IBM i
7.5

Database
Geospatial Analytics

Note

Before using this information and the product it supports, read the information in "Notices" on page 83.

This edition applies to IBM i 7.5 (product number 5770-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

Contents

Geospatial Analytics

The Db2® for IBM® i database provides support for geospatial data and functions. The geospatial functions, with IBM Watson, adds industry-leading technology in the form of Geospatial Analytics in Db2 for i. Geospatial Analytics is integrated into Db2 for i. These analytic functions include projection-free ellipsoidal support and native geohashes, allowing you to use SQL to leverage Watson Geospatial technology.

Geospatial Analytics can be used to generate and analyze geospatial information about geographic features and to store and manage the data on which that information is based. A geographic feature is anything in the real world that has an identifiable location. Examples of features are:

- An object, such as a river, forest, or mountain range.
- A space such as a safety zone around a hazardous site or the marketing area serviced by a particular business.
- The location of an event, for example, an auto accident that occurred at a particular intersection, or a sales transaction at a specific store.

Geospatial information refers to facts and figures about the locations of geographic features that the database makes available to its users. Examples of geospatial information are:

- Locations of geographic features on a map, for example, the longitude and latitude of a city.
- The location of geographic features with respect to one another, for example, the location of all hospitals and clinics within a city or the proximity of city residents to a particular earthquake zone.
- Ways in which geographic features are related to each other, for example, information that a certain watershed or bridge is contained within a specific region.
- Measurements that apply to one or more geographic features, for example, the distance between an office building and its lot line, or the length of the perimeter of a wildlife preserve.

Geospatial information, either by itself or in combination with traditional relational data, can help institutions and businesses make decisions on things like choosing areas to provide services or determining the locations of possible markets. For example, suppose the owner of a restaurant chain wants to open new restaurants in nearby cities, and needs to answer to such questions as: Where in these cities are concentrations of the types of people who typically frequent restaurants like mine? Where are the major highways? Where are competing restaurants located? The analysis of geospatial data can help to answer these questions.

What's new for IBM i 7.5

This section highlights the changes made to this topic for IBM® i 7.5.

What's new as of May 2023

"ST_GEOHASHVALUE scalar function" on page 56 scalar function was added.

What's new as of December 2022

This topic is new.

How to see what's new or changed

To help you see where technical changes have been made, the information center uses:

- The » image to mark where new or changed information begins.
- The « image to mark where new or changed information ends.

In PDF files, you might see revision bars (|) in the left margin of new and changed information.

To find other information about what's new or changed this release, see the Memo to users.

PDF file for Geospatial Analytics

You can view and print a PDF file of this information.

To view or download the PDF version of this document, select Geospatial Analytics.

Saving PDF files

To save a PDF on your workstation for viewing or printing:

1. Right-click the PDF link in your browser.
2. Click the option that saves the PDF locally.
3. Navigate to the directory in which you want to save the PDF.
4. Click **Save**.

Downloading Adobe Reader

You need Adobe Reader installed on your system to view or print these PDFs. You can download a free copy from the Adobe Web site (http://get.adobe.com/reader/).

Code license and disclaimer information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

The nature of geospatial data

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

A coordinate is a number that denotes either:

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

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

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

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

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

Geodetic data

Geodetic data is geospatial data that is expressed in latitude and longitude coordinates, using a coordinate system that describes a round, continuous, closed surface.

Geospatial Analytics uses geodetic data which treats the earth as a globe that has no edges or seams at the poles or the dateline. When the earth is treated as flat map, it requires projected coordinates to transform spherical coordinates to planar coordinates. Geodetic data, which treats the earth as a globe, uses latitude and longitude on an ellipsoidal model of the Earth's surface. Calculations such as line intersection, area overlap, distance, and area, are accurate and precise, regardless of location.

Geographic features, geospatial information, geospatial data, and geometries

Some basic concepts underlie the operations of Geospatial Analytics including geographic features, geospatial information, geospatial data, and geometries.

Geospatial Analytics lets you obtain facts and figures that pertain to things that can be defined geographically - that is, in terms of their location on earth, or within a region of the earth. These facts and figures are referred to as **geospatial information**, and the things as **geographic features**, or features for short.

For example, you could use Geospatial Analytics to determine whether any populated areas overlap the proposed site for a landfill. The populated areas and the proposed site are both features. A finding as to whether any overlap exists is an example of geospatial information. If overlap exists, the extent of overlap is another example of geospatial information.

To produce geospatial information, Geospatial Analytics must process data that defines the locations of features. Such data, called **geospatial data**, consists of coordinates that reference the locations on a map or similar projection. For example, to determine whether one feature overlaps another, Geospatial Analytics must determine where the coordinates of one of the features are situated with respect to the coordinates of the other.

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

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

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

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

Representing geographic features in a table

A geographic feature can be represented by data saved in a table.

For example, consider office buildings and residences. In the following sample tables, each row in the BRANCHES table represents a branch office of a bank. Each row in the CUSTOMERS table represents a customer of the bank. A subset of each row—specifically, the address, city, postal code, state, and country —represents the location of the branch office or the customer's residence. The data in these columns, taken as a whole, represents a geographic feature.

Table 1. BRANCHES table

NAME	ADDRESS	CITY	ZIPCODE	STATE	COUNTRY
Airport-Multern	92467 Airport Blvd	San Jose	95141	CA	USA

Table 2. CUSTOMERS table

LASTNAME	FIRSTNAME	ADDRESS	CITY	ZIPCODE	STATE	COUNTRY
Kriner	Endela	9 Concourt Circle	San Jose	95141	CA	USA

The values that denote the branch and customer addresses can be translated into values from which geospatial information is generated. For example, in the CUSTOMER table, one branch office's address is 92467 Airport Blvd, San Jose, CA 95141, USA. In the CUSTOMER table, one customer's address is 9 Concourt Circle, San Jose, CA 95141, USA. These addresses can be translated into coordinate values that indicate where the branches and the customers' homes are located. The coordinate values can then be used to indicate where these locations are with respect to one another. The BRANCH table and CUSTOMER table can be altered to add a column to contain the coordinates that correspond to the address. The following two tables have a new column, LOCATION, that is designated to contain such values.

Table 3. BRANCHES table with a geospatial column

NAME	ADDRESS	CITY	ZIPCODE	STATE	COUNTRY	LOCATION
Airport-Multern	92467 Airport Blvd	San Jose	95141	CA	USA	

Table 4. CUSTOMERS table with a geospatial column

LASTNAME	FIRSTNAME	ADDRESS	CITY	ZIPCODE	STATE	COUNTRY	LOCATION
Kriner	Endela	9 Concourt Circle	San Jose	95141	CA	USA	

Geospatial information is derived from the data stored in the LOCATION column. This data is referred to as geospatial data.

Key concepts

This section describes geometries, coordinate systems, spatial reference systems, geohashing, and supported data formats.

Geometries

For Geospatial Analytics, the operational definition of geometry is "a model of a geographic feature."

In Geospatial Analytics, the model can be expressed in terms of the feature's coordinates. The model conveys information; for example, the coordinates identify the position of the feature with respect to fixed points of reference. Also, the model can be used to produce additional information; for example, the ST_OVERLAPS function can take the coordinates of two regions as input and return information as to whether the regions overlap or not.

The coordinates of a feature that a geometry represents are regarded as properties of the geometry. Several kinds of geometries have other properties as well; for example, area, length, and boundary.

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

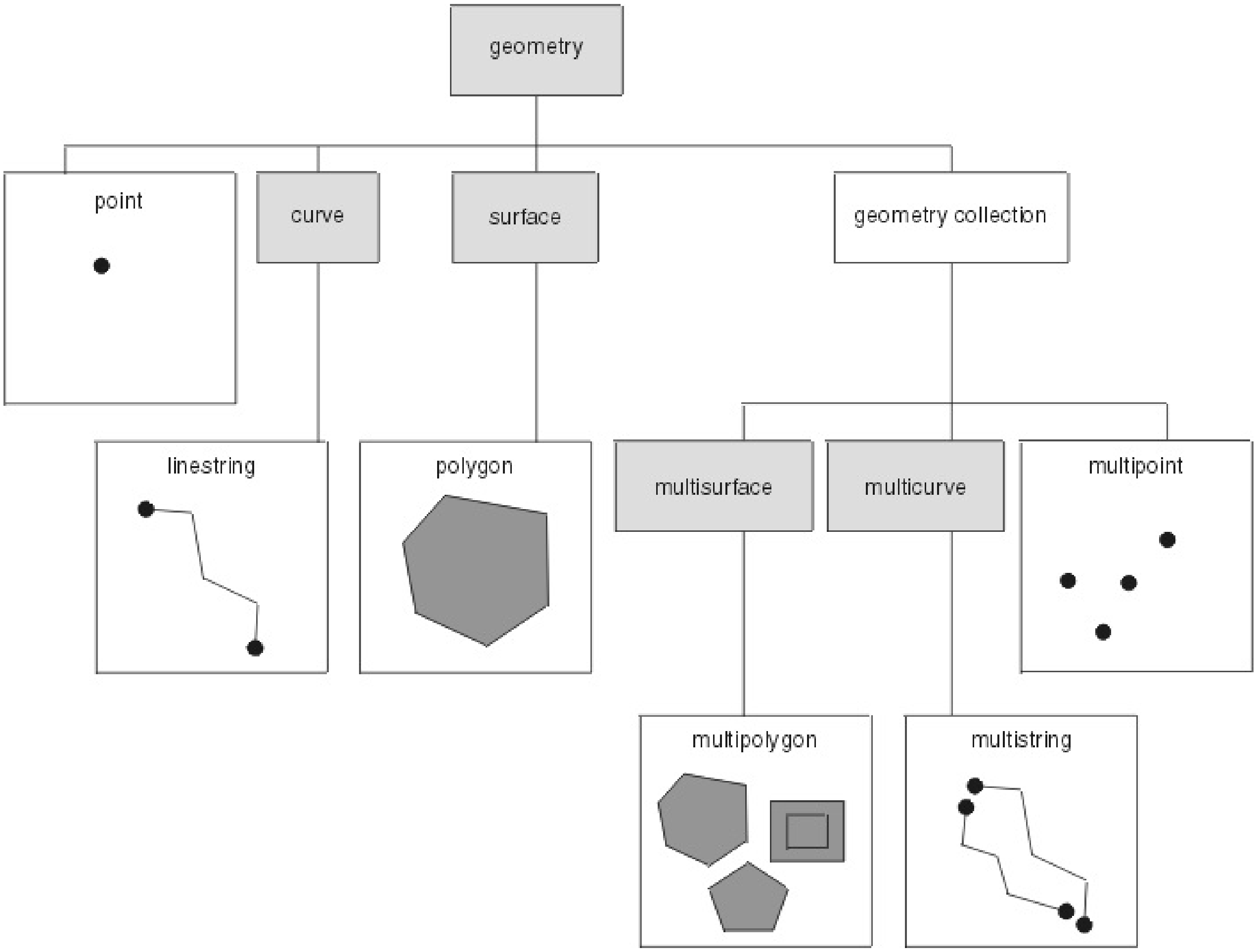


Figure 1. Hierarchy of geometries supported by Geospatial Analytics.

The spatial data types supported by Geospatial Analytics are implementations of the geometries shown in the figure. See "Geospatial data types" on page 17 for details.

As the figure indicates, a superclass called geometry is the root of the hierarchy. The root type and other proper subtypes in the hierarchy are not instantiable.

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

The base geometries are:

Points
A single point. Points represent discrete features that are perceived as occupying the locus where an east-west coordinate line (such as a parallel) intersects a north-south coordinate line (such as a meridian). For example, suppose that the notation on a world map shows that each city on the map is located at the intersection of a parallel and a meridian. A point could represent each city.

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

Polygons
A polygon or surface within a polygon. Polygons represent multisided geographic features such as districts, forests, and wildlife habitats.

Geometry collection subtypes include both homogeneous collections and heterogeneous collections.

The heterogeneous collection is:

Geometry Collection
A geometry collection containing one or more geometry types. Geometry collections represent multipart features with a variety of components such as a group of lakes (polygons) and rivers (linestrings) that form a watershed.

The homogeneous collections are:

Multipoints
A geometry collection containing multiple points. Multipoints represent multipart features whose components are each located at the intersection of an east-west coordinate line and a north-south coordinate line. An example is an island chain whose members are each situated at an intersection of a parallel and a meridian.

Multilinestrings
A geometry collection containing multiple linestrings. Multilinestrings represent multipart features made up of more than one linear component such as river systems and highway systems.

Multipolygons
A geometry collection containing multiple polygons. Multipolygons represent multipart features made up of multisided units or components such as the collective farmlands in a specific region or a system of lakes.

Properties of geometries

The following properties are defined for geometries:

- The type that a geometry belongs to
- Geometry coordinates
- A geometry's interior, boundary, and exterior
- The quality of being simple or non-simple
- The quality of being empty or not empty
- A geometry's minimum bounding rectangle, sometimes called its envelope
- Dimension
- The identifier of the spatial reference system with which a geometry is associated

Geometry coordinates

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

An X coordinate value denotes a location that is relative to a point of reference to the east or west. A Y coordinate value denotes a location that is relative to a point of reference to the north or south. X and Y coordinates are represented as double-precision floating point numbers.

Geospatial Analytics uses (longitude, latitude) ordering.

Simple and non-simple geometries

The values of linestrings, multipoints, and multilinestrings are considered either simple or non-simple.

A geometry is simple if it obeys all the topological rules imposed on its subtype and non-simple if it does not.

- A linestring is simple if it does not pass through the same point twice except when the end points are the same point.
- A multipoint is simple if none of its elements occupy the same coordinate space.
- Points, polygons, multipolygons, and empty geometries are always simple.

Any linestring, multipoint, or multilinestring that does not adhere to the rules for a simple geometry is considered non-simple.

Empty geometries

A geometry is empty if it does not contain any points.

An empty geometry is considered a simple geometry. An empty geometry can only be assigned to the ST_GEOMETRY type.

For example, an empty point is represented by the following WKT: `'point empty'`

WKB cannot represent an empty geometry.

Interior, boundary, and exterior

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

Interior
The interior is the space occupied by the geometry except for the boundary.

Exterior
The exterior of a geometry is all space not occupied by the geometry or the boundary.

Boundary
The boundary of a geometry serves as the interface between its interior and exterior.

These concepts apply to points, linestrings, and polygons with the following rules:

- Point
  - A point has no boundary.
  - The point is the interior.
- Linestring
  - The endpoints are the boundary.
  - The rest of the linestring is the interior.
  - A linestring where the start and end are the same point (forming a linear ring) has no boundary.
- Polygon
  - The boundary is its outer ring and any inner rings.

– The interior consists of the space enclosed by the outer ring, excluding any space defined by inner rings within the polygon.

Minimum bounding rectangle

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

The MBRs of geometries form a boundary rectangle except for the following special cases.

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

Coordinate systems

A coordinate system is a framework for defining the relative locations of things in a specified area; for example, an area on the Earth's surface or the Earth's surface as a whole. Geospatial Analytics supports the Geographic Coordinate System using WGS_1984 datum (GCS_WGS_1984).

Information about the coordinate system can be accessed through the QSYS2.ST_COORDINATE_SYSTEMS catalog view.

Geographic coordinate systems

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

For example, the figure below shows a geographic coordinate system where a location is represented by the coordinates longitude 80 degree East and latitude 55 degree North.

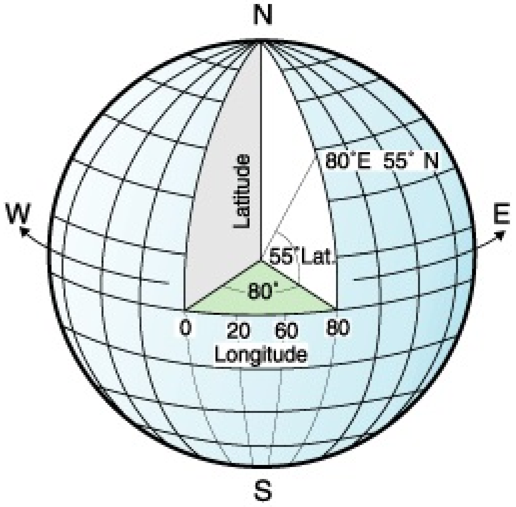


Figure 2. A geographic coordinate system

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

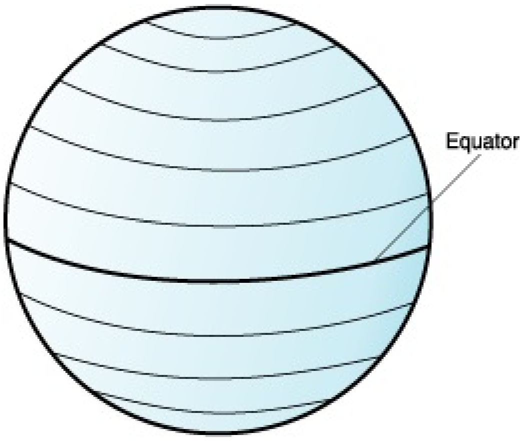


Figure 3. Latitude lines

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

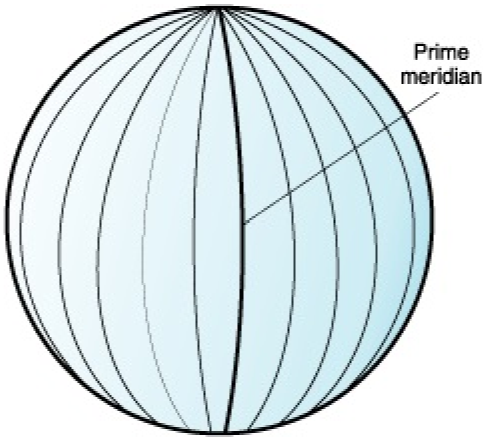


Figure 4. Longitude lines

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

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

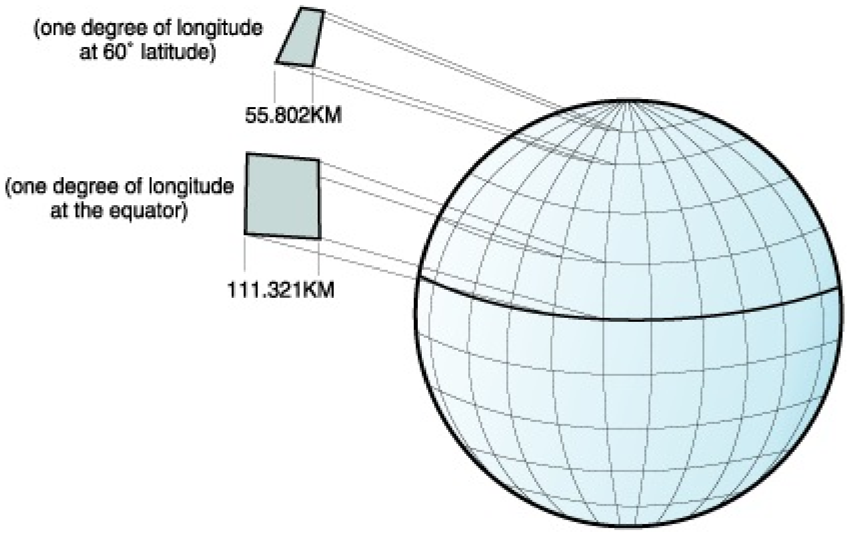


Figure 5. Different dimensions between locations on the graticule

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

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

The figure below shows the sphere and spheroid approximations of the Earth and the major and minor axes of an ellipse.

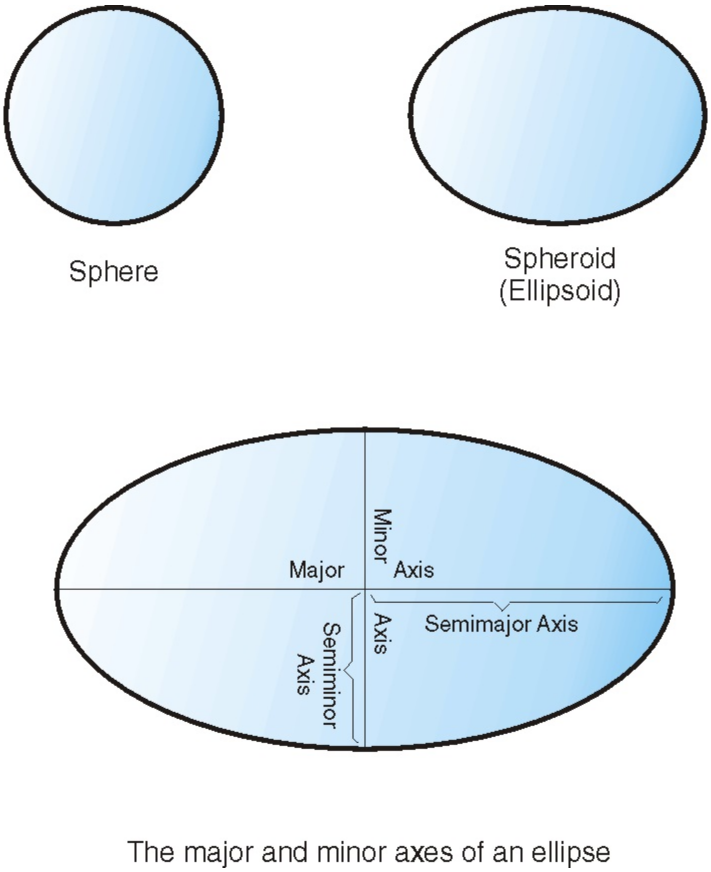


Figure 6. Sphere and spheroid approximations

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

Coordinate system syntax

The coordinate system syntax is a string representation of a coordinate system.

The well-known text (WKT) format of spatial reference systems provides a standard textual representation for coordinate system information. The definitions of the well-known text representation are defined by the OGC Simple Features for SQL specification and the ISO SQL/MM Part 3: Spatial standard.

The WKT for the supported coordinate system for Geospatial Analytics, GCS_WGS_1984, is:

```
GEOGCS["WGS 84",DATUM["WGS_1984",SPHEROID["WGS
84",6378137,298.257223563,AUTHORITY["EPSG","7030"]],AUTHORITY["EPSG","6326"]],
PRIMEM["Greenwich",0,AUTHORITY["EPSG","8901"]],UNIT["degree",0.01745329251994328,AUTHORITY["EPSG
","9122"]],AUTHORITY["EPSG","4326"]]'
```

Supported coordinate system units

The database uses a specific coordinate systems syntax and supported coordinate system values to provide a standard textual representation for coordinate system information.

The default units for the supported coordinate for Geospatial Analytics, GCS_WGS_1984, is as follows:

Supported linear unit

Table 5. Supported linear unit

Unit	Conversion factor
Meter	1.0

Supported angular unit

Table 6. Supported angular unit

Unit	Valid range for latitude	Valid range for longitude	Conversion factor
Decimal Degree	-90 and 90 degrees (inclusive)	-180 and 180 degrees (inclusive)	*pi*/180

Supported spheroid

Table 7. Supported spheroid

Name	Semi-major axis	Inverse flattening
WGS 1984	6378137.0	298.257223563

Supported prime meridian

Table 8. Supported prime meridian

Location	Coordinates
Greenwich	0° 0' 0"

Spatial reference system

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

These parameters are:

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

The numeric identifier for a spatial reference system determines which spatial reference system is used to represent the geometry. The only spatial reference system identifier supported by Geospatial Analytics is SRS ID 4326 (WGS84_SRS_4326).

The spatial reference system is shown in the table below, along with the coordinate system on which the spatial reference system is based.

Table 9. The spatial reference system

Spatial reference system	SRS ID	Coordinate system
WGS84_ SRS_4326	4326	GCS_WGS _1984

Information about the spatial reference system known to the database can be accessed through the QSYS2.ST_SPATIAL_REFERENCE_SYSTEMS catalog view.

Geohashes and geohash covers

You can use geohashes and geohash covers to simplify certain operations.

A *geohash* is a number that uniquely identifies a specific region. The geohash algorithm divides the earth into regions, called *cells*, and converts the latitude and longitude of the center of each cell into a number that uniquely identifies it. The size of each cell is determined by the depth value, which is specified by the user of the algorithm. The smaller the depth value, the larger the cell size.

A *geohash cover* is the set of geohash cells that are needed to completely cover a particular geometry. A larger depth corresponds to smaller cells, which usually results in more exact coverage (that is, a proportionally smaller area of the geohash cover that is outside of the geometry). However, when the cells are smaller, more cells are needed to compose the geohash cover.

A single-point geometry is always associated with a single cell; however, a non-single-point geometry such as a linestring, polygon, multi-polygon, or multi-point geometry might touch several cells.

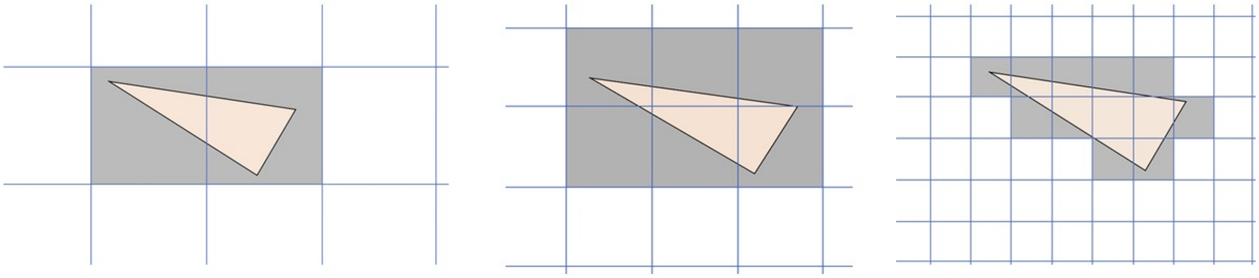


Figure 7. Examples of geohash covers of a single geometry at 3 different depths

The number of geohash values in a geohash cover increases with increasing depth.

The following table lists some of the most commonly used depths their corresponding approximate cell sizes.

Geohash Depth	Approximate Cell Size	Description	Examples
45	0.1 km^2	Single point or address	GPS-location or house
28	3 km^2	Small region	city block
23	100 km^2	Medium-sized region	forest or lake
18	3,000 km^2	Large region	county or postal code area
13	100,000 km^2	Very large (huge) region	state or country

When using geohash covers, the following considerations apply:

- Geohash values of two geometries that are to be compared should be computed using the same depth.
- Carefully consider the depth that you choose when computing geohash values. If the depth that you are using is too large, it can cause many geohash values to be generated. Generating more than 10,000 geohash values for a single geometry is not allowed and returns a failure. If this error is encountered, choose a smaller depth value.

WKT and WKB data formats

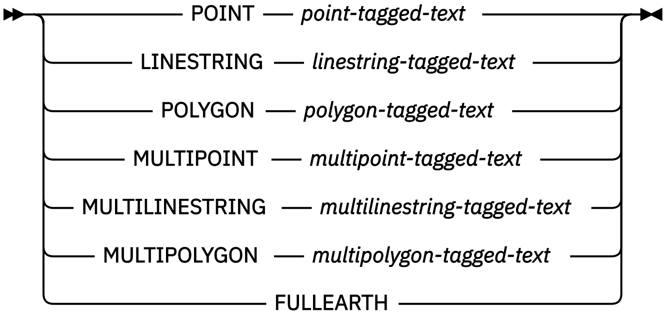
Geospatial Analytics supports industry-standard spatial data formats.

Well-known text (WKT) format

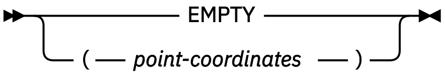
The OpenGIS Consortium Simple Features for SQL specification defines the well-known text format to exchange geometry data in ASCII format. This format is also referenced by the ISO SQL/MM Part: 3 Spatial standard.

The well-known text format of a geometry is defined as follows:

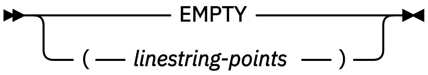
Syntax


POINT — point-tagged-text
LINESTRING — linestring-tagged-text
POLYGON — polygon-tagged-text
MULTIPOINT — multipoint-tagged-text
MULTILINESTRING — multilinestring-tagged-text
MULTIPOLYGON — multipolygon-tagged-text
FULLEARTH

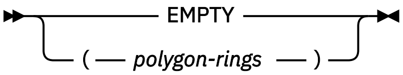
point-tagged-text



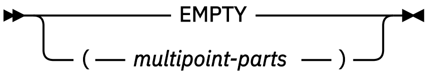
linestring-tagged-text



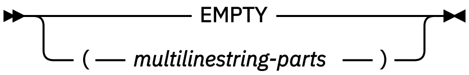
polygon-tagged-text



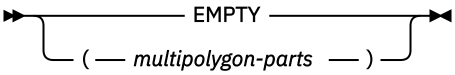
multipoint-tagged-text



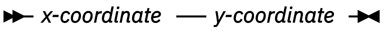
multilinestring-tagged-text



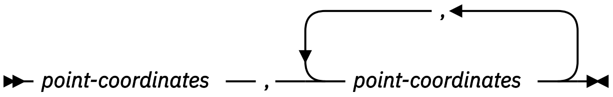
multipolygon-tagged-text



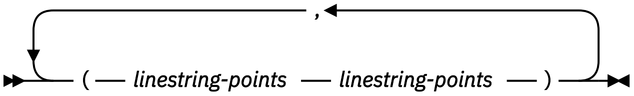
point-coordinates



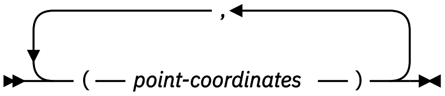
linestring-points



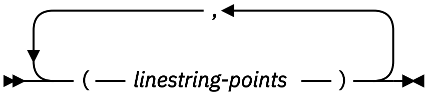
polygon-rings



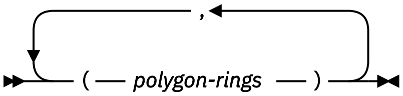
multipoint-parts



multilinestring-parts



multipolygon-parts



x-coordinate
An expression that returns a numeric value that represents the X coordinate of a point.

y-coordinate
An expression that returns a numeric value that represents the Y coordinate of a point.

To indicate an empty geometry, the keyword EMPTY is specified instead of any coordinates. To indicate the full earth, the keyword FULLEARTH is specified. Empty geometry and full earth can only be assigned to the ST_GEOMETRY type.

Examples

The following table provides some examples of well-known text formats.

Table 10. Geometry types and well-known text formats

Geometry type	**WKT format**
point	POINT(10 20)
linestring	LINESTRING(10 10, 20 20, 21 30)
polygon	POLYGON((0 0, 0 40, 40 40, 40 0, 0 0))
multipoint	MULTIPOINT((0 0), (10 20), (15 20), (30 30))
multilinestring	MULTILINESTRING((10 10, 20 20), (15 15, 30 15))
multipolygon	MULTIPOLYGON(((10 10, 10 20, 20 20, 20 15, 10 10)), ((60 60, 70 70, 80 60, 60 60)))

Table 10. Geometry types and well-known text formats (continued)

Geometry type	**WKT format**
geometry collection	GEOMETRYCOLLECTION(POINT (10 10), POINT (30 30), LINESTRING (15 15, 20 20))
empty geometry	POINT EMPTY
full earth	FULLEARTH

Well-known binary (WKB) format

The OpenGIS Consortium Simple Features for SQL specification defines the well-known binary format. This format is also defined by the International Organization for Standardization (ISO) SQL/MM Part: 3 Spatial standard.

The basic building block for well-known binary formats is the byte stream for a point, which consists of two double precision floating point values. The byte streams for other geometries are built using the byte streams for geometries that are already defined.

The following example illustrates the basic building block for well-known binary formats.

```
// Basic Type definitions
// byte : 1 byte
// uint32 : 32 bit unsigned integer  (4 bytes)
// double : double precision number (8 bytes)

// Building Blocks : Point, LinearRing

Point {
  double x;
  double y;
};

LinearRing   {
  uint32  numPoints;
  Point   points[numPoints];
};

enum wkbGeometryType {
  wkbPoint = 1,
  wkbLineString = 2,
  wkbPolygon = 3,
  wkbMultiPoint = 4,
  wkbMultiLineString = 5,
  wkbMultiPolygon = 6
};

enum wkbByteOrder {
  wkbXDR = 0,     // Big Endian
  wkbNDR = 1      // Little Endian
};

WKBPoint {
  byte     byteOrder;
  uint32   wkbType;              // 1=wkbPoint
  Point    point;
};

WKBLineString {
  byte     byteOrder;
  uint32   wkbType;              // 2=wkbLineString
  uint32   numPoints;
  Point    points[numPoints];
};

WKBPolygon    {
  byte              byteOrder;
  uint32            wkbType;     // 3=wkbPolygon
  uint32            numRings;
  LinearRing        rings[numRings];
};

WKBMultiPoint    {
  byte              byteOrder;
  uint32            wkbType;     // 4=wkbMultipoint
  uint32            num_wkbPoints;
  WKBPoint          WKBPoints[num_wkbPoints];
};

WKBMultiLineString    {
  byte              byteOrder;
  uint32            wkbType;     // 5=wkbMultiLineString
  uint32            num_wkbLineStrings;
  WKBLineString     WKBLineStrings[num_wkbLineStrings];
};

wkbMultiPolygon {
  byte              byteOrder;
  uint32            wkbType;     // 6=wkbMultiPolygon
  uint32            num_wkbPolygons;
  WKBPolygon        wkbPolygons[num_wkbPolygons];
};

WKBGeometry  {
  union {
    WKBPoint                 point;
    WKBLineString            linestring;
    WKBPolygon               polygon;
    WKBMultiPoint            mpoint;
    WKBMultiLineString       mlinestring;
    WKBMultiPolygon          mpolygon;
  }
};
```

The following figure shows an example of a geometry in well-known binary format.

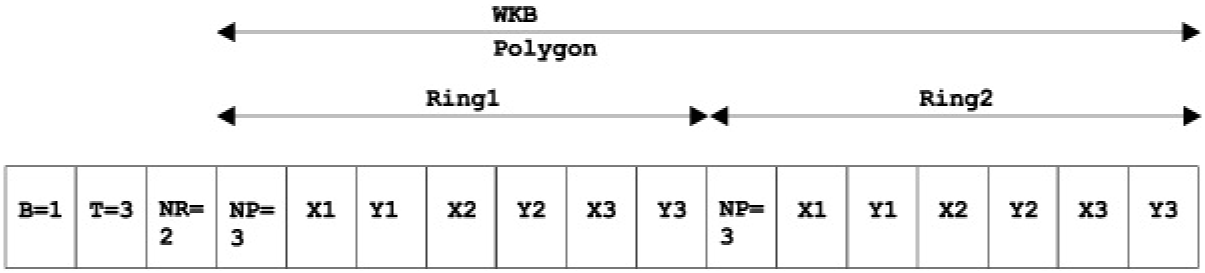


Figure 8. Spatial data in WKB format. (B=1) of type polygon (T=3) with 2 linears (NR=2), where each ring has 3 points (NP=3).

Working with geospatial data

In order to work with geospatial data, create geospatial columns in a table and populate the table with that data. This geospatial data can then be queried using geospatial functions.

Geospatial data types

Geospatial Analytics provides a hierarchy of geospatial data types.

The figure below shows this hierarchy. In this figure, the instantiable types have a white background; the uninstantiable types have a shaded background.

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

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

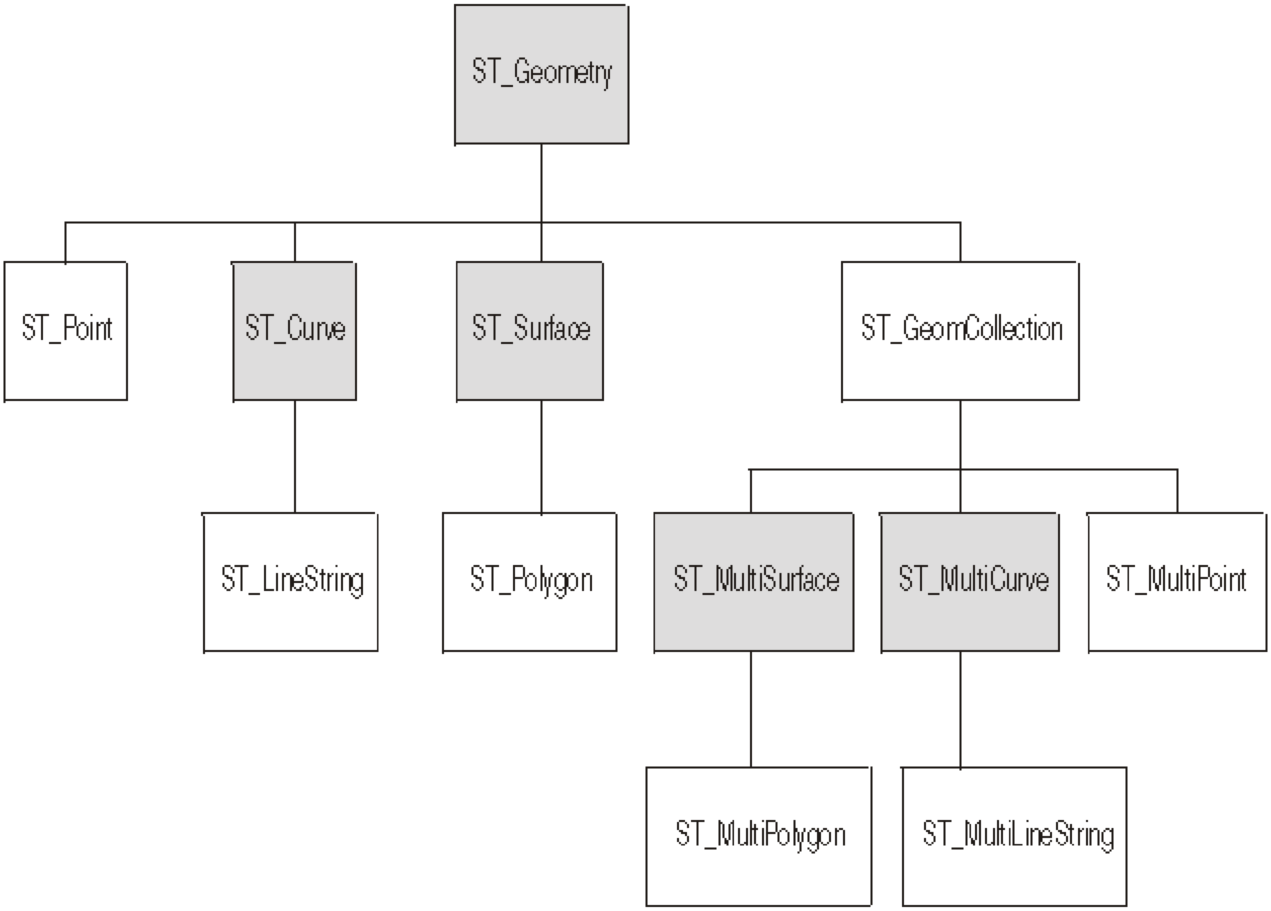


Figure 9. Hierarchy of spatial data types. Data types named in white boxes are instantiable. Data types named in shaded boxes are not instantiable.

The hierarchy includes:

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

Db2 for i provides these data types as user-defined data types in schema QSYS2.

Data types for single-unit features

Use ST_POINT, ST_LINESTRING, and ST_POLYGON to store coordinates that define the space occupied by features that can be perceived as forming a single unit.

ST_POINT

The ST_POINT data type is a zero-dimensional geometry that occupies a single location in coordinate space. Use ST_POINT when you want to indicate the point in space that is occupied by a discrete geographic feature. The feature might be a very small one, such as a water well; a very large one, such as a city; or one of intermediate size, such as a building complex or park.

In each case, the point in space can be located at the intersection of an east-west coordinate line (for example, a parallel) and a north-south coordinate line (for example, a meridian). An ST_POINT data item includes an X coordinate and a Y coordinate that define such an intersection. The X coordinate indicates where the intersection lies on the east-west line; the Y coordinate indicates where the intersection lies on the north-south line.

ST_LINESTRING

The ST_LINESTRING data type is a one-dimensional object stored as a sequence of points defining a linear path. Use ST_LINESTRING for coordinates that define the space that is occupied by linear features; for example, streets, canals, and pipelines.

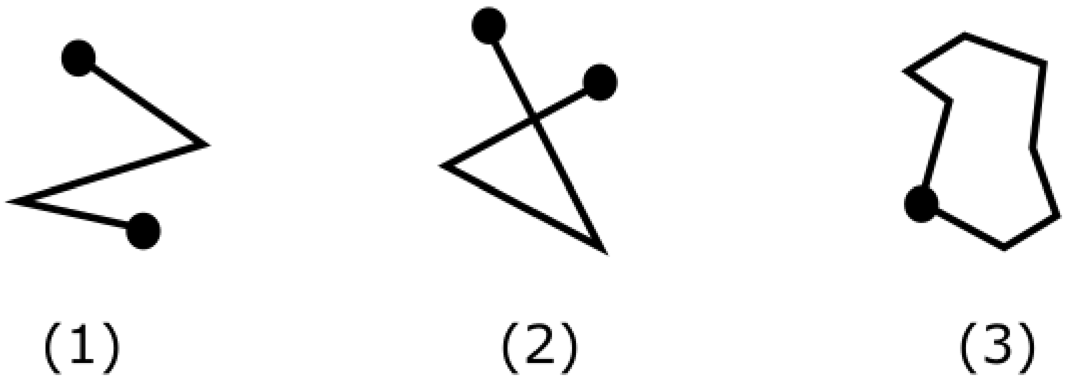


Figure 10. Examples of ST_LINESTRING objects

ST_POLYGON

The ST_POLYGON data type is a two-dimensional surface stored as a sequence of points defining its exterior bounding ring and 0 or more interior rings. Use ST_POLYGON when you want to indicate the extent of space covered by a multi-sided feature; for example, a voting district, a forest, or a wildlife habitat. An ST_POLYGON data item consists of the coordinates that define the boundary of such a feature.

The ST_POLYGON is always simple. The exterior and any interior rings define the boundary of an ST_POLYGON, and the space enclosed between the rings defines the interior of ST_POLYGON. The rings of an ST_POLYGON can intersect at a tangent point, but never cross. ST_POLYGON has an area.

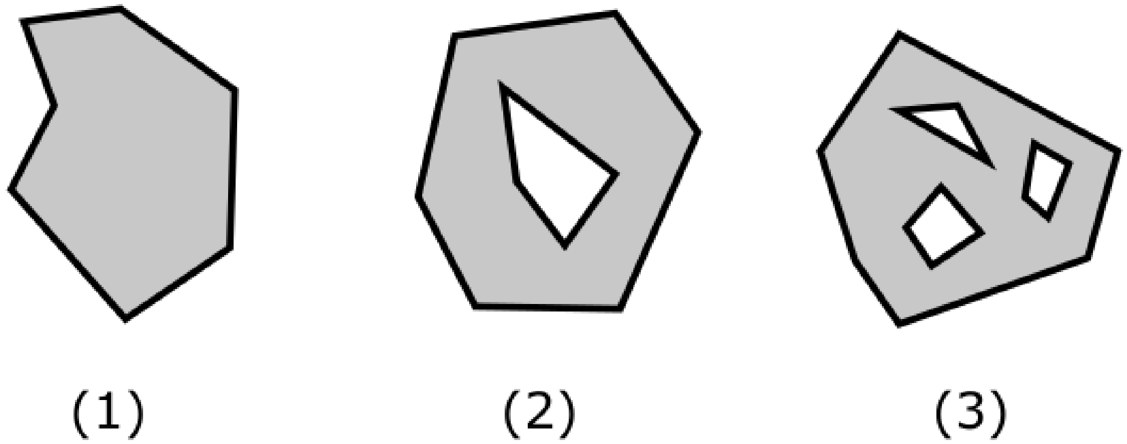


Figure 11. Examples of ST_POLYGON objects

In some cases, ST_POLYGON and ST_POINT can be used for the same feature. For example, suppose that you need spatial information about an apartment complex. If you want to represent the point in space where each building in the complex is located, you would use ST_POINT to store the X and Y coordinates that define each such point. Otherwise, if you want to represent the area occupied by the complex as a whole, you would use ST_POLYGON to store the coordinates that define the boundary of this area.

Data types for multi-unit features

Use ST_MULTIPOINT, ST_MULTILINESTRING, ST_MULTIPOLYGON, and ST_GEOMCOLLECTION to store coordinates that define spaces occupied by features that are made up of multiple units.

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

ST_MULTIPOINT

The ST_MULTIPOINT data type is a collection of ST_POINTs. Use ST_MULTIPOINT when you are representing features made up of units whose locations are each referenced by an X coordinate and a Y coordinate. For example, consider a table whose rows represent island chains. The X coordinate and Y coordinate for each island has been identified. If you want the table to include these coordinates and the coordinates for each chain as a whole, define an ST_MULTIPOINT column to hold these coordinates.

ST_MULTILINESTRING

The ST_MULTILINESTRING data type is a collection of ST_LINESTRINGs. Use ST_MULTILINESTRING when you are representing features made up of linear units, and you want to store the coordinates for the locations of these units and the location of each feature as a whole. For example, consider a table whose rows represent river systems. If you want the table to include coordinates for the locations of the systems and their components, define an ST_MULTILINESTRING column to hold these coordinates.

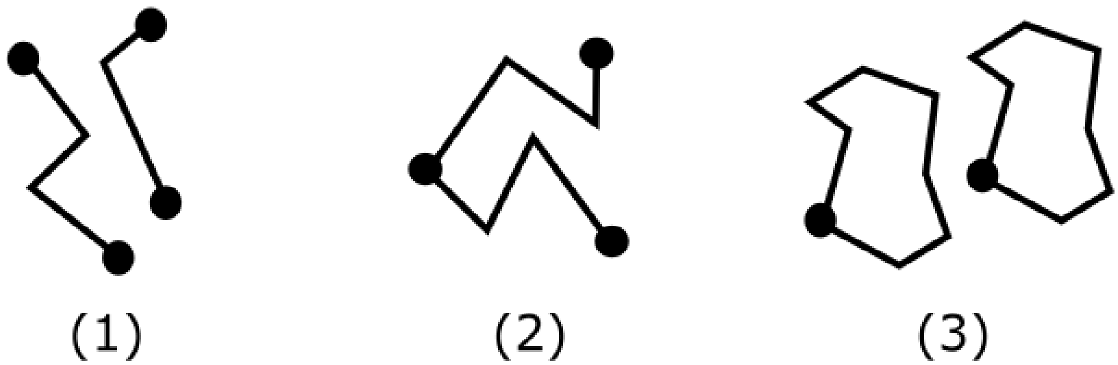


Figure 12. Examples of ST_MULTILINESTRING

ST_MULTIPOLYGON

The ST_MULTIPOLYGON data type is a collection of ST_MULTIPOLYGONS. Use ST_MULTIPOLYGON when you are representing features made up of multi-sided units, and you want to store the coordinates for the locations of these units and the location of each feature as a whole. For example, consider a table whose rows represent rural counties and the farms in each county. If you want the table to include coordinates for the locations of the counties and farms, define an ST_MULTIPOLYGON column to hold these coordinates.

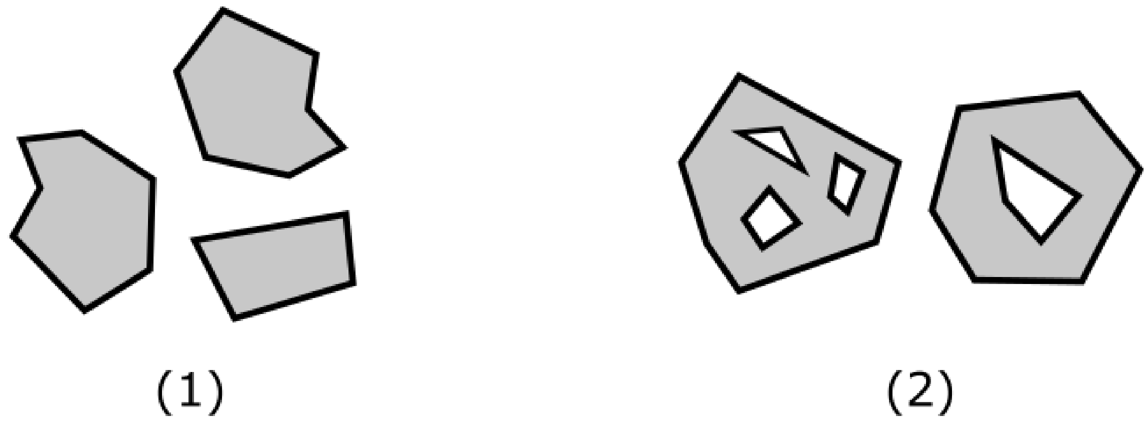


Figure 13. Examples of ST_MULTIPOLYGON

ST_GEOMCOLLECTION

The ST_GEOMCOLLECTION data type is a collection of geometries. Use ST_GEOMCOLLECTION when you are representing features made up of different heterogeneous geometric objects.

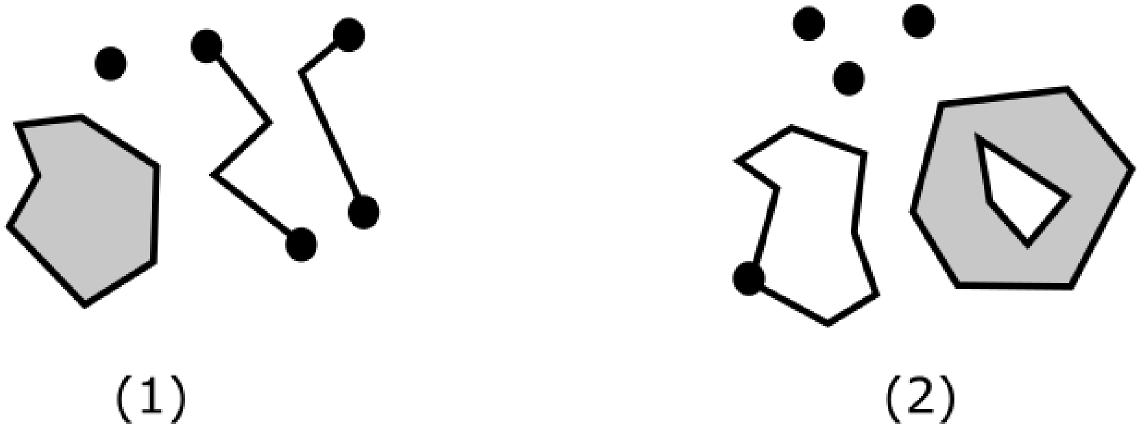


Figure 14. Examples of ST_GEOMCOLLECTION

Data types for all features

You can use ST_GEOMETRY when you are not sure which of the other data types to use.

Because ST_GEOMETRY is the root of the hierarchy to which the other geospatial data types belong, a column defined as the ST_GEOMETRY data type can contain a representation of any of the other geospatial data types.

The actual geospatial type contained in an ST_GEOMETRY column is called the dynamic type. For example, if an instance of an ST_GEOMETRY column contains a representation of a polygon, the dynamic type is ST_POLYGON.

Create a table with a geospatial column

The geospatial data types can be used to create columns for storing geospatial data.

Use the CREATE TABLE SQL statement to create a table with a geospatial column that represents the location of the business.

```
CREATE TABLE BRANCHES
  (ID INT,
   NAME VARCHAR(30),
   ADDRESS VARCHAR(100),
   CITY VARCHAR(50),
   POSTAL_CODE VARCHAR(5),
   STATE CHAR(2),
   COUNTRY VARCHAR(20),
   LOCATION QSYS2.ST_POINT);
```

Alter a table to contain a geospatial column

An existing table can be modified to add a geospatial column to store geospatial data.

Example: Use the ALTER TABLE SQL statement to add a geospatial column to an existing table to represent a customer's location.

```
CREATE TABLE CUSTOMERS
  (ID VARCHAR(7),
   LAST_NAME VARCHAR(30),
   FIRST_NAME VARCHAR(30),
   ADDRESS VARCHAR(100),
   CITY VARCHAR(50),
   POSTAL_CODE VARCHAR(5),
   STATE CHAR(2),
   COUNTRY VARCHAR(20));

ALTER TABLE CUSTOMERS ADD COLUMN LOCATION QSYS2.ST_POINT;
```

The geospatial data types are stored in a non-standard format. If a column contains WKB data, it cannot be directly altered to a Geospatial Analytics data type. The data must be explicitly converted to a geometry using a scalar function such as ST_GEOMETRY. No validation of data occurs when altering a BLOB column to one of the geometry data types, so while a direct alter of the column will be successful, the data in the column will not be usable.

Populating geospatial columns

You can populate geospatial columns with geospatial data that you create using constructor functions or by using other geospatial functions to derive it from existing business data or other spatial data.

Geospatial Analytics provides scalar functions that construct geometries from coordinates, well-known text (WKT), or well-known binary (WKB). Using these functions, you can generate geospatial data to insert into geospatial columns.

To insert geospatial data use the INSERT statement with a constructor function. The constructor functions validates that the WKT or WKB is valid, that the latitude and longitude values passed in are valid and not out of bounds, and that the WKT or WKB text passed in is of the same geometry type as the return value of the constructor. For example, a WKT defining a linestring cannot be passed to the ST_POINT function. If you are unsure which geometry will be passed to the constructor function, the base geometry constructor, ST_GEOMETRY, can be used. ST_GEOMETRY is the only constructor function that accepts an empty geometry.

Example of inserting a ST_POINT geometry type into a geospatial column

The following example creates a geospatial column for ST_POINT values and then inserts two points. The first INSERT statement creates a point geometry from the WKT representation. The second INSERT statement creates a point geometry from numeric coordinate values.

```
CREATE TABLE SAMPLE_POINTS (ID INT, GEOM QSYS2.ST_POINT);

-- The center of Yellowstone National Park in the USA, using WKT
INSERT INTO SAMPLE_POINTS VALUES
  (100, QSYS2.ST_POINT('point (-110.40 44.45)'));

-- The center of Yosemite National Park in the USA, using (longitude, latitude)
INSERT INTO SAMPLE_POINTS VALUES
  (101, QSYS2.ST_POINT(-119.539, 37.865));
```

Attempting to insert a linestring or any other geospatial type which is not a point when using the ST_POINT constructor function results in an SQL error.

Example of inserting different geometry types into a geospatial column

The following example creates a geospatial column that can contain any ST_GEOMETRY value. Then it inserts a point value using WKT and a polygon value also using WKT.

```
CREATE TABLE SAMPLE_GEOMETRY (ID INT, GEOM QSYS2.ST_GEOMETRY);

-- A point defining the center of Yellowstone National Park in the USA
INSERT INTO SAMPLE_GEOMETRY VALUES
  (100,
   QSYS2.ST_POINT('point (-110.40 44.45)'));

-- A polygon defining the approximate boundary of the area of Yellowstone National Park
INSERT INTO SAMPLE_GEOMETRY VALUES
  (200,
   QSYS2.ST_POLYGON('polygon ((-111.1259 45.1207, -110.0009 45.0659, -109.8106 44.7134,
-109.9895 44.1307,
                               -111.0974 44.1269, -111.1719 45.0544, -111.1259 45.1207))'));
```

Returning geospatial data in well-known formats

To return geospatial data in an industry-standard spatial data format, use the ST_ASTEXT or ST_ASBINARY scalar functions.

Example 1: Converting geometry objects to well-known text (WKT) format

Converting geometry objects to WKT formats allow geometries to be exchanged in text form.

The ST_ASTEXT function converts a geometry value to a WKT string. The following example selects values from the SAMPLE_GEOMETRY table. The query returns CLOB values containing UTF-8 character strings.

```
SELECT id, QSYS2.ST_ASTEXT(geom) AS geometry_wkt
  FROM sample_geometry;

ID       GEOMETRY_WKT
------   --------------------------------
   100   POINT (-110.39999999999999 44.449999999999996)
   200   POLYGON ((-111.1259 45.1207, -111.1719 45.0544, -111.0974 44.1269,
                   -109.98949999999999 44.1307, -109.8106 44.7134, -110.0009 45.0659,
                   -111.1259 45.1207))
```

Example 2: Converting geometry objects to Well-known binary (WKB) format

Converting geometry objects to WKB formats allow geometries to be exchanged in binary form. The WKB format consists of binary data structures that must be BLOB values. These BLOB values represent binary data structures that must be managed by a database application program written in a supported programming language and for which there is a language binding.

The ST_ASBINARY function converts a geometry value to the WKB format. The following example selects values from the SAMPLE_GEOMETRY table.

```
SELECT id, QSYS2.ST_ASBINARY(geom) AS geometry_wkb
  FROM sample_geometry;

ID       GEOMETRY_WKB
------   --------------------------------
   100   0000000001C05B9999999999994046399999999999
   200   00000000030000000100000007C05BC80EBEDFA44040468F7318FC5048C05BCB0068DB8BAC
         404686F694467382C05BC63BCD35A8584046103E425AEE63C05B7F53F7CED916404610BAC7
         10CB29C05B73E0DED288CE40465B50B0F27BB3C05B800EBEDFA4404046886F69446738C05B
         C80EBEDFA44040468F7318FC5048
```

Using geospatial functions

Db2 for i provides functions that perform various operations on geospatial data. Many of these functions can be categorized according to the type of operation that they perform.

This section lists the main functional categories and provides an example of each one.

Table 11. Geospatial function operations

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

Example 1: Derives new geometries from existing ones

In this example, the function ST_BUFFER derives a geometry representing a store's sales area from a geometry representing the store's location. The sales area is a circle of 10 km (10000 meters) around the store location.

```
UPDATE stores
  SET sales_area = QSYS2.ST_BUFFER(location, 10000)
  WHERE id = 10;
```

Example 2: Returns information about specific geometries

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

```
SELECT QSYS2.ST_AREA(sales_area)
  FROM stores
  WHERE id = 10;
```

Example 3: Makes comparisons

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

```
SELECT c.first_name, c.last_name, QSYS2.ST_WITHIN(c.location, s.sales_area)
  FROM customers as c. stores AS s
  WHERE s.id = 10;
```

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

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

```
INSERT INTO customer (id, first_name, last_name, location)
VALUES ( 123, 'Mary', Smith', QSYS2.ST_POINT('point (-92.503 44.058)') );
```

Working with geospatial data in embedded SQL programs

When interacting with geospatial data types in embedded SQL, host variables should be declared as either BLOB or VARBINARY data types.

Geospatial data types do not have an equivalent host language data type. The appropriate host variable data type to use is either a BLOB or a VARBINARY host variable. The rules for defining these variables are described in the Embedded SQL Programming topic: Embedded SQL Programming.

Example

In an ILE RPG program using embedded SQL, add the location of a customer to a column in the CUSTOMER table. First, the geospatial location is calculated and saved in a VARBINARY host variable. Then that location is updated in the appropriate row in the CUSTOMER table.

```
    Dcl-S PointValue    SQLTYPE(VARBINARY:1000);

   exec sql
     set :PointValue = QSYS2.ST_POINT('point (45 30)');
   exec sql
     update customer set location = :pointvalue
           where cust_id = 17;

   return;
```

Performance tuning

There are several measures you can take to improve the performance of applications that use spatial data.

Filtering

Identifying relationships between geometries, such as distance, intersections, and containment, requires Cartesian joins that typically need extensive calculations. When the tables contain large numbers of objects, this can result in long execution times. To improve performance, you can use filters to reduce the number of objects that need to be processed.

Sometimes, the data required by a filter is already available in a table. For example, in a table of geospatial objects that contains the postal code of each object, you might employ a filter to compare only objects that share the same postal code. However, if the data needed for filtering is not available, it can be calculated and added to an additional column or table that contains numeric values that represent the geometries. You can then add predicates to your queries to filter data based on these values.

The numeric values that represent a geometry are based on a geohash. There are several different types of geohash:

Geohash
A geohash is a number that uniquely identifies a specific region. A point resides in a single region and is represented by a single geohash value.

Geohash cover
A geohash cover is the set of geohash cells that are needed to completely cover a particular geometry. A line or a polygon can cross multiple regions and can therefore be represented by a set of geohash cover values.

Minimum bounding rectangle (MBR)
The MBR of a geometry is the bounding geometry formed by the minimum and maximum (X,Y) coordinates. A geohash minimum bounding rectangle cover is the set of geohash cells that are needed to cover the MBR completely.

When determining whether to use a geohash cover or a minimum bounding rectangle, consider the following:

- A geohash cover is more precise then a geohash MBR. Because of this more geometries may be filtered out before calling the more time consuming geospatial functions.
- The creation of a geohash MBR is faster then the creation of a geohash cover. Therefore, it takes less time to generate the geohash values when using a geohash MBR.

Filtering using a geohash cover

You can use a geohash cover to filter geometry objects when querying geospatial data, which will greatly improving query performance.

A *geohash cover* is the set of geohash cells that are needed to completely cover a particular geometry. The ST_GEOHASH, ST_GEOHASHCOVER, and ST_GEOHASHCOVEREXTEND table functions can be used to generate a geohash, a geohash cover, or extend a geohash cover by a given distance.

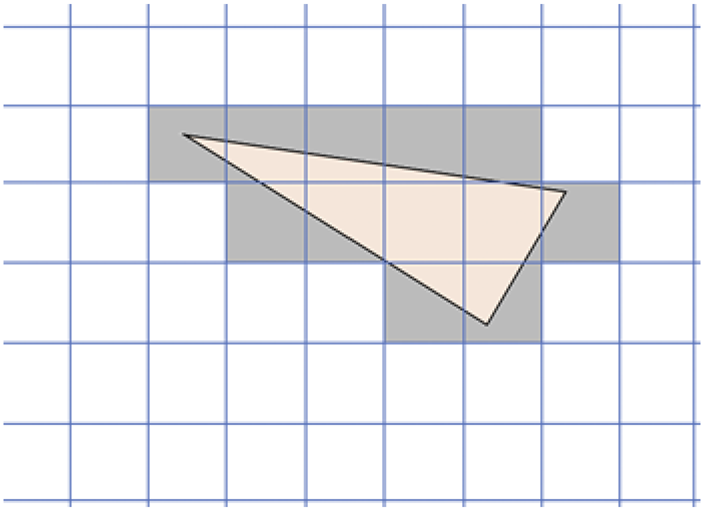


Figure 15. Geohash cover

You can use geohash covers to quickly determine whether it is possible that two geometries intersect:

- If their geohash covers share at least one cell, their maximum separation is the length of one cell diagonal. They might intersect (see Figure 16 on page 27), or they might not (see Figure 17 on page 27).

- However, if their geohash covers do not share any cells, you can be certain that they do not intersect (see Figure 18 on page 28). If geometries do not intersect, they can be filtered out of the query before intensive spatial functions are run, improving performance.

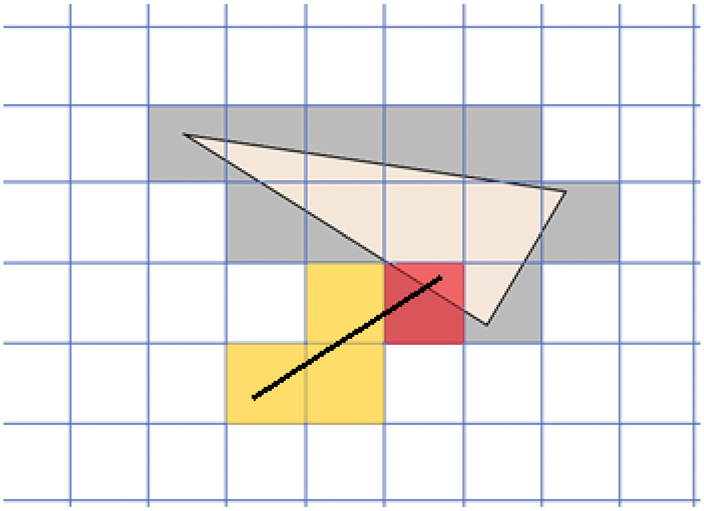


Figure 16. The geohash covers of the two geometries share a common cell, and the geometries intersect

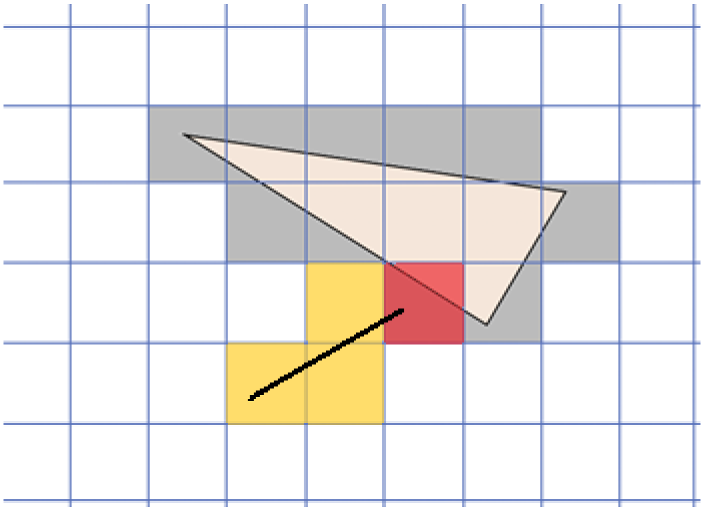


Figure 17. The geohash covers of the two geometries share a common cell, but the geometries do not intersect

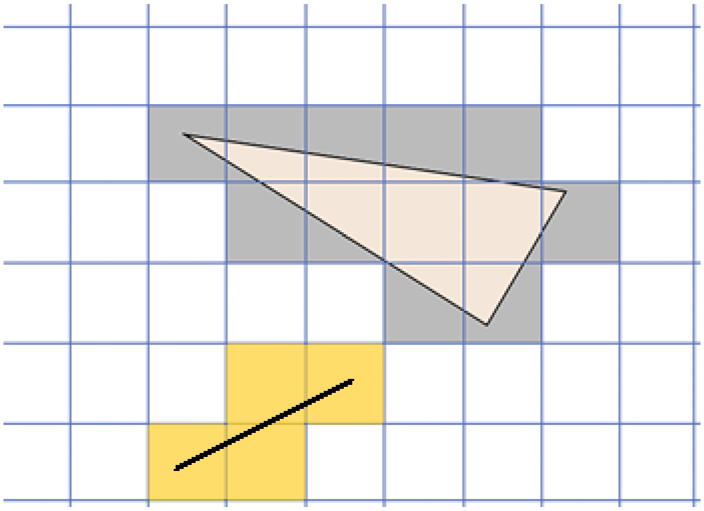


Figure 18. The geohash covers of the two geometries do not share any cells

Filtering using a minimum bounding rectangle

One way to filter objects for spatial processing is by using a minimum bounding rectangle.

A *geohash minimum bounding rectangle cover* is the set of geohash cells that are needed to cover the minimum bounding rectangle (MBR) completely. The ST_FUZZYGEOHASHCOVER and ST_FUZZYGEOHASHCOVEREXTEND table functions can be used to a create a geohash MBR cover or extend a geohash MBR cover by a given distance.

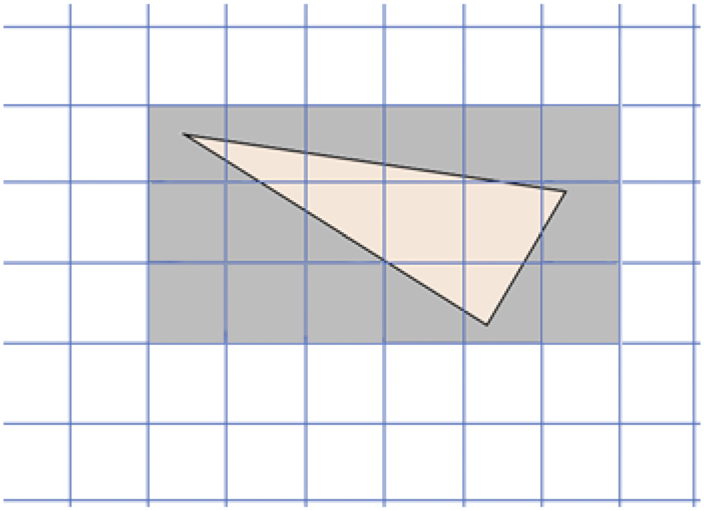


Figure 19. Geohash MBR cover

You can use fuzzy geohash covers to quickly determine whether it is possible that two geometries intersect:

- If the fuzzy geohash covers share at least one cell, their maximum separation is the length of one cell diagonal. They might intersect (see Figure 20 on page 29), or they might not (see Figure 21 on page 29).
- However, if their geohash covers do not share any cells, you can be certain that they do not intersect (see Figure 22 on page 30). If geometries do not intersect, they can be filtered out of the query before intensive spatial functions are run, improving performance.

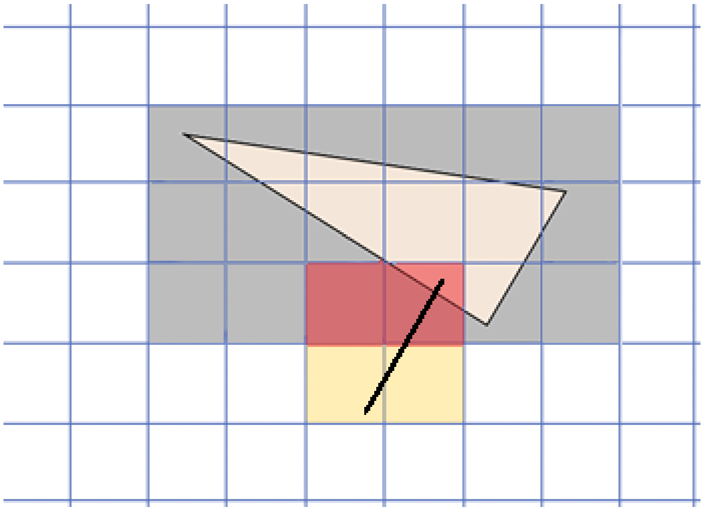


Figure 20. The fuzzy geohash covers of the two geometries share two common cells, and the geometries intersect

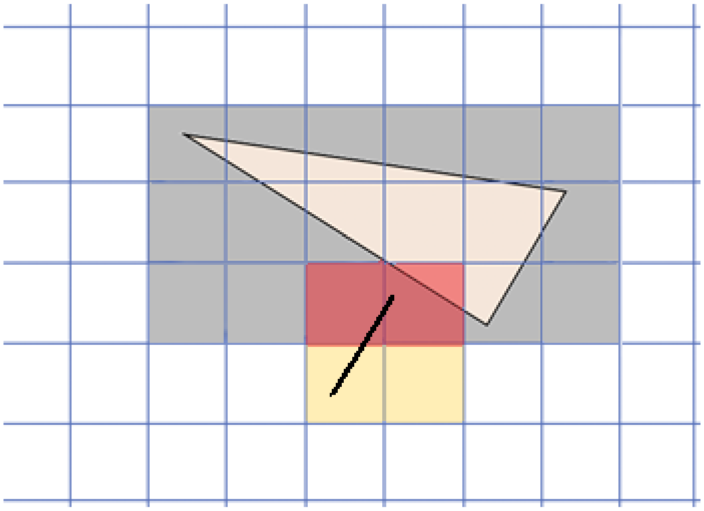


Figure 21. The fuzzy geohash covers of the two geometries share two common cells, but the geometries do not intersect

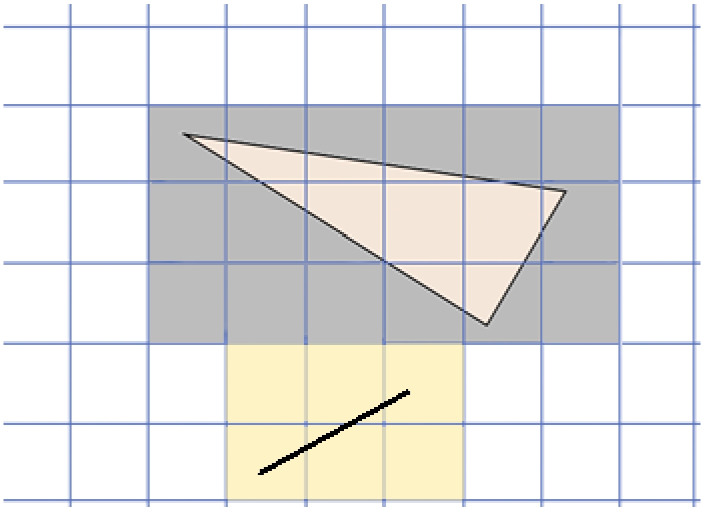


Figure 22. The fuzzy geohash covers of the two geometries do not share any cells

Examples of filtering using a geohash

These examples demonstrate how to improve query performance by filtering using geohashs.

Example 1

For this example, geohash filtering will be added to a query to improve performance.

The tables used in the example are the US_STATES table and the WALKING_PATHS table. The US_STATES table contains a row for each state in the United States. The WALKING_PATHS table contains a row for every walking path in a national park in the United States. These tables are defined as follows:

```
CREATE TABLE US_STATES (STATE_ID CHAR(2) PRIMARY KEY,
                        STATE_FULL_NAME VARCHAR(50),
                        STATE_GEO QSYS2.ST_POLYGON);

CREATE TABLE WALKING_PATHS (WALKING_ID VARCHAR(10) PRIMARY KEY,
                            WALKING_GEO QSYS2.ST_LINESTRING);
```

The following query is being enhanced to use geohashes. It returns a row for every walking path that is completely contained inside the state of Minnesota (MN) in the United States.

```
SELECT WALKING_ID,
       STATE_FULL_NAME,
       QSYS2.ST_ASTEXT(WALKING_GEO),
       QSYS2.ST_ASTEXT(STATE_GEO)
  FROM US_STATES S, WALKING_PATHS W
  WHERE S.STATE_ID = 'MN'
    AND QSYS2.ST_CONTAINS(STATE_GEO, WALKING_GEO) = 1;
```

1. First, note the primary key and geometry column of each of the base tables.

   - In US_STATES, the primary key is STATE_ID and the geometry column is STATE_GEO which contains a ST_POLYGON that represent the geometry of the state.
   - In WALKING_PATHS, the primary key is WALKING_ID and the geometry column is WALKING_GEO which contains a ST_LINESTRING that represents the geometry of the path.

   The primary key will be used to enforce a one to many relationship between the base table and the geohash table we are about to create. This guarantees all the entries in the geohash table have a parent row in the base table. While a primary key is not required, it is a best practice for a relational data model.

2. Determine the size of the geometries to be covered, and use this information to decide which depth to use. The geohash covers of the two geometries that are compared must be computed using the same depth. For this example, a depth of 13 is selected. This is the recommended depth when working with a very large region, like a state.
3. Create a geohash filter table that corresponds to each geospatial column in a base table. Define a foreign key to enforce the dependency of the rows in the geohash table to the corresponding row in the base table.

   Use the ST_GEOHASHCOVER table function to populate each of the geohash filter tables with the geohash values that correspond to the geohash covers of the geometries. Multiple geohash values are generally returned for each geohash cover that is generated. This means that multiple rows will be inserted into the table for each row containing a geometry value in the base table.

```
-- Create and populate the geohash values that correspond to the US_STATES table
CREATE TABLE US_STATES_HASH (STATE_ID CHAR(2),
                             GEOHASH BIGINT,
                             FOREIGN KEY (STATE_ID) REFERENCES US_STATES(STATE_ID));
INSERT INTO US_STATES_HASH (SELECT S.STATE_ID, T.GEOHASH
                              FROM US_STATES S,
                                   TABLE(QSYS2.ST_GEOHASHCOVER(S.STATE_GEO, 13)) T);

-- Create and populate the geohash values that correspond to the WALKING_PATHS table
CREATE TABLE WALKING_PATHS_HASH (WALKING_ID VARCHAR(10),
                                GEOHASH BIGINT,
                                FOREIGN KEY (WALKING_ID) REFERENCES
WALKING_PATHS(WALKING_ID));
INSERT INTO WALKING_PATHS_HASH (SELECT P.WALKING_ID, T.GEOHASH
                                 FROM WALKING_PATHS P,
                                      TABLE(QSYS2.ST_GEOHASHCOVER(P.WALKING_GEO, 13)) T);
```

4. Rewrite the original query to use the filter tables to reduce the amount of data that needs to be compared by the ST_CONTAINS scalar function. The tables containing the geohash values are joined to the corresponding base tables. An additional predicate is added to the WHERE clause so only rows where there is a state geohash value equal to a walking path geohash are compared by the ST_CONTAINS function.

```
SELECT W.WALKING_ID,
       STATE_FULL_NAME,
       QSYS2.ST_ASTEXT(WALKING_GEO) AS PATH_GEOMETRY,
       QSYS2.ST_ASTEXT(STATE_GEO) AS STATE_GEOMETRY
  FROM US_STATES S JOIN US_STATES_HASH SH ON S.STATE_ID = SH.STATE_ID,
       WALKING_PATHS W JOIN WALKING_PATHS_HASH WH ON W.WALKING_ID = WH.WALKING_ID
  WHERE S.STATE_ID = 'MN' AND
        SH.GEOHASH = WH.GEOHASH AND
        QSYS2.ST_CONTAINS(STATE_GEO, WALKING_GEO) = 1;
```

Example 2

In this example, we want to find every city (a point) that is less than 10 kilometers in distance from a certain trail (a linestring).

The table used in the example is the US_CITIES table, which contains a row for each city in the United States. The geometry used to represent a city is an ST_POINT. Since a point is represented by a single geohash value, the geohash value can be stored in an additional column in the table. The create table statement for the US_CITIES table is:

```
CREATE TABLE US_CITIES (CITY_ID VARCHAR(10),
                        CITY_NAME VARCHAR(50),
                        STATE_ID CHAR(2),
                        CITY_GEO QSYS2.ST_POINT,
                        CITY_GEOHASH BIGINT);
```

There are several ways to populate the CITY_GEOHASH column. For this example, the geohash will use a depth of 23, approximately the size of a forest or lake. Here are a few ways to generate the geohash value.

1. To set geohash values for every row in a table that has a point value defined for a city but does not yet have a geohash value calculated, use an UPDATE statement to set the CITY_GEOHASH column.

   ```
   UPDATE US_CITIES SET CITY_GEOHASH = (SELECT GEOHASH FROM
   TABLE(QSYS2.ST_GEOHASH(CITY_GEO,23)))
                               WHERE CITY_GEOHASH IS NULL;
   ```

2. A trigger can be used to populate the CITY_GEOHASH column when a new city is inserted into the US_CITIES table. The following create trigger statement uses the value assigned to the CITY_GEO column to generate the corresponding geohash value. Once this trigger has been created, every future insert into the US_CITIES table will fire this trigger, assigning a value to the CITY_GEOHASH column as part of the insert operation.

   ```
   CREATE TRIGGER GEOHASH_CITIES
     BEFORE INSERT ON US_CITIES
     REFERENCING NEW AS N FOR EACH ROW
     BEGIN
       SET N.CITY_GEOHASH = (SELECT GEOHASH FROM TABLE(QSYS2.ST_GEOHASH(N.CITY_GEO,23)));
     END;
   ```

The other construct that is used in this example is an ST_LINESTRING global variable which will contain the trail value. The variable is defined like this:

```
CREATE VARIABLE TRAIL QSYS2.ST_LINESTRING;
```

At some point in the application, the global variable is assigned a linestring value that corresponds to a specific state trail. This is not a piece of information that is stored in a table, so no geohash value is permanently associated with it.

```

SET TRAIL = QSYS2.ST_LINESTRING('linestring (-92.51864276919596 44.05903689443092,
                                             -92.57146126853864 44.10851586848976,
                                             -92.64386916754349 44.20339334725808)');
```

For this example, filtering will be added to the following query to improve performance. The query returns a row for every city that is less than 10 kilometers from the trail that is contained in the TRAIL global variable.

```
SELECT * FROM US_CITIES C
  WHERE QSYS2.ST_DISTANCE(C.CITY_GEO, TRAIL) < 10000;
```

1. A geohash value is available in the US_CITIES table for each city. We know it was calculated with a depth of 23.
2. The global variable is a linestring and needs a geohash cover to compare with the geohash value for each city. To have a meaningful compare, the depth for the geohash cover must also use a depth of 23. The geohash cover values for the linestring will be generated when the query is run.
3. Rewrite the original query to use the CITY_GEOHASH column to reduce the amount of data that needs to be considered for the ST_DISTANCE calculation. For this example, an exact geohash value for the linestring is not needed, so the faster ST_FUZZYGEOHASHCOVEREXTEND table function is used to generate the geohash minimum bounding rectangle to use as the filter. The EXTEND version of the function is used to get the geohash regions that form a 10 kilometer buffer around the linestring for the trail. The extended geohash values are needed since the query wants to find all cities within a 10 kilometer distance of the trail.

   The rewrite of the query is as follows:

   ```
   WITH TRAIL_GEOHASH AS
      (SELECT * FROM TABLE(QSYS2.ST_FUZZYGEOHASHCOVEREXTEND(TRAIL, 23, 10000)))
   SELECT * FROM US_CITIES C,
                 TRAIL_GEOHASH T
   WHERE C.CITY_GEOHASH = T.GEOHASH AND
      QSYS2.ST_DISTANCE(C.CITY_GEO, TRAIL) < 10000;
   ```

Best practices and considerations

When using Geospatial Analytics, there are certain things you need to consider.

Best practices

1. When inserting or updating geospatial data for a column defined as a geometry type, the constructor functions should be used to enforce data correctness. For example, to insert a value into a column defined as an ST_POINT data type, use a statement like the following:

   ```
   INSERT INTO mytable (geopoint) VALUES ST_POINT('point (30 40)');
   ```

   Without using these functions, data that does not conform to the geometry type can be inserted into the column.
2. Geometry type arguments passed to geospatial functions expect properly formatted geometry values. Use a constructor function to guarantee the value is a correctly formatted geometry value. For example, when working with data in a global variable, use statements like the following to ensure the content of `myline_gvar` is correctly formed:

   ```
   CREATE VARIABLE myline_gvar QSYS2.ST_LINESTRING;
   SET myline_gvar = QSYS2.ST_LINESTRING('linestring(10 10, 20 20)');
   SELECT * FROM mytable t where QSYS2.ST_INTERSECTS(t.linestring_column, myline_gvar) = 1;
   ```

3. When using an untyped parameter marker or the NULL value, use the CAST specification to provide a data type for the parameter marker or NULL value that function resolution can use to find the correct function. For more information, see Using parameter markers or the NULL values as function arguments
4. The geospatial functions and geospatial data types reside in the QSYS2 library. If you use an unqualified function or geospatial data type, the SQL path is used to locate the object. When using unqualified geospatial functions and data types, ensure QSYS2 is included in the SQL path. For more information, see Unqualified function, procedure, specific name, type, and variables.
5. Most of the system-provided geospatial functions are defined with the MODIFIES SQL DATA attribute. When creating a procedure or function that uses one of these functions, the MODIFIES SQL DATA attribute needs to be included in the routine definition.

Considerations

Java™ environment

Because the geospatial functions use functionality provided by Java, a Java environment is created in the current job. This requires the following conditions.

1. A JVM must not already exist in the job (with the exception of a JVM created by the Java stored procedures support).
2. The job CCSID cannot be 65535.
3. PASE must be installed and operational. The CHKPRDOPT PRDID(5770SS1) OPTION(33) CL command can be used to verify that PASE is installed.
4. The Geospatial functions are implemented via LANGUAGE JAVA functions and always run in the default activation group, ACTGRP(*DFTACTGRP). Therefore, these functions should not be used within programs that were built with ACTGRP(*NEW).

Well-Known Text (WKT) Considerations

There are cases where WKT is adjusted to a preferred representation.:

- **Polygon WKT and the right-hand rule**

  The right-hand rule is used to determine the ordering of points. Polygons are defined by a linear ring that describes the polygon's boundary. On a globe, the same boundary can be used to represent two different polygons depending which side of the linear ring is the interior of the polygon and which side of the linear ring is the exterior of the polygon. Consider a polygon that describes London and a polygon that describe the full Earth except for London. These two polygons would have the same boundary but different interiors. In order to determine which side of the boundary is the interior of the polygon you are trying to define, the "right-hand rule", as defined by RFC 7946: The GeoJSON format, is used. This rule states that for exterior rings, points must be defined in a counterclockwise direction, and for holes, points must be defined clockwise. Since many existing data sets have polygons defined in the incorrect order due to lack of awareness of this rule, Geospatial Analytics adjusts the order of the points. It assumes the area of interest is smaller than half of the full Earth.

- **Multipoint WKT**

  If the same point is defined multiple times in a multipoint geometry, the duplicate point is removed.

Geospatial functions

At the core of Geospatial Analytics are the scalar functions and table functions provided as part of Db2 for i.

The implementation of these geospatial functions follows the "OGC Simple Features for SQL" specification and parts of the ISO SQL/MM Part 3: Spatial standard.

The geospatial functions fall into the following categories:

- Construction of geometries from data exchange formats or coordinate data
- Conversion of a geometry into a data exchange format
- Comparison of geometries and discovery of relations between geometries
- Construction of new geometries from existing geometries
- Information about geometries
- Creation of geohashes

Construction of geometries from data exchange formats or coordinate data

The following scalar functions create a geometry of the ST_GEOMETRY type or one of its subtypes from coordinate data, or from a well-known text (WKT), or a well-known binary (WKB) data exchange format:

Table 12. Geospatial scalar functions that construct geometries from data exchange formats or coordinate data

Function Use	**Function Name**
Construct any geometry from a WKT, WKB	“ST_GEOMETRY scalar function” on page 57
Construct any geometry from a WKT object	“ST_WKTTOSQL scalar function” on page 79
Construct any geometry from a WKB object	“ST_WKBTOSQL scalar function” on page 78
Construct a specific geometry from a WKT or WKB	“ST_GEOMCOLLECTION scalar function” on page 57 “ST_LINESTRING scalar function” on page 62 “ST_MULTILINESTRING scalar function” on page 65 “ST_MULTIPOINT scalar function” on page 66 “ST_MULTIPOLYGON scalar function” on page 67 “ST_POINT scalar function” on page 68 “ST_POLYGON scalar function” on page 69
Construct a specific geometry from coordinates	“ST_POINT scalar function” on page 68

Conversion of a geometry to a different geometry type

The following scalar functions convert one geometry type to a different geometry type:

Table 13. Geospatial scalar functions that convert one geometry type to a different geometry type

Function Use	Function Name
Convert a geometry into a linestring geometry	"ST_TOLINESTRING scalar function" on page 72
Convert a geometry into a multiline geometry	"ST_TOMULTILINE scalar function" on page 72
Convert a geometry into a multipoint geometry	"ST_TOMULTIPOINT scalar function" on page 73
Convert a geometry into a multipolygon geometry	"ST_TOMULTIPOLYGON scalar function" on page 74
Convert a geometry into a point geometry	"ST_TOPOINT scalar function" on page 74
Convert a geometry into a polygon geometry	"ST_TOPOLYGON scalar function" on page 75

Conversion of a geometry into a data exchange format

The following scalar functions convert a geometry of the ST_GEOMETRY type or one of its subtypes into a data exchange format:

Table 14. Geospatial scalar functions that convert a geometry of the ST_GEOMETRY type or one of its subtypes into a data exchange format

Function Use	Function Name
Convert a geometry into a WKT object	"ST_ASTEXT scalar function" on page 38
Convert a geometry into a WKB object	"ST_ASBINARY scalar function" on page 38

Comparison of geometries and discovery of relations between geometries

These geospatial scalar functions return information that is the result of a comparison between geometries. They return information about ways in which geographic features relate to one another or compare with one another.

Table 15. Geospatial scalar functions that compare geometries and find relations between geometries

Function Use	Function Name
Check whether two items geometries are identical	"ST_EQUALS scalar function" on page 46
Determine the distance between geometries	"ST_DISTANCE scalar function" on page 45
Determine whether geometries intersect	"ST_CROSSES scalar function" on page 43 "ST_DISJOINT scalar function" on page 44 "ST_INTERSECTS scalar function" on page 60 "ST_OVERLAPS scalar function" on page 68 "ST_TOUCHES scalar function" on page 76
Determine whether a geometry contains another one	"ST_CONTAINS scalar function" on page 40 "ST_COVERS scalar function" on page 42 "ST_WITHIN scalar function" on page 77

Construction of new geometries from existing geometries

The following scalar functions modify properties of a geometry of type ST_GEOMETRY or one of its subtypes to construct a new geometry:

Table 16. Geospatial scalar functions that construct new geometries from existing geometries

Function Use	Function Name
Create new geometries with different space configurations	"ST_BUFFER scalar function" on page 39 "ST_DIFFERENCE scalar function" on page 44 "ST_INTERSECTION scalar function" on page 59 "ST_SYMDIFFERENCE scalar function" on page 71
Create a new geometry by combining multiple geometries	"ST_UNION scalar function" on page 77

Information about geometries

The following scalar functions return information about geometric properties such as coordinates, measures, and boundaries:

Table 17. Geospatial scalar functions that return information about properties of geometries

Function Use	Function Name
Return information about geometry types	"ST_GEOMETRYTYPE scalar function" on page 58
Return information to indicate whether a geometry is simple	"ST_ISSIMPLE scalar function" on page 61
Return information about geometry dimensions	"ST_AREA scalar function" on page 37
Return information about geometry definitions	"ST_NUMPOINTS scalar function" on page 67
Return information about coordinates and measures	"ST_ISVALID scalar function" on page 62 "ST_MAXX scalar function" on page 63 "ST_MAXY scalar function" on page 63 "ST_MINX scalar function" on page 64 "ST_MINY scalar function" on page 65
Return information about spatial reference systems	"ST_SRSID scalar function" on page 70 "ST_SRSNAME scalar function" on page 70

Creation of geohashes

The following table functions create geohashes

Table 18. Geospatial table functions that create geohash data

Function Use	Function Name
Create the geohash value for a specific point	"ST_GEOHASHVALUE scalar function" on page 56 "ST_GEOHASH table function" on page 51

Table 18. Geospatial table functions that create geohash data (continued)

Function Use	Function Name
Create the set of geohash cells that are needed to completely cover a particular geometry	"ST_GEOHASHCOVER table function" on page 52 "ST_GEOHASHCOVEREXTEND table function" on page 53
Create the set of geohash cells that are needed to create a boundary box around a particular geometry	"ST_FUZZYGEOHASHCOVER table function" on page 47 "ST_FUZZYGEOHASHCOVEREXTEND table function" on page 48

ST_AREA scalar function

The ST_AREA function takes a geometry object as an input parameter and returns a double precision floating point number containing the area covered by the specified geometry, measured in square meters.

If *geometry* is a polygon or multipolygon, the area covered by the geometry is returned. The area of a point, linestring, multipoint, or multilinestring is 0 (zero). If *geometry* is null, the result is the null value. If *geometry* is an empty geometry, the result is 0.

▶▶─ ST_AREA ── (── *geometry* ──) ─▶◀

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that determines the area.

The result of the function is DOUBLE.

Example

Find the area covered by multiple sales regions. The sales regions are polygons stored in the SAMPLE_POLYGON table. The area is calculated by applying the ST_AREA function.

```
CREATE TABLE sample_polygons (sales_region INTEGER, geometry QSYS2.ST_POLYGON);

INSERT INTO sample_polygons (sales_region, geometry) VALUES
  (1, QSYS2.ST_POLYGON('polygon((0 0, 0 10, 10 10, 10 0, 0 0))')),
  (2, QSYS2.ST_POLYGON('polygon((20 0, 30 20, 40 0, 20 0 ))')),
  (3, QSYS2.ST_POLYGON('polygon((20 30, 25 35, 30 30, 20 30))'));
```

The following SELECT statement retrieves the sales region ID and area.

```
SELECT sales_region, QSYS2.ST_AREA(geometry) as area
 FROM sample_polygons;
```

Results:

```
SALES_REGION   AREA
------------   ---------------------
           1   1.2359653226125652E12
           2   2.5100581570625015E12
           3   2.6341713503349957E11
```

ST_ASBINARY scalar function

The ST_ASBINARY function takes a geometry object as an input parameter and returns a BLOB containing the corresponding well-known binary (WKB) format.

If *geometry* is null, the result is the null value. There is no WKB representation for an empty geometry. If *geometry* is empty, an error is returned.

▸▸ ST_ASBINARY — (— *geometry* —) ▸◂

geometry
A value of type ST_GEOMETRY or one of its subtypes to be converted to the corresponding WKB format.

The result of the function is BLOB(2G).

Example

Insert the WKB value of a point into a table. Return its value in WKT format.

```
CREATE TABLE sample_points(id INTEGER, wkb BLOB(32K));

INSERT INTO sample_points values (10, QSYS2.ST_ASBINARY(QSYS2.ST_POINT(10, 20)));

SELECT id, QSYS2.ST_ASTEXT(QSYS2.ST_POINT(wkb)) as point, wkb FROM sample_points;
```

Results:

```
ID     POINT                WKB
----   -------------------  --------------------------------------------
  10   POINT (10.0 20.0)    000000000140240000000000000040340000000000000
```

ST_ASTEXT scalar function

The ST_ASTEXT function takes a geometry object as an input parameter and returns a CLOB containing the corresponding well-known text (WKT) format.

If *geometry* is null, the result is the null value. If *geometry* is empty, "POINT EMPTY" is returned.

▸▸ ST_ASTEXT — (— *geometry* —) ▸◂

geometry
A value of type ST_GEOMETRY or one of its subtypes to be converted to the corresponding WKT format.

The result of the function is CLOB(2G).

Example

View the well-known text representation of a variety of geometries. The query lists the WKT representation of the geometries by converting the geometry to text using the ST_ASTEXT function.

```
CREATE TABLE sample_geometries(id INTEGER, geometry QSYS2.ST_GEOMETRY);

INSERT INTO sample_geometries VALUES
  (1, QSYS2.ST_POINT('point(50 50)')),
  (2, QSYS2.ST_LINESTRING('linestring(20 10, 21 13, 22 14)')),
  (3, QSYS2.ST_POLYGON('polygon((-10 -12, -11 -14, -13 -14, -13 -12, -10 -12))'));

SELECT id, QSYS2.ST_GEOMETRYTYPE(geometry) AS geospatial_text, QSYS2.ST_ASTEXT(geometry) AS wkt
  FROM sample_geometries;
```

Results:

```
ID  GEOSPATIAL_TEXT  WKT
--  ---------------- ----------------------------------------------------------------------
 1  ST_POINT         POINT (50.0 50.0)
 2  ST_LINESTRING    LINESTRING (20.0 10.0, 21.0 13.0, 22.0 14.0)
 3  ST_POLYGON       POLYGON ((-10.0 -12.0, -13.0 -12.0, -13.0 -14.0, -11.0 -14.0,
                               -10.0 -12.0))
```

ST_BUFFER scalar function

The ST_BUFFER function takes a geometry object and a distance in meters as input parameters and returns a new resulting geometry where each point on the boundary of the resulting geometry is the specified distance away from the specified geometry.

Any circular curve in the boundary of the resulting geometry is approximated by linear strings. For example, the buffer around a point, which would result in a circular region, is approximated by a polygon whose boundary is a linestring. The default number of polygon edges to be used in approximating a circle is eight.

If *geometry* is null, the result is the null value. An empty geometry cannot be buffered. If *geometry* is empty, an error is returned.

►► ST_BUFFER (*geometry* , *distance*) ►◄

geometry
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry to create the buffer around.

distance
A double precision floating point value that specifies the distance to be used for the buffer around *geometry*. This distance is measured in meters.

The result of the function is ST_GEOMETRY.

Example

Apply a buffer of 10 kilometers to a variety of geometries.

```
CREATE TABLE sample_geometries(id INTEGER, geometry QSYS2.ST_GEOMETRY);

INSERT INTO sample_geometries VALUES
  (1, QSYS2.ST_POINT('point(50 50)')),
  (2, QSYS2.ST_LINESTRING('linestring(20 10, 21 13, 22 14)')),
  (3, QSYS2.ST_POLYGON('polygon((-10 -12, -11 -14, -13 -14, -13 -12, -10 -12))'));

SELECT id, QSYS2.ST_GEOMETRYTYPE(geometry) AS spatial_text,
QSYS2.ST_ASTEXT(QSYS2.ST_BUFFER(geometry, 10000)) AS buffer_10k
  FROM sample_geometries;
```

Results:

```
ID     SPATIAL_TEXT     BUFFER_10K
----   ---------------  --------------------------------------------------------------------
   1   ST_POINT         POLYGON ((49.942004 50.089817,
                                  49.860139 50.037124999999996,
                                  49.860355 49.962706999999995,
                                  49.94222 49.910154,
                                  50.05778 49.910154,
                                  50.13964499999994 49.962706999999995,
                                  50.13986099999996 50.037124999999996,
                                  50.05799599999996 50.089817,
                                  49.942004 50.089817))
   2   ST_LINESTRING    POLYGON ((20.924315999999997 13.063383,
                                  20.919014 13.047649,
                                  20.907192 13.035749,
                                  20.907564 13.013665,
                                  19.913235999999998 10.027735999999999,
                                  19.901574999999998 9.992343,
                                  19.935903 9.926048,
                                  20.007758 9.903068,
                                  20.075072 9.936854,
                                  20.086748999999998 9.972241,
                                  21.080956 12.952326999999999,
                                  22.066225 13.937234,
                                  22.093018999999998 13.963849,
                                  22.092132 14.038262999999999,
                                  22.037253999999997 14.090266999999999,
                                  21.960532 14.089376999999999,
                                  21.933739 14.062748,
                                  20.944551 13.07335,
                                  20.924315999999997 13.063383))
   3   ST_POLYGON       POLYGON ((-9.946252999999999 -11.925657,
                                  -9.962472 -11.909965,
                                  -9.979156 -11.910055,
                                  -9.994169 -11.902935,
                                  -10.015716 -11.91025,
                                  -12.961767 -11.910366999999999,
                                  -12.961972 -11.910166,
                                  -12.999749999999999 -11.910167999999999,
                                  -13.037528 -11.909965,
                                  -13.037735999999999 -11.910166,
                                  -13.038027999999999 -11.910166,
                                  -13.064748 -11.936299,
                                  -13.091617 -11.962287,
                                  -13.091619 -11.962572999999999,
                                  -13.091826 -11.962776,
                                  -13.091838 -11.99974,
                                  -13.092056999999999 -12.036705,
                                  -13.091852 -12.036907,
                                  -13.092582 -13.999794,
                                  -13.092758 -14.036812,
                                  -13.092595999999999 -14.03697,
                                  -13.092597 -14.037192,
                                  -13.065621 -14.063378,
                                  -13.038753999999999 -14.089671,
                                  -13.038525 -14.089671,
                                  -13.038364 -14.089827999999999,
                                  -13.000197 -14.089832,
                                  -11.021272 -14.089919,
                                  -11.006274999999999 -14.097042,
                                  -10.984385 -14.089768,
                                  -10.961246 -14.089671,
                                  -10.949489999999999 -14.078166999999999,
                                  -10.933695 -14.072913999999999,
                                  -10.923525999999999 -14.052754,
                                  -10.907242 -14.036812,
                                  -10.907319 -14.020617,
                                  -9.923986 -12.052576,
                                  -9.907943 -12.036705,
                                  -9.90804 -12.02038,
                                  -9.900763999999999 -12.005688,
                                  -9.908251 -11.984605,
                                  -9.908382999999999 -11.962287,
                                  -9.92025 -11.950809999999999,
                                  -9.925723999999999 -11.935388999999999,
                                  -9.946252999999999 -11.925657))
```

ST_CONTAINS scalar function

The ST_CONTAINS function takes two geometry objects as input parameters and returns the integer 1 if the first geometry completely contains the second geometry. Otherwise, it returns the integer 0 (zero) to indicate that the first geometry does not completely contain the second.

The ST_CONTAINS function returns the exact opposite result of the ST_WITHIN function.

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* or *geometry2* is empty, 0 (zero) is returned. If both *geometry1* and *geometry2* are empty, 0 is returned.

▶▶ ST_CONTAINS (*geometry1* , *geometry2*) ▶◀

geometry1
: A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is to be tested to completely contain *geometry2*.

geometry2
: A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is to be tested to be completely within *geometry1*.

The result of the function is INTEGER.

Notes

One geometry (*geometry1*) contains another (*geometry2*) if the interiors of the geometries intersect and the interior or boundary of *geometry2* does not intersect the exterior of *geometry1*.

The figure below shows examples of ST_CONTAINS:

- A multipoint geometry contains a point or multipoint geometry when all of the points are within the first geometry.
- A polygon geometry contains a multipoint geometry when all of the points are either on the boundary of the polygon or in the interior of the polygon.
- A linestring geometry contains a point, multipoint, or linestring geometry when all of the points are within the first geometry.
- A polygon geometry contains a point, linestring or polygon geometry when the second geometry is in the interior of the polygon.

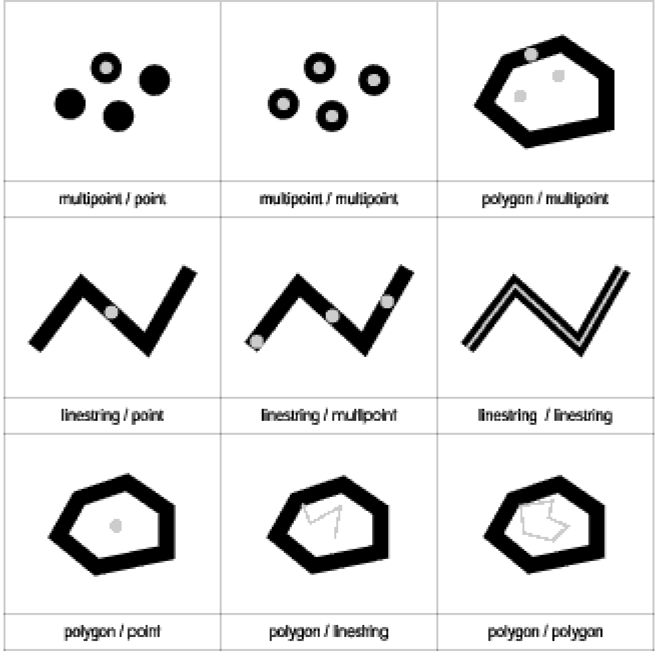


Figure 23. ST_CONTAINS. The dark geometries represent geometry1 and the gray geometries represent geometry2. In all cases, geometry1 contains geometry2 completely.

Example

Use the ST_CONTAINS function to determine which points are contained by a polygon.

```
CREATE TABLE sample_points(point_id INTEGER, point QSYS2.ST_POINT);
CREATE TABLE sample_polygons(polygon_id INTEGER, polygon QSYS2.ST_POLYGON);

INSERT INTO sample_points VALUES
  (1, QSYS2.ST_POINT(10, 20)),
  (2, QSYS2.ST_POINT('point(41 41)'));

INSERT INTO sample_polygons VALUES
```

```
  (100, QSYS2.ST_POLYGON('polygon((0 0, 0 40, 40 40, 40 0, 0 0))'));

SELECT polygon_id,
       point_id,
       CASE QSYS2.ST_CONTAINS(polygon, point)
          WHEN 0 THEN 'does not contain'
          WHEN 1 THEN 'contains'
       END AS contains
FROM sample_points, sample_polygons;
```

Results:

```

POLYGON_ID    POINT_ID    CONTAINS
-----------   ---------   -----------------
        100           1   contains
        100           2   does not contain
```

ST_COVERS scalar function

The ST_COVERS function takes two geometry objects as input parameters and returns the integer 1 if the first geometry completely covers the second geometry. Otherwise, it returns the integer 0 (zero) to indicate that the first geometry does not completely cover the second.

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* or *geometry2* is empty, 0 (zero) is returned. If both *geometry1* and *geometry2* are empty, 1 is returned.

▶▶— ST_COVERS — (— *geometry1* — , — *geometry2* —) —▶◀

geometry1
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is to be tested to completely contain *geometry2*.

geometry2
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is to be tested to be completely covered by *geometry1*.

The result of the function is INTEGER.

Notes

One geometry (*geometry1*) covers another (*geometry2*) if any of the following conditions are true:

- the interior of *geometry1* intersects the interior of *geometry2* and the interior or boundary of *geometry2* does not intersect the exterior of *geometry1*.
- the interior of *geometry1* intersects the boundary of *geometry2* and the interior or boundary of *geometry2* does not intersect the exterior of *geometry1*.
- the boundary of *geometry1* intersects the interior of *geometry2* and the interior or boundary of *geometry2* does not intersect the exterior of *geometry1*.
- the boundaries of either geometry intersect and the interior or boundary of *geometry2* does not intersect the exterior of *geometry1*.

Example

Use the ST_COVERS function to determine which points are covered by a polygon.

```
CREATE TABLE sample_points(point_id INTEGER, point QSYS2.ST_POINT);
CREATE TABLE sample_polygons(polygon_id INTEGER, polygon QSYS2.ST_POLYGON);

INSERT INTO sample_points VALUES
  (1, QSYS2.ST_POINT(10, 20)),
  (2, QSYS2.ST_POINT('point(41 41)'));

INSERT INTO sample_polygons VALUES
  (100, QSYS2.ST_POLYGON('polygon((0 0, 0 40, 40 40, 40 0, 0 0))'));
```

```
SELECT polygon_id,
       point_id,
       CASE QSYS2.ST_COVERS(polygon, point)
          WHEN 0 THEN 'does not cover'
          WHEN 1 THEN 'covers'
       END AS covers
FROM sample_points, sample_polygons;
```

Results:

```
POLYGON_ID    POINT_ID    COVERS
-----------   ---------   -----------------
        100           1   covers
        100           2   does not cover
```

ST_CROSSES scalar function

The ST_CROSSES function takes two geometry objects as input parameters and returns the integer 1 if the first geometry crosses the second. Otherwise, the integer 0 (zero) is returned.

If the intersection of the two geometries results in a geometry that has a dimension that is one less than the maximum dimension of the two geometries, and if the resulting geometry is not equal to either of the two geometries, then 1 is returned. Otherwise, the result is 0 (zero).

If *geometry1* is a polygon or a multipolygon, or if *geometry2* is a point or multipoint, the result is the null value. If *geometry1* or *geometry2* is null, the result is the null value. When null processing doesn't apply, if *geometry1* or *geometry2* is empty, 0 (zero) is returned.

```
►►─ ST_CROSSES ── ( ── geometry1 ── , ── geometry2 ── ) ─►◄
```

geometry1
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is to be tested for crossing *geometry2*.

geometry2
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is to be tested to determine if it is crossed by *geometry1*.

The result of the function is INTEGER.

Example

Use the ST_CROSSES function to determine if a linestring crosses a polygon.

```
CREATE TABLE sample_linestrings(linestring_id INTEGER, linestring QSYS2.ST_LINESTRING);
CREATE TABLE sample_polygons(polygon_id INTEGER, polygon QSYS2.ST_POLYGON);

INSERT INTO sample_linestrings VALUES
 (10, QSYS2.ST_LINESTRING('linestring(40 50, 50 40)')),
 (20, QSYS2.ST_LINESTRING('linestring(20 20, 60 60)'));

INSERT INTO sample_polygons VALUES
 (100, QSYS2.ST_POLYGON('polygon((30 30, 30 50, 50 50, 50 30, 30 30))'));

SELECT linestring_id, polygon_id, QSYS2.ST_CROSSES(linestring, polygon) AS crosses
  FROM sample_linestrings, sample_polygons;
```

Results

```
LINESTRING_ID   POLYGON_ID   CROSSES
-------------   ----------   ---------
           10          100           0
           20          100           1
```

ST_DIFFERENCE scalar function

The ST_DIFFERENCE function takes two geometry objects as input parameters and returns the part of the first geometry that does not intersect with the second geometry.

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* is empty, an empty geometry is returned. If *geometry2* is empty, *geometry1* is returned.

Syntax

▶▶─ ST_DIFFERENCE ── (── *geometry1* ── , ── *geometry2* ──) ─▶◀

geometry1
A value of type ST_GEOMETRY that represents the first geometry to use to compute the difference to *geometry2*.

geometry2
A value of type ST_GEOMETRY that represents the second geometry that is used to compute the difference to *geometry1*.

The result of the function is ST_GEOMETRY.

Example 1

Find the difference between two disjoint polygons.

```
VALUES QSYS2.ST_ASTEXT(
   QSYS2.ST_DIFFERENCE(QSYS2.ST_POLYGON('polygon((10 10, 10 20, 20 20, 20 10, 10 10))'),
                       QSYS2.ST_POLYGON('polygon((30 30, 30 50, 50 50, 50 30, 30 30))')));
```

Results:

```
00001
------------------------------------------------------------------
POLYGON ((10.0 10.0, 20.0 10.0, 20.0 20.0, 10.0 20.0, 10.0 10.0))
```

Example 2

Find the difference between two intersecting polygons.

```
VALUES QSYS2.ST_ASTEXT(
   QSYS2.ST_DIFFERENCE(QSYS2.ST_POLYGON('polygon((30 30, 30 50, 50 50, 50 30, 30 30))'),
                       QSYS2.ST_POLYGON('polygon((40 40, 40 60, 60 60, 60 40, 40 40))')));
```

Results:

```
00001
-------------------------------------------------------------------------------------------
POLYGON ((30.0 50.0, 30.0 30.0, 50.0 30.0, 50.0 40.432460999999996, 40.0 40.0,
          40.0 50.431312999999996, 30.0 50.0))
```

ST_DISJOINT scalar function

The ST_DISJOINT function takes two geometry objects as input and returns the integer 1 if the specified geometries do not intersect. If the geometries do intersect, then the integer 0 (zero) is returned.

ST_DISJOINT returns the exact opposite result of ST_INTERSECTS.

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* or *geometry2* is empty, 1 is returned.

▶▶─ ST_DISJOINT ── (── *geometry1* ── , ── *geometry2* ──) ─▶◀

geometry1
A value of type ST_GEOMETRY that represents the geometry that is tested to be disjoint with *geometry2*.

geometry2
A value of type ST_GEOMETRY that represents the geometry that is tested to be disjoint with *geometry1*.

The result of the function is INTEGER.

Notes

Two geometries are disjoint if all of the following conditions are true:

- the interior of *geometry1* does not intersect the interior of *geometry2*.
- the interior of *geometry1* does not intersect the boundary of *geometry2*.
- the boundary of *geometry1* does not intersect the interior of *geometry2*.
- the boundaries of *geometry1* and *geometry2* do not intersect.

Example

Determine if a polygon is disjoint from other polygons.

```
CREATE TABLE sample_polygons (polygon_id INTEGER, geometry QSYS2.ST_POLYGON);

INSERT INTO sample_polygons VALUES
  (10, QSYS2.ST_POLYGON('polygon((30 30, 30 50, 50 50, 50 30, 30 30))')),
  (20, QSYS2.ST_POLYGON('polygon((40 40, 40 60, 60 60, 60 40, 40 40))'));

CREATE VARIABLE my_polygon QSYS2.ST_POLYGON;
SET my_polygon = QSYS2.ST_POLYGON('polygon((20 30, 30 30, 30 40, 20 40, 20 30))');

SELECT polygon_id, QSYS2.ST_DISJOINT(my_polygon, geometry) AS disjoint
  FROM sample_polygons;
```

Results:

```

POLYGON_ID     DISJOINT
-----------    ---------
         10            0
         20            1
```

ST_DISTANCE scalar function

The ST_DISTANCE function takes two geometry objects as input parameters and returns the shortest distance, in meters, between any point in the first geometry to any point in the second geometry.

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* or *geometry2* is empty, an error is returned.

▶▶─ ST_DISTANCE ── (── *geometry1* ── , ── *geometry2* ──) ─▶◀

geometry1
A value of type ST_GEOMETRY that represents the geometry that is used to compute the distance to *geometry2*.

geometry2
A value of type ST_GEOMETRY that represents the geometry that is used to compute the distance to *geometry1*.

The result of the function is DOUBLE.

Examples

Find the distance between two points.

```
CREATE VARIABLE point1 QSYS2.ST_POINT;
CREATE VARIABLE point2 QSYS2.ST_POINT;

SET point1 = QSYS2.ST_POINT('point(10 10)');
SET point2 = QSYS2.ST_POINT('point(11 11)');

SELECT QSYS2.ST_ASTEXT(point1) AS point1,
       QSYS2.ST_ASTEXT(point2) AS point2,
       QSYS2.ST_DISTANCE(point1, point2) as distance
 FROM sysibm.sysdummy1;
```

Results:

```

POINT1                POINT2               DISTANCE
------------------    -----------------    ------------------
POINT (10.0 10.0)     POINT (11.0 11.0)    156115.9051873017
```

Example 2

Find the distance between different polygons.

```
CREATE VARIABLE linestring2 QSYS2.ST_LINESTRING;
CREATE VARIABLE polygon2 QSYS2.ST_POLYGON;

SET linestring2 = QSYS2.ST_LINESTRING('linestring(18 19, 19 19.5)');
SET polygon2 = QSYS2.ST_POLYGON('polygon((20 0, 20 20, 30 20, 30 0, 20 0))');

SELECT QSYS2.ST_GEOMETRYTYPE(linestring2) AS geometry1,
       QSYS2.ST_GEOMETRYTYPE(polygon2) AS geometry2,
       QSYS2.ST_DISTANCE(linestring2, polygon2) as distance
 FROM sysibm.sysdummy1;
```

Results:

```

GEOMETRY1        GEOMETRY2     DISTANCE
--------------   -----------   -------------------
ST_LINESTRING    ST_POLYGON    104933.77710129759
```

ST_EQUALS scalar function

The ST_EQUALS function takes two geometry objects as input parameters and returns the integer 1 if the latitude and longitude of the geometries are equal. Otherwise, the integer 0 (zero) is returned.

The order of the points used to define the geometry is not relevant for the test for equality.

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* or *geometry2* is empty, 0 (zero) is returned. If both *geometry1* and *geometry2* are empty, 1 is returned.

►► ST_EQUALS — (— *geometry1* — , — *geometry2* —) ►◄

Parameters

geometry1
A value of type ST_GEOMETRY that represents the geometry that is to be compared with *geometry2*.

geometry2
A value of type ST_GEOMETRY that represents the geometry that is to be compared with *geometry1*.

The result of the function is INTEGER.

Example

Compare two polygons to determine if they are the same. Two geometries are equal if the coordinates are the same but in a different order. This is demonstrated by the first row in the following table.

```
CREATE TABLE sample_geometry (geometry_id INTEGER, geometry QSYS2.ST_GEOMETRY);
INSERT INTO sample_geometry VALUES
  (10, QSYS2.ST_POLYGON('polygon((50 30, 30 30, 30 50, 50 50, 50 30))')),
  (20, QSYS2.ST_POLYGON('polygon((10 20, 50 50, 30 50, 30 40, 10 20))'));

CREATE VARIABLE my_polygon QSYS2.ST_POLYGON;
SET my_polygon = QSYS2.ST_POLYGON('polygon((50 30, 50 50, 30 50, 30 30, 50 30))');

SELECT geometry_id, QSYS2.ST_EQUALS(geometry, my_polygon) AS equals
  FROM sample_geometry;
```

Results:

```
GEOMETRY_ID     EQUALS
-------------   -------
           10         1
           20         0
```

ST_FUZZYGEOHASHCOVER table function

The ST_FUZZYGEOHASHCOVER table function takes a geometry object and a depth as input parameters and returns a table with one column containing the geohash cover of the minimum bounding rectangle of the specified geometry at the specified depth.

A geohash cover is the set of geohash cells that are needed to completely cover a given geometry. A larger depth corresponds to smaller cells, and therefore results in more exact coverage and less area of the geohash cover that is outside of the geometry. However, when the cells are smaller, more cells are needed.

The minimum bounding rectangle of a geometry is the bounding geometry formed by the minimum and maximum (X,Y) coordinates. See “Minimum bounding rectangle” on page 8 for more information.

▶▶— ST_FUZZYGEOHASHCOVER — (— *geometry* — , — *depth* —) —▶◀

geometry
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry for which the geohash cover values are to be calculated. If the specified geometry is NULL or empty, one row with a geohash value of NULL is returned.

depth
An integer value in the range 1 - 45 that determines the size of the geohash cell. The following table lists some of the most commonly used depths their corresponding approximate cell sizes.

Geohash Depth	Approximate Cell Size	Description	Examples
45	.1 km^2	Single point or address	GPS-location or house
28	3 km^2	Small region	city block
23	100 km^2	Medium-sized region	forest or lake
18	3,000 km^2	Large region	county or postal code area
13	100,000 km^2	Very large region	state or country

Geohash values of two geometries that are to be compared should be computed using the same depth for a meaningful comparison.

The result of the function is a table containing rows with the format shown in the following table. The column is nullable.

Table 19. Format of the resulting table for ST_FUZZYGEOHASHCOVER

Column name	Data type	Contains
GEOHASH	BIGINT	A geohash value of the minimum bounding rectangle of the specified point geometry at the specified depth.

Notes

The number of rows returned by this function cannot exceed 10,000. See “Geohashes and geohash covers” on page 13 for more information on selecting an appropriate depth value.

Example

The following SQL statement returns the geohash values that cover the minimum bounding box of the specified polygon geometry at depth 18:

```
SELECT geohash
  FROM TABLE(
    QSYS2.ST_FUZZYGEOHASHCOVER(QSYS2.ST_POLYGON('polygon((3 3, 3 5, 4 4, 3 3))'), 18));
```

Results:

```

GEOHASH
-------------------
6920906727361609728
6920977096105787392
6920941911733698560
6921012280477876224
6921047464849965056
6921117833594142720
6921082649222053888
6921153017966231552
6921469677315031040
6921540046059208704
6921504861687119872
6921575230431297536
6921610414803386368
6921680783547564032
```

ST_FUZZYGEOHASHCOVEREXTEND table function

The ST_FUZZYGEOHASHCOVEREXTEND table function takes a geometry object, a depth, and a distance as input parameters and returns a table with one column containing the geohash cover of the minimum bounding rectangle of the specified geometry at the specified depth plus a buffer zone surrounding the specified geometry.

A geohash cover is the set of geohash cells that are needed to completely cover a given geometry. A larger depth corresponds to smaller cells, and therefore results in more exact coverage and less area of the geohash cover that is outside of the geometry. However, when the cells are smaller, more cells are needed.

The buffer zone is the geometry that surrounds the geohash cover by a specified distance.

The minimum bounding rectangle of a geometry is the bounding geometry formed by the minimum and maximum (X,Y) coordinates. See “Minimum bounding rectangle” on page 8 for more information.

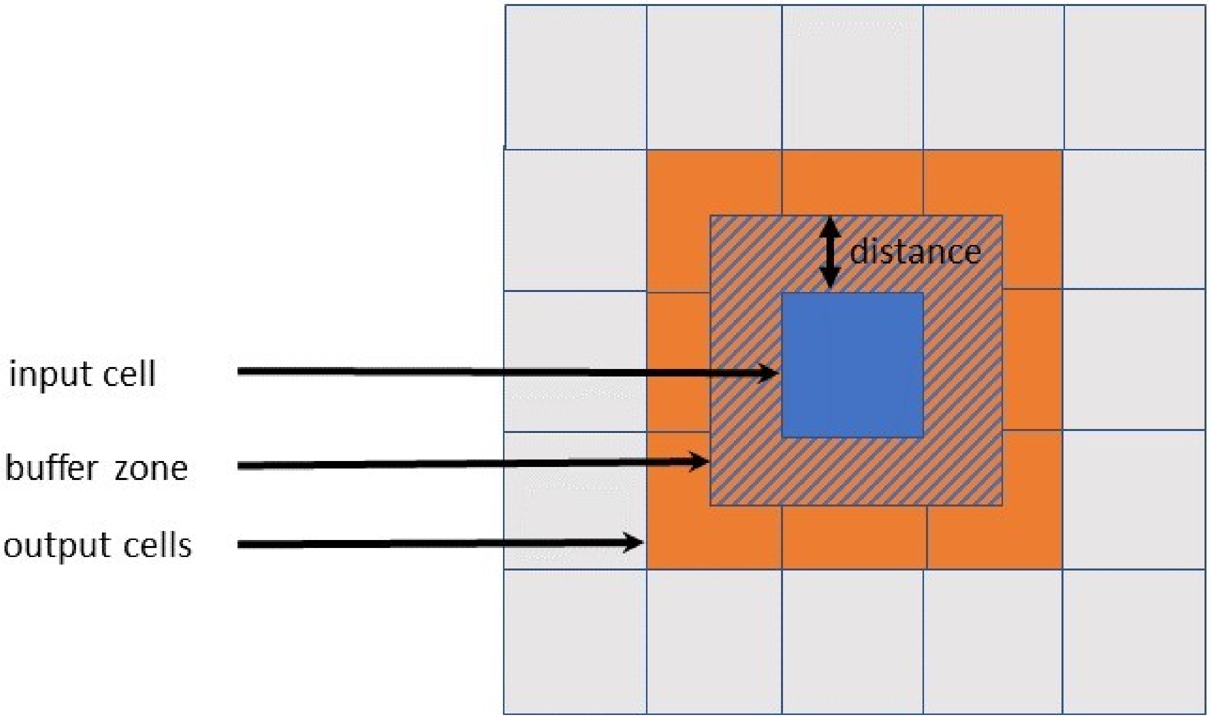


Figure 24. Geohash extension where the distance is less than the length of the side of one cell

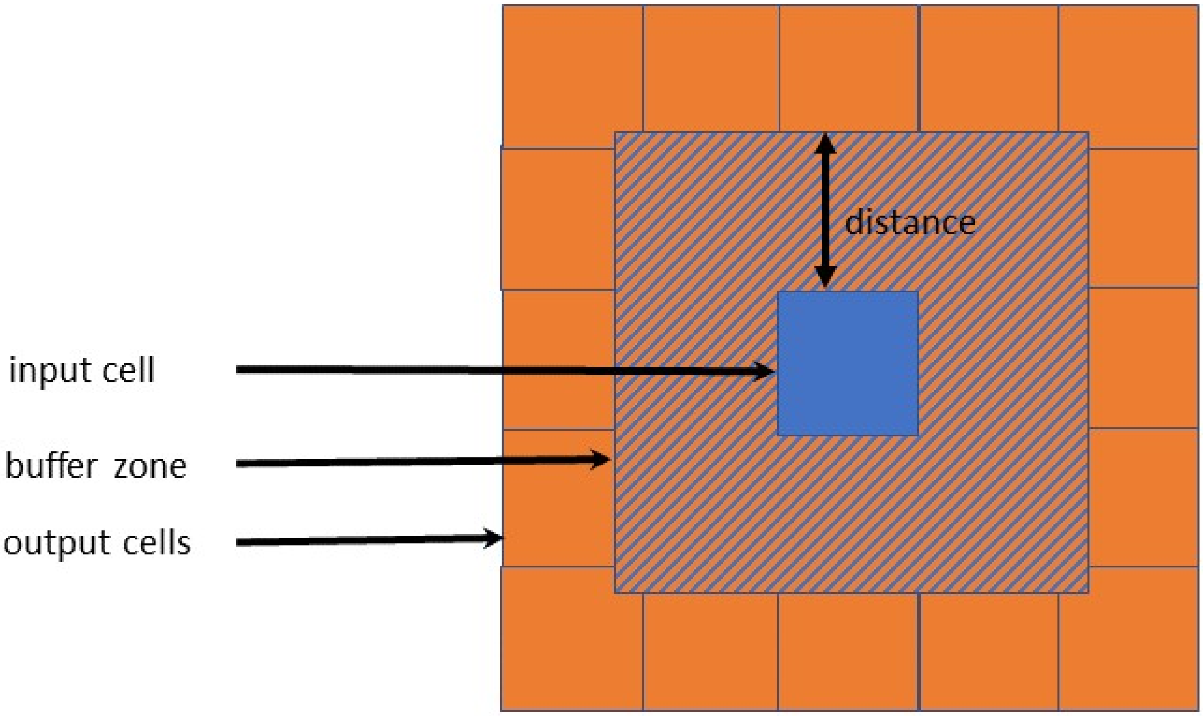


Figure 25. Geohash extension where the distance is greater than the length of the side of one cell

```
▶▶─ ST_FUZZYGEOHASHCOVEREXTEND ── ( ── geometry ── , ── depth ── , ── distance ── ) ─▶◀
```

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry for which the geohash cover values are to be calculated. If the specified geometry is NULL or empty, one row with a geohash value of NULL is returned.

depth

An integer value in the range 1 - 45 that determines the size of the geohash cell. The following table lists some of the most commonly used depths and the corresponding approximate cell sizes.

Geohash Depth	**Approximate Cell Size**	**Description**	**Examples**
45	.1 km^2	Single point or address	GPS-location or house
28	3 km^2	Small region	city block
23	100 km^2	Medium-sized region	forest or lake
18	3,000 km^2	Large region	county or postal code area
13	100,000 km^2	Very large region	state or country

Geohash values of two geometries that are to be compared must be computed using the same depth for a meaningful comparison.

distance

The distance from the edge of the input cell that determines the size of the buffer zone. The value cannot be negative or null. The unit of measure is meters.

The result of the function is a table containing rows with the format shown in the following table. The column is nullable.

Table 20. Format of the resulting table for ST_FUZZYGEOHASHCOVEREXTEND

Column name	**Data type**	**Contains**
GEOHASH	BIGINT	A geohash value of the minimum bounding rectangle of the specified point geometry at the specified depth after it has been extended in all dimensions by the specified distance.

Notes

The number of rows returned by this function cannot exceed 10,000. See "Geohashes and geohash covers" on page 13 for more information on selecting an appropriate depth value.

Example

The following SQL statement returns the geohash values that cover the minimum bounding box of the specified polygon geometry at depth 18 with a buffer of 25 kilometers:

```
SELECT geohash
  FROM TABLE(
    QSYS2.ST_FUZZYGEOHASHCOVEREXTEND(QSYS2.ST_POLYGON('polygon((3 3, 3 5, 4 4, 3 3))'),
                                     18,
                                     25000));
```

Results:

```

GEOHASH
-------------------
6918619743175835648
6919393799361789952
6919464168105967616
6919675274338500608
```

```
6920132671175655424
6920906727361609728
6920977096105787392
6921188202338320384
6920167855547744256
6920941911733698560
6921012280477876224
6921223386710409216
6920273408664010752
6921047464849965056
6921117833594142720
6921328939826675712
6920308593036099584
6921082649222053888
6921153017966231552
6921364124198764544
6920695621129076736
6921469677315031040
6921540046059208704
6921751152291741696
6920730805501165568
6921504861687119872
6921575230431297536
6921786336663830528
6920836358617432064
6921610414803386368
6921680783547564032
6921891889780097024
```

ST_GEOHASH table function

The ST_GEOHASH table function takes an ST_POINT object and a depth as input parameters and returns a table with one column containing the geohash of the specified point geometry at the specified depth.

A geohash is a number that uniquely identifies a specific region. The geohash algorithm divides the Earth into regions, called cells, and converts the latitude and longitude of the center of each cell into a number that uniquely identifies it. The size of each cell is determined by the depth value. The smaller the depth value, the larger the cell size.

ST_GEOHASH (*point_geometry* , *depth*)

point_geometry

A value of type ST_POINT that represents the point geometry for which the geohash is to be calculated. If the specified geometry is NULL or empty, one row with a geohash value of NULL is returned.

depth

An integer value in the range 1 - 45 that determines the size of the geohash cell. The following list contains some commonly used depths their corresponding approximate cell sizes.

Geohash Depth	Approximate Cell Size	Description	Examples
45	.1 km^2	Single point or address	GPS-location or house
28	3 km^2	Small region	city block
23	100 km^2	Medium-sized region	forest or lake
18	3,000 km^2	Large region	county or postal code area
13	100,000 km^2	Very large region	state or country

Geohash values of two geometries that are to be compared should be computed using the same depth for a meaningful comparison.

The result of the function is a table containing one row with the format shown in the following table. The column is nullable.

Table 21. Format of the resulting table for ST_GEOHASH

Column name	Data type	Contains
GEOHASH	BIGINT	The geohash value of the specified point geometry at the specified depth.

Example

The following SQL statement returns the geohash value that covers the specified point geometry at depth 18:

```
SELECT geohash FROM TABLE(QSYS2.ST_GEOHASH(QSYS2.ST_POINT('point(10 20)'), 18));
```

Results:

```

GEOHASH
-------------------
6970727798239395840
```

ST_GEOHASHCOVER table function

The ST_GEOHASHCOVER table function takes a geometry object and a depth as input parameters and returns a table with one column containing the geohash cover of the specified geometry at the specified depth.

A geohash cover is the set of geohash cells that are needed to completely cover a given geometry. A larger depth corresponds to smaller cells, and therefore results in more exact coverage and less area of the geohash cover that is outside of the geometry. However, when the cells are smaller, more cells are needed.

```
▶▶─ ST_GEOHASHCOVER ─ ( ─ geometry ─ , ─ depth ─ ) ─▶◀
```

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry for which the geohash cover values are to be calculated. If the specified geometry is NULL or empty, one row with a geohash value of NULL is returned.

depth

An integer value in the range 1 - 45 that determines the size of the geohash cell. The following list contains some commonly used depths their corresponding approximate cell sizes.

Geohash Depth	Approximate Cell Size	Description	Examples
45	.1 km^2	Single point or address	GPS-location or house
28	3 km^2	Small region	city block
23	100 km^2	Medium-sized region	forest or lake
18	3,000 km^2	Large region	county or postal code area
13	100,000 km^2	Very large region	state or country

Geohash values of two geometries that are to be compared should be computed using the same depth for a meaningful comparison.

The result of the function is a table containing rows with the format shown in the following table. The column is nullable.

Table 22. Format of the resulting table for ST_GEOHASHCOVER

Column name	Data type	Contains
GEOHASH	BIGINT	A geohash value of the specified point geometry at the specified depth.

Notes

The number of rows returned by this function cannot exceed 10,000. See "Geohashes and geohash covers" on page 13 for more information on selecting an appropriate depth value.

Example

The following SQL statement returns the geohash values that cover the specified polygon geometry at depth 18:

```
SELECT geohash
  FROM TABLE(
    QSYS2.ST_GEOHASHCOVER(QSYS2.ST_POLYGON('polygon((3 3, 3 5, 4 4, 3 3))'), 18));
```

Results:

```

GEOHASH
--------------------
6920906727361609728
6920941911733698560
6921047464849965056
6921117833594142720
6921082649222053888
6921153017966231552
6921469677315031040
6921540046059208704
6921504861687119872
6921610414803386368
```

ST_GEOHASHCOVEREXTEND table function

The ST_GEOHASHCOVEREXTEND table function takes a geometry object, a depth, and a distance as input parameters and returns a table with one column containing the geohash cover of the specified geometry at the specified depth plus a buffer zone surrounding the specified geometry.

A geohash cover is the set of geohash cells that are needed to completely cover a given geometry. A larger depth corresponds to smaller cells, and therefore results in more exact coverage and less area of the geohash cover that is outside of the geometry. However, when the cells are smaller, more cells are needed.

The buffer zone is the geometry that surrounds the geohash cover by a specified distance.

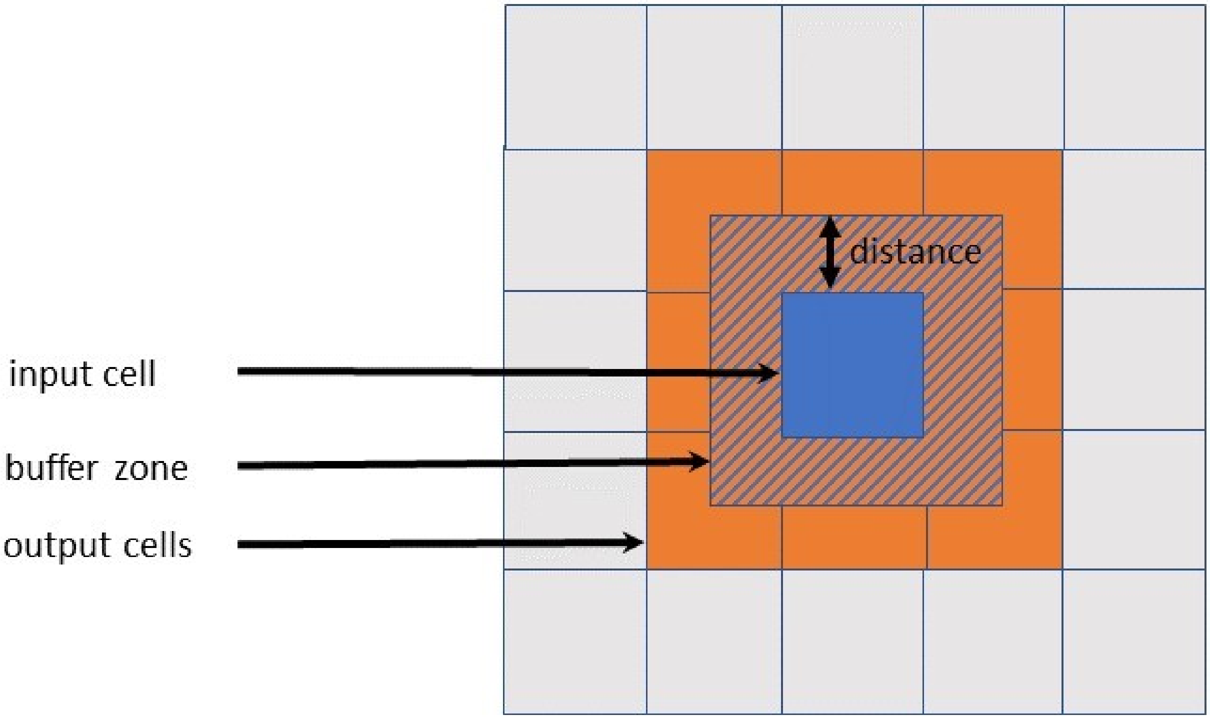


Figure 26. Geohash extension where the distance is less than the length of the side of one cell

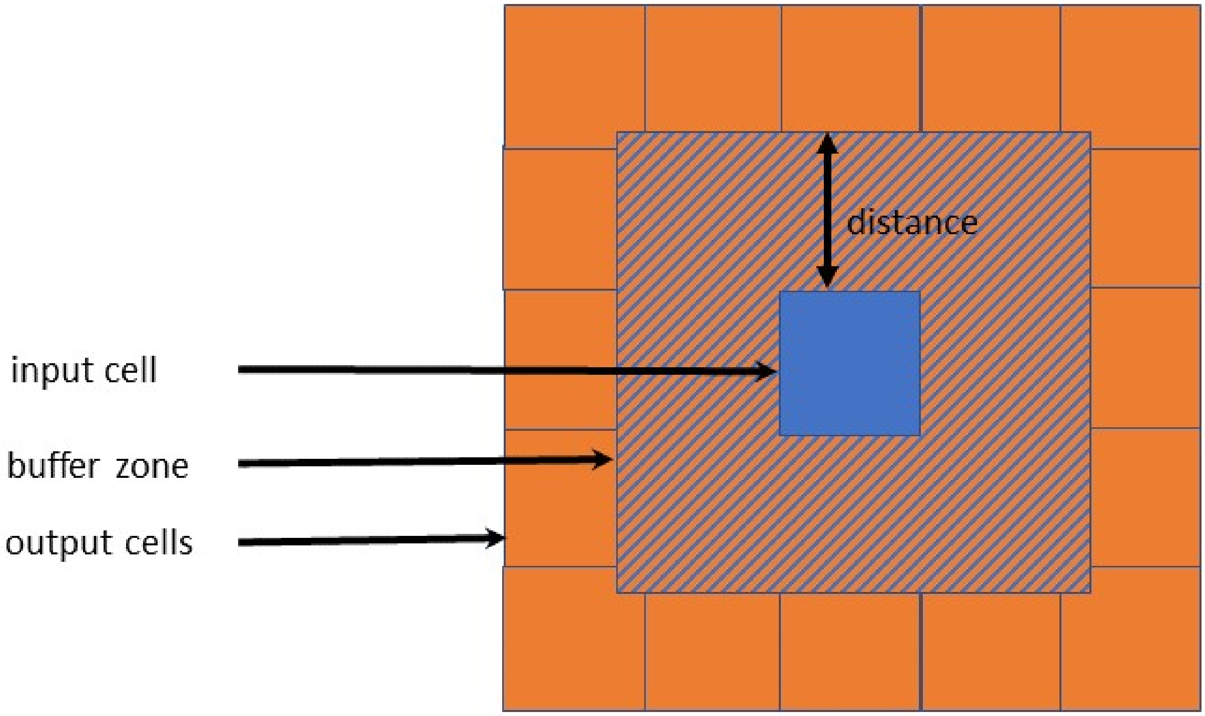


Figure 27. Geohash extension where the distance is greater than the length of the side of one cell

```
►► ST_GEOHASHCOVEREXTEND — ( — geometry — , — depth — , — distance — ) ►◄
```

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry for which the geohash cover values are to be calculated. If the specified geometry is NULL or empty, one row with a geohash value of NULL is returned.

depth

An integer value in the range 1 - 45 that determines the size of the geohash cell. The following table lists some of the most commonly used depths their corresponding approximate cell sizes.

Geohash Depth	Approximate Cell Size	Description	Examples
45	.1 km^2	Single point or address	GPS-location or house
28	3 km^2	Small region	city block
23	100 km^2	Medium-sized region	forest or lake
18	3,000 km^2	Large region	county or postal code area
13	100,000 km^2	Very large region	state or country

Geohash values of two geometries that are to be compared should be computed using the same depth for a meaningful comparison.

distance

The distance from the edge of the input cell that determines the size of the buffer zone. The value cannot be negative or null. The unit of measure is meters.

The result of the function is a table containing rows with the format shown in the following table. The column is nullable.

Table 23. Format of the resulting table for ST_GEOHASHCOVEREXTEND

Column name	Data type	Contains
GEOHASH	BIGINT	A geohash value of the specified point geometry at the specified depth after it has been extended in all dimensions by the specified distance.

Notes

The number of rows returned by this function cannot exceed 10,000. See “Geohashes and geohash covers” on page 13 for more information on selecting an appropriate depth value.

Example

The following SQL statement returns the geohash values that cover the specified polygon geometry at depth 18 with a buffer of 25 kilometers:

```
SELECT geohash
FROM TABLE(
  QSYS2.ST_GEOHASHCOVEREXTEND(QSYS2.ST_POLYGON('polygon((3 3, 3 5, 4 4, 3 3))'),
                              18,
                              25000));
```

Results:

```
GEOHASH
-------------------
6919393799361789952
6920132671175655424
6920906727361609728
6920167855547744256
6920941911733698560
```

```
6921012280477876224
6920273408664010752
6921047464849965056
6921117833594142720
6920308593036099584
6921082649222053888
6921153017966231552
6921364124198764544
6920695621129076736
6921469677315031040
6921540046059208704
6920730805501165568
6921504861687119872
6921575230431297536
6920836358617432064
6921610414803386368
```

ST_GEOHASHVALUE scalar function

The ST_GEOHASHVALUE scalar function takes an ST_POINT object and a depth as input parameters and returns a number representing the geohash of the specified point geometry at the specified depth.

A geohash is a number that uniquely identifies a specific region. The geohash algorithm divides the Earth into regions, called cells, and converts the latitude and longitude of the center of each cell into a number that uniquely identifies it. The size of each cell is determined by the depth value. The smaller the depth value, the larger the cell size.

▶▶— ST_GEOHASHVALUE —— (—— *point_geometry* —— , —— *depth* ——) —▶◀

point_geometry

A value of type ST_POINT that represents the point geometry for which the geohash is to be calculated. If the specified geometry is NULL or empty, NULL is returned.

depth

An integer value in the range 1 - 45 that determines the size of the geohash cell. The following list contains some commonly used depths their corresponding approximate cell sizes.

Geohash Depth	Approximate Cell Size	Description	Examples
45	.1 km^2	Single point or address	GPS-location or house
28	3 km^2	Small region	city block
23	100 km^2	Medium-sized region	forest or lake
18	3,000 km^2	Large region	county or postal code area
13	100,000 km^2	Very large region	state or country

Geohash values of two geometries that are to be compared should be computed using the same depth for a meaningful comparison.

The result of the function is BIGINT.

Example

The following SQL statement returns the geohash values for point geometries at depth 23:

```
CREATE TABLE EXAMPLE_POINTS (GEO_ID CHAR(5), GEO QSYS2.ST_POINT);

INSERT INTO EXAMPLE_POINTS VALUES
  ('11111', QSYS2.ST_POINT(10, 10)),
  ('22222', QSYS2.ST_POINT(15, 15)),
  ('33333', QSYS2.ST_POINT(-10, -10));

SELECT QSYS2.ST_GEOHASHVALUE(GEO, 23) AS GEOHASH FROM EXAMPLE_POINTS;
```

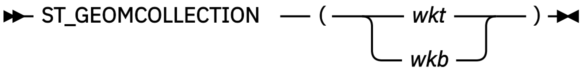
Results:

```
GEOHASH
--------------------
6935265249708736512
6975174223262121984
2288105687634411520
```

ST_GEOMCOLLECTION scalar function

The ST_GEOMCOLLECTION function constructs a geometry collection.

If *wkt* or *wkb* is null, the result is the null value. An empty geometry collection is not supported.



wkt
An expression that returns a character or graphic string value that contains the well-known text (WKT) format of the resulting geometry collection.

wkb
An expression that returns a binary string value that contains the well-known binary (WKB) format of the resulting geometry collection.

The result of the function is ST_GEOMCOLLECTION. If the argument is null, the result is the null value.

For details about the supported formats, see "WKT and WKB data formats" on page 14.

Example

The ST_GEOMCOLLECTION function can be used to create and insert a multipoint, multiline, and multipolygon from well-known text representation.

```
CREATE TABLE sample_geomcollections(id INTEGER,
                                    geometry QSYS2.ST_GEOMCOLLECTION);

INSERT INTO sample_geomcollections(id, geometry) VALUES
 (4001, QSYS2.ST_GEOMCOLLECTION('multipoint((1 2), (4 3), (5 6))')),
 (4002, QSYS2.ST_GEOMCOLLECTION('multilinestring((33 2, 34 3, 35 6),(28 4, 29 5, 31 8, 43 12),
                                                 (39 3, 37 4, 36 7))')),
 (4003, QSYS2.ST_GEOMCOLLECTION('multipolygon(((3 3, 4 6, 5 3, 3 3)),
                                              ((8 24, 9 25, 1 28, 8 24)),
                                              ((13 33, 7 36, 1 40, 10 43, 13 33)))'));

SELECT id, QSYS2.ST_ASTEXT(geometry) AS GeomCollection
  FROM sample_geomcollections;
```

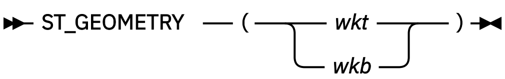
Results:

```
ID             GEOMCOLLECTION
------------   ----------------------------------------------------------------------------------
        4001   MULTIPOINT ((1.0 2.0), (4.0 3.0), (5.0 6.0))
        4002   MULTILINESTRING ((33.0 2.0, 34.0 3.0, 35.0 6.0),
                                (28.0 4.0, 29.0 5.0, 31.0 8.0, 43.0 12.0),
                                (39.0 3.0, 37.0 4.0, 36.0 7.0))
        4003   MULTIPOLYGON (((13.0 33.0, 10.0 43.0, 1.0 40.0, 7.0 36.0, 13.0 33.0)),
                             ((3.0 3.0, 5.0 3.0, 4.0 6.0, 3.0 3.0)),
                             ((8.0 24.0, 9.0 25.0, 1.0 28.0, 8.0 24.0)))
```

ST_GEOMETRY scalar function

The ST_GEOMETRY function constructs a geometry from a specified representation.

If *wkt* or *wkb* is null, the result is the null value. If *wkt* is 'POINT EMPTY', an empty geometry is returned. There is no WKB representation for an empty geometry.



wkt

An expression that returns a character or graphic string value that contains the well-known text (WKT) format of the resulting geometry.

wkb

An expression that returns a binary string value that contains the well-known binary (WKB) format of the resulting geometry.

The result of the function is ST_GEOMETRY. If the argument is null, the result is the null value.

For details about the supported formats, see "WKT and WKB data formats" on page 14.

Example

Use the ST_GEOMETRY function to construct a point, a linestring, and a polygon.

```
CREATE TABLE sample_geometries(id INTEGER, geometry QSYS2.ST_GEOMETRY);

INSERT INTO sample_geometries(id, geometry) VALUES
  (7001, QSYS2.ST_GEOMETRY('point(1 2)')),
  (7002, QSYS2.ST_GEOMETRY('linestring(33 2, 34 3, 35 6)')),
  (7003, QSYS2.ST_GEOMETRY('polygon((3 3, 4 6, 5 3, 3 3))'));

SELECT id, QSYS2.ST_ASTEXT(geometry) AS geometry
  FROM sample_geometries;
```

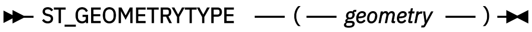
Results:

```
ID       GEOMETRY
------   ------------------------------------------------
 7001    POINT (1.0 2.0)
 7002    LINESTRING (33.0 2.0, 34.0 3.0, 35.0 6.0)
 7003    POLYGON ((3.0 3.0, 5.0 3.0, 4.0 6.0, 3.0 3.0))
```

ST_GEOMETRYTYPE scalar function

The ST_GEOMETRYTYPE function takes a geometry object as an input parameter and returns the fully qualified type name of the dynamic type of the specified geometry.

If *geometry* is null, the result is the null value. If *geometry* is empty, "EMPTYGEOMETRY" is returned.



geometry

A value of type ST_GEOMETRY for which the geometry type is to be returned.

The result of the function is VARCHAR(128).

Example

Determine the type of a geometry.

```
CREATE TABLE sample_geometries (id INTEGER, geometry QSYS2.ST_GEOMETRY);

INSERT INTO sample_geometries(id, geometry) VALUES
  (7101, QSYS2.ST_POINT('point(1 2)')),
  (7102, QSYS2.ST_LINESTRING('linestring(33 2, 34 3, 35 6)')),
  (7103, QSYS2.ST_POLYGON('polygon((3 3, 4 6, 5 3, 3 3))')),
  (7104, QSYS2.ST_MULTIPOINT('multipoint((1 2), (4 3))'));

SELECT id, QSYS2.ST_GEOMETRYTYPE(geometry) AS geometry_type
  FROM sample_geometries;
```

Results:

```
ID       GEOMETRY_TYPE
------   ----------------
 7101    ST_POINT
 7102    ST_LINESTRING
 7103    ST_POLYGON
 7104    ST_MULTIPOINT
```

ST_INTERSECTION scalar function

The ST_INTERSECTION function takes two geometry objects as input parameters and returns the geometry that is the intersection of the two specified geometries.

The intersection is the common part of the first geometry and the second geometry.

If possible, the specific type of the returned geometry will be ST_POINT, ST_LINESTRING, or ST_POLYGON. For example, the intersection of a point and a polygon is either empty or a single point, represented as ST_POINT.

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* or *geometry2* is empty, an empty geometry is returned.

►► ST_INTERSECTION — (— *geometry1* — , — *geometry2* —) ►◄

geometry1
A value of type ST_GEOMETRY or one of its subtypes that represents the first geometry to compute the intersection with *geometry2*.

geometry2
A value of type ST_GEOMETRY or one of its subtypes that represents the second geometry to compute the intersection with *geometry1*.

The result of the function is ST_GEOMETRY.

The dimension of the returned geometry is that of the input with the lower dimension.

Example

Find the intersection of two geometries.

```
CREATE TABLE sample_geometries (id INTEGER, geometry QSYS2.ST_GEOMETRY);

INSERT INTO sample_geometries VALUES
  (2, QSYS2.ST_POLYGON('polygon((20 30, 30 30, 30 40, 20 40, 20 30))')),
  (3, QSYS2.ST_POLYGON('polygon((40 40, 40 60, 60 60, 60 40, 40 40))')),
  (4, QSYS2.ST_LINESTRING('linestring(60 60, 70 70)')),
  (5, QSYS2.ST_LINESTRING('linestring(30 30, 60 60)'));

CREATE VARIABLE my_linestring QSYS2.ST_LINESTRING;
SET my_linestring = QSYS2.ST_LINESTRING('linestring(30 30, 60 60)');

SELECT id, QSYS2.ST_ASTEXT(QSYS2.ST_INTERSECTION(my_linestring, geometry)) AS intersection
  FROM sample_geometries;
```

Results:

```
ID       INTERSECTION
------   ------------------------------------------------
    2    LINESTRING (30.0 30.0, 30.0 30.0)
    3    LINESTRING (40.0 44.898573, 60.0 60.0)
    4    POINT (60.0 60.0)
    5    LINESTRING (30.0 30.0, 60.0 60.0)
```

ST_INTERSECTS scalar function

The ST_INTERSECTS function takes two geometry objects as input parameters and returns the integer value 1 if the specified geometries intersect. If the geometries do not intersect, the integer value 0 (zero) is returned.

ST_INTERSECTS returns the exact opposite result of ST_DISJOINT.

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* or *geometry2* is empty, 0 (zero) is returned.

▶▶─ ST_INTERSECTS ── (── *geometry1* ── , ── *geometry2* ──) ─▶◀

geometry1
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry to test for intersection with *geometry2*.

geometry2
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry to test for intersection with *geometry1*.

The result of the function is INTEGER.

Notes

Two geometries intersect if any of the following conditions are true:

- the interior of *geometry1* intersects the interior of *geometry2*.
- the interior of *geometry1* intersects the boundary of *geometry2*.
- the boundary of *geometry1* intersects the interior of *geometry2*.
- the boundaries of either geometry intersect.

Example

Determine whether the various geometries in the SAMPLE_GEOMETRIES1 and SAMPLE_GEOMETRIES2 tables intersect.

```
CREATE TABLE sample_geometries1(id SMALLINT, geometry QSYS2.ST_GEOMETRY);
CREATE TABLE sample_geometries2(id SMALLINT, geometry QSYS2.ST_GEOMETRY);

INSERT INTO sample_geometries1(id, geometry) VALUES
 ( 1, QSYS2.ST_POINT('point(55 15)')),
 (10, QSYS2.ST_LINESTRING('linestring(80 80, 90 80)')),
 (20, QSYS2.ST_POLYGON('polygon((50 10, 50 20, 70 20, 70 10, 50 10))'));


INSERT INTO sample_geometries2(id, geometry) VALUES
 (101, QSYS2.ST_POINT('point(55 15)')),
 (102, QSYS2.ST_POINT('point(65 20)')),
 (103, QSYS2.ST_POINT('point(80 80)')),
 (110, QSYS2.ST_LINESTRING('linestring(85 25, 85 85)')),
 (120, QSYS2.ST_POLYGON('polygon((65 50, 65 15, 80 15, 80 50, 65 50))')),
 (121, QSYS2.ST_POLYGON('polygon((20 20, 20 40, 40 40, 40 20, 20 20))'));


SELECT sg1.id AS sg1_id, QSYS2.ST_GEOMETRYTYPE(sg1.geometry) AS sg1_type,
       sg2.id AS sg2_id, QSYS2.ST_GEOMETRYTYPE(sg2.geometry) AS sg2_type,
       CASE QSYS2.ST_INTERSECTS(sg1.geometry, sg2.geometry)
          WHEN 0 THEN 'Geometries do not intersect'
          WHEN 1 THEN 'Geometries intersect'
       END AS intersects
  FROM sample_geometries1 sg1, sample_geometries2 sg2
  ORDER BY sg1.id;
```

Results:

```

SG1_ID   SG1_TYPE        SG2_ID   SG2_TYPE        INTERSECTS
```

```
------   -------------   ------   -------------   -----------------------
     1   ST_POINT           101   ST_POINT        Geometries intersect
     1   ST_POINT           102   ST_POINT        Geometries do not intersect
     1   ST_POINT           103   ST_POINT        Geometries do not intersect
     1   ST_POINT           110   ST_LINESTRING   Geometries do not intersect
     1   ST_POINT           120   ST_POLYGON      Geometries do not intersect
     1   ST_POINT           121   ST_POLYGON      Geometries do not intersect
    10   ST_LINESTRING      101   ST_POINT        Geometries do not intersect
    10   ST_LINESTRING      102   ST_POINT        Geometries do not intersect
    10   ST_LINESTRING      103   ST_POINT        Geometries intersect
    10   ST_LINESTRING      110   ST_LINESTRING   Geometries intersect
    10   ST_LINESTRING      120   ST_POLYGON      Geometries do not intersect
    10   ST_LINESTRING      121   ST_POLYGON      Geometries do not intersect
    20   ST_POLYGON         101   ST_POINT        Geometries intersect
    20   ST_POLYGON         102   ST_POINT        Geometries intersect
    20   ST_POLYGON         103   ST_POINT        Geometries do not intersect
    20   ST_POLYGON         110   ST_LINESTRING   Geometries do not intersect
    20   ST_POLYGON         120   ST_POLYGON      Geometries intersect
    20   ST_POLYGON         121   ST_POLYGON      Geometries do not intersect
```

ST_ISSIMPLE scalar function

The ST_ISSIMPLE function takes a geometry object as an input parameter and returns the integer value 1 if the specified geometry is simple. Otherwise, the integer value 0 (zero) is returned.

Points are always simple.

If *geometry* is null, the result is the null value. An empty geometry is a simple geometry.

▶▶— ST_ISSIMPLE — (— *geometry* —) —▶◀

geometry
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry to be tested.

The result of the function is INTEGER.

Example

Return an indication of whether a geometry is simple or not.

```
CREATE TABLE sample_geometries1 (id INTEGER, geometry QSYS2.ST_GEOMETRY);

INSERT INTO sample_geometries1 VALUES
 (1, QSYS2.ST_GEOMETRY('point EMPTY')),
 (2, QSYS2.ST_POINT('point (21 33)')),
 (3, QSYS2.ST_MULTIPOINT('multipoint((10 10), (20 20), (30 30))')),
 (4, QSYS2.ST_MULTIPOINT('multipoint((10 10), (20 20), (30 30), (20 20))')),
 (5, QSYS2.ST_LINESTRING('linestring(60 60, 70 60, 70 70)')),
 (6, QSYS2.ST_LINESTRING('linestring(20 20, 30 30, 30 20, 20 30)')),
 (7, QSYS2.ST_POLYGON('polygon((40 40, 50 40, 50 50, 40 40))'));


SELECT id,
    CASE QSYS2.ST_ISSIMPLE(geometry)
      WHEN 0 THEN 'Geometry is not simple'
      WHEN 1 THEN 'Geometry is simple'
    END AS simple
  FROM sample_geometries1;
```

Results:

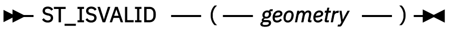
```
ID    SIMPLE
---   ----------------
  1   Geometry is simple
  2   Geometry is simple
  3   Geometry is simple
  4   Geometry is simple
  5   Geometry is simple
  6   Geometry is not simple
  7   Geometry is simple
```

ST_ISVALID scalar function

The ST_ISVALID function takes a geometry as an input parameter and returns 1 if it is valid. Otherwise 0 (zero) is returned.

A geometry is valid if the attributes are consistent and the internal representation is not corrupted.

If *geometry* is null, the result is the null value. An empty geometry is valid.



geometry
A value of type ST_GEOMETRY or one of its subtypes.

The result of the function is INTEGER.

Example

This example creates several geometries and uses ST_ISVALID to check if they are valid. Geometries that use the constructor routines, such as ST_GEOMETRY, do not allow invalid geometries to be constructed. Any type of empty geometry must be constructed as an ST_GEOMETRY.

```
CREATE TABLE sample_geoms (id INTEGER, geometry QSYS2.ST_GEOMETRY);

INSERT INTO sample_geoms VALUES
  (1, QSYS2.ST_GEOMETRY('point EMPTY')),
  (2, QSYS2.ST_POLYGON('polygon((40 20, 90 20, 90 50, 40 50, 40 20))')),
  (3, QSYS2.ST_MULTIPOINT('multipoint((10 10), (50 10), (10 30))')),
  (4, QSYS2.ST_LINESTRING('linestring (10 10, 20 10)')),
  (5, BLOB('point(10 10)'));

SELECT id, QSYS2.ST_ISVALID(geometry) Is_Valid
FROM sample_geoms;
```

Results:

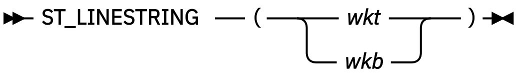
```

ID     IS_VALID
----   -------
   1      1
   2      1
   3      1
   4      1
   5      0
```

ST_LINESTRING scalar function

The ST_LINESTRING function constructs a linestring from specified inputs.

If *wkt* or *wkb* is null, the result is the null value. An empty linestring is not supported.



wkt
An expression that returns a character or graphic string value that contains the well-known text (WKT) format of the resulting linestring.

wkb
An expression that returns a binary string value that contains the well-known binary (WKB) format of the resulting linestring.

The result of the function is ST_LINESTRING. If the argument is null, the result is the null value.

For details about the supported formats, see "WKT and WKB data formats" on page 14.

Example

Use the ST_LINESTRING function to insert lines into a table using a well-known text line representation.

```
CREATE TABLE sample_lines(id SMALLINT, geometry QSYS2.ST_LINESTRING);
INSERT INTO sample_lines(id, geometry) VALUES
  (10, QSYS2.ST_LINESTRING('linestring(50 50, 85 75)')),
  (20, QSYS2.ST_LINESTRING('linestring(1 10, 2 20)'));

SELECT id, QSYS2.ST_ASTEXT(geometry) AS linestring
  FROM sample_lines;
```

Results:

```

ID     LINESTRING
----   ---------------------------------
  10   LINESTRING (50.0 50.0, 85.0 75.0)
  20   LINESTRING (1.0 10.0, 2.0 20.0)
```

ST_MAXX scalar function

The ST_MAXX function takes a geometry object as an input parameter and returns a double precision floating point value of the maximum X coordinate.

If *geometry* is null or an empty geometry, the result is the null value.

▸▸─ ST_MAXX ─ (─ *geometry* ─) ─▸◂

geometry
A value of type ST_GEOMETRY or one of its subtypes for which the maximum X coordinate is returned.

The result of the function is DOUBLE.

Example

Finds the maximum X coordinate of each polygon in SAMPLE_POLYGONS.

```
CREATE TABLE sample_polygons (id INTEGER, geometry QSYS2.ST_POLYGON);

INSERT INTO sample_polygons VALUES
  (1, QSYS2.ST_POLYGON('polygon ((11 12, 11 14, 12 13, 11 12))')),
  (2, QSYS2.ST_POLYGON('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))')),
  (3, QSYS2.ST_POLYGON('polygon ((-12 -13, -8 -4, -9 -4, -12 -13))'));

SELECT id, QSYS2.ST_MAXX(geometry) max_x
  FROM sample_polygons;
```

Results:

```

ID     MAX_X
----   -------
   1      12.0
   2       5.0
   3      -8.0
```

ST_MAXY scalar function

The ST_MAXY function takes a geometry object as an input parameter and returns a double precision floating point value of the maximum Y coordinate.

If *geometry* is null or an empty geometry, the result is the null value.

▸▸─ ST_MAXY ─ (─ *geometry* ─) ─▸◂

geometry
A value of type ST_GEOMETRY or one of its subtypes for which the maximum Y coordinate is returned.

The result of the function is DOUBLE.

Example

Finds the maximum Y coordinate of each polygon in SAMPLE_POLYGONS.

```
CREATE TABLE sample_polygons (id INTEGER, geometry QSYS2.ST_POLYGON);

INSERT INTO sample_polygons VALUES
  (1, QSYS2.ST_POLYGON('polygon ((11 12, 11 14, 12 13, 11 12))')),
  (2, QSYS2.ST_POLYGON('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))')),
  (3, QSYS2.ST_POLYGON('polygon ((-12 -13, -8 -4, -9 -4, -12 -13))'));

SELECT id, QSYS2.ST_MAXY(geometry) max_y
  FROM sample_polygons;
```

Results:

```
ID     MAX_Y
----   --------------------
   1                   14.0
   2      4.003798353045168
   3    -3.9999999999999996
```

ST_MINX scalar function

The ST_MINX function takes a geometry object as an input parameter and returns a double precision floating point value of the minimum X coordinate.

If *geometry* is null or an empty geometry, the result is the null value.

▶▶─ ST_MINX ── (── *geometry* ──) ─▶◀

geometry
A value of type ST_GEOMETRY or one of its subtypes for which the minimum X coordinate is returned.

The result of the function is DOUBLE.

Example

Finds the minimum X coordinate of each polygon in SAMPLE_POLYGONS.

```
CREATE TABLE sample_polygons (id INTEGER, geometry QSYS2.ST_POLYGON);

INSERT INTO sample_polygons VALUES
  (1, QSYS2.ST_POLYGON('polygon ((11 12, 11 14, 12 13, 11 12))')),
  (2, QSYS2.ST_POLYGON('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))')),
  (3, QSYS2.ST_POLYGON('polygon ((-12 -13, -8 -4, -9 -4, -12 -13))'));

SELECT id, QSYS2.ST_MINX(geometry) min_x
  FROM sample_polygons;
```

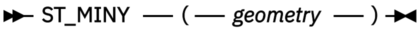
Results:

```
ID     MIN_X
----   -------
   1      11.0
   2       0.0
   3     -12.0
```

ST_MINY scalar function

The ST_MINY function takes a geometry object as an input parameter and returns a double precision floating point value of the minimum Y coordinate.

If *geometry* is null or an empty geometry, the result is the null value.



geometry
A value of type ST_GEOMETRY or one of its subtypes for which the minimum Y coordinate is returned.

The result of the function is DOUBLE.

Example

Finds the minimum Y coordinate of each polygon in SAMPLE_POLYGONS.

```
CREATE TABLE sample_polygons (id INTEGER, geometry QSYS2.ST_POLYGON);

INSERT INTO sample_polygons VALUES
  (1, QSYS2.ST_POLYGON('polygon ((11 12, 11 14, 12 13, 11 12))')),
  (2, QSYS2.ST_POLYGON('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))')),
  (3, QSYS2.ST_POLYGON('polygon ((-12 -13, -8 -4, -9 -4, -12 -13))'));

SELECT id, QSYS2.ST_MINY(geometry) min_y
  FROM sample_polygons;
```

Results:

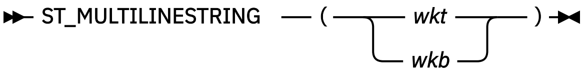
```

ID     MIN_Y
----   -------------------
   1    12.000000000000002
   2                   0.0
   3   -12.999999999999998
```

ST_MULTILINESTRING scalar function

The ST_MULTILINESTRING function constructs a multilinestring from a specified input.

If *wkt* or *wkb* is null, the result is the null value. An empty multilinestring is not supported.



wkt
An expression that returns a character or graphic string value that contains the well-known text (WKT) format of the resulting multilinestring.

wkb
An expression that returns a binary string value that contains the well-known binary (WKB) format of the resulting multilinestring.

The result of the function is ST_MULTILINESTRING. If the argument is null, the result is the null value.

For details about the supported formats, see "WKT and WKB data formats" on page 14.

Example

Use the ST_MULTILINESTRING function to create and insert a multilinestring from a well-known text linestring representation. The multilinestring consists of three strings.

```
CREATE TABLE sample_multi_lines (id SMALLINT, geometry QSYS2.ST_MULTILINESTRING);

INSERT INTO sample_multi_lines(id, geometry) VALUES
  (1110, QSYS2.ST_MULTILINESTRING ('multilinestring ((33 2, 34 3, 35 6),
                                                     (28 4, 29 5, 31 8, 43 12),
                                                     (39 3, 37 4, 36 7))'));

SELECT id, QSYS2.ST_ASTEXT(geometry) AS multilinestring
  FROM sample_multi_lines;
```

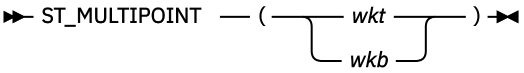
Results:

```
ID     MULTILINESTRING
----   --------------------------------------------------------------------------------------
1110   MULTILINESTRING ((33.0 2.0, 34.0 3.0, 35.0 6.0),
                        (28.0 4.0, 29.0 5.0, 31.0 8.0, 43.0 12.0),
                        (39.0 3.0, 37.0 4.0, 36.0 7.0))
```

ST_MULTIPOINT scalar function

The ST_MULTIPOINT function constructs a multipoint from a specified input.

If *wkt* or *wkb* is null, the result is the null value. An empty multipoint is not supported.



wkt
An expression that returns a character or graphic string value that contains the well-known text (WKT) format of the resulting multipoint.

wkb
An expression that returns a binary string value that contains the well-known binary (WKB) format of the resulting multipoint.

The result of the function is ST_MULTIPOINT. If the argument is null, the result is the null value.

For details about the supported formats, see "WKT and WKB data formats" on page 14.

Example

Use the ST_MULTIPOINT function to create and insert a multipoint from a well-known text line representation. The multipoint consists of three points.

```
CREATE TABLE sample_multi_points (id SMALLINT, geometry QSYS2.ST_MULTIPOINT);
INSERT INTO sample_multi_points(id, geometry) VALUES
  (1110, QSYS2.ST_MULTIPOINT ('multipoint ((1 20), (4 3), (5 6))'));

SELECT id, QSYS2.ST_ASTEXT(geometry) AS multipoints
  FROM sample_multi_points;
```

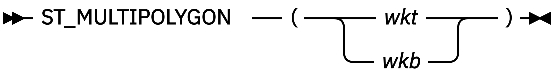
Results:

```
ID     MULTIPOINTS
----   --------------------------------------------
1110   MULTIPOINT ((1.0 20.0), (4.0 3.0), (5.0 6.0))
```

ST_MULTIPOLYGON scalar function

The ST_MULTIPOLYGON function constructs a multipolygon from a specified input.

If *wkt* or *wkb* is null, the result is the null value. An empty multipolygon is not supported.



wkt
An expression that returns a character or graphic string value that contains the well-known text (WKT) format of the resulting multipolygon.

wkb
An expression that returns a binary string value that contains the well-known binary (WKB) format of the resulting multipolygon.

The result of the function is ST_MULTIPOLYGON. If the argument is null, the result is the null value.

For details about the supported formats, see "WKT and WKB data formats" on page 14.

Example

Use the ST_MULTIPOLYGON function to create and insert a multipolygon from a well-known text multipolygon representation.

```
CREATE TABLE sample_multi_polygon (id SMALLINT, geometry QSYS2.ST_MULTIPOLYGON);
INSERT INTO sample_multi_polygon(id, geometry) VALUES
  (1110, QSYS2.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)))'));

SELECT id, QSYS2.ST_ASTEXT(geometry) AS multipolygons
  FROM sample_multi_polygon;
```

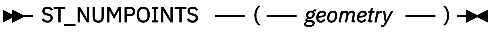
Results:

```
ID     MULTIPOLYGONS
----   -----------------------------------------------------------------------------------------
1110   MULTIPOLYGON (((13.0 33.0, 10.0 43.0, 1.0 40.0, 7.0 36.0, 13.0 33.0)),
                     ((3.0 3.0, 5.0 3.0, 4.0 6.0, 3.0 3.0)),
                     ((8.0 24.0, 9.0 25.0, 1.0 28.0, 8.0 24.0)))
```

ST_NUMPOINTS scalar function

The ST_NUMPOINTS function takes a geometry as an input parameter and returns the number of points that were used to define that geometry. For example, if the geometry is a polygon and five points were used to define that polygon, then the returned number is 5.

If *geometry* is null, the result is the null value. If *geometry* is empty, 0 (zero) is returned.



geometry
A value of type ST_GEOMETRY or one of its subtypes for which the number of points is returned.

The result of the function is INTEGER.

Example

For a variety of geometries in a table, use ST_NUMPOINTS to determine how many points are within each geometry.

```
CREATE TABLE sample_geometries (id VARCHAR(18), geometry QSYS2.ST_GEOMETRY);
```

```
INSERT INTO sample_geometries (id, geometry)
  VALUES (1, QSYS2.ST_POINT('point (44 14)')),
         (2, QSYS2.ST_LINESTRING('linestring (0 0, 20 20)')),
         (3, QSYS2.ST_POLYGON('polygon((0 0, 0 40, 40 40, 40 0, 0 0))')),
         (4, QSYS2.ST_MULTIPOINT('multipoint((0 0), (10 20), (15 20), (30 30))')),
         (5, QSYS2.ST_MULTILINESTRING('MultiLineString((10 10, 20 20), (15 15, 30 15))')),
         (6, QSYS2.ST_MULTIPOLYGON('MultiPolygon(((10 10, 10 20, 20 20, 20 15, 10 10)),
                                         ((60 60, 70 70, 80 60, 60 60 )))'));

SELECT id, QSYS2.ST_GEOMETRYTYPE(geometry) AS spatial_type,
       QSYS2.ST_NUMPOINTS (geometry) AS num_points
  FROM sample_geometries;
```

Results:

```

ID   SPATIAL_TYPE      NUM_POINTS
--   ---------------   ----------
 1   ST_POINT                   1
 2   ST_LINESTRING              2
 3   ST_POLYGON                 5
 4   ST_MULTIPOINT              4
 5   ST_MULTILINESTRING         4
 6   ST_MULTIPOLYGON            9
```

ST_OVERLAPS scalar function

The ST_OVERLAPS function takes two geometries as input parameters. If the intersection of the geometries results in a geometry of the same dimension but is not equal to either of the given geometries, it returns 1. Otherwise, it returns 0 (zero).

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* or *geometry2* is empty, 0 (zero) is returned.

▶▶— ST_OVERLAPS — (— *geometry1* — , — *geometry2* —) —▶◀

geometry1
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is tested to overlap with *geometry2*.

geometry2
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is tested to overlap with *geometry1*.

The result of the function is an INTEGER.

Example

Determine if two lines overlap.

```
VALUES CASE QSYS2.ST_OVERLAPS(QSYS2.ST_LINESTRING('linestring(50 12, 50 10, 60 8)'),
                              QSYS2.ST_LINESTRING('linestring(50 10, 50 12, 45 10)'))
         WHEN 0 THEN 'Lines do not overlap'
         WHEN 1 THEN 'Lines overlap'
       END;
```

Results:

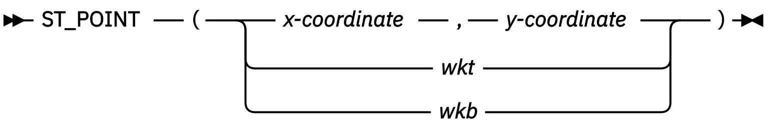
```

Lines overlap
```

ST_POINT scalar function

The ST_POINT function constructs a point from specified inputs.

If *x-coordinate*, *y-coordinate*, *wkt* or *wkb* is null, the result is the null value. An empty point is not supported.



x-coordinate
An expression that returns a numeric value that specifies the X coordinate for the resulting point.

y-coordinate
An expression that returns a numeric value that specifies the Y coordinate for the resulting point.

wkt
An expression that returns a character or graphic string value that contains the well-known text (WKT) format of the resulting point.

wkb
An expression that returns a binary string value that contains the well-known binary (WKB) format of the resulting point.

The result of the function is ST_POINT. If any of the arguments is null, the result is the null value.

For details about the supported formats, see "WKT and WKB data formats" on page 14.

Example

Use the ST_POINT function to insert two points into a table. One point uses x- and y-coordinate values and the other point provides a well-known text (WKT) point representation.

```
CREATE TABLE sample_points (id SMALLINT, geometry QSYS2.ST_POINT);
INSERT INTO sample_points(id, geometry)
  VALUES(10, QSYS2.ST_POINT(10, 20)),
        (20, QSYS2.ST_POINT('point (30 40)'));

SELECT id, QSYS2.ST_ASTEXT(geometry) AS points
  FROM sample_points;
```

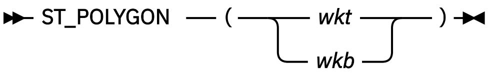
Results:

```
ID     POINTS
----   ----------------------------------
  10   POINT (10.0 20.0)
  20   POINT (30.0 40.0)
```

ST_POLYGON scalar function

The ST_POLYGON function constructs a polygon from a specified input.

If *wkt* or *wkb* is null, the result is the null value. An empty polygon is not supported.



wkt
An expression that returns a character or graphic string value that contains the well-known text (WKT) format of the resulting polygon.

wkb
An expression that returns a binary string value that contains the well-known binary (WKB) format of the resulting polygon.

The result of the function is ST_POLYGON. If the argument is null, the result is the null value.

For details about the supported formats, see "WKT and WKB data formats" on page 14.

Example

Use the ST_POLYGON function to create and insert two polygons from a well-known text polygon representation.

```
CREATE TABLE sample_polygons (id INTEGER, geometry QSYS2.ST_POLYGON);
INSERT INTO sample_polygons VALUES
  (1101, QSYS2.ST_POLYGON('polygon((10 20, 10 40, 20 30, 10 20))')),
  (1102, QSYS2.ST_POLYGON('polygon((10 20, 10 40, 30 40, 30 20, 10 20),
                                   (15 25, 15 35, 25 35, 25 35, 15 25))'));

SELECT id, QSYS2.ST_ASTEXT(geometry) as polygons FROM sample_polygons;
```

Results:

```

ID     POLYGONS
----   --------------------------------------------------------------------------
1101   POLYGON ((10.0 20.0, 20.0 30.0, 10.0 40.0, 10.0 20.0))
1102   POLYGON ((10.0 20.0, 30.0 20.0, 30.0 40.0, 10.0 40.0, 10.0 20.0),
                (15.0 25.0, 15.0 35.0, 25.0 35.0, 25.0 35.0, 15.0 25.0))
```

ST_SRSID scalar function

The ST_SRSID function takes a geometry as an input parameter and returns the integer value of the spatial reference system identifier.

If *geometry* is null, the result is the null value. If *geometry* is empty, the default spatial reference system identifier is returned.

►► ST_SRSID — (— *geometry* —) ►◄

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry for which the spatial reference system identifier is to be set or returned.

The result of the function is INTEGER.

Example

The ID of the spatial reference system that is associated with each point can be found by using the ST_SRSID function.

```
CREATE TABLE sample_points (id INTEGER, geometry QSYS2.ST_POINT);

INSERT INTO sample_points VALUES
  (1, QSYS2.ST_POINT( 'point (80 80)')),
  (2, QSYS2.ST_POINT( 'point (-92.50358 44.05847)'));

SELECT id, QSYS2.ST_SRSID(geometry) srsid
  FROM sample_points;
```

Results:

```

ID     SRSID
----   -------
   1   4326
   2   4326
```

ST_SRSNAME scalar function

The ST_SRSNAME function takes a geometry as an input parameter and returns the name of the spatial reference system in which the specified geometry is represented.

If *geometry* is null, the result is the null value. If *geometry* is empty, the default spatial reference system name is returned.

▶▶─ ST_SRSNAME ── (── *geometry* ──) ─▶◀

geometry
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry for which the name of the spatial reference system is returned.

The result of the function is VARCHAR(128).

Example

The name of the spatial reference system that is associated with each point can be found by using the ST_SRSNAME function.

```
CREATE TABLE sample_points (id INTEGER, geometry QSYS2.ST_POINT);

INSERT INTO sample_points VALUES
  (1, QSYS2.ST_POINT( 'point (80 80)')),
  (2, QSYS2.ST_POINT( 'point (-92.50358 44.05847)'));

SELECT id, QSYS2.ST_SRSNAME(geometry) srsname
  FROM sample_points;
```

Results:

```
ID     SrsName
----   -------
   1   WGS84_SRS_4326
   2   WGS84_SRS_4326
```

ST_SYMDIFFERENCE scalar function

The ST_SYMDIFFERENCE function takes two geometry objects as input parameters and returns the geometry object that is the symmetrical difference of the two geometries. The symmetrical difference is the non-intersecting part of the two specified geometries.

If the geometries are equal, an empty geometry is returned. The resulting geometry is represented in the most appropriate spatial type.

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* and *geometry2* are empty, an empty geometry is returned. If only one of *geometry1* and *geometry2* is empty, the other geometry is returned.

▶▶─ ST_SYMDIFFERENCE ── (── *geometry1* ── , ── *geometry2* ──) ─▶◀

geometry1
A value of type ST_GEOMETRY or one of its subtypes that represents the first geometry to compute the symmetrical difference with *geometry2*.

geometry2
A value of type ST_GEOMETRY or one of its subtypes that represents the second geometry to compute the symmetrical difference with *geometry1*.

The result of the function is ST_GEOMETRY.

Example

Find the symmetrical difference of two polygons.

```
SELECT QSYS2.ST_ASTEXT(QSYS2.ST_SYMDIFFERENCE
                           (QSYS2.ST_POLYGON('polygon((10 10,10 20,20 20,20 10,10 10))'),
                            QSYS2.ST_POLYGON('polygon((30 30,30 50,50 50,50 30,30 30))')))
                                        AS SYMMETRICAL_DIFFERENCE
  FROM SYSIBM.SYSDUMMY1;
```

Results:

```
SYMMETRICAL_DIFFERENCE
-----------------------------------------------------------------------------------------------
GEOMETRYCOLLECTION (POLYGON ((10.0 10.0, 20.0 10.0, 20.0 20.0, 10.0 20.0, 10.0 10.0)),
                    POLYGON ((30.0 30.0, 50.0 30.0, 50.0 50.0, 30.0 50.0, 30.0 30.0)))
```

ST_TOLINESTRING scalar function

The ST_TOLINESTRING function takes a geometry as an input parameter and converts it to a linestring.

The specified geometry must be a linestring.

If *geometry* is null, the result is the null value. An empty linestring is not supported.

▶▶─ ST_TOLINESTRING ─ (─ *geometry* ─) ─▶◀

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is converted to a linestring.

A geometry can be converted to a linestring if it is a linestring. If the conversion cannot be performed, then an exception condition is raised.

The result of the function is ST_LINESTRING.

Example

In this example, the ST_TOLINESTRING function is used to return a linestring converted to ST_LINESTRING from the static type of ST_GEOMETRY.

```
CREATE TABLE sample_geometries  (id INTEGER, geometry QSYS2.ST_GEOMETRY);
CREATE TABLE sample_linestrings (id INTEGER, line QSYS2.ST_LINESTRING);

INSERT INTO sample_geometries
  VALUES (1, QSYS2.ST_LINESTRING ('linestring (0 10, 0  0, 10 0)'));

INSERT INTO sample_linestrings
 (SELECT ID, QSYS2.ST_TOLINESTRING(geometry)
   FROM sample_geometries);

SELECT id, QSYS2.ST_ASTEXT(line) AS lines FROM sample_linestrings;
```

Results:

```
ID          LINES
--------    --------------------
       1    LINESTRING (0.0 10.0, 0.0 0.0, 10.0 0.0)
```

ST_TOMULTILINE scalar function

The ST_TOMULTILINE function takes a geometry as an input parameter and converts it to a multilinestring.

The *geometry* must be a multilinestring or a linestring.

If *geometry* is null, the result is the null value. An empty multilinestring is not supported.

▶▶─ ST_TOMULTILINE ─ (─ *geometry* ─) ─▶◀

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is converted to a multilinestring.

A geometry can be converted to a multilinestring if it a linestring or a multilinestring. If the conversion cannot be performed, an exception condition is raised.

The result of the function is ST_MULTILINESTRING.

Example

In the following SELECT statement, the ST_TOMULTILINE function is used to return multilinestrings converted to ST_MULTILINESTRING from the static type of ST_GEOMETRY.

```
CREATE TABLE sample_geometries (id INTEGER, geometry QSYS2.ST_GEOMETRY);
CREATE TABLE sample_multilinestrings (id INTEGER, multiline QSYS2.ST_MULTILINESTRING);

INSERT INTO sample_geometries
  VALUES (1, QSYS2.ST_MULTILINESTRING ('multilinestring ((0 10, 0 0, 10 0),(23 43, 27 34, 35
12))') ),
         (2, QSYS2.ST_LINESTRING ('linestring  (0 10, 0 0, 10 0)'));

INSERT INTO sample_multilinestrings
 (SELECT ID, QSYS2.ST_TOMULTILINE(geometry)
   FROM sample_geometries);

SELECT id, QSYS2.ST_ASTEXT(multiline) AS multilines FROM sample_multilinestrings;
```

Results:

```

ID        MULTILINES
-------   ----------------------------------------------------------------------------------
      1   MULTILINESTRING ((0.0 10.0, 0.0 0.0, 10.0 0.0), (23.0 43.0, 27.0 34.0, 35.0 12.0))
      2   MULTILINESTRING ((0.0 10.0, 0.0 0.0, 10.0 0.0))
```

ST_TOMULTIPOINT scalar function

The ST_TOMULTIPOINT function takes a geometry as an input parameter and converts it to a multipoint.

The specified geometry must be a point or a multipoint.

If *geometry* is null, the result is the null value. An empty multipoint is not supported.

▶▶─ ST_TOMULTIPOINT ── (── *geometry* ──) ─▶◀

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is converted to a multipoint.

A geometry can be converted to a multipoint if it is a point or a multipoint. If the conversion cannot be performed, then an exception condition is raised.

The result of the function is ST_MULTIPOINT.

Example

In the following SELECT statement, the ST_TOMULTIPOINT function is used to return multipoints converted to ST_MULTIPOINT from the static type of ST_GEOMETRY.

```
CREATE TABLE sample_geometries (id INTEGER, geometry QSYS2.ST_GEOMETRY);
CREATE TABLE sample_multipoints (id INTEGER, multipoint QSYS2.ST_MULTIPOINT);

INSERT INTO sample_geometries
  VALUES (1, QSYS2.ST_MULTIPOINT('multipoint ((0 0), (0 4))')),
         (2, QSYS2.ST_POINT('point (30 40)'));

INSERT INTO sample_multipoints
 (SELECT ID, QSYS2.ST_TOMULTIPOINT(geometry)
   FROM sample_geometries);

SELECT id, QSYS2.ST_ASTEXT(multipoint) AS multipoint FROM sample_multipoints;
```

Results:

```

ID       MULTIPOINT
-------  --------------------------------------------------------------
      1  MULTIPOINT ((0.0 0.0), (0.0 4.0))
      2  MULTIPOINT ((30.0 40.0))
```

ST_TOMULTIPOLYGON scalar function

The ST_TOMULTIPOLYGON function takes a geometry as an input parameter and converts it to a multipolygon.

The specified geometry must be a polygon or a multipolygon.

If *geometry* is null, the result is the null value. An empty multipolygon is not supported.

▶▶— ST_TOMULTIPOLYGON — (— *geometry* —) —▶◀

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is converted to a multipolygon.

A geometry can be converted to a multipolygon if it is a polygon or a multipolygon. If the conversion cannot be performed, then an exception condition is raised.

The result of the function is ST_MULTIPOLYGON.

Example

This example creates several geometries and then uses ST_TOMULTIPOLYGON to return multipolygons.

```
CREATE TABLE sample_geometries (id INTEGER, geometry QSYS2.ST_GEOMETRY);
CREATE TABLE sample_multipolygons (id INTEGER, multipolygon QSYS2.ST_MULTIPOLYGON);

INSERT INTO sample_geometries
  VALUES (1, QSYS2.ST_MULTIPOLYGON('multipolygon (((0 0, 0 4, 5 4, 5 0, 0 0)),
                                               ((10 10, 10 15, 13 17, 10 10)))')),
         (2, QSYS2.ST_POLYGON('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))'));

INSERT INTO sample_multipolygons
 (SELECT ID, QSYS2.ST_TOMULTIPOLYGON(geometry)
   FROM sample_geometries);

SELECT id, QSYS2.ST_ASTEXT(multipolygon) AS multipolygons FROM sample_multipolygons;
```

Results:

```

ID        MULTIPOLYGONS
-------   ------------------------------------------------------------------------
      1   MULTIPOLYGON (((10.0 10.0, 13.0 17.0, 10.0 15.0, 10.0 10.0)),
                        ((0.0 0.0, 5.0 0.0, 5.0 4.0, 0.0 4.0, 0.0 0.0)))
      2   MULTIPOLYGON (((0.0 0.0, 5.0 0.0, 5.0 4.0, 0.0 4.0, 0.0 0.0)))
```

ST_TOPOINT scalar function

The ST_TOPOINT function takes a geometry as an input parameter and converts it to a point.

The specified geometry must be a point.

If *geometry* is null, the result is the null value. An empty point is not supported.

▶▶— ST_TOPOINT — (— *geometry* —) —▶◀

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is converted to a point.

A geometry can be converted to a point if it is a point. If the conversion cannot be performed, an exception condition is raised.

The result of the function is ST_POINT.

Example

This example creates three geometries in SAMPLE_GEOMETRIES and converts each to a point.

```
CREATE TABLE sample_geometries (id INTEGER, geometry QSYS2.ST_GEOMETRY);
CREATE TABLE sample_points (id INTEGER, point QSYS2.ST_POINT);

INSERT INTO sample_geometries
  VALUES (1, QSYS2.ST_POINT('point (30 40)'));

INSERT INTO sample_points
 (SELECT ID, QSYS2.ST_TOPOINT(geometry)
   FROM sample_geometries);

SELECT id, QSYS2.ST_ASTEXT(point) AS points FROM sample_points;
```

Results:

```

ID       POINTS
-------  ----------------------------
      1  POINT (30.0 40.0)
```

ST_TOPOLYGON scalar function

The ST_TOPOLYGON function takes a geometry as an input parameter and converts it to a polygon.

The specified geometry must be a polygon.

If *geometry* is null, the result is the null value. An empty polygon is not supported.

▶▶— ST_TOPOLYGON — (— *geometry* —) —▶◀

geometry

A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is converted to a polygon.

A geometry can be converted to a polygon if it is a polygon. If the conversion cannot be performed, then an exception condition is raised.

The result of the function is ST_POLYGON.

Example

This example creates three geometries in SAMPLE_GEOMETRIES and converts each to a polygon. The ST_TOPOLYGON function is used to return polygons converted to ST_POLYGON from the static type of ST_GEOMETRY.

```
CREATE TABLE sample_geometries (id INTEGER, geometry QSYS2.ST_GEOMETRY);
CREATE TABLE sample_polygons (id INTEGER, polygon QSYS2.ST_POLYGON);

INSERT INTO sample_geometries
  VALUES (1, QSYS2.ST_POLYGON ('polygon ((0 0, 0 4, 5 4, 5 0, 0 0))'));

INSERT INTO sample_polygons
 (SELECT ID, QSYS2.ST_TOPOLYGON(geometry)
   FROM sample_geometries);

SELECT id, QSYS2.ST_ASTEXT(polygon) AS polygons FROM sample_polygons;
```

Results:

```
ID        POLYGONS
--------  --------------------------------------------------------------------
       1  POLYGON ((0.0 0.0, 5.0 0.0, 5.0 4.0, 0.0 4.0, 0.0 0.0))
```

ST_TOUCHES scalar function

The ST_TOUCHES function takes two geometry objects as input parameters and returns the integer value 1 if the specified geometries spatially touch. Otherwise, the integer value 0 (zero) is returned.

Two geometries touch if the interiors of both geometries do not intersect, but the boundary of one of the geometries intersects with either the boundary or the interior of the other geometry.

If both *geometry1* and *geometry2* are points or multipoints, 0 (zero) is returned because points do not have a boundary. If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* or *geometry2* is empty, 0 (zero) is returned.

►►— ST_TOUCHES — (— *geometry1* — , — *geometry2* —) —►◄

geometry1
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is to be tested to touch *geometry2*.

geometry2
A value of type ST_GEOMETRY or one of its subtypes that represents the geometry that is to be tested to touch *geometry1*.

The result of the function is INTEGER.

Example

Use the ST_TOUCHES function to determine which geometries touch each other.

```
CREATE TABLE sample_geometries (id INTEGER, geometry QSYS2.ST_GEOMETRY);

INSERT INTO sample_geometries VALUES
  (1, QSYS2.ST_POLYGON('polygon ( (20 30, 30 30, 30 40, 20 40, 20 30) )')),
  (2, QSYS2.ST_POLYGON('polygon ( (30 30, 30 50, 50 50, 50 30, 30 30) )')),
  (3, QSYS2.ST_POLYGON('polygon ( (40 40, 40 60, 60 60, 60 40, 40 40) )')),
  (4, QSYS2.ST_LINESTRING('linestring( 60 60, 70 70 )')),
  (5, QSYS2.ST_LINESTRING('linestring( 30 30, 60 60 )'));

SELECT a.id, b.id, QSYS2.ST_TOUCHES(a.geometry, b.geometry) touches
  FROM sample_geometries a, sample_geometries b
  WHERE b.id >= a.id;
```

Results:

```
ID    ID    TOUCHES
---   ---   -------
  1     1         0
  1     2         1
  1     3         0
  1     4         0
  1     5         1
  2     2         0
  2     3         0
  2     4         0
  2     5         0
  3     3         0
  3     4         1
  3     5         0
  4     4         0
  4     5         1
  5     5         0
```

ST_UNION scalar function

The ST_UNION function takes two geometry objects as input parameters and returns the geometry object that is the union of the specified geometries.

The resulting geometry is represented in the most appropriate spatial type. If it can be represented as a point, linestring, or polygon, then one of those types is used. Otherwise, the multipoint, multilinestring, or multipolygon type is used.

If *geometry1* or *geometry2* is null, the result is the null value. If one of *geometry1* and *geometry2* is empty, the other geometry is returned. If both *geometry1* and *geometry2* are empty, an empty geometry is returned.

▸▸— ST_UNION — (— *geometry1* — , — *geometry2* —) —▸◂

geometry1
A value of type ST_GEOMETRY or one of its subtypes that is combined with *geometry2*.

geometry2
A value of type ST_GEOMETRY or one of its subtypes that is combined with *geometry1*.

The result of the function is ST_GEOMETRY.

Example

Find the union of two intersecting polygons.

```
VALUES QSYS2.ST_ASTEXT(QSYS2.ST_UNION(
                          QSYS2.ST_POLYGON('polygon((30 30, 30 50, 50 50, 50 30, 30 30))'),
                          QSYS2.ST_POLYGON('polygon((40 40, 40 60, 60 60, 60 40, 40 40))')));
```

Result:

```
POLYGON ((50.0 40.432460999999996, 60.0 40.0, 60.0 60.0, 40.0 60.0, 40.0 50.431312999999996,
          30.0 50.0, 30.0 30.0, 50.0 30.0, 50.0 40.432460999999996))
```

ST_WITHIN scalar function

The ST_WITHIN function takes two geometries as input parameters and returns 1 if the first geometry is completely within the second geometry. Otherwise, 0 (zero) is returned.

ST_WITHIN performs the same logical operation that ST_CONTAINS performs with the parameters reversed. ST_WITHIN returns the exact opposite result of ST_CONTAINS.

If *geometry1* or *geometry2* is null, the result is the null value. If *geometry1* or *geometry2* is empty, 0 (zero) is returned.

▸▸— ST_WITHIN — (— *geometry1* — , — *geometry2* —) —▸◂

geometry1
A value of type ST_GEOMETRY or one of its subtypes that is to be tested to be fully within *geometry2*.

geometry2
A value of type ST_GEOMETRY or one of its subtypes that is to be tested to fully contain *geometry1*.

The result of the function is INTEGER.

Notes

One geometry (*geometry1*) in within another (*geometry2*) if the interiors of the geometries intersect and the interior or boundary of *geometry1* does not intersect the exterior of *geometry2*.

Example

Find points from the SAMPLE_POINTS table that are in the polygons in the SAMPLE_POLYGONS table.

```
CREATE TABLE sample_points (id INTEGER, geometry QSYS2.ST_POINT);
CREATE TABLE sample_polygons (id INTEGER, geometry QSYS2.ST_POLYGON);

INSERT INTO sample_points (id, geometry)
  VALUES (1, QSYS2.ST_POINT(10, 20)),
         (2, QSYS2.ST_POINT('point (41 41)')),
         (3, QSYS2.ST_POINT('point (1 1)'));

INSERT INTO sample_polygons (id, geometry)
  VALUES (100, QSYS2.ST_POLYGON('polygon (( 0 0, 0 40, 40 40, 40 0, 0 0))'));

SELECT QSYS2.ST_ASTEXT(a.geometry) AS point, QSYS2.ST_ASTEXT(b.geometry) AS polygon
  FROM sample_points a, sample_polygons b
  WHERE QSYS2.ST_WITHIN(a.geometry, b.geometry) = 1;
```

Results:

```

POINT                POLYGON
-----------------    -----------------------------------------------------------
POINT (10.0 20.0)    POLYGON ((0.0 0.0, 40.0 0.0, 40.0 40.0, 0.0 40.0, 0.0 0.0))
POINT (1.0 1.0)      POLYGON ((0.0 0.0, 40.0 0.0, 40.0 40.0, 0.0 40.0, 0.0 0.0))
```

ST_WKBTOSQL scalar function

The ST_WKBTOSQL function takes a well-known binary (WKB) format of a geometry as an input parameter and returns the corresponding geometry.

If *wkb* is null, the result is the null value. There is no WKB representation for an empty geometry. For details about the supported formats, see "WKT and WKB data formats" on page 14

The ST_WKBTOSQL function is identical to the ST_GEOMETRY(*wkb*) function.

Syntax

▶▶— ST_WKBTOSQL — (— *wkb* —) —▶◀

Parameters

wkb

A value of type VARBINARY or BLOB that contains the WKB format of the resulting geometry.

Return type

The result of the function is ST_GEOMETRY.

Example

The ST_WKBTOSQL function is used to return the coordinates of the geometries in the WKB column.

```
CREATE TABLE sample_geometries(id INTEGER, wkb BLOB(32K));

INSERT INTO sample_geometries VALUES
 (10, QSYS2.ST_ASBINARY(QSYS2.ST_POINT('point (44 14)'))),
 (11, QSYS2.ST_ASBINARY(QSYS2.ST_POINT('point (24 13)'))),
 (12, QSYS2.ST_ASBINARY(QSYS2.ST_POLYGON('polygon ((50 20, 50 40, 70 30, 50 20))')));

SELECT id, QSYS2.ST_ASTEXT(QSYS2.ST_WKBTOSQL(wkb)) AS geometry FROM sample_geometries;
```

Results:

```

ID      GEOMETRY
```

```
-----   -----------------------------------------------------
   10   POINT (44.0 14.0)
   11   POINT (24.0 13.0)
   12   POLYGON ((50.0 20.0, 70.0 30.0, 50.0 40.0, 50.0 20.0))
```

ST_WKTTOSQL scalar function

The ST_WKTTOSQL function takes a well-known text (WKT) format of a geometry and returns the corresponding geometry.

If *wkt* is null, the result is the null value. If *wkt* is 'POINT EMPTY', an empty geometry is returned. For details about the supported formats, see “WKT and WKB data formats” on page 14

The ST_WKTTOSQL function is identical to the ST_GEOMETRY(*wkt*) function.

►► ST_WKTTOSQL — (— *wkt* —) ►◄

wkt
A value of type VARCHAR or CLOB that contains the WKT format of the resulting geometry.

The result of the function is ST_GEOMETRY.

Example

Use the ST_WKTTOSQL to create and insert geometries using their well-known text representations.

```
CREATE TABLE sample_geometries (id INTEGER, geometry QSYS2.ST_GEOMETRY);

INSERT INTO sample_geometries
  VALUES (10, QSYS2.ST_WKTTOSQL('point (44 14)' )),
         (11, QSYS2.ST_WKTTOSQL('point (24 13)' )),
         (12, QSYS2.ST_WKTTOSQL('polygon ((50 20, 50 40, 70 30, 50 20))'));

SELECT id, CAST(QSYS2.ST_ASTEXT(geometry) AS VARCHAR(120) ) geometries
  FROM sample_geometries;
```

Results:

```
ID      GEOMETRIES
-----   -----------------------------------------------------
   10   POINT (44.0 14.0)
   11   POINT (24.0 13.0)
   12   POLYGON ((50.0 20.0, 70.0 30.0, 50.0 40.0, 50.0 20.0))
```

Geospatial Analytics catalog views

Use the geospatial catalog views to obtain useful information about your geospatial data.

ST_COORDINATE_SYSTEMS catalog view

The ST_COORDINATE_SYSTEMS catalog view contains information about coordinate systems.

Authorization: None required.

The following table describes the columns in the view. The system name is ST_COORD. The schema is QSYS2.

Table 24. ST_COORDINATE_SYSTEMS view

Column Name	System Column Name	Data Type	Description
COORDSYS_NAME	COORD_NAME	VARCHAR(128)	Name of this coordinate system. The name is unique within the database.
COORDSYS_TYPE	COORD_TYPE	VARCHAR(128)	Type of this coordinate system: **GEOGRAPHIC** Three-dimensional. Uses X and Y coordinates.

Table 24. ST_COORDINATE_SYSTEMS view (continued)

Column Name	System Column Name	Data Type	Description
DEFINITION	DEFINITION	VARGRAPHIC(2048) CCSID 1200	Well-known text format of the definition of this coordinate system.
ORGANIZATION	ORG	VARCHAR(256) Nullable	Name of the organization (for example, a standards body such as the European Petrol Survey Group, or EPSG) that defined this coordinate system. Contains the null value if ORGANIZATION_COORDSYS_ID is null.
ORGANIZATION_ COORDSYS_ID	ORG_ID	INTEGER Nullable	Numeric identifier assigned to this coordinate system by the organization that defined the coordinate system. When not null, the combination of this identifier and the value in the ORGANIZATION column uniquely identify the coordinate system. Contains the null value if there is no ID assigned to the coordinate system.

ST_GEOMETRY_COLUMNS catalog view

The ST_GEOMETRY_COLUMNS catalog view returns all columns in all tables on the system that are defined with a geospatial data type.

For additional information about these columns, query the QSY2.SYSCOLUMNS2 catalog view.

Authorization: The view is shipped with *PUBLIC *EXCLUDE.

The following table describes the columns in the view. The system name is ST_GEOCOLS. The schema is QSYS2.

Table 25. ST_GEOMETRY_COLUMNS view

Column Name	System Column Name	Data Type	Description
COLUMN_NAME	COL_NAME	VARCHAR(128)	The name of a column that is defined as a geospatial type.
TABLE_SCHEMA	TABSCHEMA	VARCHAR(128)	That name of the schema that contains the table.
TABLE_NAME	TABNAME	VARCHAR(128)	The name of the table that contains the column.
DATA_TYPE_SCHEMA	TYPESCHEMA	VARCHAR(128)	The schema containing the geospatial data type.
DATA_TYPE_NAME	TYPENAME	VARCHAR(128)	The name of the geospatial data type.
SPATIAL_REFERENCE_SYSTEM _NAME	SRS_NAME	VARCHAR(128)	Name of the spatial reference system associated with this spatial reference.
SPATIAL_REFERENCE_SYSTEM _ID	SRS_ID	INTEGER	Numeric identifier of the spatial reference system.
SYSTEM_COLUMN_NAME	SYS_CNAME	VARCHAR(10)	The system name for the column.
SYSTEM_TABLE_SCHEMA	LIB_NAME	VARCHAR(10)	The system name for the schema.
SYSTEM_TABLE_NAME	FILE_NAME	VARCHAR(10)	The system name for the table.

ST_SPATIAL_REFERENCE_SYSTEMS catalog view

The ST_SPATIAL_REFERENCE_SYSTEMS catalog view contains information about registered spatial reference systems.

Each spatial reference system represents a coordinate system. The spatial reference system also includes factors to convert coordinates that use the coordinate system into values that the database can process with maximum efficiency. Db2 for i supports a single spatial reference system.

Authorization: None required.

The following table describes the columns in the view. The system name is ST_REFSYS. The schema is QSYS2.

Table 26. ST_SPATIAL_REFERENCE_SYSTEMS view

Column Name	System Column Name	Data Type	Description
SPATIAL_REFERENCE_SYSTEM_NAME	SRS_NAME	VARCHAR(128)	Name of the spatial reference system. This name is unique within the database.
SPATIAL_REFERENCE_SYSTEM_ID	SRS_ID	INTEGER	Numeric identifier of the spatial reference system. Each spatial reference system has a unique numeric identifier.
MIN_X	MIN_X	DOUBLE	Minimum possible value for X coordinates in the geometries to which this spatial reference system applies.
MAX_X	MAX_X	DOUBLE	Maximum possible value for X coordinates in the geometries to which this spatial reference system applies.
MIN_Y	MIN_Y	DOUBLE	Minimum possible value for Y coordinates in the geometries to which this spatial reference system applies.
MAX_Y	MAX_Y	DOUBLE	Maximum possible value for Y coordinates in the geometries to which this spatial reference system applies.
COORDSYS_NAME	COORD_NAME	VARCHAR(128)	Identifying name of the coordinate system on which this spatial reference system is based.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
2800 37th Street NW
Rochester, MN 55901-4441
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming interface information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux® is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Oracle, Inc. in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

Personal Use: You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use: You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Product Number: 5770-SS1