Geospatial data types

Snowflake offers native support for geospatial features such as points, lines, and polygons on the Earth's surface.

Tip
You can use the search optimization service to improve query performance. For details, see Search Optimization Service.

Data types

Snowflake provides the following data types for geospatial data:

- The GEOGRAPHY data type, which models Earth as though it were a perfect sphere.
- The GEOMETRY data type, which represents features in a planar (Euclidean, Cartesian) coordinate system.

GEOGRAPHY data type

The GEOGRAPHY data type follows the WGS 84 standard (spatial reference ID 4326; for details, see https://epsg.io/4326).

Altitude currently isn't supported.

Line segments are interpreted as great circle arcs on the Earth's surface.

Snowflake also provides geospatial functions that operate on the GEOGRAPHY data type.

If you have geospatial data (for example, longitude and latitude data, WKT, WKB, GeoJSON, and so on), we suggest converting and storing this data in GEOGRAPHY columns, rather than keeping the data in their original formats in VARCHAR, VARIANT or NUMBER columns. Storing your data in GEOGRAPHY columns can significantly improve the performance of queries that use geospatial functionality.

GEOMETRY data type

The GEOMETRY data type represents features in a planar (Euclidean, Cartesian) coordinate system.

The coordinates are represented as pairs of real numbers (x, y). Currently, only 2D coordinates are supported.

The units of the X and Y are determined by the spatial reference system (SRS) associated with the GEOMETRY object. The spatial reference system is identified by the spatial reference system identifier (SRID) number. Unless the SRID is provided when creating the GEOMETRY object or by calling ST_SETSRID, the SRID is 0.

Snowflake uses 14 decimal places to store GEOMETRY coordinates. When the data includes decimal places exceeding this limit, the coordinates are rounded to ensure compliance with the specified length constraint.

Snowflake provides a set of geospatial functions that operate on the GEOMETRY data type. For these functions:

- For functions that accept multiple GEOMETRY expressions as arguments (for example, ST_DISTANCE), the input expressions must be defined in the same SRS.

Geospatial input and output

The following sections cover the supported standard formats and object types when reading and writing geospatial data.

- Supported standard input and output formats
- Supported geospatial object types
- Specifying the output format for result sets
- Examples of inserting and querying GEOGRAPHY data

Supported standard input and output formats

The GEOGRAPHY and GEOMETRY data types support the following standard industry formats for input and output:

- Well-Known Text (WKT)
- Well-Known Binary (WKB)
- Extended WKT and WKB (EWKT and EWKB) (see the note on EWKT and EWKB handling)
- IETF GeoJSON (see the note on GeoJSON handling)

You might also find the following Open Geospatial Consortium's Simple Feature Access references helpful:

- SQL Option

Any departure from these standards is noted explicitly in the Snowflake documentation.

GeoJSON handling for GEOGRAPHY values

The WKT and WKB standards specify a format only. The semantics of WKT/WKB objects depend on the reference system (for example, a plane or a sphere).

The GeoJSON standard, on the other hand, specifies both a format and its semantics: GeoJSON points are explicitly WGS 84 coordinates, and GeoJSON line segments are planar edges (straight lines).

Contrary to that, the Snowflake GEOGRAPHY data type interprets all line segments, including those input from or output to GeoJSON format, as great circle arcs. In essence, Snowflake treats GeoJSON as JSON-formatted WKT with spherical semantics.

EWKT and EWKB handling for GEOGRAPHY values

EWKT and EWKB are non-standard formats introduced by PostGIS. They enhance the WKT and WKB formats by including a spatial reference system identifier (SRID), which specifies the coordinate reference system to use with the data. Snowflake currently supports only WGS84, which maps to SRID=4326.

By default, Snowflake issues an error if an EWKB or EWKT input value contains an SRID other than 4326. Conversely, all EWKB and EWKT output values have SRID=4326.

Supported geospatial object types

The GEOGRAPHY and GEOMETRY data types can store the following types of geospatial objects:

- WKT / WKB / EWKT / EWKB / GeoJSON geospatial objects:
  - Point
  - MultiPoint
  - LineString
  - MultiLineString
  - Polygon
  - MultiPolygon
  - GeometryCollection
- These GeoJSON-specific geospatial objects:
  - Feature
  - FeatureCollection

Specifying the output format for result sets

The session parameters GEOGRAPHY_OUTPUT_FORMAT and GEOMETRY_OUTPUT_FORMAT control the rendering of GEOGRAPHY and GEOMETRY columns in result sets (respectively).

These parameters can have one of the following values:

Parameter value	Description
`GeoJSON` (default)	The GEOGRAPHY / GEOMETRY result is rendered as an OBJECT in GeoJSON format.
`WKT`	The GEOGRAPHY / GEOMETRY result is rendered as a VARCHAR in WKT format.
`WKB`	The GEOGRAPHY / GEOMETRY result is rendered as a BINARY in WKB format.
`EWKT`	The GEOGRAPHY / GEOMETRY result is rendered as a VARCHAR in EWKT format.
`EWKB`	The GEOGRAPHY / GEOMETRY result is rendered as a BINARY in EWKB format.

For `EWKT` and `EWKB`, the SRID is always 4326 in the output. See EWKT and EWKB handling for GEOGRAPHY values.

This parameter affects all clients, including the Snowflake UI and the SnowSQL command-line client, as well as the JDBC, ODBC, Node.js, Python, and so on drivers and connectors.

For example, the JDBC Driver returns the following metadata for a GEOGRAPHY-typed result column (column `i` in this example):

- If `GEOGRAPHY_OUTPUT_FORMAT='GeoJSON'` or `GEOMETRY_OUTPUT_FORMAT='GeoJSON'`:
  - `ResultSetMetaData.getColumnType(i)` returns `java.sql.Types.VARCHAR`.
  - `ResultSetMetaData.getColumnClassName(i)` returns `"java.lang.String"`.
- If `GEOGRAPHY_OUTPUT_FORMAT='WKT'` or `'EWKT'`, or if: `GEOMETRY_OUTPUT_FORMAT='WKT'` or `'EWKT'`:

- If `GEOGRAPHY_OUTPUT_FORMAT='WKB'` or `'EWKB'`, or if `GEOMETRY_OUTPUT_FORMAT='WKB'` or `'EWKB'`:
  - `ResultSetMetaData.getColumnType(i)` returns `java.sql.Types.BINARY`.
  - `ResultSetMetaData.getColumnClassName(i)` returns `"[B"` (array of byte).

Note

APIs for retrieving database-specific type names (`getColumnTypeName` in JDBC and the `SQL_DESC_TYPE_NAME` descriptor in ODBC) always return `GEOGRAPHY` and `GEOMETRY` for the type name, regardless of the values of the `GEOGRAPHY_OUTPUT_FORMAT` and `GEOMETRY_OUTPUT_FORMAT` parameters. For details, see:

- Snowflake-specific behavior in the JDBC Driver documentation.
- Retrieving results and information about results in the ODBC Driver documentation.

Examples of inserting and querying GEOGRAPHY data

The code below shows sample input and output for the GEOGRAPHY data type. Note the following:

- For the coordinates in WKT, EWKT, and GeoJSON, longitude appears before latitude (for example, `POINT(<lon> <lat>)`).
- For the WKB and EWKB output, it is assumed that the BINARY_OUTPUT_FORMAT parameter is set to `HEX` (the default value for the parameter).

The following example creates a table with a GEOGRAPHY column, inserts data in WKT format, and returns the data in different output formats.

```
CREATE OR REPLACE TABLE geospatial_table (id INTEGER, g GEOGRAPHY);
INSERT INTO geospatial_table VALUES
  (1, 'POINT(-122.35 37.55)'),
  (2, 'LINESTRING(-124.20 42.00, -120.01 41.99)');
```

```
ALTER SESSION SET GEOGRAPHY_OUTPUT_FORMAT='GeoJSON';
```

```
SELECt g
  FROM geospatial_table
  ORDER BY id;
```

```
+-------------------------+
| G                       |
|-------------------------|
| {                       |
|   "coordinates": [      |
|     -122.35,            |
|     37.55               |
|   ],                    |
|   "type": "Point"       |
| }                       |
| {                       |
|   "coordinates": [      |
|     [                   |
|       -124.2,           |
```

```
|       41.99             |
|     ]                   |
|   ],                    |
|   "type": "LineString" |
| }                       |
+-------------------------+
```

```
ALTER SESSION SET GEOGRAPHY_OUTPUT_FORMAT='WKT';
```

```
SELECt g
  FROM geospatial_table
  ORDER BY id;
```

```
+-------------------------------------+
| G                                   |
|-------------------------------------|
| POINT(-122.35 37.55)                |
| LINESTRING(-124.2 42,-120.01 41.99) |
+-------------------------------------+
```

```
ALTER SESSION SET GEOGRAPHY_OUTPUT_FORMAT='WKB';
```

```
  ORDER BY id;
```

```
+-----------------------------------------------------------------------------------------+
| G                                                                                       |
|-----------------------------------------------------------------------------------------|
| 0101000000666666666965EC06666666666C64240                                               |
| 0102000000020000000CDCCCCCCCCC0C5FC0000000000000004540713D0AD7A3005EC01F85EB51B8FE4440 |
+-----------------------------------------------------------------------------------------+
```

```
ALTER SESSION SET GEOGRAPHY_OUTPUT_FORMAT='EWKT';
```

```
SELECt g
  FROM geospatial_table
  ORDER BY id;
```

```
+-----------------------------------------------+
| G                                             |
|-----------------------------------------------|
| SRID=4326;POINT(-122.35 37.55)                |
| SRID=4326;LINESTRING(-124.2 42,-120.01 41.99) |
+-----------------------------------------------+
```

```
ALTER SESSION SET GEOGRAPHY_OUTPUT_FORMAT='EWKB';
```

```
SELECt g
  FROM geospatial_table
  ORDER BY id;
```

```
+-----------------------------------------------------------------------------------------+
| G                                                                                       |
|-----------------------------------------------------------------------------------------|
| 0101000020E6100000666666666965EC0666666666C64240                                        |
| 0102000020E610000002000000CDCCCCCCCC0C5FC0000000000000004540713D0AD7A3005EC01F85EB51B8FE4440 |
+-----------------------------------------------------------------------------------------+
```

Using geospatial data in Snowflake

The following sections cover the supported standard formats and object types when reading and writing geospatial data.

- Understanding the effects of using different SRIDs with GEOMETRY
- Changing the spatial reference system (SRS) and SRID of a GEOMETRY object
- Performing DML operations on GEOGRAPHY and GEOMETRY columns
- Loading geospatial data from stages
- Using geospatial data with Java UDFs
- Using geospatial data with JavaScript UDFs
- Using geospatial data with Python UDFs

Understanding the effects of using different SRIDs with GEOMETRY

In a GEOMETRY column, you can insert objects that have different SRIDs. If the column contains more than one SRID, some of the important performance optimizations aren't applied. This can result in slower queries, in particular when joining on a geospatial predicate.

Changing the spatial reference system (SRS) and SRID of a GEOMETRY object

To change the SRS and SRID of an existing GEOMETRY object, call the ST_TRANSFORM function, passing in the new SRID. The function returns a new GEOMETRY object with the new SRID and the coordinates converted to use the SRS. For example, to return a GEOMETRY object for `geometry_expression` that uses the SRS for SRID 32633, execute the following statement:

```
SELECT ST_TRANSFORM(geometry_expression, 32633);
```

If the original SRID isn't set correctly in the existing GEOMETRY object, specify the original SRID as an additional argument. For example, if `geometry_expression` is a GEOMETRY object that uses the SRID 4326, and you want to transform this to use the SRID 28992, execute the following statement:

```
SELECT ST_TRANSFORM(geometry_expression, 4326, 28992);
```

If a GEOMETRY object uses the correct coordinates for a SRS but has the wrong SRID, you can fix the SRID by calling the ST_SETSRID function. For example, the following statement sets the SRID for `geometry_expression` to 4326, while leaving

```
SELECT ST_SETSRID(geometry_expression, 4326);
```

Performing DML operations on GEOGRAPHY and GEOMETRY columns

When a GEOGRAPHY or GEOMETRY column is the target of a DML operation (INSERT, COPY, UPDATE, MERGE, or CREATE TABLE AS...), the column's source expression can be any of the following types:

- GEOGRAPHY or GEOMETRY : An expression of type GEOGRAPHY or GEOMETRY is usually the result of a parsing function, a constructor function, or an existing GEOGRAPHY or GEOMETRY column. For a complete list of supported functions and categories of functions, see Geospatial functions.
- VARCHAR: Interpreted as a WKT, WKB (in hex format), EWKT, EWKB (in hex format), or GeoJSON formatted string (see TO_GEOGRAPHY(VARCHAR)).
- BINARY: Interpreted as a WKB binary (see TO_GEOGRAPHY(BINARY) and
- TO_GEOMETRY(BINARY)).
- VARIANT: Interpreted as a GeoJSON object (see TO_GEOGRAPHY(VARIANT) and TO_GEOMETRY(VARIANT)).

Loading geospatial data from stages

You can load data from CSV or JSON/AVRO files in a stage directly (that is, without copy transforms) into a GEOGRAPHY column.

See also GeoJSON handling for GEOGRAPHY values.

Loading data from other file formats (Parquet, ORC, and so on) is possible through a COPY transform.

Using geospatial data with Java UDFs

Java UDFs allow the GEOGRAPHY type as an argument and as a return value. See SQL-Java Data Type Mappings and Passing a GEOGRAPHY value to an in-line Java UDF for details.

Using geospatial data with JavaScript UDFs

JavaScript UDFs allow the GEOGRAPHY or GEOMETRY type as an argument and as a return value.

If a JavaScript UDF has an argument of type GEOGRAPHY or GEOMETRY, that argument is visible as a JSON object in GeoJSON format inside the UDF body.

If a JavaScript UDF returns GEOGRAPHY or GEOMETRY, the UDF body is expected to return a JSON object in GeoJSON format.

For example, these two JavaScript UDFs are roughly equivalent to the built-in functions ST_X and ST_MAKEPOINT:

```
CREATE OR REPLACE FUNCTION my_st_x(g GEOGRAPHY) RETURNS REAL
LANGUAGE JAVASCRIPT
```

```
      throw "Not a point"
  }
  return G["coordinates"][0]
$$;

CREATE OR REPLACE FUNCTION my_st_makepoint(lng REAL, lat REAL) RETURNS GEOGRAPHY
LANGUAGE JAVASCRIPT
AS
$$
  g = {}
  g["type"] = "Point"
  g["coordinates"] = [ LNG, LAT ]
  return g
$$;
```

Using geospatial data with Python UDFs

Python UDFs allow the GEOGRAPHY and GEOMETRY type as an argument and as a return value.

If a Python UDF has an argument of type GEOGRAPHY or GEOMETRY, that argument is represented as a GeoJSON object, which is converted to a Python `dict` object inside the UDF body.

If a Python UDF returns GEOGRAPHY or GEOMETRY, the UDF body is expected to return a Python `dict` object that complies with the structure of GeoJSON.

For example, this Python UDF returns the number of distinct geometries that constitute a composite GEOGRAPHY type:

```
RUNTIME_VERSION = 3.10
PACKAGES = ('shapely')
HANDLER = 'udf'
AS $$
from shapely.geometry import shape, mapping
def udf(geo):
    if geo['type'] not in ('MultiPoint', 'MultiLineString', 'MultiPolygon', 'GeometryCollection'):
        raise ValueError('Must be a composite geometry type')
    else:
        g1 = shape(geo)
        return len(g1.geoms)
$$;
```

Check Snowflake Labs for more samples of Python UDFs. Some of them enable complex spatial manipulations or simplify data ingestion. For example, this UDF allows reading formats that aren't supported natively, such as Shapefiles (.SHP), TAB, KML, GPKG, and others.

Note

The code samples in Snowflake Labs are intended solely for reference and educational purposes. These code samples aren't covered by any Service Level Agreement.

Using GEOGRAPHY objects with H3

H3 is a hierarchical geospatial index that partitions the world into hexagonal cells in a discrete global grid system.

Snowflake provides SQL functions that enable you to use H3 with GEOGRAPHY objects. You can use these functions to:

- Get the IDs of the H3 cells that have centroids within a GEOGRAPHY object that represents a Polygon.
- Get the GEOGRAPHY object that represents the boundary of an H3 cell.
- Get the parents and children of a given H3 cell.
- Get the longitude and latitude of the centroid of an H3 cell (and vice versa).
- Get the resolution of an H3 cell.
- Get the hexadecimal representation of an H3 cell ID (and vice versa).

For more information about these functions, see Geospatial functions.

Choosing the geospatial data type to use (GEOGRAPHY or GEOMETRY)

The next sections explain the differences between the GEOGRAPHY and GEOMETRY data types:

- Understanding the differences between GEOGRAPHY and GEOMETRY
- Examples comparing the GEOGRAPHY and GEOMETRY data types
- Understanding the differences in input data validation

Understanding the differences between GEOGRAPHY and GEOMETRY

Although both the GEOGRAPHY and GEOMETRY data types define geospatial features, the types use different models.

GEOGRAPHY data type	GEOMETRY data type
• Defines features on a sphere. • Only the WGS84 coordinate system. SRID is always 4326. • Coordinates are latitude (-90 to 90) and longitude (-180 to 180) in degrees. • Results of measurement operations (ST_LENGTH, ST_AREA, and so on) are in meters. • Segments are interpreted as great circle arcs on the Earth's surface.	• Defines features on a plane. • Any coordinate system is supported. • Unit of coordinate values are defined by the spatial reference system. • Results of measurement operations (ST_LENGTH, ST_AREA, and so on) are in the same unit as coordinates. For example, if the input coordinates are in degrees, the results are in degrees. • Segments are interpreted as straight lines on the plane.

Examples comparing the GEOGRAPHY and GEOMETRY data types

The following examples compare the output of the geospatial functions when using the GEOGRAPHY and GEOMETRY data types as input.

Example 1: Querying the distance between Berlin and San Francisco

The following table compares the output of ST_DISTANCE for GEOGRAPHY types and GEOMETRY types:

ST_DISTANCE using GEOGRAPHY input

```
SELECT ST_DISTANCE(
ST_POINT(13.4814, 52.5015),
ST_POINT(-121.8212, 36.8252))
AS distance_in_meters;
```

```
+--------------------+
| DISTANCE_IN_METERS |
|--------------------|
|   9182410.99227821 |
+--------------------+
```

ST_DISTANCE using GEOMETRY input

```
SELECT ST_DISTANCE(
ST_GEOMPOINT(13.4814, 52.5015),
ST_GEOMPOINT(-121.8212, 36.8252))
AS distance_in_degrees;
```

```
+---------------------+
| DISTANCE_IN_DEGREES |
|---------------------|
|       136.207708844 |
+---------------------+
```

As shown in the example above:

- With GEOGRAPHY input values, the input coordinates are in degrees, and the output value is in meters. (The result is 9,182 km.)
- With GEOMETRY input values, the input coordinates and output value are degrees. (The result is 136.208 degrees.)

Example 2: Querying the area of Germany

ST_AREA using GEOGRAPHY input

```
SELECT ST_AREA(border) AS
area_in_sq_meters
FROM world_countries
WHERE name = 'Germany';
```

```
+--------------------+
| AREA_IN_SQ_METERS  |
|--------------------|
|  356379183635.591  |
+--------------------+
```

ST_AREA using GEOMETRY input

```
SELECT ST_AREA(border) as
area_in_sq_degrees
FROM world_countries_geom
WHERE name = 'Germany';
```

```
+---------------------+
| AREA_IN_SQ_DEGREES  |
|---------------------|
|       45.930026848  |
+---------------------+
```

As shown in the example above:

- With GEOGRAPHY input values, the input coordinates are in degrees, the output value is in square meters. (The result is 356,379 km^2.)
- With GEOMETRY input values, the input coordinates in degrees, and the output value is in square degrees. (The result is 45.930 square degrees.)

Example 3: Querying the names of countries overlapping the line from Berlin to San Francisco

ST_INTERSECTS using GEOGRAPHY input

```
SELECT name FROM world_countries WHERE
ST_INTERSECTS(border,
TO_GEOGRAPHY(
'LINESTRING(13.4814 52.5015, -121.8212
36.8252)'
));
```

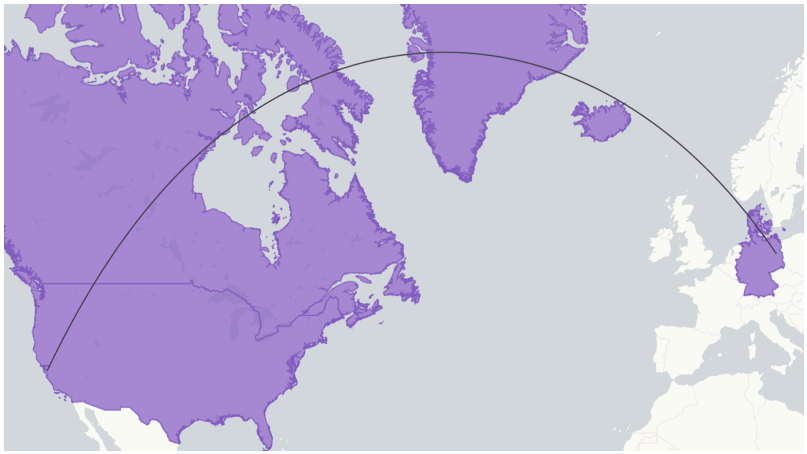
```
+---------------------------+
| NAME                      |
|---------------------------|
|                   Germany |
|                   Denmark |
|                   Iceland |
|                 Greenland |
|                    Canada |
| United States of America  |
+---------------------------+
```

ST_INTERSECTS using GEOMETRY input

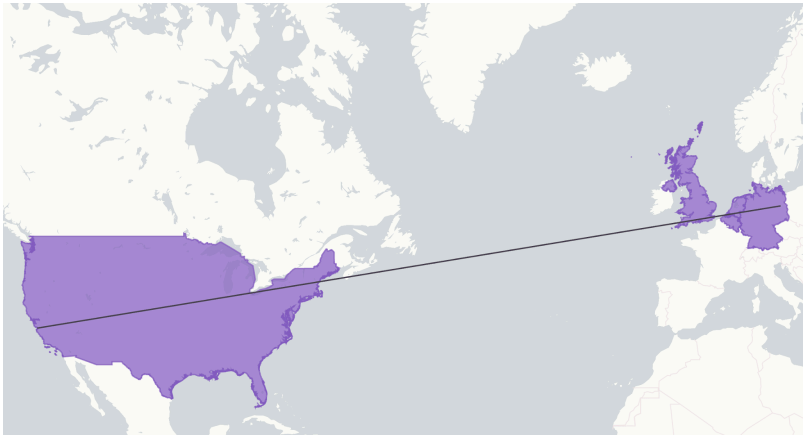
```
SELECT name FROM world_countries_geom
WHERE
ST_INTERSECTS(border,
TO_GEOMETRY(
'LINESTRING(13.4814 52.5015, -121.8212
36.8252)'
));
```

```
+---------------------------+
| NAME                      |
|---------------------------|
|                   Germany |
|                   Belgium |
|               Netherlands |
|           United Kingdom  |
| United States of America  |
+---------------------------+
```

ST_INTERSECTS using GEOGRAPHY input



ST_INTERSECTS using GEOMETRY input



Understanding the differences in input data validation

To create a GEOMETRY or GEOGRAPHY object for an input shape, you must use a shape that is well-formed and valid, according to the OGC rules for Simple Features. The next sections explain how the validity of input data differs between GEOMETRY and GEOGRAPHY.

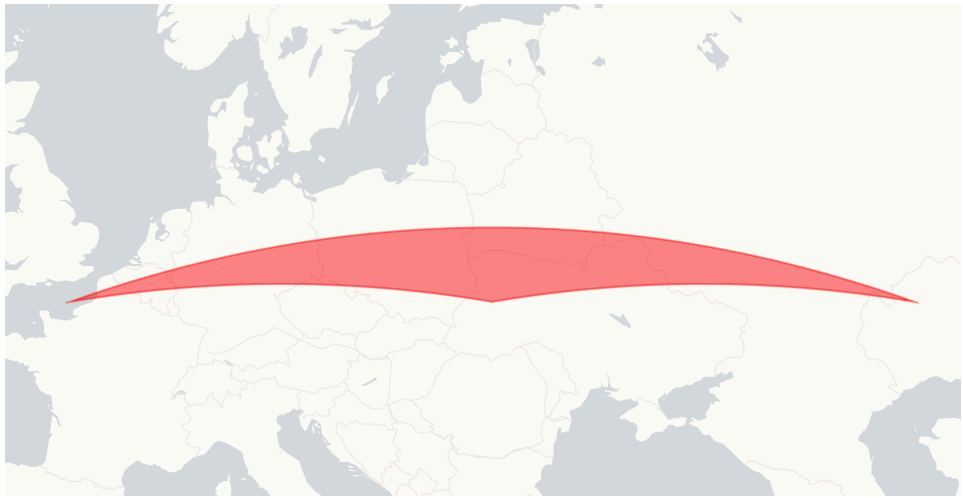
A shape can be valid GEOGRAPHY but invalid GEOMETRY

A given shape can be a valid GEOGRAPHY object but an invalid GEOMETRY object (and vice versa).

```
POLYGON((0 50, 25 50, 50 50, 0 50))
```

In the Cartesian domain, this polygon degrades to a line and, as a result, is invalid.

However, on a sphere, this same polygon doesn't intersect itself and is valid:



Conversion and constructor functions handle validation differently

When the input data is invalid, the GEOMETRY and GEOGRAPHY functions handle validation in different ways:

- Some of the functions for constructing and converting to GEOGRAPHY objects might attempt to repair the shape to handle problems such as unclosed loops, spikes, cuts, and self-intersecting loops in polygons. For example, when either the TO_GEOGRAPHY function or the ST_MAKEPOLYGON function is used to construct a polygon, the function

- The functions for constructing and converting to GEOMETRY objects (for example, TO_GEOMETRY) don't support the ability to repair the shape.

Converting between GEOGRAPHY and GEOMETRY

Snowflake supports converting from a GEOGRAPHY object to a GEOMETRY object (and vice versa). Snowflake also supports transformations of objects that use different spatial reference systems (SRS).

The following example converts a GEOGRAPHY object that represents a point to a GEOMETRY object with the SRID 0:

```
SELECT TO_GEOMETRY(TO_GEOGRAPHY('POINT(-122.306100 37.554162)'));
```

To set the SRID of the new GEOMETRY object, pass the SRID as an argument to the constructor function. For example:

```
SELECT TO_GEOMETRY(TO_GEOGRAPHY('POINT(-122.306100 37.554162)', 4326));
```

If you need to set the SRID of an existing GEOMETRY object, see Changing the spatial reference system (SRS) and SRID of a GEOMETRY object.

Specifying how invalid geospatial shapes are handled

1. The function attempts to validate the shape in the input data.
2. The function determines if the shape is valid according to the Open Geospatial Consortium's Simple Feature Access / Common Architecture standard.
3. If the shape is invalid, the function attempts to repair the data (for example, fixing polygons by closing the rings).
4. If the shape is still invalid after the repairs, the function reports an error and doesn't create the GEOGRAPHY or GEOMETRY object. (For the TRY_* functions, the functions return NULL, rather than reporting an error.)

With this feature, you have more control over the validation and repair process. You can:

- Allow these conversion functions to create GEOGRAPHY and GEOMETRY objects for invalid shapes.
- Determine if the shape for a GEOGRAPHY or GEOMETRY object is invalid.

Understanding the effects of invalid shapes on geospatial functions

Different geospatial functions have different effects when you pass in a GEOGRAPHY or GEOMETRY object for an invalid shape.

Effects on GEOMETRY objects

For GEOMETRY objects:

- The following functions return results based on the original invalid shape:
  - ST_AREA

- ST_ASWKT
- ST_CENTROID
- ST_CONTAINS
- ST_DIMENSION
- ST_DISTANCE
- ST_ENVELOPE
- ST_INTERSECTS
- ST_LENGTH
- ST_NPOINTS , ST_NUMPOINTS
- ST_PERIMETER
- ST_SETSRID
- ST_SRID
- ST_X
- ST_XMAX
- ST_XMIN
- ST_Y
- ST_YMAX
- ST_YMIN

- The following functions validate the shape and fail with an error if the shape is invalid:
  - ST_MAKELINE
  - ST_MAKEPOLYGON

Effects on GEOGRAPHY objects

For GEOGRAPHY objects:

- The following functions return results based on the original invalid shape:
  - ST_ASWKB
  - ST_ASWKT
  - ST_ASGEOJSON
  - ST_AZIMUTH
  - ST_COLLECT
  - ST_DIMENSION
  - ST_GEOHASH
  - ST_HAUSDORFFDISTANCE
  - ST_MAKELINE
  - ST_NPOINTS , ST_NUMPOINTS
  - ST_POINTN
  - ST_SRID
  - ST_ENDPOINT
  - ST_STARTPOINT
  - ST_X
  - ST_Y
- The following functions validate the shape and fail with an error if the shape is invalid:

- ST_MAKEPOLYGONORIENTED

- The following functions return NULL if it isn't possible to compute the value:
  - ST_AREA
  - ST_CENTROID
  - ST_CONTAINS
  - ST_COVERS
  - ST_DIFFERENCE
  - ST_DISTANCE
  - ST_DWITHIN
  - ST_ENVELOPE
  - ST_INTERSECTION
  - ST_INTERSECTION_AGG
  - ST_INTERSECTS
  - ST_LENGTH
  - ST_PERIMETER
  - ST_SIMPLIFY
  - ST_SYMDIFFERENCE
  - ST_UNION
  - ST_UNION_AGG
  - ST_XMAX
  - ST_XMIN

Working with invalid shapes

The next sections explain how to allow functions to create invalid shapes and how to determine if a GEOGRAPHY or GEOMETRY object represents an invalid or repaired shape.

Allowing conversion functions to create invalid shapes

To allow the following conversion functions to create invalid geospatial objects, pass `TRUE` for the second argument (`<allowInvalid>`):

```
TO_GEOGRAPHY( <input> [, <allowInvalid> ] )

ST_GEOGFROMWKB( <input> [, <allowInvalid> ] )

ST_GEOGFROMWKT( <input> [, <allowInvalid> ] )

TO_GEOMETRY( <input> [, <allowInvalid> ] )

ST_GEOMFROMWKB( <input> [, <allowInvalid> ] )
```

```
ST_GEOMFROMWKT( <input> [, <allowInvalid> ] )
```

By default, the `<allowInvalid>` argument is `FALSE`.

When you pass `TRUE` for the `<allowInvalid>` argument, the conversion function returns a GEOGRAPHY or GEOMETRY object, even when the input shape is invalid and can't be repaired successfully.

For example, the following input shape is a LineString that consists of the same two Points. Passing `TRUE` for the `<allowInvalid>` argument returns a GEOMETRY object that represents an invalid shape:

```
SELECT TO_GEOMETRY('LINESTRING(100 102,100 102)', TRUE);
```

Determining if a shape is invalid

To determine if a GEOGRAPHY or GEOMETRY object is invalid, call the ST_ISVALID function.

The following example checks if an object is valid:

```
SELECT TO_GEOMETRY('LINESTRING(100 102,100 102)', TRUE) AS g, ST_ISVALID(g);
```