |
| falsey  | DOUBLE      | <p>A number that, when subtracted from a negative Y coordinate value, leaves a non-negative number (that is, a positive number or a zero).</p> <p>This parameter is not nullable.</p>         |
| xyunits | DOUBLE      | <p>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.</p> <p>This parameter is not nullable.</p> |
| falsez  | DOUBLE      | <p>A number that, when subtracted from a negative Z coordinate value, leaves a non-negative number (that is, a positive number or a zero).</p> <p>This parameter is not nullable.</p>         |
| zunits  | DOUBLE      | <p>A number that, when multiplied by a decimal Z coordinate, yields an integer that can be stored as a 32-bit data item.</p> <p>This parameter is not nullable.</p>                           |
| falsem  | DOUBLE      | <p>A number that, when subtracted from a negative measure, leaves a non-negative number (that is, a positive number or a zero).</p> <p>This parameter is not nullable.</p>                    |

## Deprecated stored procedures

Table 55. Input parameters for the `db2gse.gse_enable_sref` stored procedure. (continued)

| Name                | Data type | Description  |
|---------------------|-----------|--|
| <code>munits</code> | DOUBLE    | A number that, when multiplied by a decimal measure, yields an integer that can be stored as a 32-bit data item.<br><br>This parameter is not nullable.  |
| <code>sclId</code>  | INTEGER   | Numeric identifier of the coordinate system from which the spatial reference system is being derived. To find out what a coordinate system's numeric identifier is, consult the <code>DB2GSE.COORD_REF_SYS</code> catalog view.<br><br>This parameter is not nullable. |

### Results:

Table 56. Output parameters for the `db2gse.gse_enable_sref` stored procedure.

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

## `db2gse.gse_export_shape`

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

### Authorization:

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



### Parameters:

*Table 57. Input parameters for the db2gse.gse\_export\_shape stored procedure.*

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

### Results:

*Table 58. Output parameters for the db2gse.gse\_export\_shape stored procedure.*

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

### Restriction:

You can export only one layer at a time.

## Deprecated stored procedures

---

### db2gse.gse\_disable\_autogc

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

#### Authorization:

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

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

**Note:** For CONTROL and ALTER privileges, you must have DROPIN authority on the DB2GSE schema.

#### Parameters:

*Table 59. Input parameters for the db2gse.gse\_disable\_autogc stored procedure.*

| Name     | Data type | Description  |
|----------|-----------|--|
| operMode | SMALLINT  | Indicates whether the triggers are to be dropped or temporarily disabled.<br><br>Dropped triggers have no effect on SQL statements.<br><br>Temporarily disabled triggers can be re-created without having to re-specify previously set parameters.<br><br>This parameter is not nullable.<br><br>To drop triggers, use the GSE_AUTOGC_DROP macro. To temporarily disable them, use the GSE_AUTOGC_INVALIDATE macro. To find out what values are associated with these macros, consult the db2gse.h file. On AIX, this file is stored in the \$DB2INSTANCE/sqlib/include/ directory. On Windows NT, it is stored in the %DB2PATH%\include\ directory. |

---

Table 59. Input parameters for the `db2gse.gse_disable_autogc` stored procedure. (continued)

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

### Results:

Table 60. Output parameters for the `db2gse.gse_disable_autogc` stored procedure.

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

## `db2gse.gse_disable_db`

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

The purpose of this stored procedure is to help you resolve problems or issues that arise after you enable your database for spatial operations, but *before* you add any spatial table columns or data to it. For example, if, after you enable a database for spatial operations, if you decide to use DB2 Spatial Extender for

## Deprecated stored procedures

another database 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.

### Authorization:

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

### Results:

*Table 61. Output parameters for the db2gse.gse\_disable\_db stored procedure.*

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

---

## db2gse.gse\_disable\_sref

Use this stored procedure to drop a spatial reference system. When this stored procedure is processed, information about the spatial reference system is removed from the DB2GSE.SPATIAL\_REF\_SYS catalog view.

### Prerequisites:

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

### Authorization:

None required.

### Process:

*Table 62. Input parameter for the db2gse.gse\_disable\_sref stored procedure.*

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

### Results:

Table 63. Output parameters for the `db2gse.gse_disable_sref` stored procedure.

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

### `db2gse.gse_import_shape`

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

- If the shape file is targeted for an existing table that has a registered layer column, DB2 Spatial Extender will load the table with the file's data.
- If the shape file is targeted for a table that does not exist, DB2 Spatial Extender will create a table that has a spatial column, register this column as a layer, and load the layer and the table's other columns with the file's data.

When you import a set of ESRI shape representations, you receive at least two files. All the files have the same file name prefix, but different extensions. For example, the extensions of the two files that you always receive are `.shp` and `.shx`.

To receive the files for a set of shape representations, assign the name that the files have in common to the `fileName` parameter. Do not specify an extension. This way, you can be sure that all the files that you need—the `.shp` file, the `.shx` file, and any others that might be included—will be imported.

For example, suppose that a set of ESRI shape representations is stored in files called `Lakes.shp` and `Lakes.shx`. When importing these representations, you would assign only the name `Lakes` to the `fileName` parameter.

SDE transfer files have names but not extensions. Therefore, when importing an SDE transfer file, you assign its name, but no extension, to the `fileName` parameter.

### Authorization:

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

## Deprecated stored procedures

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

### Parameters:

*Table 64. Input parameters for the db2gse.gse\_import\_shape stored procedure.*

| Name          | Data type    | Description   |
|---------------|--------------|---|
| layerSchema   | VARCHAR(30)  | Name of the schema to which the table or view specified in the layerTable parameter belongs.<br><br>This parameter is nullable.<br><br>If you do not supply a value for the layerSchema parameter, it will default to the user ID under which the db2gse.gse_import_shape stored procedure is invoked.        |
| layerTable    | VARCHAR(128) | Name of the table into which the imported shape file is to be loaded.<br><br>This parameter is not nullable.  |
| layerColumn   | VARCHAR(30)  | Name of the column that has been registered as the layer into which shape data is to be loaded.<br><br>This parameter is not nullable.  |
| fileName      | VARCHAR(128) | Name of the shape file that is to be imported.<br><br>This parameter is not nullable.   |
| exceptionFile | VARCHAR(128) | Path and name of the file in which the shapes that could not be imported are stored. This is a new file that will be created when the db2gse.gse_import_shape stored procedure is run.<br><br>Assign a file name, but not an extension, to the exceptionFile parameter<br><br>This parameter is not nullable. |
| srId          | INTEGER      | Identifier of the spatial reference system to be used for the layer into which shape data is to be loaded.<br><br>This parameter is nullable.<br><br>If this identifier is not specified, the internal transformation will be set to the maximum resolution possible resolution for the shape file.           |

*Table 64. Input parameters for the db2gse.gse\_import\_shape stored procedure. (continued)*

| Name        | Data type | Description  |
|-------------|-----------|--|
| commitScope | INTEGER   | Number of records per checkpoint.<br><br>This parameter is I was nullable. |

### Results:

*Table 65. Output parameters for the db2gse.gse\_import\_shape stored procedure.*

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

---

## db2gse.gse\_register\_gc

Use this stored procedure to register a geocoder other than the default. To find out whether a geocoder is already registered, consult the DB2GSE.SPATIAL\_GEOCODER catalog view.

### Authorization:

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

### Parameters:

*Table 66. Input parameters for the db2gse.gse\_register\_gc stored procedure.*

| Name | Data type | Description  |
|------|-----------|--|
| gcId | INTEGER   | Numeric identifier of the geocoder that you are registering.<br><br>This identifier must be unique within the database.<br><br>This parameter is not nullable. |

## Deprecated stored procedures

Table 66. Input parameters for the *db2gse.gse\_register\_gc* stored procedure. (continued)

| Name           | Data type    | Description   |
|----------------|--------------|---|
| gcName         | VARCHAR(64)  | Short description of the geocoder that you are registering.<br><br>This description must be a unique character string within the database.<br><br>This parameter is not nullable.   |
| vendorName     | VARCHAR(64)  | Name of vendor that supplied the geocoder that you are registering.<br><br>This parameter is not nullable.  |
| primaryUDF     | VARCHAR(256) | Fully-qualified name of the geocoder that you are registering.<br><br>This parameter is not nullable.   |
| precisionLevel | INTEGER      | The degree to which source data must match corresponding reference data in order for the geocoder to process the source data successfully.<br><br>The precision level can range from 1 to 100 percent.<br><br>This parameter is not nullable. |
| vendorSpecific | VARCHAR(256) | Technical information provided by the vendor; for example, the path and name of a file that the vendor uses to set parameters.<br><br>This parameter is nullable.   |
| geoArea        | VARCHAR(256) | Geographical area to be geocoded.<br><br>This parameter is nullable.  |
| description    | VARCHAR(256) | Remarks provided by the vendor<br><br>This parameter is nullable.   |

### Results:

Table 67. Output parameters for the *db2gse.gse\_register\_gc* stored procedure.

| Name    | Data type | Description  |
|---------|-----------|--|
| msgCode | INTEGER   | Code associated with the messages that the caller of this stored procedure can return. |



*Table 67. Output parameters for the db2gse.gse\_register\_gc stored procedure. (continued)*

| Name    | Data type     | Description  |
|---------|---------------|--|
| msgText | VARCHAR(1024) | Complete error message, as constructed at the DB2 Spatial Extender server. |

---

### db2gse.gse\_register\_layer

Use this stored procedure to register a spatial column as a layer. When this stored procedure is processed, information about the layer being registered is added to the DB2GSE.GEOMETRY\_COLUMNS catalog view.

#### Authorization:

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

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

#### Parameters:

*Table 68. Input parameters for the db2gse.gse\_register\_layer stored procedure.*

| Name        | Data type    | Description  |
|-------------|--------------|--|
| layerSchema | INTEGER(30)  | Name of the schema to which the table or view specified in the layerTable parameter belongs.<br><br>This parameter is nullable.<br><br>If you do not supply a value for the layerSchema parameter, it will default to the user ID under which the db2gse.gse_register_layer stored procedure is invoked. |
| layerTable  | VARCHAR(128) | Name of the table or view that contains the column that is being registered as a layer.<br><br>This parameter is not nullable.   |

## Deprecated stored procedures

Table 68. Input parameters for the `db2gse.gse_register_layer` stored procedure. (continued)

| Name                       | Data type                 | Description   |
|----------------------------|---------------------------|---|
| <code>layerColumn</code>   | <code>VARCHAR(128)</code> | <p>Name of the column that is being registered as a layer. For a table, if the column does not exist, DB2 Spatial Extender will add it using the ALTER statement. For a view, the column must already exist.</p> <p>Only one column can be specified for the <code>layerColumn</code> parameter. Thus, when you register multiple columns of a table or view as layers, you must execute this stored procedure separately for each column.</p> <p>This parameter is not nullable.</p>   |
| <code>layerTypeName</code> | <code>VARCHAR(64)</code>  | <p>Data type of the column that is being registered as a layer. Only data types provided by DB2 Spatial Extender are accepted. You must specify the data type in uppercase; for example: <code>ST_POINT</code></p> <p>You do not need to specify a schema name because it is automatically added.</p> <p>This parameter is not nullable if the column is a table column that is to be created when this stored procedure is processed. Otherwise, if the column is an existing column within a table or view, this parameter is nullable.</p> |
| <code>srId</code>          | <code>INTEGER</code>      | <p>Identifier of the spatial reference system used for this layer.</p> <p>This parameter is not nullable for a table layer. DB2 Spatial Extender ignores this parameter when you register a view layer.</p>   |

Table 68. Input parameters for the `db2gse.gse_register_layer` stored procedure. (continued)

| Name                     | Data type                 | Description  |
|--------------------------|---------------------------|--|
| <code>geoSchema</code>   | <code>VARCHAR(30)</code>  | <p>The schema of the table that underlies the view to which the column belongs. This parameter applies when you register a view column as a layer.</p> <p>This parameter is nullable when you register a view column as a layer. DB2 Spatial Extender ignores this parameter when you register a table column as a layer.</p> <p>Views based on more than one base table or other views are not supported by this parameter.</p> <p>If you do not supply a value for the <code>geoSchema</code> parameter, it will default to the value of the <code>layerSchema</code> parameter.</p> |
| <code>geoTable</code>    | <code>VARCHAR(128)</code> | <p>The name of the table that underlies the view to which the column belongs. This parameter applies when you register a view column as a layer.</p> <p>Views based on more than one base table or other views are not supported by this parameter.</p> <p>This parameter is not nullable when you register a view column as a layer. DB2 Spatial Extender ignores this parameter when you register a table column as a layer.</p>   |
| <code>geoColumn</code>   | <code>VARCHAR(128)</code> | <p>The name of the table column that underlies this view column. This parameter applies when you register a view column as a layer.</p> <p>Views based on more than one base table or other views are not supported by this parameter.</p> <p>This parameter is not nullable when you register a view column as a layer. DB2 Spatial Extender ignores this parameter when you register a table column as a layer.</p>  |
| <code>nAttributes</code> | <code>SMALLINT</code>     | <p>Number of columns that contain the source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p>  |

## Deprecated stored procedures

Table 68. Input parameters for the `db2gse.gse_register_layer` stored procedure. (continued)

| Name                   | Data type                 | Description  |
|------------------------|---------------------------|--|
| <code>attr1Name</code> | <code>VARCHAR(128)</code> | <p>Name of the first column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>If you intend to use the default geocoder, you need to store street addresses in the <code>attr1Name</code> column.</p>                 |
| <code>attr2Name</code> | <code>VARCHAR(128)</code> | <p>Name of the second column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>If you intend to use the default geocoder, you need to store names of cities in the <code>attr2Name</code> column.</p>                 |
| <code>attr3Name</code> | <code>VARCHAR(128)</code> | <p>Name of the third column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>If you intend to use the default geocoder, you need to store names or abbreviations of states in the <code>attr3Name</code> column.</p> |
| <code>attr4Name</code> | <code>VARCHAR(128)</code> | <p>Name of the fourth column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>If you intend to use the default geocoder, you need to store zip codes in the <code>attr4Name</code> column.</p>                       |

*Table 68. Input parameters for the db2gse.gse\_register\_layer stored procedure. (continued)*

| Name      | Data type    | Description   |
|-----------|--------------|---|
| attr5Name | VARCHAR(128) | <p>Name of the fifth column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>The default geocoder ignores the Attr5Name column.</p>   |
| attr6Name | VARCHAR(128) | <p>Name of the sixth column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>The default geocoder ignores the Attr6Name column.</p>   |
| attr7Name | VARCHAR(128) | <p>Name of the seventh column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>The default geocoder ignores the Attr7Name column.</p> |
| attr8Name | VARCHAR(128) | <p>Name of the eighth column that contains source data that is to be geocoded for this layer.</p> <p>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.</p> <p>The default geocoder ignores the Attr8Name column.</p>  |

## Deprecated stored procedures

*Table 68. Input parameters for the db2gse.gse\_register\_layer stored procedure. (continued)*

| Name       | Data type    | Description   |
|------------|--------------|---|
| attr9Name  | VARCHAR(128) | Name of the ninth column that contains source data that is to be geocoded for this layer.<br><br>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.<br><br>The default geocoder ignores the Attr9Name column.  |
| attr10Name | VARCHAR(128) | Name of the tenth column that contains source data that is to be geocoded for this layer.<br><br>This parameter is nullable when you register a table column as a layer. DB2 Spatial Extender ignores this parameter when you register a view column as a layer.<br><br>The default geocoder ignores the Attr10Name column. |

### Results:

*Table 69. Output parameters for the db2gse.gse\_register\_layer stored procedure.*

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

### Restrictions:

This stored procedure does not work on the following table types:

- A = Alias
- H = Hierarchy table
- N = Nickname
- S = Summary table
- U = Typed table
- W = Typed view

The following restrictions also apply:

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

---

### db2gse.gse\_run\_gc

Use this stored procedure to run a geocoder in batch mode.

#### Authorization:

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

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

#### Parameters:

*Table 70. Input parameters for the db2gse.gse\_run\_gc stored procedure.*

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

---

## Deprecated stored procedures

Table 70. Input parameters for the `db2gse.gse_run_gc` stored procedure. (continued)

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

### Results:

Table 71. Output parameters for the `db2gse.gse_run_gc` stored procedure.

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

---

## `db2gse.gse_unregist_gc`

Use this stored procedure to unregister a geocoder other than the default geocoder. To find information about the geocoder that you want to unregister, consult the `DB2GSE.SPATIAL_GEOCODER` catalog view.

### Authorization:



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

### Parameters::

Table 72. Input parameter for the `db2gse.gse_unregist_gc` stored procedure.

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

### Results:

Table 73. Output parameters for the `db2gse.gse_unregist_gc` stored procedure.

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

---

## db2gse.gse\_unregist\_layer

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

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

When address data in a table row is geocoded, the resulting spatial data is placed in the same row. Therefore, if the row is deleted, the address data and spatial data are deleted at the same time. Triggers do not delete the spatial data. When the stored procedure is processed, information about the layer is removed from the DB2GSE.GEOMETRY\_COLUMNS catalog view.

### Authorization:

## Deprecated stored procedures

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

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

### Parameters:

*Table 74. Input parameters for the db2gse.gse\_unregister\_layer stored procedure.*

| Name        | Data type    | Description  |
|-------------|--------------|--|
| layerSchema | VARCHAR(30)  | Name of the schema to which the table specified in the layerTable parameter belongs.<br><br>This parameter is nullable.<br><br>If you do not supply a value for the layerSchema parameter, it will default to the user ID under which the db2gse.gse_unregister_layer stored procedure is invoked.<br><br>You must specify in uppercase any schema name, table name, view name, column name, or layer name that you assign to a parameter. |
| layerTable  | VARCHAR(128) | Name of the table that contains the column specified in the layerColumn parameter.<br><br>This parameter is not nullable.  |
| layerColumn | VARCHAR(128) | Name of the spatial column that is defined as the layer that you want to unregister.<br><br>This parameter is not nullable.<br><br>Only one layer can be specified for the layerColumn parameter. Thus, when you unregister multiple layers in a table or view, you must execute this stored procedure separately for each layer.  |

### Results:

*Table 75. Output parameters for the db2gse.gse\_unregister\_layer stored procedure.*

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

### Restriction::

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



---

## Appendix B. Deprecated catalog views

This topic outlines the deprecated catalog views.

**Note:** Recommendation: Develop all new applications with the views defined in DB2 Spatial Extender Version 8. All current applications should also be updated to use the views defined in Version 8. Applications which reference the undocumented underlying catalog tables defined in Version 7 will no longer work after migrating to Version 8 and should be modified to use the documented Version 8 catalog views.

---

### DB2GSE.COORD\_REF\_SYS

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

Table 76. Columns in the DB2GSE.COORD\_REF\_SYS catalog view

| Name      | Data Type     | Nullable? | Content  |
|-----------|---------------|-----------|--|
| CSID      | INTEGER       | Yes       | Unique numeric identifier for this coordinate system. If the coordinate system was created using the V8 administration interface, then no CSID will be recorded and null is used instead |
| CS_NAME   | VARCHAR(64)   | No        | Name of this coordinate system.  |
| AUTH_NAME | VARCHAR(256)  | Yes       | Name of the organization that compiled this coordinate system adheres to; for example, the European Petroleum Survey Group (EPSG).   |
| AUTH_SRID | INTEGER       | Yes       | A numeric identifier assigned to this coordinate system by the organization specified in the AUTH_NAME column.   |
| DESC      | VARCHAR(256)  | Yes       | Description of this coordinate system.   |
| SRTEXT    | VARCHAR(2048) | No        | Annotation text for this coordinate system.  |

## Deprecated catalog views

### DB2GSE.GEOMETRY\_COLUMNS

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

Table 77. Columns in the DB2GSE.GEOMETRY\_COLUMNS catalog view

| Name          | Data Type    | Nullable? | Content  |
|---------------|--------------|-----------|--|
| LAYER_CATALOG | VARCHAR(30)  | Yes       | NULL.<br><br>There is no concept of LAYER_CATALOG in DB2 Spatial Extender.   |
| LAYER_SCHEMA  | VARCHAR(30)  | No        | Schema of the table or view that contains the column that was registered as this layer.  |
| LAYER_TABLE   | VARCHAR(128) | No        | Name of the table or view that contains the column that was registered as this layer.  |
| LAYER_COLUMN  | VARCHAR(128) | No        | Name of the column that was registered as this layer.  |
| GEOMETRY_TYPE | INTEGER      | Yes       | Data type of the column that was registered as this layer. If the column has a user-defined subtype of any of the geometry types defined by the spatial extender, then this value will be null |
| SRID          | INTEGER      | No        | Identifier of the spatial reference system used for the values in the column that was registered as this layer.  |
| STORAGE_TYPE  | INTEGER      | Yes       | NULL.  |

### DB2GSE.SPATIAL\_GEOCODER

Available geocoders are registered in a catalog table. Selected columns from this table comprise the DB2GSE.SPATIAL\_GEOCODER catalog view, which is described in the following table.

Table 78. Columns in the DB2GSE.SPATIAL\_GEOCODER catalog view

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

Table 78. Columns in the DB2GSE.SPATIAL\_GEOCODER catalog view (continued)

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

### DB2GSE.SPATIAL\_REF\_SYS

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

Table 79. 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.                        |
| 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). |

## Deprecated catalog views

Table 79. Columns in the DB2GSE.SPATIAL\_REF\_SYS catalog view (continued)

| Name    | Data Type | Nullable? | Content   |
|---------|-----------|-----------|---|
| 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.                                |



## Appendix C. Deprecated spatial functions

This topic outlines the functions have been deprecated. The table below lists all the deprecated spatial functions and the new replacement functions for Version 8.

*Table 80. The deprecated functions and their new counterparts.*

| Deprecated function | New Function       |
|---------------------|--------------------|
| AsShape             | ST_AsShape         |
| EnvelopesIntersect  | ST_EnvIntersects   |
| GeometryFromShape   | ST_Geometry        |
| Is3D                | ST_Is3D            |
| IsMeasured          | ST_IsMeasured      |
| LineFromShape       | ST_LineString      |
| LocateAlong         | ST_FindMeasure     |
| LocateBetween       | ST_MeasureBetween  |
| M                   | ST_M               |
| MLine FromShape     | ST_MultiLineString |
| MPointFromShape     | ST_MultiPoint      |
| MPolyFromShape      | ST_MultiPolygon    |
| PointFromShape      | ST_Point           |
| PolyFromShape       | ST_Polygon         |
| ShapeToSQL          | ST_Geometry        |
| ST_GeomFromText     | ST_Geometry        |
| ST_GeomFromWKB      | ST_Geometry        |
| ST_LineFromText     | ST_LineString      |
| ST_LineFromWKB      | ST_LineString      |
| ST_MLineFromText    | ST_MultiLineString |
| ST_MLineFromWKB     | ST_MultiLineString |
| ST_MPointFromText   | ST_MultiPoint      |
| ST_MPointFromWKB    | ST_MultiPoint      |
| ST_MPolyFromText    | ST_MultiPolygon    |
| ST_MPolyFromWKB     | ST_MultiPolygon    |
| ST_OrderingEquals   |                    |

## Deprecated spatial functions

Table 80. The deprecated functions and their new counterparts. (continued)

|   |                                    |
|---|------------------------------------|
| ST_Point(Double, Double, db2gse.coordref)     | ST_Point(Double, Double, Integer)  |
| ST_PointFromText                              | ST_Point                           |
| ST_PolyFromText                               | ST_Polygon                         |
| ST_PolyFromWKB                                | ST_Polygon                         |
| ST_Transform(Double, Double, db2gse.coordref) | ST_Transform(ST_Geometry, Integer) |
| ST_SymmetricDiff                              | ST_SymDifference                   |
| Z   | ST_Z                               |

---

### AsShape

**Purpose:**

AsShape takes a geometry object and returns a BLOB.

**Format:**

db2gse.AsShape(g db2gse.ST\_Geometry)

**Results:**

BLOB(1m)

---

### EnvelopesIntersect

**Purpose:**

EnvelopesIntersect returns 1 if the envelopes of two geometries intersect; otherwise it returns 0.

**Format:**

db2gse.EnvelopesIntersect(g1 db2gse.ST\_Geometry, g2 db2gse.ST\_Geometry)

**Results:**

Integer

---

## GeometryFromShape

**Purpose:**

GeometryFromShape takes a shape and a spatial reference system identifier to return a geometry object.

**Format:**

db2gse.GeometryFromShape(ShapeGeometry Blob(1M), SRID db2gse.coordref)

**Results:**

db2gse.ST\_Geometry

---

## Is3d

**Purpose:**

Is3d takes a geometry object and returns 1 if the object has 3D coordinates; otherwise, it returns 0.

**Format:**

db2gse.Is3d(g db2gse.ST\_Geometry)

**Results:**

Integer

---

## IsMeasured

**Purpose:**

IsMeasured takes a geometry object and returns 1 if the object has measures; otherwise it returns 0.

**Format:**

db2gse.IsMeasured(g db2gse.ST\_Geometry)

**Results:**

Integer

## Deprecated spatial functions

---

### LineFromShape

**Purpose:**

LineFromShape takes a shape of type point and a spatial reference system identifier and returns a linestring.

**Format:**

```
db2gse.Line FromShape(ShapeLineString Blob(1M), SRID db2gse.coordref)
```

**Results:**

```
db2gse.ST_LineString
```

---

### LocateAlong

**Purpose:**

LocateAlong takes a geometry object and a measure to return as a multipoint the set of points found at the measure.

If LocateAlong is given a multipoint and a measure as input, and if the multipoint does not include this measure, then LocateAlong returns POINT EMPTY.

**Format:**

```
db2gse.LocateAlong(g db2gse.ST_Geometry, measure Double)
```

**Results:**

```
db2gse.ST_Geometry
```

---

### LocateBetween

**Purpose:**

LocateBetween takes a geometry object and two measure locations and returns a geometry that represents the set of disconnected paths between the two measure locations.

**Format:**

db2gse.LocateBetween(g db2gse.ST\_Geometry, measure Double, measure Double)

**Results:**

db2gse.ST\_Geometry

---

## M

**Purpose:**

M takes a point and returns its measure.

**Format:**

db2gse.M(p db2gse.ST\_Point)

**Results:**

Double

---

## MLine FromShape

**Purpose:**

MLine FromShape takes a shape of type multilinestring and a spatial reference system identifier and returns a multilinestring.

**Format:**

db2gse.MLineFromShape(ShapeMultiLineString Blob(1M), SRID db2gse.coordref)

**Results:**

db2gse.ST\_MultiLineString

---

## MPointFromShape

**Purpose:**

## Deprecated spatial functions

MPointFromShape takes a shape of type multipoint and a spatial reference system identifier to return a multipoint.

**Format:**

```
db2gse.MPointFromShape(ShapeMultiPoint BLOB(1M), SRID db2gse.coordref)
```

**Results:**

```
db2gse.ST_MultiPoint
```

---

## MPolyFromShape

**Purpose:**

MPolyFromShape takes a shape of type multipolygon and a spatial reference system identifier to return a multipolygon.

**Format:**

```
db2gse.MPolyFromShape(ShapeMultiPolygon Blob(1m), SRID db2gse.coordref)
```

**Results:**

```
db2gse.ST_MultiPolygon
```

---

## PointFromShape

**Purpose:**

PointFromShape takes a shape of type point and a spatial reference system identifier to return a point.

**Format:**

```
db2gse.PointFromShape(db2gse.ShapePoint blob(1M), SRID db2gse.coordref)
```

**Results:**

```
db2gse.ST_Point
```

---

## PolyFromShape

**Purpose:**

PolyFromShape takes a shape of type polygon and a spatial reference system identifier to return a polygon.

**Format:**

db2gse.PolyFromShape (ShapePolygon Blob(1M), SRID db2gse.coordref)

**Results:**

db2gse.ST\_Polygon

---

### ShapeToSQL

**Purpose:**

ShapeToSQL constructs a db2gse.ST\_Geometry value given its shape representation. The SRID value of 0 is automatically used.

**Format:**

db2gse.ShapeToSQL(ShapeGeometry blob(1M))

**Results:**

db2gse.ST\_Geometry

---

### ST\_GeomFromText

**Purpose:**

ST\_GeomFromText takes a well-known text representation and a spatial reference system identifier and returns a geometry object.

**Format:**

db2gse.ST\_GeomFromText(geometryTaggedText Varchar(4000), SRID db2gse.coordref)

**Results:**

db2gse.ST\_Geometry

## Deprecated spatial functions

---

### ST\_GeomFromWKB

**Purpose:**

ST\_GeomFromWKB takes a well-known binary representation and a spatial reference system identifier and returns a geometry object.

**Format:**

db2gse.ST\_GeomFromWKB(WKBGeometry Blob(1M), SRID db2gse.coordref)

**Results:**

db2gse.ST\_Geometry

---

### ST\_LineFromText

**Purpose:**

ST\_LineFromText takes a well-known text representation of type linestring and a spatial reference system identifier and returns a linestring.

**Format:**

db2gse.ST\_LineFromText(lineStringTaggedText Varchar(4000), SRID db2gse.coordref)

**Results:**

db2gse.ST\_LineString

---

### ST\_LineFromWKB

**Purpose:**

ST\_LineFromWKB takes a well-known binary representation of the type linestring and a spatial reference system identifier, and returns a linestring.

**Format:**

db2gse.ST\_LineFromWKB(WKBLineString Blob(1M), SRID db2gse.coordref)

**Results:**



db2gse.ST\_LineString

---

## ST\_MLineFromText

**Purpose:**

ST\_MLineFromText takes a well-known text representation of type multilinestring and a spatial reference system identifier and returns a multilinestring.

**Format:**

db2gse.ST\_MLineFromText(multiLineStringTaggedText String, SRID db2gse.coordref)

**Results:**

db2gse.ST\_MultiLineString

---

## ST\_MLineFromWKB

**Purpose:**

ST\_MLineFromWKB takes a well-known binary representation of type multilinestring and a spatial reference system identifier and returns a multilinestring.

**Format:**

db2gse.ST\_MLineFromWKB(WKBMultiLineString Blob(1M), SRID db2gse.coordref)

**Results:**

db2gse.ST\_MultiLineString

---

## ST\_MPointFromText

**Purpose:**

ST\_MPointFromText takes a well-known text representation of type multipoint and a spatial reference system identifier and returns a multipoint.

**Format:**

## Deprecated spatial functions

db2gse.ST\_MPointFromText(multiPointTaggedText Varchar(4000), SRID db2gse.coordref)

**Results:**

db2gse.ST\_MultiPoint

---

## ST\_MPointFromWKB

**Purpose:**

ST\_MPointFromWKB takes a well-known binary representation of type multipoint and a spatial reference system identifier to return a multipoint.

**Format:**

db2gse.ST\_MPointFromWKB(WKBMultiPoint Blob(1M), SRID db2gse.coordref)

**Results:**

db2gse.ST\_MultiPoint

---

## ST\_MPolyFromText

**Purpose:**

ST\_MPolyFromText takes a well-known text representation of type multipolygon and a spatial reference system identifier and returns a multipolygon.

This function cannot take as input a multipolygon that contains multiple polygons with the same coordinates.

**Format:**

db2gse.ST\_MPolyFromText(multiPolygonTaggedText Varchar(4000), SRID db2gse.coordref)

**Results:**

db2gse.ST\_MultiPolygon

---

## ST\_MPolyFromWKB

**Purpose:**

ST\_MPolyFromWKB takes a well-known binary representation of type multipolygon and a spatial reference system identifier and returns a multipolygon.

**Format:**

db2gse.ST\_MPolyFromWKB(WKBMultiPolygon Blob(1M), SRID db2gse.coordref)

**Results:**

db2gse.ST\_MultiPolygon

---

## ST\_OrderingEquals

**Purpose:**

ST\_OrderingEquals compares two geometries and returns 1 (TRUE) if the geometries are equal and the coordinates are in the same order; otherwise it returns 0 (FALSE).

**Format:**

db2gse.ST\_OrderingEquals(g1 db2gse.ST\_Geometry, g2 db2gse.ST\_Geometry)

**Results:**

Integer

---

## ST\_Point

**Purpose:**

ST\_Point returns an ST\_Point, given an x-coordinate, y-coordinate, and spatial reference.

**Format:**

db2gse.ST\_Point(X Double, Y Double, SRID db2gse.coordref)

## Deprecated spatial functions

### Results:

db2gse.ST\_Point

---

## ST\_PointFromText

### Purpose:

ST\_PointFromText takes a well-known text representation of type point and a spatial reference system identifier and returns a point.

### Format:

```
db2gse.ST_PointFromText(pointTaggedText Varchar(4000), SRID  
db2gse.coordref)
```

### Results:

db2gse.ST\_Point

---

## ST\_PolyFromText

### Purpose:

ST\_PolyFromText takes a well-known text representation of type polygon and a spatial reference system identifier and returns a polygon.

### Format:

```
db2gse.ST_PolyFromText(polygonTaggedText Varchar(4000), SRID  
db2gse.coordref)
```

### Results:

db2gse.ST\_Polygon

---

## ST\_PolyFromWKB

### Purpose:

ST\_PolyFromWKB takes a well-known binary representation of type polygon and a spatial reference system identifier to return a polygon.

### Format:

```
db2gse.ST_PolyFromWKB(WKBPolygon Blob(1M), SRID db2gse.coordref)
```

**Results:**

```
db2gse.ST_Polygon
```

---

**ST\_Transform****Purpose:**

ST\_Transform associates a geometry to a spatial reference system other than the spatial reference system to which the geometry is currently associated.

**Format:**

```
db2gse.ST_Transform(g db2gse.ST_Geometry, SRID db2gse.coordref)
```

**Results:**

```
db2gse.ST_Geometry
```

---

**ST\_SymmetricDiff****Purpose:**

ST\_SymmetricDiff takes two geometry objects and returns a geometry object that is the symmetrical difference of the source objects.

The ST\_SymmetricDiff function returns the symmetric difference (the Boolean logical XOR of space) of two intersecting geometries that have the same dimension. If these geometries are equal, ST\_SymmetricDiff returns an empty geometry. If they are not equal, then a portion of one or both of them will lie outside the area of intersection. ST\_SymmetricDiff returns the non-intersection portion or portions as a collection; for example, as a multipolygon.

If ST\_SymmetricDiff is given geometries of different dimensions as input, it returns a null.

**Format:**

```
db2gse.ST_SymmetricDiff(g1 db2gse.ST_Geometry, g2 db2gse.ST_Geometry)
```

**Results:**

## Deprecated spatial functions

db2gse.ST\_Geometry

---

### Z

#### **Purpose:**

Z takes a point and returns its Z coordinate.

#### **Format:**

db2gse.Z(p db2gse.ST\_Point)

#### **Results:**

Double

#### **Related reference:**

- “ST\_AsShape” on page 300
- “ST\_MeasureBetween, ST\_LocateBetween” on page 397
- “ST\_EnvIntersects” on page 333
- “ST\_FindMeasure or ST\_LocateAlong” on page 341
- “ST\_Geometry” on page 351
- “ST\_Is3d” on page 366
- “ST\_LineString” on page 381
- “ST\_M” on page 384
- “ST\_MultiLineString” on page 418
- “ST\_MultiPoint” on page 420
- “ST\_MultiPolygon” on page 422
- “ST\_Point” on page 437
- “ST\_Polygon” on page 449
- “ST\_SymDifference” on page 460
- “ST\_Transform” on page 473
- “ST\_Z” on page 485

---

## Notices

### Index entries

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country/region or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country/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 and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product, and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licenseses 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  
Office of the Lab Director  
8200 Warden Avenue  
Markham, Ontario  
L6G 1C7  
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 may contain 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.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. *\_enter the year or years\_*. All rights reserved.

---

## Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both, and have been used in at least one of the documents in the DB2 UDB documentation library.

|   |                  |
|---|------------------|
| ACF/VTAM  | LAN Distance     |
| AISPO   | MVS              |
| AIX   | MVS/ESA          |
| AIXwindows                                      | MVS/XA           |
| AnyNet  | Net.Data         |
| APPN  | NetView          |
| AS/400  | OS/390           |
| BookManager                                     | OS/400           |
| C Set++   | PowerPC          |
| C/370   | pSeries          |
| CICS  | QBIC             |
| Database 2                                      | QMF              |
| DataHub   | RACF             |
| DataJoiner                                      | RISC System/6000 |
| DataPropagator                                  | RS/6000          |
| DataRefresher                                   | S/370            |
| DB2   | SP               |
| DB2 Connect                                     | SQL/400          |
| DB2 Extenders                                   | SQL/DS           |
| DB2 OLAP Server                                 | System/370       |
| DB2 Universal Database                          | System/390       |
| Distributed Relational<br>Database Architecture | SystemView       |
| DRDA  | Tivoli           |
| eServer   | VisualAge        |
| Extended Services                               | VM/ESA           |
| FFST  | VSE/ESA          |
| First Failure Support Technology                | VTAM             |
| IBM   | WebExplorer      |
| IMS   | WebSphere        |
| IMS/ESA   | WIN-OS/2         |
| iSeries   | z/OS             |
|   | zSeries          |

The following terms are trademarks or registered trademarks of other companies and have been used in at least one of the documents in the DB2 UDB documentation library:

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

related links



---

# Index

## A

APP\_CTL\_HEAP\_SZ parameter, tuning 53  
APPLHEAPSZ configuration parameter tuning 53  
application control heap size configuration parameter 53  
application heap size parameter (APPLHEAPSZ) 53  
applications  
  sample program 134  
  spatial 131  
  spatial applications including header files 131  
  Spatial Extender calling stored procedures 132  
ArcExplorer  
  downloading 45  
  using as interface 119  
AsShape, deprecated spatial function 551

## C

catalog views  
  ST\_COORDINATE\_SYSTEMS 229  
  ST\_GEOCODER\_PARAMETERS 232  
  ST\_GEOCODERS 233  
  ST\_GEOCODING 234  
  ST\_GEOCODING\_PARAMETERS 236  
  ST\_GEOMETRY\_COLUMNS 231  
  ST\_SIZINGS 238  
  ST\_SPATIAL\_REFERENCE\_SYSTEMS 238  
  ST\_UNITS\_OF\_MEASURE 242  
command line processor (CLP) messages 150  
configuration parameters  
  spatial applications tuning 53  
  values 53  
Control Center messages 153  
COORD\_REF\_SYS, spatial deprecated catalog view 547

coordinate systems 507  
  angular units 507  
  creating 73  
  description 65  
  geodetic datums 507  
  linear units 507  
  map projections 507  
  prime meridians 507  
  selecting 73  
  spheroids 507  
ST\_COORDINATE\_SYSTEMS catalog view 229  
ST\_SPATIAL\_REFERENCE\_SYSTEMS catalog view 238

## D

data formats  
  Geography Markup Language (GML) 505  
  shape representation 505  
  well-known binary (WKB) representation 503  
  well-known text (WKT) representation 497  
data maps  
  Spatial Extender 47  
database  
  enabling spatial operations 60  
database configuration parameters  
  spatial applications  
    APP\_CTL\_HEAP\_SZ parameter 53  
    APPLHEAPSZ parameter 53  
    LOGFILSZ parameter 53  
    LOGPRIMARY parameter 53  
    LOGSECOND parameter 53  
    tuning 53  
database manager configuration parameter, tuning for spatial applications 53  
databases  
  configuring for spatial applications 53  
  enabling for spatial operations discussion 59  
  setting up for spatial applications 53  
db2diag.log 155  
db2se commands 123

db2trc command 154  
downloading ArcExplorer 45

## E

enabling  
  spatial operations 60  
EnvelopesIntersect, deprecated spatial function 551  
exporting data  
  SDE transfer files 95  
  shape files 94

## F

function messages 148

## G

geocoders  
  description 96  
  reference data 45  
  registering 62  
  running in batch mode 102  
  ST\_GEOCODER\_PARAMETERS catalog view 232  
  ST\_GEOCODERS catalog view 233  
  ST\_GEOCODING catalog view 234  
  ST\_GEOCODING\_PARAMETERS catalog view 236  
  ST\_SIZINGS catalog view 238  
geocoding  
  batch mode 102  
geographic features  
  description 3  
  represented by data 4  
Geographic Markup Language (GML), data format 505  
geometries  
  discussion 11  
  properties 13  
GEOMETRY\_COLUMNS, spatial deprecated catalog view 547  
GeometryFromShape, deprecated spatial function 551  
grid indexes  
  creating 106  
gse\_disable\_autogc stored procedure 182

# Index

`gse_disable_autogc`, deprecated  
  spatial stored procedure 519  
`gse_disable_db` stored  
  procedure 184  
`gse_disable_db`, deprecated spatial  
  stored procedure 519  
`gse_disable_sref` stored  
  procedure 187  
`gse_disable_sref`, spatial deprecated  
  stored procedure 519  
`gse_enable_autogc` stored  
  procedure 189  
`gse_enable_autogc`, deprecated  
  spatial stored procedure 519  
`gse_enable_db` stored  
  procedure 191  
`gse_enable_db`, deprecated spatial  
  stored procedure 519  
`gse_enable_idx`, deprecated spatial  
  stored procedure 519  
`gse_enable_sref` stored  
  procedure 174  
`gse_enable_sref`, deprecated spatial  
  stored procedure 519  
`GSE_export_sde` stored  
  procedure 160  
`gse_export_shape` 194  
`gse_import_sde` stored  
  procedure 162  
`GSE_import_sde` stored  
  procedure 162  
`gse_import_shape` stored  
  procedure 198  
`gse_import_shape`, deprecated  
  spatial stored procedure 519  
`gse_register_gc` stored  
  procedure 207  
`gse_register_gc`, deprecated spatial  
  stored procedure 519  
`gse_register_layer` stored  
  procedure 212  
`gse_register_layer`, deprecated spatial  
  stored procedure 519  
`gse_run_gc` stored procedure 217  
`gse_run_gc`, deprecated spatial  
  stored procedure 519  
`gse_unregist_gc` stored  
  procedure 224  
`gse_unregist_gc`, deprecated spatial  
  stored procedure 519  
`gse_unregist_layer` stored  
  procedure 226  
`gse_unregist_layer`, deprecated  
  spatial stored procedures 519

## H

h header file 131  
header file, including DB2 Spatial  
  Extender 131

## I

importing  
  SDE transfer data 92  
  shape data 91  
Index Advisor  
  using 111  
indexes  
  creating a spatial grid 106  
installing  
  DB2 Spatial Extender  
    AIX 32  
    hardware and software  
      requirements 28  
    HP-UX 34  
    Linux and Linux 390 38  
    Solaris Operating  
      Environment 36  
    verifying 42  
    Windows 30

instances 40

interfaces

  DB2 Spatial Extender 17

Is3d, deprecated spatial  
  function 551

IsMeasured, deprecated spatial  
  function 551

## L

LineFromShape, deprecated spatial  
  function 551

LocateAlong, deprecated spatial  
  function 551

LocateBetween, deprecated spatial  
  function 551

LOGFILSZ configuration  
  parameter 53

LOGPRIMARY configuration  
  parameter 53

LOGSECOND configuration  
  parameter  
  tuning 53

## M

M, deprecated spatial function 551

messages

  functions 148

  migration information 150

  shape information 150

  Spatial Extender

    CLP 150

messages (*continued*)

  Spatial Extender (*continued*)

    parts of 143

    stored procedures 146

migration

  Spatial Extender 49

MLineFromShape, deprecated spatial  
  function 551

MPointFromShape, deprecated spatial  
  function 551

MPolyFromShape, deprecated spatial  
  function 551

## P

PointFromShape, deprecated spatial  
  function 551

programming considerations  
  Spatial Extender sample  
  program 131

## Q

queries

  spatial functions to perform 119

  spatial indexes, exploiting 121

  spatial, interfaces to submit 119

## R

reference data

  DB2 Spatial Extender 61

    setting up access 61

registering

  geocoders 62

  spatial columns 86

## S

sample programs 134

settings

  automatic geocoding 101

  DB2 Spatial Extender 27

  geocoding operation 98

shape representation, data

  format 505

ShapeToSQL, deprecated spatial  
  function 551

spatial applications

  including header files 131

  stored procedures

    calling from applications 132

spatial catalog views, deprecated

  COORD\_REF\_SYS 547

  GEOMETRY\_COLUMNS 547

  SPATIAL\_GEOCODER 547

  SPATIAL\_REF\_SYS 547

spatial columns

  creating 86

  using views to access 118

- spatial data
  - columns 83
  - data types 83
  - description 4
  - exporting 89
  - importing 89
  - retrieving and analyzing
    - exploiting indexes 121
    - functions 119
    - interfaces 119
  - ST\_GEOMETRY\_
    - COLUMNS 231
  - using 8
- Spatial Extender
  - enabling 60
  - installation 38
  - reference data 61
    - setting up access 61
- spatial functions
  - associated data types 285
  - categorized by operations
    - performed 119
  - comparisons 252
  - considerations 285
  - converting geometries 243
  - deprecated 551
  - generating new geometries 273
  - MBR aggregate 291
  - miscellaneous 282
  - properties of geometries 266
  - ST\_AppendPoint 292
  - ST\_Area 294
  - ST\_AsBinary 297
  - ST\_AsGML 298
  - ST\_AsShape 300
  - ST\_AsText 301
  - ST\_Boundary 302
  - ST\_Buffer 304
  - ST\_Centroid 307
  - ST\_ChangePoint 308
  - ST\_Contains 310
  - ST\_ConvexHull 312
  - ST\_CoordDim 314
  - ST\_Crosses 315
  - ST\_Difference 317
  - ST\_Dimension 319
  - ST\_Disjoint 320
  - ST\_Distance 327
  - ST\_Edge\_GC\_USA 322
  - ST\_Endpoint 331
  - ST\_Envelope 332
  - ST\_EnvIntersects 333
  - ST\_EqualCoordsys 335
  - ST\_Equals 336
  - ST\_EqualSRS 338
- spatial functions (*continued*)
  - ST\_ExteriorRing 339
  - ST\_FindMeasure
    - ST\_LocateAlong 341
  - ST\_Generalize 343
  - ST\_GeomCollection 345
  - ST\_GeomCollFromTxt 347
  - ST\_GeomCollFromWKB 349
  - ST\_Geometry 351
  - ST\_GeometryN 353
  - ST\_GeometryType 354
  - ST\_GeomFromText 355
  - ST\_GeomFromWKB 357
  - ST\_GetIndexParms 358
  - ST\_InteriorRingN 361
  - ST\_Intersection 363
  - ST\_Intersects 364
  - ST\_Is3d 366
  - ST\_IsClosed 368
  - ST\_IsEmpty 370
  - ST\_IsMeasured 371
  - ST\_IsRing 372
  - ST\_IsSimple 373
  - ST\_IsValid 375
  - ST\_Length 376
  - ST\_LineFromText 378
  - ST\_LineFromWKB 379
  - ST\_LineString 381
  - ST\_LineStringN 383
  - ST\_LocateAlong
    - ST\_FindMeasure 341
  - ST\_LocateBetween
    - ST\_MeasureBetween 397
  - ST\_M 384
  - ST\_MaxM 386
  - ST\_MaxX 388
  - ST\_MaxY 390
  - ST\_MaxZ 392
  - ST\_MBR 393
  - ST\_MBRIntersects 395
  - ST\_MeasureBetween
    - ST\_LocateBetween 397
  - ST\_MidPoint 398
  - ST\_MinM 400
  - ST\_MinX 401
  - ST\_MinY 403
  - ST\_MinZ 405
  - ST\_MLineFromText 407
  - ST\_MLineFromWKB 408
  - ST\_MPointFromText 411
  - ST\_MPointFromWKB 412
  - ST\_MPolyFromText 414
  - ST\_MPolyFromWKB 416
  - ST\_MultiLineString 418
  - ST\_MultiPoint 420
- spatial functions (*continued*)
  - ST\_MultiPolygon 422
  - ST\_NumGeometries 424
  - ST\_NumInteriorRing 425
  - ST\_NumLineStrings 427
  - ST\_NumPoints 428
  - ST\_NumPolygons 429
  - ST\_Overlaps 430
  - ST\_Perimeter 432
  - ST\_PerpPoints 434
  - ST\_Point 437
  - ST\_PointFromText 440
  - ST\_PointFromWKB 442
  - ST\_PointN 443
  - ST\_PointOnSurface 444
  - ST\_PolyFromText 446
  - ST\_PolyFromWKB 447
  - ST\_Polygon 449
  - ST\_PolygonN 452
  - ST\_Relate 453
  - ST\_RemovePoint 454
  - ST\_SRID
    - ST\_SrsId 456
  - ST\_SrsID
    - ST\_SRID 456
  - ST\_SrsName 458
  - ST\_StartPoint 459
  - ST\_SymDifference 460
  - ST\_ToGeomColl 463
  - ST\_ToLineString 464
  - ST\_ToMultiLine 465
  - ST\_ToMultiPoint 466
  - ST\_ToMultiPolygon 468
  - ST\_ToPoint 469
  - ST\_ToPolygon 470
  - ST\_Touches 471
  - ST\_Transform 473
  - ST\_Union 475
  - ST\_Within 478
  - ST\_WKBToSQL 479
  - ST\_WKTTToSQL 481
  - ST\_X 482
  - ST\_Y 484
  - ST\_Z 485
  - Union aggregate 487
    - using to exploit spatial
      - indexes 121
  - spatial grid indexes
    - creating 106
  - spatial indexes
    - description 105
    - exploiting 121
  - spatial reference systems
    - creating 76
    - description 74

## Index

- spatial reference systems (*continued*)
    - selecting 76
  - spatial stored procedures
    - deprecated 519
  - SPATIAL\_GEOCODER, deprecated
    - spatial catalog view 547
  - SPATIAL\_REF\_SYS, deprecated
    - spatial catalog view 547
  - ST\_alter\_coordsys stored
    - procedure 165
  - ST\_alter\_srs 167
  - ST\_COORDINATE\_SYSTEMS 229
  - ST\_create\_coordsys stored
    - procedure 172
  - ST\_create\_srs 174
  - ST\_disable\_autogeocoding 182
  - ST\_disable\_db stored
    - procedure 184
  - ST\_drop\_coordsys stored
    - procedure 186
  - ST\_drop\_srs 187
  - ST\_enable\_autogeocoding stored
    - procedure 189
  - ST\_enable\_db stored procedure 191
  - ST\_export\_shape stored
    - procedure 194
  - ST\_GEOCODER\_  
PARAMETERS 232
  - ST\_GEOCODERS 233
  - ST\_GEOCODING 234
  - ST\_GEOCODING\_  
PARAMETERS 236
  - ST\_GEOMETRY\_COLUMNS 231
  - ST\_GeomFromText, deprecated
    - spatial function 551
  - ST\_GeomFromWKB, deprecated
    - spatial function 551
  - ST\_import\_shape stored
    - procedure 198
  - ST\_LineFromText, deprecated spatial
    - function 551
  - ST\_LineFromWKB, deprecated
    - spatial functions 551
  - ST\_MLineFromText, deprecated
    - spatial functions 551
  - ST\_MLineFromWKB, deprecated
    - spatial function 551
  - ST\_MPointFromText, deprecated
    - spatial function 551
  - ST\_MPointFromWKB, deprecated
    - spatial function 551
  - ST\_MPolyFromText, deprecated
    - spatial function 551
  - ST\_MPolyFromWKB, deprecated
    - spatial function 551
  - ST\_OrderingEquals, deprecated
    - spatial function 551
  - ST\_Point, deprecated spatial
    - function 551
  - ST\_PointFromText, deprecated
    - spatial function 551
  - ST\_PolyFromText, deprecated spatial
    - function 551
  - ST\_PolyFromWKB, deprecated
    - spatial function 551
  - ST\_register\_geocoder stored
    - procedure 207
  - ST\_register\_spatial\_column stored
    - procedure 212
  - ST\_remove\_geocoding\_setup stored
    - procedure 215
  - ST\_run\_geocoding stored
    - procedure 217
  - ST\_setup\_geocoding stored
    - procedure 220
  - ST\_SIZINGS 238
  - ST\_SPATIAL\_  
REFERENCE\_SYSTEMS 238
  - ST\_SymmetricDiff, deprecated
    - spatial function 551
  - ST\_Transform, deprecated spatial
    - function 551
  - ST\_UNITS\_OF\_MEASURE 242
  - ST\_UNITS\_OF\_MEASURE catalog
    - view 242
  - ST\_unregister\_geocoder stored
    - procedure 224
  - ST\_unregister\_spatial\_column stored
    - procedure 226
  - stored procedures
    - calling
      - spatial applications 131
    - calling from spatial
      - applications 132
    - GSE\_export\_sde 160
    - GSE\_import\_sde 162
    - problems 146
    - ST\_alter\_coordsys 165
    - ST\_alter\_srs 167
    - ST\_create\_coordsys 172
    - ST\_create\_srs 174
    - ST\_disable\_autogeocoding 182
    - ST\_disable\_db 184
    - ST\_drop\_coordsys 186
    - ST\_drop\_srs 187
    - ST\_enable\_autogeocoding 189
    - ST\_enable\_db 191
    - ST\_export\_shape 194
    - ST\_import\_shape 198
    - ST\_register\_geocoder 207
  - stored procedures (*continued*)
    - ST\_register\_spatial\_column 212
    - ST\_remove\_geocoding\_  
setup 215
    - ST\_run\_geocoding 217
    - ST\_setup\_geocoding 220
    - ST\_unregister\_geocoder 224
    - ST\_unregister\_spatial\_  
column 226
- ## T
- tessellation 276
  - tracing events to isolate
    - problems 154
  - transform groups
    - ST\_GML 489
    - ST\_Shape 489
    - ST\_WellKnownBinary  
(WKB) 489
    - ST\_WellKnownText (WKT) 489
  - troubleshooting
    - functions 148
    - migration messages 150
    - shape information messages 150
  - Spatial Extender
    - messages 143
    - sample program 44
    - stored procedures 146
    - tracing 154
    - using runGseDemo 44
- ## V
- verifying
    - Spatial Extender installation 42
  - views
    - DB2 Spatial Extender
      - access spatial columns 118
- ## W
- well-known binary (WKB)
    - representation, data format 503
  - well-known text (WKT)
    - representation, data format 497
- ## Z
- Z, deprecated spatial function 551



---

## Contacting IBM

In the United States, call one of the following numbers to contact IBM:

- 1-800-237-5511 for customer service
- 1-888-426-4343 to learn about available service options
- 1-800-IBM-4YOU (426-4968) for DB2 marketing and sales

In Canada, call one of the following numbers to contact IBM:

- 1-800-IBM-SERV (1-800-426-7378) for customer service
- 1-800-465-9600 to learn about available service options
- 1-800-IBM-4YOU (1-800-426-4968) for DB2 marketing and sales

To locate an IBM office in your country or region, check IBM's Directory of Worldwide Contacts on the web at [www.ibm.com/planetwide](http://www.ibm.com/planetwide)

---

## Product information

Information regarding DB2 Universal Database products is available by telephone or by the World Wide Web at [www.ibm.com/software/data/db2/udb](http://www.ibm.com/software/data/db2/udb)

This site contains the latest information on the technical library, ordering books, client downloads, newsgroups, FixPaks, news, and links to web resources.

If you live in the U.S.A., then you can call one of the following numbers:

- 1-800-IBM-CALL (1-800-426-2255) to order products or to obtain general information.
- 1-800-879-2755 to order publications.

For information on how to contact IBM outside of the United States, go to the IBM Worldwide page at [www.ibm.com/planetwide](http://www.ibm.com/planetwide)



Part Number: CT19HNA

Printed in U.S.A.

SC27-1226-00



(1P) P/N: CT19HNA



Spine information:



IBM® DB2® Spatial Extender

# DB2 Spatial Extender User's Guide and Reference

Version 8