# ST_Geometry SQL function reference

THE SCIENCE OF WHERE

# Table of Contents

SQL functions used with ST_Geometry . . . 6
SQL and Esri ST_Geometry . . . 11
Load the SQLite ST_Geometry library . . . 13
Constructor functions for ST_Geometry . . . 14
Spatial accessor functions . . . 18
Spatial relationships . . . 25
Spatial relationship functions . . . 26
Spatial operations . . . 36
Spatial operation functions . . . 38
Parametric circles, ellipses, and wedges . . . 44
ST_Aggr_ConvexHull . . . 47
ST_Aggr_Intersection . . . 49
ST_Aggr_Union . . . 52
ST_Area . . . 55
ST_AsBinary . . . 58
ST_AsText . . . 60
ST_Boundary . . . 62
ST_Buffer . . . 66
ST_Centroid . . . 70
ST_Contains . . . 73
ST_ConvexHull . . . 77
ST_CoordDim . . . 81
ST_Crosses . . . 86
ST_Curve . . . 90
ST_Difference . . . 92
ST_Dimension . . . 96
ST_Disjoint . . . 100
ST_Distance . . . 104
ST_DWithin . . . 110
ST_EndPoint . . . 116
ST_Entity . . . 119
ST_Envelope . . . 123

ST_EnvIntersects . . . 129
ST_Equals . . . 132
ST_Equalsrs . . . 135
ST_ExteriorRing . . . 136
ST_GeomCollection . . . 139
ST_GeomCollFromWKB . . . 142
ST_Geometry . . . 144
ST_GeometryN . . . 151
ST_GeometryType . . . 153
ST_GeomFromCollection . . . 157
ST_GeomFromText . . . 159
ST_GeomFromWKB . . . 162
ST_GeoSize . . . 166
ST_InteriorRingN . . . 167
ST_Intersection . . . 169
ST_Intersects . . . 174
ST_Is3d . . . 178
ST_IsClosed . . . 182
ST_IsEmpty . . . 187
ST_IsMeasured . . . 191
ST_IsRing . . . 195
ST_IsSimple . . . 198
ST_Length . . . 201
ST_LineFromText . . . 204
ST_LineFromWKB . . . 206
ST_LineString . . . 209
ST_M . . . 211
ST_MaxM . . . 214
ST_MaxX . . . 217
ST_MaxY . . . 220
ST_MaxZ . . . 223
ST_MinM . . . 226
ST_MinX . . . 229
ST_MinY . . . 232

ST_MinZ . . . 235
ST_MLineFromText . . . 238
ST_MLineFromWKB . . . 240
ST_MPointFromText . . . 243
ST_MPointFromWKB . . . 245
ST_MPolyFromText . . . 248
ST_MPolyFromWKB . . . 250
ST_MultiCurve . . . 253
ST_MultiLineString . . . 254
ST_MultiPoint . . . 256
ST_MultiPolygon . . . 258
ST_MultiSurface . . . 260
ST_NumGeometries . . . 261
ST_NumInteriorRing . . . 264
ST_NumPoints . . . 267
ST_OrderingEquals . . . 270
ST_Overlaps . . . 272
ST_Perimeter . . . 276
ST_Point . . . 281
ST_PointFromText . . . 283
ST_PointFromWKB . . . 285
ST_PointN . . . 289
ST_PointOnSurface . . . 292
ST_PolyFromText . . . 295
ST_PolyFromWKB . . . 297
ST_Polygon . . . 300
ST_Relate . . . 302
ST_SRID . . . 307
ST_StartPoint . . . 309
ST_Surface . . . 312
ST_SymmetricDiff . . . 314
ST_Touches . . . 318
ST_Transform . . . 321
ST_Union . . . 328

ST_Within . . . . . 332
ST_X . . . . . 336
ST_Y . . . . . 339
ST_Z . . . . . 342

# SQL functions used with ST_Geometry

This reference document provides a list and description of the functions available for use with the Esri ST_Geometry spatial data type in Oracle, PostgreSQL, and SQLite.

Esri ST_Geometry SQL functions and types are created when you do any of the following:

- Create a geodatabase in an Oracle database.
- Use ST_Geometry when creating a geodatabase in a PostgreSQL database.
- Install the ST_Geometry spatial data type in an Oracle or PostgreSQL database.
- Create a SQLite database that includes the ST_Geometry spatial data type using the Create SQLite Database geoprocessing tool or ArcPy function and load ST_Geometry functions to use with the database.
- Load the ST_Geometry functions to use with a mobile geodatabase.

In Oracle and PostgreSQL databases, the ST_Geometry type and its functions are created in a schema named sde. In SQLite, the type and functions are stored in a library that you must load before you run SQL against the SQLite database or mobile geodatabase.

**Tip:**

For information about the Esri ST_Geometry type, see the following ArcGIS Pro help pages:

- ST_Geometry in PostgreSQL
- ST_Geometry in Oracle
- Databases and ST_Geometry
- Load the SQLite ST_Geometry library
- Load ST_Geometry to a mobile geodatabase for SQL access

## Format of the SQL function pages

The function pages in this document are structured as follows:

- Definition—A brief statement of what the function does
- Syntax—The SQL syntax to use the function

  **Note:**

  With relational operators, the order in which the parameters are specified is important: The first parameter must be for the table from which the selection is being made, and the second parameter must be for the table that is being used as a filter.

- Return type—The type of data that is returned when the function is issued
- Example—Samples that use the specific function

## List of SQL functions

Click the links below to go to the functions you can use with the ST_Geometry type in Oracle, PostgreSQL, and SQLite.

When using ST_Geometry functions in Oracle, you must qualify the functions and operators with `sde`. For example, ST_Buffer would be sde.ST_Buffer. Adding `sde.` indicates to the software that the function is stored in the schema of the sde user. For PostgreSQL, the qualification is optional, but it is a good practice to include the qualifier. Do not include the qualification when using the functions with SQLite, because there is no sde schema in SQLite databases.

When you provide well-known text strings as input with an ST_Geometry SQL function, you can use scientific notation to specify very large or very small values. For example, if you specify coordinates using well-known text when constructing a feature, and one of the coordinates is 0.000023500001816501026, you can type `2.3500001816501026e-005` instead.

**Tip:**

For other spatial types—such as the PostGIS types, Oracle SDO_Geometry, Microsoft SQL Server spatial types, IBM Db2 ST_Geometry, or SAP HANA ST_Geometry—consult the documentation provided by the database management system vendor for information on the functions used by each of these.

The Esri ST_Geometry SQL functions below are grouped based on their use.

## Constructor functions

Constructor functions take one type of geometry or a text description of geometry and create a geometry. The following table lists the constructor functions and indicates which ST_Geometry implementations support each one.

**Constructor functions**

| Function | Oracle | PostgreSQL | SQLite |
|---|---|---|---|
| ST_Centroid | X | X | X |
| ST_Curve | X | | X |
| ST_GeomCollection | X | X | |
| ST_GeomCollFromWKB | | X | |
| ST_Geometry | X | X | X |
| ST_GeomFromText | X | | X |
| ST_GeomFromWKB | X | X | X |
| ST_LineFromText | X | | X |
| ST_LineFromWKB | X | X | X |
| ST_LineString | X | X | X |
| ST_MLineFromText | X | | X |
| ST_MLineFromWKB | X | X | X |
| ST_MPointFromText | X | | X |
| ST_MPointFromWKB | X | X | X |
| ST_MPolyFromText | X | | X |
| ST_MPolyFromWKB | X | X | X |
| ST_MultiCurve | X | | |

| Function | Oracle | PostgreSQL | SQLite |
|---|---|---|---|
| ST_MultiLineString | X | X | X |
| ST_MultiPoint | X | X | X |
| ST_MultiPolygon | X | X | X |
| ST_MultiSurface | X | | |
| ST_Point | X | X | X |
| ST_PointFromText | X | | X |
| ST_PointFromWKB | X | X | X |
| ST_PolyFromText | X | | X |
| ST_PolyFromWKB | X | X | X |
| ST_Polygon | X | X | X |
| ST_Surface | X | | X |

## Accessor functions

There are a number of functions that take a geometry or geometries as input and return specific information about them.

Some of these accessor functions check to see whether a feature or features meet certain criteria. If the geometry meets the criteria, the function returns 1 (Oracle and SQLite) or t (PostgreSQL) for true. If the geometry does not meet the criteria, it returns 0 (Oracle and SQLite) or f (PostgreSQL) for false.

These functions apply to all implementations except where noted otherwise.

**Accessor functions**

| |
|---|
| ST_Area |
| ST_AsBinary |
| ST_AsText |
| ST_CoordDim |
| ST_Dimension |
| ST_EndPoint |
| ST_Entity |
| ST_Equalsrs (PostgreSQL only) |
| ST_ExteriorRing |
| ST_GeomFromCollection (PostgreSQL only) |
| ST_GeometryType |
| ST_GeoSize (PostgreSQL only) |
| ST_Is3d |
| ST_IsClosed |

| |
|---|
| ST_IsEmpty |
| ST_IsMeasured |
| ST_IsRing |
| ST_IsSimple |
| ST_Length |
| ST_M |
| ST_MaxM |
| ST_MaxX |
| ST_MaxY |
| ST_MaxZ |
| ST_MinM |
| ST_MinX |
| ST_MinY |
| ST_MinZ |
| ST_NumGeometries |
| ST_NumInteriorRing |
| ST_NumPoints |
| ST_Perimeter |
| ST_SRID |
| ST_StartPoint |
| ST_X |
| ST_Y |
| ST_Z |

## Relational functions

Relational functions take geometries as input and determine whether a spatial relationship exists between the geometries. If the conditions of spatial relationship are met, these functions return 1 (Oracle and SQLite) or t (PostgreSQL) for true. If the conditions are not met (no relationship exists), these functions return 0 (Oracle and SQLite) or f (PostgreSQL) for false.

These functions apply to all implementations except where noted otherwise.

**Relational functions**

| |
|---|
| ST_Contains |
| ST_Crosses |
| ST_Disjoint |
| ST_Distance |

| |
|---|
| ST_DWithin |
| ST_EnvIntersects (Oracle and SQLite only) |
| ST_Equals |
| ST_Intersects |
| ST_OrderingEquals (Oracle and PostgreSQL only) |
| ST_Overlaps |
| ST_Relate |
| ST_Touches |
| ST_Within |

## Geometry operation functions

These functions take spatial data, perform spatial operations on it, and return a geometry.

These functions apply to all implementations except where noted otherwise.

**Geometry operation functions**

| |
|---|
| ST_Aggr_ConvexHull (Oracle and SQLite only) |
| ST_Aggr_Intersection (Oracle and SQLite only) |
| ST_Aggr_Union |
| ST_Boundary |
| ST_Buffer |
| ST_ConvexHull |
| ST_Difference |
| ST_Envelope |
| ST_ExteriorRing |
| ST_GeometryN |
| ST_InteriorRingN |
| ST_Intersection |
| ST_PointN |
| ST_PointOnSurface |
| ST_SymmetricDiff |
| ST_Transform |
| ST_Union |

# SQL and Esri ST_Geometry

You can use the database management system's Structured Query Language (SQL), data types, and table formats to work with the information stored in a geodatabase or database where the ST_Geometry type is installed. SQL is a database language that supports data definition and data manipulation commands.

Accessing the data through SQL allows external applications to work with the tabular data managed by the geodatabase or database. These external applications can be nonspatial database applications or custom spatial applications.

When inserting data to or editing data in a geodatabase or database using SQL, issue a `COMMIT` or `ROLLBACK` statement after you run the SQL statement to ensure changes are either committed to the database or undone. This helps prevent locks from being held on the rows, pages, or tables you are editing.

## Insert ST_Geometry data using SQL

You can use SQL to insert spatial data to a database or geodatabase table that has an ST_Geometry column. You use ST_Geometry constructor functions to insert specific geometry types. You can also specify that the output of certain spatial operation functions be output to an existing table.

When you insert a geometry to a table using SQL, be aware of the following:

- You must specify a valid spatial reference ID (SRID).
- All geometries in the same column must use the same SRID.
- To continue using the table with ArcGIS, the field being used as the object ID cannot be null or contain nonunique values.

### Spatial reference IDs

The SRID you specify when inserting a geometry to a table in Oracle that uses the ST_Geometry spatial type must be in the ST_SPATIAL_REFERENCES table and have a matching record in the SDE.SPATIAL_REFERENCES table. The SRID you specify when inserting a geometry to a table in PostgreSQL that uses the ST_Geometry spatial type must be in the public.sde_spatial_references table. These tables are prepopulated with spatial references and SRIDs.

The SRID you specify when inserting a geometry to a table in SQLite that uses the ST_Geometry spatial type must be in the st_spatial_reference_systems table.

If you need to use a custom spatial reference that is not present in the table, the easiest way to do this is to use to load or create a feature class that has the spatial reference values you want. Ensure the feature class you create is using ST_Geometry storage. This creates a record in the SDE.SPATIAL_REFERENCES and ST_SPATIAL_REFERENCES tables in Oracle, a record in the public.sde_spatial_references table in PostgreSQL, or a record in the st_aux_spatial_reference_systems_table in SQLite.

In geodatabases, you can query the LAYERS (Oracle) or sde_layers (PostgreSQL) table to discover the SRID assigned to the spatial table. You could then use that SRID when you create spatial tables and insert data using SQL.

**Note:**
For the purpose of using the samples in this document, a record has been added to the ST_SPATIAL_REFERENCES and sde_spatial_references tables to denote an unknown spatial reference. This record has an SRID of 0. You can use this SRID for the examples in this document. However, this is not an official SRID—it is provided for the purpose of performing example SQL code. It is recommended that you do not use this SRID for your production data.

### Object IDs

For ArcGIS to query data, it requires that the table contain a unique object identifier field.

Feature classes created with ArcGIS always have an object ID field that is used as the identifier field. When inserting records to the feature class using ArcGIS, a unique value is always inserted to the object ID field. The object ID field in a geodatabase table is maintained by ArcGIS. The object ID field in a database table created from ArcGIS is maintained by the database management system.

When you insert records to a geodatabase table using SQL, you must insert a valid, unique object ID value.

Database tables you create outside of ArcGIS must have a field (or set of fields) that ArcGIS can use as an object ID. If you use your database's native autoincrementing data type for the ID field in your table, this field will be populated by the database when you insert a record using SQL. If you are manually maintaining the values in your unique identifier field, be sure to provide a unique value for the ID when editing the table from SQL.

**Note:**
You cannot publish data from tables that have a unique identifier field that is not maintained by ArcGIS or the database management system.

## Edit ST_Geometry data using SQL

SQL edits to existing records often affect the nonspatial attributes stored in the table; however, you can edit the data in the ST_Geometry column using constructor functions inside SQL `UPDATE` statements.

If the data is stored in a geodatabase, there are additional guidelines you must follow when editing with SQL:

- Do not update records using SQL if the data has been registered as versioned or enabled for geodatabase archiving.
- Do not modify any attributes that affect other objects in the database that participate in geodatabase behavior such as relationship classes, feature-linked annotation, topology, attribute rules, or networks.
- Do not use SQL to alter table schemas.

**Caution:**
Using SQL to access the geodatabase bypasses geodatabase functionality, such as versioning, topology, networks, terrains, feature-linked annotation, or other class or workspace extensions. It may be possible to use database management system features such as triggers and stored procedures to maintain the relationships between tables needed for certain geodatabase functionality. However, running SQL commands against the geodatabase without taking this extra functionality into account—for example, issuing `INSERT` statements to add records to a table that has geodatabase archiving enabled on it or adding a column to an existing feature class—will circumvent geodatabase functionality and possibly corrupt the relationships between data in the geodatabase.

# Load the SQLite ST_Geometry library

Before running SQL commands that contain ST_Geometry functions against an SQLite database, do the following:

1. Download the ArcGIS Pro ST_Geometry Libraries (SQLite) zip file from My Esri and unzip it.
2. Install an SQL editor on the same machine as the database.
3. Place the ST_Geometry file in a location that is accessible to the SQLite database and the SQL editor from which you will load ST_Geometry.
   If the SQLite database is on a Microsoft Windows machine, use the `stgeometry_sqlite.dll` file. If the SQLite database is on a Linux machine, use the `libstgeometry_sqlite.so` file.
4. Open the SQL editor and connect to the SQLite database.
5. Load the ST_Geometry library.
   In the first example below, the library is loaded for a SQLite database on a Windows machine. The second example loads the library for a SQLite database on a Linux machine.

```
--Load the ST_Geometry library on Windows.
SELECT load_extension(
 'stgeometry_sqlite.dll',
 'SDE_SQL_funcs_init'
);

--Load the ST_Geometry library on Linux.
SELECT load_extension(
 'libstgeometry_sqlite.so',
 'SDE_SQL_funcs_init'
);
```

You can now run SQL commands that contain ST_Geometry functions against the SQLite database.

# Constructor functions for ST_Geometry

Constructor functions create a geometry from a well-known text description or well-known binary.

When you provide a well-known text description to construct a geometry, the measure coordinate must be specified last. For example, if your text includes coordinates for x, y, z, and m, they must be provided in that order.

A geometry can have zero or more points. A geometry is considered empty if it has zero points. The point subtype is the only geometry that is restricted to zero or one point; all other subtypes can have zero or more.

The following sections describe the geometry superclass and subclass geometries and list the functions that can construct each one.

You can also construct geometries as the result of a spatial operation performed on existing geometries.

## Geometry superclass

The ST_Geometry superclass cannot be instantiated; although you can define a column as an ST_Geometry type, the actual data inserted is defined as either point, linestring, polygon, multipoint, multilinestring, or multipolygon entities.

You can use the following functions to create a superclass to hold any of the aforementioned entity types.

- ST_Geometry
- ST_GeomFromText (Oracle and SQLite only)
- ST_GeomFromWKB

## Subclasses

You can define a feature as a specific subclass, in which case only the entity type allowed for that subclass can be inserted. For example, ST_PointFromWKB can only construct point entities.

### ST_Point

An ST_Point is a zero-dimensional geometry that occupies a single location in coordinate space. An ST_Point has a single x,y coordinate value, is always simple, and has a NULL boundary. ST_Point can be used to define features such as oil wells, landmarks, and water sample collection sites.

The following functions create a point:

- ST_Point
- ST_PointFromText (Oracle and SQLite only)
- ST_PointFromWKB

### ST_MultiPoint

An ST_MultiPoint is a collection of ST_Points and, like its elements, has a dimension of 0. An ST_MultiPoint is simple if none of its elements occupy the same coordinate space. The boundary of an ST_MultiPoint is NULL. ST_MultiPoints can define such things as aerial broadcast patterns and incidents of a disease outbreak.

The following functions create a multipoint geometry:

- ST_MultiPoint
- ST_MPointFromText (Oracle only)

- ST_MPointFromWKB

## ST_LineString

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

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

Functions that create linestrings include the following:

- ST_LineString
- ST_LineFromText (Oracle and SQLite only)
- ST_LineFromWKB
- ST_Curve (Oracle and SQLite only)

## ST_MultiLineString

An ST_MultiLineString is a collection of ST_LineStrings.

The boundary of an ST_MultiLineString is the nonintersected endpoints of the ST_LineString elements. The boundary of an ST_MultiLineString is NULL if all the endpoints of all the elements are intersected. In addition to the other properties inherited from the superclass ST_Geometry, ST_MultiLineStrings have length. ST_MultiLineStrings are used to define noncontiguous linear features, such as streams or road networks.

Functions that construct multilinestrings are as follows:

- ST_MultiLineString
- ST_MLineFromText (Oracle and SQLite only)
- ST_MLineFromWKB
- ST_MultiCurve (Oracle only)

## ST_Polygon

An ST_Polygon is a two-dimensional surface stored as a sequence of points defining its exterior bounding ring and 0 or more interior rings. ST_Polygons are always simple. ST_Polygons define features that have spatial extent, such as parcels of land, water bodies, and areas of jurisdiction.

This graphic shows examples of ST_Polygon objects: 1 is an ST_Polygon for which the boundary is defined by an exterior ring. 2 is an ST_Polygon with a boundary defined by an exterior ring and two interior rings. The area inside the interior rings is part of the ST_Polygon's exterior. 3 is a legal ST_Polygon, because the rings intersect at a single tangent point.


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

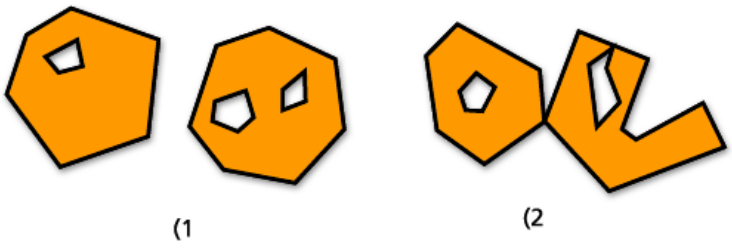
Functions that create polygons include the following:

- ST_Polygon
- ST_PolyFromText (Oracle and SQLite only)
- ST_PolyFromWKB
- ST_Surface (Oracle and SQLite only)

## ST_MultiPolygon

The boundary of an ST_MultiPolygon is the cumulative length of its elements' exterior and interior rings. The interior of an ST_MultiPolygon is defined as the cumulative interiors of its element ST_Polygons. The boundary of an ST_MultiPolygon's elements can only intersect at a tangent point. In addition to the other properties inherited from the superclass ST_Geometry, ST_MultiPolygons have area. ST_MultiPolygons define features such as a forest stratum or a noncontiguous parcel of land such as a Pacific island chain.

The graphic below provides examples of ST_MultiPolygon: 1 is an ST_MultiPolygon with two ST_Polygon elements. The boundary is defined by the two exterior rings and the three interior rings. 2 is also an ST_MultiPolygon with two ST_Polygon elements, but the boundary is defined by the two exterior rings and the two interior rings, and the two ST_Polygon elements intersect at a tangent point.




The following functions create multipolygons:

- ST_MultiPolygon
- ST_MPolyFromText (Oracle and SQLite only)
- ST_MPolyFromWKB
- ST_MultiSurface (Oracle only)

## Constructing geometries from existing geometries

Though not strictly constructor functions, the following functions return a new geometry by taking existing geometries as input and performing analyses on them:

- ST_Aggr_ConvexHull (Oracle and SQLite only)
- ST_Aggr_Intersection (Oracle and SQLite only)
- ST_Aggr_Union
- ST_Boundary
- ST_Buffer
- ST_Centroid
- ST_ConvexHull
- ST_Difference
- ST_Envelope
- ST_ExteriorRing
- ST_Intersection
- ST_SymmetricDiff
- ST_Transform
- ST_Union

# Spatial accessor functions for ST_Geometry

Spatial accessor functions return the property of a geometry. There are accessor functions to determine the following properties of an ST_Geometry feature:

## Dimensionality

The dimensions of a geometry are the minimum coordinates (none, x, y) required to define the spatial extent of the geometry.

A geometry can have a dimension of 0, 1, or 2.

The dimensions are as follows:

- 0—Has neither length nor area
- 1—Has a length (x or y)
- 2—Contains area (x and y)

Point and multipoint subtypes have a dimension of 0. Points represent zero-dimensional features that can be modeled with a single coordinate, while multipoints represent data that must be modeled with a cluster of unconnected coordinates.

Linestring and multilinestring subtypes have a dimension of 1. They store features such as road segments, branching river systems, and any other features that are linear in nature.

Polygon and multipolygon subtypes have a dimension of 2. Forest stands, parcels, water bodies, and other features that have perimeters that enclose a definable area can be rendered by either the polygon or multipolygon data type.

Dimension is important not only as a property of the subtype but also in determining the spatial relationship of two features. The dimension of the resultant feature or features determines whether or not the operation was successful. Spatial accessor functions examine the dimensions of the features to determine how they should be compared.

To evaluate the dimension of a geometry, use the ST_Dimension function, which takes an ST_Geometry feature and returns its dimension as an integer.

The coordinates of a geometry also have dimensions. If a geometry has only x- and y-coordinates, the coordinate dimension is 2. If a geometry has x-, y-, and z-coordinates, the coordinate dimension is 3. If a geometry has x-, y-, z-, and m-coordinates, the coordinate dimension is 4.

You can use the ST_CoordDim function to determine the coordinate dimensions present in a geometry.

## Z-coordinates

Some geometries have an associated altitude or depth—a third dimension. Each of the points that form the geometry of a feature can include an optional z-coordinate that represents an altitude or depth relative to the earth's surface.

The ST_Is3d predicate function takes an ST_Geometry as input, and returns true if the function has z-coordinates or returns false if it does not.

You can determine a point's z-coordinate using the ST_Z function.

The ST_MaxZ function returns the maximum z-coordinate, and the ST_MinZ function returns the minimum z-coordinate of a geometry.

## Measures

Measures are values assigned to each coordinate. They are used for linear referencing and dynamic segmentation applications. For example, milepost locations along a highway can contain measures that indicate their position. The value represents anything that can be stored as a double-precision number.

The ST_IsMeasured predicate function takes a geometry and returns true if it contains measures and false if it does not. This function is used with the Oracle and SQLite implementations of ST_Geometry only.

You can discover the measure value of a point using the ST_M function.

The ST_MaxM function returns the maximum m-coordinate, and the function ST_MinM returns the minimum m-coordinate of a geometry.

## Geometry type

The geometry type refers to type of geometric entity. These include the following:

- Points and multipoints
- Lines and multilines
- Polygons and multipolygons

ST_Geometry is a superclass that can store various subtypes. To determine what subtype a geometry is, use the ST_GeometryType or ST_Entity (Oracle and SQLite only) function.

## Point (vertex) collection and number of points

A geometry can have zero or more points. A geometry is considered empty if it has zero points. The point subtype is the only geometry that is restricted to zero or one point; all other subtypes can have zero or more.

### ST_Point

An ST_Point is a zero-dimensional geometry that occupies a single location in coordinate space. An ST_Point has a single x,y coordinate value, is always simple, and has a NULL boundary. ST_Point can be used to define features such as oil wells, landmarks, and water sample collection sites.

Functions that operate solely on the ST_Point data type include the following:

- ST_X—Returns a point data type's x-coordinate value as a double-precision number
- ST_Y—Returns a point data type's y-coordinate value as a double-precision number
- ST_Z—Returns a point data type's z-coordinate value as a double-precision number
- ST_M—Returns a point data type's m-coordinate value as a double-precision number

### ST_MultiPoint

An ST_MultiPoint is a collection of ST_Points and, like its elements, has a dimension of 0. An ST_MultiPoint is simple if none of its elements occupy the same coordinate space. The boundary of an ST_MultiPoint is NULL. ST_MultiPoints can define such things as aerial broadcast patterns and incidents of a disease outbreak.

You can use the ST_NumGeometries function to determine the number of points in a multipoint geometry.

## Length, area, and perimeter

Length, area, and perimeter are measurable characteristics of geometries. Linestrings and the elements of multilinestrings are one dimensional and possess the characteristic of length. Polygons and the elements of multipolygons are two-dimensional surfaces and, therefore, have an area and perimeters you can measure. You can use the functions ST_Length, ST_Area, and ST_Perimeter to determine these properties. Units of measurement vary depending on how the data is stored.

### ST_LineString

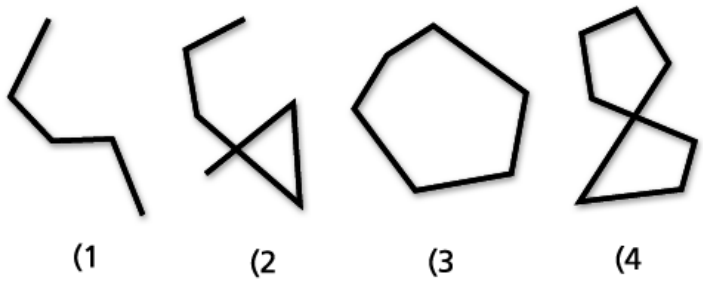
An ST_LineString is a one-dimensional object stored as a sequence of points defining a linear interpolated path. The ST_LineString is simple if it does not intersect its interior. The endpoints (the boundary) of a closed ST_LineString occupy the same point in space. An ST_LineString is a ring if it is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

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

Functions that operate on ST_LineStrings include the following:

- ST_StartPoint—Returns the first point of the specified ST_LineString
- ST_EndPoint—Returns the last point of an ST_LineString
- ST_IsClosed—A predicate function that returns true if the specified ST_LineString is closed (the start point and endpoint of the linestring intersect) and false if it is not
- ST_IsRing—A predicate function that returns true if the specified ST_LineString is a ring and false if it is not a ring
- ST_Length—Returns the length of an ST_LineString as a double-precision number
- ST_NumPoints—Evaluates an ST_LineString and returns the number of points in its sequence as an integer
- ST_PointN—Takes an ST_LineString and an index to the nth point and returns that point

The graphic below shows examples of ST_LineString objects: (1 is a simple, nonclosed ST_LineString; (2 is a nonsimple, nonclosed ST_LineString; (3 is a closed, simple ST_LineString and, therefore, a ring; and (4 is a closed, nonsimple ST_LineString, but it is not a ring.

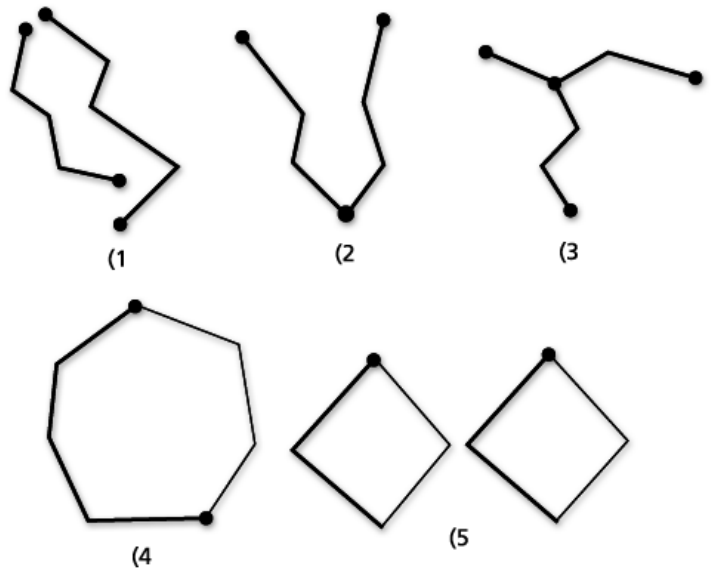


## ST_MultiLineString

An ST_MultiLineString is a collection of ST_LineStrings. ST_MultiLineStrings are simple if they only intersect at the endpoints of the ST_LineString elements. ST_MultiLineStrings are nonsimple if the interiors of the ST_LineString elements intersect.

The boundary of an ST_MultiLineString is the nonintersected endpoints of the ST_LineString elements. The boundary of an ST_MultiLineString is NULL if all the endpoints of all the elements are intersected. In addition to the other properties inherited from the superclass ST_Geometry, ST_MultiLineStrings have length. ST_MultiLineStrings are used to define noncontiguous linear features, such as streams or road networks.

The following graphic provides examples of ST_MultiLineStrings: (1 is a simple ST_MultiLineString for which the boundary is the four endpoints of its two ST_LineString elements. (2 is a simple ST_MultiLineString, because only the endpoints of the ST_LineString elements intersect. The boundary is two nonintersected endpoints. (3 is a nonsimple ST_MultiLineString, because the interior of one of its ST_LineString elements is intersected. The boundary of this ST_MultiLineString is the three nonintersected endpoints. (4 is a simple nonclosed ST_MultiLineString. It is not closed because its element ST_LineStrings are not closed. It is simple because none of the interiors of any of the element ST_LineStrings intersect. (5 is a single, simple, closed ST_MultiLineString. It is closed because all its elements are closed. It is simple because none of its elements intersect at the interiors.




Functions that operate on ST_MultiLineStrings include the following:

- ST_IsClosed—This predicate function returns a value indicating true if the specified ST_MultiLineString is closed and a false value if it is not closed.
- ST_Length—This function evaluates an ST_MultiLineString and returns the cumulative length of all its ST_LineString elements as a double-precision number.
- ST_NumGeometries—This function returns the number of lines in a multilinestring.

## ST_Polygon

An ST_Polygon is a two-dimensional surface stored as a sequence of points defining its exterior bounding ring and 0 or more interior rings. ST_Polygons are always simple. ST_Polygons define features that have spatial extent, such as parcels of land, water bodies, and areas of jurisdiction.

This graphic shows examples of ST_Polygon objects: 1 is an ST_Polygon for which the boundary is defined by an exterior ring. 2 is an ST_Polygon with a boundary defined by an exterior ring and two interior rings. The area inside the interior rings is part of the ST_Polygon's exterior. 3 is a legal ST_Polygon, because the rings intersect at a single tangent point.


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

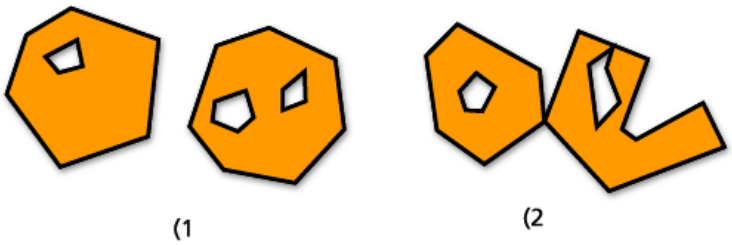
Functions that operate on an ST_Polygon include the following:

- ST_Area—Returns the area of an ST_Polygon as a double-precision number
- ST_Centroid—Returns an ST_Point that represents the center of the ST_Polygon's envelope
- ST_ExteriorRing—Returns the exterior ring of an ST_Polygon as an ST_LineString
- ST_InteriorRingN—Evaluates an ST_Polygon and an index and returns the nth interior ring as an ST_LineString
- ST_NumInteriorRing—Returns a count of interior rings that an ST_Polygon contains
- ST_PointOnSurface—Returns an ST_Point that is guaranteed to be on the surface of the specified ST_Polygon

## ST_MultiPolygon

The boundary of an ST_MultiPolygon is the cumulative length of its elements' exterior and interior rings. The interior of an ST_MultiPolygon is defined as the cumulative interiors of its element ST_Polygons. The boundary of an ST_MultiPolygon's elements can only intersect at a tangent point. In addition to the other properties inherited from the superclass ST_Geometry, ST_MultiPolygons have area. ST_MultiPolygons define features such as a forest stratum or a noncontiguous parcel of land such as a Pacific island chain.

The graphic below provides examples of ST_MultiPolygon: 1 is an ST_MultiPolygon with two ST_Polygon elements. The boundary is defined by the two exterior rings and the three interior rings. 2 is also an ST_MultiPolygon with two ST_Polygon elements, but the boundary is defined by the two exterior rings and the two interior rings, and the two ST_Polygon elements intersect at a tangent point.




Functions that operate on ST_MultiPolygons include the following:

- ST_Area—Returns a double-precision number that represents the cumulative ST_Area of an ST_MultiPolygon's ST_Polygon elements.
- ST_Centroid—Returns an ST_Point that is the center of an ST_MultiPolygon's envelope.
- ST_NumGeometries—Returns a count of the number of polygons in a multipolygon.
- ST_PointOnSurface—Evaluates an ST_MultiPolygon and returns an ST_Point that is guaranteed to be normal to the surface of one of its ST_Polygon elements.

## Simple geometries in a multipart geometry

Multipart geometries are made up of individual simple geometries.

You may want to determine how many individual geometries are in a multipart geometry, such as an ST_MultiPoint, ST_MultiLineString, and ST_MultiPolygon. To do this, use the ST_NumGeometries predicate function. This function returns a count of the individual elements in a collection of geometries.

Using the ST_GeometryN function, you can determine which geometry in the multipart geometry exists in position N; N being a number that you provide with the function. For example, if you want to return the third point of a multipoint geometry, you would include 3 when you run the function.

To return the individual geometries and their position from a multipart geometry in PostgreSQL, use the ST_GeomFromCollection function.

## Interior, boundary, exterior

All geometries occupy a position in space defined by their interiors, boundaries, and exteriors. The exterior of a geometry is all the space not occupied by the geometry. The interior is the space occupied by the geometry. The boundary of a geometry is the location between its interior and exterior. The subtype inherits the interior and exterior properties directly; however, the boundary property differs for each.

Use the ST_Boundary function to determine the source ST_Geometry's boundary.

## Simple or nonsimple

Some subtypes of ST_Geometry are always simple, such as ST_Points or ST_Polygons. However, the subtypes ST_LineStrings, ST_MultiPoints, and ST_MultiLineStrings can be either simple or nonsimple. They are simple if they obey all topological rules imposed on them and nonsimple if they do not.

Topological rules include the following:

- An ST_LineString is simple if it does not intersect its interior and nonsimple if it does.
- An ST_MultiPoint is simple if no two of its elements occupy the same coordinate space (have the same x,y coordinates) and nonsimple if they do.
- An ST_MultiLineString is simple if none of its elements' interiors is intersected by its own interior and nonsimple if any of the elements' interiors do intersect.

The ST_IsSimple predicate function is used to determine whether an ST_LineString, ST_MultiPoint, or ST_MultiLineString is simple or nonsimple. ST_IsSimple takes an ST_Geometry and returns true if the geometry is simple and false if it is not.

## Empty or not empty

A geometry is empty if it does not have any points. An empty geometry has a null envelope, boundary, interior, and exterior. An empty geometry is always simple. Empty linestrings and multilinestrings have a 0 length. Empty polygons and multipolygons have a 0 area.

The ST_IsEmpty predicate function can be used to determine whether a geometry is empty. It analyzes an ST_Geometry and returns true if the geometry is empty and false if it is not.

## IsClosed and IsRing

Linestring geometries can be closed or be rings. Linestrings can be closed without being rings. You can determine whether a linestring is closed by using the ST_IsClosed predicate function; it returns true if the start point and endpoint of the linestring intersect. Rings are linestrings that are closed and simple. The ST_IsRing predicate function can be used to test whether a linestring is truly a ring; it returns true if the linestring is closed and is simple.

## Envelope

Every geometry has an envelope. The envelope of a geometry is the bounding geometry formed by the minimum and maximum x,y coordinates. For point geometries, since the minimum and maximum x,y coordinates are the same, a rectangle, or envelope, is created around these coordinates. For line geometries, the endpoints of the line represent two sides of the envelope and the other two sides are created just above and just below the line.

The ST_Envelope function takes an ST_Geometry and returns an ST_Geometry that represents the source ST_Geometry's envelope.

To find the individual minimum and maximum x,y coordinates of a geometry, use the functions ST_MinX, ST_MinY, ST_MaxX, and ST_MaxY.

## Spatial reference system

The spatial reference system identifies the coordinate transformation matrix for each geometry. It is made up of a coordinate system, resolution, and tolerance.

All spatial reference systems known to the geodatabase are stored in a geodatabase system table.

The following functions get information about spatial reference systems of geometries:

- ST_SRID—Takes an ST_Geometry and returns its spatial reference identifier (SRID) as an integer.
- ST_Equalsrs—Determines whether the spatial reference systems of two different feature classes are identical (true) or not (false).

## Size of features (PostgreSQL only)

The features (spatial records in a table) take up a certain amount of storage space in bytes. You can use the ST_GeoSize function to determine how big each feature in a table is.

## Text and binary definitions of a geometry

To obtain the well-known text definition or well-known binary definition of the geometry in a specific row in the spatial table, use the ST_AsText and ST_AsBinary functions respectively.

# Spatial relationships

A primary function of a GIS is to determine the spatial relationships between features: Do they overlap? Is one contained by the other? Does one cross the other?

Geometries can be spatially related in different ways. The following are examples of how one geometry can be spatially related to another:

- Geometry A passes through geometry B.
- Geometry A is completely contained by geometry B.
- Geometry A completely contains geometry B.
- The geometries do not intersect or touch one another.
- The geometries are completely coincident.
- The geometries overlap each other.
- The geometries touch at one point.

To determine whether these relationships exist or not, use spatial relationship functions. These functions compare the following properties of the geometries you specify in a query:

- The exterior (E) of the geometries—The exterior is all of the space not occupied by a geometry.
- The interior (I) of the geometries—The interior is the space occupied by a geometry.
- The boundary (B) of the geometries—The boundary is the interface between a geometry's interior and its exterior.

When you construct a spatial relationship query, specify the type of spatial relationship you are looking for and the geometries you want to compare. The queries return as either true or false. In other words, either the geometries participate with one another in the specified spatial relationship or they do not. In most cases, you use a spatial relationship query to filter a result set by placing it in the `WHERE` clause.

For example, if you have a table that stores the locations of proposed development sites and another table that stores the location of archaeologically significant sites, you can issue a query to ensure none of the development sites intersect archaeology sites and, if any do, return the ID of those proposed developments. In this example, the ST_Disjoint function is used in PostgreSQL.

```
SELECT d.projname,a.siteid
 FROM dev d, archsites a
 WHERE sde.st_disjoint(d.shape,a.shape)= 'f'

projname        siteid

bow wow chow    A1009
```

This query returns the name of the development (projname) and the ID of the archaeological site (siteid) that are not disjoint—in other words, the sites that intersect one another. It returns one development project, Bow Wow Chow, which intersects archaeological site A1009.

# Relational functions for ST_Geometry

Relational functions use predicates to test for different types of spatial relationships. The tests achieve this by comparing the relationships between the following:

- The exterior (E) of the geometries—The exterior is all of the space not occupied by a geometry.
- The interior (I) of the geometries—The interior is the space occupied by a geometry.
- The boundary (B) of the geometries—The boundary is the interface between a geometry's interior and its exterior.

Predicates test relationships. They return 1 (Oracle and SQLite) or t (PostgreSQL) if a comparison meets the function's criteria; otherwise, they return 0 (Oracle and SQLite) or f (PostgreSQL). Predicates that test for a spatial relationship compare pairs of geometry that can be a different type or dimension.

Predicates compare the x- and y-coordinates of the submitted geometries. The z-coordinates and measure values, if they exist, are ignored. Geometries that have z-coordinates or measures can be compared with those that do not.

The Dimensionally Extended 9 Intersection Model (DE-9IM) developed by Clementini, et al. dimensionally extends the 9 Intersection Model of Egenhofer and Herring. 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 among all types of geometry as pair-wise intersections of their interior, boundary, and exterior with consideration for the dimension of the resulting intersections.

Given geometries a and b, I(a), B(a), and E(a) represent the interior, boundary, and exterior of a, and I(b), B(b), and E(b) represent the interior, boundary, and exterior of b. The intersections of I(a), B(a), and E(a) with I(b), B(b), and E(b) produce a three-by-three matrix. Each intersection can result in geometries of different dimensions. For example, the intersection of the boundaries of two polygons could consist of a point and a linestring, in which case the dim (dimension) 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 that is returned when no intersection is found or dim(Ãƒ˜).

| | **Interior** | **Boundary** | **Exterior** |
|---|---|---|---|
| Interior | dim(I(a) intersects I(b)) | dim(I(a) intersects B(b)) | dim(I(a) intersects E(b)) |
| Boundary | dim(B(a) intersects I(b)) | dim(B(a) intersects B(b)) | dim(B(a) intersects E(b)) |
| Exterior | dim(E(a) intersects I(b)) | dim(E(a) intersects B(b)) | dim(E(a) intersects E(b)) |

*An example intersection of a predicate*

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

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

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 or not; dim = -1, 0, 1, or 2

0—An intersection must exist and its maximum dimension must be 0; dim = 0

1—An intersection must exist and its maximum dimension must be 1; dim = 1

2—An intersection must exist and its maximum dimension must be 2; dim = 2

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

The pattern matrix of the ST_Within predicate for geometry combinations has the following form:

| | | | **Geometry b** | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | T | * | F |
| | **Boundary** | * | * | F |
| | **Exterior** | * | * | * |

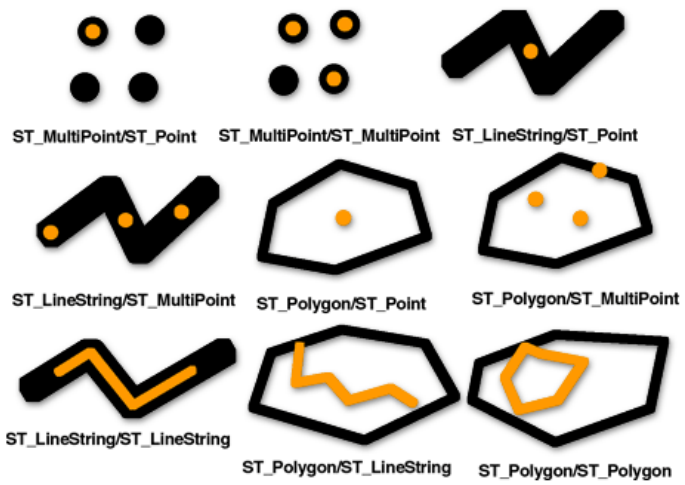
*Example pattern matrix*

The ST_Within predicate is true when the interiors of both geometries intersect, and the interior and boundary of geometry a does not intersect the exterior of geometry b. All other conditions do not matter.

The sections below describe different predicates used for spatial relationships. In the diagrams in these sections, the first input geometry listed is shown in black and the second is depicted in orange.

## ST_Contains

ST_Contains returns 1 or t (true) if the second geometry is completely contained by the first geometry. The ST_Contains predicate returns the exact opposite result of the ST_Within predicate.



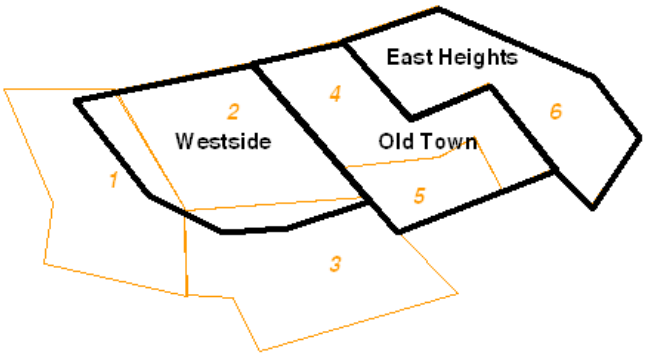

The pattern matrix of the ST_Contains predicate states that the interiors of both geometries must intersect and that the interior and boundary of the secondary (geometry b) must not intersect the exterior of the primary (geometry a).

| | | | **Geometry b** | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | T | * | * |
| | **Boundary** | * | * | * |

*ST_Contains matrix*

| | | | | |
|---|---|---|---|---|
| | **Exterior** | F | F | * |

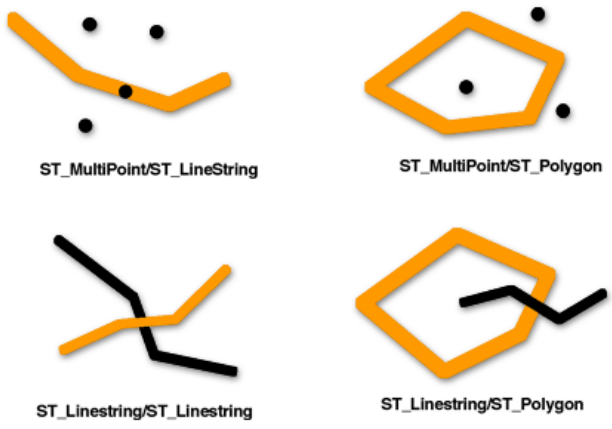
The ST_Within an ST_Contains functions identify only those geometries that fall entirely inside another geometry. This helps to eliminate features from your selection that could skew your results. In the example below, a traveling ice cream vendor wants to determine which neighborhoods contain the largest number of children (potential customers) and restrict his route to those areas. The vendor compares polygons of designated neighborhoods to census tracts, which possess an attribute of a total number of children under the age of 16.




Unless all the children living in census tract 1 and census tract 3 live in the slivers of land that fall within Westside, including these tracts in the selection could erroneously elevate the count of children in the Westside neighborhood. By specifying that only census tracts entirely within neighborhoods be included (ST_Within = 1), the ice cream vendor potentially saves time and money by not venturing into those portions of Westside.

## ST_Crosses

ST_Crosses returns 1 or t (true) if the intersection results in a geometry that has a dimension that is one less than the maximum dimension of the two source geometries and the intersection set is interior to both source geometries. ST_Crosses returns 1 or t (true) for only ST_MultiPoint/ST_Polygon, ST_MultiPoint/ST_LineString, ST_Linestring/ST_LineString, ST_LineString/ST_Polygon, and ST_LineString/ST_MultiPolygon comparisons.




The following ST_Crosses predicate pattern matrix applies to ST_MultiPoint/ST_LineString, ST_MultiPoint/ ST_MultiLineString, ST_MultiPoint/ST_Polygon, ST_MultiPoint/ST_MultiPolygon, ST_LineString/ST_Polygon, and ST_LineString/ST_MultiPolygon. The matrix states that the interiors must intersect and that at least the interior of the primary (geometry a) must intersect the exterior of the secondary (geometry b).

| | | | Geometry b | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | T | * | T |
| | **Boundary** | * | * | * |
| | **Exterior** | * | * | * |

*ST_Crosses matrix 1*

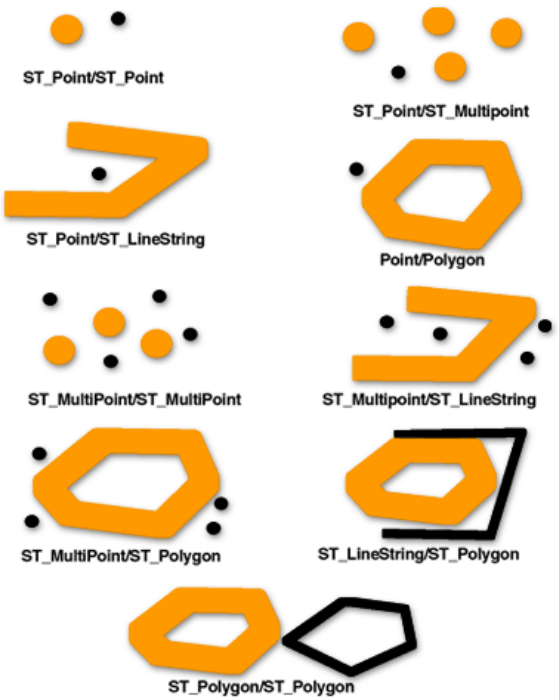
The following ST_Crosses predicate matrix applies to ST_LineString/ST_LineString, ST_LineString/ST_MultiLineString, and ST_MultiLineString/ST_MultiLineString. The matrix states that the dimension of the intersection of the interiors must be 0 (intersect at a point). If the dimension of this intersection was 1 (intersect at a linestring), the ST_Crosses predicate would return false, but the ST_Overlaps predicate would return true.

| | | | Geometry b | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | 0 | * | * |
| | **Boundary** | * | * | * |
| | **Exterior** | * | * | * |

*ST_Crosses matrix 2*

## ST_Disjoint

ST_Disjoint returns 1 or t (true) if the intersection of the two geometries is an empty set. In other words, geometries are disjoint if they do not intersect one another.



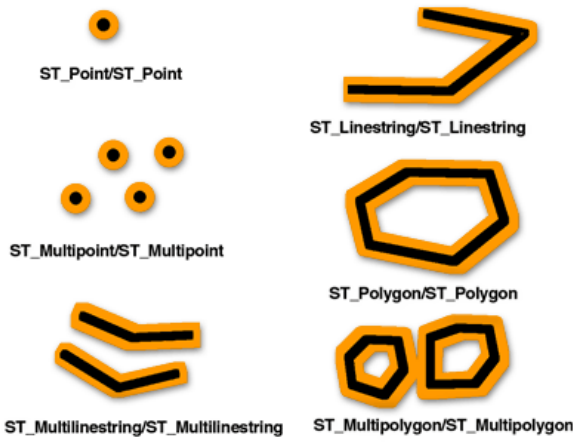
The ST_Disjoint predicate's pattern matrix states that neither the interiors nor the boundaries of either geometry intersect.

| | | | Geometry b | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | **F** | **F** | ***** |
| | **Boundary** | **F** | **F** | ***** |
| | **Exterior** | * | * | * |

*ST_Disjoint matrix*

## ST_Equals

ST_Equals returns 1 or t (true) if two geometries of the same type have identical x,y coordinate values. The first and second floors of an office building could have identical x,y coordinates and, therefore, be equal. ST_Equals can also identify whether two features were mistakenly placed one on top of the other.




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

| | | | Geometry b | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | T | * | F |
| | **Boundary** | * | * | F |
| | **Exterior** | F | F | * |

*ST_Equals matrix*

## ST_Intersects

ST_Intersects returns 1 or t (true) if the intersection does not result in an empty set. ST_Intersects returns the exact opposite result of ST_Disjoint.

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

The ST_Intersects predicate returns true if the interiors of both geometries intersect.

| | | | Geometry b | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | T | * | * |
| | **Boundary** | * | * | * |
| | **Exterior** | * | * | * |

*ST_Intersects matrix 1*

The ST_Intersects predicate returns true if the interior of the first geometry intersects the boundary of the second geometry.

| | | | Geometry b | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | * | T | * |
| | **Boundary** | * | * | * |
| | **Exterior** | * | * | * |

*ST_Intersects matrix 2*

The ST_Intersects predicate returns true if the boundary of the first geometry intersects the interior of the second.

| | | | Geometry b | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | * | * | * |
| | **Boundary** | T | * | * |
| | **Exterior** | * | * | * |

*ST_Intersects matrix 3*

The ST_Intersects predicate returns true if the boundaries of either geometry intersect.

| | | | Geometry b | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | * | * | * |
| | **Boundary** | * | T | * |
| | **Exterior** | * | * | * |

*ST_Intersects matrix 4*

## ST_Overlaps

ST_Overlaps compares two geometries of the same dimension and returns 1 or t (true) if their intersection set results in a geometry different from both input geometries but is of the same dimension.

ST_Overlaps returns 1 or t (true) only for geometries of the same dimension and only when their intersection set results in a geometry of the same dimension. In other words, if the intersection of two ST_Polygons results in an ST_Polygon, overlap returns 1 or t (true).

ST_LineString/ST_LineString

ST_Polygon/ST_Polygon

ST_MultiPoint/ST_MultiPoint

This pattern matrix applies to ST_Polygon/ST_Polygon, ST_MultiPoint/ST_MultiPoint, and ST_MultiPolygon/ST_MultiPolygon overlays. For these combinations, the overlap predicate returns true if the interior of both geometries intersects the other's interior and exterior.

| | | | **Geometry b** | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | T | * | T |
| | **Boundary** | * | * | * |
| | **Exterior** | T | * | * |

*ST_Overlaps matrix 1*

The following pattern matrix applies to ST_LineString/ST_LineString and ST_MultiLineString/ST_MultiLineString overlaps. In this case, the intersection of the geometries must result in a geometry that has a dimension of 1 (another ST_LineString or ST_MultiLineString). If the dimension of the intersection of the interiors resulted in 0 (a point), the ST_Overlaps predicate would return false. However, the ST_Crosses predicate would have returned true.

| | | | **Geometry b** | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | 1 | * | T |
| | **Boundary** | * | * | * |
| | **Exterior** | T | * | * |

*ST_Overlaps matrix 2*

## ST_Relate

ST_Relate returns a value of 1 or t (true) if the spatial relationship specified by the pattern matrix is valid. A value of 1 or t (true) indicates that some sort of spatial relationship exists between the geometries.

If the interiors or boundaries of geometries a and b are related in any way, ST_Relate is true. Whether or not the exteriors of one geometry intersect the interior or boundary of the other is irrelevant.

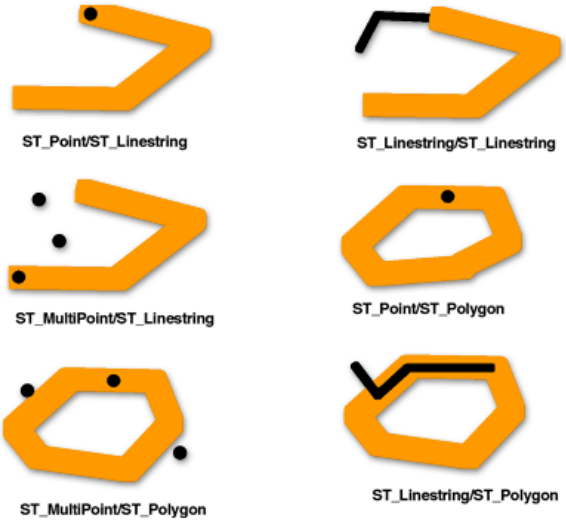
| | | | **Geometry b** | |
|---|---|---|---|---|

*ST_Relate matrix*

| | | Interior | Boundary | Exterior |
|---|---|---|---|---|
| **Geometry a** | **Interior** | T | T | * |
| | **Boundary** | T | T | * |
| | **Exterior** | * | * | * |

## ST_Touches

ST_Touches returns 1 or t (true) if none of the points common to both geometries intersect the interiors of both geometries. At least one geometry must be an ST_LineString, ST_Polygon, ST_MultiLineString, or ST_MultiPolygon.




The pattern matrices show that the ST_Touches predicate is true when the interiors of the geometry do not intersect, and the boundary of either geometry intersects the other's interior or boundary.

The ST_Touches predicate returns true if the boundary of geometry b intersects the interior of geometry a, but the interiors do not intersect.

| | | | Geometry b | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | F | T | * |
| | **Boundary** | * | * | * |
| | **Exterior** | * | * | * |

*ST_Touches matrix 1*

The ST_Touches predicate returns true if the boundary of geometry a intersects the interior of geometry b, but the interiors do not intersect.

| | | | Geometry b | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | F | * | * |

*ST_Touches matrix 2*

| | | | | |
|---|---|---|---|---|
| | **Boundary** | T | * | * |
| | **Exterior** | * | * | * |

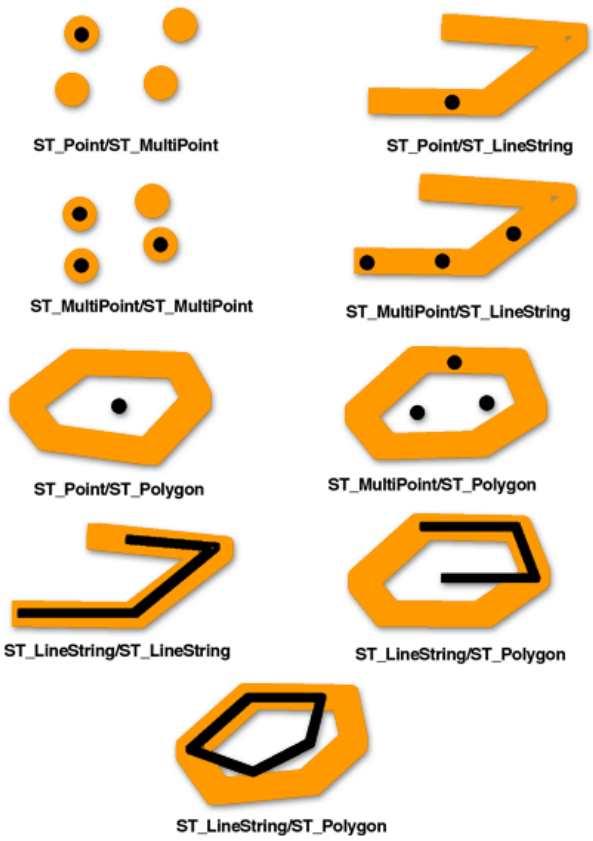
The ST_Touches predicate returns true if the boundaries of both geometries intersect, but the interiors do not.

| | | | **Geometry b** | |
|---|---|---|---|---|
| | | **Interior** | **Boundary** | **Exterior** |
| **Geometry a** | **Interior** | F | * | * |
| | **Boundary** | * | T | * |
| | **Exterior** | * | * | * |

*ST_Touches matrix 3*

## ST_Within

ST_Within returns 1 or t (true) if the first geometry is completely within the second geometry. ST_Within tests for the exact opposite result of ST_Contains.




The ST_Within predicate pattern matrix states that the interiors of both geometries must intersect and that the interior and boundary of the primary geometry (geometry a) must not intersect the exterior of the secondary geometry (geometry b).

| | | | **Geometry b** | |
|---|---|---|---|---|

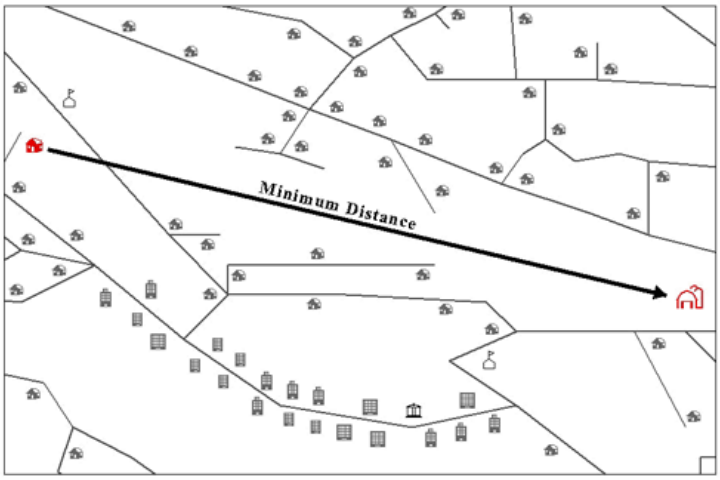
*ST_Within matrix*

| | | Interior | Boundary | Exterior |
|---|---|---|---|---|
| **Geometry a** | **Interior** | T | * | F |
| | **Boundary** | * | * | F |
| | **Exterior** | * | * | * |

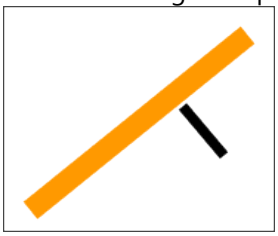
## Other spatial relationships

The following functions compare the spatial relationship between geometries, but they compare more than just the interior, boundary, and exteriors of the geometries.

- ST_Distance—This function takes two disjoint geometries as input and returns the minimum distance between the two. If the geometries are not disjoint (in other words, they are coincident), the function reports a zero minimum distance.
  The minimum distance separating features represents the shortest distance between two locations. For instance, this is not the distance you would travel if you drive from one location to the other, but the distance you would calculate if you drew a straight line between two locations on a map.




- ST_DWithin—You provide a distance value along with the geometries to be compared. ST_DWithin returns true if the geometries are located within the specified distance of one another.
- ST_EnvIntersects—This function evaluates whether the spatial envelopes of the specified geometries intersect; whereas ST_Intersects evaluates whether the geometries themselves intersect.
  In the following example, the envelopes of the two lines intersect but the lines themselves do not:



- ST_OrderingEquals—This function extends the comparison done by ST_Equals to also compare that the geometries' coordinates are defined in the same order (x,y versus y,x). Even if the geometries occupy the same space, if their x- and y-coordinates are not defined in the same order, ST_OrderingEquals returns false.

# Spatial operations

Spatial operations use geometry functions to take spatial data as input, analyze the data, then produce output data that is the derivative of the analysis performed on the input data.

Derived data you can obtain from a spatial operation includes the following:

- A polygon that is a buffer around an input feature
- A single feature that is a result of analysis performed on a collection of geometries
- A single feature that is the result of a comparison to determine the part of a feature that does not inhabit the same physical space as another feature
- A single feature that is the result of a comparison to find the parts of a feature that intersect the physical space of another feature
- A multipart feature that is made up of the parts of both input features that do not inhabit the same physical space as one another
- A feature that is the union of two geometries

The analysis performed on the input data returns the coordinates or text representation of the resultant geometries. You can use that information as part of a larger query to perform further analysis, or you can use the results as input to another table.

For example, you could include a buffer operation in the `WHERE` clause of an intersect query to determine whether the specified geometry intersects an area of specified size around another geometry.

**Note:**
The following examples use ST_Geometry functions. For the specific geometry functions and syntax used for another database and spatial data type, read the documentation specific to that database and data type.

In this example, notifications have to be sent to all property owners within 1,000 feet of a street closure. The `WHERE` clause generates a 1,000-foot buffer around the street that will be closed. That buffer is then compared to the properties in the area to see which ones are intersected by the buffer.

```
SELECT p.owner,p.address,s.stname
FROM parcels p, streets s
WHERE s.stname = 'Main'
AND sde.st_intersects (p.shape, sde.st_buffer (s.shape, 1000)) = 't';
```

In this example, one specific street (Main) is chosen in the `WHERE` clause, then a buffer is created around the street and compared to the features in the parcels table to determine whether they intersect.* For all parcels that are intersected by the buffer on Main Street, the parcel owner name and address are returned.

**Note:**
*The order in which the parts of the `WHERE` clause are run depends on the database optimizer.

The following is an example of taking the results of a spatial operation (union) performed on tables containing neighborhood and school district areas and inserting the resultant features into another table:

```
INSERT INTO combo c (shape)
VALUES (
(SELECT sde.st_union (n.shape,d.shape)
FROM neighborhoods n, school_districts d),5);
```

For more information on using spatial operators with ST_Geometry, see Spatial operation functions for ST_Geometry.

# Spatial operation functions for ST_Geometry

Spatial operations use geometry functions to take spatial data as input, analyze the data, then produce output data that is the derivative of the analysis performed on the input data.

You can perform the operations described in the following sections to create features from input features.

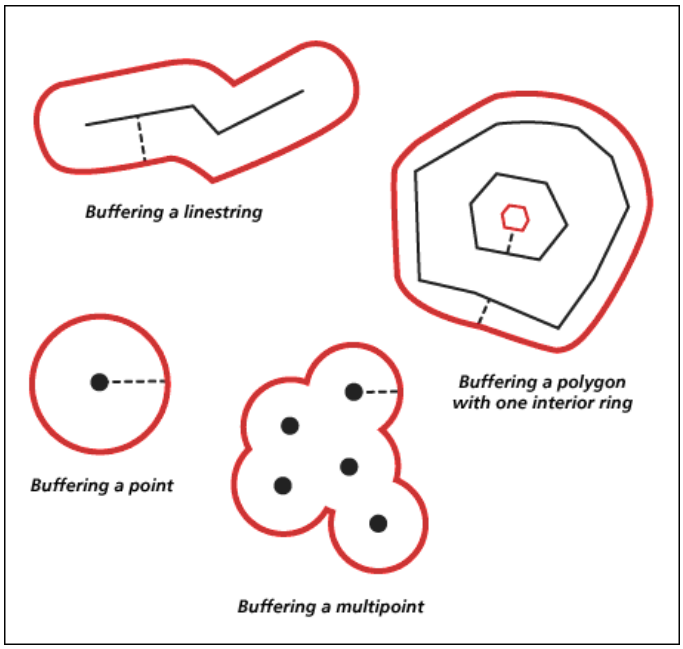
## Buffering geometry

The ST_Buffer function generates a geometry by encircling the geometry you specify at the distance that you specify. A single polygon results when a primary geometry is buffered or when the buffer polygons of a collection are close enough to overlap. When enough separation exists between the elements of a buffered collection, individual buffer ST_Polygons result in an ST_MultiPolygon.

The ST_Buffer function accepts both positive and negative distances, but you can apply negative distances to two-dimension geometries only (ST_Polygon and ST_MultiPolygon). ST_Buffer uses the absolute value of the buffer distance when the source geometry has fewer than two dimensions, in other words, all geometries that are not ST_Polygon or ST_MultiPolygon. Positive buffer distances generate polygon rings that are away from the center of the source geometry and—for the exterior ring of an ST_Polygon or ST_MultiPolygon—toward the center when the distance is negative. For interior rings of an ST_Polygon or ST_MultiPolygon, the buffer ring is toward the center when the buffer distance is positive and away from the center when it is negative.

The buffering process merges buffer polygons that overlap. Negative distances greater than one-half the maximum interior width of a polygon result in an empty geometry.

In the following diagram, buffers are drawn in red.




## ConvexHull

The ST_ConvexHull function returns the convex hull polygon of any geometry that has at least three vertices forming a convex. If vertices of the geometry do not form a convex, ST_ConvexHull returns a null. For example, using ST_ConvexHull on a line composed of two vertices will return a null. Similarly, using the ST_ConvexHull

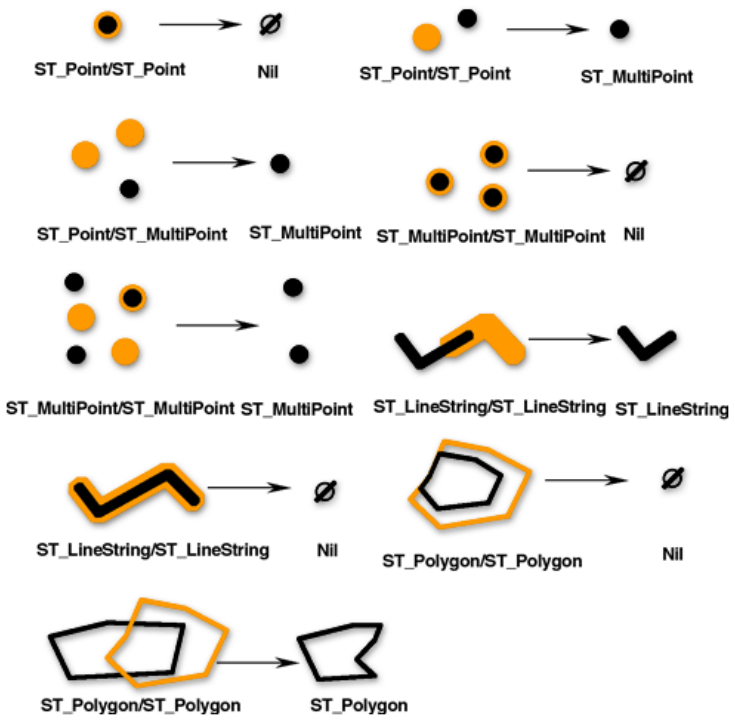
operation on a point feature will return a null. Creating a convex hull is often the first step when tessellating a set of points to create a triangulated irregular network (TIN).

## Difference of geometries

The ST_Difference function returns the portion of the primary geometry that is not intersected by the secondary geometry. This is the logical `AND NOT` of space.

The ST_Difference function operates on geometries of similar dimension only and returns a collection that has the same dimension as the source geometries. If the source geometries are equal, an empty geometry is returned.

In the diagram below, the first input geometries are black and the second input geometries are orange.

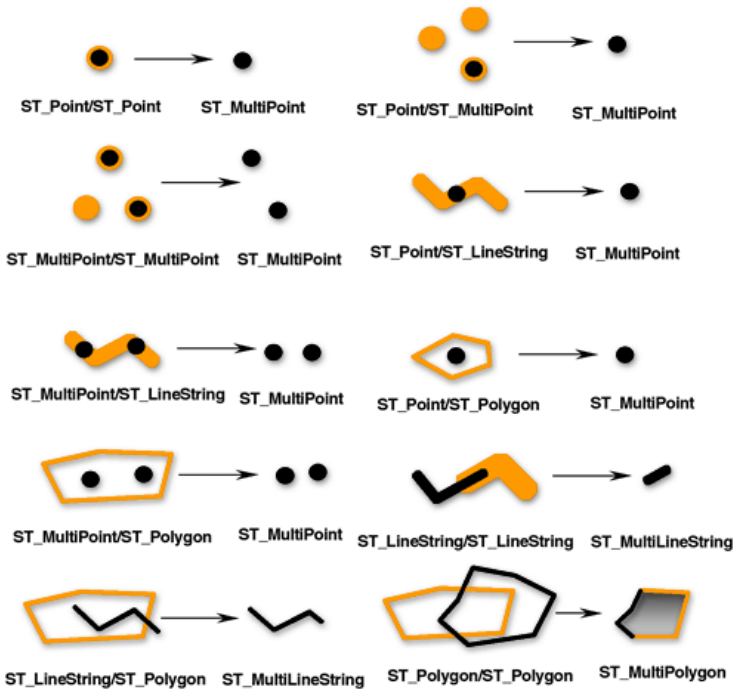



## Intersection of geometries

The ST_Intersection function returns the intersection set of two geometries. The intersection set is always returned as a collection that is the minimum dimension of the source geometries.

For example, for an ST_LineString that intersects an ST_Polygon, the ST_Intersection function returns that portion of the ST_LineString common to the interior and boundary of the ST_Polygon as an ST_MultiLineString. The ST_MultiLineString contains more than one ST_LineString if the source ST_LineString intersected the ST_Polygon with two or more discontinuous segments. If the geometries do not intersect or if the intersection results in a dimension less than both source geometries, an empty geometry is returned.

The following figure illustrates examples of the ST_Intersection function. The first input geometries are black and the second input geometries are orange.

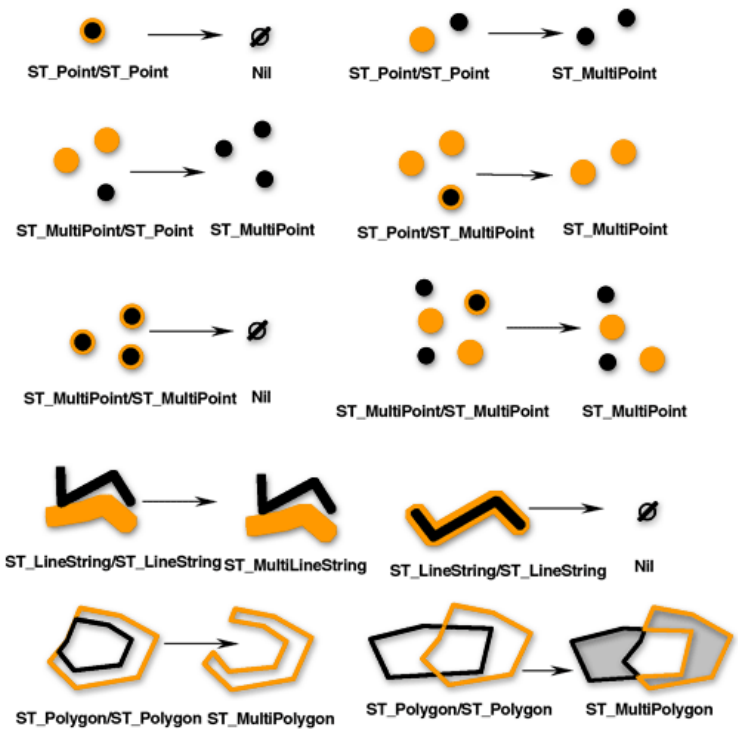



## Symmetric difference of geometries

The ST_SymmetricDiff function returns the portions of the source geometries that are not part of the intersection set. This is the logical `XOR` of space.

The source geometries must be of the same dimension. If the geometries are equal, the ST_SymmetricDiff function returns an empty geometry; otherwise, the function returns the result as a collection.

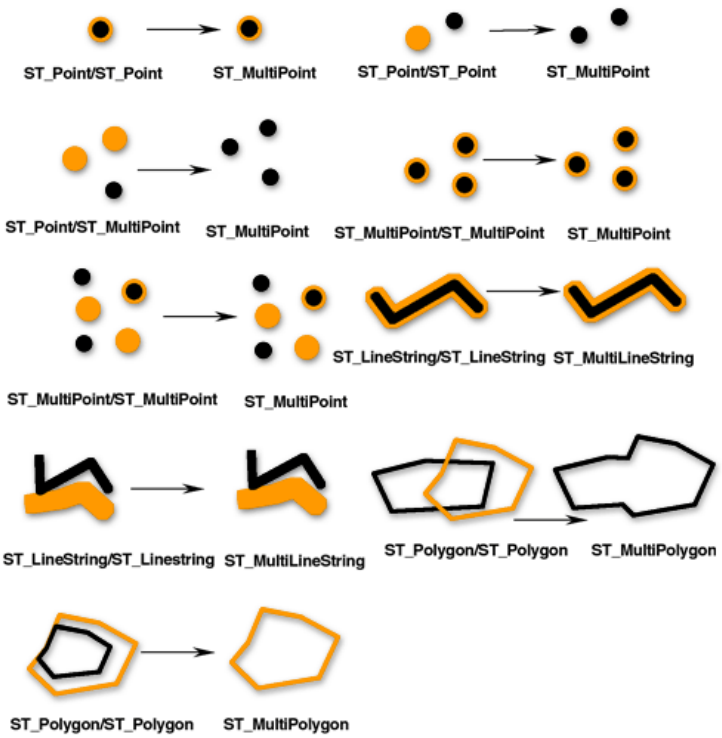
In the diagram below, the first input geometries are black and the second input geometries are orange.




## Union of geometries

The ST_Union function returns the union set of two geometries. This is the Boolean logical `OR` of space. The source geometries must have the same dimension. ST_Union always returns the result as a collection.

In the diagram below, the first input geometries are black and the second input geometries are orange.




## Aggregates

Aggregate operations return a single geometry as a result of analysis performed on a collection of geometries. The ST_Aggr_ConvexHull function returns the multipolygon composed of the convex hull polygons of each of the input geometries. Any input geometry with fewer than three vertices will not have a convex hull. If all of the input geometries have fewer than three vertices, ST_Aggr_ConvexHull returns a null.

The ST_Aggr_Intersection function returns a single geometry that is an aggregation of the intersections of all input geometries.

ST_Aggr_Intersection finds the intersection of multiple geometries, whereas ST_Intersection only finds the intersection between two geometries. For example, if you wanted to find property that was covered by different specific services—such as a specific school district, phone service, and high-speed internet provider, and was represented by a specific council person—you need to find the intersection of all those areas. Finding the intersection of only two of those areas would not return all of the information you need, so you would use the ST_Aggr_Intersection function so that all the areas could be evaluated in the same query.

As a further example, when finding the intersections of lines and points in two feature classes, each function would return the following:

- ST_Intersection—An ST_Point geometry is returned for every intersection.
- ST_Aggr_Intersection—One ST_MultiPoint geometry is returned that is composed of all the points of intersection. (However, if only one point feature and one line feature intersect, you will get an ST_Point geometry.)

The ST_Aggr_Union function returns one geometry that is the union of all the provided geometries.

The input geometries must be of the same type; for example, you can union ST_LineStrings with ST_LineStrings or you can union ST_Polygons with ST_Polygons, but you cannot union an ST_LineString feature class with an ST_Polygon feature class.

The geometry that results from the aggregate union is usually a collection. For example, if you want the aggregate union of all the vacant parcels smaller than half an acre, the returned geometry will be a multipolygon, unless all the parcels that meet the criteria are contiguous, then one polygon would be returned.

## Minimum distance

The previous functions returned new geometries. The ST_Distance function performs a spatial operation—it evaluates the minimum distance between two geometries—but it does not return a new geometry.

# Parametric circles, ellipses, and wedges

You can create and query parametric circles, ellipses, or wedges in ST_Geometry columns using the ST_Geometry function.

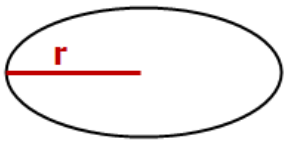
Parametric circles, ellipses, and wedges are polygons that are defined by specific parameters, such as coordinate values, angles, and radii. The database stores these parameters instead of specific vertices and lines. By storing the parameters that define the shape, parametric shapes can be more accurate and use less storage space than storing them as many-sided polygon representations. The use of parametric shapes also allows the inclusion of z-coordinate and measure (m)-value parameters.

Seven parameters are expected when creating a circle:

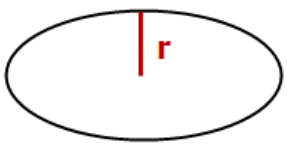
- An x-coordinate value of the center point of the circle
- A y-coordinate value of the center point of the circle
- A z-coordinate value of the center point of the circle
- An m-value
- The radius of the circle to be created
- The number of points used to define the circle
  The minimum number of points you can specify is 9. If you do not specify a number of points, 50 are used by default. These points are not stored with the shape but are generated when the circle is generated to validate the shape.
- The spatial reference ID (SRID) used to place the circle in space

Nine parameters are expected when creating an ellipse:

- An x-coordinate value of the center point of the ellipse
- A y-coordinate value of the center point of the ellipse
- A z-coordinate value of the center point of the ellipse
- An m-value
- The semimajor axis of the ellipse
  The semimajor axis is the longest radius of an ellipse. The value specified for the semimajor axis must be greater than the semiminor axis.



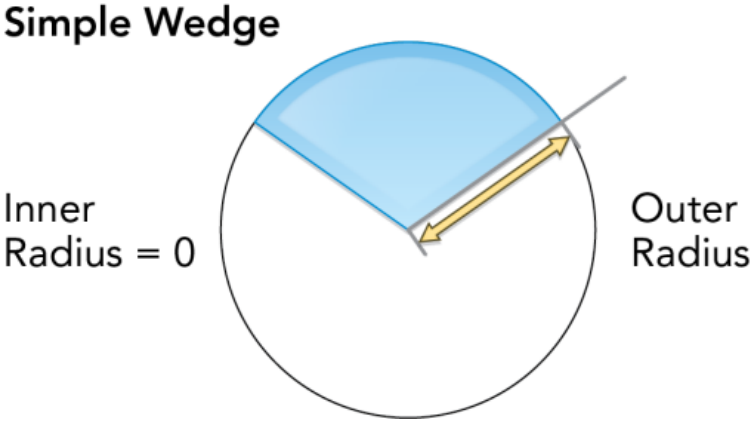

- The semiminor axis of the ellipse
  The semiminor axis is the shortest radius of an ellipse. The value specified for the semiminor axis must be greater than 0.0.



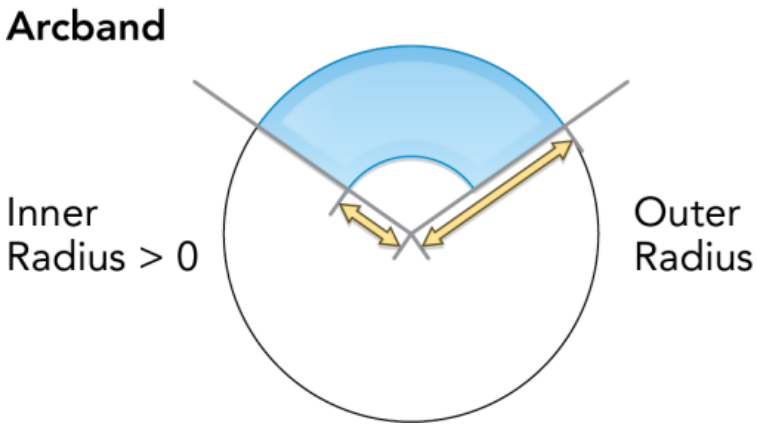
- The angle of rotation of the ellipse
  The value specified for the angle of rotation is specified in degrees and must be greater than 0.0 but less than 360. Rotation is in a clockwise direction.
- The number of points used to define the ellipse
  The minimum number of points you can specify is 9. If you do not specify a number of points, 50 points are used by default. These points are not stored with the shape but are generated when the ellipse is generated to validate the shape.
- The SRID used to place the ellipse in space

Ten parameters are expected when creating a wedge:

- An x-coordinate value of the center point of the circle that defines the wedge
- A y-coordinate value of the center point of the circle that defines the wedge
- A z-coordinate value of the center point of the circle that defines the wedge
- An m-value
- The start angle of the wedge
  The start angle defines the beginning of the wedge as the number of degrees measured counterclockwise from 0 degrees.
- The end angle of the wedge
  The end angle defines the end of the wedge as the number of degrees measured counterclockwise from 0 degrees.
- The outer radius
  The outer radius defines the distance from the center of a circle to the outermost point of the wedge.
- The inner radius
  The inner radius defines the distance from the center of a circle to the innermost point of the wedge, thereby defining the start of the wedge. If the inner radius is 0, the shape is a simple wedge.




If the inner radius is greater than 0, the wedge is technically an arcband.




- The number of points used to define the wedge
  The minimum number of points you can specify is 9. If you do not specify a number of points, 80 points are used by default. These points are not stored with the shape but are generated when the wedge is generated to validate the shape.
- The SRID used to place the wedge in space

All radii, including the semimajor and semiminor axes and inner and outer radii, are defined in the units determined by the coordinate reference specified with the SRID.

See the ST_Geometry function for syntax and examples to create a parametric circle, ellipse, or wedge.

# ST_Aggr_ConvexHull

**Note:**
Oracle and SQLite only

## Definition

ST_Aggr_ConvexHull creates a single geometry that is a convex hull of a geometry that resulted from a union of all input geometries. In effect, ST_Aggr_ConvexHull is equivalent to ST_ConvexHull(ST_Aggr_Union(geometries)).

## Syntax

### Oracle

```
sde.st_aggr_convexhull (geometry sde.st_geometry)
```

### SQLite

```
st_aggr_convexhull (geometry st_geometry)
```

## Return type

### Oracle

ST_Geometry

### SQLite

Geometryblob

## Example

The example creates a service_territories table and runs a `SELECT` statement that aggregates all the geometries, thereby generating a single geometry representing the convex hull of the union of all the shapes.

### Oracle

```
CREATE TABLE service_territories
 (ID integer not null, UNITS number, SHAPE sde.st_geometry);

INSERT INTO service_territories (id, units, shape) VALUES (
 1,
 1250,
 sde.st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO service_territories (id, units, shape) VALUES (
 2,
 875,
 sde.st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);
```

```
INSERT INTO service_territories (id, units, shape) VALUES (
 3,
 1700,
 sde.st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

SELECT sde.st_astext(sde.st_aggr_convexhull(shape)) CONVEX_HULL
 FROM service_territories
 WHERE units >= 1000;

CONVEX_HULL

POLYGON  (( 20.00000000 40.00000000, 20.00000000 30.00000000, 30.00000000 30.00000000,
60.00000000 40.00000000, 60.00000000 60.00000000, 40.00000000 60.00000000, 20.00000000
40.00000000))
```

## SQLite

```
CREATE TABLE service_territories (
  ID integer primary key autoincrement not null,
  UNITS numeric
);

SELECT AddGeometryColumn(
 NULL,
 'service_territories',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO service_territories (units, shape) VALUES (
 1250,
 st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO service_territories (units, shape) VALUES (
 875,
 st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO service_territories (units, shape) VALUES (
 1700,
 st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

SELECT st_astext(st_aggr_convexhull(shape)) AS "CONVEX HULL"
 FROM service_territories
 WHERE units >= 1000;

CONVEX HULL

POLYGON  (( 20.00000000 40.00000000, 20.00000000 30.00000000, 30.00000000 30.00000000,
60.00000000 40.00000000, 60.00000000 60.00000000, 40.00000000 60.00000000, 20.00000000
40.00000000))
```

# ST_Aggr_Intersection

**Note:**
Oracle and SQLite only

## Definition

ST_Aggr_Intersection returns a single geometry that is a union of the intersection of all input geometries.

## Syntax

### Oracle

```
sde.st_aggr_intersection (geometry1 sde.st_geometry)
```

### SQLite

```
st_aggr_intersection (geometry1 geometryblob)
```

## Return type

### Oracle

ST_Geometry

### SQLite

Geometryblob

## Example

In this example, a biologist is trying to find the intersection of three wildlife habitats.

### Oracle

First, create the table that stores the habitats.

```
CREATE TABLE habitats (
 id integer not null,
 shape sde.st_geometry
);
```

Next, insert the three polygons to the table.

```
INSERT INTO habitats (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((5 5, 12 5, 12 10, 5 10, 5 5))', 4326)
);

INSERT INTO habitats (id, shape) VALUES (
 2,
```

```
 sde.st_polygon ('polygon ((10 8, 14 8, 14 15, 10 15, 10 8))', 4326)
);

INSERT INTO habitats (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((6 8, 20 8, 20 20, 6 20, 6 8))', 4326)
);
```

Finally, select for the intersection of the habitats.

```
SELECT sde.st_astext(sde.st_aggr_intersection(shape)) AGGR_SHAPES
 FROM habitats;

AGGR_SHAPES

POLYGON  (( 10.00000000 8.00000000, 12.00000000 8.00000000, 12.00000000 10.00000000,
10.00000000 10.00000000, 10.00000000 8.00000000))
```

## SQLite

First, create the table that stores the habitats.

```
CREATE TABLE habitats (
  id integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'habitats',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);
```

Next, insert the three polygons to the table.

```
INSERT INTO habitats (shape) VALUES (
 st_polygon ('polygon ((5 5, 12 5, 12 10, 5 10, 5 5))', 4326)
);

INSERT INTO habitats (shape) VALUES (
 st_polygon ('polygon ((10 8, 14 8, 14 15, 10 15, 10 8))', 4326)
);

INSERT INTO habitats (shape) VALUES (
 st_polygon ('polygon ((6 8, 20 8, 20 20, 6 20, 6 8))', 4326)
);
```

Finally, select the intersection of the habitats.

```
SELECT st_astext(st_aggr_intersection(shape))
  AS "AGGR_SHAPES"
  FROM habitats;
```

```
AGGR_SHAPES

POLYGON  (( 10.00000000 8.00000000, 12.00000000 8.00000000, 12.00000000 10.00000000,
10.00000000 10.00000000, 10.00000000 8.00000000))
```

# ST_Aggr_Union

## Definition

ST_Aggr_Union returns a single geometry that is the union of all input geometries.

## Syntax

### Oracle and PostgreSQL

```
sde.st_aggr_union(geometry sde.st_geometry)
```

### SQLite

```
st_aggr_union(geometry geometryblob)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

A marketing analyst needs to create a single geometry of all the service areas for which sales exceeded 1,000 units. For this example, you will create a service_territories1 table and populate it with sales value numbers. You will then use st_aggr_union in a SELECT statement to return the multipolygon that is the union of all geometries for which sales numbers are equal to or greater than 1,000 units.

### Oracle and PostgreSQL

```
--Create and populate tables.
CREATE TABLE service_territories1 (
 ID integer not null,
 UNITS number,
 SHAPE sde.st_geometry);

INSERT INTO service_territories1 (id, units, shape) VALUES (
 1,
 1250,
 sde.st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO service_territories1 (id, units, shape) VALUES (
 2,
 875,
 sde.st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);
```

```
INSERT INTO service_territories1 (id, units, shape) VALUES (
 3,
 1700,
 sde.st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
```

```
--Union of all geometries for which sales numbers are >= 1,000 units.
SELECT sde.st_astext(sde.st_aggr_union(shape)) UNION_SHAPE
  FROM service_territories1
  WHERE units >= 1000;

UNION_SHAPE

MULTIPOLYGON  ((( 20.00000000 30.00000000, 30.00000000 30.00000000, 30.00000000
40.00000000, 20.00000000 40.00000000, 20.00000000 30.00000000)),(( 40.00000000
40.00000000,
60.00000000 40.00000000, 60.00000000 60.00000000, 40.00000000 60.00000000,
40.00000000 40.00000000)))
```

## SQLite

```
--Create table, add geometry column to it, and populate table.
CREATE TABLE service_territories1 (
 id integer primary key autoincrement not null,
 units number
);

SELECT AddGeometryColumn(
 NULL,
 'service_territories1',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO service_territories1 (units, shape) VALUES (
 1250,
 st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO service_territories1 (units, shape) VALUES (
 875,
 st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO service_territories1 (units, shape) VALUES (
 1700,
 st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
```

```
--Union of all geometries for which sales numbers are >= 1,000 units.
SELECT st_astext(st_aggr_union(shape))
 AS "UNION_SHAPE"
 FROM service_territories1
 WHERE units >= 1000;
```

```
UNION_SHAPE

MULTIPOLYGON ((( 40.00000000 40.00000000, 60.00000000 40.00000000, 60.00000000 6
0.00000000, 40.00000000 60.00000000, 40.00000000 40.00000000)),(( 20.00000000 30
.00000000, 30.00000000 30.00000000, 30.00000000 40.00000000, 20.00000000 40.0000
0000, 20.00000000 30.00000000)))
```

# ST_Area

## Definition

ST_Area returns the area of a polygon or multipolygon.

## Syntax

### Oracle and PostgreSQL

```
sde.st_area (polygon sde.st_geometry)
sde.st_area (multipolygon sde.st_geometry)
```

### SQLite

```
st_area (polygon st_geometry)
st_area (polygon st_geometry, unit_name)
```

## Return type

Double precision

## Example

The city engineer needs a list of building areas. To create the list, a GIS technician selects the building ID and area of each building's footprint.

The building footprints are stored in the bfp table.

To satisfy the city engineer's request, the technician selects the unique key, the building_id, and the area of each building footprint from the bfp table.

## Oracle

```
--Create and populate table.
CREATE TABLE bfp (
 building_id integer not null,
 footprint sde.st_geometry);

INSERT INTO BFP (building_id, footprint) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO BFP (building_id, footprint) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO BFP (building_id, footprint) VALUES (
 3,
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--Get area of geometries.
SELECT building_id, sde.st_area (footprint) Area
 FROM BFP;

BUILDING_ID       Area

          1        100
          2        200
          3         25
```

## PostgreSQL

```
--Create and populate table.
CREATE TABLE bfp (
 building_id serial,
 footprint sde.st_geometry);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--Get area of geometries.
SELECT building_id, sde.st_area (footprint)
 AS Area
 FROM bfp;

building_id       area
```

```
          1        100
          2        200
          3         25
```

## SQLite

```
--Create table, add geometry column to it, and populate the table.
CREATE TABLE bfp (
 building_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'bfp',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--Get area of geometries.
SELECT building_id, st_area (footprint)
 AS "area"
 FROM bfp;

building_id       area

          1        100.0
          2        200.0
          3         25.0
```

# ST_AsBinary

## Definition

ST_AsBinary takes a geometry object and returns its well-known binary representation.

## Syntax

### Oracle and PostgreSQL

```
sde.st_asbinary (geometry sde.st_geometry)
```

### SQLite

```
st_asbinary (geometry geometryblob)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

This example populates the WKB column of record 1111 with the contents from the GEOMETRY column of record 1100.

### Oracle

```
CREATE TABLE sample_points (
 id integer not null,
 geometry sde.st_geometry,
 wkb blob
);

INSERT INTO SAMPLE_POINTS (id, geometry) VALUES (
 1100,
 sde.st_geometry ('point (10 20)', 4326)
);

INSERT INTO SAMPLE_POINTS (id, wkb) VALUES (
 1111,
 (SELECT sde.st_asbinary (geometry) FROM sample_points WHERE id = 1100)
);

SELECT id, sde.st_astext (sde.st_geomfromwkb (wkb, 4326))
 FROM SAMPLE_POINTS
 WHERE id = 1111;
```

```
ID              Point
1111     POINT (10.00000000 20.00000000)
```

## PostgreSQL

```
CREATE TABLE sample_points (
 id serial,
 geometry sde.st_geometry,
 wkb bytea);

INSERT INTO sample_points (geometry) VALUES (
 sde.st_point (10, 20, 4326)
);

INSERT INTO sample_points (wkb) VALUES (
 (SELECT sde.st_asbinary (geometry) FROM sample_points WHERE id = 1100)
);

SELECT id, sde.st_astext (sde.st_geomfromwkb (wkb, 4326))
 FROM sample_points
 WHERE id = 1111;

ID              st_astext
1111     POINT (10 20)
```

## SQLite

```
CREATE TABLE sample_points (
 id integer primary key autoincrement not null,
 wkb blob
);

SELECT AddGeometryColumn(
 NULL,
 'sample_points',
 'geometry',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO sample_points (geometry) VALUES (
 st_point (10, 20, 4326)
);

INSERT INTO sample_points (wkb) VALUES (
 (SELECT st_asbinary (geometry) FROM sample_points WHERE id = 1)
);

SELECT id, st_astext (st_geomfromwkb (wkb, 4326))
 FROM sample_points
 WHERE id = 2;

ID              st_astext
2        POINT (10.00000000 20.00000000)
```

# ST_AsText

## Definition

ST_AsText takes a geometry and returns its well-known text representation.

## Syntax

### Oracle and PostgreSQL

```
sde.st_astext (geometry sde.st_geometry)
```

### SQLite

```
st_astext (geometry geometryblob)
```

## Return type

### Oracle

CLOB

### PostgreSQL and SQLite

Text

## Example

The ST_AsText function converts the hazardous_sites location point into its text description.

### Oracle

```
CREATE TABLE hazardous_sites (
 site_id integer not null,
 name varchar(40),
 loc sde.st_geometry);

INSERT INTO HAZARDOUS_SITES (site_id, name, loc) VALUES (
 102,
 'W. H. KleenareChemical Repository',
 sde.st_geometry ('point (1020.12 324.02)', 4326)
);

SELECT site_id, name, sde.st_astext (loc) Location
 FROM HAZARDOUS_SITES;

SITE_ID      NAME                              Location

   102    W. H. KleenareChemical Repository   POINT (1020.12000000 324.02000000)
```

## PostgreSQL

```
CREATE TABLE hazardous_sites (
 site_id serial,
 name varchar(40),
 loc sde.st_geometry);

INSERT INTO hazardous_sites (name, loc) VALUES (
 'W. H. KleenareChemical Repository',
 sde.st_point ('point (1020.12 324.02)', 4326)
);

SELECT site_id, name, sde.st_astext (loc)
 AS location
 FROM hazardous_sites;

site_id      name                              location

   102    W. H. KleenareChemical Repository   POINT (1020.12000001 324.01999999)
```

## SQLite

```
CREATE TABLE hazardous_sites (
 site_id integer primary key autoincrement not null,
 name varchar(40)
);

SELECT AddGeometryColumn(
 NULL,
 'hazardous_sites',
 'loc',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO hazardous_sites (name, loc) VALUES (
 'W. H. KleenareChemical Repository',
 st_point ('point (1020.12 324.02)', 4326)
);

SELECT site_id, name, st_astext (loc)
 FROM hazardous_sites;

   1    W. H. KleenareChemical Repository   POINT (1020.12000000 324.02000000)
```

# ST_Boundary

## Definition

ST_Boundary takes a geometry and returns its combined boundary as a geometry object.

## Syntax

### Oracle and PostgreSQL

```
sde.st_boundary (geometry sde.st_geometry)
```

### SQLite

```
st_boundary (geometry geometryblob)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

In this example, the boundaries table is created with two columns: type and geometry. The subsequent INSERT statements add one record for each one of the subclass geometries. The ST_Boundary function retrieves the boundary of each subclass stored in the geometry column. Note that the dimension of the resulting geometry is always one less than the input geometry. Points and multipoints always result in a boundary that is an empty geometry, dimension –1. Linestrings and multilinestrings return a multipoint boundary, dimension 0. A polygon or multipolygon always returns a multilinestring boundary, dimension 1.

### Oracle

```
CREATE TABLE boundaries (
 geotype varchar(20),
 geometry sde.st_geometry
);

INSERT INTO BOUNDARIES VALUES (
 'Point',
 sde.st_pointfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO BOUNDARIES VALUES (
 'Linestring',
 sde.st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO BOUNDARIES VALUES (
```

```
 'Polygon',
 sde.st_polyfromtext ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01))', 4326)
);

INSERT INTO BOUNDARIES VALUES (
 'Multipoint',
 sde.st_mpointfromtext ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))',
4326)
);

INSERT INTO BOUNDARIES VALUES (
 'Multilinestring',
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 0)
);

INSERT INTO BOUNDARIES VALUES (
 'Multipolygon',
 sde.st_mpolyfromtext ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15
33.94, 10.02 20.01), (51.71 21.73,73.36 27.04,71.52 32.87, 52.43 31.90, 51.71
21.73)))', 4326)
);

SELECT geotype, sde.st_astext (sde.st_boundary (geometry)) "The boundary"
 FROM BOUNDARIES;

GEOTYPE            The boundary

Point              POINT EMPTY
Linestring         MULTIPOINT((10.02000000 20.01000000), (11.92000000 25.64000000))
Polygon            MULTILINESTRING ((10.02000000 20.01000000, 19.15000000
33.94000000,25.02000000 34.15000000, 11.92000000 35.64000000, 10.02000000 20.01000000))
Multipoint         POINT EMPTY
Multilinestring    MULTIPOINT ((9.55000000 23.75000000), (10.02000000 20.01000000),
(11.92000000 25.64000000), (15.36000000 30.11000000))
Multipolygon       MULTILINESTRING((51.71000000 21.73000000, 73.36000000 27.04000000,
71.52000000 32.87000000, 52.43000000 31.90000000, 51.71000000 21.73000000),
(10.02000000 20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000, 11.92000000
35.64000000, 10.02000000 20.01000000))
```

## PostgreSQL

```
CREATE TABLE boundaries (
 geotype varchar(20),
 geometry st_geometry
);

INSERT INTO boundaries VALUES (
 'Point',
 st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO boundaries VALUES (
 'Linestring',
 st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO boundaries VALUES (
 'Polygon',
 st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
```

```
20.01))', 4326)
);

INSERT INTO boundaries VALUES (
'Multipoint',
st_multipoint ('multipoint (10.02 20.01, 10.32 23.98, 11.92 25.64)', 0)
);

INSERT INTO boundaries VALUES (
 'Multilinestring',
 st_multilinestring ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO boundaries VALUES (
 'Multipolygon',
 st_multipolygon ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01),
(51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90, 51.71 21.73)))', 4326)
);

SELECT geotype, st_astext (st_boundary (geometry))
 AS "The boundary"
 FROM boundaries;

geotype            The boundary

Point              EMPTY
Linestring         MULTIPOINT(10.02000000 20.01000000, 11.92000000 25.64000000)
Polygon            LINESTRING ((10.02000000 20.01000000, 19.15000000
33.94000000,25.02000000 34.15000000, 11.92000000 35.64000000, 10.02000000
20.01000000))
Multipoint         EMPTY
Multilinestring    MULTIPOINT (9.55000000 23.75000000, 10.02000000 20.01000000,
11.92000000 25.64000000, 15.36000000 30.11000000)
Multipolygon       MULTILINESTRING((51.71000000 21.73000000, 73.36000000 27.04000000,
71.52000000 32.87000000, 52.43000000 31.90000000,
51.71000000 21.73000000), (10.02000000 20.01000000, 19.15000000 33.94000000,
25.02000000 34.15000000, 11.92000000 35.64000000,
10.02000000 20.01000000))
```

## SQLite

```
CREATE TABLE boundaries (
 geotype varchar(20)
);

SELECT AddGeometryColumn (
 NULL,
 'boundaries',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO boundaries VALUES (
 'Point',
 st_point ('point (10.02 20.01)', 4326)
);
```

```
INSERT INTO boundaries VALUES (
 'Linestring',
 st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO boundaries VALUES (
 'Polygon',
 st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
);

INSERT INTO boundaries VALUES (
 'Multipoint',
 st_multipoint ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO boundaries VALUES (
 'Multilinestring',
 st_multilinestring ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO boundaries VALUES (
 'Multipolygon',
 st_multipolygon ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01),
(51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90, 51.71 21.73)))', 4326)
);

SELECT geotype, st_astext (st_boundary (geometry))
 FROM boundaries;

Point              EMPTY
Linestring         MULTIPOINT((10.02000000 20.01000000), (11.92000000 25.64000000))
Polygon            LINESTRING ((10.02000000 20.01000000, 19.15000000
33.94000000,25.02000000 34.15000000, 11.92000000 35.64000000, 10.02000000
20.01000000))
Multipoint         EMPTY
Multilinestring    MULTIPOINT ((9.55000000 23.75000000), (10.02000000 20.01000000),
(11.92000000 25.64000000), (15.36000000 30.11000000))
Multipolygon       MULTILINESTRING((51.71000000 21.73000000, 73.36000000 27.04000000,
71.52000000 32.87000000, 52.43000000 31.90000000,
51.71000000 21.73000000), (10.02000000 20.01000000, 19.15000000 33.94000000,
25.02000000 34.15000000, 11.92000000 35.64000000,
10.02000000 20.01000000))
```

# ST_Buffer

## Definition

ST_Buffer takes a geometry object and distance and returns a geometry object that is the buffer surrounding the source object.

## Syntax

`Unit_name` is the unit of measure for the buffer distance (for example, `meters`, `kilometers`, `feet`, or `mile`). See the first table in the `Projected coordinate system tables.pdf`, which you can access from Coordinate systems, projections, and transformations.

### Oracle

```
sde.st_buffer (geometry sde.st_geometry, distance double_precision)
sde.st_buffer (geometry sde.st_geometry, distance double, varchar2 unit_name)
```

### PostgreSQL

```
sde.st_buffer (geometry sde.st_geometry, distance double_precision)
sde.st_buffer (geometry sde.st_geometry, distance double, text unit_name)
```

### SQLite

```
st_buffer (geometry geometryblob, distance double_precision)
st_buffer (geometry geometryblob, distance double, text unit_name)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

This example creates two tables, sensitive_areas and hazardous_sites; populates the tables; uses ST_Buffer to generate a buffer around the polygons in the hazardous_sites table; and finds where these buffers overlap the sensitive_areas polygons.

### Oracle

```
CREATE TABLE sensitive_areas (
 id integer,
 zone sde.st_geometry
);
```

```
CREATE TABLE hazardous_sites (
 site_id integer,
 name varchar(40),
 location sde.st_geometry
);

INSERT INTO SENSITIVE_AREAS VALUES (
 1,
 sde.st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO SENSITIVE_AREAS VALUES (
 2,
 sde.st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO SENSITIVE_AREAS VALUES (
3,
sde.st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 0)
);

INSERT INTO HAZARDOUS_SITES VALUES (
 102,
 'W. H. KleenareChemical Repository',
 sde.st_pointfromtext ('point (60 60)', 4326)
);

SELECT sa.id "Sensitive Areas", hs.name "Hazardous Sites"
 FROM SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
 WHERE sde.st_overlaps (sa.zone, sde.st_buffer (hs.location, .01)) = 1;
```

## PostgreSQL

```
CREATE TABLE sensitive_areas (
 id serial,
 zone sde.st_geometry
);

CREATE TABLE hazardous_sites (
 site_id serial,
 name varchar(40),
 location sde.st_geometry
);

INSERT INTO sensitive_areas (zone) VALUES (
 sde.st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas (zone) VALUES (
 sde.st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sensitive_areas (zone) VALUES (
 sde.st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'W. H. KleenareChemical Repository',
 sde.st_point ('point (60 60)', 4326)
);
```

```
SELECT sa.id AS "Sensitive Areas", hs.name AS "Hazardous Sites"
 FROM sensitive_areas sa, hazardous_sites hs
 WHERE sde.st_overlaps (sa.zone, sde.st_buffer (hs.location, .01)) = 't';

Sensitive Areas               Hazardous Sites
             3               W.H. KleenareChemical Repository
```

## SQLite

```
CREATE TABLE sensitive_areas (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'sensitive_areas',
 'zone',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE hazardous_sites (
 site_id integer primary key autoincrement not null,
 name varchar(40)
);

SELECT AddGeometryColumn (
 NULL,
 'hazardous_sites',
 'location',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO sensitive_areas (zone) VALUES (
 st_polygon ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas (zone) VALUES (
 st_polygon ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sensitive_areas (zone) VALUES (
 st_polygon ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'W. H. KleenareChemical Repository',
 st_point ('point (60 60)', 4326)
);

SELECT sa.id AS "Sensitive Areas", hs.name AS "Hazardous Sites"
 FROM sensitive_areas sa, hazardous_sites hs
 WHERE st_overlaps (sa.zone, st_buffer (hs.location, .01)) = 1;
```

```
Sensitive Areas                Hazardous Sites
             3               W.H. KleenareChemical Repository
```

# ST_Centroid

## Definition

ST_Centroid takes a polygon, multipolygon, or multilinestring and returns the point that is in the center of the geometry's envelope. That means that the centroid point is halfway between the geometry's minimum and maximum x and y extents.

## Syntax

### Oracle and PostgreSQL

```
sde.st_centroid (polygon sde.st_geometry)
sde.st_centroid (multipolygon sde.st_geometry)
sde.st_centroid (multilinestring sde.st_geometry)
```

### SQLite

```
st_centroid (polygon geometryblob)
st_centroid (multipolygon geometryblob)
st_centroid (multilinestring geometryblob)
```

## Return type

ST_Point

## Example

The city GIS technician wants to display the building footprint multipolygons as single points in a building density graphic. The building footprints are stored in the bfp table that was created and populated with the statements shown for each database.

## Oracle

```
--Create and populate table
CREATE TABLE bfp (
 building_id integer,
 footprint sde.st_geometry);

INSERT INTO bfp VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO bfp VALUES (
 3,
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--The ST_Centroid function returns the centroid of each building footprint
multipolygon.
--The ST_AsText function converts each centroid point into a text representation
recognized by the application.
SELECT building_id,
 sde.st_astext (sde.st_centroid (footprint)) Centroid
 FROM bfp;

BUILDING_ID                  Centroid

   1                    POINT  (5.00000000 5.00000000)
   2                    POINT  (30.00000000 10.00000000)
   3                    POINT  (25.00000000 32.50000000)
```

## PostgreSQL

```
--Create and populate table
CREATE TABLE bfp (
 building_id serial,
 footprint sde.st_geometry);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--The ST_Centroid function returns the centroid of each building footprint
multipolygon.
```

```
--The ST_AsText function converts each centroid point into a text representation
recognized by the application.
SELECT building_id, sde.st_astext (sde.st_centroid (footprint))
 AS centroid
 FROM bfp;

building_id                 centroid

   1                    POINT  (5 5)
   2                    POINT  (30 10)
   3                    POINT  (25 33)
```

## SQLite

```
--Create table, add geometry column, and populate table
CREATE TABLE bfp (
 building_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'bfp',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);
```

```
--The ST_Centroid function returns the centroid of each building footprint
multipolygon.
--The ST_AsText function converts each centroid point into a text representation
recognized by the application.
SELECT building_id, st_astext (st_centroid (footprint))
 AS "centroid"
 FROM bfp;

building_id             centroid

   1                    POINT  (5.00000000 5.00000000)
   2                    POINT  (30.00000000 10.0000000)
   3                    POINT  (25.00000000 32.5000000)
```

# ST_Contains

## Definition

ST_Contains takes two geometry objects and returns 1 (Oracle and SQLite) or t (PostgreSQL) if the first object completely contains the second; otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_contains (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_contains (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

Boolean

## Example

In the examples below, two tables are created. One, buildingfootprints, contains a city's building footprints, while the other, lots, contains the parcels. The city engineer wants to ensure that all building footprints are completely inside their parcels.

The city engineer uses ST_Intersects and ST_Contains to select the buildings that are not completely contained within one lot.

### Oracle

```
--Create tables and insert values.
CREATE TABLE bfp (
 building_id integer,
 footprint sde.st_geometry
);

CREATE TABLE lots (
 lot_id integer,
 lot sde.st_geometry
);

INSERT INTO BFP (building_id, footprint) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO BFP (building_id, footprint) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO BFP (building_id, footprint) VALUES (
```

```
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO LOTS (lot_id, lot) VALUES (
 1,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO LOTS (lot_id, lot) VALUES (
 2,
 sde.st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO LOTS (lot_id, lot) VALUES (
 3,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
--Select the buildings that are not completely contained within one lot.
SELECT UNIQUE (building_id)
 FROM BFP, LOTS
 WHERE sde.st_intersects (lot, footprint) = 1
 AND sde.st_contains (lot, footprint) = 0;

BUILDING_ID

          2
```

## PostgreSQL

```
--Create tables and insert values.
CREATE TABLE bfp (
 building_id serial,
 footprint st_geometry);

CREATE TABLE lots
 (lot_id serial,
 lot st_geometry);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);
```

```
INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
--Select the buildings that are not completely contained within one lot.
SELECT DISTINCT (building_id)
 FROM bfp, lots
 WHERE st_intersects (lot, footprint) = 't'
 AND st_contains (lot, footprint) = 'f';

building_id

          2
```

## SQLite

```
--Create tables, add geometry columns, and insert values.
CREATE TABLE bfp (
 building_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'bfp',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE lots
 (lot_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'lots',
 'lot',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO lots (lot) VALUES (
```

```
 st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
--Select the buildings that are not completely contained within one lot.
SELECT DISTINCT (building_id)
 FROM bfp, lots
 WHERE st_intersects (lot, footprint) = 1
 AND st_contains (lot, footprint) = 0;

building_id

2
```

# ST_ConvexHull

## Definition

ST_ConvexHull returns the convex hull of an ST_Geometry object.

## Syntax

### Oracle and PostgreSQL

```
sde.st_convexhull (geometry1 sde.st_geometry)
```

### SQLite

```
st_convexhull (geometry1 geometryblob)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

These examples create the sample_geometries table with three columns: id, spatial_type, and geometry. The spatial_type field stores the type of geometry that is created in the geometry column. Three features—a linestring, a polygon, and a multipoint—are inserted into the table.

The `SELECT` statement that includes the ST_ConvexHull function returns the convex hull of each geometry.

### Oracle

```
--Create table and insert three sample geometries.
CREATE TABLE sample_geometries (
 id integer,
 spatial_type varchar(18),
 geometry sde.st_geometry
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 1,
 'ST_LineString',
 sde.st_geometry ('linestring (20 20, 30 30, 20 40, 30 50)', 4326)
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 2,
 'ST_Polygon',
 sde.st_geometry ('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))', 4326)
```

```
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 3,
 'ST_MultiPoint',
 sde.st_geometry ('multipoint ((20 20), (30 30), (20 40), (30 50))', 4326)
);
```

```
--Find the convex hull of each geometry subtype.
SELECT id, spatial_type, sde.st_astext (sde.st_convexhull (geometry)) CONVEXHULL
 FROM  SAMPLE_GEOMETRIES;

ID    SPATIAL_TYPE       CONVEXHULL

 1    ST_LineString      POLYGON ((20.00000000 40.00000000,
                           20.00000000 20.00000000, 30.00000000
                           30.00000000, 30.00000000 50.00000000,
                           20.00000000 40.00000000))

 2    ST_Polygon         POLYGON ((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))

 3    ST_MultiPoint      POLYGON ((20.00000000 40.00000000,
                           20.00000000 20.00000000, 30.00000000
                           30.00000000, 30.00000000 50.00000000,
                           20.00000000 40.00000000))
```

## PostgreSQL

```
--Create table and insert three sample geometries.
CREATE TABLE sample_geometries (
 id integer,
 spatial_type varchar(18),
 geometry sde.st_geometry
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 1,
 'ST_LineString',
 sde.st_geometry ('linestring (20 20, 30 30, 20 40, 30 50)', 4326)
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 2,
 'ST_Polygon',
 sde.st_geometry ('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))', 4326)
);

INSERT INTO sample_geometries (id, spatial_type, geometry) VALUES (
 3,
 'ST_MultiPoint',
 sde.st_geometry ('multipoint (20 20, 30 30, 20 40, 30 50)', 4326)
);
```

```
--Find the convex hull of each geometry subtype.
SELECT id, spatial_type, st_astext (sde.st_convexhull (geometry))
 AS CONVEXHULL
 FROM sample_geometries;

id    spatial_type       convexhull

 1    ST_LineString      POLYGON (( 20 40, 20 20, 30 30, 30 50,
                           20 40))

 2    ST_Polygon         POLYGON (( 15 50, 25 35, 30 30, 60 30,
                           75 40, 80 90, 40 85, 35 80, 15 50))

 3    ST_MultiPoint      POLYGON (( 20 40, 20 20, 30 30, 30 50,
                           20 40))
```

## SQLite

```
--Create table and insert three sample geometries.
CREATE TABLE sample_geometries (
 id integer primary key autoincrement not null,
 spatial_type varchar(18)
);

SELECT AddGeometryColumn(
 NULL,
 'sample_geometries',
 'geometry',
 4326,
 'geometry',
 'xy',
```

```
 'null'
);

INSERT INTO sample_geometries (spatial_type, geometry) VALUES (
 'ST_LineString',
 st_geometry ('linestring (20 20, 30 30, 20 40, 30 50)', 4326)
);

INSERT INTO sample_geometries (spatial_type, geometry) VALUES (
 'ST_Polygon',
 st_geometry ('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))', 4326)
);

INSERT INTO sample_geometries (spatial_type, geometry) VALUES (
 'ST_MultiPoint',
 st_geometry ('multipoint ((20 20), (30 30), (20 40), (30 50))', 4326)
);
```

```
--Find the convex hull of each geometry subtype.
SELECT id, spatial_type, st_astext (st_convexhull (geometry))
 AS CONVEXHULL
 FROM sample_geometries;

id    spatial_type       CONVEXHULL

 1    ST_LineString      POLYGON ((20.00000000 40.00000000,
                           20.00000000 20.00000000, 30.00000000
                           30.00000000, 30.00000000 50.00000000,
                           20.00000000 40.00000000))

 2    ST_Polygon         POLYGON ((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))

 3    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

## Definition

ST_CoordDim returns the dimensions of coordinate values for a geometry column.

## Syntax

### Oracle and PostgreSQL

```
sde.st_coorddim (geometry1 sde.st_geometry)
```

### SQLite

```
st_coorddim (geometry1 geometryblob)
```

## Return type

Integer

2 = x,y coordinates

3 = x,y,z or x,y,m coordinates

4 = x,y,z,m coordinates

## Example

In these examples, the coorddim_test table is created with the columns geotype and g1. The geotype column stores the name of the geometry subclass and dimension stored in the g1 geometry column.

The SELECT statement lists the subclass name stored in the geotype column with the dimension of the coordinates of that geometry.

### Oracle

```
--Create test table.
CREATE TABLE coorddim_test (
 geotype varchar(20),
 g1 sde.st_geometry
);
```

```
--Insert values to the test table.
INSERT INTO COORDDIM_TEST VALUES (
 'Point',
 sde.st_geometry ('point (60.567222 -140.404)', 4326)
);

INSERT INTO COORDDIM_TEST VALUES (
 'Point Z',
 sde.st_geometry ('point Z (60.567222 -140.404 5959)', 4326)
);
```

```
INSERT INTO COORDDIM_TEST VALUES (
 'Point M',
 sde.st_geometry ('point M (60.567222 -140.404 5250)', 4326)
);

INSERT INTO COORDDIM_TEST VALUES (
 'Point ZM',
 sde.st_geometry ('point ZM (60.567222 -140.404 5959 5250)', 4326)
);
```

```
--Determine the dimensionality of each feature.
SELECT geotype, sde.st_coorddim (g1) coordinate_dimension
 FROM COORDDIM_TEST;

GEOTYPE               coordinate_dimension

Point                           2
Point Z                         3
Point M                         3
Point ZM                        4
```

## PostgreSQL

```
--Create test table.
CREATE TABLE coorddim_test (
 geotype varchar(20),
 g1 sde.st_geometry
);
```

```
--Insert values to the test table.
INSERT INTO coorddim_test VALUES (
 'Point',
 st_point ('point (60.567222 -140.404)', 4326)
);

INSERT INTO coorddim_test VALUES (
 'Point Z',
 st_point ('point z (60.567222 -140.404 5959)', 4326)
);

INSERT INTO coorddim_test VALUES (
 'Point M',
 st_point ('point m (60.567222 -140.404 5250)', 4326)
);

INSERT INTO coorddim_test VALUES (
 'Point ZM',
 st_point ('point zm (60.567222 -140.404 5959 5250)', 4326)
);
```

```
--Determine the dimensionality of each feature.
SELECT geotype, st_coorddim (g1)
 AS coordinate_dimension
 FROM coorddim_test;
```

```
geotype              coordinate_dimension

Point                          2
Point Z                        3
Point M                        3
Point ZM                       4
```

## SQLite

```
--Create test tables and add geometry columns.
CREATE TABLE coorddim_test (
 geotype varchar(20)
);

SELECT AddGeometryColumn(
 NULL,
 'coorddim_test',
 'g1',
 4326,
 'pointzm',
 'xyzm',
 'null'
);

CREATE TABLE coorddim_test2 (
 geotype varchar(20)
);

SELECT AddGeometryColumn(
 NULL,
 'coorddim_test2',
 'g1',
 4326,
 'pointz',
 'xyz',
 'null'
);

CREATE TABLE coorddim_test3 (
 geotype varchar(20)
);

SELECT AddGeometryColumn(
 NULL,
 'coorddim_test3',
 'g1',
 4326,
 'pointm',
 'xym',
 'null'
);

CREATE TABLE coorddim_test4 (
 geotype varchar(20)
);

SELECT AddGeometryColumn(
 NULL,
 'coorddim_test4',
 'g1',
 4326,
```

```
 'point',
 'xy',
 'null'
);
```

```
--Insert values to the test table.
INSERT INTO coorddim_test4 VALUES (
 'Point',
 st_point ('point (60.567222 -140.404)', 4326)
);

INSERT INTO coorddim_test2 VALUES (
 'Point Z',
 st_point ('point z (60.567222 -140.404 5959)', 4326)
);

INSERT INTO coorddim_test3 VALUES (
 'Point M',
 st_point ('point m (60.567222 -140.404 5250)', 4326)
);

INSERT INTO coorddim_test VALUES (
 'Point ZM',
 st_point ('point zm (60.567222 -140.404 5959 5250)', 4326)
);
```

```
--Determine the dimensionality of features in each table.
SELECT geotype, st_coorddim (g1)
 AS coordinate_dimension
 FROM coorddim_test;

geotype              coordinate_dimension

Point ZM                       4


SELECT geotype, st_coorddim (g1)
 AS coordinate_dimension
 FROM coorddim_test2;

geotype              coordinate_dimension

Point Z                        3


SELECT geotype, st_coorddim (g1)
 AS coordinate_dimension
 FROM coorddim_test3;

geotype              coordinate_dimension

Point M                        3


SELECT geotype, st_coorddim (g1)
 AS coordinate_dimension
 FROM coorddim_test4;

geotype              coordinate_dimension
```

```
Point                          2
```

# ST_Crosses

## Definition

ST_Crosses takes two ST_Geometry objects and returns 1 (Oracle and SQLite) or t (PostgreSQL) if their intersection results in a geometry object whose dimension is one less than the maximum dimension of the source objects. The intersection object must contain points that are interior to both source geometries and are not equal to either of the source objects. Otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

```
sde.st_crosses (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### Oracle and PostgreSQL

```
sde.st_crosses (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_crosses (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

Boolean

## Example

The county government is considering a new regulation stating that all hazardous waste storage facilities within the county may not be in a specific radius of any waterway. The county GIS manager has an accurate representation of rivers and streams stored as linestrings in the waterways table, but he only has a single point location for each of the hazardous waste storage facilities.

To determine whether he must alert the county supervisor to any existing facilities that would violate the proposed regulation, the GIS manager must buffer the hazardous_sites locations to see if any rivers or streams cross the buffer polygons. The cross predicate compares the buffered hazardous_sites points with waterways, returning only those records where the waterway crosses over the county's proposed regulated radius.

### Oracle

```
--Define tables and insert values.
CREATE TABLE waterways (
 id integer,
 name varchar(128),
 water sde.st_geometry
);

CREATE TABLE hazardous_sites (
 site_id integer,
 name varchar(40),
 location sde.st_geometry
);
```

```
INSERT INTO waterways VALUES (
 2,
 'Zanja',
 sde.st_geometry ('linestring (40 50, 50 40)', 4326)
);

INSERT INTO waterways VALUES (
 3,
 'Keshequa',
 sde.st_geometry ('linestring (20 20, 60 60)', 4326)
);

INSERT INTO hazardous_sites VALUES (
 4,
 'StorIt',
 sde.st_point ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites VALUES (
 5,
 'Glowing Pools',
 sde.st_point ('point (30 30)', 4326)
);
```

```
--Buffer hazardous waste sites and find if any buffers cross a waterway.
SELECT UNIQUE (ww.name) "River or stream", hs.name "Hazardous sites"
 FROM WATERWAYS ww, HAZARDOUS_SITES hs
 WHERE sde.st_crosses (sde.st_buffer (hs.location, .01), ww.water) = 1;

River or stream                                Hazardous sites

Keshequa                                                    StorIt
Keshequa           Glowing Pools
```

## PostgreSQL

```
--Define tables and insert values.
CREATE TABLE waterways (
 id serial,
 name varchar(128),
 water sde.st_geometry
);

CREATE TABLE hazardous_sites (
 site_id integer,
 name varchar(40),
 location sde.st_geometry
);

INSERT INTO waterways (name, water) VALUES (
 'Zanja',
 sde.st_geometry ('linestring (40 50, 50 40)', 4326)
);

INSERT INTO waterways (name, water) VALUES (
 'Keshequa',
 sde.st_geometry ('linestring (20 20, 60 60)', 4326)
);
```

```
INSERT INTO hazardous_sites (name, location) VALUES (
 'StorIt',
 sde.st_point ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'Glowing Pools',
 sde.st_point ('point (30 30)', 4326)
);
```

```
--Buffer hazardous waste sites and find if any buffers cross a waterway.
SELECT DISTINCT (ww.name) AS "River or stream", hs.name AS "Hazardous sites"
 FROM waterways ww, hazardous_sites hs
 WHERE sde.st_crosses (sde.st_buffer (hs.location, .01), ww.water) = 't';

River or stream                          Hazardous sites

Keshequa                                                 StorIt
Keshequa             Glowing Pools
```

## SQLite

```
--Define tables and insert values.
CREATE TABLE waterways (
 id integer primary key autoincrement not null,
 name varchar(128)
);

SELECT AddGeometryColumn(
 NULL,
 'waterways',
 'water',
 4326,
 'linestring',
 'xy',
 'null'
);

CREATE TABLE hazardous_sites (
 site_id integer primary key autoincrement not null,
 name varchar(40)
);

SELECT AddGeometryColumn(
 NULL,
 'hazardous_sites',
 'location',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO waterways (name, water) VALUES (
 'Zanja',
 st_geometry ('linestring (40 50, 50 40)', 4326)
);
```

```
INSERT INTO waterways (name, water) VALUES (
 'Keshequa',
 st_geometry ('linestring (20 20, 60 60)', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'StorIt',
 st_point ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites (name, location) VALUES (
 'Glowing Pools',
 st_point ('point (30 30)', 4326)
);
```

```
--Buffer hazardous waste sites and find if any buffers cross a waterway.
SELECT DISTINCT (ww.name) AS "River or stream", hs.name AS "Hazardous sites"
 FROM waterways ww, hazardous_sites hs
 WHERE st_crosses (st_buffer (hs.location, .01), ww.water) = 1;

River or stream                          Hazardous sites

Keshequa                                                StorIt
Keshequa             Glowing Pools
```

# ST_Curve

**Note:**
Oracle and SQLite only

## Definition

ST_Curve constructs a curve feature from a well-known text representation.

## Syntax

### Oracle

```
sde.st_curve (wkt clob, srid integer)
```

### SQLite

```
st_curve (wkt text, srid int32)
```

## Return type

ST_LineString

## Example

This example creates a table with a curve geometry, inserts values into it, and selects one feature from the table.

### Oracle

```
CREATE TABLE curve_test (
 id integer,
 geometry sde.st_curve
);

INSERT INTO CURVE_TEST VALUES (
 1910,
 sde.st_curve ('linestring (33 2, 34 3, 35 6)', 4326)
);

SELECT id, sde.st_astext (geometry) CURVE
 FROM CURVE_TEST;

ID     CURVE

1110   LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000,
               35.00000000 6.00000000)
```

### SQLite

```
CREATE TABLE curve_test (
 id integer primary key autoincrement not null
```

```
);

SELECT AddGeometryColumn(
 NULL,
 'curve_test',
 'geometry',
 4326,
 'linestring',
 'xy',
 'null'
);

INSERT INTO CURVE_TEST (geometry) VALUES (
 st_curve ('linestring (33 2, 34 3, 35 6)', 4326)
);

SELECT id, st_astext (geometry)
 AS curve
 FROM curve_test;

id     curve

1   LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000,
       35.00000000 6.00000000)
```

# ST_Difference

## Definition

ST_Difference takes two geometry objects and returns a geometry object that is the difference of the source objects.

## Syntax

### Oracle and PostgreSQL

```
sde.st_difference (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_difference (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

In the following examples, the city engineer needs to know the total area of the city's lot area not covered by buildings; therefore, she wants the sum of the lot area after the building area has been removed.

The city engineer equally joins the footprints and lots table on the lot_id and takes the sum of the area of the difference of the lots minus the footprints.

### Oracle

```
--Create tables and insert values
CREATE TABLE footprints (
 building_id integer,
 footprint sde.st_geometry
);

CREATE TABLE lots (
 lot_id integer,
 lot sde.st_geometry
);

INSERT INTO footprints (building_id, footprint) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
```

```
INSERT INTO footprints (building_id, footprint) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO footprints (building_id, footprint) VALUES (
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO lots (lot_id, lot) VALUES (
 1,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO lots (lot_id, lot) VALUES (
 2,
 sde.st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO lots (lot_id, lot) VALUES (
 3,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT SUM (sde.st_area (sde.st_difference (lot, footprint)))
  FROM FOOTPRINTS bf, LOTS
  WHERE bf.building_id = lots.lot_id;

SUM(ST_AREA(ST_DIFFERENCE(LOT,FOOTPRINT)))

114
```

## PostgreSQL

```
--Create tables and insert values
CREATE TABLE footprints (
 building_id integer,
 footprint sde.st_geometry
);

CREATE TABLE lots (
 lot_id integer,
 lot sde.st_geometry
);

INSERT INTO footprints (building_id, footprint) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO footprints (building_id, footprint) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO footprints (building_id, footprint) VALUES (
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
```

```
);

INSERT INTO lots (lot_id, lot) VALUES (
 1,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO lots (lot_id, lot) VALUES (
 2,
 sde.st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO lots (lot_id, lot) VALUES (
 3,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT SUM (sde.st_area (sde.st_difference (lot, footprint)))
 FROM footprints bf, lots
 WHERE bf.building_id = lots.lot_id;

sum

114
```

## SQLite

```
--Create tables, add geometry columns, and insert values
CREATE TABLE footprints (
 building_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'footprints',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE lots (
 lot_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'lots',
 'lot',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO footprints (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
```

```
INSERT INTO footprints (footprint) VALUES (
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO footprints (footprint) VALUES (
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO lots (lot) VALUES (
 st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT SUM (st_area (st_difference (lot, footprint)))
 FROM footprints bf, lots
 WHERE bf.building_id = lots.lot_id;

sum

114.0
```

# ST_Dimension

## Definition

ST_Dimension returns the dimension of a geometry object. In this case, dimension refers to length and width. For example, a point has neither length nor width, so its dimension is 0; whereas a line has length but no width, so its dimension is 1.

## Syntax

### Oracle and PostgreSQL

```
sde.st_dimension (geometry1 sde.st_geometry)
```

### SQLite

```
st_dimension (geometry1 geometryblob)
```

## Return type

Integer

## Example

The dimension_test table is created with the columns geotype and g1. The geotype column stores the name of the subclass stored in the g1 geometry column.

The SELECT statement lists the subclass name stored in the geotype column with the dimension of that geotype.

### Oracle

```
CREATE TABLE dimension_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO DIMENSION_TEST VALUES (
 'Point',
 sde.st_pointfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO DIMENSION_TEST VALUES (
 'Linestring',
 sde.st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO DIMENSION_TEST VALUES (
 'Polygon',
 sde.st_polyfromtext ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01))', 4326)
);

INSERT INTO DIMENSION_TEST VALUES (
 'Multipoint',
 sde.st_mpointfromtext ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))',
```

```
4326)
);

INSERT INTO DIMENSION_TEST VALUES (
 'Multilinestring',
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO DIMENSION_TEST VALUES (
 'Multipolygon',
 sde.st_mpolyfromtext ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90, 51.71 21.73)))', 4326)
);
```

```
SELECT geotype, sde.st_dimension (g1) Dimension
 FROM DIMENSION_TEST;

GEOTYPE                    Dimension

Point                       0
Linestring                  1
Polygon                     2
Multipoint                  0
Multilinestring             1
Multipolygon                2
```

## PostgreSQL

```
CREATE TABLE dimension_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO dimension_test VALUES (
 'Point',
 sde.st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO dimension_test VALUES (
 'Linestring',
 sde.st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO dimension_test VALUES (
 'Polygon',
 sde.st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
);

INSERT INTO dimension_test VALUES (
 'Multipoint',
 sde.st_multipoint ('multipoint (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO dimension_test VALUES (
 'Multilinestring',
 sde.st_multilinestring ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64),
```

```
(9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO dimension_test VALUES (
 'Multipolygon',
 sde.st_multipolygon ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90, 51.71 21.73)))', 4326)
);
```

```
SELECT geotype, sde.st_dimension (g1)
 AS Dimension
 FROM dimension_test;

geotype                  dimension

Point                       0
Linestring                  1
Polygon                     2
Multipoint                  0
Multilinestring             1
Multipolygon                2
```

## SQLite

```
CREATE TABLE dimension_test (
 geotype varchar(20)
);

SELECT AddGeometryColumn (
 NULL,
 'dimension_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO dimension_test VALUES (
 'Point',
 st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO dimension_test VALUES (
 'Linestring',
 st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO dimension_test VALUES (
 'Polygon',
 st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
);

INSERT INTO dimension_test VALUES (
 'Multipoint',
 st_multipoint ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);
```

```
INSERT INTO dimension_test VALUES (
 'Multilinestring',
 st_multilinestring ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO dimension_test VALUES (
 'Multipolygon',
 st_multipolygon ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90, 51.71 21.73)))', 4326)
);
```

```
SELECT geotype, st_dimension (g1)
 AS "Dimension"
 FROM dimension_test;

geotype                  Dimension

Point                       0
Linestring                  1
Polygon                     2
Multipoint                  0
Multilines                  1
Multipolyg                  2
```

# ST_Disjoint

## Definition

ST_Disjoint takes two geometries and returns 1 (Oracle and SQLite) or t (PostgreSQL) if the intersection of two geometries produces an empty set; otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_disjoint (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_disjoint (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

Boolean

## Example

In this example, two tables are created (distribution_areas and factories), and values are inserted to each. Next, a buffer is created around the factories and st_disjoint is used to discover which factory buffers do not cross distribution areas.

**Tip:**

You could use the ST_Intersects function instead in this query by equating the result of the function to 0, because ST_Intersects and ST_Disjoint return opposite results. The ST_Intersects function uses the spatial index when evaluating the query, whereas the ST_Disjoint function does not.

### Oracle

```
--Create tables and insert values.
CREATE TABLE distribution_areas (
 id integer,
 areas sde.st_geometry
);

CREATE TABLE factories (
 id integer,
 loc sde.st_geometry
);

INSERT INTO distribution_areas (id, areas) VALUES (
 1,
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO distribution_areas (id, areas) VALUES (
 2,
```

```
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO distribution_areas (id, areas) VALUES (
 3,
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO factories (id,loc) VALUES (
 4,
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO factories (id,loc) VALUES (
 5,
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
--Buffer factories and find which buffers are separate from distribution areas.
SELECT da.id
 FROM DISTRIBUTION_AREAS da, FACTORIES f
 WHERE sde.st_disjoint ((sde.st_buffer (f.loc, .001)), da.areas) = 1;
```

## PostgreSQL

```
--Create tables and insert values.
CREATE TABLE distribution_areas (
 id serial,
 areas sde.st_geometry
);

CREATE TABLE factories (
 id serial,
 loc sde.st_geometry
);

INSERT INTO distribution_areas (areas) VALUES (
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO distribution_areas (areas) VALUES (
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO distribution_areas (areas) VALUES (
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO factories (loc) VALUES (
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO factories (loc) VALUES (
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
--Buffer factories and find which buffers are separate from distribution areas.
SELECT da.id
 FROM distribution_areas da, factories f
 WHERE sde.st_disjoint ((sde.st_buffer (f.loc, .001)), da.areas) = 't';
```

## SQLite

```
--Create tables and insert values.
CREATE TABLE distribution_areas (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'distribution_areas',
 'areas',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE factories (
 id integer primary key autoincrement not null
);
```

```
SELECT AddGeometryColumn (
 NULL,
 'factories',
 'loc',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO distribution_areas (areas) VALUES (
 st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO distribution_areas (areas) VALUES (
 st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO distribution_areas (areas) VALUES (
 st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO factories (loc) VALUES (
 st_geometry ('point (60 60)', 4326)
);

INSERT INTO factories (loc) VALUES (
 st_geometry ('point (30 30)', 4326)
);
```

```
--Buffer factories and find which buffers are separate from distribution areas.
SELECT da.id
 FROM distribution_areas da, factories f
 WHERE st_disjoint ((st_buffer (f.loc, .001)), da.areas) = 1;

id

1
2
3
```

# ST_Distance

## Definition

ST_Distance returns the distance between two geometries. The distance is measured from the closest vertices of the two geometries.

## Syntax

### Oracle and PostgreSQL

```
sde.st_distance (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

```
sde.st_distance (geometry1 sde.st_geometry, geometry2 sde.st_geometry, unit_name text)
```

### SQLite

```
st_distance (geometry1 geometryblob, geometry2 geometryblob)
```

```
st_distance (geometry1 geometryblob, geometry2 geometryblob, unit_name text)
```

Valid unit names are as follows:

| | | | |
|---|---|---|---|
| Millimeter | Inch | Yard | Link |
| Centimeter | Inch_US | Yard_US | Link_US |
| Decimeter | Foot | Yard_Clarke | Link_Clarke |
| Meter | Foot_US | Yard_Sears | Link_Sears |
| Meter_German | Foot_Clarke | Yard_Sears_1922_Truncated | Link_Sears_1922_Truncated |
| Kilometer | Foot_Sears | Yard_Benoit_1895_A | Link_Benoit_1895_B |
| 50_Kilometers | Foot_Sears_1922_Truncated | Yard_Indian | Chain |
| 150_Kilometers | Foot_Benoit_1895_A | Yard_Indian_1937 | Chain_US |
| Vara_US | Foot_1865 | Yard_Indian_1962 | Chain_Clarke |
| Smoot | Foot_Indian | Yard_Indian_1975 | Chain_Sears |
| | Foot_Indian_1937 | Fathom | Chain_Sears_1922_Truncated |
| | Foot_Indian_1962 | Mile_US | Chain_Benoit_1895_A |
| | Foot_Indian_1975 | Statute_Mile | Rod |
| | Foot_Gold_Coast | Nautical_Mile | Rod_US |
| | Foot_British_1936 | Nautical_Mile_US | |
| | | Nautical_Mile_UK | |

## Return type

Double precision

## Example

Two tables—study1 and zones—are created and populated. The ST_Distance function is then used to determine the distance between the boundary of each subarea and the polygons in the study1 area table that have a use code of 400. Since there are three zones on this shape, three records should be returned.

If you do not specify units, ST_Distance uses the units of the projection system of the data. In the first example, that is decimal degrees. In the last two examples, kilometer is specified; therefore, the distance is returned in kilometers.

## Oracle and PostgreSQL

```
--Create tables and insert values.
CREATE TABLE zones (
 sa_id integer,
 usecode integer,
 shape sde.st_geometry
);

CREATE TABLE study1 (
 code integer unique,
 shape sde.st_geometry
);

INSERT INTO zones (sa_id, usecode, shape) VALUES (
 1,
 400,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO zones (sa_id, usecode, shape) VALUES (
 2,
 400,
 sde.st_polygon ('polygon ((12 3, 12 6, 15 6, 15 3, 12 3))', 4326)
);

INSERT INTO zones (sa_id, usecode, shape) VALUES (
 3,
 400,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO zones (sa_id, usecode, shape) VALUES (
 4,
 402,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO study1 (code, shape) VALUES (
 400,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 19 11, 31 11, 31 -1, 19 -1, 11 -1, -1
-1))', 4326)
);

INSERT INTO study1 (code, shape) VALUES (
 402,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
--Oracle SELECT statement without units
SELECT UNIQUE s.code, z.sa_id, sde.st_distance(z.shape, sde.st_boundary(s.shape))
DISTANCE
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY DISTANCE;

CODE       SA_ID      DISTANCE
---------- ---------- ----------
400                         1                                                1
400                                3                                        1
400                                3                                        4
```

```
--PostgreSQL SELECT statement without units
SELECT DISTINCT s.code, z.sa_id, sde.st_distance(z.shape, sde.st_boundary(s.shape))
 AS Distance
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY Distance;

code             sa_id                distance
400            1                                  1
400            3                                  1
400                   2                                     4

--Oracle SELECT statement with values returned in kilometers
SELECT UNIQUE s.code, z.sa_id, sde.st_distance(z.shape, sde.st_boundary(s.shape),
'kilometer') DISTANCE
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY DISTANCE;

      CODE      SA_ID   DISTANCE
---------- ---------- ----------
       400          1 109.639196
       400          3 109.639196
       400          2 442.300258

--PostgreSQL SELECT statement with values returned in kilometers
SELECT DISTINCT s.code, z.sa_id, sde.st_distance(z.shape, sde.st_boundary(s.shape),
'kilometer')
 AS Distance
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY Distance;

code             sa_id                distance
400       1      109.63919620267
400       3      109.63919620267
400       2      442.300258454087
```

## SQLite

```
--Create tables, add geometry columns, and insert values.
CREATE TABLE zones (
 sa_id integer primary key autoincrement not null,
 usecode integer
);

SELECT AddGeometryColumn (
 NULL,
 'zones',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE study1 (
 code integer unique
);
```

```
SELECT AddGeometryColumn (
 NULL,
 'study1',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO zones (usecode, shape) VALUES (
 400,
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO zones (usecode, shape) VALUES (
 400,
 st_polygon ('polygon ((12 3, 12 6, 15 6, 15 3, 12 3))', 4326)
);

INSERT INTO zones (usecode, shape) VALUES (
 400,
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO zones (usecode, shape) VALUES (
 402,
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO study1 (code, shape) VALUES (
 400,
 st_polygon ('polygon ((-1 -1, -1 11, 11 11, 19 11, 31 11, 31 -1, 19 -1, 11 -1, -1
-1))', 4326)
);

INSERT INTO study1 (code, shape) VALUES (
 402,
 st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
--SQLite SELECT statement without units
SELECT DISTINCT s.code, z.sa_id, st_distance(z.shape, st_boundary(s.shape))
 AS "Distance(km)"
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY "Distance(km)";

code             sa_id                distance
400            1                                  1
400            3                                  1
400                   2                                     4

--SQLite SELECT statement with units
SELECT DISTINCT s.code, z.sa_id, st_distance(z.shape, st_boundary(s.shape),
"kilometer")
 AS "Distance(km)"
 FROM zones z, study1 s
 WHERE z.usecode = s.code AND s.code = 400
 ORDER BY "Distance(km)";
```

```
code                  sa_id      Distance(km)

400                       1                                        109.63919620267
400                               3
109.63919620267
400                               2
442.30025845408
```

# ST_DWithin

## Definition

ST_DWithin takes two geometries as input and returns true if the geometries are within the specified distance of one another; otherwise, false is returned. The spatial reference system of the geometries determines what unit of measure is applied to the specified distance. Therefore, the geometries you provide to ST_DWithin must both use the same coordinate projection and spatial reference ID (SRID).

## Syntax

### Oracle and PostgreSQL

```
sde.st_dwithin (st_geometry geometry1, st_geometry geometry2, double_precision distance);
```

### SQLite

```
st_dwithin (geometryblob geometry1, geometryblob geometry2, double_precision distance);
```

## Return type

Boolean

## Examples

In the following examples, two tables are created and features are inserted to them. Next, the ST_DWithin function is used in two different SELECT statements: one to determine if a point in the first table is within 100 meters of a polygon in the second table, and one to determine which features are within 300 meters of one another.

### Oracle

```
--Create table to store points.
CREATE TABLE dwithin_test_pt (id INT, geom sde.st_geometry);

--Create table to store polygons.
CREATE TABLE dwithin_test_poly (id INT, geom sde.st_geometry);

--Insert features into each table.

INSERT INTO dwithin_test_pt
 VALUES
 (
  1,
  sde.st_geometry('point (1 2)', 4326)
 )
;

INSERT INTO dwithin_test_pt
 VALUES
 (
  2,
  sde.st_geometry('point (10.02 20.01)', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
```

```
 (
  1,
  sde.st_geometry('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
 (
  2,
  sde.st_geometry('polygon ((101.02 200.01, 111.92 350.64, 250.02 340.15, 190.15
330.94, 101.02 200.01))', 4326)
 )
;
```

Next, use ST_DWithin to determine which features in each table are within 100 meters of each other and which are not. The ST_Distance function is included in this statement to show the actual distance between the features.

```
--Determine which features in the point table are within 100 meters of the features in
the polygon table.
SELECT pt.id, poly.id, sde.st_distance(pt.geom, poly.geom) distance_meters,
sde.st_dwithin(pt.geom, poly.geom, 100) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

The statement returns the following:

```
ID     ID   DISTANCE_METERS   DWITHIN
1      1         20.1425048         1
1      2          221.83769         0
2      1                  0         1
2      2         201.695315         0
```

In the following example, ST_DWithin is used to find which features are within 300 meters of one another:

```
--Determine which features in the point table are within 300 meters of the features in
the polygon table.
SELECT pt.id, poly.id, sde.st_distance(pt.geom, poly.geom) distance_meters,
sde.st_dwithin(pt.geom, poly.geom, 300) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

The second select statement returns the following when run on data in Oracle:

```
ID     ID   DISTANCE_METERS    DWITHIN
1      1         20.1425048          1
1      2          221.83769          1
2      1                  0          1
2      2         201.695315          1
```

### PostgreSQL

```
--Create table to store points.
CREATE TABLE dwithin_test_pt (id INT, geom sde.st_geometry);

--Create table to store polygons.
CREATE TABLE dwithin_test_poly (id INT, geom sde.st_geometry);

--Insert features into each table.

INSERT INTO dwithin_test_pt
 VALUES
 (
  1,
  sde.st_geometry('point (1 2)', 4326)
 )
;

INSERT INTO dwithin_test_pt
 VALUES
 (
  2,
  sde.st_geometry('point (10.02 20.01)', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
 (
  1,
  sde.st_geometry('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
 (
  2,
  sde.st_geometry('polygon ((101.02 200.01, 111.92 350.64, 250.02 340.15, 190.15
330.94, 101.02 200.01))', 4326)
 )
;
```

Next, use ST_DWithin to determine which features in each table are within 100 meters of each other and which are not. The ST_Distance function is included in this statement to show the actual distance between the features.

```
--Determine which features in the point table are within 100 meters of the features in
the polygon table.
SELECT pt.id, poly.id, sde.st_distance(pt.geom, poly.geom) distance_meters,
sde.st_dwithin(pt.geom, poly.geom, 100) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

The statement returns the following:

```
id     id     distance_meters        dwithin
1      1      20.1425048094819       t
1      2      221.837689538996       f
```

```
2      1      0                      t
2      2      201.69531476958        f
```

In the following example, ST_DWithin is used to find which features are within 300 meters of one another:

```
--Determine which features in the point table are within 300 meters of the features in
the polygon table.
SELECT pt.id, poly.id, sde.st_distance(pt.geom, poly.geom) distance_meters,
sde.st_dwithin(pt.geom, poly.geom, 300) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

This second select statement returns the following:

```
id     id     distance_meters        dwithin
1      1      20.1425048094819       t
1      2      221.837689538996       t
2      1      0                      t
2      2      201.69531476958        t
```

## SQLite

```
--Create table to store points.
CREATE TABLE dwithin_test_pt (
 id integer not null
);

SELECT AddGeometryColumn(
 NULL,
 'dwithin_test_pt',
 'geom',
 4326,
 'point',
 'xy',
 'null'
);

--Create table to store polygons.
CREATE TABLE dwithin_test_poly (
 id integer not null
);

SELECT AddGeometryColumn(
 NULL,
 'dwithin_test_poly',
 'geom',
 4326,
 'polygon',
 'xy',
 'null'
);

--Insert features into each table.

INSERT INTO dwithin_test_pt
 VALUES
 (
  1,
```

```
  st_geometry('point (1 2)', 4326)
 )
;

INSERT INTO dwithin_test_pt
 VALUES
 (
  2,
  st_geometry('point (10.02 20.01)', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
 (
  1,
  st_geometry('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01))', 4326)
 )
;

INSERT INTO dwithin_test_poly
 VALUES
 (
  2,
  st_geometry('polygon ((101.02 200.01, 111.92 350.64, 250.02 340.15, 190.15 330.94,
101.02 200.01))', 4326)
 )
;
```

Next, use ST_DWithin to determine which features in each table are within 100 meters of each other and which are not. The ST_Distance function is included in this statement to show the actual distance between the features.

```
--Determine which features in the point table are within 100 meters of the features in
the polygon table.
SELECT pt.id, poly.id, st_distance(pt.geom, poly.geom) distance_meters,
st_dwithin(pt.geom, poly.geom, 100) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

The statement returns the following:

```
1|1|20.1425048094819|1
1|2|221.837689538996|0
2|1|0.0|1
2|2|201.69531476958|0
```

In the following example, ST_DWithin is used to find which features are within 300 meters of one another:

```
--Determine which features in the point table are within 300 meters of the features in
the polygon table.
SELECT pt.id, poly.id, st_distance(pt.geom, poly.geom) distance_meters,
st_dwithin(pt.geom, poly.geom, 300) DWithin
 FROM dwithin_test_pt pt, dwithin_test_poly poly;
```

This second select statement returns the following:

```
1|1|20.1425048094819|1
1|2|221.837689538996|1
2|1|0.0|1
2|2|201.69531476958|1
```

# ST_EndPoint

## Definition

ST_EndPoint returns the last point of a linestring.

## Syntax

### Oracle and PostgreSQL

```
sde.st_endpoint (line1 sde.st_geometry)
```

### SQLite

```
st_endpoint (line1 geometryblob)
```

## Return type

ST_Point

## Example

The endpoint_test table stores the gid integer column, which uniquely identifies each row, and the ln1 ST_LineString column, which stores linestrings.

The INSERT statements insert linestrings into the endpoint_test table. The first linestring doesn't have z-coordinates or measures, while the second one does.

The query lists the gid column and the ST_Point geometry generated by the ST_EndPoint function.

### Oracle

```
--Create table and insert values.
CREATE TABLE endpoint_test (
 gid integer,
 ln1 sde.st_geometry
);

INSERT INTO ENDPOINT_TEST VALUES (
 1,
 sde.st_linefromtext ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO ENDPOINT_TEST VALUES (
 2,
 sde.st_linefromtext ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1,30.10
40.23 6.9 7.2)', 4326)
);
```

```
--Find the end point of each line.
SELECT gid, sde.st_astext (sde.st_endpoint (ln1)) Endpoint
 FROM ENDPOINT_TEST;
```

```
GID         Endpoint

 1    POINT (30.10 40.23)
 2    POINT ZM (30.10 40.23 6.9 7.2)
```

## PostgreSQL

```
--Create table and insert values.
CREATE TABLE endpoint_test (
 gid integer,
 ln1 sde.st_geometry
);

INSERT INTO endpoint_test VALUES (
 1,
st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO endpoint_test VALUES (
 2,
 st_linestring ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1,30.10 40.23 6.9
7.2)', 4326)
);
```

```
--Find the end point of each line.
SELECT gid, st_astext (st_endpoint (ln1))
 AS endpoint
 FROM endpoint_test;

gid          endpoint

 1    POINT (30.10 40.23)
 2    POINT ZM (30.10 40.23 6.9 7.2)
```

## SQLite

```
--Create table, add geometry column, and insert values.
CREATE TABLE endpoint_test (
 gid integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'endpoint_test',
 'ln1',
 4326,
 'linestringzm',
 'xyzm',
 'null'
);

INSERT INTO endpoint_test (ln1) VALUES (
 st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO endpoint_test (ln1) VALUES (
 st_linestring ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1,30.10 40.23 6.9
```

```
7.2)', 4326)
);
```

```
--Find the end point of each line.
SELECT gid, st_astext (st_endpoint (ln1))
 AS "endpoint"
 FROM endpoint_test;

gid          endpoint

 1    POINT (30.10000000 40.23000000)
 2    POINT ZM (30.10000000 40.23000000 6.90000000 7.20000000)
```

# ST_Entity

## Definition

ST_Entity returns the spatial entity type of a geometry object. The spatial entity type is the value stored in the entity member field of the geometry object.

## Syntax

### Oracle and PostgreSQL

```
sde.st_entity (geometry1 sde.st_geometry)
```

### SQLite

```
st_entity (geometry1 geometryblob)
```

## Return type

A number (Oracle) or integer (SQLite and PostgreSQL) is returned that represents the following entity types:

| | |
|---|---|
| 0 | nil shape |
| 1 | point |
| 2 | line (includes spaghetti lines) |
| 4 | linestring |
| 8 | area |
| 257 | multipoint |
| 258 | multiline (includes spaghetti lines) |
| 260 | multilinestring |
| 264 | multiarea |

## Example

The following examples create a table and insert different geometries to the table. ST_Entity is run on the table to return the geometry subtype of each record in the table.

### Oracle

```
CREATE TABLE sample_geos (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO sample_geos (id, geometry) VALUES (
 1901,
 sde.st_geometry ('point (1 2)', 4326)
);
```

```
INSERT INTO sample_geos (id, geometry) VALUES (
 1902,
 sde.st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO sample_geos (id, geometry) VALUES (
 1903,
 sde.st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);

SELECT sde.st_entity (geometry) entity, UPPER (sde.st_geometrytype (geometry)) TYPE
 FROM sample_geos;
```

The SELECT statement returns the following values:

```
 ENTITY        TYPE

    1    ST_POINT
    4    ST_LINESTRING
    8    ST_POLYGON
```

## PostgreSQL

```
CREATE TABLE sample_geos (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO sample_geos (id, geometry) VALUES (
 1900,
 sde.st_geometry ('Point Empty', 4326)
);

INSERT INTO sample_geos (id, geometry) VALUES (
 1901,
 sde.st_geometry ('point (1 2)', 4326)
);

INSERT INTO sample_geos (id, geometry) VALUES (
 1902,
 sde.st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO sample_geos (id, geometry) VALUES (
 1903,
 sde.st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);

INSERT INTO sde.entity_test (id, geometry) VALUES (
 1904,
 sde.st_geometry ('multipoint (10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74)',
4326)
);

INSERT INTO sde.entity_test (id, geometry) VALUES (
 1905,
 sde.st_geometry ('multilinestring ((10.01 20.03, 10.52 40.11, 30.29 41.56,31.78
10.74), (20.93 20.81, 21.52 40.10))', 4326)
);
```

```
INSERT INTO sde.entity_test (id, geometry) VALUES (
 1906,
 sde.st_geometry ('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)))', 4326)
);


SELECT id AS "id",
 sde.st_entity (geometry) AS "entity",
 sde.st_geometrytype (geometry) AS "geom_type"
 FROM sample_geos;
```

The SELECT statement returns the following values:

```
 id       entity     geom_type

    1900     0          "ST_GEOMETRY"
    1901     1          "ST_POINT"
    1902     4          "ST_LINESTRING"
    1903     8          "ST_POLYGON"
    1904     257        "ST_MULTIPOINT"
    1905     260        "ST_MULTILINESTRING"
    1906     264        "ST_MULTIPOLYGON"
```

## SQLite

```
CREATE TABLE sample_geos (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'sample_geos',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO sample_geos (geometry) VALUES (
 st_geometry ('point (1 2)', 4326)
);

INSERT INTO sample_geos (geometry) VALUES (
 st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO sample_geos (geometry) VALUES (
 st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);

SELECT st_entity (geometry) AS "entity",
 st_geometrytype (geometry) AS "type"
 FROM sample_geos;
```

The SELECT statement returns the following values:

```
entity    type

1         ST_POINT
4         ST_LINESTRING
8         ST_POLYGON
```

# ST_Envelope

## Definition

ST_Envelope returns the minimum bounding box of a geometry object as a polygon.

**Dive-in:**
This function conforms with the Open Geospatial Consortium (OGC) Simple Features specification that states that ST_Envelope return a polygon. To work with special cases of point geometries or horizontal or vertical lines, the ST_Envelope function returns a polygon around these shapes, which is a small envelope tolerance calculated based on the XY scale factor for the geometry's spatial reference system. This tolerance is subtracted from the min x and y and added to the max x and y coordinates to return the polygon around these shapes.

## Syntax

### Oracle and PostgreSQL

```
sde.st_envelope (geometry1 sde.st_geometry)
```

### SQLite

```
st_envelope (geometry1 geometryblob)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

The envelope_test table's geotype column stores the name of the geometry subclass stored in the g1 column. The INSERT statements insert each geometry subclass into the envelope_test table.

Next, the ST_Envelope function is run to return the polygon envelope around each geometry.

### Oracle

```
--Create table and insert values.
CREATE TABLE envelope_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO ENVELOPE_TEST VALUES (
'Point',
```

```
sde.st_geometry ('point (-1509734.232 -36684.757)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Linestring',
sde.st_geometry ('linestring (-1511144.181 -37680.015, -1509734.232 -38841.149,
-1508656.036 -39753.469)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Polygon',
sde.st_geometry ('polygon ((-1506333.768 -36435.943, -1504343.252 -36767.695,
-1502684.489 -35357.747, -1506333.768 -36435.943))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multipoint',
sde.st_geometry ('multipoint ((-1493229.539 -40665.789), (-1494141.859 -40831.665),
(-1495800.622 -42739.242))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multilinestring',
sde.st_geometry ('multilinestring ((-1504757.943 -33201.355, -1507411.964 -35606.561),
(-1502518.613 -38094.706, -1499781.653 -37099.448, -1498952.272 -34694.241))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multipolygon',
sde.st_geometry ('multipolygon (((-1492068.405 -47300.841, -1492814.848 -45725.016,
-1493975.983 -46471.459,
-1493478.354 -47798.47, -1492068.405 -47300.841), (-1497874.076 -48047.284,
-1498537.581 -50618.367, -1497210.571 -50037.8,
-1497874.076 -48047.284)))', 102004)
);
```

```
--Return the polygon envelope around each geometry in well-known text.
SELECT geotype geometry_type,
 sde.st_astext (sde.st_envelope (g1)) envelope
 FROM ENVELOPE_TEST;

GEOMETRY_TYPE    ENVELOPE

Point           |POLYGON (( -1509734.23220000 -36684.75720000, -1509734.23180000
-36684.75720000,
-1509734.23180000 -36684.75680000, -1509734.23220000 -36684.75680000, -1509734.23220000
-36684.75720000))

Linestring      |POLYGON (( -1511144.18100000 -39753.46900000, -1508656.03600000
-39753.46900000,
-1508656.03600000 -37680.01500000, -1511144.18100000 -37680.01500000, -1511144.18100000
-39753.46900000))

Polygon         |POLYGON (( -1506333.76800000 -36767.69500000, -1502684.48900000
-36767.69500000,
-1502684.48900000 -35357.74700000, -1506333.76800000 -35357.74700000, -1506333.76800000
-36767.69500000))

Multipoint      |POLYGON (( -1495800.62200000 -42739.24200000, -1493229.53900000
-42739.24200000,
-1493229.53900000 -40665.78900000, -1495800.62200000 -40665.78900000, -1495800.62200000
```

```
-42739.24200000))

Multilinestring |POLYGON (( -1507411.96400000 -38094.70600000, -1498952.27200000
-38094.70600000,
-1498952.27200000 -33201.35500000, -1507411.96400000 -33201.35500000, -1507411.96400000
-38094.70600000))

Multipolygon    |POLYGON (( -1498537.58100000 -50618.36700000, -1492068.40500000
-50618.36700000,
-1492068.40500000 -45725.01600000, -1498537.58100000 -45725.01600000, -1498537.58100000
-50618.36700000))
```

## PostgreSQL

```
--Create table and insert values.
CREATE TABLE envelope_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO ENVELOPE_TEST VALUES (
'Point',
sde.st_geometry ('point (-1509734.232 -36684.757)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Linestring',
sde.st_geometry ('linestring (-1511144.181 -37680.015, -1509734.232 -38841.149,
-1508656.036 -39753.469)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Polygon',
sde.st_geometry ('polygon ((-1506333.768 -36435.943, -1504343.252 -36767.695,
-1502684.489 -35357.747, -1506333.768 -36435.943))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multipoint',
sde.st_geometry ('multipoint (-1493229.539 -40665.789, -1494141.859 -40831.665,
-1495800.622 -42739.242)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multilinestring',
sde.st_geometry ('multilinestring ((-1504757.943 -33201.355, -1507411.964 -35606.561),
(-1502518.613 -38094.706, -1499781.653 -37099.448, -1498952.272 -34694.241))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
'Multipolygon',
sde.st_geometry ('multipolygon (((-1492068.405 -47300.841, -1492814.848 -45725.016,
-1493975.983 -46471.459,
-1493478.354 -47798.47, -1492068.405 -47300.841), (-1497874.076 -48047.284,
-1498537.581 -50618.367, -1497210.571 -50037.8,
-1497874.076 -48047.284)))', 102004)
);
```

```
--Return the polygon envelope around each geometry in well-known text.
SELECT geotype AS geometry_type,
 sde.st_astext (sde.st_envelope (g1)) AS Envelope
 FROM envelope_test;

geometry_type     envelope

"Point"           |"POLYGON (( -1509734.23220000 -36684.75720000, -1509734.23180000
-36684.75720000,
-1509734.23180000 -36684.75680000, -1509734.23220000 -36684.75680000, -1509734.23220000
-36684.75720000))"

"Linestring"      |"POLYGON (( -1511144.18100000 -39753.46900000, -1508656.03600000
-39753.46900000,
-1508656.03600000 -37680.01500000, -1511144.18100000 -37680.01500000, -1511144.18100000
```

```
-39753.46900000))"

"Polygon"         |"POLYGON (( -1506333.76800000 -36767.69500000, -1502684.48900000
-36767.69500000,
-1502684.48900000 -35357.74700000, -1506333.76800000 -35357.74700000, -1506333.76800000
-36767.69500000))"

"Multipoint"      |"POLYGON (( -1495800.62200000 -42739.24200000, -1493229.53900000
-42739.24200000,
-1493229.53900000 -40665.78900000, -1495800.62200000 -40665.78900000, -1495800.62200000
-42739.24200000))"

"Multilinestring" |"POLYGON (( -1507411.96400000 -38094.70600000, -1498952.27200000
-38094.70600000,
-1498952.27200000 -33201.35500000, -1507411.96400000 -33201.35500000, -1507411.96400000
-38094.70600000))"

"Multipolygon"    |"POLYGON (( -1498537.58100000 -50618.36700000, -1492068.40500000
-50618.36700000,
-1492068.40500000 -45725.01600000, -1498537.58100000 -45725.01600000, -1498537.58100000
-50618.36700000))"
```

## SQLite

```
--Create table and insert values.
CREATE TABLE envelope_test (
 geotype varchar(20)
);

SELECT AddGeometryColumn (
 NULL,
 'envelope_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Point',
 st_geometry ('point (-1509734.232 -36684.757)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Linestring',
 st_geometry ('linestring (-1511144.181 -37680.015, -1509734.232 -38841.149,
-1508656.036 -39753.469)', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Polygon',
 st_geometry ('polygon ((-1506333.768 -36435.943, -1504343.252 -36767.695, -1502684.489
-35357.747, -1506333.768 -36435.943))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Multipoint',
 st_geometry ('multipoint ((-1493229.539 -40665.789), (-1494141.859 -40831.665),
(-1495800.622 -42739.242))', 102004)
);
```

```
INSERT INTO ENVELOPE_TEST VALUES (
 'Multilinestring',
 st_geometry ('multilinestring ((-1504757.943 -33201.355, -1507411.964 -35606.561),
(-1502518.613 -38094.706, -1499781.653 -37099.448, -1498952.272 -34694.241))', 102004)
);

INSERT INTO ENVELOPE_TEST VALUES (
 'Multipolygon',
 st_geometry ('multipolygon (((-1492068.405 -47300.841, -1492814.848 -45725.016,
-1493975.983 -46471.459,
-1493478.354 -47798.47, -1492068.405 -47300.841), (-1497874.076 -48047.284,
-1498537.581 -50618.367, -1497210.571 -50037.8,
-1497874.076 -48047.284)))', 102004)
);
```

```
--Return the polygon envelope around each geometry in well-known text.
SELECT geotype AS geometry_type,
 st_astext (st_envelope (g1)) AS "Envelope"
 FROM envelope_test;

geometry_type   Envelope

Point           POLYGON (( -1509734.23220000 -36684.75720000, -1509734.23180000
-36684.75720000,
-1509734.23180000 -36684.75680000, -1509734.23220000 -36684.75680000, -1509734.23220000
-36684.75720000))

Linestring      POLYGON (( -1511144.18100000 -39753.46900000, -1508656.03600000
-39753.46900000,
-1508656.03600000 -37680.01500000, -1511144.18100000 -37680.01500000, -1511144.18100000
-39753.46900000))

Polygon         POLYGON (( -1506333.76800000 -36767.69500000, -1502684.48900000
-36767.69500000,
-1502684.48900000 -35357.74700000, -1506333.76800000 -35357.74700000, -1506333.76800000
-36767.69500000))

Multipoint      POLYGON (( -1495800.62200000 -42739.24200000, -1493229.53900000
-42739.24200000,
-1493229.53900000 -40665.78900000, -1495800.62200000 -40665.78900000, -1495800.62200000
-42739.24200000))

Multilinestring POLYGON (( -1507411.96400000 -38094.70600000, -1498952.27200000
-38094.70600000,
-1498952.27200000 -33201.35500000, -1507411.96400000 -33201.35500000, -1507411.96400000
-38094.70600000))

Multipolygon    POLYGON (( -1498537.58100000 -50618.36700000, -1492068.40500000
-50618.36700000,
-1492068.40500000 -45725.01600000, -1498537.58100000 -45725.01600000, -1498537.58100000
-50618.36700000))
```

# ST_EnvIntersects

**Note:**
Oracle and SQLite only

## Definition

ST_EnvIntersects returns 1 (true) if the envelopes of two geometries intersect; otherwise, it returns 0 (false).

## Syntax

### Oracle

```
sde.st_envintersects (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
sde.st_envintersects (geometry1 sde.st_geometry, minx number, miny number, maxx number,
maxy number)
```

### SQLite

```
st_envintersects (geometry1 geometryblob, geometry2 geoemtryblob)
st_envintersects (geometry1 geoemtryblob, minx float64, miny float64, maxx float64,
maxy float64)
```

## Return type

Boolean

## Example

This example searches for a geometry that has an envelope intersected by the defined polygon.

The first SELECT statement compares the envelopes of the two geometries and the geometries themselves to see whether the features or the envelopes intersect.

The second SELECT statement uses an envelope to detect which features, if any, fall inside the envelope you pass in with the WHERE clause of the SELECT statement.

## Oracle

```
--Define and populate the table.
CREATE TABLE sample_geoms (
 id integer,
 geometry sde.st_geometry);

INSERT INTO SAMPLE_GEOMS (id, geometry) VALUES (
 1,
 sde.st_geometry ('linestring (10 10, 50 50)', 4326)
);

INSERT INTO SAMPLE_GEOMS (id, geometry) VALUES (
 2,
 sde.st_geometry ('linestring (10 20, 50 60)', 4326)
);
```

```
--Find the intersection of the geometries and the geometries' envelopes.
SELECT a.id, b.id, sde.st_intersects (a.geometry, b.geometry) Intersects,
sde.st_envintersects (a.geometry, b.geometry) Envelope_Intersects
 FROM SAMPLE_GEOMS a, SAMPLE_GEOMS b
 WHERE a.id = 1 AND b.id=2;

ID      ID      INTERSECTS   ENVELOPE_INTERSECTS

1       2       0            1
```

```
--Find the geometries whose envelopes intersect the specified envelope.
SELECT id
 FROM SAMPLE_GEOMS
 WHERE sde.st_envintersects(geometry, 5, 5, 60, 65) = 1;

ID

1
2
```

## SQLite

```
--Define and populate the table.
CREATE TABLE sample_geoms (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'sample_geoms',
 'geometry',
 4326,
 'linestring',
 'xy',
 'null'
);

INSERT INTO SAMPLE_GEOMS (geometry) VALUES (
 st_geometry ('linestring (10 10, 50 50)', 4326)
);
```

```
INSERT INTO SAMPLE_GEOMS (geometry) VALUES (
 st_geometry ('linestring (10 20, 50 60)', 4326)
);
```

```
--Find the intersection of the geometries and the geometries' envelopes.
SELECT a.id AS aid, b.id AS bid, st_intersects (a.geometry, b.geometry) AS "Intersects",
 st_envintersects (a.geometry, b.geometry) AS "Envelope_Intersects"
 FROM SAMPLE_GEOMS a, SAMPLE_GEOMS b
 WHERE a.id = 1 AND b.id = 2;

aid     bid     Intersects   Envelope_Intersects

1       2       0            1
```

```
--Find the geometries whose envelopes intersect the specified envelope.
SELECT id
 FROM SAMPLE_GEOMS
 WHERE st_envintersects(geometry, 5, 5, 60, 65) = 1;

ID

1
2
```

# ST_Equals

## Definition

ST_Equals compares two geometries and returns 1 (Oracle and SQLite) or t (PostgreSQL) if the geometries are identical; otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_equals (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_equals (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

Boolean

## Example

The GIS technician suspects that some of the data in the studies table was duplicated. To alleviate his concern, he queries the table to determine whether any of the shape multipolygons are equal.

The studies table was created and populated with the following statements. The id column uniquely identifies the study areas, and the shape field stores the area's geometry.

Next, the studies table is spatially joined to itself by the equal predicate, which returns 1 (Oracle and SQLite) or t (PostgreSQL) whenever it finds two multipolygons that are equal. The s1.id<>s2.id condition eliminates the comparison of a geometry to itself.

### Oracle

```
CREATE TABLE studies (
 id integer unique,
 shape sde.st_geometry
);

INSERT INTO studies (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO studies (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO studies (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);
```

```
INSERT INTO studies (id, shape) VALUES (
 4,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
```

```
SELECT UNIQUE (s1.id), s2.id
 FROM STUDIES s1, STUDIES s2
 WHERE sde.st_equals (s1.shape, s2.shape) = 1
 AND s1.id <> s2.id;

ID     ID

 4       1
 1       4
```

## PostgreSQL

```
CREATE TABLE studies (
 id serial,
 shape st_geometry
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
```

```
SELECT DISTINCT (s1.id), s2.id
 FROM studies s1, studies s2
 WHERE st_equals (s1.shape, s2.shape) = 't'
 AND s1.id <> s2.id;

id    id

 1      4
 4      1
```

## SQLite

```
CREATE TABLE studies (
 id integer primary key autoincrement not null
);
```

```
SELECT AddGeometryColumn (
 NULL,
 'studies',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO studies (shape) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);
```

```
SELECT DISTINCT (s1.id), s2.id
 FROM studies s1, studies s2
 WHERE st_equals (s1.shape, s2.shape) = 1
 AND s1.id <> s2.id;

id    id

 1      4
 4      1
```

# ST_Equalsrs

**Note:**
PostgreSQL only

## Definition

ST_Equalsrs checks to find whether two spatial reference systems of two different feature classes are identical. If the spatial reference systems are identical, t (true) is returned. If the spatial reference systems are not identical, ST_Equalsrs returns f (false).

## Syntax

```
sde.st_equalsrs (srid1 integer, srid2 integer)
```

## Return type

Boolean

## Example

In this example, the spatial reference IDs (SRID) of different feature classes are discovered, then ST_Equalsrs is used to see whether the SRIDs represent the same spatial reference system.

```
SELECT srid, table_name
FROM sde_layers
WHERE table_name = 'transmains' OR table_name = 'streets';
```

| srid | table_name |
|---|---|
| 2 | streets |
| 6 | transmains |

*Results of sde_layers query*

Now, use ST_Equalsrs to determine whether the spatial reference systems identified by these two SRIDs are the same.

```
SELECT sde.st_equalsrs(2,6) ;

 st_equalsrs

-------------

f

(1 row)
```

# ST_ExteriorRing

## Definition

ST_ExteriorRing returns the exterior ring of a polygon as a linestring.

## Syntax

```
sde.st_exteriorring (polygon1 sde.st_geometry)
```

### Oracle and PostgreSQL

```
sde.st_exteriorring (polygon1 sde.st_geometry)
```

### SQLite

```
st_exteriorring (polygon1 geometryblob)
```

## Return type

ST_LineString

## Example

An ornithologist, wanting to study the bird population on several islands, knows that the feeding zone of the bird species she is interested in is restricted to the shoreline. As part of her calculation of the islands' carrying capacity, the ornithologist requires the islands' perimeters. Some of the islands are so large they have several lakes on them. However, the shoreline of the lakes is inhabited exclusively by another more aggressive bird species. Therefore, the ornithologist requires the perimeter of only the exterior ring of the islands.

The ID and name columns of the islands table identify each island, while the land polygon column stores the island's geometry.

The ST_ExteriorRing function extracts the exterior ring from each island polygon as a linestring. The length of the linestring is calculated by the ST_Length function. The linestring lengths are summarized by the SUM function.

The exterior rings of the islands represent the ecological interface each island shares with the sea.

### Oracle

```
--Create the table and insert two polygons.
CREATE TABLE islands (
 id integer,
 name varchar(32),
 land sde.st_geometry
);

INSERT INTO islands VALUES (
1,
'Bear',
sde.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))', 4326)
);

INSERT INTO islands VALUES (
 2,
 'Johnson',
sde.st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
--Extract the exterior ring from each island and find its length.
SELECT SUM (sde.st_length (sde.st_exteriorring (land)))
 FROM ISLANDS;

SUM(ST_LENGTH(ST_EXTERIORRING(LAND)))

                            264.72136
```

## PostgreSQL

```
--Create the table and insert two polygons.
CREATE TABLE islands (
 id serial,
 name varchar(32),
 land sde.st_geometry
);

INSERT INTO islands (name, land) VALUES (
 'Bear',
 sde.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))', 4326)
);

INSERT INTO islands (name, land) VALUES (
 'Johnson',
 sde.st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
--Extract the exterior ring from each island and find its length.
SELECT SUM (sde.st_length (sde.st_exteriorring (land)))
 FROM islands;

sum

264.721359549996
```

## SQLite

```
--Create the table and insert two polygons.
CREATE TABLE islands (
 id integer primary key autoincrement not null,
 name varchar(32)
);

SELECT AddGeometryColumn (
```

```
 NULL,
 'islands',
 'land',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO islands (name, land) VALUES (
 'Bear',
 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))', 4326)
);

INSERT INTO islands (name, land) VALUES (
 'Johnson',
 st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
--Extract the exterior ring from each island and find its length.
SELECT SUM (st_length (st_exteriorring (land)))
 FROM islands;

sum

264.721359549996
```

# ST_GeomCollection

**Note:**
Oracle and PostgreSQL only

## Definition

ST_GeomCollection constructs a geometry collection from a well-known text representation.

## Syntax

### Oracle

```
sde.st_multilinestring (wkt clob, srid integer)
sde.st_multipoint (wkt clob, srid integer)
sde.st_multipolygon (wkt clob, srid integer)
```

### PostgreSQL

```
sde.st_multilinestring (wkt, srid integer)
sde.st_multilinestring (esri_shape bytea, srid integer)
sde.st_multipoint (wkt, srid integer)
sde.st_multipoint (esri_shape bytea, srid integer)
sde.st_multipolygon (wkt, srid integer)
sde.st_multipolygon (esri_shape bytea, srid integer)
```

## Return type

ST_GeomCollection

## Example

### Oracle

Create a table, geomcoll_test and insert geometries to it.

```
CREATE TABLE geomcoll_test (id integer, geometry sde.st_geometry);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1901,
sde.st_multipoint ('multipoint ((1 2), (4 3), (5 6))', 0)
);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1902,
sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 0)
);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1903,
sde.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)))', 0)
);
```

Select the geometry collection from the geomcoll_test table.

```
SELECT id, sde.st_astext (geometry) Geomcollection
FROM GEOMCOLL_TEST;

        ID     GEOMCOLLECTION

      1901     MULTIPOINT ((1.00000000 2.00000000), (4.00000000 3.00000000),
                      (5.00000000 6.00000000))

      1902     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))

      1903     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)))
```

## PostgreSQL

Create a table, geomcoll_test and insert geometries to it.

```
CREATE TABLE geomcoll_test (id integer, geometry sde.st_geometry);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1901,
sde.st_multipoint ('multipoint (1 2, 4 3, 5 6)', 0)
);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1902,
sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 0)
);

INSERT INTO geomcoll_test (id, geometry) VALUES (
1903,
sde.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)))', 0)
);
```

Select the geometry collection from the geomcoll_test table.

```
SELECT id, sde.st_astext (geometry)
AS geomcollection
FROM geomcoll_test;

        id     geomcollection

1901     MULTIPOINT (1 2, 4 3, 5 6)

1902     MULTILINESTRING ((33 2, 34        3, 35 6),(28 4,
29 5, 31 8, 43 12),(39 3, 37 4, 36 7))
```

```
1903     MULTIPOLYGON (((13 33, 10 43, 1 40, 7 36,
13 33)),((8 24, 9 25, 1 28, 8 24)), 3 3, 5 3, 4 6, 3 3)))
```

# ST_GeomCollFromWKB

**Note:**
PostgreSQL only

## Definition

ST_GeomCollFromWKB constructs a geometry collection from a well-known binary representation.

## Syntax

```
sde.st_geomcollfromwkb (wkb bytea, srid integer)
```

## Return type

ST_GeomCollection

## Example

**Note:**
Hard returns are inserted for readability. Remove them if you copy the statements.

Create a table, gcoll_test.

```
CREATE TABLE gcoll_test (pkey integer, shape sde.st_geomcollection );
```

Insert values into the table.

```
INSERT INTO gcoll_test VALUES
(1,
st_geomcollfromwkb (sde.st_asbinary(sde.st_geomcollection
('multipoint(20 20, 30 30, 20 40, 30 50)', 0)), 0));

INSERT INTO gcoll_test VALUES
(2,
sde.st_geomcollfromwkb (sde.st_asbinary(sde.st_geomcollection
('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64),
(9.55 23.75,15.36 30.11),(10 10,20 20,30 30,40 40, 90 90))', 0)), 0));

INSERT INTO gcoll_test VALUES
(3,
sde.st_geomcollfromwkb (sde.st_asbinary(sde.st_geomcollection
('multipolygon(((0 0,4 0,4 4,0 4,0 0),(1 1,2 1,2 2,1 2,1 1)),
((-1 -1,-1 -2,-2 -2,-2 -1,-1 -1)))', 0)), 0));
```

Select the geometry from the gcoll_test table.

```
SELECT pkey, sde.st_astext(shape) from gcoll_test;

 pkey   st_astext
```

```
   1          MULTIPOINT ( 20 20, 30 30, 20 40, 30 50)

   3           MULTIPOLYGON ((( 0 0, 4 0, 4 4, 0 4, 0 0), (1 1, 1 2,
2 2, 2 1, 1 1)), ((-1 -1, -2 -1, -2 -2, -1 -2, -1 -1))
```

# ST_Geometry

## Definition

ST_Geometry constructs a geometry from a well-known text representation.

**Note:**
When creating spatial tables that will be used with ArcGIS, it is best to create the column as the geometry supertype (for example, ST_Geometry) rather than specifying an ST_Geometry subtype.

## Syntax

### Oracle

- For linestrings, polygons, and points

```
sde.st_geometry (wkt clob, srid integer)
```

- For optimized points (which do not launch an extproc agent and, therefore, process the query more rapidly)

```
sde.st_geometry (x, y, z, m, srid)
```

  Use optimized point construction when performing batch inserts of large numbers of point data.

- For parametric circles

```
sde.st_geometry (x, y, z, m, radius, number_of_points, srid)
```

- For parametric ellipses

```
sde.st_geometry (x, y, z, m, semi_major_axis, semi_minor_axis, angle,
number_of_points, srid)
```

- For parametric wedges

```
sde.st_geometry (x, y, z, m, startangle, endangle, outerradius, innerradius,
number_of_points, srid)
```

### PostgreSQL

- For linestrings, polygons, and points

```
sde.st_geometry (wkt, srid integer)
sde.st_geometry (esri_shape bytea, srid integer)
```

- For parametric circles

```
sde.st_geometry (x, y, z, m, radius, number_of_points, srid)
```

- For parametric ellipses

```
sde.st_geometry (x, y, z, m, semi_major_axis, semi_minor_axis, angle,
number_of_points, srid)
```

- For parametric wedges

```
sde.st_geometry (x, y, z, m, startangle, endangle, outerradius, innerradius,
number_of_points, srid)
```

### SQLite

- For linestrings, polygons, and points

```
st_geometry (text  WKT_string,int32 srid)
```

- For parametric circles

```
st_geometry (x, y, z, m, radius, number_of_points, srid)
```

- For parametric ellipses

```
st_geometry (x, y, z, m, semi_major_axis, semi_minor_axis, angle_of_rotation,
number_of_points, srid)
```

- For parametric wedges

```
st_geometry (x, y, z, m, start_angle, end_angle, outer_radius, inner_radius,
number_of_points, srid)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Examples

### Creating and querying point, linestring, and polygon features

These examples create a table (geoms) and insert point, linestring, and polygon values into it.

*Oracle*

```
CREATE TABLE geoms (
 id integer,
```

```
 geometry sde.st_geometry
);
```

```
INSERT INTO GEOMS (id, geometry) VALUES (
 1901,
 sde.st_geometry ('point (1 2)', 4326)
);

--To insert the same point using optimized point construction:
INSERT INTO GEOMS (id, geometry) VALUES (
 1901,
 sde.st_geometry (1,2,null,null,4326)
);

INSERT INTO GEOMS (id, geometry) VALUES (
 1902,
 sde.st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO GEOMS (id, geometry) VALUES (
 1903,
 sde.st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);
```

*PostgreSQL*

```
CREATE TABLE geoms (
 id serial,
 geometry sde.st_geometry
);
```

```
INSERT INTO geoms (geometry) VALUES (
 sde.st_geometry ('point (1 2)', 4326)
);

INSERT INTO geoms (geometry) VALUES (
 sde.st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO geoms (geometry) VALUES (
 sde.st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);
```

*SQLite*

```
CREATE TABLE geoms (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'geoms',
 'geometry',
 4326,
```

```
 'geometry',
 'xy',
 'null'
);
```

```
INSERT INTO geoms (geometry) VALUES (
 st_geometry ('point (1 2)', 4326)
);

INSERT INTO geoms (geometry) VALUES (
 st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO geoms (geometry) VALUES (
 st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);
```

## Creating and querying parametric circles

Create a table, radii, and insert circles into it.

*Oracle*

```
CREATE TABLE radii (
 id integer,
 geometry sde.st_geometry
);
```

```
INSERT INTO RADII (id, geometry) VALUES (
 1904,
 sde.st_geometry (10,10,NULL,NULL,25,50,4326)
);

INSERT INTO RADII (id, geometry) VALUES (
 1905,
 sde.st_geometry (5,15,NULL,NULL,10,20,4326)
);
```

*PostgreSQL*

```
CREATE TABLE radii (
 id serial,
 geometry sde.st_geometry
);
```

```
INSERT INTO radii (geometry) VALUES (
 sde.st_geometry (10,10,NULL,NULL,25,50,4326)
);

INSERT INTO radii (geometry) VALUES (
 sde.st_geometry (5,15,NULL,20,10,30,4326)
);
```

*SQLite*

```
CREATE TABLE radii (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'radii',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);
```

```
INSERT INTO radii (geometry) VALUES (
 st_geometry (10,10,NULL,NULL,25,50,4326)
);

INSERT INTO radii (geometry) VALUES (
 st_geometry (5,15,NULL,20,10,30,4326)
);
```

## Creating and querying parametric ellipses

Create a table, track, and insert ellipses into it.

*Oracle*

```
CREATE TABLE track (
 id integer,
 geometry sde.st_geometry
);
```

```
INSERT INTO TRACK (id, geometry) VALUES (
 1907,
 sde.st_geometry (0,0,NULL,NULL,10,5,0,50,4326)
);

INSERT INTO TRACK (id, geometry) VALUES (
 1908,
 sde.st_geometry (4,19,10,20,10,5,0,40,4326)
);
```

*PostgreSQL*

```
CREATE TABLE track (
 id serial,
 geometry sde.st_geometry
);
```

```
INSERT INTO track (geometry) VALUES (
 sde.st_geometry (0,0,NULL,NULL,10,5,0,50,4326)
);

INSERT INTO track (geometry) VALUES (
 sde.st_geometry (4,19,10,20,10,5,0,40,4326)
);
```

*SQLite*

```
CREATE TABLE track (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'track',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);
```

```
INSERT INTO track (geometry) VALUES (
 st_geometry (0,0,NULL,NULL,10,5,0,50,4326)
);

INSERT INTO track (geometry) VALUES (
 st_geometry (4,19,10,20,10,5,0,40,4326)
);
```

## Creating and querying parametric wedges

Create a table, pwedge, and insert a wedge into it.

*Oracle*

```
CREATE TABLE pwedge (
 id integer,
 label varchar2(8),
 shape sde.st_geometry
);
```

```
INSERT INTO PWEDGE (id, label, shape) VALUES (
 1,
 'Wedge1',
 sde.st_geometry (10,30,NULL,NULL,45,145,5,2,60,4326)
);
```

*PostgreSQL*

```
CREATE TABLE pwedge (
 id serial,
 label varchar(8),
 shape sde.st_geometry
);
```

```
INSERT INTO pwedge (label, shape) VALUES (
 'Wedge',
 sde.st_geometry(10,30,NULL,NULL,45,145,5,2,60,4326)
);
```

*SQLite*

```
CREATE TABLE pwedge (
 id integer primary key autoincrement not null,
 label varchar(8)
);

SELECT AddGeometryColumn (
 NULL,
 'pwedge',
 'shape',
 4326,
 'geometry',
 'xy',
 'null'
);
```

```
INSERT INTO pwedge (label, shape) VALUES (
 'Wedge',
 st_geometry(10,30,NULL,NULL,45,145,5,2,60,4326)
);
```

# ST_GeometryN

## Definition

ST_GeometryN takes a collection and an integer index and returns the nth ST_Geometry object in the collection.

## Syntax

### Oracle and PostgreSQL

```
sde.st_geometryn (mpt1 sde.st_multipoint, index integer)
sde.st_geometryn (mln1 sde.st_multilinestring, index integer)
sde.st_geometryn (mpl1 sde.st_multipolygon, index integer)
```

### SQLite

```
st_geometryn (mpt1 st_multipoint, index integer)
st_geometryn (mln1 st_multilinestring, index integer)
st_geometryn (mpl1 st_multipolygon, index integer)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

In this example, a multipolygon is created. Then ST_GeometryN is used to list the second element of the multipolygon.

### Oracle

```
CREATE TABLE districts (
 dist_id integer,
 shape sde.st_multipolygon
);

INSERT INTO DISTRICTS (dist_id, shape) VALUES (
 1,
 sde.st_multipolygon ('multipolygon (((-1 -1, -1 11, 11 11, 11 -1, -1 -1),
(19 -1, 19 11, 29 9, 31 -1, 19 -1), (39 -1, 39 11, 51 11, 51 -1, 39 -1)))', 4326)
);

SELECT sde.st_astext (sde.st_geometryn (shape, 2)) Second_Element
 FROM DISTRICTS;

Second_Element

POLYGON  ((-1.00000000 -1.00000000, 11.00000000 -1.00000000, 11.0000000 0 11.000
```

### PostgreSQL

```
CREATE TABLE districts (
 dist_id serial,
 shape sde.st_geometry
);

INSERT INTO districts (shape) VALUES (
 sde.st_multipolygon ('multipolygon (((-1 -1, -1 11, 11 11, 11 -1, -1 -1),
(19 -1, 19 11, 29 9, 31 -1, 19 -1), (39 -1, 39 11, 51 11, 51 -1, 39 -1)))', 4326)
);

SELECT sde.st_astext (sde.st_geometryn (shape, 2)) AS Second_Element
 FROM districts;

second_element

POLYGON  ((39 -1, 51 -1, 51 11, 39 11, 39 -1))
```

### SQLite

```
CREATE TABLE districts (
 dist_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'districts',
 'shape',
 4326,
 'multipolygon',
 'xy',
 'null'
);

INSERT INTO districts (shape) VALUES (
 st_multipolygon ('multipolygon (((-1 -1, -1 11, 11 11, 11 -1, -1 -1),
(19 -1, 19 11, 29 9, 31 -1, 19 -1), (39 -1, 39 11, 51 11, 51 -1, 39 -1)))', 4326)
);

SELECT st_astext (st_geometryn (shape, 2))
 AS "Second_Element"
 FROM districts;

Second_Element

POLYGON  ((39.00000000 -1.00000000, 51.00000000 -1.00000000, 51.00000000 11.00000000,
39.00000000 11.00000000, 39.00000000 -1.00000000))
```

# ST_GeometryType

## Definition

ST_GeometryType takes a geometry object and returns its geometry type (for example, point, line, polygon, multipoint) as a string.

## Syntax

### Oracle and PostgreSQL

```
sde.st_geometrytype (g1 sde.st_geometry)
```

### SQLite

```
st_geometrytype (g1 geometryblob)
```

## Return type

Varchar(32) (Oracle and PostgreSQL) or text (SQLite) containing one of the following:

- ST_Point
- ST_LineString
- ST_Polygon
- ST_MultiPoint
- ST_MultiLineString
- ST_MultiPolygon

## Example

The geometrytype_test table contains the g1 geometry column.

The INSERT statements insert each geometry subclass into the g1 column.

The SELECT query lists the geometry type of each subclass stored in the g1 geometry column.

### Oracle

```
CREATE TABLE geometrytype_test (g1 sde.st_geometry);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('point (10.02 20.01)', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
```

```
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64), (9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01)), ((51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90,51.71 21.73)))', 4326)
);
```

```
SELECT UPPER (sde.st_geometrytype (g1)) Geometry_type
FROM GEOMETRYTYPE_TEST;

Geometry_type

ST_POINT
ST_LINESTRING
ST_POLYGON
ST_MULTIPOINT
ST_MULTILINESTRING
ST_MULTIPOLYGON
```

## PostgreSQL

```
CREATE TABLE geometrytype_test (g1 sde.st_geometry);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('point (10.02 20.01)', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multipoint (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64), (9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO geometrytype_test VALUES (
 sde.st_geometry ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01)), ((51.71 21.73, 73.36 27.04, 71.52 32.87,
```

```
52.43 31.90,51.71 21.73)))', 4326)
);
```

```
SELECT (sde.st_geometrytype (g1))
 AS Geometry_type
 FROM geometrytype_test;

Geometry_type

ST_POINT
ST_LINESTRING
ST_POLYGON
ST_MULTIPOINT
ST_MULTILINESTRING
ST_MULTIPOLYGON
```

## SQLite

```
CREATE TABLE geometrytype_test (id integer primary key autoincrement not null);

SELECT AddGeometryColumn (
 NULL,
 'geometrytype_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('point (10.02 20.01)', 4326)
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 4326)
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64), (9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO geometrytype_test (g1) VALUES (
 st_geometry ('multipolygon (((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01)), ((51.71 21.73, 73.36 27.04, 71.52 32.87,
52.43 31.90,51.71 21.73)))', 4326)
);
```

```
SELECT (st_geometrytype (g1))
 AS "Geometry_type"
 FROM geometrytype_test;

Geometry_type

ST_POINT
ST_LINESTRING
ST_POLYGON
ST_MULTIPOINT
ST_MULTILINESTRING
ST_MULTIPOLYGON
```

# ST_GeomFromCollection

**Note:**
PostgreSQL only

## Definition

ST_GeomFromCollection returns a set of st_geometry rows. Each row comprises a geometry and an integer. The integer represents the geometry's position in the set.

Use the ST_GeomFromCollection function to access each individual geometry in a multipart geometry. When the input geometry is a collection or multipart geometry (for example, ST_MultiLineString, ST_MultiPoint, ST_MultiPolygon), ST_GeomFromCollection returns a record for each of the collection components, and the path expresses the position of the component in the collection.

If you use ST_GeomFromCollection on a simple geometry (for example, ST_Point, ST_LineString, ST_Polygon), a single record is returned with an empty path since there is only one geometry.

## Syntax

`sde.st_geomfromcollection (shape sde.st_geometry)`

To return only the geometry, use `(sde.st_geomfromcollection (shape)).st_geo`.

To return only the position of the geometry, use `(sde.st_geomfromcollection (shape)).path[1]`.

## Return type

ST_Geometry set

## Example

In this example, create a multiline feature class (ghanasharktracks) containing a single feature with a four-part shape.

```
--Create the feature class.
CREATE TABLE ghanasharktracks (objectid integer, shape sde.st_geometry);

--Insert a multiline with four parts using SRID 4326.

INSERT INTO ghanasharktracks VALUES
 (1,
  sde.st_geometry('MULTILINESTRING Z (( 1 1 0, 1 6 0),(1 3 0, 3 3 0),(3 1 0, 3 3 0), (4
1 0, 4 6 0))',
  4326
  )
 );
```

To confirm the field contains data, query the table. Use ST_AsText directly on the shape field to see the shape coordinates as text. Note that the text description of the multilinestring is returned.

```
--View inserted feature.
SELECT gst_orig.objectid, sde.st_astext(gst_orig.shape) shapetext
 FROM ghanasharktracks gst_orig;
```

```
shapetext
------------------------------
"MULTILINESTRING Z (( 1.00000000 1.00000000 0.00000000, 1.00000000 6.00000000
0.00000000),(1.00000000 3.00000000 0.00000000, 3.00000000 3.00000000
0.00000000),(3.00000000 1.00000000 0.00000000, 3.00000000 3.00000000 0.00000000),
(4.00000000 1.00000000 0.00000000, 4.00000000 6.00000000 0.00000000))"
```

To return each linestring geometry individually, use the ST_GeomFromCollection function. To see the geometry as text, this example uses the ST_AsText function with the ST_GeomFromCollection function.

```
--Return each linestring in the multilinestring

SELECT sde.st_astext((sde.st_geomfromcollection(gst.shape)).st_geo) shapetext,
((sde.st_geomfromcollection(gst.shape)).path[1]) path
 FROM ghanasharktracks gst;

shapetext
      path
------------------------------------------------------------------------------------------
------------------
"LINESTRING Z ( 1.00000000 1.00000000 0.00000000, 1.00000000 6.00000000
0.00000000)"          1
"LINESTRING Z ( 1.00000000 3.00000000 0.00000000, 3.00000000 3.00000000
0.00000000)"          2
"LINESTRING Z ( 3.00000000 1.00000000 0.00000000, 3.00000000 3.00000000
0.00000000)"          3
"LINESTRING Z ( 4.00000000 1.00000000 0.00000000, 4.00000000 6.00000000
0.00000000)"          4
```

# ST_GeomFromText

**Note:**
Used in Oracle and SQLite only; for PostgreSQL, use ST_Geometry.

## Definition

ST_GeomFromText takes a well-known text representation and a spatial reference ID and returns a geometry object.

## Syntax

### Oracle

```
sde.st_geomfromtext (wkt clob, srid integer)
```

```
sde.st_geomfromtext (wkt clob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### SQLite

```
st_geomfromtext (wkt text, srid int32)
```

```
st_geomfromtext (wkt text)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

### Oracle

ST_Geometry

### SQLite

Geometryblob

## Example

The geometry_test table contains the integer gid column, which uniquely identifies each row, and the g1 column, which stores the geometry.

The INSERT statements insert the data into the gid and g1 columns of the geometry_test table. The ST_GeomFromText function converts the text representation of each geometry into its corresponding instantiable subclass. The SELECT statement at the end is done to ensure data was inserted into the g1 column.

## Oracle

```
CREATE TABLE geometry_test (
 gid smallint unique,
 g1 sde.st_geometry
);
```

```
INSERT INTO GEOMETRY_TEST VALUES (
 1,
 sde.st_geomfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO GEOMETRY_TEST VALUES (
 2,
 sde.st_geomfromtext('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 4326)
);

INSERT INTO GEOMETRY_TEST VALUES (
 3,
 sde.st_geomfromtext('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);

INSERT INTO GEOMETRY_TEST VALUES (
 4,
 sde.st_geomfromtext('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO GEOMETRY_TEST VALUES (
 5,
 sde.st_geomfromtext ('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64), (9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO GEOMETRY_TEST VALUES (
 6,
 sde.st_geomfromtext ('multipolygon (((10.02 20.01, 11.92 35.64,
25.02 34.15, 19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04,
71.52 32.87, 52.43 31.90, 51.71 21.73)))', 4326)
);

SELECT sde.st_astext(g1)
 FROM GEOMETRY_TEST;

POINT ( 10.02000000 20.01000000)
LINESTRING ( 10.01000000 20.01000000, 10.01000000 30.01000000, 10.01000000 40.01000000)
POLYGON (( 10.02000000 20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000,
11.92000000 35.64000000, 10.02000000 20.01000000))
MULTIPOINT (( 10.02000000 20.01000000), (10.32000000 23.98000000), (11.92000000
25.64000000))
MULTILINESTRING (( 10.02000000 20.01000000, 10.32000000 23.98000000, 11.92000000
25.64000000),( 9.55000000 23.75000000, 15.36000000 30.11000000))
MULTIPOLYGON ((( 51.71000000 21.73000000, 73.36000000 27.04000000, 71.52000000
32.87000000, 52.43000000 31.90000000, 51.71000000 21.73000000)),(( 10.02000000
20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000, 11.92000000 35.64000000,
10.02000000 20.01000000)))
```

## SQLite

```
CREATE TABLE geometry_test (
 gid integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'geometry_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);
```

```
INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext('linestring (10.01 20.01, 10.01 30.01, 10.01 40.01)', 4326)
);

INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);

INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext('multipoint ((10.02 20.01), (10.32 23.98), (11.92 25.64))', 4326)
);

INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext ('multilinestring ((10.02 20.01, 10.32 23.98,
11.92 25.64), (9.55 23.75, 15.36 30.11))', 4326)
);

INSERT INTO GEOMETRY_TEST (g1) VALUES (
 st_geomfromtext ('multipolygon (((10.02 20.01, 11.92 35.64,
25.02 34.15, 19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04,
71.52 32.87, 52.43 31.90, 51.71 21.73)))', 4326)
);

SELECT st_astext(g1)
 FROM geometry_test;

POINT (10.02000000 20.01000000)
LINESTRING (10.01000000 20.01000000, 10.01000000 30.01000000, 10.01000000 40.01000000)
POLYGON ((10.02000000 20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000,
11.92000000 35.64000000, 10.02000000 20.01000000))
MULTIPOINT ((10.02000000 20.01000000), (10.32000000 23.98000000), (11.92000000
25.64000000))
MULTILINESTRING ((10.02000000 20.01000000, 10.32000000 23.98000000, 11.92000000
25.64000000),(9.55000000 23.75000000, 15.36000000 30.11000000))
MULTIPOLYGON (((51.71000000 21.73000000, 73.36000000 27.04000000, 71.52000000
32.87000000, 52.43000000 31.90000000, 51.71000000 21.73000000)),((10.02000000
20.01000000, 19.15000000 33.94000000, 25.02000000 34.15000000, 11.92000000 35.64000000,
10.02000000 20.01000000)))
```

# ST_GeomFromWKB

## Definition

ST_GeomFromWKB takes a well-known binary (WKB) representation and a spatial reference ID to return a geometry object.

## Syntax

### Oracle

```
sde.st_geomfromwkb (wkb blob, srid integer)
```

```
sde.st_geomfromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### PostgreSQL

```
sde.st_geomfromwkb (wkb, srid integer)
```

```
sde.st_geomfromwkb (esri_shape bytea, srid integer)
```

### SQLite

```
st_geomfromwkb (wkb blob, srid int32)
```

```
st_geomfromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

In the following example, the lines of results have been reformatted for readability. The spacing in your results will vary according to your online display. The following code illustrates how the ST_GeomFromWKB function can be

used to create and insert a line from a WKB line representation. The following example inserts a record into the sample_gs table with an ID and a geometry in spatial reference system 4326 in a WKB representation.

### Oracle

```
CREATE TABLE sample_gs (
 id integer,
 geometry sde.st_geometry,
 wkb blob
);

INSERT INTO sample_gs (id, geometry) VALUES (
 1901,
 sde.st_geomfromtext ('point (1 2)', 4326)
);

INSERT INTO sample_gs (id, geometry) VALUES (
 1902,
 sde.st_geomfromtext ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO sample_gs (id, geometry) VALUES (
 1903,
 sde.st_geomfromtext ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);

UPDATE sample_gs
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1901;

UPDATE sample_gs
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1902;

UPDATE sample_gs
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1903;

SELECT id, sde.st_astext (sde.st_geomfromwkb (wkb, 4326))
 FROM sample_gs;

ID   GEOMETRY

1901 POINT (1.00000000 2.00000000)
1902 LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000
6.00000000)
1903 POLYGON ((3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000,
3.00000000 3.00000000))
```

### PostgreSQL

```
CREATE TABLE sample_gs (
 id integer,
 geometry sde.st_geometry,
 wkb bytea);

INSERT INTO sample_gs (id, geometry) VALUES (
 1901,
 sde.st_geometry ('point (1 2)', 4326)
);
```

```
INSERT INTO sample_gs (id, geometry) VALUES (
 1902,
 sde.st_geometry ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO sample_gs (id, geometry) VALUES (
 1903,
 sde.st_geometry ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);

UPDATE sample_gs
 SET wkb = sde.st_asshape (geometry)
 WHERE id = 1901;

UPDATE sample_gs
 SET wkb = sde.st_asshape (geometry)
 WHERE id = 1902;

UPDATE sample_gs
 SET wkb = sde.st_asshape (geometry)
 WHERE id = 1903;

SELECT id, sde.st_astext (sde.st_geomfromshape (wkb, 4326))
 FROM sample_gs;

id   st_astext

1901 POINT (1 2)
1902 LINESTRING (33 2, 34 3, 35 6)
1903 POLYGON ((3 3, 5 3, 4 6, 3 3))
```

## SQLite

```
CREATE TABLE sample_gs (
 id integer primary key autoincrement not null,
 wkb blob
);

SELECT AddGeometryColumn (
 NULL,
 'sample_gs',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO sample_gs (geometry) VALUES (
 st_geomfromtext ('point (1 2)', 4326)
);

INSERT INTO sample_gs (geometry) VALUES (
 st_geomfromtext ('linestring (33 2, 34 3, 35 6)', 4326)
);

INSERT INTO sample_gs (geometry) VALUES (
 st_geomfromtext ('polygon ((3 3, 4 6, 5 3, 3 3))', 4326)
);
```

```
--Replace IDs with actual values.
UPDATE sample_gs
 SET wkb = st_asbinary (geometry)
 WHERE id = 1;

UPDATE sample_gs
 SET wkb = st_asbinary (geometry)
 WHERE id = 2;

UPDATE sample_gs
 SET wkb = st_asbinary (geometry)
 WHERE id = 3;

SELECT id, st_astext (st_geomfromwkb (wkb, 4326))
 FROM sample_gs;

ID   GEOMETRY

1    POINT (1.00000000 2.00000000)
2    LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000
6.00000000)
3    POLYGON ((3.00000000 3.00000000, 5.00000000 3.00000000, 4.00000000 6.00000000,
3.00000000 3.00000000))
```

# ST_GeoSize

**Note:**
PostgreSQL only

## Definition

ST_GeoSize takes an ST_Geometry object and returns its size in bytes.

## Syntax

```
st_geosize (st_geometry)
```

## Return type

Integer

## Example

You can discover the size of the features created in the ST_GeomFromWKB example by querying the geometry column of the sample_geometries table.

```
SELECT st_geosize (geometry)
FROM sample_geometries;

st_geosize
        512
        592
        616
```

# ST_InteriorRingN

## Definition

ST_InteriorRingN returns the nth interior ring of a polygon as an ST_LineString.

The order of the rings cannot be predefined since the rings are organized according to the rules defined by the internal geometry verification routines and not by geometric orientation. If the index exceeds the number of interior rings possessed by a polygon, a null value is returned.

## Syntax

### Oracle

```
sde.st_interiorringn (polygon1 sde.st_polygon, INDEX integer)
```

### PostgreSQL

```
sde.st_interiorringn (polygon1 sde.st_polygon, ring_number integer)
```

### SQLite

```
st_interiorringn (polygon1 sde.st_polygon, ring_number int32)
```

## Return type

ST_LineString

## Example

Create a table, sample_polys, and add a record, then select the ID and the geometry of the interior ring.

### Oracle

```
CREATE TABLE sample_polys (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO sample_polys VALUES (
 1,
 sde.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))', 4326)
);
```

```
SELECT id, sde.st_astext (sde.st_interiorringn (geometry, 2)) Interior_Ring
 FROM SAMPLE_POLYS;

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)
```

## PostgreSQL

```
CREATE TABLE sample_polys (
 id serial,
 geometry sde.st_geometry
);

INSERT INTO sample_polys (geometry) VALUES (
 sde.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))', 4326)
);
```

```
SELECT id, sde.st_astext (st_interiorringn (geometry, 2))
 AS Interior_Ring
 FROM sample_polys;

id interior_ring

1  LINESTRING (70 130, 70 140, 80 140, 80 130, 70 130)
```

## SQLite

```
CREATE TABLE sample_polys (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'sample_polys',
 'geometry',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO sample_polys (geometry) VALUES (
 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))', 4326)
);
```

```
SELECT id, st_astext (st_interiorringn (geometry, 2))
 AS "Interior_Ring"
 FROM sample_polys;

id Interior_Ring

1  LINESTRING (70.00000000 130.00000000, 70.00000000 140.00000000, 80.00000000
140.00000000, 80.00000000 130.00000000, 70.00000000 130.00000000)
```

# ST_Intersection

## Definition

ST_Intersection takes two geometry objects and returns the intersection set as a two-dimensional geometry object.

## Syntax

### Oracle and PostgreSQL

```
sde.st_intersection (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_intersection (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

The fire marshal must obtain the areas of the hospitals, schools, and nursing homes intersected by the radius of a possible hazardous waste contamination.

The hospitals, schools, and nursing homes are stored in the population table created with the CREATE TABLE statement that follows. The shape column, defined as a polygon, stores the outline of each of the sensitive areas.

The hazardous sites are stored in the waste_sites table created with the CREATE TABLE statement that follows. The site column, defined as a point, stores a location that is the geographic center of each hazardous site.

The ST_Buffer function generates a buffer surrounding the hazardous waste sites. The ST_Intersection function generates polygons from the intersection of the buffered hazardous waste sites and the sensitive areas.

### Oracle

```
CREATE TABLE population (
 id integer,
 shape sde.st_geometry
);

CREATE TABLE waste_sites (
 id integer,
 site sde.st_geometry
);
```

```
INSERT INTO population VALUES (
 1,
 sde.st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);

INSERT INTO population VALUES (
 2,
 sde.st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);

INSERT INTO population VALUES (
 3,
 sde.st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);

INSERT INTO waste_sites VALUES (
 40,
 sde.st_geometry ('point (.60 .60)', 4326)
);

INSERT INTO waste_sites VALUES (
 50,
 sde.st_geometry ('point (.30 .30)', 4326)
);
```

```
SELECT sa.id, sde.st_astext (sde.st_intersection (sde.st_buffer (hs.site, .1),
sa.shape)) Intersection
 FROM population sa, waste_sites hs
 WHERE hs.id = 50
 AND sde.st_astext (sde.st_intersection (sde.st_buffer (hs.site, .01), sa.shape))
 NOT LIKE '%EMPTY%';

  ID  INTERSECTION

1     POLYGON (( 0.29000000 0.30000000, 0.30000000 0.30000000, 0.30000000
0.31000000, 0.29934597 0.30997859, 0.29869474 0.30991445, 0.29804910 0.30980785,
 0.29741181 0.30965926, 0.29678561 0.30946930, 0.29617317 0.30923880, 0.29557711
 0.30896873, 0.29500000 0.30866025, 0.29444430 0.30831470, 0.29391239 0.30793353
, 0.29340654 0.30751840, 0.29292893 0.30707107, 0.29248160 0.30659346, 0.2920664
7 0.30608761, 0.29168530 0.30555570, 0.29133975 0.30500000, 0.29103127 0.3044228
9, 0.29076121 0.30382683, 0.29053070 0.30321440, 0.29034074 0.30258819, 0.290192
15 0.30195090, 0.29008555 0.30130526, 0.29002141 0.30065403, 0.29000000 0.300000
00))

2     POLYGON (( 0.30000000 0.30000000, 0.31000000 0.30000000, 0.30997859
0.30065403, 0.30991445 0.30130526, 0.30980785 0.30195090, 0.30965926 0.30258819,
 0.30946930 0.30321440, 0.30923880 0.30382683, 0.30896873 0.30442289, 0.30866025
 0.30500000, 0.30831470 0.30555570, 0.30793353 0.30608761, 0.30751840 0.30659346
, 0.30707107 0.30707107, 0.30659346 0.30751840, 0.30608761 0.30793353, 0.3055557
0 0.30831470, 0.30500000 0.30866025, 0.30442289 0.30896873, 0.30382683 0.3092388
0, 0.30321440 0.30946930, 0.30258819 0.30965926, 0.30195090 0.30980785, 0.301305
26 0.30991445, 0.30065403 0.30997859, 0.30000000 0.31000000, 0.30000000 0.300000
00))
```

## PostgreSQL

```
CREATE TABLE population (
 id serial,
 shape sde.st_geometry
```

```
);

CREATE TABLE waste_sites (
 id serial,
 site sde.st_geometry
);


INSERT INTO population (shape) VALUES (
 sde.st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);

INSERT INTO population (shape) VALUES (
 sde.st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);

INSERT INTO population (shape) VALUES (
 sde.st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);

INSERT INTO waste_sites (site) VALUES (
 sde.st_geometry ('point (.60 .60)', 4326)
);

INSERT INTO waste_sites (site) VALUES (
 sde.st_geometry ('point (.30 .30)', 4326)
);
```

```
--Replace hs.id with ID value of second record in waste_sites table if not 2.
SELECT sa.id, sde.st_astext (sde.st_intersection (sde.st_buffer (hs.site, .01),
sa.shape))
 AS Intersection
 FROM population sa, waste_sites hs
 WHERE hs.id = 2
 AND sde.st_astext (sde.st_intersection (sde.st_buffer (hs.site, .1),
sa.shape))::varchar
 NOT LIKE '%EMPTY%';

  id  intersection

1      POLYGON (( 0.29000000 0.30000000, 0.30000000 0.30000000, 0.30000000
0.31000000, 0.29934597 0.30997859, 0.29869474 0.30991445, 0.29804910 0.30980785,
 0.29741181 0.30965926, 0.29678561 0.30946930, 0.29617317 0.30923880, 0.29557711
 0.30896873, 0.29500000 0.30866025, 0.29444430 0.30831470, 0.29391239 0.30793353
, 0.29340654 0.30751840, 0.29292893 0.30707107, 0.29248160 0.30659346, 0.2920664
7 0.30608761, 0.29168530 0.30555570, 0.29133975 0.30500000, 0.29103127 0.3044228
9, 0.29076121 0.30382683, 0.29053070 0.30321440, 0.29034074 0.30258819, 0.290192
15 0.30195090, 0.29008555 0.30130526, 0.29002141 0.30065403, 0.29000000 0.300000
00))

2      POLYGON (( 0.30000000 0.30000000, 0.31000000 0.30000000, 0.30997859
0.30065403, 0.30991445 0.30130526, 0.30980785 0.30195090, 0.30965926 0.30258819,
 0.30946930 0.30321440, 0.30923880 0.30382683, 0.30896873 0.30442289, 0.30866025
 0.30500000, 0.30831470 0.30555570, 0.30793353 0.30608761, 0.30751840 0.30659346
, 0.30707107 0.30707107, 0.30659346 0.30751840, 0.30608761 0.30793353, 0.3055557
0 0.30831470, 0.30500000 0.30866025, 0.30442289 0.30896873, 0.30382683 0.3092388
0, 0.30321440 0.30946930, 0.30258819 0.30965926, 0.30195090 0.30980785, 0.301305
26 0.30991445, 0.30065403 0.30997859, 0.30000000 0.31000000, 0.30000000 0.300000
00))
```

## SQLite

```
CREATE TABLE population (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'population',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE waste_sites (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'waste_sites',
 'site',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO population (shape) VALUES (
 st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);

INSERT INTO population (shape) VALUES (
 st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);

INSERT INTO population (shape) VALUES (
 st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);

INSERT INTO waste_sites (site) VALUES (
 st_geometry ('point (.60 .60)', 4326)
);

INSERT INTO waste_sites (site) VALUES (
 st_geometry ('point (.30 .30)', 4326)
);
```

```
--Replace hs.id with ID value of second record in waste_sites table if not 2.
SELECT sa.id, st_astext (st_intersection (st_buffer (hs.site, .01), sa.shape))
 AS "Intersection"
 FROM population sa, waste_sites hs
 WHERE hs.id = 2
 AND st_astext (st_intersection (st_buffer (hs.site, .1), sa.shape))
 NOT LIKE '%EMPTY%';

  id  Intersection

1     POLYGON (( 0.29000000 0.30000000, 0.30000000 0.30000000, 0.30000000
```

```
0.31000000, 0.29934597 0.30997859, 0.29869474 0.30991445, 0.29804910 0.30980785,
 0.29741181 0.30965926, 0.29678561 0.30946930, 0.29617317 0.30923880, 0.29557711
 0.30896873, 0.29500000 0.30866025, 0.29444430 0.30831470, 0.29391239 0.30793353
, 0.29340654 0.30751840, 0.29292893 0.30707107, 0.29248160 0.30659346, 0.2920664
7 0.30608761, 0.29168530 0.30555570, 0.29133975 0.30500000, 0.29103127 0.3044228
9, 0.29076121 0.30382683, 0.29053070 0.30321440, 0.29034074 0.30258819, 0.290192
15 0.30195090, 0.29008555 0.30130526, 0.29002141 0.30065403, 0.29000000 0.300000
00))

2     POLYGON (( 0.30000000 0.30000000, 0.31000000 0.30000000, 0.30997859
0.30065403, 0.30991445 0.30130526, 0.30980785 0.30195090, 0.30965926 0.30258819,
 0.30946930 0.30321440, 0.30923880 0.30382683, 0.30896873 0.30442289, 0.30866025
 0.30500000, 0.30831470 0.30555570, 0.30793353 0.30608761, 0.30751840 0.30659346
, 0.30707107 0.30707107, 0.30659346 0.30751840, 0.30608761 0.30793353, 0.3055557
0 0.30831470, 0.30500000 0.30866025, 0.30442289 0.30896873, 0.30382683 0.3092388
0, 0.30321440 0.30946930, 0.30258819 0.30965926, 0.30195090 0.30980785, 0.301305
26 0.30991445, 0.30065403 0.30997859, 0.30000000 0.31000000, 0.30000000 0.300000
00))
```

# ST_Intersects

## Definition

ST_Intersects returns 1 (Oracle and SQLite) or t (PostgreSQL) if the intersection of two geometries doesn't result in an empty set; otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_intersects (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_intersects (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

Boolean

## Example

The fire marshal wants a list of sensitive areas within a radius of a hazardous waste site.

The sensitive areas are stored in the sensitive_areas table. The shape column, defined as a polygon, stores the outline of each of the sensitive areas.

The hazardous sites are stored in the hazardous_sites table. The site column, defined as a point, stores a location that is the geographic center of each hazardous site.

The SELECT query creates a buffer radius around each hazardous site and returns a list of sensitive areas that intersect the hazardous sites buffers.

### Oracle

```
--Create and populate tables.
CREATE TABLE sensitive_areas (
 id integer,
 shape sde.st_geometry
);

CREATE TABLE hazardous_sites (
 id integer,
 site sde.st_geometry
);

INSERT INTO sensitive_areas VALUES (
 1,
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas VALUES (
 2,
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
```

```
);

INSERT INTO sensitive_areas VALUES (
 3,
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO hazardous_sites VALUES (
 4,
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites VALUES (
 5,
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
--Create a buffer around the hazardous sites, then find the hazardous site buffers that
intersect sensitive areas.
SELECT sa.id SA_ID, hs.id HS_ID
 FROM SENSITIVE_AREAS sa, HAZARDOUS_SITES hs
 WHERE sde.st_intersects (sde.st_buffer (hs.site, .1), sa.shape) = 1
 ORDER BY sa.id;

SA_ID      HS_ID

    1          5
    2          5
    3          4
```

## PostgreSQL

```
--Create and populate tables.
CREATE TABLE sensitive_areas (
 id serial,
 shape sde.st_geometry
);

CREATE TABLE hazardous_sites (
 id serial,
 site sde.st_geometry
);

INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO hazardous_sites (site) VALUES (
 sde.st_geometry ('point (60 60)', 4326)
);
```

```
INSERT INTO hazardous_sites (site) VALUES (
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
--Create a buffer around the hazardous sites, then find the hazardous site buffers that
intersect sensitive areas.
SELECT sa.id AS sid, hs.id AS hid
 FROM sensitive_areas sa, hazardous_sites hs
 WHERE sde.st_intersects (sde.st_buffer (hs.site, .1), sa.shape) = 't'
 ORDER BY sa.id;

sid     hid

  1       2
  2       2
  3       1
```

## SQLite

```
--Create and populate tables.
CREATE TABLE sensitive_areas (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'sensitive_areas',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE hazardous_sites (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'hazardous_sites',
 'site',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
```

```
INSERT INTO hazardous_sites (site) VALUES (
 st_geometry ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites (site) VALUES (
 st_geometry ('point (30 30)', 4326)
);
```

```
--Create a buffer around the hazardous sites, then find the hazardous site buffers that
intersect sensitive areas.
SELECT sa.id AS "sid", hs.id AS "hid"
 FROM sensitive_areas sa, hazardous_sites hs
 WHERE st_intersects (st_buffer (hs.site, .1), sa.shape) = 1
 ORDER BY sa.id;

sid     hid

  1       2
  2       2
  3       1
```

# ST_Is3d

## Definition

ST_Is3d takes an ST_Geometry as an input parameter and returns 1 (Oracle and SQLite) or t (PostgreSQL) if the given geometry has z-coordinates; otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_is3d (geometry1 sde.st_geometry)
```

### SQLite

```
st_is3d (geometry1 geometryblob)
```

## Return type

Boolean

## Example

This example creates a table, is3d_test, and populates it with records.

Next, using ST_Is3d, check to see whether any of the records has a z-coordinate.

## Oracle

```
CREATE TABLE is3d_test (
 id integer,
 geo sde.st_geometry
);

INSERT INTO IS3D_TEST VALUES (
 1902,
 sde.st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO IS3D_TEST VALUES (
 1903,
 sde.st_geometry ('multipoint m((10 10 5), (50 10 6), (10 30 8))' , 4326)
);

INSERT INTO IS3D_TEST VALUES (
 1904,
 sde.st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);

INSERT INTO IS3D_TEST VALUES (
 1905,
 sde.st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, sde.st_is3d (geo) Is_3D
 FROM IS3D_TEST;

        ID      IS_3D

      1902          0
      1903          0
      1904          1
      1905          1
```

## PostgreSQL

```
CREATE TABLE is3d_test (
 id integer,
 geo sde.st_geometry
);

INSERT INTO IS3D_TEST VALUES (
 1902,
 sde.st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO IS3D_TEST VALUES (
 1903,
 sde.st_geometry ('multipoint m(10 10 5, 50 10 6, 10 30 8)' , 4326)
);

INSERT INTO IS3D_TEST VALUES (
 1904,
 sde.st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);
```

```
INSERT INTO IS3D_TEST VALUES (
 1905,
 sde.st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, sde.st_is3d (geo)
 AS Is_3D
 FROM is3d_test;

        id      is_3d

      1902          f
      1903          f
      1904          t
      1905          t
```

## SQLite

```
CREATE TABLE is3d_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'is3d_test',
 'geo',
 4326,
 'geometryzm',
 'xyzm',
 'null'
);

INSERT INTO is3d_test VALUES (
 1902,
 st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO is3d_test VALUES (
 1903,
 st_geometry ('multipoint m((10 10 5), (50 10 6), (10 30 8))' , 4326)
);

INSERT INTO is3d_test VALUES (
 1904,
 st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);

INSERT INTO is3d_test VALUES (
 1905,
 st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, st_is3d (geo)
 AS "Is_3D"
 FROM is3d_test;

      id      Is_3D
```

```
      1902          0
      1903          0
      1904          1
      1905          1
```

# ST_IsClosed

## Definition

ST_IsClosed takes an ST_LineString or ST_MultiLineString and returns 1 (Oracle and SQLite) or t (PostgreSQL) if it is closed; otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_isclosed (line1 sde.st_geometry)
sde.st_isclosed (multiline1 sde.st_geometry)
```

### SQLite

```
st_isclosed (geometry1 geometryblob)
```

## Return type

Boolean

## Examples

### Testing a linestring

The closed_linestring table is created with a single linestring column.

The INSERT statements insert two records into the closed_linestring table. The first record is not a closed linestring, while the second is.

The query returns the results of the ST_IsClosed function. The first row returns a 0 or f because the linestring is not closed, while the second row returns a 1 or t because the linestring is closed.

*Oracle*

```
CREATE TABLE closed_linestring (ln1 sde.st_geometry);
```

```
INSERT INTO CLOSED_LINESTRING VALUES (
 sde.st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO CLOSED_LINESTRING VALUES (
 sde.st_linefromtext ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01)', 4326)
);
```

```
SELECT sde.st_isclosed (ln1) Is_it_closed
 FROM CLOSED_LINESTRING;
```

```
Is_it_closed

0
1
```

*PostgreSQL*

```
CREATE TABLE closed_linestring (ln1 sde.st_geometry);
```

```
INSERT INTO closed_linestring VALUES (
 sde.st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO closed_linestring VALUES (
sde.st_linestring ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01)', 4326)
);
```

```
SELECT sde.st_isclosed (ln1) AS Is_it_closed
FROM closed_linestring;

is_it_closed

f
t
```

*SQLite*

```
CREATE TABLE closed_linestring (id integer);

SELECT AddGeometryColumn (
 NULL,
 'closed_linestring',
 'ln1',
 4326,
 'linestring',
 'xy',
 'null'
);
```

```
INSERT INTO closed_linestring VALUES (
 1,
 st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO closed_linestring VALUES (
 2,
 st_linefromtext ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01)', 4326)
);
```

```
SELECT st_isclosed (ln1)
 AS "Is_it_closed"
 FROM closed_linestring;

Is_it_closed

0
1
```

## Testing a multilinestring

The closed_mlinestring table is created with a single ST_MultiLineString column.

The INSERT statements insert an ST_MultiLineString record that is not closed and another that is.

The query lists the results of the ST_IsClosed function. The first row returns 0 or f because the multilinestring is not closed. The second row returns 1 or t because the multilinestring stored in the ln1 column is closed. A multilinestring is closed if all its linestring elements are closed.

*Oracle*

```
CREATE TABLE closed_mlinestring (mln1 sde.st_geometry);
```

```
INSERT INTO closed_mlinestring VALUES (
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO closed_mlinestring VALUES (
```

```
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 11.92 35.64, 25.02
34.15, 19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90,
51.71 21.73))', 4326)
);
```

```
SELECT sde.st_isclosed (mln1) Is_it_closed
 FROM CLOSED_MLINESTRING;

Is_it_closed

0
1
```

*PostgreSQL*

```
CREATE TABLE closed_mlinestring (mln1 sde.st_geometry);
```

```
INSERT INTO closed_mlinestring VALUES (
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO closed_mlinestring VALUES (
 sde.st_mlinefromtext ('multilinestring ((10.02 20.01, 11.92 35.64, 25.02
34.15, 19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90,
51.71 21.73))', 4326)
);
```

```
SELECT st_isclosed (mln1)
 AS Is_it_closed
 FROM closed_mlinestring;

is_it_closed

f
t
```

*SQLite*

```
CREATE TABLE closed_mlinestring (mln1 geometryblob);

SELECT AddGeometryColumn (
 NULL,
 'closed_mlinestring',
 'mln1',
 4326,
 'multilinestring',
 'xy',
 'null'
);
```

```
INSERT INTO closed_mlinestring VALUES (
 st_mlinefromtext ('multilinestring ((10.02 20.01, 10.32 23.98, 11.92 25.64), (9.55
23.75, 15.36 30.11))', 4326)
);

INSERT INTO closed_mlinestring VALUES (
 st_mlinefromtext ('multilinestring ((10.02 20.01, 11.92 35.64, 25.02
34.15, 19.15 33.94, 10.02 20.01), (51.71 21.73, 73.36 27.04, 71.52 32.87, 52.43 31.90,
51.71 21.73))', 4326)
);
```

```
SELECT sde.st_isclosed (mln1)
 AS "Is_it_closed"
 FROM CLOSED_MLINESTRING;

Is_it_closed

0
1
```

# ST_IsEmpty

## Definition

ST_IsEmpty returns 1 (Oracle and SQLite) or t (PostgreSQL) if the ST_Geometry object is empty; otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_isempty (geometry1 sde.st_geometry)
```

### SQLite

```
st_isempty (geometry1 geometryblob)
```

## Return type

Boolean

## Example

The CREATE TABLE statement below creates the empty_test table with geotype, which stores the data type of the subclasses stored in the g1 column.

The INSERT statements insert two records for the geometry subclasses point, linestring, and polygon: one that is empty and one that is not.

The SELECT query returns the geometry type from the geotype column and the results of the ST_IsEmpty function.

### Oracle

```
CREATE TABLE empty_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO EMPTY_TEST VALUES (
 'Point',
 sde.st_pointfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO EMPTY_TEST VALUES (
 'Point',
 sde.st_pointfromtext ('point empty', 4326)
);

INSERT INTO EMPTY_TEST VALUES (
 'Linestring',
 sde.st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO EMPTY_TEST VALUES (
 'Linestring',
```

```
 sde.st_linefromtext ('linestring empty', 4326)
);

INSERT INTO EMPTY_TEST VALUES (
 'Polygon',
 sde.st_polyfromtext ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);

INSERT INTO EMPTY_TEST VALUES (
 'Polygon',
 sde.st_polyfromtext('polygon empty', 4326)
);
```

```
SELECT geotype, sde.st_isempty (g1) Is_it_empty
FROM EMPTY_TEST;

GEOTYPE    Is_it_empty

Point         0
Point         1
Linestring    0
Linestring    1
Polygon       0
Polygon       1
```

## PostgreSQL

```
CREATE TABLE empty_test (
 geotype varchar(20),
 g1 sde.st_geometry
);

INSERT INTO empty_test VALUES (
 'Point',
 sde.st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO empty_test VALUES (
 'Point',
 sde.st_point ('point empty', 4326)
);

INSERT INTO empty_test VALUES (
 'Linestring',
 sde.st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO empty_test VALUES (
 'Linestring',
 sde.st_linestring ('linestring empty', 4326)
);

INSERT INTO empty_test VALUES (
 'Polygon',
 sde.st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);
```

```
INSERT INTO empty_test VALUES (
 'Polygon',
 sde.st_polygon ('polygon empty', 4326)
);
```

```
SELECT geotype, sde.st_isempty (g1)
 AS Is_it_empty
 FROM empty_test;

geotype   is_it_empty

Point         f
Point         t
Linestring    f
Linestring    t
Polygon       f
Polygon       f
```

## SQLite

```
CREATE TABLE empty_test (
 geotype text(20)
);

SELECT AddGeometryColumn (
 NULL,
 'empty_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO empty_test VALUES (
 'Point',
 st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO empty_test VALUES (
 'Point',
 st_point ('point empty', 4326)
);

INSERT INTO empty_test VALUES (
 'Linestring',
 st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO empty_test VALUES (
 'Linestring',
 st_linestring ('linestring empty', 4326)
);

INSERT INTO empty_test VALUES (
 'Polygon',
 st_polygon ('polygon ((10.02 20.01, 11.92 35.64, 25.02 34.15,
19.15 33.94, 10.02 20.01))', 4326)
);
```

```
INSERT INTO empty_test VALUES (
 'Polygon',
 st_polygon ('polygon empty', 4326)
);
```

```
SELECT geotype, st_isempty (g1)
 AS "Is_it_empty"
 FROM empty_test;

GEOTYPE    Is_it_empty

Point         0
Point         1
Linestring    0
Linestring    1
Polygon       0
Polygon       1
```

# ST_IsMeasured

## Definition

ST_IsMeasured takes a geometry object as an input parameter and returns 1 (Oracle and SQLite) or t (PostgreSQL) if the given geometry has measures; otherwise, a 0 (Oracle and SQLite) or f (PostgreSQL) is returned.

## Syntax

### Oracle and PostgreSQL

```
sde.st_ismeasured (geometry1 sde.st_geometry)
```

### SQLite

```
st_ismeasured (geometry1 geometryblob)
```

## Return type

Boolean

## Example

Create a table, ism_test, insert values to it, then determine which rows in the ism_test table contain measures.

## Oracle

```
CREATE TABLE ism_test (
 id integer,
 geom sde.st_geometry
);

INSERT INTO ISM_TEST VALUES (
 19,
 sde.st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO ISM_TEST VALUES (
 20,
 sde.st_geometry ('multipoint m((10 10 5), (50 10 6), (10 30 8))' , 4326)
);

INSERT INTO ISM_TEST VALUES (
 21,
 sde.st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);

INSERT INTO ISM_TEST VALUES (
 22,
 sde.st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, sde.st_ismeasured (geom) M_values
 FROM ISM_TEST;

      ID    M_values

      19          0
      20          1
      21          0
      22          1
```

## PostgreSQL

```
CREATE TABLE ism_test (
 id integer,
 geom sde.st_geometry
);

INSERT INTO ISM_TEST VALUES (
 19,
 sde.st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO ISM_TEST VALUES (
 20,
 sde.st_geometry ('multipoint m(10 10 5, 50 10 6, 10 30 8)' , 4326)
);

INSERT INTO ISM_TEST VALUES (
 21,
 sde.st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);
```

```
INSERT INTO ISM_TEST VALUES (
 22,
 sde.st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, sde.st_ismeasured (geom)
  AS has_measures
  FROM ism_test;

      id   has_measures

      19          f
      20          t
      21          f
      22          t
```

## SQLite

```
CREATE TABLE ism_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'ism_test',
 'geom',
 4326,
 'geometryzm',
 'xyzm',
 'null'
);

INSERT INTO ISM_TEST VALUES (
 19,
 st_geometry ('polygon ((40 120, 90 120, 90 150, 40 150, 40 120))', 4326)
);

INSERT INTO ISM_TEST VALUES (
 20,
 st_geometry ('multipoint m((10 10 5), (50 10 6), (10 30 8))' , 4326)
);

INSERT INTO ISM_TEST VALUES (
 21,
 st_geometry ('linestring z(10 10 166, 20 10 168)', 4326)
);

INSERT INTO ISM_TEST VALUES (
 22,
 st_geometry ('point zm(10 10 16 30)', 4326)
);
```

```
SELECT id, st_ismeasured (geom)
 AS "M_values"
 FROM ism_test;

        ID    M_values
```

```
      19          0
      20          1
      21          0
      22          1
```

# ST_IsRing

## Definition

ST_IsRing takes an ST_LineString and returns 1 (Oracle and SQLite) or t (PostgreSQL) if it is a ring (for example, the ST_LineString is closed and simple); otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_isring (line1 sde.st_geometry)
```

### SQLite

```
st_isring (line1 geometryblob)
```

## Return type

Boolean

## Example

The ring_linestring table is created with a single ST_LineString column, ln1.

The INSERT statements insert three linestrings into the ln1 column. The first row contains a linestring that's not closed and isn't a ring. The second row contains a closed and simple linestring that is a ring. The third row contains a linestring that is closed but not simple because it intersects its own interior. It is also not a ring.

The SELECT query returns the results of the ST_IsRing function. The first row returns 0 or f because the linestrings aren't rings, while the second and third rows return 1 or t because they are rings.

### Oracle

```
CREATE TABLE ring_linestring (ln1 sde.st_geometry);
```

```
INSERT INTO RING_LINESTRING VALUES (
 sde.st_linefromtext ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO RING_LINESTRING VALUES (
 sde.st_linefromtext ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01)', 4326)
);

INSERT INTO ring_linestring (ln1) VALUES (
 sde.st_linestring ('linestring (11 31, 11.25 31.12, 21.83 44.13, 16.45 44.24, 11 31)',
4326)
);
```

```
SELECT sde.st_isring (ln1) Is_it_a_ring
 FROM RING_LINESTRING;

Is_it_a_ring

0
1
1
```

### PostgreSQL

```
CREATE TABLE ring_linestring (ln1 sde.st_geometry);
```

```
INSERT INTO ring_linestring VALUES (
 sde.st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO ring_linestring VALUES (
 sde.st_linestring ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94,
10.02 20.01)', 4326)
);

INSERT INTO ring_linestring (ln1) VALUES (
 sde.st_linestring ('linestring (11 31, 11.25 31.12, 21.83 44.13, 16.45 44.24, 11 31)',
4326)
);
```

```
SELECT sde.st_isring (ln1)
 AS Is_it_a_ring
 FROM ring_linestring;

Is_it_a_ring

f
```

```
t
t
```

## SQLite

```
CREATE TABLE ring_linestring (id integer primary key autoincrement not null);

SELECT AddGeometryColumn (
 NULL,
 'ring_linestring',
 'ln1',
 4326,
 'linestring',
 'xy',
 'null'
);
```

```
INSERT INTO ring_linestring (ln1) VALUES (
 st_linestring ('linestring (10.02 20.01, 10.32 23.98, 11.92 25.64)', 4326)
);

INSERT INTO ring_linestring (ln1) VALUES (
 st_linestring ('linestring (10.02 20.01, 11.92 35.64, 25.02 34.15, 19.15 33.94, 10.02
20.01)', 4326)
);

INSERT INTO ring_linestring (ln1) VALUES (
 st_linestring ('linestring (11 31, 11.25 31.12, 21.83 44.13, 16.45 44.24, 11 31)',
4326)
);
```

```
SELECT st_isring (ln1)
 AS "Is it a ring?"
 FROM ring_linestring;

Is it a ring?

0
1
1
```

# ST_IsSimple

## Definition

ST_IsSimple returns 1 (Oracle and SQLite) or t (PostgreSQL) if the geometry object is simple as defined by the Open Geospatial Consortium (OGC); otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_issimple (geometry1 sde.st_geometry)
```

### SQLite

```
st_issimple (geometry1 geometryblob)
```

## Return type

Boolean

## Example

The table issimple_test is created with two columns. The pid column is a smallint data type containing the unique identifier for each row. The g1 column stores the simple and nonsimple geometry samples.

The INSERT statements insert two records into the issimple_test table. The first is a simple linestring because it doesn't intersect its interior. The second is nonsimple, as defined by the OGC, because it does intersect its interior.

The query returns the results of the ST_IsSimple function. The first record returns 1 or t because the linestring is simple, while the second record returns 0 or f because the linestring is not simple.

## Oracle

```
CREATE TABLE issimple_test (
 pid smallint,
 g1 sde.st_geometry
);
```

```
INSERT INTO ISSIMPLE_TEST VALUES (
 1,
 sde.st_linefromtext ('linestring (10 10, 20 20, 30 30)', 4326)
);

INSERT INTO ISSIMPLE_TEST VALUES (
 2,
 sde.st_linefromtext ('linestring (10 10, 20 20, 20 30, 10 30, 10 20,
20 10)', 4326)
);
```

```
SELECT pid, sde.st_issimple (g1) Is_it_simple
 FROM ISSIMPLE_TEST;

PID                 Is_it_simple

1                    1
2                    0
```

## PostgreSQL

```
CREATE TABLE issimple_test (
 pid smallint,
 g1 sde.st_geometry
);
```

```
INSERT INTO issimple_test VALUES (
 1,
 sde.st_linestring ('linestring (10 10, 20 20, 30 30)', 4326)
);

INSERT INTO issimple_test VALUES (
 2,
 sde.st_linestring  ('linestring (10 10, 20 20, 20 30, 10 30, 10 20, 20 10)', 4326)
);
```

```
SELECT pid, sde.st_issimple (g1)
 AS Is_it_simple
 FROM issimple_test;

pid                is_it_simple

1                    t
2                    f
```

## SQLite

```
CREATE TABLE issimple_test (
 pid integer
);

SELECT AddGeometryColumn (
 NULL,
 'issimple_test',
 'g1',
 4326,
 'linestring',
 'xy',
 'null'
);
```

```
INSERT INTO issimple_test VALUES (
 1,
 st_linestring ('linestring (10 10, 20 20, 30 30)', 4326)
);

INSERT INTO issimple_test VALUES (
 2,
 st_linestring  ('linestring (10 10, 20 20, 20 30, 10 30, 10 20, 20 10)', 4326)
);
```

```
SELECT pid, st_issimple (g1)
 AS Is_it_simple
 FROM issimple_test;

PID                 Is_it_simple

1                    1
2                    0
```

# ST_Length

## Definition

ST_Length returns the length of a line string or multiline string.

## Syntax

### Oracle and PostgreSQL

```
sde.st_length (line1 sde.st_geometry)
sde.st_length (multiline1 sde.st_geometry)
```

### SQLite

```
st_length (line1 geometryblob)
st_length (multiline1 geometryblob)
st_length (line1  geometryblob, unit_name text)
st_length (multiline1  geometryblob, unit_name text)
```

For a list of supported unit names, see ST_Distance.

## Return type

Double precision

## Example

An ecologist studying the migratory patterns of the salmon population in the country's waterways wants the length of all stream and river systems within the country.

The waterways table is created with the ID and name columns, which identify each stream and river system stored in the table. The water column is a multilinestring because the river and stream systems are often aggregates of several linestrings.

The SELECT query returns the name of the system along with the length of the system generated by the ST_Length function. In Oracle and PostgreSQL, the units are those used by the coordinate system. In SQLite, kilometer units are specified.

### Oracle

```
CREATE TABLE waterways (
 oid integer,
 name varchar(128),
 water sde.st_geometry
);
```

```
INSERT INTO waterways (oid, name, water) VALUES (
  1111,
  'Genesee',
  sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6),
```

```
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 4326)
);
```

```
SELECT name, sde.st_length (water) "Length"
 FROM WATERWAYS;

NAME        Length

Genesee        27.6437123
```

## PostgreSQL

```
CREATE TABLE waterways (
 oid serial,
 name varchar(128),
 water sde.st_geometry
);
```

```
INSERT INTO waterways (name, water) VALUES (
  'Genesee',
  sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 4326)
);
```

```
SELECT name AS "Watershed Name",
 sde.st_length (water) AS "Length"
 FROM waterways;

Watershed Name         |  Length

Genesee               | 27.6437123387202
```

## SQLite

```
CREATE TABLE waterways (
 oid integer primary key autoincrement not null,
 name text(128)
);

SELECT AddGeometryColumn (
 NULL,
 'waterways',
 'water',
 4326,
 'multilinestring',
 'xy',
 'null'
);
```

```
INSERT INTO waterways (name, water) VALUES (
  'Genesee',
  st_multilinestring ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 4326)
);
```

```
SELECT name AS "Watershed Name",
 st_length (water, 'kilometer') AS "Length"
 FROM waterways1;

Watershed Name            Length

Genesee                   3047.75515002795
```

# ST_LineFromText

**Note:**
Supported in Oracle and SQLite only; for PostgreSQL, use ST_LineString.

## Definition

ST_LineFromText takes a well-known text representation of type ST_LineString and a spatial reference ID and returns an ST_LineString.

## Syntax

### Oracle

```
sde.st_linefromtext (wkt clob, srid integer)
```

```
sde.st_linefromtext (wkt clob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### SQLite

```
st_linefromtext (wkt text, srid int32)
```

```
st_linefromtext (wkt text)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

ST_LineString

## Example

The linestring_test table is created with a single ln1 ST_LineString column.

The INSERT statement inserts an ST_LineString into the ln1 column using the ST_LineFromText function.

### Oracle

```
CREATE TABLE linestring_test (ln1 sde.st_geometry);
```

```
INSERT INTO LINESTRING_TEST VALUES (
 sde.st_linefromtext ('linestring (10.01 20.03, 35.93 19.04)', 4326)
);
```

## SQLite

```
CREATE TABLE linestring_test (id integer);

SELECT AddGeometryColumn (
 NULL,
 'linestring_test',
 'ln1',
 4326,
 'linestring',
 'xy',
 'null'
);
```

```
INSERT INTO LINESTRING_TEST (id, ln1) VALUES (
 1,
 st_linefromtext ('linestring (10.01 20.03, 35.93 19.04)', 4326)
);
```

# ST_LineFromWKB

## Definition

ST_LineFromWKB takes a well-known binary (WKB) representation of type ST_LineString and a spatial reference ID and returns an ST_LineString.

## Syntax

### Oracle

```
sde.st_linefromwkb (wkb blob, srid integer)
```

```
sde.st_linefromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### PostgreSQL

```
sde.st_linefromwkb (wkb bytea, srid integer)
```

### SQLite

```
st_linefromwkb (wkb blob, srid int32)
```

```
st_linefromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

ST_LineString

## Example

The following commands create a table (sample_lines) and use the ST_LineFromWKB function to insert lines from a WKB representation. The row is inserted into the sample_lines table with an ID and a line in spatial reference system 4326 in the WKB representation.

### Oracle

```
CREATE TABLE sample_lines (
 id smallint,
 geometry sde.st_linestring,
 wkb blob
);
```

```
INSERT INTO SAMPLE_LINES (id, geometry) VALUES (
 1901,
 sde.st_linestring ('linestring (850 250, 850 850)', 4326)
);

INSERT INTO SAMPLE_LINES (id, geometry) VALUES (
 1902,
 sde.st_linestring ('linestring (33 2, 34 3, 35 6)', 4326)
);

UPDATE SAMPLE_LINES
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1901;

UPDATE SAMPLE_LINES
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1902;

SELECT id, sde.st_astext (sde.st_linefromwkb (wkb,4326)) LINE
 FROM SAMPLE_LINES;

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)
```

## PostgreSQL

```
CREATE TABLE sample_lines (
 id serial,
 geometry sde.st_linestring,
 wkb bytea
);

INSERT INTO sample_lines (geometry) VALUES (
 sde.st_linestring ('linestring (850 250, 850 850)', 4326)
);

INSERT INTO sample_lines (geometry) VALUES (
 sde.st_linestring ('linestring (33 2, 34 3, 35 6)', 4326)
);

--Replace ID values if necessary.
UPDATE sample_lines
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1;

UPDATE sample_lines
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 2;

SELECT id, sde.st_astext (st_linefromwkb (wkb,4326))
 AS LINE
 FROM sample_lines;

id   line

1    LINESTRING (850 250, 850 850)
2    LINESTRING (33 2, 34 3, 35 6)
```

## SQLite

```
CREATE TABLE sample_lines (
 id integer primary key autoincrement not null,
 wkb blob
);

SELECT AddGeometryColumn (
 NULL,
 'sample_lines',
 'geometry',
 4326,
 'linestring',
 'xy',
 'null'
);

INSERT INTO sample_lines (geometry) VALUES (
 st_linestring ('linestring (850 250, 850 850)', 4326)
);

INSERT INTO sample_lines (geometry) VALUES (
 st_linestring ('linestring (33 2, 34 3, 35 6)', 4326)
);

--Replace ID values if necessary.
UPDATE sample_lines
 SET wkb = st_asbinary (geometry)
 WHERE id = 1;

UPDATE sample_lines
 SET wkb = st_asbinary (geometry)
 WHERE id = 2;

SELECT id, st_astext (st_linefromwkb (wkb,4326))
 AS LINE
 FROM sample_lines;

id   LINE

1    LINESTRING (850.00000000 250.00000000, 850.00000000 850.00000000)
2    LINESTRING (33.00000000 2.00000000, 34.00000000 3.00000000, 35.00000000 6.00000000)
```

# ST_LineString

## Definition

ST_LineString is an accessor function that constructs a linestring from a well-known text representation.

**Note:**
When creating spatial tables that will be used with ArcGIS, it is best to create the column as the geometry supertype (for example, ST_Geometry) rather than specifying an ST_Geometry subtype.

## Syntax

### Oracle

```
sde.st_linestring (wkt clob, srid integer)
```

### PostgreSQL

```
sde.st_linestring (wkt text, srid integer)
sde.st_linestring (esri_shape bytea, srid integer)
```

### SQLite

```
st_linestring (wkt text, srid int32)
```

## Return type

ST_LineString

## Example

### Oracle

```
CREATE TABLE lines_test (
 id smallint,
 geometry sde.st_geometry
);

INSERT INTO LINES_TEST (id, geometry) VALUES (
 1901,
 sde.st_linestring ('linestring (750 150, 750 750)', 4326)
);

SELECT id, sde.st_astext (geometry) Linestring
 FROM   LINES_TEST;

  ID  LINESTRING

1901  LINESTRING  (750.00000000 150.00000000,
750.00000000 750.00000000)
```

### PostgreSQL

```
CREATE TABLE lines_test (
 id serial,
 geometry sde.st_geometry
);

INSERT INTO lines_test (geometry) VALUES (
 sde.st_linestring ('linestring (750 150, 750 750)', 4326)
);

SELECT id, sde.st_astext (geometry)
 AS Linestring
 FROM lines_test;

  id  linestring

  1  LINESTRING  (750 150, 750 750)
```

### SQLite

```
CREATE TABLE lines_test (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'lines_test',
 'geometry',
 4326,
 'linestring',
 'xy',
 'null'
);

INSERT INTO lines_test (geometry) VALUES (
 st_linestring ('linestring (750 150, 750 750)', 4326)
);

SELECT id, st_astext (geometry)
 AS "Linestring"
 FROM lines_test;

  id  linestring

  1  LINESTRING  (750.00000000 150.00000000, 750.00000000 750.00000000)
```

# ST_M

## Definition

ST_M takes an ST_Point as an input parameter and returns its measure (m) coordinate.

In SQLite, ST_M can also be used to update a measure value.

## Syntax

### Oracle and PostgreSQL

```
sde.st_m (point1 sde.st_point)
```

### SQLite

```
st_m (point1 geometryblob)
st_m (point1 geometryblob, new_Mvalue double)
```

## Return type

### Oracle and PostgreSQL

Number

### SQLite

Double precision if querying for a measure value; a geometryblob if updating a measure value

## Examples

### Oracle

The table, m_test, is created and three points inserted to it. All three contain measure values. A SELECT statement is run with the ST_M function to return the measure value for each point.

```
CREATE TABLE m_test (
 id integer,
 geometry sde.st_point);

INSERT INTO M_TEST VALUES (
 1,
 sde.st_point (2, 3, 32, 5, 4322)
);

INSERT INTO M_TEST VALUES (
 2,
 sde.st_point (4, 5, 20, 4, 4326)
);

INSERT INTO M_TEST VALUES (
 3,
 sde.st_point (3, 8, 23, 7, 4326)
);
```

```
SELECT id, sde.st_m (geometry) M_COORD
 FROM M_TEST;

        ID    M_COORD

         1          5
         2          4
         3          7
```

## PostgreSQL

The table, m_test, is created and three points inserted to it. All three contain measure values. A SELECT statement is run with the ST_M function to return the measure value for each point.

```
CREATE TABLE m_test (
 id serial,
 geometry sde.st_point
);

INSERT INTO m_test (geometry) VALUES (
 sde.st_point (2, 3, 32, 5, 4326)
);

INSERT INTO m_test (geometry) VALUES (
 sde.st_point (4, 5, 20, 4, 4326)
);

INSERT INTO m_test (geometry) VALUES (
 sde.st_point (3, 8, 23, 7, 4326)
);

SELECT id, sde.st_m (geometry)
 AS M_COORD
 FROM m_test;

        id    m_coord

         1          5
         2          4
         3          7
```

## SQLite

In the first example, the table, m_test, is created and three points inserted to it. All three contain measure values. A SELECT statement is run with the ST_M function to return the measure value for each point.

```
CREATE TABLE m_test (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn (
 NULL,
 'm_test',
 'geometry',
 4326,
 'pointzm',
 'xyzm',
```

```
 'null'
);

INSERT INTO m_test (geometry) VALUES (
 st_point (2, 3, 32, 5, 4326)
);

INSERT INTO m_test (geometry) VALUES (
 st_point (4, 5, 20, 4, 4326)
);

INSERT INTO m_test (geometry) VALUES (
 st_point (3, 8, 23, 7, 4326)
);

SELECT id, st_m (geometry)
 AS M_COORD
 FROM m_test;

id    m_coord

1     5.0
2     4.0
3     7.0
```

In this second example, the measure value is updated for record 3 in the m_test table.

```
SELECT st_m (geometry, 7.5)
 FROM m_test
 WHERE id = 3;
```

# ST_MaxM

## Definition

ST_MaxM takes a geometry as an input parameter and returns its maximum m-coordinate.

## Syntax

### Oracle and PostgreSQL

```
sde.st_maxm (geometry1 sde.st_geometry)
```

### SQLite

```
st_maxm (geometry1 geometryblob)
```

## Return type

### Oracle and PostgreSQL

Number

If no m values are present, NULL is returned.

### SQLite

Double precision

If no m values are present, NULL is returned.

## Example

The table, maxm_test, is created and two polygons inserted to it. Then, ST_MaxM is run to determine the maximum m-value in each polygon.

### Oracle

```
CREATE TABLE maxm_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MAXM_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO MAXM_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
```

```
SELECT id, sde.st_maxm (geometry) Max_M
 FROM MAXM_TEST;

        ID             MAX_M

      1901                 4
      1902                12
```

## PostgreSQL

```
CREATE TABLE maxm_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO maxm_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO maxm_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_maxm (geometry)
 AS Max_M
 FROM maxm_test;

        id             max_m

      1901                 4
      1902                12
```

## SQLite

```
CREATE TABLE maxm_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'maxm_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);

INSERT INTO maxm_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);

INSERT INTO maxm_test VALUES (
```

```
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);

SELECT id, st_maxm (geometry)
 AS "Max M"
 FROM maxm_test;

id             Max M

1901           4.0
1902           12.0
```

# ST_MaxX

## Definition

ST_MaxX takes a geometry as an input parameter and returns its maximum x-coordinate.

## Syntax

### Oracle and PostgreSQL

```
sde.st_maxx (geometry1 sde.st_geometry)
```

### SQLite

```
st_maxx (geometry1 geometryblob)
```

## Return type

### Oracle and PostgreSQL

Number

### SQLite

Double precision

## Example

The table, maxx_test, is created and two polygons inserted to it. The ST_MaxX function is then used to determine the maximum x-value in each polygon.

### Oracle

```
CREATE TABLE maxx_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MAXX_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO MAXX_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_maxx (geometry) Max_X
 FROM MAXX_TEST;

        ID      MAX_X
```

```
      1901        120
      1902          5
```

## PostgreSQL

```
CREATE TABLE maxx_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO maxx_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO maxx_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_maxx (geometry)
 AS Max_X
 FROM maxx_test;

        id      max_x

      1901        120
      1902          5
```

## SQLite

```
CREATE TABLE maxx_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'maxx_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);

INSERT INTO maxx_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);

INSERT INTO maxx_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
```

```
SELECT id, st_maxx (geometry)
 AS "max_x"
 FROM maxx_test;

id      max_x

1901    120.0
1902    5.00000000
```

# ST_MaxY

## Definition

ST_MaxY takes a geometry as an input parameter and returns its maximum y-coordinate.

## Syntax

### Oracle and PostgreSQL

```
sde.st_maxy (geometry1 sde.st_geometry)
```

### SQLite

```
st_maxy (geometry1 geometryblob)
```

## Return type

### Oracle and PostgreSQL

Number

### SQLite

Double precision

## Example

The table, maxy_test, is created and two polygons inserted to it. The ST_MaxY function is then used to determine the maximum y-value in each polygon.

### Oracle

```
CREATE TABLE maxy_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MAXY_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO MAXY_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_maxy (geometry) Max_Y
 FROM MAXY_TEST;

        ID      MAX_Y
```

```
      1901        140
      1902          4
```

## PostgreSQL

```
CREATE TABLE maxy_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO maxy_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO maxy_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_maxy (geometry)
 AS Max_Y
 FROM maxy_test;

        id      max_y

      1901        140
      1902          4
```

## SQLite

```
CREATE TABLE maxy_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'maxy_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);

INSERT INTO maxy_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);

INSERT INTO maxy_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
```

```
SELECT id, st_maxy (geometry)
 AS "max_y"
 FROM maxy_test;

id      max_y

1901    140.0
1902    4.00000000
```

# ST_MaxZ

## Definition

ST_MaxZ takes a geometry as an input parameter and returns its maximum z-coordinate.

## Syntax

### Oracle and PostgreSQL

```
sde.st_maxz (geometry1 sde.st_geometry)
```

### SQLite

```
st_maxz (geometry1 geometryblob)
```

## Return type

### Oracle and PostgreSQL

Number

If no z values are present, NULL is returned.

### SQLite

Double precision

If no z values are present, NULL is returned.

## Example

In the following example, table maxz_test is created and two polygons inserted to it. ST_MaxZ is then run to return the maximum z-value for each polygon.

### Oracle

```
CREATE TABLE maxz_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MAXZ_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO MAXZ_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
```

```
SELECT id, sde.st_maxz (geometry) Max_Z
 FROM MAXZ_TEST;

        ID      MAX_Z

      1901         26
      1902         40
```

## PostgreSQL

```
CREATE TABLE maxz_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO maxz_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO maxz_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_maxz (geometry)
 AS Max_Z
 FROM maxz_test;

        id      max_z

      1901         26
      1902         40
```

## SQLite

```
CREATE TABLE maxz_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'maxz_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);

INSERT INTO maxz_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);

INSERT INTO maxz_test VALUES (
```

```
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);

SELECT id AS "ID", st_maxz (geometry) AS "Max Z"
 FROM maxz_test;

ID      Max Z

1901    26.0
1902    40.0
```

# ST_MinM

## Definition

ST_MinM takes a geometry as an input parameter and returns its minimum m-coordinate.

## Syntax

### Oracle and PostgreSQL

```
sde.st_minm (geometry1 sde.st_geometry)
```

### SQLite

```
st_minm (geometry1 geometryblob)
```

## Return type

### Oracle and PostgreSQL

Number

If no m values are present, NULL is returned.

### SQLite

Double precision

If no m values are present, NULL is returned.

## Example

The table, minm_test, is created and two polygons inserted to it. ST_MinM is then run to determine the minimum measure value in each polygon.

**PostgreSQL**

### Oracle

```
CREATE TABLE minm_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MINM_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO MINM_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
```

```
);

SELECT id, sde.st_minm (geometry) MinM
 FROM MINM_TEST;

        ID       MINM

      1901          3
      1902          5
```

## PostgreSQL

```
CREATE TABLE minm_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO minm_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO minm_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_minm (geometry)
 AS MinM
 FROM minm_test;

        id       minm

      1901          3
      1902          5
```

## SQLite

```
CREATE TABLE minm_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'minm_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);

INSERT INTO minm_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);
```

```
INSERT INTO minm_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);

SELECT id, st_minm (geometry)
 AS "MinM"
 FROM minm_test;

id       MinM

1901     3.0
1902     5.0
```

# ST_MinX

## Definition

ST_MinX takes a geometry as an input parameter and returns its minimum x-coordinate.

## Syntax

### Oracle and PostgreSQL

```
sde.st_minx (geometry1 sde.st_geometry)
```

### SQLite

```
st_minx (geometry1 geometryblob)
```

## Return type

### Oracle and PostgreSQL

Number

### SQLite

Double precision

## Example

The table, minx_test, is created and two polygons inserted to it. ST_MinX is then run to determine the minimum x-coordinate value in each polygon.

### Oracle

```
CREATE TABLE minx_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MINX_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO MINX_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_minx (geometry) MinX
 FROM MINX_TEST;

        ID       MINX
```

```
      1901        110
      1902          0
```

## PostgreSQL

```
CREATE TABLE minx_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO minx_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO minx_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_minx (geometry)
 AS MinX
 FROM minx_test;

        id       minx

      1901        110
      1902          0
```

## SQLite

```
CREATE TABLE minx_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'minx_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);

INSERT INTO minx_test VALUES (
 1914,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);

INSERT INTO minx_test VALUES (
 1915,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);
```

```
SELECT id AS "ID", st_minx (geometry) AS "MinX"
 FROM minx_test;

ID       MinX

1914     110.0
1915       0.0
```

# ST_MinY

## Definition

ST_MinY takes a geometry as an input parameter and returns its minimum y-coordinate.

## Syntax

### Oracle and PostgreSQL

```
sde.st_miny (geometry1 sde.st_geometry)
```

### SQLite

```
st_miny (geometry1 geometryblob)
```

## Return type

### Oracle and PostgreSQL

Number

### SQLite

Double precision

## Example

The table, miny_test, is created and two polygons inserted to it. ST_MinY is then run to determine the minimum y-coordinate value in each polygon.

**PostgreSQL**

### Oracle

```
CREATE TABLE miny_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MINY_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO MINY_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_miny (geometry) MinY
 FROM MINY_TEST;
```

```
        ID       MINY

      1901        120
      1902          0
```

## PostgreSQL

```
CREATE TABLE miny_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO miny_test VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO miny_test VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);

SELECT id, sde.st_miny (geometry)
 AS MinY
 FROM miny_test;

        id       miny

      1901        120
      1902          0
```

## SQLite

```
CREATE TABLE miny_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'miny_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);

INSERT INTO miny_test VALUES (
 101,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);

INSERT INTO miny_test VALUES (
 102,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
```

```
);

SELECT id, st_miny (geometry)
 AS "MinY"
 FROM miny_test;

id       MinY

101      120.0
102        0.0
```

# ST_MinZ

## Definition

ST_MinZ takes a geometry as an input parameter and returns its minimum z-coordinate.

## Syntax

### Oracle and PostgreSQL

```
sde.st_minz (geometry1 sde.st_geometry)
```

### SQLite

```
st_minz (geometry1 geometryblob)
```

## Return type

### Oracle and PostgreSQL

Number

If no z values are present, NULL is returned.

### SQLite

Double precision

If no z values are present, NULL is returned.

## Example

The table, minz_test, is created and two polygons inserted to it. ST_MinZ is then run to determine the minimum z-coordinate value in each polygon.

### Oracle

```
CREATE TABLE minz_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MINZ_TEST VALUES (
 1901,
 sde.st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20
3))', 4326)
);

INSERT INTO MINZ_TEST VALUES (
 1902,
 sde.st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))',
4326)
);
```

```
SELECT id, sde.st_minz (geometry) MinZ
 FROM MINZ_TEST;

        ID       MINZ

      1901         20
      1902         31
```

## PostgreSQL

```
CREATE TABLE minz_test (
 id integer,
 geometry st_geometry
);

INSERT INTO minz_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);

INSERT INTO minz_test VALUES (
 1902,
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);

SELECT id, st_minz (geometry)
 AS MinZ
 FROM minz_test;

        id       minz

      1901         20
      1902         31
```

## SQLite

```
CREATE TABLE minz_test (
 id integer
);

SELECT AddGeometryColumn (
 NULL,
 'minz_test',
 'geometry',
 4326,
 'polygonzm',
 'xyzm',
 'null'
);

INSERT INTO minz_test VALUES (
 1901,
 st_polygon ('polygon zm((110 120 20 3, 110 140 22 3, 120 130 26 4, 110 120 20 3))',
4326)
);

INSERT INTO minz_test VALUES (
 1902,
```

```
 st_polygon ('polygon zm((0 0 40 7, 0 4 35 9, 5 4 32 12, 5 0 31 5, 0 0 40 7))', 4326)
);

SELECT id, st_minz (geometry)
 AS "MinZ"
 FROM minz_test;

id        MinZ

1901      20.0
1902      31.0
```

# ST_MLineFromText

**Note:**
Used in Oracle and SQLite only; for PostgreSQL, use ST_MultiLineString.

## Definition

ST_MLineFromText takes a well-known text representation of type ST_MultiLineString and a spatial reference ID and returns an ST_MultiLineString.

## Syntax

### Oracle

```
sde.st_mlinefromtext (wkt clob, srid integer)
```

```
sde.st_mlinefromtext (wkt clob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### SQLite

```
st_mlinefromtext (wkt text, srid int32)
```

```
st_mlinefromtext (wkt text)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

ST_MultiLineString

## Example

The mlinestring_test table is created with the gid smallint column that uniquely identifies the row and the ml1 ST_MultiLineString column.

The INSERT statement inserts the ST_MultiLineString using the ST_MLineFromText function.

### Oracle

```
CREATE TABLE mlinestring_test (
 gid smallint,
 ml1 sde.st_geometry
);
```

```
INSERT INTO MLINESTRING_TEST VALUES (
 1,
 sde.st_mlinefromtext ('multilinestring ((10.01 20.03, 10.52 40.11, 30.29 41.56,
31.78 10.74), (20.93 20.81, 21.52 40.10))', 4326)
);
```

### SQLite

```
CREATE TABLE mlinestring_test (
 gid integer
);

SELECT AddGeometryColumn (
 NULL,
 'mlinestring_test',
 'ml1',
 4326,
 'multilinestring',
 'xy',
 'null'
);
```

```
INSERT INTO MLINESTRING_TEST VALUES (
 1,
 st_mlinefromtext ('multilinestring ((10.01 20.03, 10.52 40.11, 30.29 41.56,
31.78 10.74), (20.93 20.81, 21.52 40.10))', 4326)
);
```

# ST_MLineFromWKB

## Definition

ST_MLineFromWKB takes a well-known binary (WKB) representation of type ST_MultiLineString and a spatial reference ID and creates an ST_MultiLineString.

## Syntax

### Oracle

```
sde.st_mlinefromwkb (wkb blob, srid integer)
```

```
sde.st_mlinefromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### PostgreSQL

```
sde.st_mlinefromwkb (wkb bytea, srid integer)
```

### SQLite

```
st_mlinefromwkb (wkb blob, srid int32)
```

```
st_mlinefromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

ST_MultiLineString

## Example

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 4326. In this example, the multilinestring is stored with ID = 10 in the geometry column of the sample_mlines table, and 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 sample_mlines table has a geometry column, where the multilinestring is stored, and a wkb column, where the multilinestring's WKB representation is stored.

The SELECT statement includes the ST_MLineFromWKB function, which is used to retrieve the multilinestring from the wkb column.

## Oracle

```
CREATE TABLE sample_mlines (
 id integer,
 geometry sde.st_geometry,
 wkb blob
);

INSERT INTO SAMPLE_MLINES (id, geometry) VALUES (
 10,
 sde.st_multilinestring ('multilinestring ((61 2, 64 3, 65 6), (58 4, 59 5, 61 8), (69
3, 67 4, 66 7, 68 9))', 4326)
);

UPDATE SAMPLE_MLINES
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mlinefromwkb (wkb,0)) MULTI_LINE_STRING
 FROM SAMPLE_MLINES
 WHERE id = 10;

ID    MULTI_LINE_STRING

10    MULTILINESTRING ((61.00000000 2.00000000, 64.00000000 3.00000000, 65.00000000
6.00000000), (58.00000000 4.00000000, 59.00000000 5.00000000, 61.00000000 8.0000000),
(69.00000000 3.00000000, 67.00000000 4.00000000, 66.00000000 7.00000000, 68.00000000
9.00000000 ))
```

## PostgreSQL

```
CREATE TABLE sample_mlines (
 id integer,
 geometry sde.st_geometry,
 wkb bytea);

INSERT INTO sample_mlines (id, geometry) VALUES (
 10,
 sde.st_multilinestring ('multilinestring ((61 2, 64 3, 65 6), (58 4, 59 5, 61 8), (69
3, 67 4, 66 7, 68 9))', 4326)
);

UPDATE sample_mlines
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mlinefromwkb (wkb,4326))
 AS MULTI_LINE_STRING
 FROM sample_mlines
 WHERE id = 10;

id   multi_line_string

10    MULTI_LINE_STRING ((61 2, 64 3, 65 6), (58 4, 59 5,61 8), (69 3, 67 4, 66 7, 68 9
))
```

## SQLite

```
CREATE TABLE sample_mlines (
 id integer,
 wkb blob);

SELECT AddGeometryColumn (
 NULL,
 'sample_mlines',
 'geometry',
 4326,
 'multilinestring',
 'xy',
 'null'
);

INSERT INTO sample_mlines (id, geometry) 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))', 4326)
);

UPDATE sample_mlines
 SET wkb = st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, st_astext (st_mlinefromwkb (wkb,4326))
 AS MULTI_LINE_STRING
 FROM sample_mlines
 WHERE id = 10;

id   multi_line_string

10    MULTI_LINE_STRING ((61.00000000 2.00000000, 64.00000000 3.00000000, 65.00000000
6.00000000),
 (58.00000000 4.00000000, 59.00000000 5.00000000, 61.00000000 8.00000000),
 (69.00000000 3.00000000, 67.00000000 4.00000000, 66.00000000 7.00000000, 68.00000000
9.00000000 ))
```

# ST_MPointFromText

**Note:**
Oracle and SQLite only; for PostgreSQL, use ST_MultiPoint.

## Definition

ST_MPointFromText takes a well-known text (WKT) representation of type ST_MultiPoint and a spatial reference ID and creates an ST_Multipoint.

## Syntax

### Oracle

```
sde.st_mpointfromtext (wkt clob, srid integer)
```

```
sde.st_mpointfromtext (wkt clob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### SQLite

```
st_mpointfromtext (wkt text, srid int32)
```

```
st_mpointfromtext (wkt text)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

ST_MultiPoint

## Example

The multipoint_test table is created with the single ST_MultiPoint mpt1 column.

The INSERT statement inserts a multipoint into the mpt1 column using the ST_MpointFromText function.

### Oracle

```
CREATE TABLE multipoint_test (mpt1 sde.st_geometry);
```

```
INSERT INTO MULTIPOINT_TEST VALUES (
 sde.st_mpointfromtext ('multipoint ((10.01 20.03), (10.52 40.11), (30.29 41.56),
(31.78 10.74))', 4326));
```

## SQLite

```
CREATE TABLE multipoint_test (id integer);

SELECT AddGeometryColumn (
 NULL,
 'multipoint_test',
 'pt1',
 4326,
 'multipoint',
 'xy',
 'null'
);
```

```
INSERT INTO MULTIPOINT_TEST VALUES (
 1,
 st_mpointfromtext ('multipoint ((10.01 20.03), (10.52 40.11), (30.29 41.56), (31.78
10.74))', 4326));
```

# ST_MPointFromWKB

## Definition

ST_MPointFromText takes a well-known binary (WKB) representation of type ST_MultiPoint and a spatial reference ID and creates an ST_MultiPoint.

## Syntax

### Oracle

```
sde.st_mpointfromwkb (wkb blob, srid integer)
```

```
sde.st_mpointfromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### PostgreSQL

```
sde.st_mpointfromwkb (wkb bytea, srid integer)
```

### SQLite

```
st_mpointfromwkb (wkb blob, srid int32)
```

```
st_mpointfromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

ST_MultiPoint

## Example

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 4326. In this example, the multipoint gets stored with ID = 10 in the GEOMETRY column of the SAMPLE_MPOINTS table, 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 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.

In the following SELECT statement, the ST_MPointFromWKB function is used to retrieve the multipoint from the WKB column.

## Oracle

```
CREATE TABLE sample_mpoints (
 id integer,
 geometry sde.st_geometry,
 wkb blob
);

INSERT INTO SAMPLE_MPOINTS (id, geometry) VALUES (
 10,
 sde.st_multipoint ('multipoint ((4 14), (35 16), (24 13))', 4326)
);

UPDATE SAMPLE_MPOINTS
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mpointfromwkb (wkb,4326)) MULTI_POINT
 FROM SAMPLE_MPOINTS
 WHERE id = 10;


ID           MULTI_POINT

10    MULTIPOINT ((4.00000000 14.00000000), (35.00000000 16.00000000), (24.00000000
13.00000000))
```

## PostgreSQL

```
CREATE TABLE sample_mpoints (
 id integer,
 geometry sde.st_geometry,
 wkb bytea
);

INSERT INTO sample_mpoints (id, geometry) VALUES (
 10,
 sde.st_multipoint ('multipoint (4 14, 35 16, 24 13)', 4326)
);

UPDATE sample_mpoints
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mpointfromwkb (wkb,4326))
 AS "MULTI_POINT"
 FROM sample_mpoints
 WHERE id = 10;


id           MULTI_POINT

10     MULTIPOINT (4 14, 35 16, 24 13)
```

## SQLite

```
CREATE TABLE sample_mpoints (
 id integer,
 wkb blob
);

SELECT AddGeometryColumn (
 NULL,
 'sample_mpoints',
 'geometry',
 4326,
 'multipointzm',
 'xyzm',
 'null'
);

INSERT INTO SAMPLE_MPOINTS (id, geometry) VALUES (
 10,
 st_multipoint ('multipoint ((4 14), (35 16), (24 13))', 4326)
);

UPDATE sample_mpoints
 SET wkb = st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id AS "ID",
 st_astext (st_mpointfromwkb (wkb,4326))
 AS "MULTI_POINT"
 FROM sample_mpoints
 WHERE id = 10;


ID           MULTI_POINT

10    MULTIPOINT ((4.00000000 14.00000000), (35.00000000 16.00000000), (24.00000000
13.00000000))
```

# ST_MPolyFromText

**Note:**
Oracle and SQLite only; for PostgreSQL, use ST_MultiPolygon.

## Definition

ST_MPointFromText takes a well-known text (WKT) representation of type ST_MultiPolygon and a spatial reference ID and returns an ST_MultiPolygon.

## Syntax

If you do not specify an SRID, the spatial reference defaults to 4326.

### Oracle

```
sde.st_mpolyfromtext (wkt clob, srid integer)
```

```
sde.st_mpolyfromtext (wkt clob)
```

### SQLite

```
st_mpolyfromtext (wkt text, srid int32)
```

```
st_mpolyfromtext (wkt text)
```

## Return type

ST_MultiPolygon

## Example

The multipolygon_test table is created with an ST_MultiPolygon column, mpl1.

The INSERT statement inserts an ST_MultiPolygon into the mpl1 column using the ST_MpolyFromText function.

### Oracle

```
CREATE TABLE mpolygon_test (mpl1 sde.st_geometry);
```

```
INSERT INTO MPOLYGON_TEST VALUES (
 sde.st_mpolyfromtext ('multipolygon (((10.01 20.03, 10.52 40.11, 30.29 41.56,
31.78 10.74, 10.01 20.03), (21.23 15.74, 21.34 35.21, 28.94 35.35,
29.02 16.83, 21.23 15.74)), ((40.91 10.92, 40.56 20.19, 50.01 21.12,
51.34 9.81, 40.91 10.92)))', 4326)
);
```

## SQLite

```
CREATE TABLE mpolygon_test (id integer);

SELECT AddGeometryColumn(
 NULL,
 'mpolygon_test',
 'mpl1',
 4326,
 'multipolygon',
 'xy',
 'null'
);
```

```
INSERT INTO MPOLYGON_TEST VALUES (
 1,
 st_mpolyfromtext ('multipolygon (((10.01 20.03, 10.52 40.11, 30.29 41.56,
31.78 10.74, 10.01 20.03), (21.23 15.74, 21.34 35.21, 28.94 35.35,
29.02 16.83, 21.23 15.74)), ((40.91 10.92, 40.56 20.19, 50.01 21.12,
51.34 9.81, 40.91 10.92)))', 4326)
);
```

# ST_MPolyFromWKB

## Definition

ST_MPointFromWKB takes a well-known binary (WKB) representation of type ST_MultiPolygon and a spatial reference ID to return an ST_MultiPolygon.

## Syntax

### Oracle

```
sde.st_mpolyfromwkb (wkb blob, srid integer)
```

```
sde.st_mpolyfromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### PostgreSQL

```
sde.st_mpolyfromwkb (wkb bytea, srid integer)
```

### SQLite

```
st_mpolyfromwkb (wkb blob, srid int32)
```

```
st_mpolyfromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

ST_MultiPolygon

## Example

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 4326. In this example, the multipolygon is stored with ID = 10 in the geometry column of the sample_mpolys table, 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 sample_mpolys table has a geometry column, where the multipolygon is stored, and a wkb column, where the multipolygon's WKB representation is stored.

The SELECT statement includes the ST_MPolyFromWKB function, which is used to retrieve the multipolygon from the WKB column.

## Oracle

```
CREATE TABLE sample_mpolys (
 id integer,
 geometry sde.st_geometry,
 wkb blob
);

INSERT INTO SAMPLE_MPOLYS (id, geometry) VALUES (
 10,
 sde.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)))', 4326)
);

UPDATE SAMPLE_MPOLYS
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mpolyfromwkb (wkb,4326)) MULTIPOLYGON
 FROM SAMPLE_MPOLYS
 WHERE id = 10;

ID   MULTIPOLYGON

10   MULTIPOLYGON (((10.00000000 20.00000000, 30.00000000 41.00000000, 10.00000000
40.00000000, 10.00000000 20.00000000)), (1.00000000 72.00000000, 5.00000000
76.00000000, 4.00000000 79.0000000, 1.00000000 72,00000000)), (9.00000000 43.00000000,
6.00000000 47.00000000, 7.00000000 44.00000000, 9.00000000 43.00000000 )))
```

## PostgreSQL

```
CREATE TABLE sample_mpolys (
 id integer,
 geometry sde.st_geometry,
 wkb bytea
);

INSERT INTO sample_mpolys (id, geometry) VALUES (
 10,
 sde.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)))', 4326)
);

UPDATE sample_mpolys
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, sde.st_astext (sde.st_mpolyfromwkb (wkb,4326))
 AS MULTIPOLYGON
 FROM sample_mpolys
 WHERE id = 10;

id   multipolygon

10   MULTIPOLYGON (((10 20, 30 41, 10 40, 10 20)),
 ((1 72, 5 76, 4 79, 1 72)), ((9 43, 6 47, 7 44, 9 43)))
```

## SQLite

```
CREATE TABLE sample_mpolys (
 id integer,
 wkb blob
);

SELECT AddGeometryColumn(
 NULL,
 'sample_mpolys',
 'geometry',
 4326,
 'multipolygon',
 'xy',
 'null'
);

INSERT INTO SAMPLE_MPOLYS (id, geometry) 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)))', 4326)
);

UPDATE SAMPLE_MPOLYS
 SET wkb = st_asbinary (geometry)
 WHERE id = 10;
```

```
SELECT id, st_astext (st_mpolyfromwkb (wkb,4326))
 AS "Multipolygon"
 FROM sample_mpolys
 WHERE id = 10;

id   Multipolygon

10   MULTIPOLYGON ((( 10.00000000 20.00000000, 30.00000000 41.00000000, 10.00000000
40.00000000, 10.00000000 20.00000000)),
 ((1.00000000 72.00000000, 5.00000000 76.00000000, 4.00000000 79.00000000, 1.00000000
72.00000000)),
 ((9.00000000 43.00000000, 6.00000000 47.00000000, 7.00000000 44.00000000, 9.00000000
43.00000000)))
```

# ST_MultiCurve

**Note:**
Oracle only

## Definition

ST_MultiCurve constructs a multicurve feature from a well-known text representation.

## Syntax

```
sde.st_multicurve (wkt clob, srid integer)
```

## Return type

ST_MultiLinestring

## Example

```
CREATE TABLE mcurve_test (id integer, geometry sde.st_geometry);

INSERT INTO MCURVE_TEST VALUES (
1910,
sde.st_multicurve ('multilinestring ((33 2, 34 3, 35 6),
(28 4, 29 5, 31 8, 43 12), (39 3, 37 4, 36 7))', 0)
);

SELECT sde.st_astext (geometry) MCURVE
FROM MCURVE_TEST;

ID        MCURVE

1110      MULTILINESTRING ((33.00000000 2.00000000, 34.00000000
           3.00000000, 35.00000000 6.00000000), (28.00000000 4.00000000,
           29.00000000 5.00000000, 31.00000000 8.0000000, 43.00000000
           12.00000000), (39.00000000 3.00000000, 37.00000000 4.00000000,
           36.00000000 7.00000000 ))
```

# ST_MultiLineString

## Definition

ST_MultiLineString constructs a multilinestring from a well-known text representation.

**Note:**
When creating spatial tables that will be used with ArcGIS, it is best to create the column as the geometry supertype (for example, ST_Geometry) rather than specifying an ST_Geometry subtype.

## Syntax

### Oracle

```
sde.st_multilinestring (wkt clob, srid integer)
```

### PostgreSQL

```
sde.st_multilinestring (wkt clob, srid integer)
sde.st_multilinestring (esri_shape bytea, srid integer)
```

### SQLite

```
st_multilinestring (wkt text, srid int32)
```

## Return type

ST_MultiLineString

## Example

A table, mlines_test, is created, and one multiline is inserted to it using the ST_MultiLineString function.

### Oracle

```
CREATE TABLE mlines_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MLINES_TEST VALUES (
 1910,
 sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6), (28 4, 29 5, 31 8, 43
12), (39 3, 37 4, 36 7))', 4326)
);
```

### PostgreSQL

```
CREATE TABLE mlines_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO mlines_test VALUES (
 1910,
 sde.st_multilinestring ('multilinestring ((33 2, 34 3, 35 6), (28 4, 29 5, 31 8, 43
12), (39 3, 37 4, 36 7))', 4326)
);
```

### SQLite

```
CREATE TABLE mlines_test (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'mlines_test',
 'geometry',
 4326,
 'multilinestring',
 'xy',
 'null'
);

INSERT INTO mlines_test VALUES (
 1910,
 st_multilinestring ('multilinestring ((33 2, 34 3, 35 6), (28 4, 29 5, 31 8, 43 12),
(39 3, 37 4, 36 7))', 4326)
);
```

# ST_MultiPoint

## Definition

ST_MultiPoint constructs a multipoint feature from a well-known text representation.

**Note:**
When creating spatial tables that will be used with ArcGIS, it is best to create the column as the geometry supertype (for example, ST_Geometry) rather than specifying an ST_Geometry subtype.

## Syntax

### Oracle

```
sde.st_multipoint (wkt clob, srid integer)
```

### PostgreSQL

```
sde.st_multipoint (wkt clob, srid integer)
sde.st_multipoint (esri_shape bytea, srid integer)
```

### SQLite

```
st_multipoint (wkt text, srid int32)
```

## Return type

ST_MultiPoint

## Example

A table, mpoint_test, is created, and one multipoint is inserted to it using the ST_MultiPoint function.

### Oracle

```
CREATE TABLE mpoint_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MPOINT_TEST VALUES (
 1110,
 sde.st_multipoint ('multipoint ((1 2), (3 4), (5 6))', 4326)
);
```

### PostgreSQL

```
CREATE TABLE mpoint_test (
 id integer,
```

```
 geometry sde.st_geometry
);

INSERT INTO mpoint_test VALUES (
 1110,
 sde.st_multipoint ('multipoint (1 2, 3 4, 5 6)', 4326)
);
```

## SQLite

```
CREATE TABLE mpoint_test (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'mpoint_test',
 'geometry',
 4326,
 'multipoint',
 'xy',
 'null'
);

INSERT INTO mpoint_test VALUES (
 1110,
 st_multipoint ('multipoint ((1 2), (3 4), (5 6))', 4326)
);
```

# ST_MultiPolygon

## Definition

ST_MultiPolygon constructs a multipolygon feature from a well-known text representation.

**Note:**
When creating spatial tables that will be used with ArcGIS, it is best to create the column as the geometry supertype (for example, ST_Geometry) rather than specifying an ST_Geometry subtype.

## Syntax

### Oracle

```
sde.st_multipolygon (wkt clob, srid integer)
```

### PostgreSQL

```
sde.st_multipolygon (wkt clob, srid integer)
sde.st_multipolygon (esri_shape bytea, srid integer)
```

### SQLite

```
st_multipolygon (wkt text, srid int32)
```

## Return type

ST_MultiPolygon

## Example

A table, mpoly_test, is created, and one multipolygon is inserted to it using the ST_MultiPolygon function.

### Oracle

```
CREATE TABLE mpoly_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO MPOLY_TEST VALUES (
 1110,
 sde.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)))', 4326)
);
```

### PostgreSQL

```
CREATE TABLE mpoly_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO mpoly_test VALUES (
 1110,
 sde.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)))', 4326)
);
```

### SQLite

```
CREATE TABLE mpoly_test (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'mpoly_test',
 'geometry',
 4326,
 'multipolygon',
 'xy',
 'null'
);

INSERT INTO mpoly_test 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)))', 4326)
);
```

# ST_MultiSurface

**Note:**
Oracle only

## Definition

ST_MultiSurface constructs a multisurface feature from a well-known text representation.

## Syntax

```
sde.st_multisurface (wkt clob, srid integer)
```

## Return type

ST_MultiSurface

## Example

```
CREATE TABLE msurf_test (id integer, geometry sde.st_geometry);

INSERT INTO MSURF_TEST VALUES (
1110,
sde.st_multisurface ('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)))', 0)
);

SELECT id, sde.st_astext (geometry) MULTI_SURFACE
FROM MSURF_TEST
WHERE id = 1110;

     ID    MULTI_SURFACE

   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 28.0000000, 8.00000000 24.00000000)),
           ((3.00000000 3.00000000, 5.00000000 3.00000000,
           4.00000000 6.00000000, 3.00000000 3.00000000)))
```

# ST_NumGeometries

## Definition

ST_NumGeometries takes a geometry collection and returns the number of geometries in the collection.

## Syntax

### Oracle

```
sde.st_numgeometries (multipoint1 sde.st_geometry)
sde.st_numgeometries (multiline1 sde.st_geometry)
sde.st_numgeometries (multipolygon1 sde.st_geometry)
```

### PostgreSQL

```
sde.st_numgeometries (geometry1 sde.st_geomcollection)
```

### SQLite

```
st_numgeometries (geometry1 geometryblob)
```

## Return type

Integer

## Example

In the following example, a table named sample_numgeom is created. One multipolygon and one multipoint are inserted to it. In the SELECT statement, the ST_NumGeometries function is used to determine the number of geometries (or features) in each geometry.

### Oracle

```
CREATE TABLE sample_numgeom (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO SAMPLE_NUMGEOM VALUES (
 1,
 sde.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)))', 4326)
);

INSERT INTO SAMPLE_NUMGEOM VALUES (
 2,
 sde.st_multipoint ('multipoint ((1 2), (4 3), (5 6), (7 6), (8 8))', 4326)
);

SELECT id, sde.st_numgeometries (geometry) NUM_GEOMS_IN_COLL
 FROM SAMPLE_NUMGEOM;
```

```
ID          NUM_GEOMS_IN_COLL

  1                 3
  2                 5
```

## PostgreSQL

```
CREATE TABLE sample_numgeom (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO sample_numgeom VALUES (
 1,
 sde.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)))', 4326)
);

INSERT INTO sample_numgeom VALUES (
 2,
 sde.st_multipoint ('multipoint (1 2, 4 3, 5 6, 7 6, 8 8)', 4326)
);

SELECT id, sde.st_numgeometries (geometry)
 AS "number of geometries"
 FROM sample_numgeom;

id          number of geometries

  1                 3
  2                 5
```

## SQLite

```
CREATE TABLE sample_numgeom (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'sample_numgeom',
 'geometry',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO sample_numgeom 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)))', 4326)
);

INSERT INTO sample_numgeom VALUES (
 2,
 st_multipoint ('multipoint ((1 2), (4 3), (5 6), (7 6), (8 8))', 4326)
);
```

```
SELECT id, st_numgeometries (geometry)
 AS "number of geometries"
 FROM sample_numgeom;

id          number of geometries

  1                 3
  2                 5
```

# ST_NumInteriorRing

## Definition

ST_NumInteriorRing takes an ST_Polygon and returns the number of its interior rings.

## Syntax

### Oracle and PostgreSQL

```
sde.st_numinteriorring (polygon1 sde.st_geometry)
```

### SQLite

```
st_numinteriorring (polygon1 geometryblob)
```

## Return type

Integer

## Example

An ornithologist wants to study a bird population on several south sea islands. She wants to identify which islands contain one or more lakes, because the bird species she is interested in feeds only in freshwater lakes.

The ID and name columns of the islands table identify each island, while the land ST_Polygon column stores the islands' geometry.

Because interior rings represent the lakes, the SELECT statement that includes the ST_NumInteriorRing function lists only those islands that have at least one interior ring.

## Oracle

```
CREATE TABLE islands (
 id integer,
 name varchar(32),
 land sde.st_geometry
);

INSERT INTO islands VALUES (
 1,
 'Bear',
 sde.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))', 4326)
);

INSERT INTO islands VALUES (
 2,
 'Johnson',
 sde.st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
SELECT name
 FROM ISLANDS
 WHERE sde.st_numinteriorring (land)> 0;

NAME

Bear
```

## PostgreSQL

```
CREATE TABLE islands (
 id integer,
 name varchar(32),
 land sde.st_geometry
);

INSERT INTO islands VALUES (
 1,
 'Bear',
 sde.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))', 4326)
);

INSERT INTO islands VALUES (
 2,
 'Johnson',
 sde.st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
SELECT name
 FROM islands
 WHERE sde.st_numinteriorring (land)> 0;

name

Bear
```

## SQLite

```
CREATE TABLE islands (
 id integer,
 name varchar(32)
);

SELECT AddGeometryColumn(
 NULL,
 'islands',
 'land',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO islands VALUES (
 1,
 'Bear',
 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))', 4326)
);

INSERT INTO islands VALUES (
 2,
 'Johnson',
 st_polygon ('polygon ((10 10, 50 10, 10 30, 10 10))', 4326)
);
```

```
SELECT name
 FROM islands
 WHERE st_numinteriorring (land)> 0;

name

Bear
```

# ST_NumPoints

## Definition

ST_NumPoints returns the number of points (vertices) in a geometry.

For polygons, both the starting and ending vertices are counted, even though they occupy the same location.

Note that this number is different than the NUMPTS attribute of the ST_Geometry type. The NUMPTS attribute contains a count of the vertices in all the parts of the geometry including separators that occur between parts. There is one separator between each part. For example, a multipart linestring with three parts has two separators. In the NUMPTS attribute, each separator is counted as one vertex. Conversely, the ST_NumPoints function does not include the separators in the vertex count.

## Syntax

### Oracle and PostgreSQL

```
sde.st_numpoints (geometry1 sde.st_geometry)
```

### SQLite

```
st_numpoints (geometry1 geometryblob)
```

## Return type

Integer

## Example

The numpoints_test table is created with the geotype column, which contains the geometry type stored in the g1 column.

The INSERT statements insert a point, a linestring, and a polygon.

The SELECT query uses the ST_NumPoints function to get the number of points in each feature for each feature type.

### Oracle

```
CREATE TABLE numpoints_test (
 geotype varchar(12),
 g1 sde.st_geometry
);

INSERT INTO NUMPOINTS_TEST VALUES (
 'point',
 sde.st_pointfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO NUMPOINTS_TEST VALUES (
 'linestring',
 sde.st_linefromtext ('linestring (10.02 20.01, 23.73 21.92)', 4326)
);
```

```
INSERT INTO NUMPOINTS_TEST VALUES (
 'polygon',
 sde.st_polyfromtext ('polygon ((10.02 20.01, 23.73 21.92, 24.51 12.98, 11.64 13.42,
10.02 20.01))', 4326)
);
```

```
SELECT geotype, sde.st_numpoints (g1) Number_of_points
 FROM NUMPOINTS_TEST;

GEOTYPE     Number_of_points

point                       1
linestring                  2
polygon                     5
```

## PostgreSQL

```
CREATE TABLE numpoints_test (
 geotype varchar(12),
 g1 sde.st_geometry
);

INSERT INTO numpoints_test VALUES (
 'point',
 sde.st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO numpoints_test VALUES (
 'linestring',
 sde.st_linestring ('linestring (10.02 20.01, 23.73 21.92)', 4326)
);

INSERT INTO numpoints_test VALUES (
 'polygon',
 sde.st_polygon ('polygon ((10.02 20.01, 23.73 21.92, 24.51 12.98, 11.64 13.42, 10.02
20.01))', 4326)
);
```

```
SELECT geotype, sde.st_numpoints (g1)
 AS Number_of_points
 FROM numpoints_test;

geotype     number_of_points

point                       1
linestring                  2
polygon                     5
```

## SQLite

```
CREATE TABLE numpoints_test (
 geotype text(12)
);
```

```
SELECT AddGeometryColumn(
 NULL,
 'numpoints_test',
 'g1',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO numpoints_test VALUES (
 'point',
 st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO numpoints_test VALUES (
 'linestring',
 st_linestring ('linestring (10.02 20.01, 23.73 21.92)', 4326)
);

INSERT INTO numpoints_test VALUES (
 'polygon',
 st_polygon ('polygon ((10.02 20.01, 23.73 21.92, 24.51 12.98, 11.64 13.42, 10.02
20.01))', 4326)
);
```

```
SELECT geotype AS "Type of geometry", st_numpoints (g1) AS "Number of points"
 FROM numpoints_test;

Type of geometry     Number of points

point                       1
linestring                  2
polygon                     5
```

# ST_OrderingEquals

**Note:**
Oracle and PostgreSQL only

## Definition

ST_OrderingEquals compares two ST_Geometries and returns 1 (Oracle) or t (PostgreSQL) if the geometries are equal and the coordinates are in the same order; otherwise, it returns 0 (Oracle) or f (PostgreSQL).

## Syntax

```
sde.st_orderingequals (g1 sde.st_geometry, g2 sde.st_geometry)
```

## Return type

Boolean

## Example

### Oracle

The following `CREATE TABLE` statement creates the LINESTRING_TEST table, which has two linestring columns, ln1 and ln2.

```
CREATE TABLE linestring_test (
lid integer,
ln1 sde.st_geometry,
ln2 sde.st_geometry);
```

The following `INSERT` statement inserts two ST_LineString values into ln1 and ln2 that are equal and have the same coordinate ordering.

```
INSERT INTO LINESTRING_TEST VALUES (
1,
sde.st_geometry ('linestring (10.01 20.02, 21.50 12.10)', 0),
sde.st_geometry ('linestring (21.50 12.10, 10.01 20.02)', 0)
);
```

The following `SELECT` statement and corresponding result set shows how the ST_Equals function returns 1 (true) regardless of the order of the coordinates. The ST_OrderingEquals function returns 0 (false) if the geometries are not both equal and have the same coordinate ordering.

```
SELECT lid, sde.st_equals (ln1, ln2) Equals, sde.st_orderingequals (ln1, ln2)
OrderingEquals
FROM LINESTRING_TEST;

lid Equals     OrderingEquals

1   1          0
```

## PostgreSQL

The following `CREATE TABLE` statement creates the LINESTRING_TEST table, which has two linestring columns, ln1 and ln2.

```
CREATE TABLE linestring_test (
lid integer,
ln1 sde.st_geometry,
ln2 sde.st_geometry);
```

The following `INSERT` statement inserts two ST_LineString values into ln1 and ln2 that are equal and have the same coordinate ordering.

```
INSERT INTO linestring_test VALUES (
1,
sde.st_linestring ('linestring (10.01 20.02, 21.50 12.10)', 0),
sde.st_linestring ('linestring (21.50 12.10, 10.01 20.02)', 0)
);
```

The following `SELECT` statement and corresponding result set shows how the ST_Equals function returns t (true) regardless of the order of the coordinates. The ST_OrderingEquals function returns f (false) if the geometries are not both equal and have the same coordinate ordering.

```
SELECT lid, sde.st_equals (ln1, ln2) AS Equals, sde.st_orderingequals (ln1, ln2)
AS OrderingEquals
FROM linestring_test;

lid equals     orderingequals

1   t          f
```

# ST_Overlaps

## Definition

ST_Overlaps takes two geometry objects and returns 1 (Oracle and SQLite) or t (PostgreSQL) if the intersection of the objects results in a geometry object of the same dimension but not equal to either source object; otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_overlaps (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_overlaps (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

Boolean

## Example

The county supervisor needs a list of sensitive areas that overlap the buffered radius of hazardous waste sites. The sensitive_areas table contains several columns that describe the threatened institutions in addition to the shape column, which stores the institutions' ST_Polygon geometries.

The hazardous_sites table stores the identity of the sites in the id column, while the actual geographic location of each site is stored in the site point column.

The sensitive_areas and hazardous_sites tables are joined by the ST_Overlaps function, which returns the ID for all sensitive_areas rows that contain polygons that overlap the buffered radius of the hazardous_sites points.

### Oracle

```
CREATE TABLE sensitive_areas (
 id integer,
 shape sde.st_geometry
);

CREATE TABLE hazardous_sites (
 id integer,
 site sde.st_geometry
);


INSERT INTO sensitive_areas VALUES (
 1,
 sde.st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);

INSERT INTO sensitive_areas VALUES (
 2,
```

```
 sde.st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);

INSERT INTO sensitive_areas VALUES (
 3,
 sde.st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);

INSERT INTO hazardous_sites VALUES (
 4,
 sde.st_geometry ('point (.60 .60)', 4326)
);

INSERT INTO hazardous_sites VALUES (
 5,
 sde.st_geometry ('point (.30 .30)', 4326)
);
```

```
SELECT UNIQUE (hs.id)
 FROM HAZARDOUS_SITES hs, SENSITIVE_AREAS sa
 WHERE sde.st_overlaps (sde.st_buffer (hs.site, .001), sa.shape) = 1;

ID

4
5
```

## PostgreSQL

```
CREATE TABLE sensitive_areas (
 id serial,
 shape sde.st_geometry
);

CREATE TABLE hazardous_sites (
 id serial,
 site sde.st_geometry
);


INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 sde.st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);

INSERT INTO hazardous_sites (site) VALUES (
 sde.st_geometry ('point (.60 .60)', 4326)
);

INSERT INTO hazardous_sites (site) VALUES (
 sde.st_geometry ('point (.30 .30)', 4326)
);
```

```
SELECT DISTINCT (hs.id) AS "Hazardous Site ID"
 FROM hazardous_sites hs, sensitive_areas sa
 WHERE sde.st_overlaps (sde.st_buffer (hs.site, .001), sa.shape) = 't';

id

1
2
```

## SQLite

```
CREATE TABLE sensitive_areas (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'sensitive_areas',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE hazardous_sites (
```

```
 id integer primary key autoincrement not null,
 site_name varchar(30)
);

SELECT AddGeometryColumn(
 NULL,
 'hazardous_sites',
 'site',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((.20 .30, .30 .30, .30 .40, .20 .40, .20 .30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((.30 .30, .30 .50, .50 .50, .50 .30, .30 .30))', 4326)
);

INSERT INTO sensitive_areas (shape) VALUES (
 st_geometry ('polygon ((.40 .40, .40 .60, .60 .60, .60 .40, .40 .40))', 4326)
);

INSERT INTO hazardous_sites (site_name, site) VALUES (
 'Kemlabs',
 st_geometry ('point (.60 .60)', 4326)
);

INSERT INTO hazardous_sites (site_name, site) VALUES (
 'Medi-Waste',
 st_geometry ('point (.30 .30)', 4326)
);
```

```
SELECT DISTINCT (hs.site_name) AS "Hazardous Site"
 FROM hazardous_sites hs, sensitive_areas sa
 WHERE st_overlaps (st_buffer (hs.site, .001), sa.shape) = 1;

Hazardous Site

Kemlabs
Medi-Waste
```

# ST_Perimeter

## Definition

ST_Perimeter returns the length of the continuous line that forms the boundary of a closed polygon or multipolygon feature.

This function is new in 10.8.1.

## Syntax

The first two options in each section return the perimeter in the units of the coordinate system defined for the feature. The second two options allow you to specify the linear unit of measure. For a list of supported values for `linear_unit_name`, see ST_Distance.

### Oracle and PostgreSQL

```
sde.st_perimeter (polygon sde.st_geometry)

sde.st_perimeter (multipolygon sde.st_geometry)

sde.st_perimeter (polygon sde.st_geometry, linear_unit_name text)

sde.st_perimeter (multipolygon sde.st_geometry, linear_unit_name text)
```

### SQLite

```
st_perimeter (polygon sde.st_geometry)

st_perimeter (multipolygon sde.st_geometry)

st_perimeter (polygon sde.st_geometry, linear_unit_name text)

st_perimeter (multipolygon sde.st_geometry, linear_unit_name text)
```

## Return type

Double precision

## Examples

### Oracle

In the following example, an ecologist studying shoreline birds needs to determine the length of the shoreline for the lakes in a particular area. The lakes are stored as polygons in the waterbodies table. A `SELECT` statement using the ST_Perimeter function is used to return the perimeter of each lake (feature) in the waterbodies table.

```
--Create table named waterbodies
CREATE TABLE waterbodies (wbid INTEGER not null, waterbody sde.st_geometry);

--Insert a polygon feature to the waterbodies table
INSERT INTO waterbodies VALUES (
  1,
  sde.ST_Polygon ('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))', 1)
);
```

```
--Find the perimeter of the polygon
SELECT sde.ST_Perimeter (waterbody)
  FROM waterbodies;
```

The SELECT statement returns the following:

```
ID PERIMETER
1 +1.8000000
```

In the next example, you will create a table named bfp, insert three features, and calculate the perimeter of each feature in units of linear measure:

```
--Create table named bfp
CREATE TABLE bfp (
 building_id integer not null,
 footprint sde.st_geometry);

--Insert polygon features to the bfp table
INSERT INTO BFP (building_id, footprint) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO BFP (building_id, footprint) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO BFP (building_id, footprint) VALUES (
 3,
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);

--Find the perimeter of each polygon
SELECT sde.ST_Perimeter(footprint)
            ,sde.ST_Perimeter(footprint, 'meter') as Meter
            ,sde.ST_Perimeter(footprint, 'km') as KM
            ,sde.ST_Perimeter(footprint, 'yard') As Yard
FROM bfp;
```

The SELECT statement returns the perimeter of each feature in three units:

```
st_perimeter    |       meter       |         km         |        yard
------------------+-------------------+--------------------+--------------------
 40.00000000000001 | 4421256.128972424 |  4421.256128972425 |   4835144.49800134
  64.7213595499958 | 7159231.951087892 | 7159.2319510878915 |  7829431.267593933
 24.14213562373095 | 2417672.365575198 |  2417.672365575198 | 2643998.6500166208
```

## PostgreSQL

In the following example, an ecologist studying shoreline birds needs to determine the length of the shoreline for the lakes in a particular area. The lakes are stored as polygons in the waterbodies table. A SELECT statement using the ST_Perimeter function is used to return the perimeter of each lake (feature) in the waterbodies table.

```
--Create table named waterbodies
CREATE TABLE waterbodies (wbid INTEGER not null, waterbody sde.st_geometry);

--Insert a polygon feature to the waterbodies table
INSERT INTO waterbodies VALUES (
  1,
  sde.ST_Polygon ('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))', 1)
);

--Find the perimeter of the polygon
SELECT sde.ST_Perimeter (waterbody)
  FROM waterbodies;
```

The SELECT statement returns the following:

```
ID PERIMETER
1 +1.8000000
```

In the next example, you will create a table named bfp, insert three features, and calculate the perimeter of each feature in units of linear measure:

```
--Create table named bfp
CREATE TABLE bfp (
 building_id serial,
 footprint sde.st_geometry);

--Insert polygon features to the bfp table
INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 sde.st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);

--Find the perimeter of each polygon
SELECT sde.st_perimeter(footprint)
            ,sde.st_perimeter(footprint, 'meter') as Meter
            ,sde.st_perimeter(footprint, 'km') as KM
            ,sde.st_perimeter(footprint, 'yard') As Yard
FROM bfp;
```

The SELECT statement returns the perimeter of each feature in three units:

```
 st_perimeter    |       meter       |         km         |        yard
-----------------+-------------------+--------------------+--------------------
 40.00000000000001 | 4421256.128972424 |  4421.256128972425 |   4835144.49800134
  64.7213595499958 | 7159231.951087892 | 7159.2319510878915 |  7829431.267593933
 24.14213562373095 | 2417672.365575198 |  2417.672365575198 | 2643998.6500166208
```

## SQLite

In the following example, an ecologist studying shoreline birds needs to determine the length of the shoreline for the lakes in a particular area. The lakes are stored as polygons in the waterbodies table. A SELECT statement using the ST_Perimeter function is used to return the perimeter of each lake (feature) in the waterbodies table.

```
--Create table named waterbodies and add a spatial column (waterbody) to it
CREATE TABLE waterbodies (wbid integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'waterbodies',
 'waterbody',
 4326,
 'polygon',
 'xy',
 'null'
);

--Insert a polygon feature to the waterbodies table
INSERT INTO waterbodies VALUES (
  1,
  ST_Polygon ('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))', 1)
);

--Find the perimeter of the polygon
SELECT ST_Perimeter (waterbody)
  FROM waterbodies;
```

The SELECT statement returns the following:

```
ID PERIMETER
1 +1.8000000
```

In the next example, you will create a table named bfp, insert three features, and calculate the perimeter of each feature in units of linear measure:

```
--Create table named bfp and add a spatial column (footprints) to it
CREATE TABLE bfp (
 building_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'bfp',
 'footprint',
```

```
 4326,
 'polygon',
 'xy',
 'null'
);

--Insert polygon features to the bfp table
INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 0, 30 20, 40 0, 20 0))', 4326)
);

INSERT INTO bfp (footprint) VALUES (
 st_polygon ('polygon ((20 30, 25 35, 30 30, 20 30))', 4326)
);

--Find the perimeter of each polygon
SELECT ST_Perimeter(footprint)
            ,ST_Perimeter(footprint, 'meter') as Meter
            ,ST_Perimeter(footprint, 'km') as KM
            ,ST_Perimeter(footprint, 'yard') As Yard
FROM bfp;
```

The SELECT statement returns the perimeter of each feature in three units:

```
 st_perimeter    |       meter       |         km         |        yard
-------------------+-------------------+--------------------+--------------------
 40.00000000000001 | 4421256.128972424 |  4421.256128972425 |   4835144.49800134
  64.7213595499958 | 7159231.951087892 | 7159.2319510878915 |  7829431.267593933
 24.14213562373095 | 2417672.365575198 |  2417.672365575198 | 2643998.6500166208
```

# ST_Point

## Definition

ST_Point takes a well-known text object or coordinates and an spatial reference ID and returns an ST_Point.

**Note:**
When creating spatial tables that will be used with ArcGIS, it is best to create the column as the geometry supertype (for example, ST_Geometry) rather than specifying an ST_Geometry subtype.

## Syntax

### Oracle

```
sde.st_point (wkt clob, srid integer)
sde.st_point (x number, y number, srid integer)
sde.st_point (x number, y number, m number, srid integer)
sde.st_point (x number, y number, z number, srid integer)
sde.st_point (x number, y number, z number, m number, srid integer)
```

### PostgreSQL

```
sde.st_point (wkt clob, srid integer)
sde.st_point (esri_shape bytea, srid integer)sde.
sde.st_point (x double precision, y double precision, srid integer)
sde.st_point (x double precision, y double precision, m double precision, srid integer)
sde.st_point (x double precision, y double precision, z double precision, srid integer)
sde.st_point (x double precision, y double precision, z double precision, m double
precision, srid integer)
```

### SQLite

```
st_point (wkt text, srid int32)
st_point (x float64, y float64, srid int32)
st_point (x float64, y float64, z float64, m float64, srid int32)
```

## Return type

ST_Point

## Example

The following `CREATE TABLE` statement creates the point_test table, which has a single point column, PT1.

The ST_Point function converts the point coordinates into an ST_Point geometry before it is inserted into the pt1 column.

### Oracle

```
CREATE TABLE point_test (pt1 sde.st_geometry);
```

```
INSERT INTO point_test VALUES (
 sde.st_point (10.01, 20.03, 4326)
);
```

### PostgreSQL

```
CREATE TABLE point_test (pt1 sde.st_geometry);
```

```
INSERT INTO point_test VALUES (
 sde.st_point (10.01, 20.03, 4326)
);
```

### SQLite

```
CREATE TABLE point_test (id integer);

SELECT AddGeometryColumn(
 NULL,
 'point_test',
 'pt1',
 4326,
 'point',
 'xy',
 'null'
);
```

```
INSERT INTO point_test VALUES (
 1,
 st_point (10.01, 20.03, 4326)
);
```

# ST_PointFromText

**Note:**
Used in Oracle and SQLite only; for PostgreSQL, use ST_Point.

## Definition

ST_PointFromText takes a well-known text representation of type point and a spatial reference ID and returns a point.

## Syntax

### Oracle

```
sde.st_pointfromtext (wkt varchar2, srid integer)
```

```
sde.st_pointfromtext (wkt varchar2)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### SQLite

```
st_pointfromtext (wkt text, srid int32)
```

```
st_pointfromtext (wkt text)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

ST_Point

## Example

The point_test table is created with the single ST_Point column pt1.

The ST_PointFromText function converts the point text coordinates to the point format before the INSERT statement inserts the point into the pt1 column.

### Oracle

```
CREATE TABLE point_test (pt1 sde.st_geometry);
```

```
INSERT INTO POINT_TEST VALUES (
 sde.st_pointfromtext ('point (10.01 20.03)', 4326)
);
```

## SQLite

```
CREATE TABLE pt_test (id integer);

SELECT AddGeometryColumn(
 NULL,
 'pt_test',
 'pt1',
 4326,
 'point',
 'xy',
 'null'
);
```

```
INSERT INTO pt_test VALUES (
 1,
 st_pointfromtext ('point (10.01 20.03)', 4326)
);
```

# ST_PointFromWKB

## Definition

ST_PointFromWKB takes a well-known binary (WKB) representation and a spatial reference ID to return an ST_Point.

## Syntax

### Oracle

```
sde.st_pointfromwkb (wkb blob, srid integer)
```

```
sde.st_pointfromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### PostgreSQL

```
sde.st_pointfromwkb (wkb bytea, srid integer)
```

### SQLite

```
st_pointfromwkb (wkb blob, srid int32)
```

```
st_pointfromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

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 4326. In this example, the points are stored in the geometry column of the sample_points table, 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.

In the SELECT statement, the ST_PointFromWKB function is used to retrieve the points from the WKB column.

### Oracle

```
CREATE TABLE sample_points (
 id integer,
```

```
 geometry sde.st_point,
 wkb blob
);

INSERT INTO SAMPLE_POINTS (id, geometry) VALUES (
 10,
 sde.st_point ('point (44 14)', 4326)
);

INSERT INTO SAMPLE_POINTS (id, geometry) VALUES (
 11,
 sde.st_point ('point (24 13)', 4326)
);

UPDATE SAMPLE_POINTS
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;

UPDATE SAMPLE_POINTS
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 11;
```

```
SELECT id, sde.st_astext (sde.st_pointfromwkb(wkb, 4326)) POINTS
 FROM SAMPLE_POINTS;

ID POINTS

10 POINT (44.00000000 14.00000000)
11 POINT (24.00000000 13.00000000)
```

## PostgreSQL

```
CREATE TABLE sample_points (
 id integer,
 geometry sde.st_point,
 wkb bytea
);

INSERT INTO sample_points (id, geometry) VALUES (
 10,
 sde.st_point ('point (44 14)', 4326)
);

INSERT INTO sample_points (id, geometry) VALUES (
 11,
 sde.st_point ('point (24 13)', 4326)
);

UPDATE sample_points
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 10;

UPDATE sample_points
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 11;
```

```
SELECT id, sde.st_astext (sde.st_pointfromwkb(wkb, 4326))
 AS points
 FROM sample_points;

id points

10 POINT (44 14)
11 POINT (24 13)
```

## SQLite

```
CREATE TABLE sample_pts (
 id integer,
 wkb blob
);

SELECT AddGeometryColumn(
 NULL,
 'sample_pts',
 'geometry',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO sample_pts (id, geometry) VALUES (
 10,
 st_point ('point (44 14)', 4326)
);

INSERT INTO sample_pts (id, geometry) VALUES (
 11,
```

```
 st_point ('point (24 13)', 4326)
);

UPDATE sample_pts
 SET wkb = st_asbinary (geometry)
 WHERE id = 10;

UPDATE sample_pts
 SET wkb = st_asbinary (geometry)
 WHERE id = 11;
```

```
SELECT id, st_astext (st_pointfromwkb(wkb, 4326))
 AS "points"
 FROM sample_pts;

id points

10 POINT (44.00000000 14.00000000)
11 POINT (24.00000000 13.00000000)
```

# ST_PointN

## Definition

ST_PointN takes an ST_LineString and an integer index and returns a point that is the nth vertex in the ST_LineString's path.

## Syntax

### Oracle and PostgreSQL

```
sde.st_pointn (line1 sde.st_linestring, index integer)
```

### SQLite

```
st_pointn (line1 st_linestring, index int32)
```

## Return type

ST_Point

## Example

The pointn_test table is created with the gid column, which uniquely identifies each row, and the ln1 ST_LineString column. The INSERT statements insert two linestring values. The first linestring doesn't have z-coordinates or measures, while the second linestring has both.

The SELECT query uses the ST_PointN and ST_AsText functions to return the well-known text for the second vertex of each linestring.

### Oracle

```
CREATE TABLE pointn_test (
 gid integer,
 ln1 sde.st_geometry
);


INSERT INTO POINTN_TEST VALUES (
 1,
 sde.st_linefromtext ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO POINTN_TEST VALUES (
 2,
 sde.st_linefromtext ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1, 30.10
40.23 6.9 7.2)', 4326)
);
```

```
SELECT gid, sde.st_astext (sde.st_pointn (ln1, 2)) The_2ndvertex
 FROM POINTN_TEST;
```

```
GID The_2ndvertex

1   POINT (23.73 21.92)
2   POINT ZM (23.73 21.92 6.5 7.1)
```

## PostgreSQL

```
CREATE TABLE pointn_test (
 gid serial,
 ln1 sde.st_geometry
);


INSERT INTO pointn_test (ln1) VALUES (
 sde.st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO pointn_test (ln1) VALUES (
 sde.st_linestring ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1, 30.10
40.23 6.9 7.2)', 4326)
);
```

```
SELECT gid, sde.st_astext (sde.st_pointn (ln1, 2))
 AS The_2ndvertex
 FROM pointn_test;

gid the_2ndvertex

1   POINT (23.73 21.92)
2   POINT ZM (23.73 21.92 6.5 7.1)
```

## SQLite

```
CREATE TABLE pointn_test (
 gid integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'pointn_test',
 'ln1',
 4326,
 'linestringzm',
 'xyzm',
 'null'
);

INSERT INTO pointn_test (ln1) VALUES (
 st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO pointn_test (ln1) VALUES (
 st_linestring ('linestring zm(10.02 20.01 5.0 7.0, 23.73 21.92 6.5 7.1, 30.10 40.23
6.9 7.2)', 4326)
);
```

```
SELECT gid, st_astext (st_pointn (ln1, 2))
 AS "Second Vertex"
 FROM pointn_test;

gid  Second Vertex

1   POINT ( 23.73000000 21.92000000)
2   POINT ZM ( 23.73000000 21.92000000 6.50000000 7.10000000)
```

# ST_PointOnSurface

## Definition

ST_PointOnSurface takes an ST_Polygon or ST_MultiPolygon and returns an ST_Point guaranteed to lie on its surface.

## Syntax

### Oracle and PostgreSQL

```
sde.st_pointonsurface (polygon1 sde.st_geometry)
sde.st_pointonsurface (multipolygon1 sde.st_geometry)
```

### SQLite

```
st_pointonsurface (polygon1 geometryblob)
st_pointonsurface (multipolygon1 geometryblob)
```

## Return type

ST_Point

## Example

The city engineer wants to create a label point for each historical building's footprint. The historical building footprints are stored in the hbuildings table that was created with the following CREATE TABLE statement:

The ST_PointOnSurface function generates a point that is guaranteed to be on the surface of the building footprints. The ST_PointOnSurface function returns a point that the ST_AsText function converts to text for use by the application.

## Oracle

```
CREATE TABLE hbuildings (
 hbld_id integer,
 hbld_name varchar(40),
 footprint sde.st_geometry
);
```

```
INSERT INTO hbuildings (hbld_id, hbld_name, footprint) VALUES (
 1,
 'First National Bank',
 sde.st_polygon ('polygon ((0 0, 0 .010, .010 .010, .010 0, 0 0))', 4326)
);

INSERT INTO hbuildings (hbld_id, hbld_name, footprint) VALUES (
 2,
 'Courthouse',
 sde.st_polygon ('polygon ((.020 0, .020 .010, .030 .010, .030 0, .020 0))', 4326)
);
```

```
SELECT sde.st_astext (sde.st_pointonsurface (footprint)) Historic_Site
 FROM HBUILDINGS;

HISTORIC_SITE

POINT  (0.00500000 0.00500000)
POINT  (0.02500000 0.00500000)
```

## PostgreSQL

```
CREATE TABLE hbuildings (
 hbld_id serial,
 hbld_name varchar(40),
 footprint sde.st_geometry
);
```

```
INSERT INTO hbuildings (hbld_name, footprint) VALUES (
 'First National Bank',
 sde.st_polygon ('polygon ((0 0, 0 .010, .010 .010, .010 0, 0 0))', 4326)
);

INSERT INTO hbuildings (hbld_name, footprint) VALUES (
 'Courthouse',
 sde.st_polygon ('polygon ((.020 0, .020 .010, .030 .010, .030 0, .020 0))', 4326)
);
```

```
SELECT sde.st_astext (sde.st_pointonsurface (footprint))
 AS "Historic Site"
 FROM hbuildings;

Historic Site
```

```
POINT  (0.00500000 0.00500000)
POINT  (0.02500000 0.00500000)
```

## SQLite

```
CREATE TABLE hbuildings (
 hbld_id integer primary key autoincrement not null,
 hbld_name text(40)
);

SELECT AddGeometryColumn(
 NULL,
 'hbuildings',
 'footprint',
 4326,
 'polygon',
 'xy',
 'null'
);
```

```
INSERT INTO hbuildings (hbld_name, footprint) VALUES (
 'First National Bank',
 st_polygon ('polygon ((0 0, 0 .010, .010 .010, .010 0, 0 0))', 4326)
);

INSERT INTO hbuildings (hbld_name, footprint) VALUES (
 'Courthouse',
 st_polygon ('polygon ((.020 0, .020 .010, .030 .010, .030 0, .020 0))', 4326)
);
```

```
SELECT st_astext (st_pointonsurface (footprint))
 AS "Historic Site"
 FROM hbuildings;

Historic Site

POINT  (0.00500000 0.00500000)
POINT  (0.02500000 0.00500000)
```

# ST_PolyFromText

**Note:**
Oracle and SQLite only

## Definition

ST_PolyFromText takes a well-known text representation and a spatial reference ID and returns an ST_Polygon.

## Syntax

### Oracle

```
sde.st_polyfromtext (wkt clob, srid integer)
```

```
sde.st_polyfromtext (wkt clob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### SQLite

```
st_polyfromtext (wkt text, srid int32)
```

```
st_polyfromtext (wkt text)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

ST_Polygon

## Example

The polygon_test table is created with the single polygon column.

The INSERT statement inserts a polygon into the polygon column using the ST_PolyFromText function.

### Oracle

```
CREATE TABLE polygon_test (pl1 sde.st_geometry);
```

```
INSERT INTO polygon_test VALUES (
 sde.st_polyfromtext ('polygon ((10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74,
10.01 20.03))', 4326)
);
```

## SQLite

```
CREATE TABLE polygon_test (id integer);

SELECT AddGeometryColumn(
 NULL,
 'polygon_test',
 'pl1',
 4326,
 'polygon',
 'xy',
 'null'
);
```

```
INSERT INTO polygon_test VALUES (
 1,
 st_polyfromtext ('polygon ((10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74, 10.01
20.03))', 4326)
);
```

# ST_PolyFromWKB

## Definition

ST_PolyFromWKB takes a well-known binary (WKB) representation and a spatial reference ID and returns an ST_Polygon.

## Syntax

### Oracle

```
sde.st_polyfromwkb (wkb blob, srid integer)
```

```
sde.st_polyfromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

### PostgreSQL

```
sde.st_polyfromwkb (wkb bytea, srid integer)
```

### SQLite

```
st_polyfromwkb (wkb blob, srid int32)
```

```
st_polyfromwkb (wkb blob)
```

If you do not specify an SRID, the spatial reference defaults to 4326.

## Return type

ST_Polygon

## Example

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 4326. In this example, the polygon is stored with ID = 1115 in the geometry column of the sample_polys table, and then the wkb column is updated with its WKB representation (using the ST_AsBinary function). Finally, the ST_PolyFromWKB function is used to return the multipolygon from the WKB column. The sample_polys table has a geometry column, where the polygon is stored, and a wkb column, where the polygon's WKB representation is stored.

In the SELECT statement, the ST_PointFromWKB function is used to retrieve the points from the WKB column.

## Oracle

```
CREATE TABLE sample_polys (
 id integer,
 geometry sde.st_geometry,
 wkb blob
);

INSERT INTO SAMPLE_POLYS (id, geometry) VALUES (
 1115,
 sde.st_polyfromtext ('polygon ((10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74,
10.01 20.03))', 4326)
);

UPDATE SAMPLE_POLYS
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1115;
```

```
SELECT id, sde.st_astext (sde.st_polyfromwkb (wkb, 4326)) POLYS
 FROM SAMPLE_POLYS;

ID            POLYS

1115    POLYGON (10.01000000 20.03000000, 31.78000000 10.74000000, 30.29000000
41.56000000, 10.52000000 40.11000000, 10.01000000 20.03000000)
```

## PostgreSQL

```
CREATE TABLE sample_polys (
 id integer,
 geometry sde.st_geometry,
 wkb bytea
);

INSERT INTO sample_polys (id, geometry) VALUES (
 1115,
 sde.st_polygon ('polygon  ((10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74, 10.01
20.03))', 4326)
);

UPDATE sample_polys
 SET wkb = sde.st_asbinary (geometry)
 WHERE id = 1115;
```

```
SELECT id, sde.st_astext (sde.st_polyfromwkb (wkb, 4326))
 AS POLYS
 FROM sample_polys;

id    polys

1115    POLYGON (10.01000000 20.03000000, 31.78000000 10.74000000, 30.29000000
41.56000000, 10.52000000 40.11000000, 10.01000000 20.03000000)
```

## SQLite

```
CREATE TABLE sample_polys(
 id integer,
 wkb blob
);

SELECT AddGeometryColumn(
 NULL,
 'sample_polys',
 'geometry',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO sample_polys (id, geometry) VALUES (
 1115,
 st_polyfromtext ('polygon ((10.01 20.03, 10.52 40.11, 30.29 41.56, 31.78 10.74, 10.01
20.03))', 4326)
);

UPDATE sample_polys
 SET wkb = st_asbinary (geometry)
 WHERE id = 1115;
```

```
SELECT id, st_astext (st_polyfromwkb (wkb, 4326))
 AS "polygons"
 FROM sample_polys;

id    polygons

1115    POLYGON (10.01000000 20.03000000, 31.78000000 10.74000000, 30.29000000
41.56000000, 10.52000000 40.11000000, 10.01000000 20.03000000)
```

# ST_Polygon

## Definition

The ST_Polygon accessor function takes a well-known text (WKT) representation and a spatial reference ID (SRID) and generates an ST_Polygon.

**Note:**
When creating spatial tables that will be used with ArcGIS, it is best to create the column as the geometry supertype (for example, ST_Geometry) rather than specifying an ST_Geometry subtype.

## Syntax

### Oracle

```
sde.st_polygon (wkt clob, srid integer)
```

### PostgreSQL

```
sde.st_polygon (wkt clob, srid integer)
sde.st_polygon (esri_shape bytea, srid integer)
```

### SQLite

```
st_polygon (wkt text, srid int32)
```

## Return type

ST_Polygon

## Example

The following `CREATE TABLE` statement creates the polygon_test tables, which have a single column, p1. The subsequent `INSERT` statement converts a ring (a polygon that is both closed and simple) into an ST_Polygon and inserts it into the p1 column using the ST_Polygon function.

### Oracle

```
CREATE TABLE polygon_test (p1 sde.st_geometry);

INSERT INTO polygon_test VALUES (
 sde.st_polygon ('polygon ((10.01 20.03, 20.94 21.34, 35.93 10.04, 10.01 20.03))', 4326)
);
```

### PostgreSQL

```
CREATE TABLE polygon_test (p1 sde.st_geometry);

INSERT INTO polygon_test VALUES (
 sde.st_polygon ('polygon ((10.01 20.03, 20.94 21.34, 35.93 10.04, 10.01 20.03))', 4326)
);
```

### SQLite

```
CREATE TABLE poly_test (id integerp1 geometryblob);

SELECT AddGeometryColumn(
 NULL,
 'poly_test',
 'p1',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO poly_test VALUES (
 1,
 st_polygon ('polygon ((10.01 20.03, 20.94 21.34, 35.93 10.04, 10.01 20.03))', 4326)
);
```

# ST_Relate

## Definition

ST_Relate compares two geometries and returns 1 (Oracle and SQLite) or t (PostgreSQL) if the geometries meet the conditions specified by the DE-9IM pattern matrix string; otherwise, 0 (Oracle and SQLite) or f (PostgreSQL) is returned.

There is a second option when using ST_Relate in SQLite and Oracle: you can compare two geometries to return a string representing the DE-9IM pattern matrix that defines the geometries' relationship to one another.

## Syntax

### Oracle

Option 1

```
sde.st_relate (geometry1 sde.st_geometry, geometry2 sde.st_geometry, patternMatrix
string)
```

Option 2

```
sde.st_relate (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### PostgreSQL

```
sde.st_relate (geometry1 sde.st_geometry, geometry2 sde.st_geometry, patternMatrix
string)
```

### SQLite

Option 1

```
st_relate (geometry1 geometryblob, geometry2 geometryblob, patternMatrix string)
```

Option 2

```
st_relate (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

Boolean is returned for PostgreSQL.

Option 1 for SQLite and Oracle returns an integer.

Option 2 for SQLite and Oracle returns a string.

## Examples

A DE-9IM pattern matrix is a device for comparing geometries. There are several types of such matrices. For example, you can use the ST_Relate function and the equals pattern matrix (T*F**FFF*) to discover if any two geometries are equal, but you can also provide the DE-9IM pattern (1*F**FFF*). With the latter pattern, ST_Relate will tell you if two geometries are equal with the first position, which indicates if the interiors of the intersection of both geometries is a line (dimension of 1).

In the examples below, a table, relate_test, is created with three spatial columns, and point features are inserted into each one. The ST_Relate function is used in the `SELECT` statement to test if the points are equal.

If you want to determine if geometries are equal and you don't need to find the dimensionality of the relationship, use the ST_Equals function instead.

### Oracle

The first example shows the first ST_Relate option, which compares geometries based on a DE-9IM pattern matrix to return 1 if the geometries meet the requirements defined in the matrix or 0 if the geometries do not.

```
CREATE TABLE relate_test (
 id NUMBER GENERATED ALWAYS AS IDENTITY minvalue 0,
 g1 sde.st_geometry
);

CREATE TABLE relate_test2 (
 id NUMBER GENERATED ALWAYS AS IDENTITY minvalue 0,
 g2 sde.st_geometry
);

CREATE TABLE relate_test3 (
 id NUMBER GENERATED ALWAYS AS IDENTITY minvalue 0,
 g3 sde.st_geometry
);
```

```
INSERT INTO relate_test (g1) VALUES (sde.st_geometry ('point (10.02 20.01)', 4326));

INSERT INTO relate_test2 (g2) VALUES (sde.st_geometry ('point (10.02 20.01)', 4326));

INSERT INTO relate_test3 (g3) VALUES (sde.st_geometry ('point (30.01 20.01)', 4326));
```

```
SELECT sde.st_relate (relate_test.g1, relate_test2.g2, 'T*F**FFF*') AS "g1=g2",
       sde.st_relate (relate_test.g1, relate_test3.g3, 'T*F**FFF*') AS "g1=g3",
       sde.st_relate (relate_test2.g2, relate_test3.g3, 'T*F**FFF*') AS "g2=g3"
FROM relate_test, relate_test2, relate_test3;
```

It returns the following:

```
g1=g2     g1=g3      g2=g3

1                0                0
```

This example shows the second option. It compares two geometries and returns the DE-9IM pattern matrix.

```
SELECT sde.st_relate (relate_test.g1,relate_test2.g2) AS "g1 rel g2"
 FROM relate_test, relate_test2;
```

It returns the following:

```
g1 rel g2
0FFFFFFF2
```

## PostgreSQL

The example compares geometries based on a DE-9IM pattern matrix to return t if the geometries meet the requirements defined in the matrix or f if the geometries do not.

```
CREATE TABLE relate_test (
 id SERIAL,
 g1 sde.st_geometry
);

CREATE TABLE relate_test2 (
 id SERIAL,
 g2 sde.st_geometry
);

CREATE TABLE relate_test3 (
 id SERIAL,
 g3 sde.st_geometry
);
```

```
INSERT INTO relate_test(g1) VALUES (sde.st_geometry ('point (10.02 20.01)', 4326));

INSERT INTO relate_test2 (g2) VALUES (sde.st_geometry ('point (10.02 20.01)', 4326));

INSERT INTO relate_test3 (g3) VALUES (sde.st_geometry ('point (30.01 20.01)', 4326));
```

```
SELECT sde.st_relate (relate_test.g1, relate_test2.g2, 'T*F**FFF*') AS "g1=g2",
       sde.st_relate (relate_test.g1, relate_test3.g3, 'T*F**FFF*') AS "g1=g3",
       sde.st_relate (relate_test2.g2, relate_test3.g3, 'T*F**FFF*') AS "g2=g3"
FROM relate_test, relate_test2, relate_test3;
```

It returns the following:

```
g1=g2     g1=g3     g2=g3

t          f         f
```

## SQLite

This first example shows the first ST_Relate option, which compares eometries based on a DE-9IM pattern matrix to return 1 if the geometries meet the requirements defined in the matrix or 0 if the geometries do not.

```
CREATE TABLE relate_test (id integer primary key autoincrement not null);

SELECT AddGeometryColumn(
 NULL,
 'relate_test',
 'g1',
 4326,
 'point',
 'xy',
 'null'
);

CREATE TABLE relate_test2 (id integer primary key autoincrement not null);

SELECT AddGeometryColumn(
 NULL,
 'relate_test2',
 'g2',
 4326,
 'point',
 'xy',
 'null'
);

CREATE TABLE relate_test3 (id integer primary key autoincrement not null);

SELECT AddGeometryColumn(
 NULL,
 'relate_test3',
 'g3',
 4326,
 'point',
 'xy',
 'null'
);
```

```
INSERT INTO relate_test (g1) VALUES (
 st_geometry ('point (10.02 20.01)', 4326)
);

INSERT INTO relate_test2 (g2) VALUES (
 st_geometry ('point (10.02 20.01)', 4326)
);

INSERT INTO relate_test3 (g3) VALUES (
 st_geometry ('point (30.01 20.01)', 4326)
);
```

```
SELECT st_relate (relate_test.g1, relate_test2.g2, 'T*F**FFF*') AS "g1=g2",
       st_relate (relate_test.g1, relate_test3.g3, 'T*F**FFF*') AS "g1=g3",
       st_relate (relate_test2.g2, relate_test3.g3, 'T*F**FFF*') AS "g2=g3"
 FROM relate_test, relate_test2, relate_test3;
```

It returns the following:

```
g1=g2     g1=g3     g2=g3

1          0         0
```

This example shows the second option. It compares two geometries and returns the DE-9IM pattern matrix.

```
SELECT st_relate (relate_test.g1,relate_test2.g2) AS "g1 rel g2"
 FROM relate_test, relate_test2;
```

It returns the following:

```
g1 rel g2
0FFFFFFF2
```

# ST_SRID

## Definition

ST_SRID takes a geometry object and returns its spatial reference ID.

## Syntax

### Oracle and PostgreSQL

```
sde.st_srid (geometry1 sde.st_geometry)
```

### SQLite

```
st_srid (geometry1 geometryblob)
```

## Return type

Integer

## Examples

The following table is created:

In the next statement, a point geometry located at coordinate (10.01, 50.76) is inserted into the geometry column g1. When the point geometry is created, it is assigned the SRID value of 4326.

The ST_SRID function returns the spatial reference ID of the geometry just entered.

### Oracle

```
CREATE TABLE srid_test (g1 sde.st_geometry);
```

```
INSERT INTO SRID_TEST VALUES (
 sde.st_geometry ('point (10.01 50.76)', 4326)
);
```

```
SELECT sde.st_srid (g1) SRID_G1
 FROM SRID_TEST;

SRID_G1

4326
```

### PostgreSQL

```
CREATE TABLE srid_test (g1 sde.st_geometry);
```

```
INSERT INTO srid_test VALUES (
 sde.st_point ('point (10.01 50.76)', 4326)
);
```

```
SELECT sde.st_srid (g1)
 AS SRID_G1
 FROM srid_test;

srid_g1

4326
```

### SQLite

```
CREATE TABLE srid_test (id integer);

SELECT AddGeometryColumn(
 NULL,
 'srid_test',
 'g1',
 4326,
 'point',
 'xy',
 'null'
);
```

```
INSERT INTO srid_test VALUES (
 1,
 st_point ('point (10.01 50.76)', 4326)
);
```

```
SELECT st_srid (g1)
 AS "SRID"
 FROM srid_test;

SRID

4326
```

# ST_StartPoint

## Definition

ST_StartPoint returns the first point of a linestring.

## Syntax

### Oracle and PostgreSQL

```
sde.st_startpoint (ln1 sde.st_geometry)
```

### SQLite

```
st_startpoint (ln1 geometryblob)
```

## Return type

ST_Point

## Examples

The startpoint_test table is created with the gid integer column, which uniquely identifies the rows of the table, and the ln1 ST_LineString column.

The INSERT statements insert the ST_LineStrings into the ln1 column. The first ST_LineString does not have z-coordinates or measures, while the second ST_LineString has both.

The ST_StartPoint function extracts the first point of each ST_LineString. The first point in the list does not have a z-coordinate or measure, while the second point has both, because the source linestring does.

### Oracle

```
CREATE TABLE startpoint_test (
 gid integer,
 ln1 sde.st_geometry
);
```

```
INSERT INTO STARTPOINT_TEST VALUES (
 1,
 sde.st_linefromtext ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO STARTPOINT_TEST VALUES (
 2,
 sde.st_linefromtext ('linestring zm(10.02 20.01 5 7, 23.73 21.92 6.5 7.1, 30.10 40.23
6.9 7.2)', 4326)
);
```

```
SELECT gid, sde.st_astext (sde.st_startpoint (ln1)) Startpoint
 FROM STARTPOINT_TEST;
```

```
GID  Startpoint

1    POINT (10.02000000 20.01000000)
2    POINT ZM (10.02000000 20.01000000 5.00000000 7.00000000)
```

### PostgreSQL

```
CREATE TABLE startpoint_test (
 gid serial,
 ln1 sde.st_geometry
);
```

```
INSERT INTO startpoint_test (ln1) VALUES (
 sde.st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO startpoint_test (ln1) VALUES (
 sde.st_linestring ('linestring zm(10.02 20.01 5 7, 23.73 21.92 6.5 7.1, 30.10 40.23
6.9 7.2)', 4326)
);
```

```
SELECT gid, sde.st_astext (sde.st_startpoint (ln1))
 AS Startpoint
 FROM startpoint_test;

gid  startpoint

1    POINT (10.02000000 20.01000000)
2    POINT ZM (10.02000000 20.01000000 5.00000000 7.00000000)
```

### SQLite

```
CREATE TABLE startpoint_test (
 gid integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'startpoint_test',
 'ln1',
 4326,
 'linestringzm',
 'xyzm',
 'null'
);
```

```
INSERT INTO startpoint_test (ln1) VALUES (
 st_linestring ('linestring (10.02 20.01, 23.73 21.92, 30.10 40.23)', 4326)
);

INSERT INTO startpoint_test(ln1) VALUES (
```

```
 st_linestring ('linestring zm(10.02 20.01 5 7, 23.73 21.92 6.5 7.1, 30.10 40.23 6.9
7.2)', 4326)
);
```

```
SELECT gid, st_astext (st_startpoint (ln1))
 AS "Startpoint"
 FROM startpoint_test;

gid  Startpoint

1    POINT (10.02000000 20.01000000)
2    POINT ZM (10.02000000 20.01000000 5.00000000 7.00000000)
```

# ST_Surface

**Note:**
Oracle and SQLite only

## Definition

ST_Surface constructs a surface feature from a well-known text representation. Surfaces are similar to polygons, but they have values at every point across their extent.

## Syntax

### Oracle

```
sde.st_surface (wkt clob, srid integer)
```

### SQLite

```
st_surface (wkt text, srid int32)
```

## Return type

ST_Polygon

## Example

A table, surf_test, is created, and a surface geometry is inserted into it.

### Oracle

```
CREATE TABLE surf_test (
 id integer,
 geometry sde.st_geometry
);

INSERT INTO SURF_TEST VALUES (
 1110,
 sde.st_surface ('polygon ((110 120, 110 140, 120 130, 110 120))', 4326)
);
```

### SQLite

```
CREATE TABLE surf_test (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'surf_test',
 'geometry',
 4326,
```

```
 'polygon',
 'xy',
 'null'
);

INSERT INTO SURF_TEST VALUES (
 1110,
 st_surface ('polygon ((110 120, 110 140, 120 130, 110 120))', 4326)
);
```

# ST_SymmetricDiff

## Definition

ST_SymmetricDiff takes two geometry objects and returns a geometry object composed of the parts of the source objects that are not common to both.

## Syntax

### Oracle and PostgreSQL

```
sde.st_symmetricdiff (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_symmetricdiff (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

For a special report, the county supervisor must determine the watershed and hazardous plumes radii areas that aren't intersected.

The watershed table contains an id column, a column to store the watershed name (wname), and a shape column, which stores the watershed area geometry.

The plumes table stores the identity of the site in the id column, while the actual geographic location of each site is stored in the site point column.

The ST_Buffer function generates a buffer surrounding the hazardous waste site points. The ST_SymmetricDiff function returns the polygons of the buffered hazardous waste sites and the watersheds that don't intersect.

The symmetric difference of the hazardous waste sites and the watershed results in the subtraction of the intersected areas.

### Oracle

```
CREATE TABLE watershed (
 id integer,
 wname varchar(40),
 shape sde.st_geometry
);

CREATE TABLE plumes (
```

```
 id integer,
 site sde.st_geometry
);
```

```
INSERT INTO WATERSHED (ID, WNAME, SHAPE) VALUES (
 1,
 'Big River',
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO WATERSHED (ID, WNAME, SHAPE) VALUES (
 2,
 'Lost Creek',
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO WATERSHED (ID, WNAME, SHAPE) VALUES (
 3,
 'Szymborska Stream',
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO PLUMES (ID, SITE) VALUES (
 20,
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO PLUMES (ID, SITE) VALUES (
 21,
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
SELECT ws.id WS_ID,
 sde.st_area (sde.st_symmetricdiff (sde.st_buffer (p.site, .1), ws.shape)) AREA_NO_INT
 FROM PLUMES p, WATERSHED ws
 WHERE p.id = 20;

     SA_ID     AREA_NO_INT

         1     100.031393
         2     400.031393
         3     400.015697
```

## PostgreSQL

```
CREATE TABLE watershed (
 id serial,
 wname varchar(40),
 shape sde.st_geometry
);

CREATE TABLE plumes (
 id serial,
 site sde.st_geometry
);
```

```
INSERT INTO watershed (wname, shape) VALUES (
 'Big River',
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO watershed (wname, shape) VALUES (
 'Lost Creek',
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO watershed (wname, shape) VALUES (
 'Szymborska Stream',
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO plumes (site) VALUES (
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO plumes (site) VALUES (
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
SELECT ws.id AS WS_ID,
 sde.st_area (sde.st_symmetricdiff (sde.st_buffer (p.site, .1), ws.shape)) AS "no
intersection"
 FROM plumes p, watershed ws
 WHERE p.id = 1;

    ws_id     no intersection

         1    100.031393502001
         2    400.031393502001
         3    400.01569751
```

## SQLite

```
CREATE TABLE watershed (
 id integer primary key autoincrement not null,
 wname text(40)
);

SELECT AddGeometryColumn(
 NULL,
```

```
 'watershed',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE plumes (
 id integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'plumes',
 'site',
 4326,
 'point',
 'xy',
 'null'
);
```

```
INSERT INTO watershed (wname, shape) VALUES (
 'Big River',
 st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO watershed (wname, shape) VALUES (
 'Lost Creek',
 st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO watershed (wname, shape) VALUES (
 'Szymborska Stream',
 st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO plumes (site) VALUES (
 st_geometry ('point (60 60)', 4326)
);

INSERT INTO plumes (site) VALUES (
 st_geometry ('point (30 30)', 4326)
);
```

```
SELECT ws.id AS WS_ID,
 st_area (st_symmetricdiff (st_buffer (p.site, .1), ws.shape)) AS "no intersection"
 FROM plumes p, watershed ws
 WHERE p.id = 1;

    WS_ID     no intersection

         1    400.031393502001
         2    100.031393502001
         3    400.01569751
```

# ST_Touches

## Definition

ST_Touches returns 1 (Oracle and SQLite) or t (PostgreSQL) if none of the points common to both geometries intersect the interiors of both geometries; otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL). At least one geometry must be an ST_LineString, ST_Polygon, ST_MultiLineString, or ST_MultiPolygon.

## Syntax

### Oracle and PostgreSQL

```
sde.st_touches (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_touches (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

Boolean

## Example

The GIS technician has been asked by his boss to provide a list of all sewer lines that possess endpoints that intersect another sewer line.

The sewerlines table is created with three columns. The first column, sewer_id, uniquely identifies each sewer line. The integer class column identifies the type of sewer line generally associated with the line's capacity. The sewer column stores the sewer line's geometry.

The SELECT query uses the ST_Touches function to return a list of sewers that touch one another.

### Oracle

```
CREATE TABLE sewerlines (
 sewer_id integer,
 sewer sde.st_geometry
);

INSERT INTO SEWERLINES VALUES (
 1,
 sde.st_mlinefromtext ('multilinestring ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO SEWERLINES VALUES (
 2,
 sde.st_mlinefromtext ('multilinestring ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO SEWERLINES VALUES (
 3,
 sde.st_mlinefromtext ('multilinestring ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);
```

```
INSERT INTO SEWERLINES VALUES (
 4,
 sde.st_linestring ('linestring (60 60, 70 70)', 4326)
);

INSERT INTO SEWERLINES VALUES (
 5,
 sde.st_linestring ('linestring (30 30, 60 60)', 4326)
);
```

```
SELECT s1.sewer_id, s2.sewer_id
 FROM SEWERLINES s1, SEWERLINES s2
 WHERE sde.st_touches (s1.sewer, s2.sewer) = 1;

  SEWER_ID   SEWER_ID

         1          5
         3          4
         4          3
         4          5
         5          1
         5          3
         5          4
```

## PostgreSQL

```
CREATE TABLE sewerlines (
 sewer_id serial,
 sewer sde.st_geometry);

INSERT INTO sewerlines (sewer) VALUES (
 sde.st_multilinestring ('multilinestring ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sewerlines (sewer) VALUES (
 sde.st_multilinestring ('multilinestring ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sewerlines (sewer) VALUES (
 sde.st_multilinestring ('multilinestring ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO sewerlines (sewer) VALUES (
 sde.st_linestring ('linestring (60 60, 70 70)', 4326)
);

INSERT INTO sewerlines (sewer) VALUES (
 sde.st_linestring ('linestring (30 30, 60 60)', 4326)
);
```

```
SELECT s1.sewer_id, s2.sewer_id
 FROM sewerlines s1, sewerlines s2
 WHERE sde.st_touches (s1.sewer, s2.sewer) = 't';

  SEWER_ID   SEWER_ID
```

```
         1          5
         3          4
         4          3
         4          5
         5          1
         5          3
         5          4
```

## SQLite

```
CREATE TABLE sewerlines (
 sewer_id integer primary key autoincrement not null
);

SELECT AddGeometryColumn(
 NULL,
 'sewerlines',
 'sewer',
 4326,
 'geometry',
 'xy',
 'null'
);

INSERT INTO sewerlines (sewer) VALUES (
 st_multilinestring ('multilinestring ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sewerlines (sewer) VALUES (
 st_multilinestring ('multilinestring ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sewerlines (sewer) VALUES (
 st_multilinestring ('multilinestring ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO sewerlines (sewer) VALUES (
 st_linestring ('linestring (60 60, 70 70)', 4326)
);

INSERT INTO sewerlines (sewer) VALUES (
 st_linestring ('linestring (30 30, 60 60)', 4326)
);
```

```
SELECT s1.sewer_id, s2.sewer_id
 FROM SEWERLINES s1, SEWERLINES s2
 WHERE st_touches (s1.sewer, s2.sewer) = 1;

sewer_id   sewer_id

1          5
3          4
3          5
4          3
4          5
5          1
5          3
5          4
```

# ST_Transform

## Definition

ST_Transform takes two-dimensional ST_Geometry data as input and returns values converted into the spatial reference specified by the spatial reference ID (SRID) you provide.

**⚠ Caution:**
If you registered the spatial column with the PostgreSQL database using the st_register_spatial_column function, the SRID at the time of registration is written to the sde_geometry_columns table. If you created a spatial index on the spatial column in an Oracle database, the SRID at the time the spatial index was created is written to the st_geometry_columns table. Using ST_Transform to alter the SRID of the ST_Geometry data does not update the SRID in the sde_geometry_columns or st_geometry_columns table.

When the geographic coordinate systems are different, ST_Transform performs a geographic transformation. A geographic transformation converts between two geographic coordinate systems. A geographic transformation is defined in a particular direction, for example, from NAD 1927 to NAD 1983, but the ST_Transform function will correctly apply the transformation no matter what the source and destination coordinate systems are.

Geographic transformation methods can be divided into two types: equation based and file based. Equation-based methods are self-contained and do not need any external information. File-based methods use on-disk files to calculate offset values. They are usually more accurate than equation-based methods. File-based methods are commonly used in Australia, Canada, Germany, New Zealand, Spain, and the United States. The files (except for the Canadian ones) can be obtained from an ArcGIS Pro installation or directly from the various national mapping agencies.

To support file-based transformations, you must place the files on the server where the database is installed in the same relative folder structure as the pedata folder in the ArcGIS Pro installation directory.

For example, there is a folder called `pedata` in the `Resources` folder of the ArcGIS Pro installation directory. The `pedata` folder contains several subfolders, but the three folders that contain supported file-based methods are `harn`, `nadcon`, and `ntv2`. Either copy the `pedata` folder and its contents from the ArcGIS installation directory to the database server, or create a directory on the database server that contains the supported file-based transformation method subdirectories and files. Once the files are on the database server, set an operating system environment variable called PEDATAHOME on the same server. Set the PEDATAHOME variable to the location of the directory that contains the subdirectories and files; for example, if the pedata folder is copied to `C:\pedata` on a Microsoft Windows server, set the PEDATAHOME environment variable to `C:\pedata`.

Refer to your operating system documentation for information on how to set an environment variable.

After setting PEDATAHOME, you must start a new SQL session before you can use the ST_Transform function; however, the server does not need to be restarted.

### Using ST_Transform with PostgreSQL

In PostgreSQL, you can convert between spatial references that have the same geographic coordinate system or different geographic coordinate systems.

If the data is stored in a database (rather than a geodatabase), do the following to change the spatial reference of the ST_Geometry data when geographic coordinate systems are the same:

1. Create a backup copy of the table.
2. Create a second (destination) ST_Geometry column on the table.
3. Register the destination ST_Geometry column, specifying the new SRID.
   This specifies the spatial reference of the column by placing a record in the sde_geometry_columns system table.
4. Run the ST_Transform function and specify that the transformed data be output to the destination ST_Geometry column.
5. Unregister the first (source) ST_Geometry column.

If the data is stored in a geodatabase, you should use ArcGIS tools to reproject the data to a new feature class. Running ST_Transform on a geodatabase feature class bypasses functionality to update geodatabase system tables with the new SRID.

### Using ST_Transform with Oracle

In Oracle, you can convert between spatial references that have the same geographic coordinate system or different geographic coordinate systems.

If the data is stored in a database (rather than a geodatabase) and no spatial index has been defined on the spatial column, you can add a second ST_Geometry column and output the transformed data to it. You can keep both the original (source) ST_Geometry column and the destination ST_Geometry column in the table, though you can only display one column at a time in ArcGIS using a view or altering the query layer definition for the table.

If the data is stored in a database (rather than a geodatabase) and the spatial column has a spatial index defined on it, you cannot preserve the original ST_Geometry column. Once a spatial index has been defined on an ST_Geometry column, the SRID is written to the st_geometry_columns metadata table. ST_Transform does not update that table.

1. Create a backup copy of the table.
2. Create a second (destination) ST_Geometry column on the table.
3. Run the ST_Transform function and specify that the transformed data be output to the destination ST_Geometry column.
4. Drop the spatial index from the source ST_Geometry column.
5. Drop the source ST_Geometry column.
6. Create a spatial index on the destination ST_Geometry column.

If the data is stored in a geodatabase, you should use ArcGIS tools to reproject the data to a new feature class.Running ST_Transform on a geodatabase feature class bypasses functionality to update geodatabase system tables with the new SRID.

### Using ST_Transform with SQLite

In SQLite, you can convert between spatial references that have the same geographic coordinate system or different geographic coordinate systems.

## Syntax

### Source and destination spatial references have the same geographic coordinate system

*Oracle and PostgreSQL*

```
sde.st_transform (geometry1 sde.st_geometry, srid integer)
```

*SQLite*

```
st_transform (geometry1 geometryblob, srid in32)
```

### Source and destination spatial references do not have the same geographic coordinate system

*Oracle*

```
sde.st_transform (g1 sde.st_geometry, srid integer, geogtrans_id integer)
```

*PostgreSQL*

Option 1: `sde.st_transform (g1 sde.st_geometry, srid int)`

Option 2: `sde.st_transform (g1 sde.st_geometry, srid int, [geogtrans_id int])`

Option 3: `sde.st_transform (g1 sde.st_geometry, srid int, [extent double] [prime meridian double] [unit conversion factor double])`

In option 3, you can optionally specify the extent as a comma separated list of coordinates in the following order: lower left x-coordinate, lower left y-coordinate, upper right x-coordinate, upper right y-coordinate. If you do not specify an extent, ST_Transform uses a larger, more general extent.

When specifying an extent, the prime meridian and unit conversion factor parameters are optional. You only need to provide this information if the extent values you specify do not use the Greenwich prime meridian or decimal degrees.

*SQLite*

```
st_transform (geometry1 geometryblob, srid int32, geogtrans_id int32)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Examples

### Transforming data when the source and destination spatial references have the same geographic coordinate system

The following example creates a table, transform_test, which has two linestring columns: ln1 and ln2. A line is inserted into ln1 with SRID 4326. The ST_Transform function is then used in an `UPDATE` statement to take the

linestring in ln1, convert it from the coordinate reference assigned to SRID 4326 to the coordinate reference assigned to SRID 3857, and place it in column ln2.

**Note:**
SRIDs 4326 and 3857 have the same geographic datum.

*Oracle*

```
CREATE TABLE transform_test (
 ln1 sde.st_geometry,
 ln2 sde.st_geometry);

INSERT INTO transform_test (ln1) VALUES (
 sde.st_geometry ('linestring (10.01 40.03, 92.32 29.39)', 4326)
);
```

```
UPDATE transform_test
 SET ln2 = sde.st_transform (ln1, 3857);
```

*PostgreSQL*

```
CREATE TABLE transform_test (
 ln1 sde.st_geometry,
 ln2 sde.st_geometry);

INSERT INTO transform_test (ln1) VALUES (
 sde.st_geometry ('linestring (10.01 40.03, 92.32 29.39)', 4326)
);
```

```
UPDATE transform_test
 SET ln2 = sde.st_transform (ln1, 3857);
```

*SQLite*

```
CREATE TABLE transform_test (id integer);

SELECT AddGeometryColumn(
 NULL,
 'transform_test',
 'ln1',
 4326,
 'linestring',
 'xy',
 'null'
);

INSERT INTO transform_test (ln1) VALUES (
 st_geometry ('linestring (10.01 40.03, 92.32 29.39)', 4326)
);
```

```
UPDATE transform_test
 SET ln1 = st_transform (ln1, 3857);
```

## Transforming data when the source and destination spatial references do not have the same geographic coordinate system

In the following example, table n27 is created, which contains an ID column and a geometry column. A point is inserted to table n27 with an SRID of 4267. The 4267 SRID uses the NAD 1927 geographic coordinate system.

Next, table n83 is created and the ST_Transform function is used to insert the geometry from table n27 into table n83, but with an SRID of 4269 and geographic transformation ID 1241. SRID 4269 uses the NAD 1983 geographic coordinate system and 1241 is the well-known ID for the NAD_1927_To_NAD_1983_NADCON transformation. This transformation is file based and can be used for the lower 48 states of the United States.

**Tip:**

For lists of supported geographic transformations, see Esri technical article 000004829 and the links provided in the article's **Related Information** section.

*Oracle*

```
--Create table.
CREATE TABLE n27 (
 id integer,
 geometry sde.st_geometry
);

--Insert point with SRID 4267.
INSERT INTO N27 (id, geometry) VALUES (
 1,
 sde.st_geometry ('point (-123.0 49.0)', 4267)
);

--Create the n83 table as the destination table of the transformation.
CREATE TABLE n83 (
 id integer,
 geometry sde.st_geometry
```

```
);

--Run the transformation.
INSERT INTO N83 (id, geometry)(
 select c.id, sde.st_transform (c.geometry, 4269, 1241)
 from N27 c
);
```

If PEDATAHOME is defined correctly, a SELECT statement run against the n83 table will return the following:

```
SELECT id, sde.st_astext (geometry) description
 FROM N83;

ID        DESCRIPTION
1 | POINT((-123.00130569 48.999828199))
```

*PostgreSQL*

```
--Option 1
--Gets geographic transformation from ST_Geometry libraries.
--Does not require you to provide a GTid.
--Performs an equation-based transformation between two geographic coordinate systems
--with different datums. (SRID 4267/DATUM NAD27 to SRID 4269/DATUM NAD 83)

--Provide point to transform.
SELECT sde.ST_AsText(sde.ST_Transform(
  sde.ST_Geometry('point (-155.7029 63.6096)',4267), 4269));

--Returns output in SRID 4269.
"POINT ( -155.70290000 63.60960000)"
```

```
--Option 2
--Example uses input point in SRID 3857(DATUM: WGS 1984)
--and geographic transformation ID (GTid) 1251.
--Transforms point to SRID 102008 (DATUM: NAD 83)

--Provide point to transform.
SELECT sde.ST_AsText(sde.ST_Transform(
  sde.ST_Geometry('point (-13244252.9404 4224702.5198)', 3857), 102008, 1251));

--Returns output in SRID 102008.
"POINT (-1957193.14740000 -297059.19680000)"
```

*SQLite*

```
--Create source table.
CREATE TABLE n27 (id integer);

SELECT AddGeometryColumn(
 NULL,
 'n27',
 'geometry',
 4267,
 'point',
 'xy',
```

```
 'null'
);

--Insert point with SRID 4267.
INSERT INTO n27 (id, geometry) VALUES (
 1,
 st_geometry ('point (-123.0 49.0)', 4267)
);

--Create the n83 table as the destination table of the transformation.
CREATE TABLE n83 (id integer);

SELECT AddGeometryColumn(
 NULL,
 'n83',
 'geometry',
 4269,
 'point',
 'xy',
 'null'
);

--Run the transformation.
INSERT INTO n83 (id, geometry) VALUES (
 1,
 st_transform ((select geometry from n27 where id=1), 4269, 1241)
);
```

# ST_Union

## Definition

ST_Union returns a geometry object that is the combination of two source objects.

## Syntax

### Oracle and PostgreSQL

```
sde.st_union (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_union (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

### Oracle and PostgreSQL

ST_Geometry

### SQLite

Geometryblob

## Example

The sensitive_areas table stores the IDs of threatened institutions in addition to the shape column, which stores the institutions' polygon geometries.

The hazardous_sites table stores the identity of the sites in the id column, while the actual geographic location of each site is stored in the site point column.

The ST_Buffer function generates a buffer surrounding the hazardous waste sites. The ST_Union function generates polygons from the union of the buffered hazardous waste sites and sensitive area polygons. The ST_Area function returns the area of these polygons.

### Oracle

```
CREATE TABLE sensitive_areas (
 id integer,
 shape sde.st_geometry
);

CREATE TABLE hazardous_sites (
 id integer,
 site sde.st_geometry
);


INSERT INTO SENSITIVE_AREAS VALUES (
 1,
```

```
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO SENSITIVE_AREAS VALUES (
 2,
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO SENSITIVE_AREAS VALUES (
 3,
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO HAZARDOUS_SITES VALUES (
 4,
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO HAZARDOUS_SITES VALUES (
 5,
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
SELECT sa.id SA_ID, hs.id HS_ID,
 sde.st_area (sde.st_union (sde.st_buffer (hs.site, .01), sa.shape)) UNION_AREA
 FROM HAZARDOUS_SITES hs, SENSITIVE_AREAS sa;

     SA_ID  HS_ID     UNION_AREA

         1    4       100.000313935011
         2    4       400.000313935011
         3    4       400.000235451258
         1    5       100.000235451258
         2    5       400.000235451258
         3    5       400.000313935011
```

## PostgreSQL

```
CREATE TABLE sensitive_areas (
 id integer,
 shape sde.st_geometry
);

CREATE TABLE hazardous_sites (
 id integer,
 site sde.st_geometry
);


INSERT INTO SENSITIVE_AREAS VALUES (
 1,
 sde.st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO SENSITIVE_AREAS VALUES (
 2,
 sde.st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);
```

```
INSERT INTO SENSITIVE_AREAS VALUES (
 3,
 sde.st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO HAZARDOUS_SITES VALUES (
 4,
 sde.st_geometry ('point (60 60)', 4326)
);

INSERT INTO HAZARDOUS_SITES VALUES (
 5,
 sde.st_geometry ('point (30 30)', 4326)
);
```

```
SELECT sa.id AS SA_ID, hs.id AS HS_ID,
 sde.st_area (sde.st_union (sde.st_buffer (hs.site, .01), sa.shape)) AS UNION_AREA
 FROM hazardous_sites hs, sensitive_areas sa;

     sa_id  hs_id   union_area

         1    4       100.000313935011
         2    4       400.000313935011
         3    4       400.000235451258
         1    5       100.000235451258
         2    5       400.000235451258
         3    5       400.000313935011
```

## SQLite

```
CREATE TABLE sensitive_areas (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'sensitive_areas',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE hazardous_sites (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'hazardous_sites',
 'site',
 4326,
 'point',
 'xy',
 'null'
);

INSERT INTO sensitive_areas VALUES (
```

```
 10,
 st_geometry ('polygon ((20 30, 30 30, 30 40, 20 40, 20 30))', 4326)
);

INSERT INTO sensitive_areas VALUES (
 11,
 st_geometry ('polygon ((30 30, 30 50, 50 50, 50 30, 30 30))', 4326)
);

INSERT INTO sensitive_areas VALUES (
 12,
 st_geometry ('polygon ((40 40, 40 60, 60 60, 60 40, 40 40))', 4326)
);

INSERT INTO hazardous_sites VALUES (
 40,
 st_geometry ('point (60 60)', 4326)
);

INSERT INTO hazardous_sites VALUES (
 41,
 st_geometry ('point (30 30)', 4326)
);
```

```
SELECT sa.id AS "sa_id", hs.id AS "hs_id",
 st_area (st_union (st_buffer (hs.site, .01), sa.shape)) AS "union"
 FROM hazardous_sites hs, sensitive_areas sa;

     sa_id  hs_id   union

         1    4       100.000313935011
         2    4       400.000313935011
         3    4       400.000235451258
         1    5       100.000235451258
         2    5       400.000235451258
         3    5       400.000313935011
```

# ST_Within

## Definition

ST_Within returns 1 (Oracle and SQLite) or t (PostgreSQL) if the first ST_Geometry object is completely inside the second; otherwise, it returns 0 (Oracle and SQLite) or f (PostgreSQL).

## Syntax

### Oracle and PostgreSQL

```
sde.st_within (geometry1 sde.st_geometry, geometry2 sde.st_geometry)
```

### SQLite

```
st_within (geometry1 geometryblob, geometry2 geometryblob)
```

## Return type

Boolean

## Example

In the example below, two tables are created: zones and squares. The SELECT statement finds all squares that intersect but are not completely within a lot.

### Oracle

```
CREATE TABLE squares (
 id integer,
 shape sde.st_geometry);

CREATE TABLE zones (
 id integer,
 shape sde.st_geometry);


INSERT INTO squares (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO squares (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO squares (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO zones (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
```

```
);

INSERT INTO zones (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO zones (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT s.id sq_id
 FROM SQUARES s, ZONES z
 WHERE sde.st_intersects (s.shape, z.shape) = 1
 AND sde.st_within (s.shape, z.shape) = 0;

SQ_ID

2
```

## PostgreSQL

```
CREATE TABLE squares (
 id integer,
 shape sde.st_geometry);

CREATE TABLE zones (
 id integer,
 shape sde.st_geometry);


INSERT INTO squares (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO squares (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO squares (id, shape) VALUES (
 3,
 sde.st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO zones (id, shape) VALUES (
 1,
 sde.st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO zones (id, shape) VALUES (
 2,
 sde.st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO zones (id, shape) VALUES (
 3,
```

```
 sde.st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT s.id
 AS sq_id
 FROM squares s, zones z
 WHERE st_intersects (s.shape, z.shape) = 't'
 AND st_within (s.shape, z.shape) = 'f';

sq_id

2
```

## SQLite

```
CREATE TABLE squares (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'squares',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

CREATE TABLE zones (
 id integer
);

SELECT AddGeometryColumn(
 NULL,
 'zones',
 'shape',
 4326,
 'polygon',
 'xy',
 'null'
);

INSERT INTO squares (id, shape) VALUES (
 1,
 st_polygon ('polygon ((0 0, 0 10, 10 10, 10 0, 0 0))', 4326)
);

INSERT INTO squares (id, shape) VALUES (
 2,
 st_polygon ('polygon ((20 0, 20 10, 30 10, 30 0, 20 0))', 4326)
);

INSERT INTO squares (id, shape) VALUES (
 3,
 st_polygon ('polygon ((40 0, 40 10, 50 10, 50 0, 40 0))', 4326)
);

INSERT INTO zones (id, shape) VALUES (
```

```
 1,
 st_polygon ('polygon ((-1 -1, -1 11, 11 11, 11 -1, -1 -1))', 4326)
);

INSERT INTO zones (id, shape) VALUES (
 2,
 st_polygon ('polygon ((19 -1, 19 11, 29 9, 31 -1, 19 -1))', 4326)
);

INSERT INTO zones (id, shape) VALUES (
 3,
 st_polygon ('polygon ((39 -1, 39 11, 51 11, 51 -1, 39 -1))', 4326)
);
```

```
SELECT s.id
 AS "sq_id"
 FROM squares s, zones1 z
 WHERE st_intersects (s.shape, z.shape) = 1
 AND st_within (s.shape, z.shape) = 0;

sq_id

2
```

# ST_X

## Definition

ST_X takes an ST_Point as an input parameter and returns its x-coordinate. In SQLite, ST_X can also update the x-coordinate of an ST_Point.

## Syntax

### Oracle and PostgreSQL

```
sde.st_x (point1 sde.st_point)
```

### SQLite

```
st_x (point1 geometryblob)
st_x (input_point geometryblob, new_Xvalue double)
```

## Return type

Double precision

The ST_X function can be used with SQLite to update the x-coordinate of a point. In that case, a geometryblob is returned.

## Examples

The x_test table is created with two columns: the gid column, which uniquely identifies the row, and the pt1 point column.

The INSERT statements insert two rows. One is a point without a z-coordinate or measure. The other column has both a z-coordinate and measure.

The SELECT query uses the ST_X function to get the x-coordinate of each point feature.

## Oracle

```
CREATE TABLE x_test (
 gid integer unique,
 pt1 sde.st_point
);
```

```
INSERT INTO X_TEST VALUES (
 1,
 sde.st_pointfromtext ('point (10.02 20.01)', 4326)
);

INSERT INTO X_TEST VALUES (
 2,
 sde.st_pointfromtext ('point zm(10.1 20.01 5 7)', 4326)
);
```

```
SELECT gid, sde.st_x (pt1) "The X coordinate"
 FROM X_TEST;

       GID       The X coordinate

         1            10.02
         2            10.10
```

## PostgreSQL

```
CREATE TABLE x_test (
 gid integer unique,
 pt1 sde.st_point
);
```

```
INSERT INTO x_test VALUES (
 1,
 sde.st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO x_test VALUES (
 2,
 sde.st_point ('point zm(10.1 20.01 5 7)', 4326)
);
```

```
SELECT gid, sde.st_x (pt1)
 AS "The X coordinate"
 FROM x_test;

       gid       The X coordinate

         1            10.02
         2            10.10
```

## SQLite

```
CREATE TABLE x_test (gid integer);

SELECT AddGeometryColumn(
 NULL,
 'x_test',
 'pt1',
 4326,
 'pointzm',
 'xyzm',
 'null'
);
```

```
INSERT INTO x_test VALUES (
 1,
 st_point ('point (10.02 20.01)', 4326)
);

INSERT INTO x_test VALUES (
 2,
 st_point ('point zm(10.1 20.01 5 7)', 4326)
);
```

```
SELECT gid, st_x (pt1)
 AS "The X coordinate"
 FROM x_test;

       gid       The X coordinate

         1            10.02
         2            10.10
```

The ST_X function can also be used to update the coordinate value of an existing point. In this example, ST_X is used to update the x-coordinate value of the first point in x_test.

```
UPDATE x_test
 SET pt1=st_x(
  (SELECT pt1 FROM x_test WHERE gid=1),
  10.04
  )
 WHERE gid=1;
```

# ST_Y

## Definition

ST_Y takes an ST_Point as an input parameter and returns its y-coordinate. In SQLite, ST_Y can also update the y-coordinate of an ST_Point.

## Syntax

### Oracle and PostgreSQL

```
sde.st_y (point1 sde.st_point)
```

### SQLite

```
double   st_y (point1  geometryblob)
geometry st_y (input_shape geometryblob, new_Yvalue double)
```

## Return type

Double precision

The ST_Y function can be used with SQLite to update the y-coordinate of a point. In that case, a geometryblob is returned.

## Example

The y_test table is created with two columns: the gid column, which uniquely identifies the row, and the pt1 point column.

The INSERT statements insert two rows. One is a point without a z-coordinate or measure. The other has both a z-coordinate and a measure.

The SELECT query uses the ST_Y function to return the y-coordinate of each point.

### Oracle

```
CREATE TABLE y_test (
 gid integer unique,
 pt1 sde.st_point
);
```

```
INSERT INTO Y_TEST VALUES (
 1,
 sde.st_pointfromtext ('point (10.02 20.02)', 4326)
);

INSERT INTO Y_TEST VALUES (
 2,
 sde.st_pointfromtext ('point zm(10.1 20.01 5.0 7.0)', 4326)
);
```

```
SELECT gid, sde.st_y (pt1) "The Y coordinate"
 FROM Y_TEST;

       GID     The Y coordinate

         1          20.02
         2          20.01
```

### PostgreSQL

```
CREATE TABLE y_test (
 gid integer unique,
 pt1 sde.st_point
);
```

```
INSERT INTO y_test VALUES (
 1,
 sde.st_point ('point (10.02 20.02)', 4326)
);

INSERT INTO y_test VALUES (
 2,
 sde.st_point ('point zm(10.1 20.01 5.0 7.0)', 4326)
);
```

```
SELECT gid, sde.st_y (pt1)
 AS "The Y coordinate"
 FROM y_test;

       gid    The Y coordinate

         1          20.02
         2          20.01
```

### SQLite

```
CREATE TABLE y_test (gid integer);

SELECT AddGeometryColumn(
 NULL,
 'y_test',
 'pt1',
 4326,
 'pointzm',
 'xyzm',
 'null'
);
```

```
INSERT INTO y_test VALUES (
 1,
 st_point ('point (10.02 20.02)', 4326)
);

INSERT INTO y_test VALUES (
 2,
 st_point ('point zm(10.1 20.01 5.0 7.0)', 4326)
);
```

```
SELECT gid, st_y (pt1)
 AS "The Y coordinate"
 FROM y_test;

gid    The Y coordinate

1          20.02
2          20.01
```

The ST_Y function can also be used to update the coordinate value of an existing point. In this example, ST_Y is used to update the y-coordinate value of the second point in y_test.

```
UPDATE y_test
 SET pt1=st_y(
  (SELECT pt1 FROM y_test WHERE gid=2),
  20.1
  )
 WHERE gid=2;
```

# ST_Z

## Definition

ST_Z takes an ST_Point as an input parameter and returns its z- (elevation) coordinate. In SQLite, ST_Z can also update the z-coordinate of an ST_Point.

## Syntax

### Oracle and PostgreSQL

```
sde.st_z (geometry1 sde.st_point)
```

### SQLite

```
st_z (geometry  geometryblob)
st_z (input_shape geometryblob, new_Zvalue double)
```

## Return type

### Oracle

Number

### PostgreSQL

Integer

### SQLite

Double precision is returned when ST_Z is used to return the z-coordinate of a point. A geometryblob is returned when ST_Z is used to update the z-coordinate of a point.

## Example

The z_test table is created with two columns: the id column, which uniquely identifies the row, and the geometry point column. The INSERT statement inserts a row into the z_test table.

The SELECT statement lists the id column and the double-precision z-coordinate of the point inserted with the previous statement.

### Oracle

```
CREATE TABLE z_test (
 id integer unique,
 geometry sde.st_point
);

INSERT INTO z_test (id, geometry) VALUES (
 1,
 sde.st_point (2, 3, 32, 5, 4326)
);
```

```
SELECT id, sde.st_z (geometry) Z_COORD
 FROM Z_TEST;

        ID      Z_COORD

         1        32
```

### PostgreSQL

```
CREATE TABLE z_test (
 id integer unique,
 geometry sde.st_point
);

INSERT INTO z_test (id, geometry) VALUES (
 1,
 sde.st_point (2, 3, 32, 5, 4326)
);
```

```
SELECT id, sde.st_z (geometry)
 AS Z_COORD
 FROM z_test;

        id      z_coord

         1        32
```

### SQLite

```
CREATE TABLE z_test (id integer);

SELECT AddGeometryColumn(
 NULL,
 'z_test',
 'pt1',
 4326,
 'pointzm',
 'xyzm',
 'null'
);

INSERT INTO z_test (id, pt1) VALUES (
 1,
```

```
 st_point (2, 3, 32, 5, 4326)
);
```

```
SELECT id, st_z (pt1)
 AS "The z coordinate"
 FROM z_test;

id      The z coordinate

1        32.0
```

The ST_Z function can also be used to update the coordinate value of an existing point. In this example, ST_Z is used to update the z-coordinate value of the first point in z_test.

```
UPDATE z_test
 SET pt1=st_z(
  (SELECT pt1 FROM z_test where id=1), 32.04)
 WHERE id=1;
```